COURSE TEXTBOOK

Operating Systems

Sercan Külcü

Giresun University · Department of Computer Engineering

Operating Systems

Sercan Külcü
Giresun University, Department of Computer Engineering
Giresun, Türkiye

This book is the printed form of the course notes published at
`sercankulcu.github.io/teaching/operating-systems/`

Slides, study questions and simulators are on that page. The code and labs are in the `github.com/sercankulcu/operating-systems-labs-en` repository.

© Sercan Külcü. All rights reserved.

Generated from the notes of 24 November 2026.

Preface

An operating system is the software that sits between the hardware and applications: it manages processor time, memory, storage and devices, and gives programs abstractions through which they can use them safely and share them. This book is made up of the notes for the Operating Systems course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course. The examples are written in C, and in four chapters also in Java.

The book has four parts. In order, they are **Foundations** (Chapters 1–2), **Process Management** (Chapters 3–7), **Memory and Storage** (Chapters 8–11), and **Advanced Topics** (Chapters 12–15).

Every chapter follows the same layout. It opens with learning outcomes. Formal definitions, theorems and worked examples are set in named boxes. *Common mistake* boxes point out the misconceptions students fall into most often. Code traces show, step by step, how a program or an algorithm runs. Each chapter closes with a summary, a list of key terms, *Test Yourself* questions with their answers, suggestions for further reading and a bibliography. The glossary at the end of the book defines each key term in a sentence and names the chapter that explains it.

The code and lab exercises for each chapter are in the companion repository `github.com/sercankulcu/operating-systems-labs-en`, and the Java examples in `github.com/sercankulcu/operating-systems-java-en`; the programs use the same names as the listings in this book. Slides, study questions and simulators are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · FOUNDATIONS

1 Introduction 4

1.1 What is an operating system? 5
1.2 Position and modes 10
1.3 Roles of the operating system 16
1.4 History 29
1.5 Hardware 37
1.6 I/O methods 43
1.7 Kinds of operating systems 45
1.8 Key concepts 48
1.9 Summary 53
1.10 Key terms 54
1.11 Test Yourself 54
1.12 Further reading 57
1.13 References 57

2 Operating-System Structures 58

2.1 Structure and services 59
2.2 Core concepts 62
2.3 Protection 71
2.4 System calls 75
2.5 POSIX and Win32 calls 88
2.6 Kernel structures 96
2.7 Virtual machines and multiprocessor systems 111
2.8 Linking, loading, and booting 114
2.9 Summary 117
2.10 Key terms 118
2.11 Test Yourself 118
2.12 Further reading 121
2.13 References 121

PART II · PROCESS MANAGEMENT

3 Processes 123

3.1 The process concept 124
3.2 Process creation 132
3.3 Process termination 147
3.4 Process states and the PCB 153
3.5 Scheduling infrastructure 159

3.6 Interprocess communication 171
3.7 Multitasking on mobile systems 181
3.8 Summary 182
3.9 Key terms 183
3.10 Test Yourself 184
3.11 Further reading 187
3.12 References 187

4 Threads 188

4.1 Basic concepts 189
4.2 Multicore programming 203
4.3 Thread models 210
4.4 Thread libraries 221
4.5 Implicit threading 231
4.6 Thread issues 241
4.7 Process or thread? 250
4.8 Summary 251
4.9 Key terms 252
4.10 Test Yourself 252
4.11 Further reading 254
4.12 References 255

5 CPU Scheduling 256

5.1 Basic concepts 257
5.2 Basic algorithms 266
5.3 Round robin 282
5.4 Multilevel queues 289
5.5 Proportional share 294
5.6 Multiprocessor scheduling 299
5.7 Real-time scheduling 306
5.8 Operating-system examples 314
5.9 Evaluating algorithms 319
5.10 Summary 323
5.11 Key terms 324
5.12 Test Yourself 324
5.13 Further reading 326
5.14 References 327

6 Synchronization 328

6.1 Race conditions 329
6.2 The critical-section problem 335
6.3 Software approaches 338
6.4 The memory model 345
6.5 Hardware support 355
6.6 Locks 363
6.7 Semaphores 371

6.8 Classic problems 377
6.9 Monitors 388
6.10 POSIX and Java 397
6.11 Liveness 403
6.12 Introduction to lock-free structures 406
6.13 Summary 409
6.14 Key terms 410
6.15 Test Yourself 410
6.16 Further reading 413
6.17 References 413

7 Deadlocks 415

7.1 Deadlock and resources 416
7.2 The four conditions for deadlock 423
7.3 Modeling deadlock 427
7.4 Strategies for coping 436
7.5 Detection and recovery 437
7.6 Deadlock avoidance 451
7.7 Deadlock prevention 467
7.8 Other issues 477
7.9 Deadlock in practice 483
7.10 Summary 489
7.11 Key terms 490
7.12 Test Yourself 490
7.13 Further reading 493
7.14 References 493

PART III · MEMORY AND STORAGE

8 Memory Management 496

8.1 Memory hardware 497
8.2 The address space 506
8.3 Linking and loading 514
8.4 Swapping 519
8.5 Contiguous allocation 521
8.6 Free-space management 529
8.7 Paging 534
8.8 The TLB 544
8.9 Page-table structures 548
8.10 Segmentation 556
8.11 Real architectures and practice 561
8.12 Summary 564
8.13 Key terms 565
8.14 Test Yourself 565
8.15 Further reading 567
8.16 References 568

9 Virtual Memory 569
9.1 Introduction to virtual memory 570
9.2 Demand paging 574
9.3 Performance: effective access time 583
9.4 Copy-on-write 586
9.5 Memory-mapped files 589
9.6 Page replacement 592
9.7 The working set 612
9.8 Comparing the algorithms 617
9.9 Frame allocation 620
9.10 Thrashing 622
9.11 Design issues 626
9.12 Kernel memory 634
9.13 Real systems 639
9.14 Summary 642
9.15 Key terms 643
9.16 Test Yourself 643
9.17 Further reading 645
9.18 References 646

10 File Systems 647
10.1 Why a file system? 648
10.2 The file concept 649
10.3 Directories 661
10.4 Protection 669
10.5 Disk layout 671
10.6 Allocation methods 675
10.7 Implementing directories 687
10.8 Free-space management 689
10.9 Performance 694
10.10 Consistency and recovery 700
10.11 VFS and the network file system 709
10.12 Summary 711
10.13 Key terms 713
10.14 Test Yourself 713
10.15 Further reading 715
10.16 References 715

11 Input/Output 717
11.1 Overview of the I/O system 718
11.2 I/O hardware 722
11.3 Programmed I/O 726
11.4 Interrupts 729
11.5 Direct memory access (DMA) 736
11.6 I/O software 741
11.7 I/O interfaces 750

11.8 Clocks and other devices 754
11.9 Disks 760
11.10 Disk-arm scheduling 765
11.11 SSDs and NVMe 777
11.12 RAID and stable storage 782
11.13 Summary 790
11.14 Key terms 792
11.15 Test Yourself 792
11.16 Further reading 794
11.17 References 795

PART IV · ADVANCED TOPICS

12 Multiprocessor Systems 797

12.1 Why multiple processors? 798
12.2 Multiprocessor hardware 800
12.3 Multiprocessor operating system types 817
12.4 Synchronization 821
12.5 Multiprocessor scheduling 829
12.6 Multicomputers 833
12.7 Virtualization 846
12.8 Distributed systems 852
12.9 Summary 864
12.10 Key terms 865
12.11 Test Yourself 865
12.12 Further reading 867
12.13 References 868

13 Security 869

13.1 Security goals and threats 870
13.2 Cryptography 878
13.3 Protection 890
13.4 Formal security models 903
13.5 Authentication 910
13.6 Software vulnerabilities 918
13.7 Insider attacks 926
13.8 Malware 927
13.9 Defense and isolation 932
13.10 Hardware-level security 937
13.11 Summary 940
13.12 Key terms 941
13.13 Test Yourself 941
13.14 Further reading 943
13.15 References 944

14 Operating-System Design 945

14.1 Design challenges and goals 946
14.2 Interface design 949
14.3 System structure 958
14.4 Design principles 964
14.5 Useful techniques 973
14.6 Performance 980
14.7 Project management 994
14.8 Testing and debugging 996
14.9 Trends in operating systems 1008
14.10 Summary 1011
14.11 Key terms 1012
14.12 Test Yourself 1012
14.13 Further reading 1014
14.14 References 1015

15 Linux 1016

15.1 The roots of Linux 1017
15.2 Interfaces and the shell 1020
15.3 The structure of the kernel 1025
15.4 Processes and threads 1028
15.5 Signals 1033
15.6 Scheduling 1037
15.7 Booting 1041
15.8 Memory management 1042
15.9 Input/output 1052
15.10 File systems 1056
15.11 Security 1067
15.12 Android 1074
15.13 The course's concepts in Linux 1075
15.14 Summary 1076
15.15 Key terms 1077
15.16 Test Yourself 1077
15.17 Further reading 1079
15.18 References 1080

APPENDICES

A Glossary of Terms 1081

B Prerequisites for Operating Systems 1092

C Pioneers of Operating Systems 1098

OPERATING SYSTEMS

Course Map

This page puts every chapter of the course on one map. Each chapter is a node, and an arrow shows which idea from an earlier topic the next one relies on. Read it before the course starts to see where the topics are heading, or during the course to see where you are.

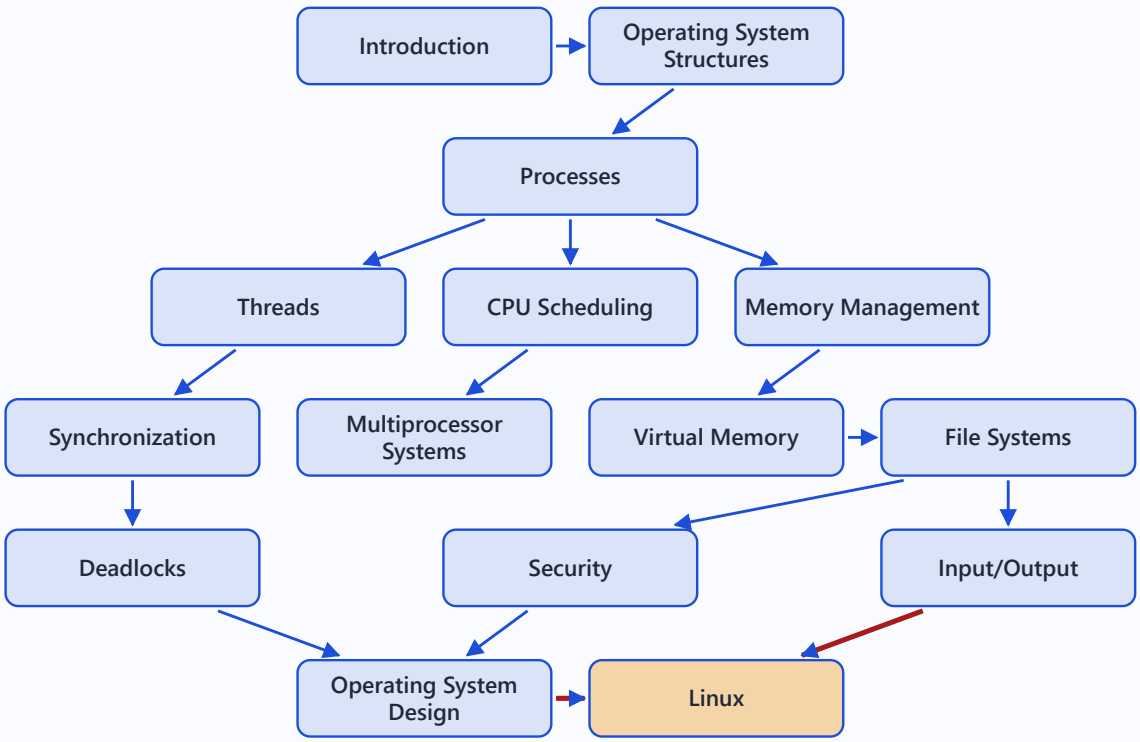


Figure 1: The course map. Each arrow points from a topic to one that builds on it. Every path passes through Processes and all of them meet in Linux.

Figure 1 shows the connections between the topics. Each node is a chapter, and an arrow indicates which idea of the earlier topic the later one relies on. Only the closest connections are drawn on the map; every chapter that a given chapter depends on is listed in Table 1.

Topic (chapter)	Builds on	Connection
Introduction (1)	—	The two roles of an operating system: extended machine and resource manager; kernel mode and user mode.
Operating System Structures (2)	Introduction	Services are provided through system calls: the concepts of process, address space, file and pipe.

Topic (chapter)	Builds on	Connection
Processes (3)	Operating System Structures	The running program: fork, exec and wait; process states and the process control block.
Threads (4)	Processes	Several flows of execution in the same address space; user-level and kernel-level threads.
CPU Scheduling (5)	Processes	Which of the processes in the ready queue gets the processor: FCFS, SJF, Round Robin and multilevel feedback queues.
Synchronization (6)	Threads	Concurrent access to shared data creates a race condition: the critical section, mutex and semaphore.
Deadlocks (7)	Synchronization	Acquiring locks in the wrong order leads to deadlock: the four conditions, the banker's algorithm.
Memory Management (8)	Operating System Structures, Processes	The address space of each process: logical and physical addresses, contiguous allocation, paging.
Virtual Memory (9)	Memory Management	The demand-driven form of paging; page faults and page replacement algorithms.
File Systems (10)	Operating System Structures, Virtual Memory	The on-disk counterpart of the file concept; memory-mapped files and the page cache make use of virtual memory.
Input/Output (11)	Processes, File Systems	The layer beneath the file system: device drivers, interrupts and DMA; a blocked process moves to a wait queue.
Multiprocessor Systems (12)	CPU Scheduling, Synchronization	Scheduling and synchronization on several processors: cache coherence, the scaling of spinlocks.
Security (13)	Operating System Structures, File Systems	Protection from kernel and user mode to file permissions; access control lists and capabilities.
Operating System Design (14)	Deadlocks, Multiprocessor Systems, Security	The common principles behind the choices in the earlier chapters: separating mechanism from policy, trade-offs.
Linux (15)	Input/Output, Multiprocessor Systems, Operating System Design	All the concepts in a real system: task_struct, the EEVDF scheduler, the virtual address space, the VFS.

Table 1: The connections on the map.

The backbone of the map can be read as **process → sharing → memory → storage → a real system**. The process concept is the center of the course, and three branches grow out of it. The processor branch continues with scheduling, the concurrency branch with threads, synchronization and deadlocks, and the memory branch with paging and virtual memory. Virtual memory leads to the file system, and the file system to the I/O layer. Multiprocessor systems raise scheduling and synchronization together again. The design chapter gathers the common principles behind these choices, and the Linux chapter shows all of them in a working kernel.

PART I

Foundations

Chapters 1–2

PART I · CHAPTER 1

Introduction

Every program on a computer relies on a layer of software beneath it: the operating system. A browser writes a file to disk without knowing the disk's brand. A game draws to the screen without touching the graphics card's registers. A compiler runs without being able to corrupt the memory of other programs. The operating system provides all of this convenience and protection. These notes begin with the place of the operating system between hardware and applications and with its two basic roles: the extended machine, which presents the hardware through useful abstractions, and the resource manager, which shares resources among programs. They then cover kernel mode and user mode, the controlled transition between the two, the historical development of operating systems, and the hardware features that drove that development. The last sections deal with I/O methods, the kinds of operating systems, the separation of policy from mechanism, and how a C program becomes a running process.

LEARNING OUTCOMES

Definition and place
Explain where the operating system sits between hardware and applications and describe its two basic roles (extended machine, resource manager).

Protection
Explain kernel mode and user mode and the controlled transitions between them (system call, interrupt, exception).

Historical development
Describe which problems the ideas of batch processing, multiprogramming, and time sharing solved; draw the timeline of two jobs and compute CPU utilization.

Hardware foundations
Interpret how the processor and its registers, the memory hierarchy, and multicore chips affect the operating system.

I/O methods
State the basic difference between polling, interrupts, and DMA, compute the CPU load, and distinguish an interrupt from a trap.

Concepts
Illustrate the kinds of operating systems, the policy–mechanism separation, and the transformation of a program into an executable file with examples.

1.1 What is an operating system?

In everyday speech, “operating system” is often confused with the desktop on the screen: the Start menu, the file explorer, the wallpaper, and the icons. These are only the visible face of the operating system, and most of them are ordinary programs. The real work happens out of sight, in the software layer that decides which program runs when, who uses which part of memory, and in what order requests to the disk are served. This section explains what that layer is and why it exists.

1.1.1 Why do we need an intermediate layer?

Consider the inside of an ordinary laptop today. It has a multicore processor, gigabytes of memory, an SSD, wireless and wired network cards, a graphics processing unit (GPU), a keyboard, a touchpad, a camera, a microphone, speakers, and a few USB ports. Each of these devices comes from a different manufacturer, and each has its own way of being controlled. To write data to an SSD, command queues are prepared, block addresses are computed, and error codes are checked. To receive a packet from a network card, ring buffers are set up and interrupts are configured.

It is unreasonable to expect every application developer to learn all of these details and write them directly into each program. If that were the case, a text editor would have to contain separate code for every disk model on the market. Worse, if two programs running at the same time tried to write to the same disk in their own way, the data on the disk would be mixed up. So there are two separate needs: a *common interface* that hides the complexity of the hardware, and *order and protection* among the programs that use the same hardware. The operating system is the answer to these two needs.

Definition 1.1

An **operating system** (OS) is the layer of software that manages a computer's hardware and offers applications a simple, safe, and common interface.

Each of the three adjectives in the definition corresponds to a separate job. A *simple* interface means that an application only has to say “write these bytes to that file”; the internal structure of the disk is the OS's problem. A *safe* interface means that the bug or malice of one program cannot harm other programs or the system as a whole. A *common* interface means that the same program can run unchanged on different hardware: a C program writes to a file with the same `write` call whichever SSD is installed.

Today very different operating systems run in very different environments. On the desktop, Windows 11, macOS, Linux distributions such as Ubuntu or Fedora, and ChromeOS are common. Phones and tablets use Android, iOS/iPadOS, and HarmonyOS. On servers and in the cloud, Linux is far ahead, with Windows Server and FreeBSD also present. Inside cars, household appliances, and Internet of Things (IoT) devices run small, special-purpose systems such as Zephyr, FreeRTOS, Contiki-NG, VxWorks, and QNX. These systems differ greatly in size and goals, but all of them solve the same two basic problems.

1.1.2 The kernel and the rest of the software

When an operating system is installed, thousands of files are written to disk: drivers, libraries, utilities, fonts, configuration tools, the desktop environment. Calling all of them “the operating system” would make the term almost meaningless. For this reason, textbooks distinguish a narrow and a broad meaning.

> **Definition 1.2**
>
> The **kernel** is the part of the operating system that stays in memory as long as the computer is on and runs in the processor's privileged mode. In the narrow sense, “operating system” means the kernel.

The kernel creates and schedules processes, allocates memory, hosts the file systems and device drivers, and handles interrupts. Most of the rest of the software runs on top of the kernel like ordinary programs. The two interfaces through which the user talks to the computer are both in this second group: the **shell**, that is, the command line, and the **GUI** (graphical user interface). Both are discussed in detail below; what matters here is that neither is part of the kernel.

1.1.3 Components of a computer system

Thinking of a computer system in layers is the easiest way to see who does what. Figure 1.1 shows these layers from bottom to top.

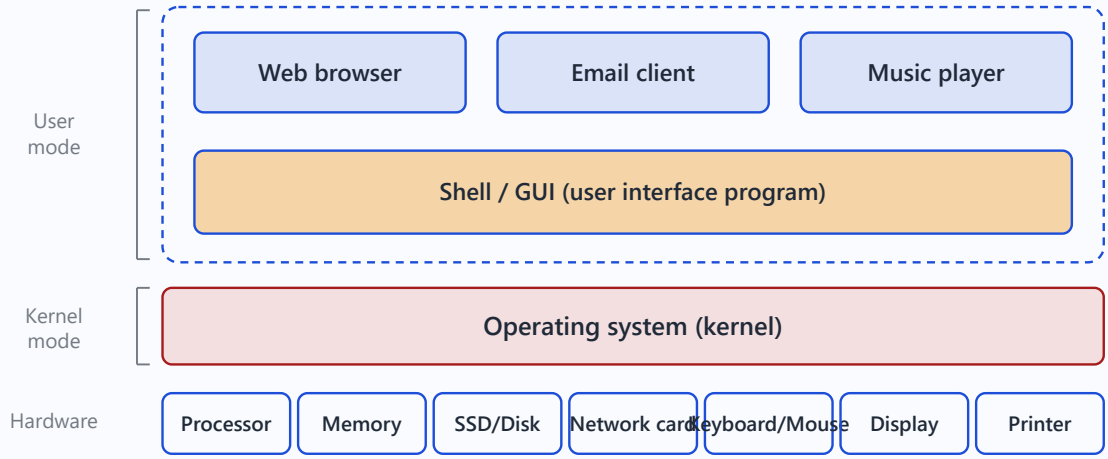


Figure 1.1: The layers of a computer system. At the bottom is the hardware (processor, memory, SSD/disk, network card, keyboard/mouse, display, printer). Above it sits the operating system, running in kernel mode. At the top, inside the dashed frame, are the programs that run in user mode: a web browser, an email client, a music player, and, at the same level, the shell or GUI.

The lowest layer, the **hardware**, contains the processor, the memory, the storage devices, and the input/output (I/O) devices. These are physical components; on their own they understand only very simple commands. The **operating system** just above it is the only software that controls the hardware directly, and it runs in the processor's privileged mode. The top layer holds everything else: the applications the user runs and, at the same level, the shell and the GUI.

The separation in the figure divides the software world in two. The software that runs in kernel mode is the operating system. Everything that runs in user mode, even the shell and the GUI, is an ordinary user program. Why this separation matters so much is explained in 1.2.2, which covers the modes.

Common mistake

Taking the desktop, the windows, or the command line to be the operating system itself. A Linux server has no GUI at all, yet its operating system runs in full. On top of the same Linux kernel there can be GNOME, KDE Plasma, or just a terminal. The visible interface changes; the kernel stays the same.

1.1.4 The shell and the GUI

When users want to get something done, they say so either by typing a command or by clicking an icon with the mouse. A separate program serves each of these two routes.

Definition 1.3

The **shell** is the command-line interface (CLI) that reads the commands a user types, interprets them, and starts the corresponding programs. A **GUI**, in contrast, is a visual interaction environment made of windows, icons, menus, and a pointer (WIMP).

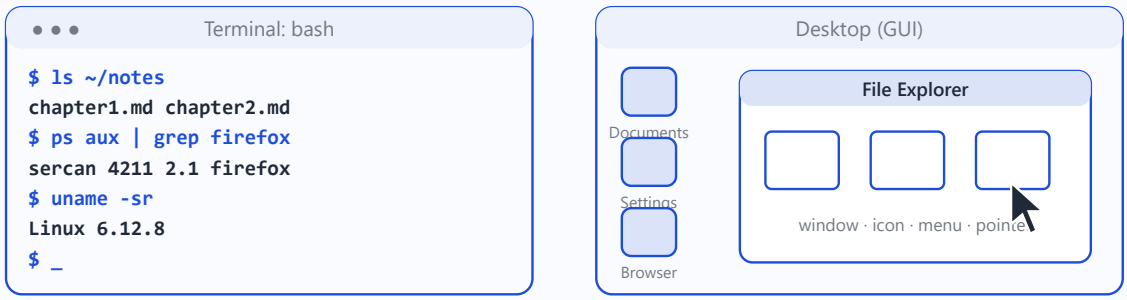


Figure 1.2: On the left, the bash shell in a terminal window: `ls ~/notes` lists two files, `ps aux | grep firefox` finds the running Firefox process, and `uname -sr` prints the kernel version (Linux 6.12.8). On the right, a desktop (GUI) made of windows, icons, menus, and a pointer.

Common shells include bash, zsh, and fish on Linux and macOS, and PowerShell and the older cmd.exe on Windows. On the GUI side, Windows Explorer, macOS Finder, and the GNOME and KDE Plasma environments on Linux are well known. Figure 1.2 shows that two interfaces can run side by side on the same computer.

Neither the shell nor the GUI is part of the kernel. Both are programs running in user mode, and when they want a service from the kernel, for example to open a file or start a new program, they make a *system call*. This has practical consequences. The shell can be replaced: switching from bash to zsh does not require touching the kernel. Several shells can be open at once. If a shell crashes, only that window closes and the system keeps running. Users can customize their interfaces because what they change is only a program.

1.1.5 Shell or GUI?

There is no contest between the two interfaces; each excels at different jobs. The greatest strength of the shell is **automation**. Once commands are written to a script file, the same job can be repeated thousands of times without a single click. Its second strength is that small tools can be combined. The output of one command is connected to the input of the next with a **pipeline**:

```
$ ps aux | grep ssh | wc -l
```

In this line, `ps aux` lists all running processes. `grep ssh` selects the lines of that list that contain "ssh". `wc -l` counts the remaining lines. The combination of three simple tools answers the question "how many process lines contain ssh?" in one line. The shell uses few resources and is the natural way to manage a remote server over an SSH connection. It does specific tasks quickly and precisely.

The strength of the GUI is **discoverability**. Users do not have to memorize commands; they see what can be done by looking at the menus. It is the natural environment for working with visual content such as photos, maps, or designs. It reaches a wide audience and works together with accessibility tools such as screen readers. Touch and voice interfaces belong to the same family.

In practice the two complement each other: the GUI for everyday tasks, the shell for repetitive and remote ones. Both use the same kernel services. From the kernel's point of view, it makes no difference whether a file was copied with the shell's `cp` command or by drag and drop in the file explorer. In both cases, what reaches the kernel is system calls of the same kind.

1.1.6 Device Manager

A concrete way to see how the operating system views the hardware is the **Device Manager** tool in Windows. It lists the hardware devices attached to the computer, and the software that drives them, in a tree organized by category. It is opened by typing `devmgmt.msc` on the command line and is a snap-in written for the Microsoft Management Console (MMC).

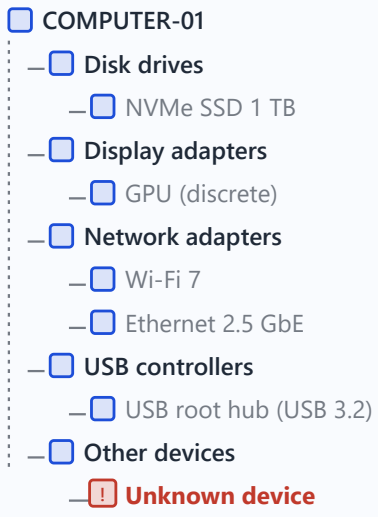


Figure 1.3: The tree view of Device Manager. Under COMPUTER-01 are listed disk drives (NVMe SSD 1 TB), display adapters (discrete GPU), network adapters (Wi-Fi 7, Ethernet 2.5 GbE), and USB controllers. The unknown device under "Other devices" is marked with a warning icon because no driver was found for it.

Device Manager shows each device's type, manufacturer, driver version, and the resources it uses (which interrupt line, which memory range). From here a driver can be updated or rolled back to a previous version, and a device can be disabled. Problem devices are marked with a warning icon, as

in Figure 1.3.

Note that the tool itself does not do the work. What detects devices, assigns them resources, and loads the appropriate driver is the Plug and Play (PnP) manager in the kernel, together with the device drivers. Device Manager is only a window onto the information they hold. On Linux the same information is read with the `lspci` (PCI devices), `lsusb` (USB devices), `lsblk` (block devices), and `lsmod` (loaded kernel modules) commands. The files under `/sys` and the `udev` service, which manages device events, expose the same information as well.

1.2 Position and modes

It is easy to say that the operating system stands “between the hardware and the applications”. The real question is how that position is protected. Why can't an application simply write to the disk itself? Why can't a malicious program bypass the operating system? The answer lies not in software but in the processor itself. This section first describes the position of the operating system and then the hardware feature that secures it: the processor modes.

1.2.1 Where the operating system sits

Hardware is the name for the physical components: the CPU, the RAM, and the disk. Software is the code that describes what the computer will do. The operating system is the interface between the two. Applications do not reach the hardware directly; they reach it through the **system calls** that the OS offers.

> **Definition 1.4**
>
> A **system call** is the controlled entry point a user program uses to request a service from the kernel. Opening, reading, and writing files, creating new processes, and requesting memory, in short everything that needs privilege, is done through system calls.

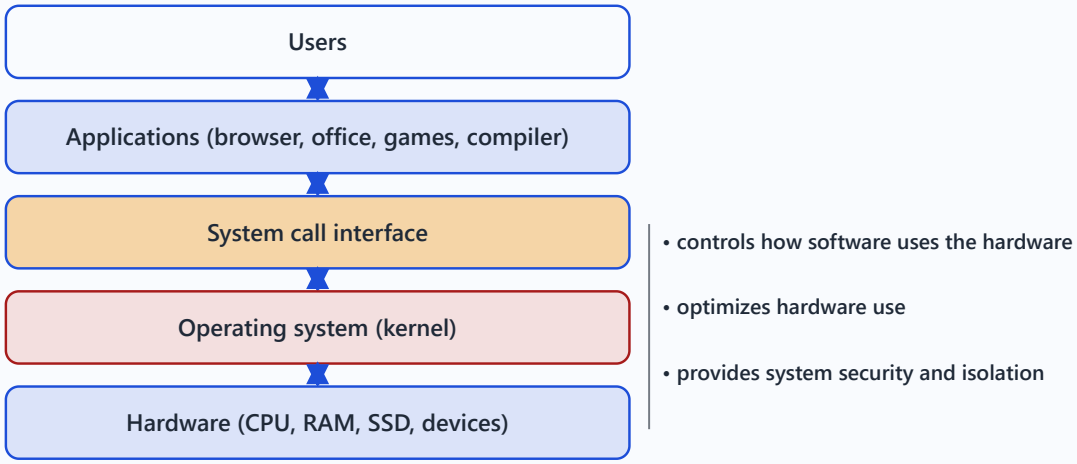


Figure 1.4: The position of the operating system. Users run applications (browser, office suite, games, compiler); applications reach the operating system through the system call interface; the operating system controls the hardware (CPU, RAM, SSD, devices). The notes on the right list the three duties of the OS: it controls how software uses the hardware, it optimizes hardware use, and it provides system security and isolation.

Figure 1.4 shows five layers stacked on top of each other. The layer that stands out is the thin "system call interface" between the applications and the operating system. This interface is the only door between two worlds. Everything an application can ask of the OS is written on this door's list; an application cannot get done anything that is not on the list. On Linux the list holds a few hundred calls, and each has a number.

The three items to the right of the figure summarize what the operating system does from this position. *Control*: the OS decides which program accesses which hardware, when, and how much. *Optimization*: it keeps the hardware from sitting idle and makes sure jobs finish in reasonable time. *Security and isolation*: it prevents programs from harming each other and the system.

1.2.2 Kernel mode and user mode

It is impossible to stop a program from accessing the hardware directly merely by telling it that it should not. After all, the program runs on the processor, and the processor executes every instruction it is given. The barrier is built by a hardware feature placed inside the processor.

Definition 1.5

The processor has a **mode bit** (or a privilege level of a few bits) that shows the current privilege level. In **kernel mode** all instructions and all of memory can be used. In **user mode** privileged instructions are forbidden; when one is attempted, the processor does not execute it, raises an **exception**, and hands control to the kernel.

Privileged instructions are those that can affect the system as a whole. Accessing I/O devices directly, disabling interrupts, and changing the page table (that is, which virtual address maps to which physical memory) belong to this group. When one of them is attempted in user mode, the processor raises a **trap**: it abandons the current instruction, switches to kernel mode, and jumps to a handler the kernel set up in advance. The kernel usually terminates the program. The rest of the system is unaffected.

Different processor families give these levels different names. The x86 architecture has four **rings**: ring 0 is the most privileged and ring 3 the most restricted. In ARMv8 the exception levels run from EL0 to EL3; applications run at EL0 and the kernel at EL1. RISC-V has U (user), S (supervisor), and M (machine) modes.

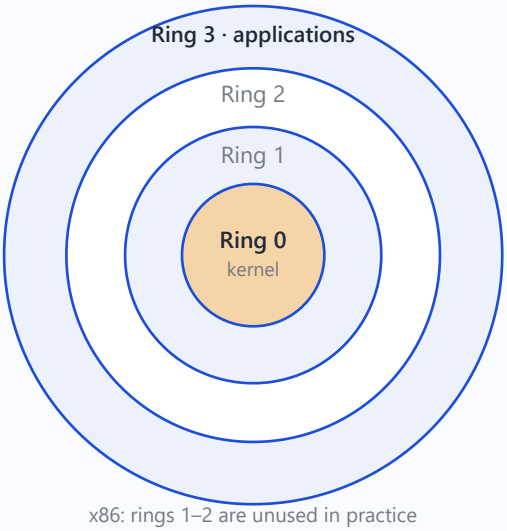


Figure 1.5: The x86 privilege rings. The kernel runs in ring 0 at the center, and applications run in ring 3 on the outside. Rings 1 and 2 are not used in practice; today's operating systems make do with just two levels.

Figure 1.5 shows that operating systems use only two of the four levels. The reason is portability: Linux and Windows run on many processor families, and the common denominator is two levels. The shell and the GUI sit on the lowest rung of user-mode software; even they have no access to ring 0.

There are only three controlled ways to move from user mode to kernel mode:

- **System call**: the program deliberately requests a service from the kernel with a special instruction (`syscall` on x86-64).
- **Interrupt**: a device, such as the timer or the network card, sends a signal to the processor from outside.
- **Exception**: the executing instruction itself causes a problem, such as division by zero, access to an invalid address, or a privileged instruction in user mode.

What the three paths have in common is that the program cannot choose *where* to jump. The target addresses are written in a table that the kernel fills in at boot. The program can only say "I want to go to the kernel"; it cannot decide at which point of the kernel it enters. The protection rests on exactly this.

Common mistake

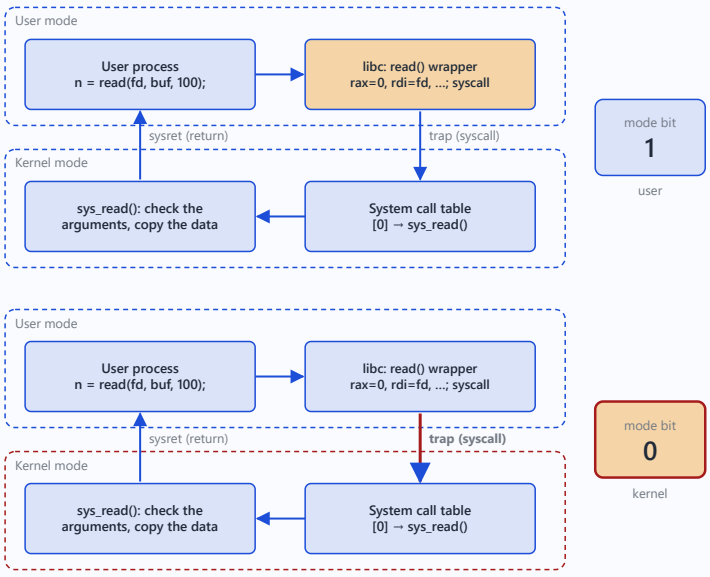
Confusing kernel mode with "running as the administrator (root or Administrator) account". A program started with the administrator account also runs in user mode; the kernel merely allows it more. For example, it can open any file. But everything such a program does is still done through system calls, and the program itself never runs in ring 0.

1.2.3 A mode switch step by step: read()

To make the mode switch concrete, let us follow the single line in which a C program reads from a file:

```
char buf[100];
int n = read(fd, buf, 100);   /* n: number of bytes read */
```

To the programmer this is an ordinary function call. In reality, this line starts a five-step journey from user mode into kernel mode and back. The values below are for Linux on the x86-64 architecture.



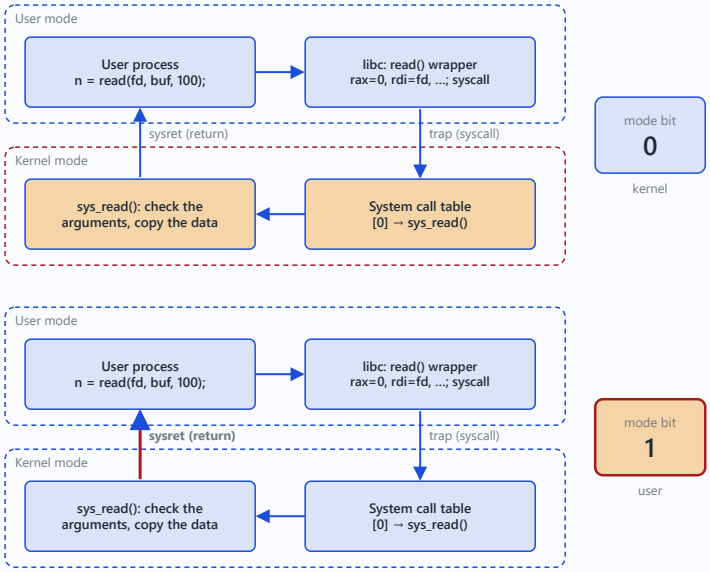


Figure 1.6: The mode switch in the `read()` system call (left to right, top to bottom). (a) The libc wrapper puts the call number and the arguments into registers; the mode bit is 1, that is, user mode. (b) The `syscall` instruction raises a trap; the mode bit becomes 0. (c) The kernel finds `sys_read()` in entry 0 of the system call table, checks the arguments, and copies the data. (d) `sysret` sets the mode bit back to 1 and the process continues where it left off.

1. **Start.** The process is running in user mode (mode bit = 1). It wants to read from a file, but it cannot reach the disk itself; instructions that touch the disk controller's registers are privileged.
2. **The wrapper.** The `read()` that the program calls is really a small *wrapper* function in the C library (libc). It puts the call number into the `rax` register (on x86-64 Linux the number of `read` is 0) and places the arguments in the `rdi`, `rsi`, and `rdx` registers in order: `fd`, `buf`, 100. Then it executes the `syscall` instruction.
3. **The trap.** With the `syscall` instruction the hardware does three things: it sets the mode bit to 0, saves the return address, and jumps to the kernel's fixed entry point. The process cannot choose where to jump; the kernel wrote the entry point into a register at boot.
4. **Work in the kernel.** The kernel uses the number in `rax` to look into the *system call table* and finds `sys_read()` at entry 0. This function first validates the arguments: is `fd` an open file of this process, and does the address `buf` really lie in the process's own memory? If the checks pass, it copies the data from the file into the process's buffer.
5. **The return.** The result, the number of bytes read, is written to `rax`. The `sysret` instruction sets the mode bit back to 1, and the process continues from the instruction after `read()`. As far as the program can tell, the variable `n` has received the number of bytes read.

The validation in the fourth step is the heart of protection. A malicious program could pass the address of the kernel's own memory as `buf`. If the kernel used this address without checking, it would write the file's data over its own structures and become open to takeover. For this reason the kernel trusts no value that comes from the user.

Example 1.1

Which of the following operations cause a transition from user mode to kernel mode? (a) `int x = a + b;` (b) `read(fd, buf, 100);` (c) an instruction accessing an invalid address (d) `strlen(s);` (e) the timer raising an interrupt.

Solution. (a) Addition is an ordinary operation on registers and the ALU; the mode does not change. (b) This is a system call; the libc wrapper deliberately raises a trap with the `syscall` instruction, so a transition occurs. (c) This is an exception caused by the instruction itself; the processor jumps to the kernel's handler, so a transition occurs. (d) `strlen` is a library function that counts characters in memory up to the zero byte; it stays entirely in user mode. (e) This is a hardware interrupt coming from outside; a transition occurs. Result: (b), (c), and (e) go to the kernel; they are, respectively, a system call, an exception, and an interrupt.

Common mistake

Thinking that every library function goes to the kernel. Functions such as `strlen`, `memcpy`, and `sqrt` live in libc but work only with user memory and registers. Only system call wrappers such as `read`, `write`, `open`, and `fork` go to the kernel. `printf` is a mixture of the two: it does the formatting in user mode and finally sends the output with the `write` system call.

1.2.4 Physical devices, microarchitecture, machine language

There are layers below the operating system as well. Separating them helps to delimit the OS's job.

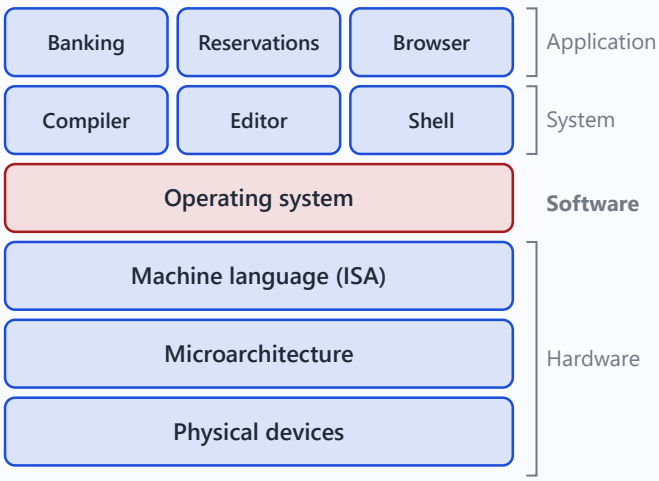


Figure 1.7: The six-layer view. On the hardware side, from bottom to top: physical devices, microarchitecture, and machine language (ISA); on the software side: the operating system, system programs (compiler, editor, shell), and application programs (banking, reservations, browser).

- **Physical devices**: chips, memory modules, SSDs, and I/O devices; the electronic components themselves.
- **Microarchitecture**: the internal design of the processor. The data path, registers, arithmetic logic unit (ALU), caches, and control unit are at this level.
- **Machine language**: the binary instructions that the hardware executes directly. The instruction set of a processor family is called its **instruction set architecture** (ISA). x86-64, ARM64, and RISC-V are today's main ISAs.

The operating system's job in these layers is to translate, through drivers, the requests of the upper layer into low-level commands the device understands. There is one boundary to keep in mind: it is the *compiler* that translates a program into machine language, not the operating system. The compiler is an ordinary user program that sits among the system programs, as in Figure 1.7. The operating system loads the compiled program into memory, runs it, and mediates its access to the hardware.

1.3 Roles of the operating system

The operating system can be viewed from two different angles. From above, that is, from the application's point of view, the OS is a layer that hides the ugly details of the hardware and puts useful concepts in their place. From below, that is, from the hardware's point of view, it is a manager that arbitrates among many programs that all want the same processor, the same memory, and the same devices. Textbooks call these two views the *extended machine* and the *resource manager*.

1.3.1 The extended machine

The interface that bare hardware offers is both complicated and fragile for the programmer. To write a single block to an SSD, one first prepares NVMe command queues and computes the logical block address (LBA). The command is placed in the queue and the controller is notified. Then the completion queue is watched and any error codes are interpreted. The application needs to know none of this; it only says:

```
write(fd, "Welcome", 7);
```

It is the operating system that bridges the gap. Tanenbaum describes this role in a concise sentence: the operating system turns *ugly hardware into beautiful abstractions*.

Definition 1.6

An **abstraction** is a simplified model that hides the unnecessary details of something and shows only its useful properties. Because the operating system builds abstractions on top of the hardware, it is viewed as an **extended machine**: it presents the programmer with a more useful, "extended" computer than the bare hardware.

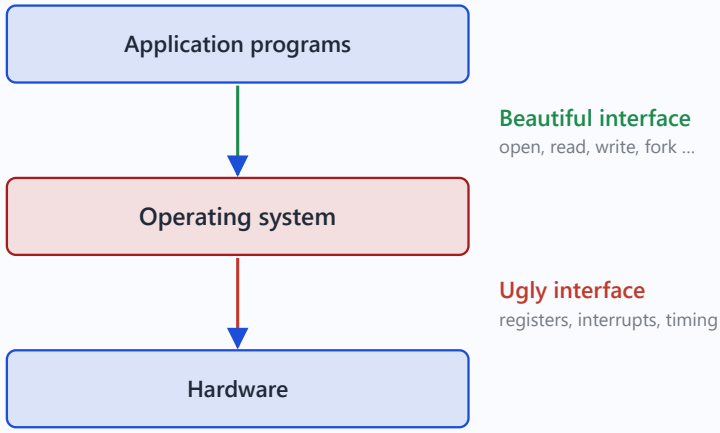


Figure 1.8: The extended machine. Application programs see the beautiful interface that the operating system provides (`open` , `read` , `write` , `fork` ...). The operating system deals with the ugly interface of the hardware (registers, interrupts, timing).

The extended machine role has three consequences. The first is *convenience*: the capabilities of the hardware become usable and their complicated details are hidden. The second is *protection*: because applications cannot reach the hardware directly, they cannot interfere with each other's work. The third is *portability*: even if the SSD changes, or storage over a network is used instead of a disk, the `write` call stays the same.

1.3.2 Abstractions and roles

Each abstraction that the operating system builds corresponds to a particular piece of hardware. Table 1.1 gives the main correspondences.

Hardware	OS abstraction
Processor (CPU)	Process and thread
Memory (RAM)	Address space (virtual memory)
Storage (SSD/disk)	File and directory
Network card	Socket
Display, keyboard	Window, terminal

Table 1.1: Hardware components and the abstractions the operating system builds for them.

Each abstraction in the table makes a *limited and shared* resource of the hardware look as if it belonged to each program alone. There is a single processor, yet each process is written as if it were the only one using it. There is a single physical memory, yet each process sees its own address space starting at address 0. The disk holds millions of blocks, yet the program knows only a file name. The network card is one, yet each connection has its own socket.

Four distinct roles can be told apart around these abstractions. The **hardware designer** designs the ISA, the devices, and the devices' interfaces. The **kernel developer** implements the abstractions on top of this hardware; the process, virtual memory, and the file system are the developer's products. The **application developer** uses the abstractions through system calls and application programming interfaces (APIs). The **end user** sees only the applications and the interface. Most of this course is devoted to understanding the work of the second role.

1.3.3 The journey of a write() call

To see what an abstraction hides, let us follow the call `write(fd, "Welcome", 7)` from the application down to the SSD. This call asks to write the 7-byte string “Welcome” to the file the program has opened.

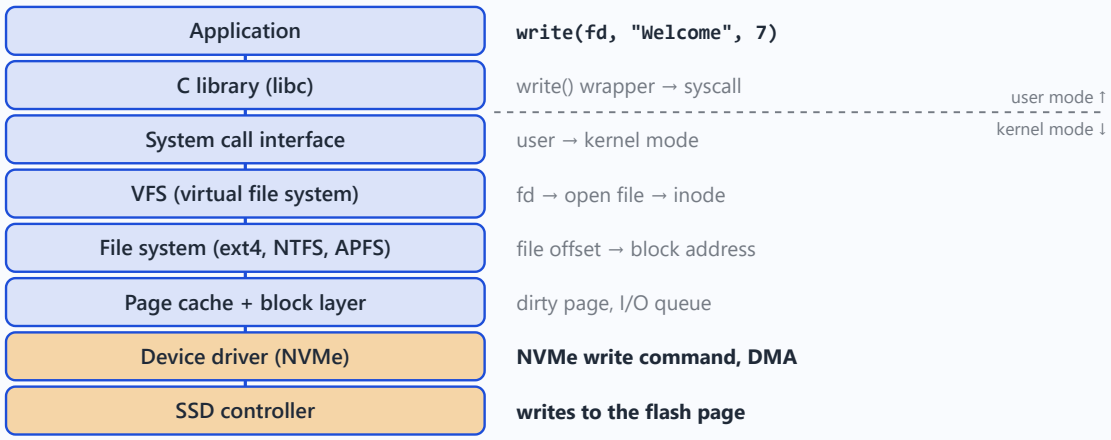


Figure 1.9: The layers of a `write()` call. In user mode are the application and the C library; in kernel mode are the system call interface, the virtual file system (VFS), the file system (ext4, NTFS, APFS), the page cache and block layer, and the device driver (NVMe); at the very bottom is the SSD controller. In the last step the driver and the SSD controller are highlighted.

1. The **application** supplies only three things: a file descriptor (`fd`), a buffer, and a length. It does not know the type of the disk, nor even whether the data is going to a local disk or over the network.
2. The **libc wrapper** puts the arguments into registers and moves to the kernel with `syscall` , just like `read()` in 1.2.3.
3. The **virtual file system** (VFS) resolves `fd` through the process's open file table to a file object, and from there to the file's inode. The VFS is the layer common to all file systems.

4. The **file system** (ext4, NTFS, APFS …) translates the position in the file, that is, the file offset, into a block address on the disk. If the file is growing, it allocates a new block.
5. The **page cache**: the data is usually written first to the page cache in memory, and `write()` returns immediately. The page is marked "dirty"; the write to disk happens later. If the data must become durable at once, the program calls `fsync()`.
6. The **device driver** prepares an NVMe write command. The SSD controller fetches the data from memory by direct memory access (DMA), writes it to a flash page, and raises an interrupt when the job is done.

Six layers have worked for a seven-byte write. Only the first of them is the application's own code. The detail in the fifth step matters too: when `write()` returns, the data is most likely not yet on the disk. If the power fails at that moment, the most recent writes may be lost. That is why programs that need reliability, such as databases, use `fsync()` deliberately.

1.3.4 The resource manager

Seen from below, the operating system's task looks different. Many programs want to run at the same time; all of them need the processor, memory, disk, and network. The operating system divides these resources among the programs and protects each program's share against the others.

> **Definition 1.7**
>
> As a **resource manager**, the operating system allocates and protects resources such as the CPU, memory, and I/O devices among multiple programs. Its goal is to use the resources *efficiently* and to share them *fairly* among programs.

Resources are shared in two ways. In **time multiplexing**, programs use the resource in turn: first one, then another. In **space multiplexing**, the resource is divided into parts and each program uses its own part at the same time.

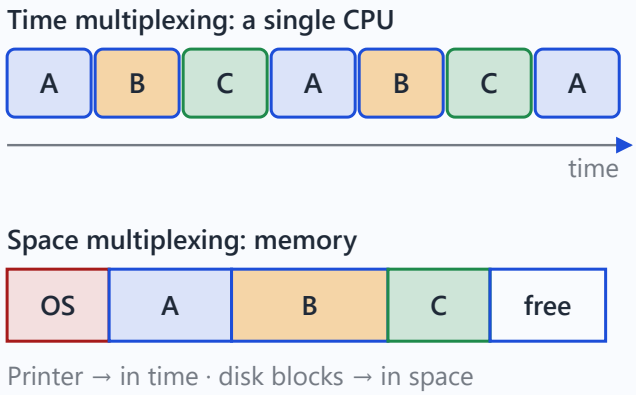


Figure 1.10: Top: time multiplexing; processes A, B, and C use the single CPU in turn. Bottom: space multiplexing; memory is divided into the OS, A, B, C, and a free region. The printer is shared in time, disk blocks in space.

Which resource is shared in which way depends on whether the resource can be divided. The processor executes only one instruction stream at a time; it cannot be divided and is shared in time. Memory can be divided: several programs reside in memory at once and each uses its own region. Disk space is divided too, with separate blocks given to each file; but because read and write requests to the disk are served one after another, the disk is also shared in time. A printer cannot be divided; there is no sense in giving half a page to one program and half to another.

The resource manager view raises the basic questions of operating system design. What happens if many programs want to print at the same time? How is the amount of resources each process uses measured? As resources multiply, for example as the number of cores grows, how are fairness and efficiency achieved? The following subsections take up these questions in turn.

1.3.5 Printing at the same time: spooling

Imagine three people in an office sending documents to print almost at the same time. One sends a report from a word processor, one a photo from a PDF viewer, and one a note from a browser. There is only one printer. If all three programs could write to the printer directly, the pages would be mixed up line by line: under the first line of the report, a piece of the photo would be printed, and below that the first sentence of the note.

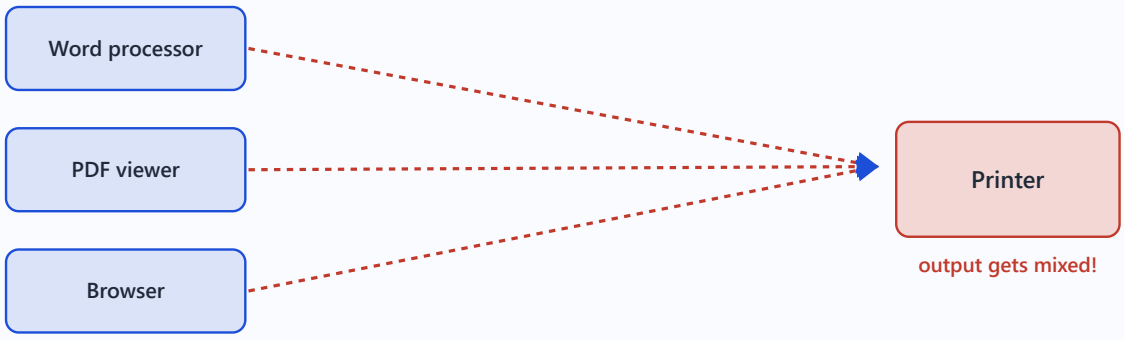


Figure 1.11: Without spooling: the word processor, the PDF viewer, and the browser try to write to the single printer at the same time, and the outputs get mixed.

The operating system's solution is to give the printer to a single process alone.

Definition 1.8

Spooling means that jobs headed for a slow, unshareable device are first written to a queue on disk, and a single service process then sends these jobs to the device one at a time.

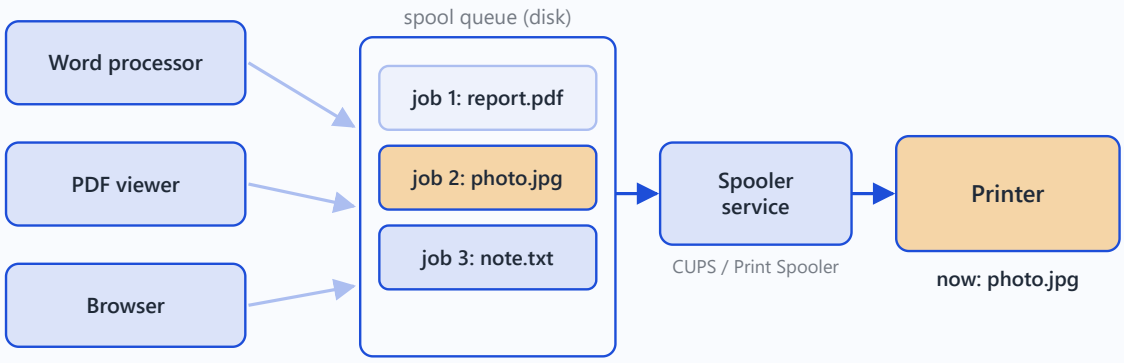


Figure 1.12: Printing with spooling. Three programs write their jobs to the spool queue on disk (job 1: report.pdf, job 2: photo.jpg, job 3: note.txt). Only the spooler service (CUPS on Linux, Print Spooler on Windows) uses the printer. Job 1 has finished, and job 2 (photo.jpg) is printing right now.

Figure 1.12 shows how it works. Programs write their jobs not to the printer but to the operating system's spool queue on disk, and return to their own work at once. The only process that uses the printer is the spooler service: CUPS on Linux, Print Spooler on Windows. This monopoly prevents the mix-up, because only one job goes to the printer at a time. The service usually sends jobs in first-come, first-served (FCFS) order. An administrator can give a job priority or pause a job.

The second gain of spooling is time. A printer is extremely slow compared with the processor; printing a page takes seconds. If programs had to wait for the printer to finish, the user would stare at a frozen window for minutes at every print. Thanks to spooling, while the slow printer works, programs and the processor do other things. I/O and computation advance at the same time; that is, they *overlap*. This idea comes up again and again in the history of operating systems.

1.3.6 Code trace: the spool queue

The essence of spooling can be modeled with a little C code. The queue is an array of 4 slots; programs call `enqueue()` and the spooler service calls the `spooler()` function.

```
char *queue[4];
int in = 0, out = 0, count = 0;
void enqueue(char *file) {   /* programs */
  queue[in] = file;
  in = (in + 1) % 4;
  count++;
}
void spooler(void) {
  while (count > 0) {
    send_to_printer(queue[out]);
    queue[out] = NULL;   /* slot freed */
    out = (out + 1) % 4;
    count--;
  }
}
```

The code reads line by line as follows. The first line defines an array of 4 pointers that will hold the file names. The three counters in the second line keep the state of the queue: `in` is the index of the next free slot, `out` is the index of the oldest job to be printed, and `count` is the number of jobs in the queue. The `enqueue()` function writes the file name into slot `in`, advances `in` by one, and increments `count`. The `% 4` operation used in advancing makes the queue **circular**: after slot 3 it returns to slot 0. As long as jobs remain in the queue, `spooler()` sends the oldest job to the printer, writes `NULL` into the slot it sent to free it, advances `out` in the same way, and decrements `count`.

Suppose four programs submit a.pdf, b.txt, c.doc, and d.png in turn and then the spooler runs. Table 1.2 gives the state of the variables and the queue after each call. In the queue column, "·" marks an empty slot.

Step	Operation	in	out	count	Queue [0] [1] [2] [3]	Printed
0	start	0	0	0	· · · ·	—
1	`enqueue("a.pdf")`	1	0	1	a.pdf · · ·	—
2	`enqueue("b.txt")`	2	0	2	a.pdf b.txt · ·	—
3	`enqueue("c.doc")`	3	0	3	a.pdf b.txt c.doc ·	—

Step	Operation	`in`	`out`	`count`	Queue [0] [1] [2] [3]	Printed
4	`enqueue("d.png")`	0	0	4	a.pdf b.txt c.doc d.png	—
5	send (round 1)	0	1	3	· b.txt c.doc d.png	a.pdf
6	send (round 2)	0	2	2	· · c.doc d.png	a.pdf, b.txt
7	send (round 3)	0	3	1	· · · d.png	a.pdf, b.txt, c.doc
8	send (round 4)	0	0	0	· · · ·	a.pdf, b.txt, c.doc, d.png

Table 1.2: Trace of the spool queue.

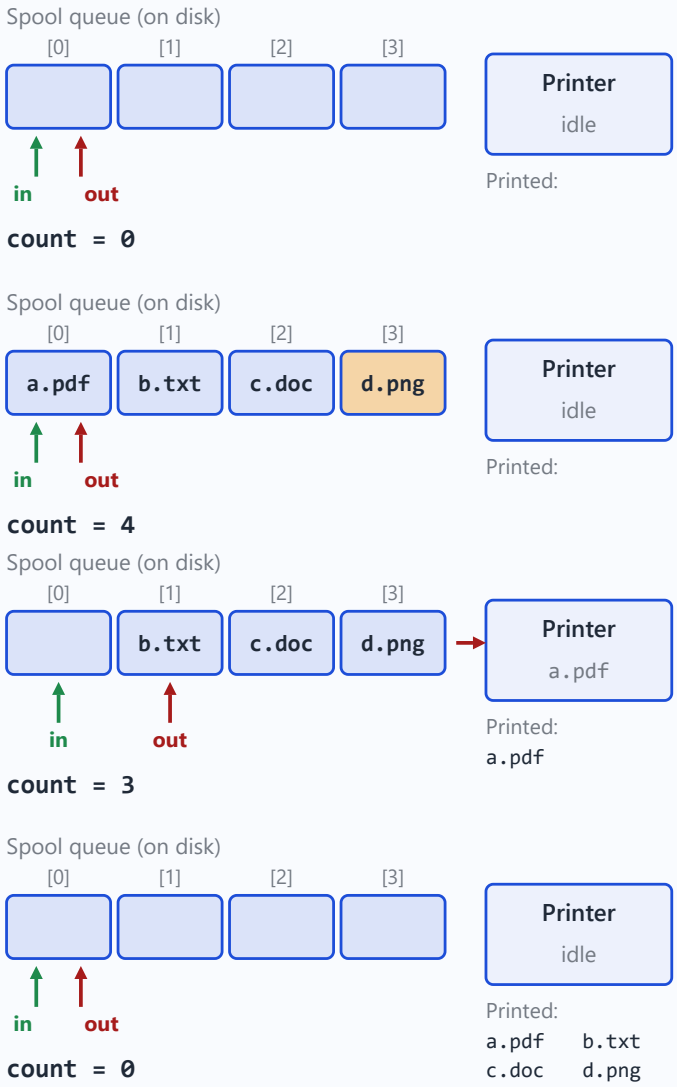


Figure 1.13: Four moments of the trace. (a) The queue is empty; `in` and `out` point to slot 0. (b) After four `enqueue` calls the queue is full, `count = 4`; `in` has wrapped around. (c) a.pdf has gone to the printer, `out = 1`, `count = 3`. (d) Four jobs have been printed, the queue is empty again, and the printer is idle.

Two rows of the table call for special attention. In the fourth step, `in = (3 + 1) % 4 = 0`: the index *wraps around* to the beginning of the array. At this moment `in` and `out` are equal, but the queue is full, not empty. Equality of the two indices alone cannot tell empty from full; that is exactly why the

`count` counter is kept. In the eighth step `out` also wraps to 0, and because `count = 0` the loop ends. The jobs have been printed in arrival order, that is, in FCFS order. Because only the service uses the printer, the outputs have not been mixed.

The line `queue[out] = NULL;`, which frees the slot, is not required for the queue to work correctly: the counters `out` and `count` already determine which slots are full. The line makes the trace readable and keeps the name of a printed job from lingering in the array. The empty slots shown as "·" from the fifth step of the table onward are the result of this line.

Cost. `enqueue()` makes three assignments; it finishes in constant time, that is, *O(1)*, regardless of the number of jobs in the queue. If the queue holds *n* jobs, `spooler()` runs its loop *n* times: *O(n)*. The queue itself takes a fixed space of 4 slots.

> **Common mistake**
>
> Using this code as it is in a real system. Two things are missing. First, `enqueue()` does not check whether the queue is full; if a fifth job arrives, it overwrites the oldest one. Second, the programs and the service are separate processes and may run at the same time. `count++` and `count--` are not a single instruction but a "read, modify, write" sequence; if the other side slips in between, one update is lost. This situation is called a race condition. Real spool queues are therefore protected by a lock.

1.3.7 Measuring resource usage

To distribute resources fairly, one must first know who uses how much. The operating system keeps this information separately for each process. Table 1.3 summarizes the metrics for three basic resources and the tools that show them.

Resource	Metric	Linux	Windows
CPU	CPU time spent in user and kernel mode, utilization percentage	`/proc/[pid]/stat`, `top`, `time`	Task Manager, Performance Monitor
Memory	Resident set, virtual size, working set; allocation and release pattern	`/proc/[pid]/status`, `smem`	"Working set" column
I/O	Bytes read and written, number of system calls and I/O requests	`/proc/[pid]/io`, `iotop`, `strace -c`	Resource Monitor

Table 1.3: Resource usage metrics and tools.

Splitting CPU time into user mode and kernel mode is meaningful. A program that is computing spends its time in user mode. A program that reads many small files spends most of its time in the kernel, inside system calls. On Linux the `time` command prints these two values separately as `user` and `sys`.

For memory, the value most often examined is the **resident set size** (RSS): the total of the process's pages that are currently in physical memory. The virtual size is the whole address space the process has reserved and can be much larger than the RSS, because not every reserved page is actually brought into memory. On Linux these values are read from files in the `/proc` directory:

```
$ grep -E 'VmRSS|Threads' /proc/4211/status
VmRSS:    412360 kB
Threads:  57
```

Here two lines are selected for process 4211 (the Firefox process of the earlier terminal example). `VmRSS` says that 412,360 kB of the process is in physical memory, and `Threads` says that the process contains 57 threads. `/proc` is not a directory that really exists on disk. The kernel generates the contents of these files from the current values on every read.

The source of these measurements is the kernel. On every context switch, every system call, and every page fault, the kernel updates the per-process counters it keeps. Tools such as Task Manager or `top` merely read and display these counters.

1.3.8 Fairness and efficiency

The growth of resources, that is, more cores and more memory, improves performance and response time. But it also makes it harder to share resources fairly and efficiently. On a four-core machine, deciding which process runs on which core is far more complicated than the “who is next?” question of a single-core machine. Operating systems approach this problem with four tools.

- **Scheduling**: which process runs when, and on which core? The kernel component that makes this decision is called the **scheduler**. Linux used CFS (the Completely Fair Scheduler) for many years; with Linux 6.6 (2023) it was replaced by the EEVDF scheduler.
- **Prioritization**: some processes are given a larger share than others. `nice` values on Linux, priority classes on Windows, and real-time classes on both systems serve this purpose.
- **Quotas and limits**: an upper bound is placed on the resources a process or a group of processes may use. On Linux, cgroups v2 (the foundation of containers), `ulimit`, and disk quotas serve this purpose; on Windows, Job objects do.
- **Trade-offs**: throughput, response time, and fairness cannot all be optimized at once. While one improves, another may get worse.

The last item rests on three concepts. **Throughput** is the number of jobs completed per unit time. **Response time** is the time from a request being made to the system's first response. **Fairness** is whether each process gets the share it deserves. For example, switching often between processes shortens response time, because no process waits long. But every switch has a cost; as the number of switches grows, the time the processor spends on real work, and therefore throughput, falls.

The right balance depends on the workload. On a server the priority is throughput: the number of requests processed per hour should be as high as possible. On a desktop computer the priority is response time: a mouse click must get an immediate response. In a real-time system the priority is that jobs finish before their deadlines.

1.3.9 Three big ideas: four small experiments

The book *Operating Systems: Three Easy Pieces* (OSTEP) by Arpaci-Dusseau and Arpaci-Dusseau organizes operating system topics around three big ideas: **virtualization**, **concurrency**, and **persistence**. Each idea can be made concrete with a small experiment run on Linux. Because virtualization is shown separately for the processor and for memory, there are four experiments.

Definition 1.9

Virtualization is the technique of making a single physical resource look to each program like a separate resource of its own. The operating system virtualizes the processor and memory.

Experiment 1: virtualizing the processor. A program called `print` prints the letter it is given to the screen in an endless loop. Three copies are started in the background on a single-core machine:

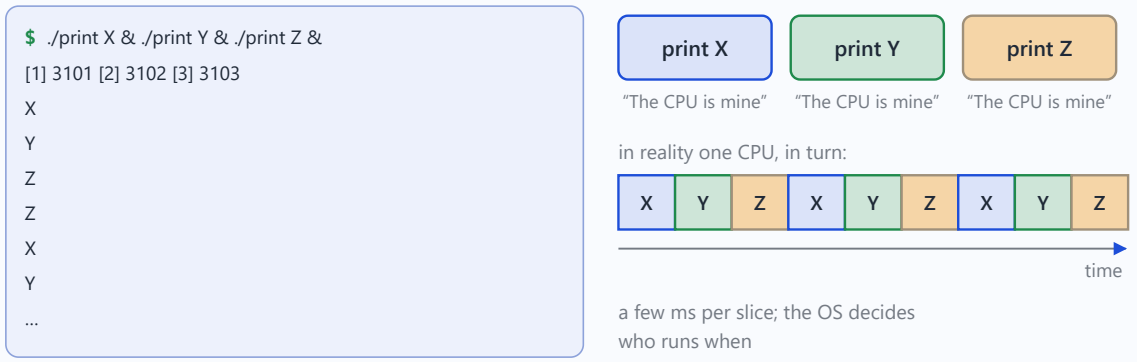


Figure 1.14: Virtualizing the processor. The command `./print X & ./print Y & ./print Z &` starts three processes numbered 3101, 3102, and 3103; the letters X, Y, and Z appear in the output in mixed order. Each program thinks "the CPU is mine"; in reality the single CPU is given to processes X, Y, and Z in turn. Each slice is a few milliseconds, and the OS decides who runs when.

All three programs appear to do their work without interruption; in the output the three letters appear in mixed order. Yet the machine has a single core. The operating system gives the processor to the three processes in turn, in slices of a few milliseconds. Because the switches are too frequent for the human eye to notice, each behaves as if it had its own processor. Two separate questions arise here. Deciding who gets the next slice is a question of *policy* and is answered by the scheduling algorithm. How the switch is carried out is a question of *mechanism* and is answered by the context switch. The distinction between the two concepts is covered in 1.8.1.

Experiment 2: virtualizing memory. A program called `counter` allocates memory for an integer, prints the address of that memory, and then increments and prints the value in a loop. Two copies are run at the same time:

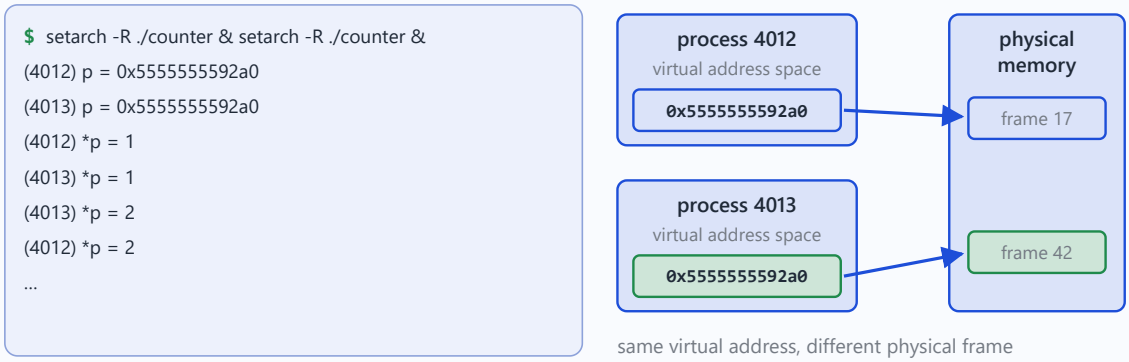


Figure 1.15: Virtualizing memory. Processes 4012 and 4013 print the same address (`0x5555555592a0`), but each one's value increases independently as 1, 2, ... Both processes use the same virtual address; the OS maps these addresses to different physical frames (17 and 42).

Both processes print the same address. If the memory were really shared, the two processes would increment the same integer and the values would advance together as 1, 2, 3, 4. Instead, in each process the value starts at 1 and increases independently. The explanation is that the address programs see is a **virtual address**. Each process sees its own address space, and the operating system, with the help of the hardware, maps the same virtual address to a different physical memory frame for each process. The experiment used the `setarch -R` command, which turns off address space layout randomization (ASLR). Had ASLR stayed on, the addresses would change on every run and the sameness would not be visible.

Experiment 3: concurrency. The `incr` program starts two threads. Each thread increments a shared `counter` variable *N* times. The expected result is *2N*.

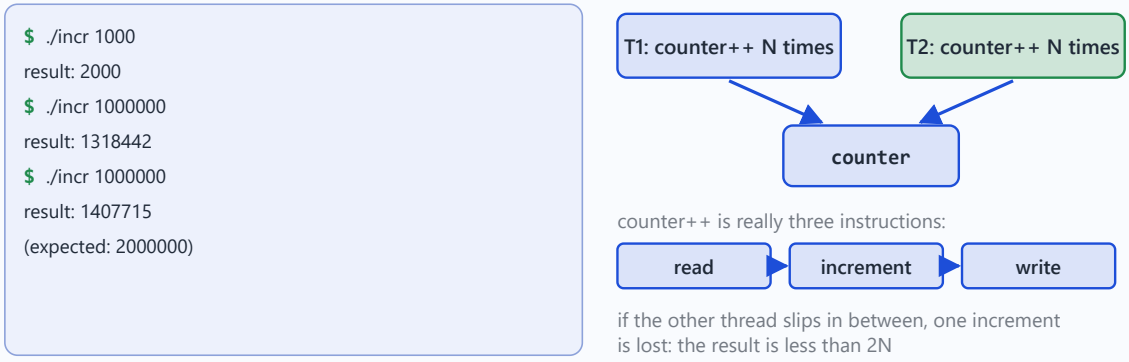


Figure 1.16: Concurrency. `./incr 1000` gives the correct result (2000). `./incr 1000000` , however, gives 1,318,442 in one run and 1,407,715 in another instead of 2,000,000. T1 and T2 increment the same

`counter`; `counter++` is really three instructions (read, increment, write), and if the other thread slips in between, one increment is lost.

For *N = 1000* the result looks correct, because the threads run so briefly that they hardly meet each other. When *N* reaches a million, the result is less than *2N* and a different value comes out on every run. The reason is that the expression `counter++` is not *atomic*: at the machine level it is three separate instructions, reading the value from memory into a register, incrementing it in the register, and writing it back. If two threads read the same old value, both increment it by one, and both write the same new value, one of the two increments is lost. Managing concurrent access to shared data correctly is one of the hardest topics in operating systems.

Definition 1.10

Concurrency is the progress of several jobs interleaved in time. Jobs can advance on a single processor by taking turns in slices; their running truly at the same time on several cores is called **parallelism**.

Experiment 4: persistence. The `save` program writes "hello disk" to the file `note.txt`. After the computer is restarted, the file can still be read:

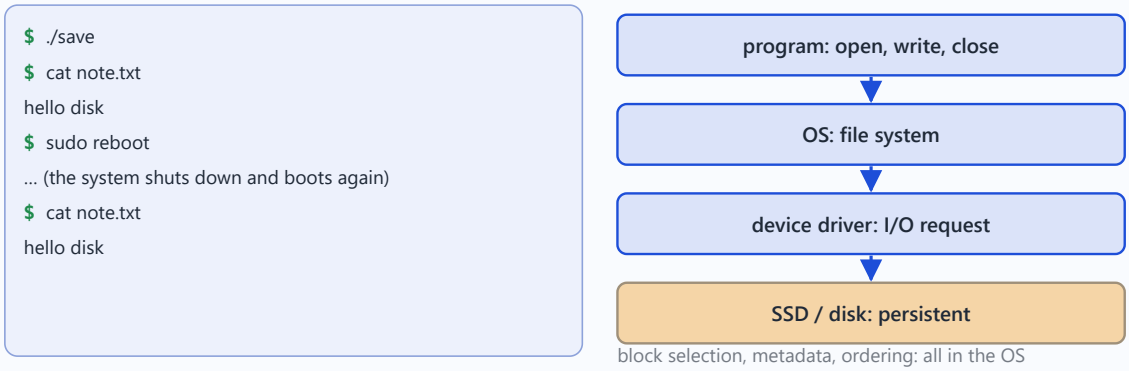


Figure 1.17: Persistence. `./save` is run, and `cat note.txt` prints "hello disk"; after the system boots again with `sudo reboot`, the same content is still read. The program only calls `open`, `write`, and `close`; the file system, the device driver, and the I/O request are inside the OS; the data is persistent on the SSD or disk.

Main memory (DRAM) is *volatile*: its contents are erased when power is cut. If the data is to survive a restart, it is written to a persistent device through the file system. All the program does is call `open`, `write`, and `close`. Which block the data goes to, where metadata such as the file's name, size, and permissions is kept, and how to talk to the device are the operating system's job.

1.4 History

Many ideas found in today's operating systems were born as solutions to a particular problem of a particular era. Spooling, multiprogramming, and time sharing are not fashions but ways of not wasting an expensive resource. Knowing the history makes it easier to understand why these ideas exist.

1.4.1 Generations

Tanenbaum divides the development of computers and operating systems into generations according to the leaps in hardware technology.

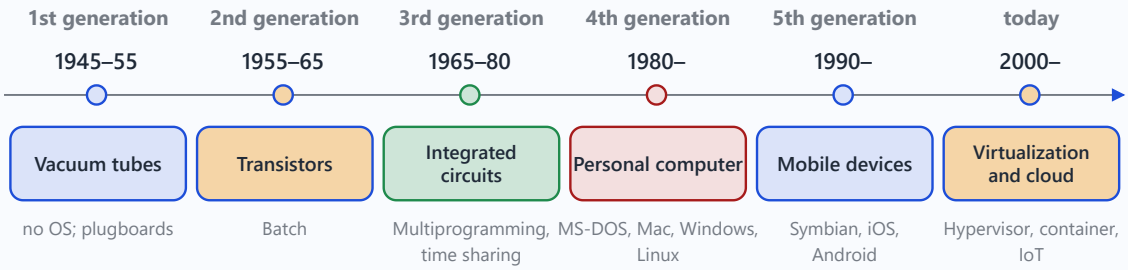


Figure 1.18: Generations of operating systems. 1st generation (1945–55): vacuum tubes, no OS, plugboards; 2nd generation (1955–65): transistors and batch processing; 3rd generation (1965–80): integrated circuits, multiprogramming, and time sharing; 4th generation (1980–): personal computers, MS-DOS, Mac, Windows, Linux; 5th generation (1990–): mobile devices, Symbian, iOS, Android. The last box (2000–, virtualization and cloud: hypervisor, container, IoT) is not one of Tanenbaum's generations; it is added to help understand today's systems.

Each generation begins with a new electronic technology: the vacuum tube, the transistor, the integrated circuit, the microprocessor. As the technology changes, the size, price, and reliability of the machines change. What does not change is the question asked anew in every generation: how can we make better use of the expensive resource? In the first generations that resource was the processor itself. As prices fell, the expensive resource became memory and finally *human time*.

1.4.2 First and second generation

The **first generation (1945–55)** is the era of machines built with vacuum tubes. Machines such as ENIAC (1945) filled a room; they were slow and broke down often because tubes burned out frequently. The same small group of engineers designed, built, programmed, and maintained the machine. Programming was done in machine language or by changing connections on plugboards; in the early 1950s punched cards began to be used. There was neither a programming language nor an operating system. The machines were mostly used to compute numerical tables.

The **second generation (1955–65)** began with the transistor. The transistor replaced the tube; machines became smaller and more reliable and used less energy. The mainframes of this era sat in air-conditioned rooms and cost millions of dollars. An **operator** came between the programmer and the machine: the programmer handed the card deck to the operator, the operator ran the job and returned the result as printed paper.

This arrangement had a big loss. When a job finished, the million-dollar machine sat idle until the operator had the cards of the next job read in and the compiler loaded. The solution was **batch processing**.

Definition 1.11

A **batch system** is a system in which jobs are collected together in advance and a small control program, the **monitor**, runs these jobs one after another without human intervention.

The purpose of batch processing was to eliminate the manual transition time between jobs. Jobs were collected on a magnetic tape, and the monitor ran them automatically in sequence; the operator only carried tapes. But two limits remained. Because only one job was in memory at a time, the processor sat idle while the job did I/O. The programmer could also get the result only hours after submitting the job; there was no interaction. Batch processing was unsuitable for interactive and real-time use.

1.4.3 An early batch system

A typical second-generation installation used two separate IBM machines together: an inexpensive IBM 1401 and an expensive IBM 7094.

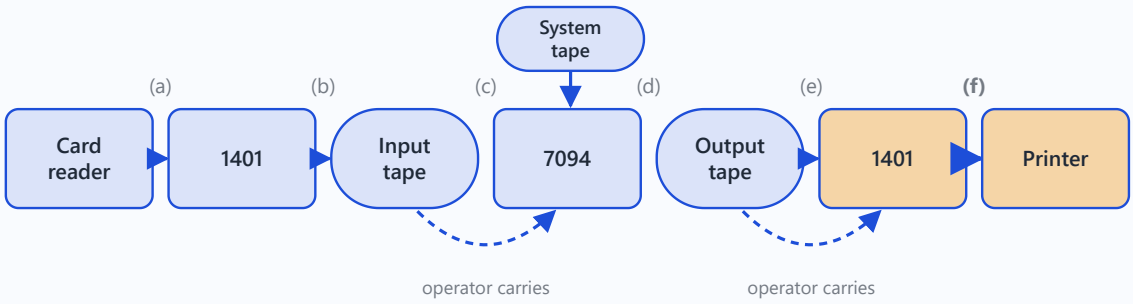


Figure 1.19: An early batch system. (a) The card reader; (b) the 1401 writes the cards to the input tape; (c) the operator carries the input tape to the 7094; (d) the 7094 computes the jobs with the monitor on the system tape and writes the results to the output tape; (e) the operator carries the output tape to the 1401; (f) the 1401 prints the results on the printer.

1. (a) Programmers bring their punched card decks to the 1401's card reader.
2. (b) The inexpensive 1401 reads the cards of a group of jobs and writes them to the input tape. In this way the slow card reading is separated from the expensive machine.

3. (c) The operator carries the filled input tape to the 7094.
4. (d) Under the control of the monitor program on the system tape, the 7094 computes the jobs one after another and writes the outputs to the output tape.
5. (e) The operator carries the output tape back to the 1401.
6. (f) The 1401 prints the outputs on the printer. This was called *offline printing*.

The logic of this arrangement is clear: the expensive 7094 never bothers with reading cards or printing. Reading from tape is much faster than reading from cards, so the expensive machine's time is left for computation. Slow I/O jobs were handed over to the cheap machine. The same principle lives on, in a slightly different form, in today's spooling.

1.4.4 IBM 1401, 7094, and System/360

The three machines of this period are each a separate step in the evolution of operating systems. Table 1.4 compares them.

Machine	Year	Characteristics
IBM 1401	1959	Commercial data processing; character-based and relatively cheap. It was also used for I/O jobs such as reading cards, copying to tape, and printing. It was one of the most widespread computers of the 1960s.
IBM 7094	1962	Scientific computing; a 36-bit word and an architecture entirely different from the 1401. It ran batch jobs under IBSYS and FMS. Time sharing with CTSS was tried at MIT on a modified 7094.
System/360	1964	A family of machines sharing the same architecture from small to large. Its operating system, OS/360, was a giant project of millions of lines, full of bugs, that included multiprogramming and spooling.

Table 1.4: Three IBM machines.

Two details in the table are often confused. First, the 1401 and the 7094 did not run under OS/360; OS/360 was written for the System/360 family announced in 1964. Second, the 7094 was not a powerful version of the 1401; it was a separate machine with a completely different architecture that complemented it. The fact that the two machines could not run each other's programs pushed IBM to build, with System/360, a family based on a single architecture. This family idea meant that a program written on a small machine could run unchanged on a large one, and it is also the foundation of today's x86 and ARM families.

The development of OS/360 became one of the classic examples of software engineering. Fred Brooks, one of the project's managers, described his experience in the book *The Mythical Man-Month*. The best-known observation in the book is that adding people to a late software project makes it later still.

1.4.5 A typical FMS job

FMS is a batch monitor written for the IBM 709 and 7094; its name is short for Fortran Monitor System. A job was a deck of cards wrapped in special cards that told the monitor what to do.

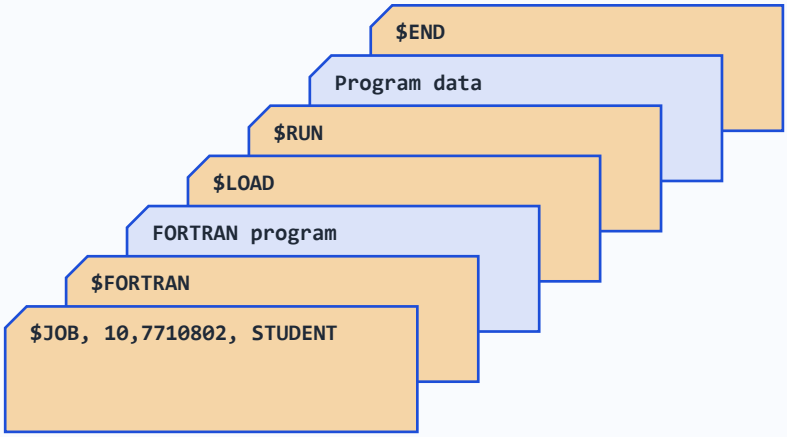


Figure 1.20: The card deck of a typical FMS job. From the first card at the front to the back: `$JOB` (maximum run time, account number, programmer name), `$FORTRAN`, the FORTRAN program, `$LOAD`, `$RUN`, the program data, and `$END`.

Cards that begin with $ are called **control cards**. The monitor reads the deck in order and does one job at each control card. `$JOB` starts a new job and opens its account. `$FORTRAN` loads the compiler and has it compile the cards that follow. `$LOAD` loads the compiled program into memory. `$RUN` runs the program and supplies the following cards as its data. `$END` ends the job. These cards are the ancestors of today's shell scripts and of IBM's job control language, JCL. A shell script, too, is a list of commands that spells out the “compile, link, run” steps in order.

Common mistake

Confusing the abbreviation FMS with the Flexible Manufacturing System of factory automation. The FMS here is the FORTRAN Monitor System; there is no connection between the two concepts.

1.4.6 Third generation: integrated circuits and multiprogramming

The **third generation (1965–80)** is the era of integrated circuits. Compatible families such as System/360 appeared; machines became smaller, cheaper, and more reliable. This generation gave operating systems three lasting ideas.

The first idea is **multiprogramming**.

Definition 1.12

Multiprogramming is the technique of dividing memory into partitions, keeping a separate job in each partition, and giving the processor immediately to another job in memory when one job waits for I/O.

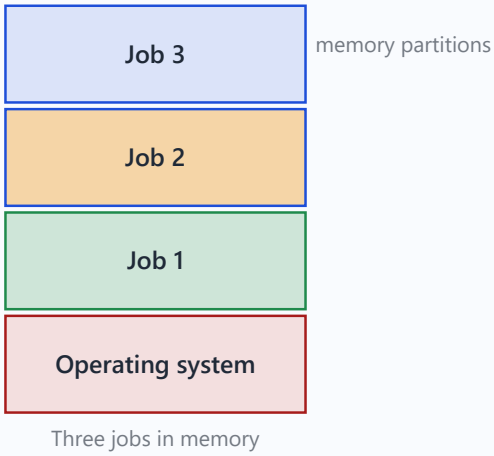


Figure 1.21: In multiprogramming, memory is divided into four partitions: the operating system at the bottom, with job 1, job 2, and job 3 above it.

In the second generation there was a single job in memory, and the processor waited while the job read from tape. In scientific computing this wait was small; but in commercial data processing jobs spent most of their time on I/O and most of the processor's time was wasted. In multiprogramming, as in Figure 1.21, several jobs sit in memory together. While one waits for I/O, the processor runs another.

On a single-processor machine the jobs do not really run at the same time; the processor executes only one at a given instant. The jobs advance interleaved in time. This is concurrency, not true parallelism. Having several jobs in memory also brings a new problem: a job must not write to another's memory, whether through a bug or on purpose. For this reason multiprogramming cannot be done safely without hardware that protects memory.

The second idea is **spooling**. Cards were no longer read through a separate machine and tape but directly onto disk. When one job finished, the next was loaded from disk into memory. The 1401-and-tape-carrying arrangement thus disappeared.

The third idea is **time sharing**.

Definition 1.13

Time sharing is rotating the processor among processes in short time slices so that many users can use the same machine interactively from their own terminals. When a process's slice runs out, the processor is taken from it by a timer interrupt even if it does no I/O.

The difference between multiprogramming and time sharing is subtle but important. In multiprogramming the processor is taken from a job only when that job starts waiting; a job that does no I/O at all can hold the processor for hours. In time sharing, a hardware timer raises an interrupt at

regular intervals and each process's slice is limited. The goal is less throughput than a short response time for every user. The programmer no longer submits a job and waits for hours; they type a command and get a reply within seconds.

The family tree of time sharing extends to today's systems: CTSS (1961), developed at MIT; then the very ambitious MULTICS project; then UNIX (1969), written by Bell Labs researchers who had left MULTICS. The BSD releases at Berkeley were born from UNIX. The POSIX standard was written so that the different UNIX derivatives would be compatible with each other. Andrew Tanenbaum developed MINIX for teaching, and Linus Torvalds, inspired by MINIX, wrote the Linux kernel in 1991.

1.4.7 Multiprogramming timeline

To see the gain of multiprogramming in numbers, let us run two jobs on a single-processor machine. Times are in time units:

- Job **A**: 2 units of CPU, 4 units of I/O, 2 units of CPU.
- Job **B**: 3 units of CPU, 3 units of I/O, 1 unit of CPU.

The total CPU need of the two jobs is *2 + 2 + 3 + 1 = 8* units. Each job's I/O is done on its own device, so A's I/O and B's I/O do not wait for each other.

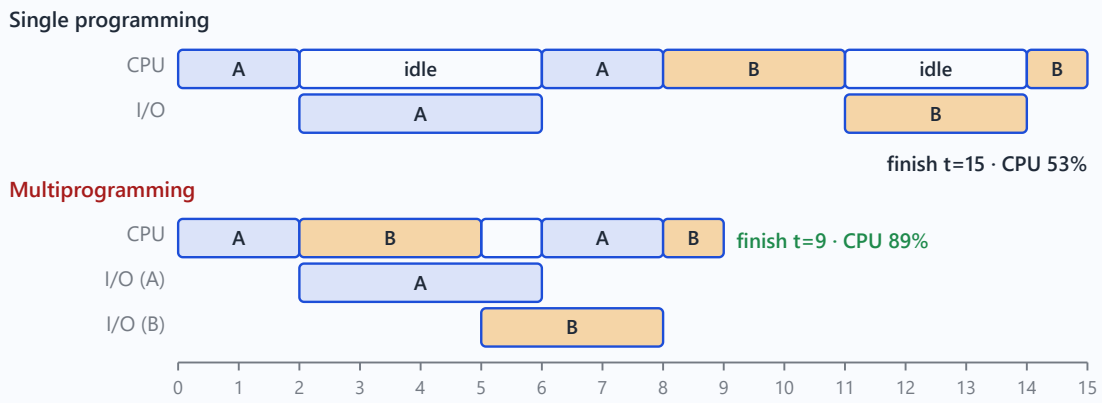


Figure 1.22: Timelines of single-programmed and multiprogrammed systems. Top, the single-programmed system: A 0–2, CPU idle 2–6, A 6–8, B 8–11, CPU idle 11–14, B 14–15; finish *t = 15*, CPU utilization 53%. Bottom, the multiprogrammed system: A 0–2, B 2–5, idle 5–6, A 6–8, B 8–9; A's I/O 2–6, B's I/O 5–8; finish *t = 9*, CPU utilization 89%.

Example 1.2

Single-programmed system. There is only one job in memory; B starts only when A has completely finished.

1. *t = 0–2*: A uses the CPU.
2. *t = 2–6*: A does I/O. Because there is no other job in memory, the CPU stays idle for 4 units.

3. *t = 6–8*: A uses its last CPU burst and finishes.
4. *t = 8–11*: B uses the CPU.
5. *t = 11–14*: B does I/O; the CPU is idle for 3 units.
6. *t = 14–15*: B uses its last CPU burst and finishes.

The total time is 15 units, and the time the CPU was working is 8 units. CPU utilization is *8 / 15 ≈ 0.53*, that is, about 53%. The 7 idle units are exactly the sum of the two jobs' I/O times (*4 + 3*).

Example 1.3

Multiprogrammed system. Both jobs are in memory; when the CPU becomes free it is given to a job that is ready.

1. *t = 0–2*: A uses the CPU; B waits, ready.
2. *t = 2*: A goes to I/O (2–6). The CPU does not stay idle; it is given to B at once: B runs during *t = 2–5*. Meanwhile A's I/O continues in the background.
3. *t = 5*: B goes to I/O (5–8). A's I/O has not finished yet; there is no ready job, and the CPU is idle only during *t = 5–6*.
4. *t = 6*: A's I/O finishes; A uses its last CPU burst during *t = 6–8* and finishes.
5. *t = 8*: B's I/O finishes at exactly this moment; B uses its last burst during *t = 8–9* and finishes.

The two jobs have finished in 9 units instead of 15. CPU utilization is *8 / 9 ≈ 0.89*, that is, about 89%. The only idle unit is the interval *t = 5–6*, when both jobs are waiting for I/O at the same time.

The difference between the two examples has a single source: in multiprogramming, CPU and I/O *overlap*. In the single-programmed system the CPU waits idle during A's 4 units of I/O, while in the multiprogrammed system 3 units of that time are filled with B's computation. The jobs' total CPU need has not changed; the waits are merely covered by other jobs' computation. That is why the gain of multiprogramming grows as the jobs' share of I/O rises. For two jobs that do no I/O at all, the gain is zero.

Common mistake

Thinking that multiprogramming makes jobs “run faster”. Each job's own CPU time stays the same; A still uses 4 units of CPU and B still uses 4. What gets shorter is the total time for all the jobs to finish. In the example A finishes at *t = 8* in both systems; the gain comes entirely from B finishing at 9 instead of 15. If A's I/O had lasted 2 units, it would have finished at *t = 4*.

Because the CPU would be with B at that time, A would wait until $t = 5$ and finish at $t = 7$; running alone it would have finished at $t = 6$. Multiprogramming speeds up the total work, but it can delay the completion of an individual job.

1.4.8 Fourth generation: personal computers

The **fourth generation (1980–)** began with very-large-scale integrated circuits that fit thousands of transistors onto a single chip, that is, microprocessors. For the first time the computer became small enough to fit on one person's desk and cheap enough for one person to buy. Table 1.5 lists the operating system milestones of this period.

Year	Development
1974	CP/M, written by Gary Kildall for the Intel 8080: the first widespread disk-based OS for microcomputers.
1981	The IBM PC and MS-DOS, derived from 86-DOS; the BASIC interpreter in ROM.
1973–84	The GUI ideas developed at Xerox PARC turned into a user-friendly interface with the Apple Lisa (1983) and the Macintosh (1984).
1985–2000	Windows 1.0 … 3.x was a graphical environment running on top of MS-DOS; Windows 95/98/ME were still MS-DOS-based.
1993–	Windows NT, written from scratch; then 2000, XP, 7, 10, and 11. The kernel of today's Windows is NT.
1991–	Linus Torvalds's Linux kernel; in 2001, the UNIX-based Mac OS X (Darwin/XNU), the ancestor of today's macOS.

Table 1.5: Milestones of the personal computer era.

In this period the goal of the operating system changed. In earlier generations the machine was expensive and human time was cheap; the operating system tried never to leave the machine idle even for a moment. When rapidly falling prices opened the computer to everyone, the balance reversed. Now what was valuable was the user's time, and the OS's goal shifted from “use the machine efficiently” to “make things easy for the user”. The rise of the GUI is a result of this shift.

The transition from MS-DOS to Windows NT is also instructive. MS-DOS was a single-tasking system with no memory protection; a bug in one program could crash the whole system. Windows 95/98/ME were built on this foundation and became known for their instability. Windows NT, on the other hand, was written from the start with protection, multitasking, and multiuser operation in mind. In 2001, with XP, home users moved to the NT kernel too.

1.4.9 Fifth generation: mobile devices and today

The **fifth generation (1990–)** is the era of mobile devices. In the 1990s PDAs and Palm OS were common, and in the 2000s Symbian and BlackBerry OS. In 2007 iOS arrived with the iPhone, and in 2008 Android, built on the Linux kernel. Microsoft's Windows Phone entered the market in 2010, and

its development was stopped in 2017. Today Android and iOS share the market. In mobile operating systems, the touch interface, battery management, and the permission model that determines which data applications can access stand out.

Alongside the mobile world, four trends shape today's operating systems:

- **Virtualization and the cloud**: **hypervisors** (KVM, Hyper-V, Xen) run many virtual machines on a single physical server. This is the foundation of cloud services.
- **Containers**: tools such as Docker and Kubernetes set up environments that share the same kernel but are isolated from each other. The isolation is provided by Linux namespaces and cgroups.
- **Heterogeneous hardware**: a chip holds a CPU, a GPU, and an NPU for artificial intelligence workloads together; large and small cores are used side by side in processors.
- **Security**: languages that provide memory safety at the language level are entering the kernel. Since Linux 6.1 the kernel has had Rust support.

1.5 Hardware

Because the operating system manages the hardware, the hardware's characteristics directly shape OS design. The number of registers determines the cost of a context switch, the speed gap between memory levels shapes scheduling decisions, and the number of cores determines the scale of concurrency problems. This section summarizes the hardware features that matter from the operating system's point of view.

1.5.1 Components of a personal computer

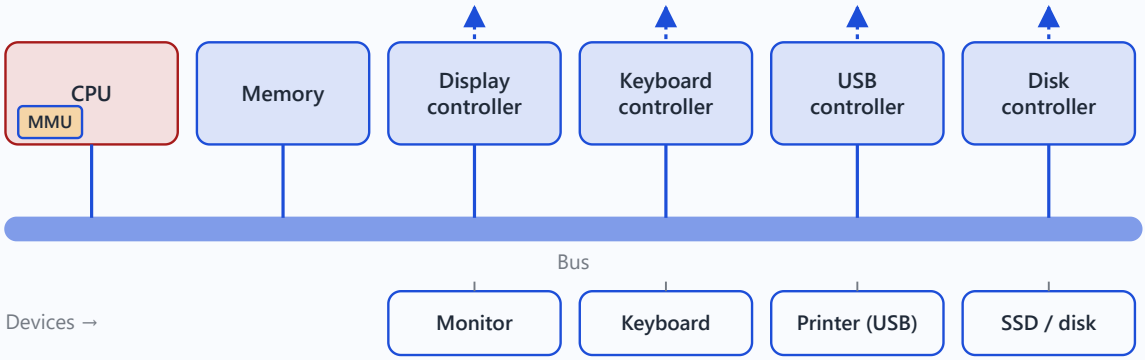


Figure 1.23: The components of a personal computer. The CPU (with the MMU inside), memory, and the display, keyboard, USB, and disk controllers are connected to a common bus. Below the controllers are the monitor, keyboard, USB printer, and SSD/disk devices.

Figure 1.23 is a simplified model. The processor, memory, and I/O devices communicate over a **bus**. Each device has a **controller**: the display controller drives the monitor, and the disk controller drives the disk. The processor talks not to the device directly but to its controller.

The **MMU** (memory management unit) seen inside the processor is the hardware that translates the virtual addresses programs use into physical addresses. It is thanks to this unit that, in the second experiment of 1.3.9, the same virtual address of two processes could be mapped to different physical frames. The translation rules, that is, the page tables, are written by the operating system; the MMU performs the translation on every memory access.

In real systems, point-to-point links such as PCIe are used instead of a single shared line; each device connects to the processor with its own link. But the conceptual model stays the same: the processor, memory, and device controllers are connected to each other, and the operating system manages the devices by talking to the controllers.

1.5.2 The processor and its registers

The processor is the computer's "brain": it executes all instructions. Its operation rests on a simple loop.

> **Definition 1.14**
>
> The processor's basic mode of operation is the **fetch–decode–execute cycle**: the next instruction is fetched from memory, what to do is decoded, and the instruction is executed. Then the processor moves on to the next instruction.

Because memory is slow compared with the processor, variables and intermediate results are kept in small, very fast storage units inside the processor, the **registers**. The instruction that moves data from memory into a register is called *load*, and the one that moves it from a register to memory is called *store*. Besides the general-purpose registers, there are special registers that hold the state of the processor.

Register	x86-64	ARM64
Program counter: address of the next instruction	`RIP`	`PC`
Stack pointer: top of the stack	`RSP`	`SP`
PSW: condition bits, mode, interrupt enable	`RFLAGS`	`PSTATE`
Number of general-purpose registers	16	31

Table 1.6: The main registers and their names in two architectures.

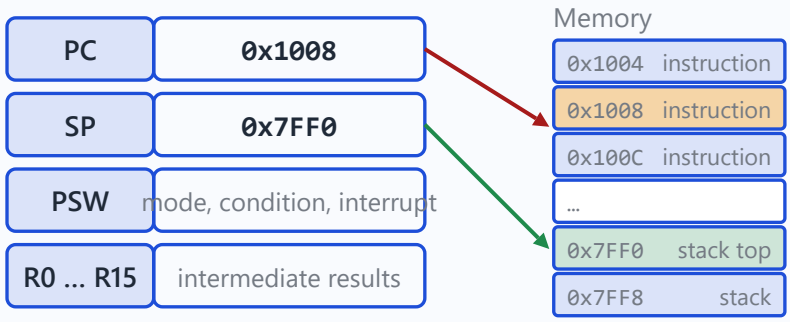


Figure 1.24: Registers and memory. The PC (`0x1008`) points to the next instruction and the SP (`0x7FF0`) to the top of the stack. The PSW holds the mode, condition, and interrupt bits; R0 … R15 hold intermediate results.

The **program counter** (PC) holds the address of the next instruction; it advances after each instruction and takes the new address on jump instructions. The **stack pointer** (SP) points to the top of the stack, where local variables and return addresses are kept during function calls. The **program status word** (PSW) holds the condition bits that record the result of the last comparison, the processor's mode (kernel or user), and whether interrupts are enabled. The mode bit in 1.2.2 is in this register as well.

Each processor family has its own instruction set. A program compiled for x86-64 does not run directly on an ARM processor; it must either be recompiled or pass through a translation layer. A processor's performance cannot be expressed by a single number: clock speed, number of cores, cache size, the number of pipeline stages through which instructions are processed, and memory bandwidth all play a part.

The real importance of registers for the operating system is the **context switch**.

Definition 1.15

A **context switch** is the processor's leaving one process and moving on to run another. When doing this, the operating system saves all the registers of the process being left to memory and restores the previously saved registers of the next process.

It is easy to see why saving is mandatory. The processor has a single set of registers. When the next process starts running, it overwrites all of these registers with its own values. If the interrupted process is later to continue where it left off, it must return to the same PC, the same SP, and the same intermediate values. If even a single register were saved incompletely, the process would continue computing with a wrong value when it returned, and the error would stay hidden so well that its cause could not be found.

1.5.3 Multicore chips

Raising the clock speed hit the heat limit in the mid-2000s. As a result, manufacturers began to put several processors on the same chip instead of a single fast one. Today even phones have eight or more cores.

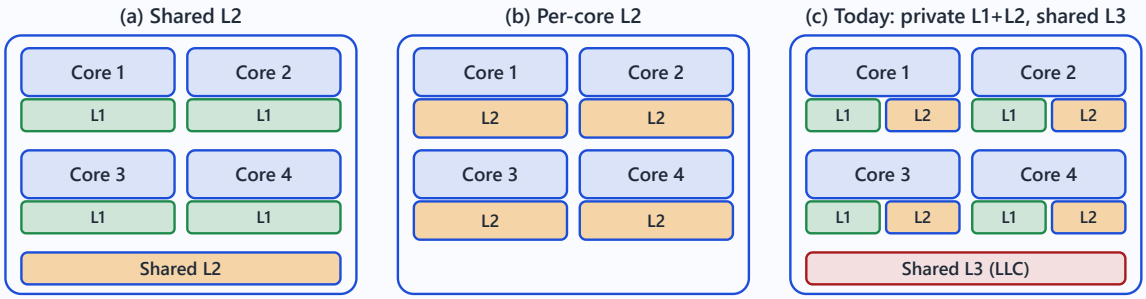


Figure 1.25: Cache layouts in four-core chips. (a) Each core has its own L1 and there is a shared L2. (b) A private L2 for each core. (c) Today's common layout: a private L1 and L2 for each core and an L3 shared by all cores (the last-level cache, LLC).

Three separate ideas coexist in multicore chips:

- **Multicore**: a chip holds several independent processors. The operating system sees each of them as a separate processor and schedules a separate process on each.
- **Simultaneous multithreading** (SMT; Intel's commercial name is Hyper-Threading): one core runs two hardware threads at the same time. The operating system sees two “logical CPUs”, but these two share the same core's execution units and caches.
- **Hybrid cores**: the same chip has performance (P) cores and efficiency (E) cores. ARM's big.LITTLE design and the processors Intel has made since 2021 are of this kind. The scheduler must know which kind of core to run which job on: a background update can proceed on an E core without draining the battery, while a game's main thread belongs on a P core.

Example 1.4

A laptop processor has 6 performance cores with SMT and 8 efficiency cores without SMT. How many logical CPUs does the operating system see?

Solution. Because each P core runs two hardware threads, it gives $6 \times 2 = 12$ logical CPUs. The E cores count as one logical CPU each: 8. In total $12 + 8 = 20$ logical CPUs are visible. These 20 CPUs are not equal in power. Two logical CPUs that share the same P core are not as fast as two separate cores; that is why the scheduler first tries to fill the idle physical cores.

For the operating system the consequences of multicore fall under three headings. **Parallelism**: jobs now really run at the same time, and the operating system must distribute them evenly across the cores. **Synchronization**: two processor cores may try to modify the same data structure of the OS kernel at the same time; the problem of the counter experiment in 1.3.9 now exists inside the OS itself. **Cache coherence**: because each core has its own cache, a value written by one core must invalidate the stale copy in the other core's cache. This last job is done by the hardware, but its cost affects operating system design.

1.5.4 The memory hierarchy

The ideal memory would be very fast, very large, and very cheap at the same time. No such technology exists. Fast memory is expensive and made small; large and cheap memory is slow. Computers therefore use different technologies together in layers.

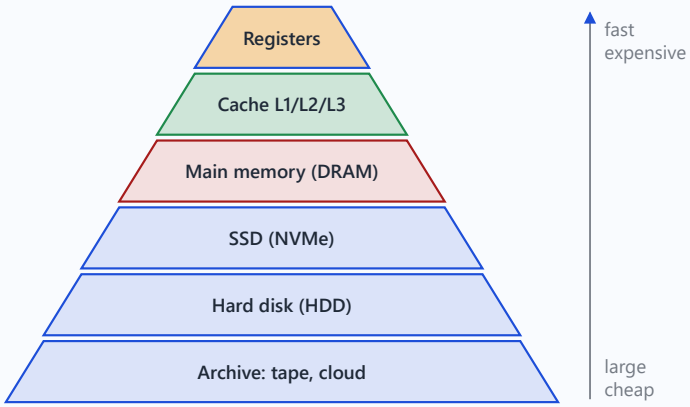


Figure 1.26: The memory hierarchy pyramid. From top to bottom: registers, L1/L2/L3 caches, main memory (DRAM), SSD (NVMe), hard disk (HDD), and archive (tape, cloud). Going up, memory gets faster and more expensive; going down, it gets larger and cheaper.

Level	Access (approx.)	Capacity (typical)
Registers	< 1 ns	< 1 KB
L1 cache	~1 ns	32–64 KB
L2 / L3	3–20 ns	1–100+ MB
DRAM	60–100 ns	8–64 GB
NVMe SSD	20–100 µs	0.5–4 TB
HDD	5–10 ms	2–24 TB

Table 1.7: Approximate access times and typical capacities of the memory levels (orders of magnitude as of 2026).

Definition 1.16

A **memory hierarchy** is the arrangement of memory technologies of different speed, capacity, and cost in layers, so that each level works as a cache for the one below it.

Two boundaries in the table deserve special attention. The first is the **volatility** boundary: DRAM and the levels above it lose their contents when power goes, while SSDs and HDDs are persistent. This makes clear why persistence requires a file system. The second is **the size of the gaps**: between neighboring levels the speed changes sometimes by a factor of ten and sometimes by a factor of a thousand. Going from DRAM to SSD, access time rises from nanoseconds to microseconds, and to milliseconds for an HDD.

The reason the hierarchy works is **locality**. Within a short time, programs usually access the same data again and again, or data at nearby addresses. A loop uses the same variables on every iteration; an array is read in order from start to finish. So a small copy kept at the upper level satisfies most accesses, and the slow lower level is rarely visited. As a result the system behaves like a memory with almost the speed of the upper level and the capacity of the lower level.

Example 1.5

How long does one clock cycle take on a core running at 2.5 GHz? To how many clock cycles does a 100 ns DRAM access correspond, and an 8 ms HDD access?

Solution. 2.5 GHz means 2.5×10^9 cycles per second. One cycle takes $1 / (2.5 \times 10^9)\ s = 0.4\ ns$; in other words, 2.5 cycles pass in 1 ns. A DRAM access corresponds to $100 \times 2.5 = 250$ cycles. The HDD access is first converted to nanoseconds: $8\ ms = 8 \times 10^6\ ns$; from this, $8 \times 10^6 \times 2.5 = 2 \times 10^7$, that is, 20 million cycles.

These numbers explain the behavior of the operating system. The hardware caches hide the 250-cycle DRAM wait; it is not worth the operating system stepping in to switch processes, because the context switch itself takes longer than that. The 20-million-cycle disk wait, however, is huge. During this time the operating system keeps the process waiting and gives the processor to another process. This is the basic justification for multiprogramming.

Common mistake

Thinking of the SSD as “a fast kind of memory”. An NVMe SSD is about a hundred times faster than an HDD, but it is still about a thousand times slower than DRAM. For the operating system an SSD is also an I/O device: while a read is awaited, the process is kept waiting and the processor is given to other work.

1.6 I/O methods

The processor's talking to devices is one of the things the operating system does most often. A key press, the arrival of a network packet, the reading of a disk block: all are I/O events. This section covers how the processor learns that an I/O job has finished.

1.6.1 Controller and driver

Every I/O device has a **device controller**. The controller is a small processor or circuit on the device; it drives the device physically and presents the processor with an interface made of a few registers. Typically it has a *command* register (what to do), a *status* register (is the job done, is there an error), and a *data* register or buffer.

The **device driver** in the kernel is the software that knows how to talk to a particular controller. The driver writes commands into the controller's registers and reads its status. Because every controller is different, each type of device has its own driver; the rest of the operating system talks to drivers through a common interface. These are the drivers listed in Device Manager in 1.1.6.

1.6.2 Polling, interrupts, and DMA

After the driver gives the controller a command, it must find out when the job is done. There are three ways to do this.

Method	Key difference
Polling	The CPU reads the status register in a loop until the device is ready; it cannot do other work while waiting (busy waiting).
Interrupt	The device signals the CPU at the end of each unit of work; in between, the CPU runs another process.
DMA (direct memory access)	The controller moves the whole block to memory itself; the CPU is involved only at the start (programming) and at the end (a single interrupt).

Table 1.8: Three I/O methods.

The logic of **polling** is simple: the processor reads the status register in a loop, reading it again until it sees the “ready” bit. This method is also called **busy waiting**, because while waiting the processor runs at full power but does no useful work. If the device responds very quickly, that is, if the wait is only a few cycles, polling is the cheapest way. If the wait is long, processor time is wasted.

With **interrupts**, after giving the controller the command, the processor goes back to running another process. When the device finishes a unit of work, for example a byte or a word, it sends the processor an **interrupt** signal. The processor completes the current instruction, switches to kernel mode, finds the relevant **interrupt handler** in a table called the interrupt vector, and runs it. The handler takes the data, gives the device the next job, and the interrupted process continues where it

left off. In this way the processor does useful work during the waits. But because an interrupt arrives at the end of every unit of work, in large transfers the number of interrupts and their total cost become very high.

DMA (direct memory access) solves this last problem. The processor gives the controller only the information "starting at this address, write this many bytes to memory". The controller moves the whole block to memory itself and raises *a single* interrupt when the transfer is over. The processor is involved only at the start, when programming the DMA, and at the end, when handling that single interrupt.

1.6.3 Computing CPU load

Example 1.6

A 4 KiB block is to be received from a device. In interrupt-driven I/O, one interrupt occurs for every 4 bytes, and each interrupt costs the CPU 2 µs. With DMA, the CPU programs the controller in 3 µs and a single interrupt (2 µs) arrives at the end of the transfer. How much time does the CPU spend on this transfer in the two methods?

1. First the block is converted to bytes. KiB is a binary prefix: $4\ KiB = 4 \times 1024 = 4096$ bytes.
2. Interrupt-driven I/O: the number of interrupts is $4096 / 4 = 1024$. The time the CPU spends is $1024 \times 2\ \mu s = 2048\ \mu s \approx 2.05\ ms$.
3. DMA: $3\ \mu s + 2\ \mu s = 5\ \mu s$.
4. Ratio: $2048 / 5 \approx 410$. DMA reduces the CPU load by a factor of about 410.

The transfer itself depends on the device's speed in both methods and takes about the same time. The difference lies in how the processor spends that time. With DMA, the processor runs other processes apart from those 5 µs. For slow devices such as a keyboard, which produce a few events per second, this difference is negligible; for fast block devices such as SSDs and network cards, working without DMA is unthinkable.

Common mistake

Thinking that DMA also provides security. DMA is a performance mechanism; it takes the copying load off the processor. On the contrary, a device that can write directly to memory can corrupt any part of memory if it is programmed wrongly or is malicious. A separate hardware unit called the IOMMU is needed to limit the memory regions that devices can access.

1.6.4 Interrupt, trap, and exception

The word interrupt is used for two different events, so care is needed here. A **hardware interrupt** comes from a device outside the processor: the timer has expired, a packet has arrived at the network card, the user has pressed a key. These events are independent of the instruction stream being executed and can arrive at any moment; that is why they are *asynchronous*.

A **trap** or **exception**, on the other hand, arises from the executing instruction itself: division by zero, an invalid memory access, a page fault, or a deliberate system call. Because it is the result of a particular instruction, it is *synchronous*: if the same program is run with the same input, the trap occurs at the same point every time.

Although their sources differ, the two events reach the kernel in the same way. The processor enters kernel mode, saves the current state, and jumps to the relevant handler from a table (the interrupt vector or the exception table) according to the type of event. When the handler finishes, it either returns to just after the interrupted instruction or, if there is a fatal error, terminates the process. This common path is the same as the "three doors" of 1.2.2: system calls and exceptions come from within the instruction stream, interrupts from outside.

Example 1.7

Which of the following events are interrupts and which are traps? (a) the disk completing a read, (b) a program reading through a `NULL` pointer, (c) a `fork()` call, (d) a mouse movement, (e) a page fault.

Solution. (a) and (d) come from outside devices and are independent of the instruction stream: interrupts. (b) is an invalid memory access and is the result of a particular instruction: a trap (exception). (c) is a system call the program makes deliberately: a trap. (e) arises because the page accessed is not in memory and is tied to the instruction that made the access: a trap (exception). When the program is rerun with the same input, (b) and (c) occur again at the same instructions; (e) also occurs at an instruction of the same kind, but only if the page is not in memory at that moment. The timing of (a) and (d), however, is different every time.

1.7 Kinds of operating systems

The name "operating system" is used both for the software that runs a giant machine in a bank's data center and for the small program on a bank card's chip. What these systems have in common is that they take on the same two roles; what differs is which resource is scarce and what is considered most important.

1.7.1 Systems from large to small

Figure 1.27: Kinds of operating systems on a logarithmic axis of typical RAM size. From left to right: smart card and sensor node (kilobytes), embedded system (megabytes), handheld and personal systems (gigabytes), server and mainframe (hundreds of gigabytes to terabytes).

Figure 1.27 keeps its axis logarithmic because the memory sizes differ by a factor of a billion. The list below goes from the largest to the smallest.

- **Mainframe**: designed for very high I/O capacity, reliability, and uninterrupted availability. It provides batch processing, thousands of small transactions per second (for example a money transfer or an airline reservation), and time-sharing services. Banks and airlines depend on these systems. Examples: IBM z/OS, z/VM, Linux on Z, Unisys MCP, Fujitsu BS2000.
- **Server**: provides web, database, file, and email services to many users over the network. Data centers, the cloud, virtualization (KVM, Hyper-V), and containers belong to this world. Examples: Linux, Windows Server, FreeBSD.
- **Multiprocessor**: uses several cores or several processors in sockets and runs jobs in parallel. It deals with synchronization and with the NUMA layout, in which some regions of memory are closer to some processors. Today all general-purpose operating systems are in this class: Linux, Windows, macOS.
- **Personal**: offers a single user a good GUI and multitasking; used for browsing, office work, games, and file management. Examples: Windows 11, macOS, Linux distributions, ChromeOS.
- **Handheld**: the touch interface, battery and thermal management, sensors, the app store, and the permission model stand out. Examples: Android, iOS/iPadOS, HarmonyOS. Windows Phone ended in 2017.
- **Embedded**: runs inside cars, household appliances, network devices, and aircraft. The user generally does not install software; the system is fixed in flash memory. Examples: embedded Linux (Yocto), Zephyr, FreeRTOS, QNX, VxWorks.
- **Sensor node**: designed to run for years on a battery with a few tens of KB of RAM. It is event-driven: it sleeps most of the time and wakes when a measurement or a message arrives. It communicates over low-power radios (IEEE 802.15.4, BLE); it is used in IoT and machine-to-machine (M2M) networks. Examples: Contiki-NG, RIOT, Zephyr. TinyOS is now a historical example.
- **Smart card**: its processor and memory are extremely limited. It does authentication, payment, cryptography, and secure storage. Bank cards, SIMs and eSIMs, and ID cards are in this class. Examples: Java Card, MULTOS.

Between the two ends of the list the priorities reverse. In a mainframe or a server the scarce resource is the processing power and I/O capacity to serve the requests of thousands of users at the same time. In a sensor node the scarce resource is energy; the operating system's most important job is to keep the processor and the radio off for as long as possible.

1.7.2 Real-time operating systems

In some systems it is not enough for a computation to be correct; it must also finish on time. A vehicle's brake control software must notice within a few milliseconds that a wheel has locked and adjust the brake pressure. A response that is correct but late is as dangerous as a wrong one.

Definition 1.17

A **real-time operating system** (RTOS) is an operating system that tries to guarantee that jobs finish before a particular **deadline**. What matters is not speed but *predictability*: even in the worst case, the response time must not exceed a known bound.

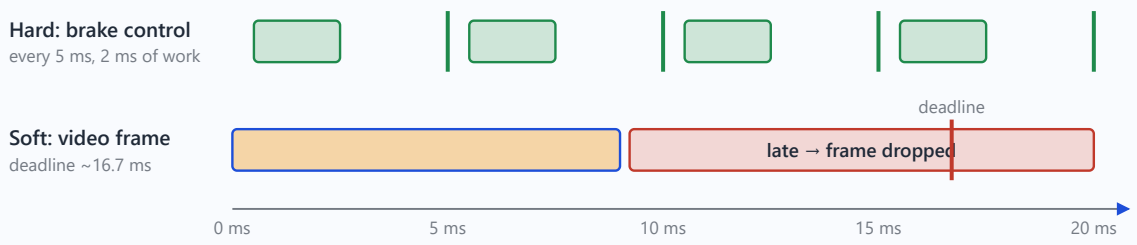


Figure 1.28: Hard and soft real-time tasks between 0 and 20 ms. Top, brake control: 2 ms of work every 5 ms, each job finishing before its own deadline (vertical lines). Bottom, a video frame: the deadline is about 16.7 ms; the second frame is late and has been dropped.

Real-time systems fall into two groups according to the consequence of missing a deadline:

- **Hard** real time: missing a deadline is a catastrophe. Brakes, airbags, flight control, and industrial robots are in this class. The brake task in Figure 1.28 arrives every 5 ms and finishes its 2 ms of work before its deadline every time.
- **Soft** real time: missing a deadline now and then is acceptable; only the quality drops. Audio and video streaming and games are in this class. For a video showing 60 frames per second, each frame must be ready within $1000 / 60 \approx 16.7$ ms. The second frame in the figure exceeded this time and was dropped; the viewer sees a brief stutter, but nobody is harmed.

Well-known RTOS examples include VxWorks, used in the Mars rovers, QNX, common in vehicle systems, and FreeRTOS and Zephyr for small microcontrollers. Linux could also run in real time with the PREEMPT_RT patches that were long kept separate. These patches were merged into the mainline

kernel in 2024 with Linux 6.12; the old RTLinux project has ended. Real-time systems are used in applications that process audio, video, motion, and sensor data, in train control, and in defense systems.

Common mistake

Understanding “real time” as “very fast”. System A's average response time is 1 ms, but now and then it rises to 500 ms. System B's average is 5 ms and it never exceeds 8 ms. On average A is faster. Still, B is suitable for a hard real-time job, because its worst case is known and bounded.

1.8 Key concepts

1.8.1 Policy and mechanism

One of the most useful principles of operating system design is to separate two different questions: *what will be done* and *how it will be done*.

Definition 1.18

Policy is the decision about what will be done. **Mechanism** is the infrastructure that makes it possible to carry out that decision.

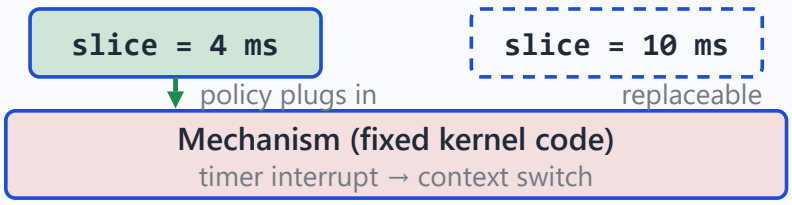


Figure 1.29: A policy is a replaceable parameter plugged into the mechanism. The mechanism (fixed kernel code): the timer interrupt and the context switch. The plugged-in policy is “slice = 4 ms”; “slice = 10 ms” can be plugged in instead.

Separating the two provides the greatest flexibility: when policy decisions change later, the mechanism need not be rewritten. The timer interrupt and the context switch are the mechanism that makes it possible to rotate the processor among processes. Whether the time slice will be 4 ms or 10 ms is a policy. If the slice value is designed as a parameter plugged into the mechanism, as in Figure 1.29, the same kernel code can run with a short slice on a desktop and a long slice on a server.

On Linux, a process's scheduling class can be changed with the `sched_setscheduler()` call and a disk's I/O scheduler by writing to a file under `/sys`, while the system is running; the kernel code does not change.

Mechanism	Policy
Timer interrupt	How many ms should the time slice be?
Priority scheduler	Which process has high priority?
File permission bits (rwx)	Who is given which permission?
Page replacement infrastructure	Which page is to be evicted?
cgroups	How much CPU and memory for which container?

Table 1.9: Examples of mechanism and policy.

In each row of the table, the column on the left is a "tool" and the column on the right is a "decision" about how to use that tool. The file permission bits form the mechanism that checks permission on every access; which user is given read and which write permission is the administrator's policy. An easy way to tell whether a distinction is a policy or a mechanism is to ask: "If this decision changes, does the code have to change?" If it does not, the decision is a policy. Operating system design takes creativity as much as engineering to strike these balances.

1.8.2 MS-DOS and FreeBSD

How a program is started says a lot about the structure of an operating system. Single-tasking MS-DOS and FreeBSD, a multitasking UNIX derivative, make a good contrast in this respect.

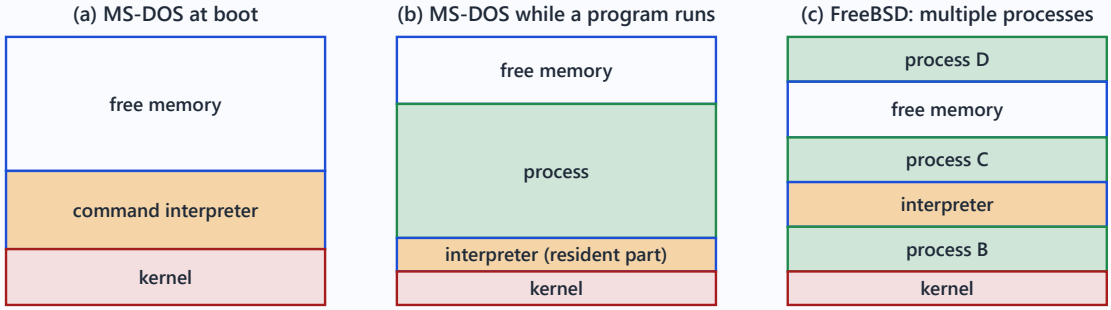


Figure 1.30: Memory layouts. (a) MS-DOS at boot: the kernel at the bottom, the command interpreter above it, and free memory at the top. (b) MS-DOS while a program runs: the process has been loaded over the transient part of the interpreter, and only the resident part remains. (c) FreeBSD: above the kernel, processes B, C, and D and the interpreter are in memory at the same time.

MS-DOS is a single-tasking system and uses a single memory area. At boot the shell, that is, COMMAND.COM, is loaded into memory. When the user runs a program, the program is loaded by overwriting the transient part of the shell; only a small resident part of the shell stays in memory.

When the program ends, the shell reloads itself from disk. Only one program runs at a time; there is no multitasking and no memory protection.

FreeBSD is multitasking. When a user logs in, a shell starts. When the user types a command, the shell first creates a new process that is a copy of itself with the `fork()` system call. This new process loads the requested program into its own memory with `exec()`. The shell either waits for the program to finish or, if `&` has been placed at the end of the command, runs the program in the background and goes on taking new commands. As in layout (c) of Figure 1.30, many processes are in memory at the same time, and each is protected in its own region.

1.8.3 Building an executable file

A C source file becomes a running process through the successive work of several separate programs. The GCC compiler driver lets you have each of these steps done separately:

```
$ gcc -E code.c -o code.i
$ gcc -S code.i -o code.s
$ gcc -c code.s -o code.o
$ gcc code.o other.o -o prog
$ ./prog
```

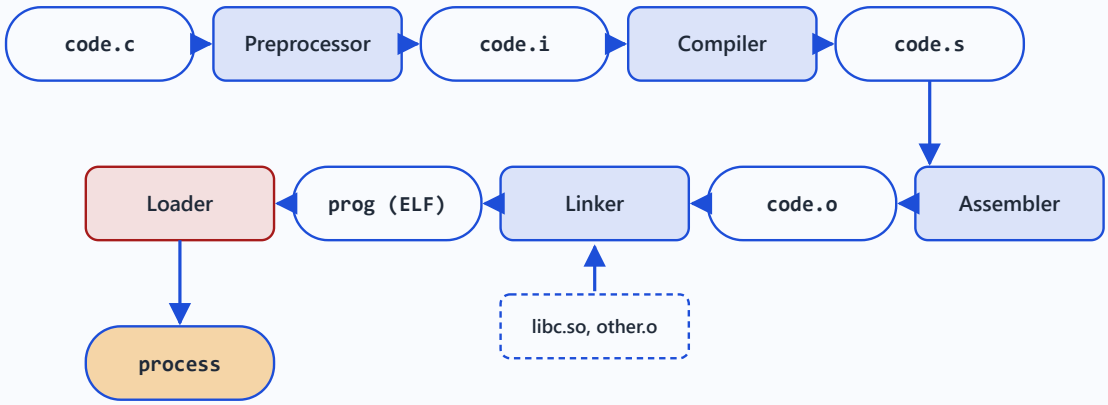


Figure 1.31: From C source to a running process. `code.c` goes through the preprocessor to become `code.i`, through the compiler to become `code.s`, and through the assembler to become `code.o`. The linker combines `code.o` with `libc.so` and `other.o` and produces the `prog` (ELF) file. The loader creates a process from this file.

1. **Source file**: `code.c` is C code that a human reads and writes.
2. **Preprocessor** (`gcc -E`): inserts the header files named in `#include` lines into the text, expands macros, and resolves conditional-compilation blocks with `#ifdef`. The result is still C code: `code.i`.

3. **Compiler** (`gcc -S`): translates the C code into the assembly language of the target processor and optimizes it: `code.s` .
4. **Assembler** (`gcc -c`): translates the assembly into machine code and produces an *object file*: `code.o` . In this file the addresses of external functions such as `printf` are not yet known; there is only a note saying "`printf` is called here".
5. **Linker** (`gcc code.o other.o -o prog`): combines the object files and libraries, resolves the open addresses, and produces the **executable file**. On Linux the format of this file is ELF, on Windows it is PE (.exe).
6. **Loading** (`./prog`): the shell does `fork` and `exec` . The kernel's loader maps the ELF file into memory, the dynamic linker attaches shared libraries such as `libc.so` , and the program starts running as a process.

In this chain the operating system's share is only in the last step. The first four steps are done by ordinary programs running in user mode: the preprocessor, the compiler, the assembler, and the linker. As noted in 1.2.4, it is the compiler that translates the program into machine language. The operating system turns the resulting file into a process: it allocates memory, maps the code and data into this memory, sets up the stack, and points the processor at the program's first instruction.

> **Common mistake**
>
> Thinking that a program and a process are the same thing. A program is a passive file sitting on disk (`prog`). A process is a running instance of that program: it has its own memory, registers, and open files. Three processes can be started at the same time from the same `prog` file; the `print` experiment in 1.3.9 does exactly this.

1.8.4 Metric units

Operating system topics involve both very small times (nanoseconds) and very large sizes (terabytes). Table 1.10 gives the decimal prefixes and Table 1.11 the binary prefixes.

Exponent	Name	Symbol	Exponent	Name	Symbol
10^{-3}	milli	m	10^{3}	kilo	k
10^{-6}	micro	µ	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	pico	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	exa	E

Table 1.10: Decimal (SI) prefixes.

Unit	Value
1 KiB (kibibyte)	2^{10} = 1024 B
1 MiB (mebibyte)	2^{20} B
1 GiB (gibibyte)	2^{30} B
1 TiB (tebibyte)	2^{40} B

Table 1.11: Binary (IEC) prefixes.

Using the two families of prefixes together often causes confusion. Memory is usually measured in binary: “16 GB RAM” is really 16 GiB, that is, $16 \times 2^{30} = 17{,}179{,}869{,}184$ bytes. Disk and network are measured in decimal: $1\ TB = 10^{12}$ B and $1\ Gb/s = 10^{9}$ bit/s. A capital B is used for byte and a small b for bit; a 1 Gb/s link carries at most $10^{9} / 8 = 125$ million bytes per second.

Example 1.8

How many GiB does an SSD labeled “512 GB” show up as in an operating system that displays sizes with binary prefixes? How many TiB is 1 TB?

Solution. The label is decimal: $512\ GB = 512 \times 10^{9}$ B. One GiB is $2^{30} = 1{,}073{,}741{,}824$ bytes. The quotient is $512 \times 10^{9} / 2^{30} \approx 476.8$ GiB. In the same way, $1\ TB = 10^{12} / 2^{40} \approx 0.91$ TiB. Nothing is missing from the disk; the same number of bytes has been shown in two different units. The difference grows as the unit grows: the 2.4% difference at the kilo level rises to about 9% at the tera level.

Figure 1.32: Typical latencies on a logarithmic time axis; each tick is a factor of 1000. From left to right: an L1 hit (about 1 ns), a DRAM access, a context switch (microseconds), an NVMe read, an HDD access (milliseconds), and a blink of an eye (tens to hundreds of milliseconds).

Figure 1.32 shows at a glance the scale of the times that appear in these notes. Between an L1 cache hit and a hard disk access there is a factor of about ten million. Human perception lies beyond all of them: in the time of a blink, the processor executes hundreds of millions of instructions. It is thanks to this difference in scale that an operating system can give a user the impression of running many programs “at the same time”.

1.9 Summary

- The operating system is the layer of software that manages the hardware and offers applications a simple, safe, and common interface. In the narrow sense, the operating system is the kernel running in privileged mode; the shell and the GUI are ordinary user programs.
- Protection rests on the mode bit in the processor. In user mode privileged instructions are forbidden; the kernel is entered only through the doors of system call, interrupt, and exception, at predetermined entry points.
- In a `read()` call, the libc wrapper puts the number and the arguments into registers, `syscall` raises a trap, the kernel finds `sys_read()` in the table and checks the arguments, and `sysret` returns to user mode.
- Viewed from above, the operating system is an extended machine: it turns ugly hardware into abstractions such as the process, the address space, the file, and the socket. Viewed from below, it is a resource manager: it shares resources in time or in space and balances throughput and fairness.
- Spooling gives a slow, unshareable device to a single service process and overlaps I/O with computation. In a circular queue, the equality of `in` and `out` does not tell empty from full; that is why the `count` counter is kept.
- Batch processing removed the manual transition between jobs. Multiprogramming raised CPU utilization by giving the CPU to another job while one waited for I/O; in the example two jobs finished in 9 units instead of 15 and utilization rose from 53% to 89%. Time sharing gave every user a short response time through the timer interrupt.
- In a context switch all registers are saved, because the processor has a single set of registers. Multicore chips bring problems of parallelism, synchronization, and cache coherence; SMT makes each core look like two logical CPUs.
- Thanks to locality, the memory hierarchy works with almost the speed of the upper level and the capacity of the lower level. Because a disk wait lasts millions of clock cycles, the operating system runs another process during that time.
- In polling the CPU busy-waits; with interrupts one interrupt arrives for every unit of work; with DMA the controller moves the whole block and raises a single interrupt. An interrupt arises from outside and asynchronously, a trap from the instruction itself and synchronously; both reach the kernel by the same path.
- Operating systems are very diverse, from mainframes to smart cards. In real-time systems what matters is not speed but predictability; in a hard system, missing a deadline is a catastrophe, and in a soft system it lowers quality.
- Policy decides what will be done and mechanism how it will be done; if the two are separated, policy can be changed without changing the kernel code.

- The preprocessor, compiler, assembler, and linker turn C source into an executable file; the operating system loads this file and turns it into a process. Memory is measured in binary (GiB), disk and network in decimal (GB, Gb/s).

1.10 Key terms

operating system (OS) kernel shell graphical user interface system call kernel mode user mode dual-mode operation trap exception interrupt interrupt handler abstraction address space virtualization virtual address concurrency parallelism race condition spooling batch system multiprogramming time sharing scheduler throughput response time register program counter (PC) fetch–decode–execute cycle context switch multicore simultaneous multithreading cache coherence memory hierarchy locality memory management unit bus device controller device driver polling busy waiting direct memory access real-time operating system (RTOS) policy / mechanism executable file hypervisor container

---

1.11 Test Yourself

1. Which of the following events cause the processor to move from user mode to kernel mode? (a) a `getpid()` call, (b) `i++`, (c) a page fault, (d) a packet arriving at the network card, (e) `memcpy` in user memory.

   Answer

   (a), (c), and (d) do. `getpid()` is a system call; the libc wrapper raises a trap with `syscall`. A page fault is an exception arising from the instruction that accessed memory. The arrival of a packet is a hardware interrupt from the network card. `i++` is an ordinary operation on registers and memory, and `memcpy` is a library function that copies within user memory; both stay in user mode.

2. In a single-CPU system, job A uses 1 unit of CPU, 3 units of I/O, and 1 unit of CPU; job B uses 2 units of CPU, 1 unit of I/O, and 2 units of CPU. When do the two jobs finish in a single-programmed and in a multiprogrammed system, and what is the CPU utilization? (In multiprogramming, when

the CPU becomes free, A is chosen first among the ready jobs.)

Answer

Single-programmed: A 0–1, idle 1–4, A 4–5; B 5–7, idle 7–8, B 8–10. Finish *t = 10*, CPU utilization *6 / 10 = 60%*. Multiprogrammed: A 0–1; when A goes to I/O, B runs 1–3; B goes to I/O (3–4), and since A's I/O also ends at 4, the CPU is idle during 3–4; at *t = 4* both jobs are ready, A runs 4–5, B 5–7. Finish *t = 7*, CPU utilization *6 / 7 ≈ 85.7%*.

3. In the spooling code, if a program calls `enqueue("e.jpg")` while the queue is full (a.pdf, b.txt, c.doc, d.png), what happens? If `spooler()` then runs, which files are printed, and in what order?

Answer

`enqueue()` does not check for fullness. Because `in = 0`, e.jpg is written into slot 0, over a.pdf; then `in = 1` and `count = 5`. The service runs its loop 5 times: `out` takes the values 0, 1, 2, 3, 0. In order, e.jpg, b.txt, c.doc, and d.png are printed; after each is sent, its slot is set to `NULL`. In the fifth round `out` is 0 again and this slot is now empty: `NULL` is sent to the printer. a.pdf is lost, and the counter has counted a job that does not exist. A correct `enqueue()` should reject the job or wait when `count == 4`.

4. On a core running at 4 GHz, how many clock cycles does a 60 ns DRAM access and a 50 µs NVMe read take? Why does the operating system behave differently for these two waits?

Answer

One cycle is *0.25 ns*; 4 cycles pass in 1 ns. DRAM: *60 × 4 = 240* cycles. NVMe: *50 µs = 50,000 ns*, *50,000 × 4 = 200,000* cycles. The DRAM wait is shorter than a context switch and is hidden by the caches; the operating system does not step in. For the 200,000-cycle SSD wait, however, the process is kept waiting and the CPU is given to another process.

5. 16 KiB of data is received from a device. In interrupt-driven I/O, one interrupt occurs for every 16 bytes and each interrupt costs the CPU 2 µs. With DMA, programming takes 5 µs and the single interrupt at the end takes 2 µs. How much time does the CPU spend in the two methods?

Answer

16 KiB = 16 × 1024 = 16,384 bytes. The number of interrupts is *16,384 / 16 = 1024*; the CPU time is *1024 × 2 = 2048 µs*. DMA: *5 + 2 = 7 µs*. The ratio is about *2048 / 7 ≈ 293*.

6. A processor has 4 cores with SMT and 4 cores without SMT. How many logical CPUs does the

operating system see? Are all of these CPUs equally powerful?

Answer

$4 \times 2 + 4 = 12$ logical CPUs are visible. They are not equally powerful: two logical CPUs that share the same core share that core's execution units and cache, so they do not run as fast as two separate cores.

7. About how many GiB does a USB drive labeled “256 GB” show up as in an operating system that uses binary prefixes?

Answer

$256 \times 10^9 / 2^{30} \approx 238.4$ GiB. The label is decimal and the operating system's display is binary; the number of bytes is the same.

8. Which of the following are mechanisms and which are policies? (a) The context switch code, (b) giving interactive processes a shorter time slice, (c) the hardware checking the valid bit in a page table entry, (d) evicting the least recently used page when memory is full.

Answer

(a) and (c) are mechanisms: they make it possible to carry out a job. (b) and (d) are policies: they are decisions answering the questions “how large a slice for whom?” and “which page is evicted?”. The same mechanisms can be used with other policies as well.

9. A pacemaker, a music streaming application, and a web server: which is hard real time, which is soft real time, and which is general purpose?

Answer

The pacemaker is hard real time: failing to deliver the pulse on time has vital consequences. Music streaming is soft real time: an occasional delay causes a brief interruption but is not dangerous. The web server is general purpose; it emphasizes average performance and fairness and has no hard deadline.

10. What hardware feature separates time sharing from multiprogramming? How does a job that does no I/O at all behave in the two systems?

Answer

The timer interrupt. In multiprogramming the CPU is taken from a job only when the job starts waiting; a job that does no I/O holds the CPU until it finishes. In time sharing the timer raises an interrupt at the end of each slice, and the kernel takes the CPU from this job and gives it to the next one; in this way the other users' response time stays short.

1.12 Further reading

- A. S. Tanenbaum and H. Bos, *Modern Operating Systems* (5th ed., 2023), Chapter 1: presents the two roles of the operating system, the generations, the IBM 1401/7094 batch arrangement, and the kinds of OS in the same order as these notes, with plenty of historical detail.
- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Chapter 2: gives the C code of the virtualization, concurrency, and persistence experiments; the book can be read online for free.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th ed., 2018), Chapter 1: covers interrupts, the memory hierarchy, DMA, and dual-mode operation in detail from the hardware side.
- F. P. Brooks, *The Mythical Man-Month* (1975; 20th anniversary edition, 1995): a classic book of essays on the software engineering lessons drawn from the OS/360 project.

1.13 References

1. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, version 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
2. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary edition, Addison-Wesley, 1995.
3. Linux manual pages: syscall(2), fork(2), execve(2), proc(5).
4. A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018.
5. A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014; 5th ed., 2023.

PART I · CHAPTER 2

Operating-System Structures

An operating system looks like a single program. Inside it, however, are dozens of components such as process management, memory management, the file system, device drivers, and the network stack. In which mode these components run and how they talk to each other determine the system's speed, reliability, and maintainability. These notes begin with the services the operating system offers to users and programs. They then introduce the basic concepts of process, address space, file, mounting, pipe, and deadlock. How the hardware protects the operating system, how programs request services from the kernel through system calls, and the basic POSIX calls are explained step by step with code traces. The last sections compare monolithic, layered, microkernel, modular, hybrid, and exokernel structures, and touch on virtual machines, the linker and loader, and the boot process.

LEARNING OUTCOMES

Services and concepts
Explain the services an operating system provides and the concepts of process, address space, file, mounting, and pipeline.

System calls
Trace step by step the transition from user mode to kernel mode, parameter passing, and the basic POSIX calls.

Kernel structures
Compare the monolithic, layered, microkernel, modular, hybrid, and exokernel structures with their strengths and weaknesses.

Virtualization and booting
Recognize the concepts of virtual machine and container, the work of the linker and loader, and the steps of booting.

2.1 Structure and services

An operating system can be studied from two angles. Someone looking from the outside asks what the system does: which services does it offer, and how do programs reach them? Someone looking from the inside asks how the system is built: where do the components run, and how do they call one another? This section takes up both questions briefly. It first introduces what is meant by “structure” and then the services the operating system offers to the outside.

2.1.1 Why does structure matter?

Today's operating system is not a small program by itself. The process manager that creates and terminates processes, the memory manager that shares out memory, and the file system that presents data on disk as files are only three of its parts. Each device has its own driver, and network packets have a stack of their own. The source code of the Linux kernel passed 40 million lines in 2025. Writing, understanding, and maintaining software of this size as a single disorganized heap is impossible.

The designer makes two decisions for each component. The first is **where** the component will run: in kernel mode, which uses all the privileges of the hardware, or in user mode, with restricted privileges? The second is **how the components will communicate**: will one call the other like an ordinary function, or will the two components send each other messages?

> **Definition 2.1**
>
> The general strategy for where the components of an operating system run (kernel mode or user mode) and how they communicate (function call or message) is called the **operating system structure**.

The choice of structure directly affects the system's everyday life. How well complexity is managed and how many files a change touches depend on this choice. So do how many lines of code a bug must be searched for in and how easily the system can be ported to another processor. However, no structure is best by every criterion. Figure 2.1 places three main goals at the corners of a triangle: performance, reliability and isolation, and flexibility and maintainability. A design that approaches one corner usually moves away from the others.

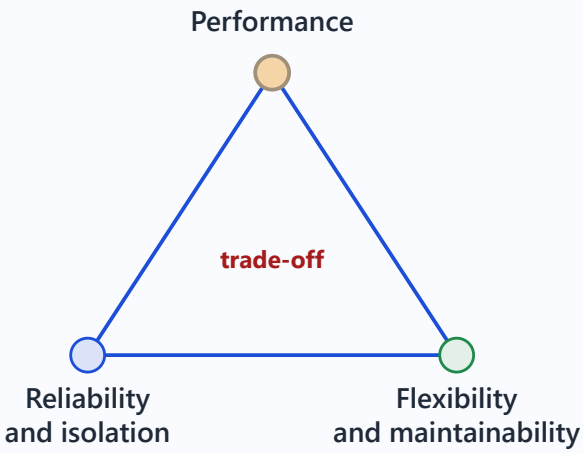


Figure 2.1: The trade-off in the choice of structure. Gathering components in the same mode and calling them directly increases performance; splitting them into separate processes increases reliability and isolation; open interfaces and replaceable parts make flexibility and maintenance easier.

An example makes this trade-off concrete. If file system code runs inside the kernel together with the other components, reading a file costs no more than a few function calls. If the same code runs in user mode as a separate process, every request needs a message and several mode switches; reading becomes more expensive. In return, a bug in the file system code crashes only that process in the second case, while in the first it can bring down the whole kernel. A desktop computer, a data center server, an embedded device, and an aircraft's flight control computer strike this balance at different points. This is why there is no single correct structure.

Common mistake

Confusing the structure with the interface the user sees. Two operating systems can offer the same system calls, the same commands, and the same disk format and still have very different structures. Structure concerns the division of labor inside the system; a user looking from outside often does not see the difference.

2.1.2 Operating system services

The operating system prepares a working environment for the programs running on it and offers a set of **services** that are part of this environment. The details of the services vary from system to system, but the main groups are the same in almost every system. Figure 2.2 shows these groups as layers.

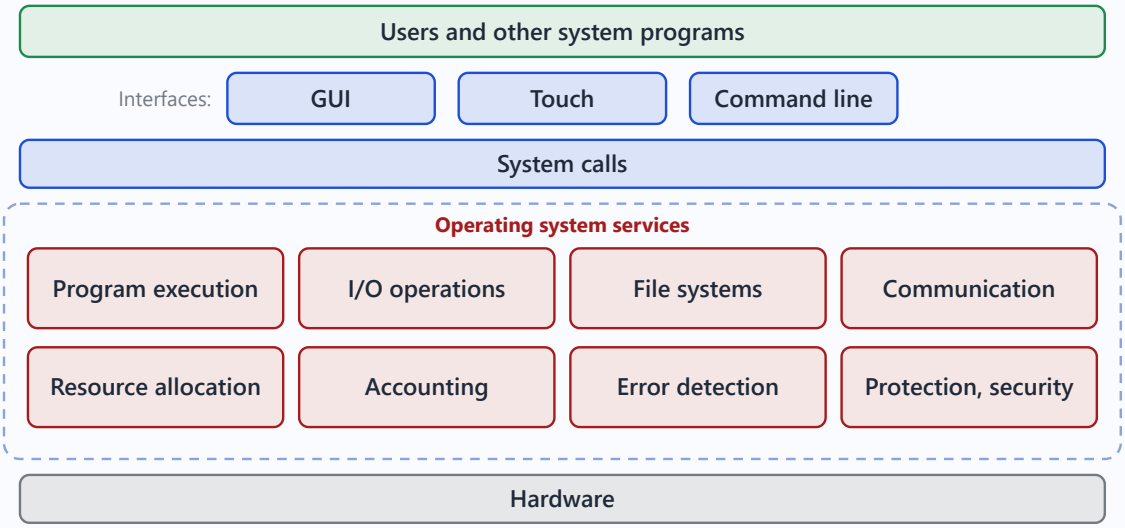


Figure 2.2: A layered view of operating system services (adapted from Silberschatz, Figure 2.1). At the top are users and system programs; below them are the graphical, touch, and command-line interfaces. Interfaces and programs reach the services through system calls. Below the services is the hardware.

The most important detail in the figure is the transition point between the layers. Whether the user clicks a window or types a command at the command line, what they do ultimately turns into one or more **system calls**. The interface is an intermediary that translates the user's request into system calls. The services themselves are below the system call layer, inside the operating system, and manage the hardware directly.

2.1.3 Whom do the services serve?

It is customary to divide the services into two groups according to the party that benefits from them. The first group helps the user and programs directly:

- **Interface.** The user talks to the system through the command line, a graphical user interface (GUI), a touch screen, or voice commands.
- **Program execution.** The system loads a program into memory, runs it, and terminates it. It also records whether the program ended normally or with an error.
- **Input/output (I/O) operations.** A program does not access a device directly; it passes its request to the operating system. In this way both efficiency and protection are achieved.
- **File system.** Creating, deleting, reading, writing, and searching files and managing permissions fall within the scope of this service.
- **Communication.** Processes exchange information on the same machine or over the network. The two basic ways are shared memory and message passing.
- **Error detection.** Situations such as a memory error, a device failure, division by zero, or unauthorized memory access are detected and handled appropriately.

The second group is not directly visible to the user; it makes the system work efficiently and safely as a whole:

- **Resource allocation.** The processor (CPU), memory, and storage are shared among many processes.
- **Accounting.** Which user or process uses how many resources is recorded; logs are kept.
- **Protection.** Processes cannot harm each other or the kernel.
- **Security.** Authentication, authorization, and defense against outside attacks are the job of this service.

Most users know the operating system not through system calls but through **system programs**. `ls`, which lists directory contents, `cp`, which copies files, `ps`, which shows running processes, the package manager, the compiler, and the file manager are such programs. They are ordinary user programs, but they make heavy use of system calls to do their work. In the user's eyes the "operating system" is often the sum of these programs.

Example 2.1

Is recalculating a formula in a spreadsheet program an operating system service?

Solution. It is not. Computing the formula is the program's own logic; the processor does this computation by executing the program's own code. During this time the operating system provides the program only with processor time and memory. When the spreadsheet is saved to disk, however, things change: the program has no authority to access the disk directly, and it uses the file system service through system calls. One job of the same program is its own code; its other job is an operating system service.

2.2 Core concepts

Operating systems hide the complicated details of the hardware behind a few basic abstractions. These abstractions are so widespread that every text describing operating systems relies on the same concepts: process, address space, file, directory, mounting, pipe, and deadlock. The aim of this section is to give a first definition of these concepts. The details of each are deepened in the chapter devoted to that concept; what matters here is to see how the concepts relate to one another.

2.2.1 Process

A program file on disk does nothing by itself. It is a passive file made of machine instructions and initial data. When the program is run, the operating system loads it into memory, allocates a stack for it, points the processor at its first instruction, and starts watching the program. This living instance that is running is called a process.

Definition 2.2

A **process** is a program in execution. The program is the passive file on disk; the process is its active instance. With each process come an address space (code, data, stack), the values of the registers (including the program counter and the stack pointer), open files, alarms, a list of related processes, and other resources.

Several processes can be created from the same program at the same time. On the machine of a user running `bash` in two separate terminals, there are two processes from the single `bash` file on disk. Both execute the same code, but the variables, open files, and current instruction of each are different.

From time to time the operating system has to suspend one process and run another. For the suspended process to continue later where it left off, all of its current state must be stored somewhere. This information is kept in the **process table**. Each entry of the table belongs to one process and is called a **process control block** (PCB). Figure 2.3 shows a small process table and the PCB contents of the running process.

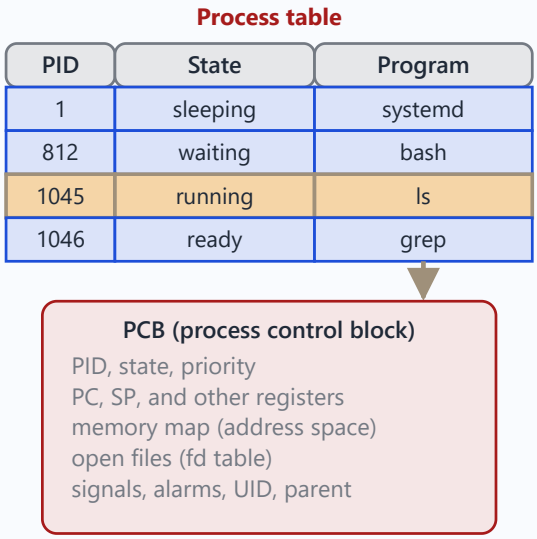


Figure 2.3: A process table and a process control block. The table has four processes: PID 1 (`systemd`, sleeping), 812 (`bash`, waiting), 1045 (`ls`, running), and 1046 (`grep`, ready). The PCB of the running process holds the PID, state, and priority; the program counter, stack pointer, and other registers; the memory map; the open file table; signals, alarms, the user ID, and parent information.

Each process in the figure is identified by a **process identifier** (PID). The state column shows what the process is doing at that moment. On a single-processor machine only one process runs at a time. The other processes either wait for their turn on the processor (ready) or wait for an event such as a

key being pressed on the keyboard. The register values in the PCB are saved when the process is taken off the processor and restored to the processor when the process is about to run again. The process does not notice the interruption in between.

Processes often do not work alone; they exchange data with each other. The common name for the mechanisms the operating system offers for this is **interprocess communication** (IPC). Pipes, shared memory, message queues, and sockets are some of these mechanisms.

2.2.2 The process tree

On UNIX-like systems a new process is created from an existing process by the `fork()` system call. The process that makes the call is called the **parent** and the process that results is called the **child**. Because a child can create children of its own, processes arrange themselves naturally in the shape of a tree. Figure 2.4 shows two stages of this tree.

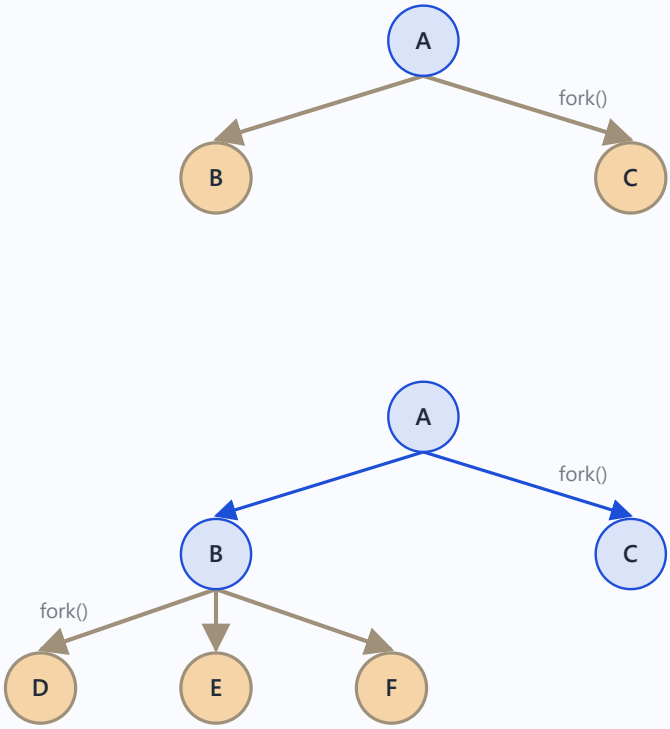


Figure 2.4: The formation of a process tree. Top: process A has called `fork()` twice; B and C are A's children. Bottom: process B has also created three children, D, E, and F.

At the beginning there is only process A. When A calls `fork()` twice, B and C are born; A is the parent of both. Then B calls `fork()` three times and processes D, E, and F are created. In this tree D's parent is B, and B's parent is A. On Linux the root of the whole tree is the `systemd` process (`init` on older systems), which the kernel starts at boot and which has PID 1. On a running system the `pstree -p` command prints this tree together with the process identifiers.

Common mistake

Thinking that `fork()` starts a new, empty program. `fork()` creates a copy of the parent; the child continues executing the same code as the parent from the same point. For the child to run a different program, an `execve` call is needed as well. This pair is traced in detail with the shell example in 2.5.2.

2.2.3 Address space and memory segments

Each process sees memory as if it were using it alone. The set of addresses a process can use is called its **address space**. The address space starts at 0 and extends to an upper limit. On Linux on the x86-64 architecture, a process's user space is 128 TiB; this is much larger than the physical memory of most machines. The technique that makes address spaces larger than physical memory possible is **virtual memory**.

Many processes are in memory at the same time. The operating system must protect these processes both from each other and from itself: a faulty pointer in one process must not corrupt another process's data or the kernel's code. This protection is provided by the memory management unit in the hardware and the operating system working together.

A process's address space is divided into several segments. Figure 2.5 shows the typical layout from bottom to top.

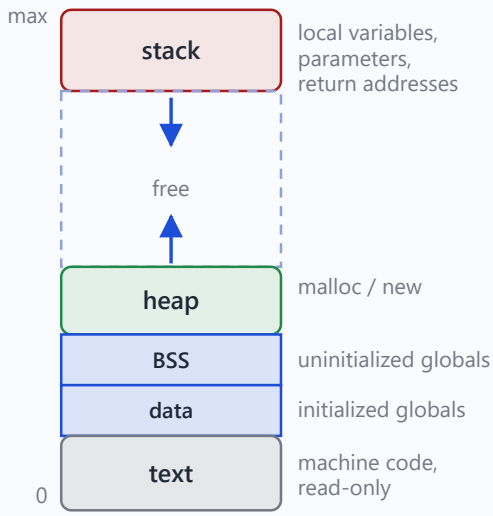


Figure 2.5: The address space of a process. At the very bottom (near address 0) is the text segment; above it are the initialized data and uninitialized data (BSS) segments, and then the heap. At the very top is the stack. The heap grows upward and the stack downward; the gap between them leaves room for both to grow.

- **Text.** The machine code of the program. It is kept read-only; processes that run the same program, and all the threads of a process, share this segment.

- **Data and BSS.** Global and `static` variables live here. Those whose initial value is given in the source code go to the data segment, and those without one go to the BSS segment.
- **Heap.** Dynamic memory allocated at run time with `malloc` or `new`. Like the data segment, the heap is shared by all threads of the process.
- **Stack.** Function parameters, local variables, and return addresses are kept here. Each thread has its own stack.

Example 2.2

In a C program, `int counter = 5;` and `int table[1000];` are defined as globals, and `int i;` and `char *p = malloc(64);` are defined inside `main`. In which segment is each object?

Solution. `counter` is a global variable with an initial value: it is in the data segment. `table` is a global array without an initial value: it is in the BSS segment and is filled with zeros at load time. `i` and the pointer `p` itself are local variables of `main`: they are on the stack. The 64-byte area that `p` points to is on the heap, because it was allocated with `malloc`. The machine code of the functions is in the text segment as well.

2.2.4 Files and the directory hierarchy

A disk by itself is just a storage area made of numbered blocks. If a program had to work with requests such as “read block number 47,291”, every programmer would have to learn the internal structure of the disk. The operating system hides this detail and offers programs a cleaner abstraction: the file.

Definition 2.3

A **file** is a named unit that stores data persistently. Text, images, video, or an executable program can be stored in a file. Files are gathered in **directories**; because directories can contain other directories, a tree results.

A file in the tree is reached by its **path name**. An absolute path starts at the root directory (`/`): `/home/ali/notes.txt`. A relative path starts at the process's current working directory. A process whose working directory is `/home` reaches the same file as `ali/notes.txt`. Figure 2.6 shows a sample directory tree.

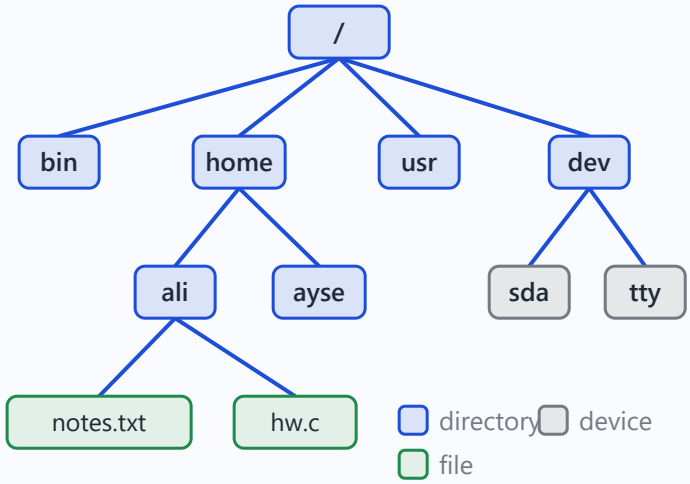


Figure 2.6: A sample file system tree. Under the root directory are the `bin`, `home`, `usr`, and `dev` directories. The `/home/ali` directory contains the files `notes.txt` and `hw.c`, and the `/dev` directory contains the device files `sda` and `tty`.

When a process opens a file, the operating system gives it a small integer. This number is called a **file descriptor** (fd). The process afterward refers to the file not by its name but by this number: "read 100 bytes from file number 3". Every process starts with three open descriptors: 0 is standard input (`stdin`), 1 is standard output (`stdout`), and 2 is standard error (`stderr`). When the process opens a new file, it is given the smallest free number; the first file opened therefore usually gets number 3.

A remarkable idea of UNIX is that devices also look like files. These are called **special files**, and there are two kinds. **Block devices** store data in fixed-size blocks and allow access to the blocks in any order: disks such as `/dev/sda` and `/dev/nvme0n1`. **Character devices** produce or consume data as a stream of characters: the terminal (`/dev/tty`) and the serial port, for example. A program uses the same `write` call whether it writes to a terminal or to a file.

2.2.5 Mounting a file system

When a USB drive is plugged into a computer, the file system on the drive is at first a separate tree. It has no connection to the system's main tree, and it cannot be reached by any path name. UNIX has a single directory tree; adding a new file system to this tree is called **mounting**.

Definition 2.4

Mounting is the operation of attaching the root directory of a file system to a directory in the existing tree. The directory where the attachment is made is called the **mount point**. After mounting, the files in the new file system are reached by path names that begin with the mount point.

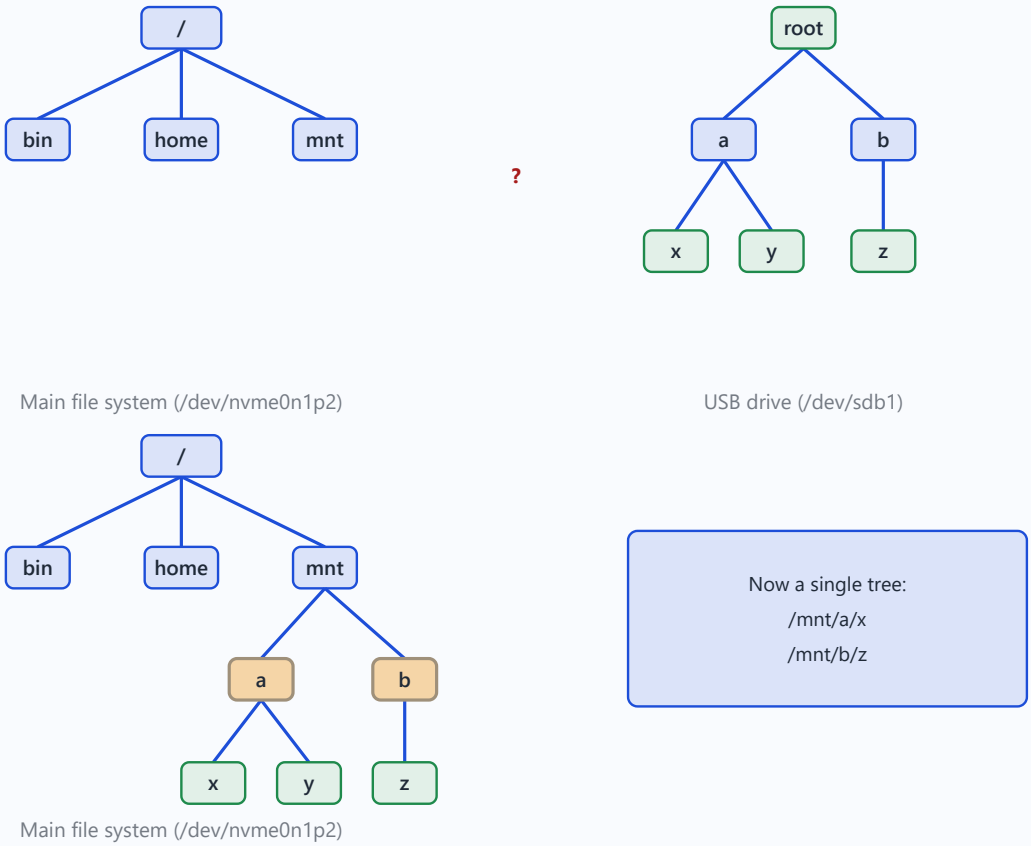


Figure 2.7: Before and after mounting. At the top are the main file system (`/dev/nvme0n1p2`) and the separate tree on the USB drive; the `a` and `b` directories on the USB drive cannot be reached by any path. At the bottom the root of the USB drive has been mounted on the `/mnt` directory; now the paths `/mnt/a/x` and `/mnt/b/z` are valid.

Figure 2.7 shows the operation at two moments. An empty directory is usually used as the mount point. If the directory is not empty, its old contents are not deleted; they merely become invisible until the mount is removed. Windows solves this question differently: each disk or partition usually gets a separate drive letter (C:, D:) and remains a separate tree.

On Linux, mounting is done with the `mount` command. The following command sequence shows the commonly used forms:

```
$ lsblk                            # list disks and partitions
$ sudo mount /dev/sdb1 /mnt        # mount sdb1 on /mnt
$ sudo mount -o ro /dev/sdb1 /mnt  # mount read-only
$ sudo mount -o remount,rw /mnt    # remount read-write
$ findmnt /mnt                     # show mount information
$ sudo umount /mnt                 # unmount
```

`lsblk` lists the disks and partitions in the system; from it one sees that the USB drive has been named `/dev/sdb1`. The second line mounts it on `/mnt` in read-write mode. The `-o ro` option mounts read-only; in recovery and forensic work this mode is preferred so as not to change the data on the disk by accident. `remount,rw` switches an already mounted file system to read-write mode. `findmnt` shows mount information, and `umount` removes the mount. Most of these commands need administrator privileges and are therefore run with `sudo`.

File systems to be mounted automatically at boot are defined in the `/etc/fstab` file or in `systemd` mount units. On desktop systems, a USB drive that is plugged in is mounted automatically under `/run/media/…` by the `udisks2` service. Figure 2.8 summarizes how, after mounting, the file `doc.pdf` on the USB drive becomes reachable by the path `/mnt/doc.pdf`.

Figure 2.8: The tree after the command `mount /dev/sdb1 /mnt`. The `/mnt` directory has been attached to the file system on the USB drive; `doc.pdf` at the root of the USB drive is now opened by the path `/mnt/doc.pdf`.

2.2.6 Pipes

When a command such as `ls -l | grep "notes"` is typed in the shell, two programs run at the same time: `ls` produces the directory listing, and `grep` selects from this listing the lines that contain “notes”. There is no temporary file between the two programs. The structure that carries data from one to the other is a pipe.

Definition 2.5

A **pipe** is a one-way byte channel, buffered in the kernel, that connects two processes. One process writes to the write end of the pipe, and the other process reads from the read end. Processes use the pipe like an ordinary file, with the `read` and `write` calls.

There are two kinds of pipes. An **anonymous pipe** is created by the `pipe()` call and has no name; it can therefore be used only among processes with a common ancestor, typically parent and child processes. A **named pipe** (FIFO) is created with `mkfifo` and has a name in the file system. Even processes unrelated to each other can use the same pipe by opening this name.

How the shell sets up the command above can be traced in four steps:

1. **The pipe is created.** The shell first calls `pipe(fd)`. The kernel allocates a buffer and returns two descriptors: `fd[0]` is the read end and `fd[1]` is the write end.

2. **Processes are created and the ends are redirected.** The shell calls `fork` twice. In the first child, standard output (1) is redirected to the write end of the pipe with the `dup2` call; in the second child, standard input (0) is redirected to the read end. Then the children replace themselves with the programs `ls` and `grep`.
3. **Data flows.** `ls` writes the listing to its standard output, that is, into the pipe. `grep` reads from its standard input, that is, from the pipe, and writes only the lines containing "notes" to the terminal. Neither program knows it is connected to a pipe.
4. **Flow control and termination.** When the buffer is full the writing process blocks, and when it is empty the reading process blocks. When `ls` finishes and all the write ends of the pipe are closed, the `read` call of `grep` returns 0. This value means end of file (EOF), and `grep` finishes too.

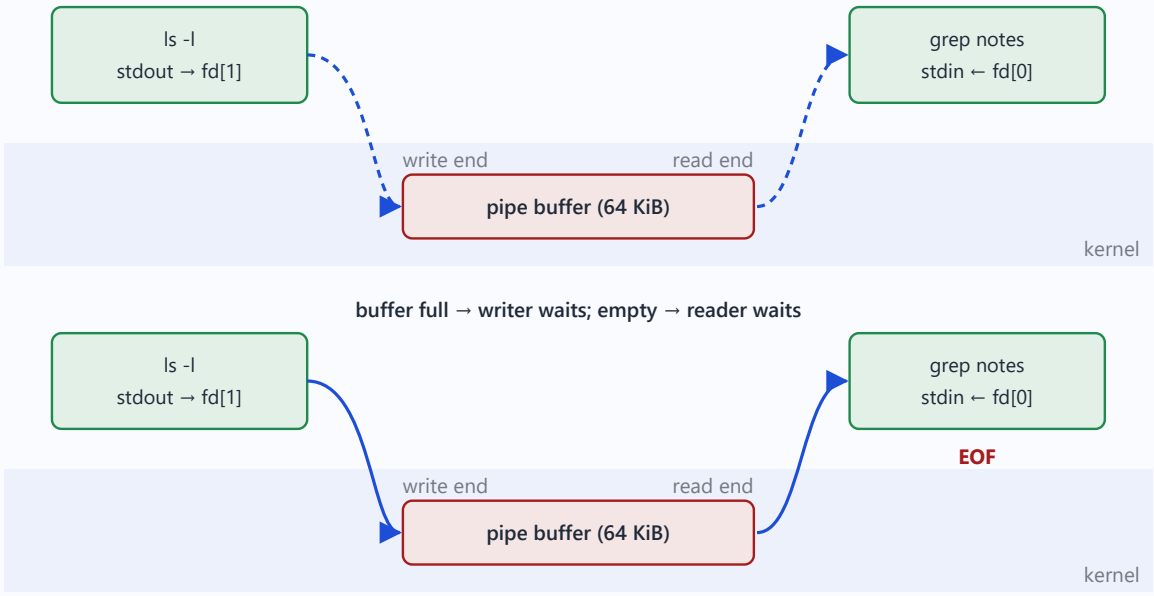


Figure 2.9: The pipeline `ls -l | grep "notes"`. Top: redirection of the ends: the standard output of `ls` is connected to `fd[1]` and the standard input of `grep` to `fd[0]`. Bottom: flow control: the pipe buffer (64 KiB) is in the kernel; when the buffer is full the writer waits, when it is empty the reader waits, and when the write end is closed the reader sees end of file.

Common mistake

The shell forgetting to close the pipe ends it holds itself. A reading process sees end of file only when *all* write ends of the pipe have been closed. If the parent shell keeps the write end in its own copy open after `fork`, `grep` waits forever for new data even after `ls` has finished.

2.2.7 Deadlock

Processes use not only the processor but also other resources such as printers, files, memory buffers, or locks. Some resources can be held by only one process at a time. If two or more processes start waiting for resources held by one another, none of them can make progress.

> **Definition 2.6**
>
> If every process in a set of processes is waiting for a resource held by another process in the set, and as a result none can make progress, there is a **deadlock** in the set.

Its counterpart in daily life is four cars that enter an intersection from four directions at the same time, each waiting for the car on its right to pass. None of the law-abiding drivers budges. The standard way to draw the situation is the **resource allocation graph**. Figure 2.10 shows an example with two processes and two resources in two stages.

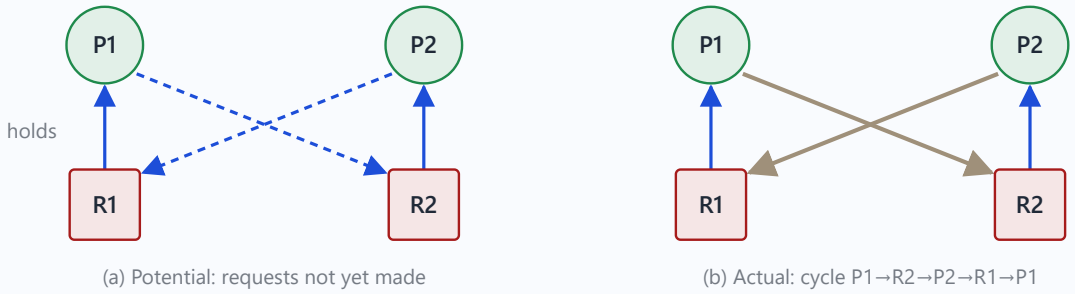


Figure 2.10: A resource allocation graph. An arrow from a square to a process shows that the resource is held by that process; an arrow from a process to a square shows that the process requests that resource. (a) Potential deadlock: P1 holds R1 and P2 holds R2; the dashed arrows show requests not yet made. (b) Actual deadlock: the requests have been made and the cycle P1 → R2 → P2 → R1 → P1 is closed.

In case (a) there is no problem yet: P1 holds R1 and P2 holds R2, and later each will request the other's resource. When both of these requests have been made, case (b) is reached. P1 waits for R2, held by P2; P2 waits for R1, held by P1. A closed cycle has formed in the graph and both processes wait forever. If each resource has a single instance, a cycle in the graph is a sure sign of deadlock. If resources have multiple instances, a cycle indicates the possibility of deadlock but is not proof by itself.

2.3 Protection

The kernel and dozens of user processes are in memory at the same time. One of the processes may be faulty or malicious. If such a process writes to the disk directly, it can corrupt the file system; if it disables interrupts, it can put the operating system out of action; if it enters an infinite loop, it may

never give up the processor. The operating system cannot defend itself against these dangers with software alone, because the problem code runs on the same processor. The foundation of protection is in the hardware.

2.3.1 Dual-mode operation

Today's processors recognize at least two modes of operation. Which mode the processor is in is kept in a **mode bit** in the processor. In the notation Silberschatz uses, this bit is 0 in kernel mode and 1 in user mode.

Definition 2.7

Dual-mode operation is the processor's distinction between **kernel mode** and **user mode**. **Privileged instructions** can be executed only in kernel mode. If such an instruction is attempted in user mode, the hardware raises a trap and control passes to the operating system.

What privileged instructions have in common is that, if used wrongly, they can harm other processes or the system as a whole. Accessing I/O devices directly, setting the timer, disabling interrupts, and changing page tables belong to this group. Jobs that harm no one, such as reading the system clock, can be done freely in user mode.

When a privileged instruction is attempted in user mode, the hardware does not execute the instruction; instead it raises a **trap**. The processor switches to kernel mode and jumps to a handler that the operating system set up beforehand. The operating system usually terminates the faulty process. There are only three legitimate ways into the kernel: a system call, a hardware interrupt, and an exception (such as division by zero or an invalid memory access). In all three the processor jumps not to an address chosen by the process but to a fixed entry point determined by the kernel.

In real processors there can be more than two modes. The x86 architecture defines four protection rings, numbered 0 to 3, but Linux and Windows use only ring 0 (kernel) and ring 3 (user). The ARM architecture has four privilege levels, EL0 to EL3; of these, EL2 is reserved for virtualization.

Common mistake

Thinking that dual mode alone is enough. The mode distinction prevents a process from executing privileged instructions, but it does not solve two problems. A process can enter an infinite loop and hold the processor without using any privileged instruction; the remedy for this is the timer. A process can try to reach another process's memory with an ordinary load instruction; the remedy for that is the memory management unit (MMU).

2.3.2 The timer

If a process enters a loop such as `while (1) ;`, it can hold the processor forever without executing any privileged instruction. Because the operating system itself needs the processor to run, it cannot notice this situation on its own. The solution is a piece of hardware that interrupts the processor after a certain time.

Definition 2.8

A **timer** is a hardware counter that raises an interrupt after a certain time. The operating system loads a value into the counter; on every tick of the physical clock the counter decreases by one; when the counter reaches zero an interrupt is raised and control passes to the operating system.

Loading a value into the counter is a privileged instruction. In this way a user program cannot stop the timer or extend its period. When the timer interrupt arrives, the operating system asks: has the process running at that moment used up the **time slice** allotted to it? If so, the process is taken off the processor and the processor is given to another process. This arrangement is called **preemptive scheduling**. The process itself is not terminated; only its turn passes to another. A process is terminated only if it exceeds the limit on total processor time granted to it (`RLIMIT_CPU` on Linux).

Example 2.3

There are two processes, A and B. Before running A, the operating system loads 3 into the counter. What happens on each tick of the clock?

1. **Loading.** In kernel mode the operating system writes 3 into the counter, then switches to user mode and starts A.
2. **Tick 1.** The counter becomes 2. A continues to run, unaware of what is going on.
3. **Tick 2.** The counter becomes 1. A is still on the processor.
4. **Tick 3.** The counter drops to 0 and the timer interrupt is raised. The processor switches to kernel mode and the interrupt handler runs.
5. **Scheduling.** The scheduler sees that A's slice has run out, saves A's state in its PCB, and chooses B. It loads the counter again and starts B in user mode.

A has used the processor for three ticks. Even if A were in an infinite loop, it would have to give up the processor at the end of this period.

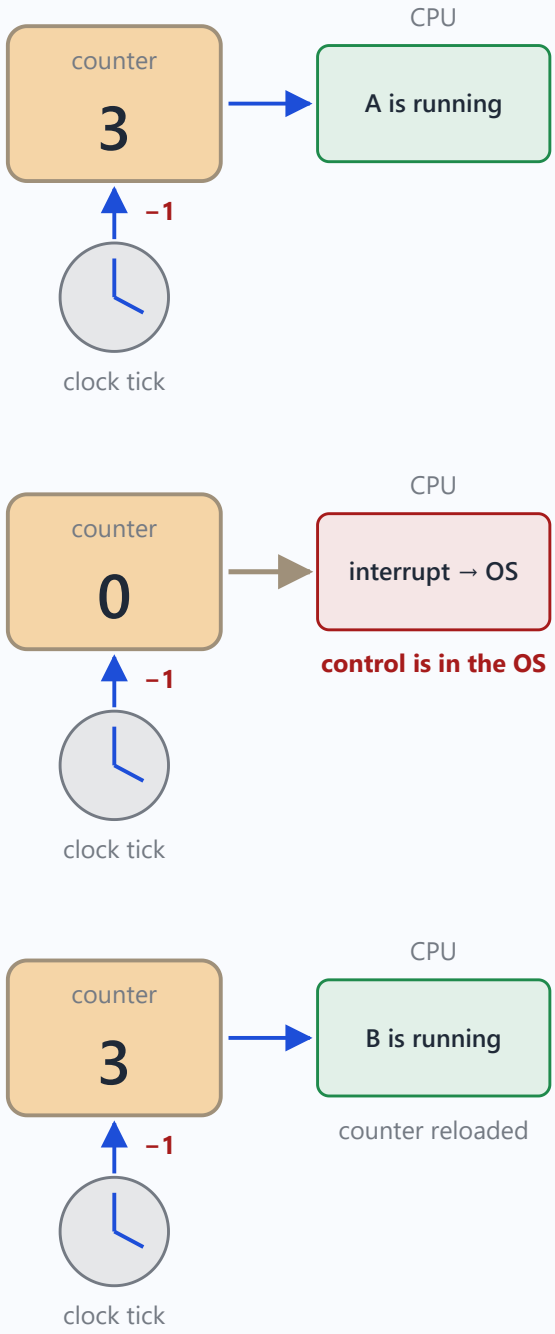


Figure 2.11: Three moments of Example 2.3. In the first frame the counter is 3 and A is running. In the second frame the counter has dropped to 0, the interrupt has been raised, and control has passed to the operating system. In the third frame the scheduler has run B and reloaded the counter.

On Linux the timer interrupt is raised at regular intervals, between 100 and 1000 times per second; each interrupt is called a **tick**. Frequent ticks give finer time slices and more responsive scheduling, but handling each tick takes time from the processor. Moreover, even when the system is idle, the processor has to wake up on every tick. For this reason current kernels also offer a mode that raises no ticks while idle (*tickless*, NO_HZ); this mode saves energy, especially on laptops.

2.4 System calls

Dual-mode operation forbids user programs to access the hardware directly. Yet programs need to read files, ask for memory, write to the screen, and create new processes. The only party that can do these jobs is the kernel. The controlled way for a program to request service from the kernel is the system call.

2.4.1 What is a system call?

Definition 2.9

A **system call** is the defined interface that programs use to reach operating system services such as files, memory, processes, time, and the network. The application starts the call; the kernel carries out the requested work in kernel mode.

Programmers often do not use system calls directly. In between sits an **application programming interface** (API). On UNIX-like systems this interface is the POSIX standard, implemented inside the C library (glibc or musl). On Windows the Windows API, and in Java the Java API, serve the same purpose. An API provides two benefits. The first is portability: a program written to POSIX can be compiled and run on another UNIX whose system call numbers and details are different. The second is convenience: API functions hide the laborious details of raw system calls.

The set of system calls varies from system to system. Linux on x86-64 has about 370 system calls. The basic concepts are common, though: every system has calls that create processes and open, read, and write files. To see which system calls a program makes, one uses `strace` on Linux, `dtruss` on macOS, and the Process Monitor tool on Windows.

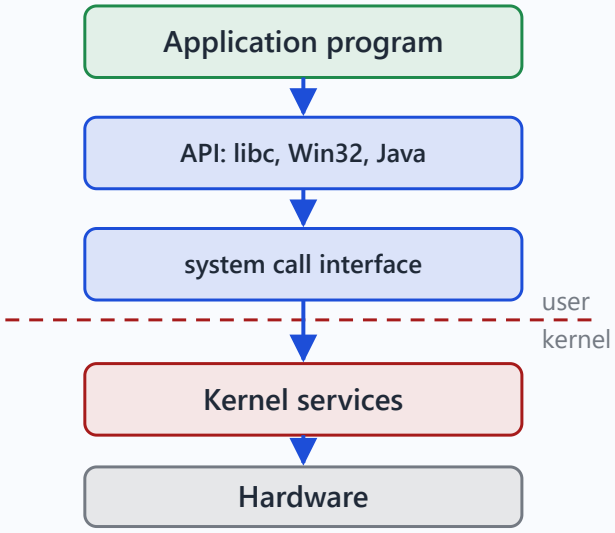


Figure 2.12: The layers a system call passes through: the application program, the API (libc, Win32, Java), the system call interface, the kernel services, and the hardware. The dashed line separates user mode from kernel mode.

2.4.2 From the API to the kernel

Let us follow the call `open("notes.txt", O_RDONLY)` of a C program. Figure 2.13 shows the call's journey from start to finish.

1. **The wrapper.** The program calls the `open()` function in the C library like an ordinary function. This function is a **wrapper**: it puts the call number and parameters into registers and then executes the trap instruction. The processor switches to kernel mode.
2. **The system call interface.** The entry code in the kernel uses the call number as an index into the **system call table**. On x86-64 Linux, `read` is number 0, `write` 1, `open` 2, and `close` 3.
3. **The implementation.** The pointer in the table points to the kernel's `open` code. This code resolves the path name, finds the file, checks the permissions, and creates an open file record.
4. **The return.** The result goes back to the user. Because 0, 1, and 2 are taken, the first free descriptor is 3; `open` returns 3 and the processor switches back to user mode.

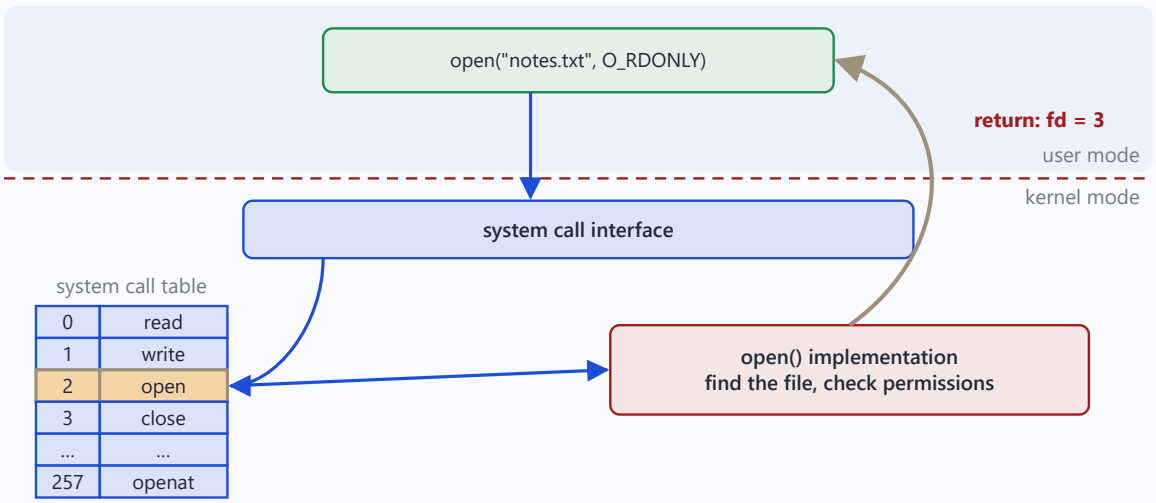


Figure 2.13: The path of the `open` call from user mode to kernel mode. The system call interface finds entry number 2, the pointer in the table leads to the kernel's `open()` implementation, and the result (`fd = 3`) returns to user mode.

One detail shows well the relationship between the library and the kernel. Current glibc actually implements the `open()` that the program calls with the `openat` system call (number 257 on x86-64). The program still writes `open` and gets the same result; the API hides the change in between. Looking with `strace`, however, the name of the call that goes to the kernel appears as `openat`.

2.4.3 The system call table

How the system call table works can be followed with a small C model. In the code below the table consists of only three entries. In the real Linux kernel the table holds a few hundred entries, but the logic is the same.

```
typedef long (*syscall_t)(long, long, long);
syscall_t table[3] = {sys_read, sys_write,
                      sys_open};
long syscall_entry(long no, long a,
                   long b, long c) {
  if (no < 0 || no >= 3)
    return -ENOSYS;     /* bad number */
  return table[no](a, b, c);
}
```

The first line defines a pointer type for a function that takes three `long` parameters and returns `long`, and names it `syscall_t`. The `table` in the second line is an array of three pointers of this type: entry 0 points to `sys_read`, entry 1 to `sys_write`, and entry 2 to `sys_open`. The

`syscall_entry` function represents the kernel's single entry point. It first checks whether the call number lies within the bounds of the table. If the number is invalid it returns `-ENOSYS`; if it is valid, it uses the pointer in the table to call the right function and returns its result.

Example 2.4

Suppose the program makes three calls in a row: a `write` that writes three bytes, `open("notes.txt", 0)`, and `syscall(99)`, which has no counterpart in the table. How does `syscall_entry` work on each call?

Call	no	Bounds check	Code run	Result
`write`	1	0 ≤ 1 < 3: valid	`table[1]` → `sys_write`	3 (bytes written)
`open`	2	0 ≤ 2 < 3: valid	`table[2]` → `sys_open`	3 (new descriptor)
`syscall(99)`	99	99 ≥ 3: invalid	none	−ENOSYS (−38); the program gets −1, `errno = ENOSYS`

In the first two calls the libc wrapper puts the number into the `rax` register and executes the `syscall` instruction. The hardware switches to kernel mode and jumps to the single entry point. The bounds check passes and the function pointer in the table calls the right handler. In the third call the number is outside the table. Without the check, `table[99]` would use a memory cell far beyond the end of the array as a function pointer; the kernel would jump to a random address. Thanks to the check, the call is rejected with the value −38. libc turns this negative value into `errno = ENOSYS` and returns −1 to the program.

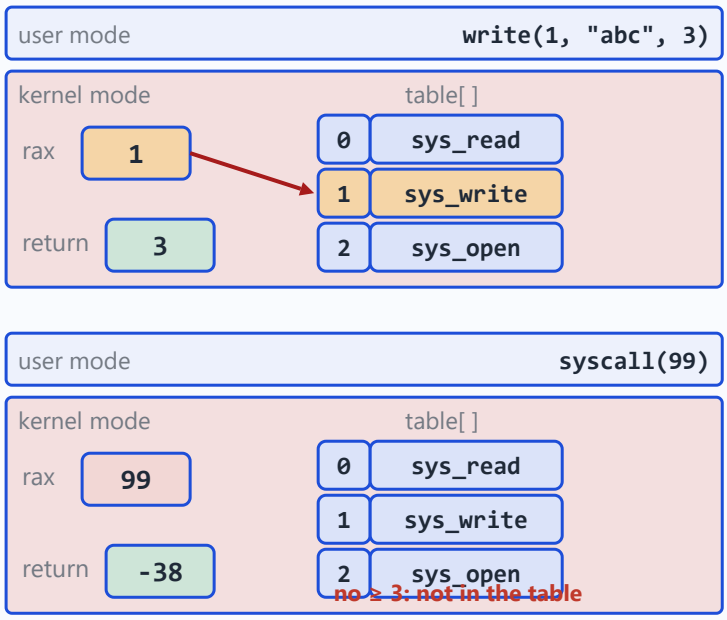


Figure 2.14: Two moments of Example 2.4. In the first frame `no = 1` : `table[1]` has been selected and `sys_write` has returned 3. In the second frame `no = 99` : because the number lies outside the table, the call has been rejected with −ENOSYS.

The lesson from this model is general: the kernel trusts no value from the user without validating it. The call number, the file descriptor, and the buffer's address and length are values written by the user program, and all of them are checked. Whether the buffer address given to a `read` call really lies in that process's memory is one of these checks.

2.4.4 Example: copying a file

Even a program that looks ordinary makes many system calls. Consider a program that copies one file to another and asks the user for the file names. The steps of the program and the system calls each step uses are as follows:

Step	System calls
Prompt for and read the input file name	`write` , `read`
Prompt for and read the output file name	`write` , `read`
Open the input file (if it does not exist, report an error and exit)	`open`
Create the output file (if it exists, report an error and exit)	`open` + `O_CREAT`
Loop: read from the input, write to the output	`read` , `write`
Close the files	`close`
Write “done” to the screen	`write`
Terminate normally	`exit`

Table 2.1: Steps of a file-copying program and their system calls.

Even getting a file name from the user takes two calls: `write` to put a prompt on the screen and `read` to read from the keyboard. Opening a file can fail: the file does not exist or there is no read permission. In that case `open` returns −1, and the program writes an error message and exits. If the output file is opened with the flags `O_WRONLY | O_CREAT | O_EXCL`, the call fails when a file of the same name already exists; in this way an existing file is not overwritten by accident. The real work is in the loop: until `read` returns 0, that is, until the end of the file is reached, the bytes read are written with `write`. Even such a simple program makes dozens of system calls; on Linux the command `strace -c cp a b` counts the calls made by the `cp` program by type.

To trace the loop itself in detail, let us consider the following piece of code:

```
char buf[4096];
int in = open("src", O_RDONLY);
int out = creat("dst", 0644);
ssize_t n;  long total = 0;
while ((n = read(in, buf, 4096)) > 0) {
  write(out, buf, n);
  total += n;
}
close(in);  close(out);
printf("%ld bytes\n", total);
```

The program allocates a 4096-byte buffer. It opens the file `src` read-only and puts its descriptor in `in`. The `creat` call creates an empty file named `dst` and sets its permissions to 0644 (the owner reads and writes, others only read). The loop condition both does the reading and tests the result: `read` reads at most 4096 bytes and writes the number of bytes it read into `n`; as long as `n` is positive the loop continues. The body of the loop writes the *n* bytes in the buffer to the destination and updates `total`. At the end the two files are closed and the number of bytes copied is printed.

Example 2.5

`src` is a file of 10,000 bytes, and only the descriptors 0, 1, and 2 of the process are open. The piece of code proceeds step by step as follows.

Step	Call	`n`	File offset	`total`
1	`open("src") → in = 3`	—	0	0
2	`creat("dst") → out = 4`	—	0	0
3	`read , write`	4096	4096	4096
4	`read , write`	4096	8192	8192
5	`read , write`	1808	10000	10000
6	`read`	0	10000	10000
7	`close(3) , close(4)`	0	—	10000
8	`printf → write(1, …)`	0	—	10000

The first `open` returns the smallest free descriptor; because 0, 1, and 2 are taken, `in = 3` and then `out = 4`. The first `read` reads 4096 bytes and the kernel advances the file offset by 4096. The second round also reads 4096 bytes; the offset reaches 8192. Only 10,000 − 8192 = 1808 bytes are now left in the file. The third `read` returns less than requested, 1808 bytes; this is not an error. The offset is at the end of the file and the fourth `read` returns 0. The loop condition becomes false and the loop ends. The program prints `10000 bytes`.

The total number of system calls is 12: 1 `open`, 1 `creat`, 4 `read`, 3 `write`, 2 `close`, and 1 `write` made by `printf`.

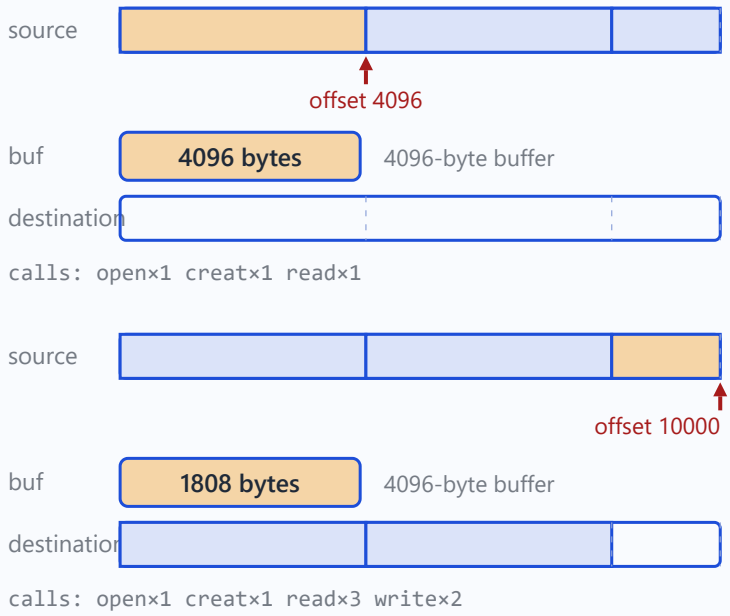


Figure 2.15: Two moments of the copy loop. In the first frame, the first `read`: the first 4096 bytes of the source have been taken into the buffer. In the second frame, the third `read`: the remaining 1808 bytes of the file have been read.

Common mistake

Writing `write(out, buf, 4096)` in the loop body. In the last round the buffer holds only 1808 valid bytes; the remaining 2288 bytes are old data left over from the previous round. If a fixed 4096 is written, the destination file becomes 12,288 bytes and meaningless data is appended to its end. The correct way is to write as many bytes as `n`, the value `read` returned.

The buffer size directly determines the number of system calls made. Because of the two mode switches and the in-kernel input-output work, each system call is much more expensive than an ordinary function call. If the same 10,000 bytes had been copied with a 1-byte buffer, about 20,000

calls would have been needed for reading and writing alone: 10,001 `read` calls and 10,000 `write` calls. With a 4096-byte buffer this number is 7.

2.4.5 The eleven steps of a read call

Tanenbaum's textbook describes the `read(fd, buffer, nbytes)` call by dividing it into eleven steps. The example is based on the historical 32-bit x86 convention: parameters are passed on the stack. Today's x86-64 systems pass the first six parameters in registers, but the order and meaning of the steps do not change.

1. **Steps 1–3.** The user program pushes the parameters on the stack: first `nbytes`, then the address of `buffer`, and last `fd`. C compilers push parameters from last to first.
2. **Step 4.** The program calls the `read()` function in the library like an ordinary function; the return address goes on the stack.
3. **Step 5.** The library puts the call number into the register the kernel expects (`rax = 0` on Linux x86-64).
4. **Step 6.** The trap instruction (`syscall` on x86-64) executes. The processor switches to kernel mode and jumps to a fixed address in the kernel.
5. **Step 7.** The kernel looks at the call number and branches to the right handler through the system call table.
6. **Step 8.** The handler copies at most `nbytes` bytes from the file that `fd` refers to into `buffer`. If the data is not ready, for example if a line has not yet arrived from the keyboard, the process blocks and the processor is given to another process.
7. **Step 9.** When the work is done, control returns to user mode, to the instruction after the trap instruction. The result, the number of bytes read, is in a register.
8. **Step 10.** The `read()` function in the library returns to the user program that called it.
9. **Step 11.** The program cleans up the parameters it pushed on the stack (increments the stack pointer) and continues with the next instruction.

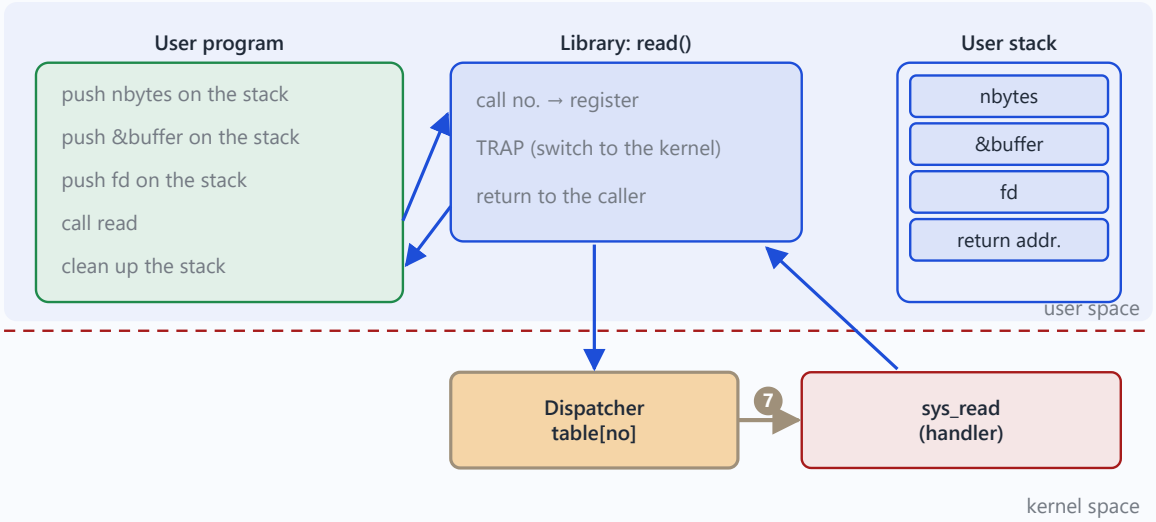


Figure 2.16: Step 7 of the `read` call. The parameters (`nbytes` , `&buffer` , `fd`) and the return address are on the user stack. The library has moved into the kernel with the trap instruction, and the dispatcher in the kernel has branched to the `sys_read` handler through `table[no]` . The dashed line separates user space from kernel space.

The first four steps and the last two take place entirely in user mode; these are ordinary function call steps. Entry into the kernel happens at step 6 and exit at step 9. Step 8 shows the most important difference between a system call and an ordinary function: the call can wait for an event. Until the awaited event happens, the process does not use the processor; the operating system uses this time for other processes.

2.4.6 Parameter passing

There are three general ways to pass parameters to a system call:

1. **Registers.** The fastest way. Linux passes at most six parameters to a system call in registers.
2. **A block or table in memory.** The parameters are written into a structure in memory; the address of the structure is passed in a register. This way is used when there are many parameters. The `clone3(struct clone_args *)` call in Linux is an example.
3. **The stack.** The program pushes the parameters on the stack and the operating system pulls them from there. It was common in the historical 32-bit systems.

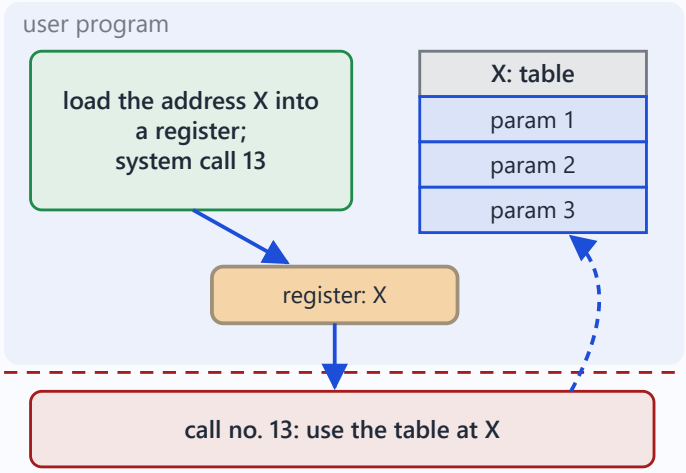


Figure 2.17: Passing parameters through a table in memory. The user program writes the three parameters into the table at address X, loads X into a register, and makes system call number 13. The kernel reaches the parameters by reading the table at X.

On x86-64 Linux the roles of the registers are fixed:

Register	Role
`rax`	call number; the result on return
`rdi` , `rsi` , `rdx`	1st, 2nd, and 3rd parameters
`r10` , `r8` , `r9`	4th, 5th, and 6th parameters (`r10` instead of the `rcx` of the C function call convention)
`syscall`	the instruction that moves into the kernel; the values of the `rcx` and `r11` registers are destroyed
ARM64	`x8` call number, `x0` – `x5` parameters; the transition instruction is `svc #0`

Table 2.2: System call registers on x86-64 Linux.

The answer to the question of why the fourth parameter is in `r10` instead of `rcx` is in the last row of the table. The `syscall` instruction saves the return address in `rcx` and the flags register in `r11` . A parameter placed in `rcx` would therefore be erased before reaching the kernel.

Error reporting follows a common rule as well. When there is an error, the kernel returns a value between −1 and −4095; this value is the negative of the error code (−errno). When the libc wrapper sees this value, it writes the error code into the `errno` variable and gives −1 to the caller. The program notices the error by checking whether a system call returned −1 and learns the cause from `errno` .

2.4.7 Code trace with write and read

One of the smallest programs that writes text to the screen is this one, which uses `write` directly instead of `printf` :

```c
#include <unistd.h>      // write()
#include <string.h>      // strlen()

int main(void) {
    const char *msg = "Hello, Türkiye!\n";
    size_t len = strlen(msg);        // 17 bytes
    ssize_t n = write(1, msg, len);  // 1: stdout
    if (n == -1)
        return 1;                    // error
    return 0;
}
```

The pointer `msg` points to the string in the read-only data segment. `strlen(msg)` gives the string's length in bytes. `write(1, msg, len)` writes `len` bytes to descriptor 1, that is, to standard output. If the returned value is −1, an error has occurred and the program exits with 1; otherwise it exits with 0.

Example 2.6

The program runs line by line as follows. The non-ASCII character in the string affects the byte count.

1. `strlen`. In UTF-8 encoding, “Hello, ” is 7 bytes (including the space after the comma). “Türkiye” is 8 bytes, because the letter ü takes 2 bytes. “!” and the newline character “\n” are 2 bytes. The total is `len = 17`; the number of characters is 16.
2. **The wrapper.** libc fills the registers: `rax = 1` (`write`), `rdi = 1`, `rsi = msg`, `rdx = 17`. Then it executes the `syscall` instruction.
3. **The kernel.** In kernel mode, `sys_write` passes the 17 bytes to the terminal to which descriptor 1 is attached. `Hello, Türkiye!` appears on the screen.
4. **The return.** Control returns to user mode and `n = 17`. This value is the number of bytes written; in some cases it can be smaller than requested (a partial write).
5. **Error check.** `n` is not −1, so there is no error. Had there been an error, `write` would have returned −1 and `errno` would have been set.
6. **Exit.** `main` returns 0; libc terminates the process with the `exit_group(0)` call.

If the programmer had computed the length by hand and written a fixed 15, the last two bytes (“!” and “\n”) would not be sent and the output would be cut off as “Hello, Türkiye”. Computing the length with `strlen` prevents this mistake.

The counterpart program on the reading side reads one line from the keyboard and writes it back to the screen unchanged:

```c
#include <unistd.h>
#include <stdio.h>

int main(void) {
    char buf[100];
    ssize_t n = read(0, buf, sizeof(buf));
    if (n == -1) {
        perror("read error");
        return 1;
    }
    write(1, buf, n);    // exactly n bytes
    return 0;
}
```

Example 2.7

While the program is running, the user types `hello` and presses the Enter key. How does the program proceed?

1. **The buffer.** `buf` is a 100-byte array allocated on the stack; its contents are uninitialized, that is, they hold random values.
2. **Blocking.** `read(0, buf, 100)` wants to read from standard input, that is, from the keyboard. No line has been typed yet. The process blocks and the scheduler runs another process.
3. **The user types.** The user types `hello` and presses Enter. The terminal driver echoes the characters to the screen and accumulates the line in its own buffer.
4. **Wake-up.** When the line is complete, 6 bytes (`hello` and "\n") are copied into `buf` , the process wakes up, and `read` returns 6. No "\0" is added to the end of the buffer.
5. **Error check.** `n = 6` , there is no error; `perror` is skipped.
6. **Writing.** `write(1, buf, 6)` writes exactly 6 bytes to the screen. `hello` appears twice on the screen: the first is the terminal driver's echo, the second is the program's output.
7. **Exit.** The program terminates with 0.

Common mistake

Printing the buffer that was read with `printf("%s", buf)` . `read` does not put a string-terminating character ("\0") at the end of the buffer; `buf` is not a C string. `printf` keeps reading until it finds the first zero byte and dumps the random bytes in uninitialized memory onto the screen as well. The correct way is to write exactly `n` bytes, using the value `n` that `read` returned.

2.4.8 The standard C library

The `printf` that a C programmer uses every day is not a system call. It is a library function that runs in user mode and makes a system call when needed. Figure 2.18 shows the path of the call `printf("Hello\n")`.

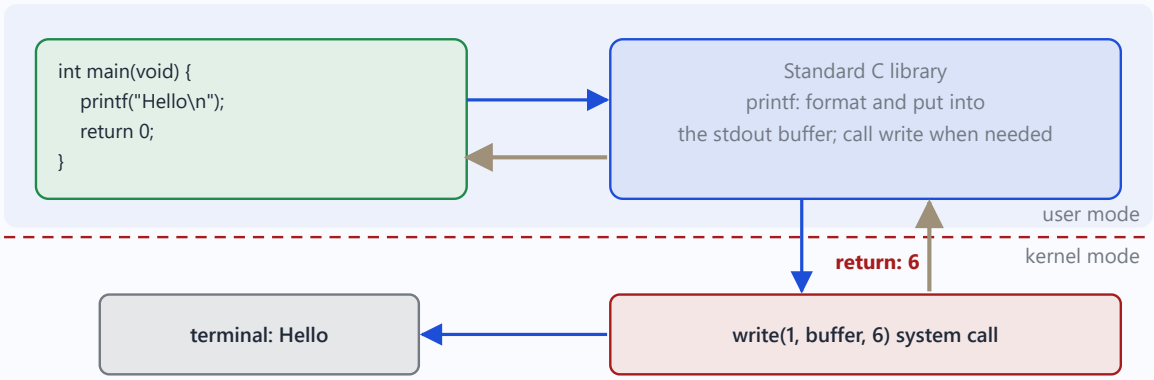


Figure 2.18: The transformation of a `printf` call into the `write` system call. The program calls `printf`; the standard C library formats the text and puts it into the `stdout` buffer. When the newline arrives, the library makes the system call `write(1, buffer, 6)`; the kernel passes the bytes to the terminal and returns 6.

1. The program calls `printf`. This is an ordinary function call running in user mode.
2. libc processes the format string and puts the result into the buffer of `stdout`. If the output goes to a terminal, the buffer is **line-buffered**: it is flushed when a newline arrives. If the output is redirected to a file or a pipe, the buffer is **fully buffered**: it is flushed when it fills up or when the program ends.
3. Because the text ends with "\n", libc makes the call `write(1, buffer, 6)`. "Hello" is 5 bytes and the newline is 1 byte.
4. `write` returns 6, and `printf` returns to the caller. The command `strace ./hello` shows only `write`; because `printf` is not a system call, it does not appear in the list.

The functions of the standard C library are grouped by header file: `stdio.h` (`printf`, `fopen`), `stdlib.h` (`malloc`, `exit`), `string.h` (`strlen`, `memcpy`), `math.h`, and `time.h`. POSIX also defines system call wrappers such as `fork`, `read`, and `write` in the `unistd.h` header. The widespread implementations of the library are glibc and musl on Linux and UCRT on Windows. Library functions fall into two groups. Functions such as `strlen` run entirely inside the library and never enter the kernel. Functions such as `write` wrap a system call.

2.4.9 Types of system calls

System calls are grouped into six groups according to the work they do. Table 2.3 lists the tasks of each group and their POSIX examples.

Type	Tasks	POSIX examples
Process control	create, terminate, load, wait, send a signal	`fork`, `execve`, `exit`, `waitpid`, `kill`
File management	create, delete, open, close, read, write, get attributes	`open`, `read`, `write`, `close`, `stat`
Device management	request and release a device, read, write, control	`ioctl`, `read`, `write`, `mmap`
Information maintenance	time, date, system and process information	`getpid`, `uname`, `clock_gettime`
Communication	set up a connection, send and receive messages, shared memory	`pipe`, `socket`, `sendmsg`, `shm_open`
Protection	get and set permissions, user identity	`chmod`, `chown`, `umask`, `setuid`

Table 2.3: Types of system calls (after Silberschatz, Figure 2.8).

The same groups exist on Windows; only the names change. `CreateProcess` is used to create a process, `ReadFile` to read a file, `DeviceIoControl` for device control, `GetSystemTime` to read the clock, `CreatePipe` to create a pipe, and `SetFileSecurity` for file permissions. It is natural for some calls to appear in more than one group: `read` and `write` are used for both files and devices, because on UNIX devices look like files too.

2.5 POSIX and Win32 calls

The POSIX standard defines hundreds of calls, but a few dozen are enough to understand the basic behavior of a UNIX system. This section takes up the basic calls chosen by Tanenbaum in three groups: process management, file management, and directory management. At the end of the section, the Windows counterparts of the same calls are given. A common notation is used in the tables: `pid` stands for a process identifier, `fd` for a file descriptor, `n` for a byte count, and `s` for the return code of the call. In case of error all of the calls return −1 and set the `errno` variable.

2.5.1 Process management

Call	Description
`pid = fork()`	Creates a child process that is a copy of the parent.
`pid = waitpid(pid, &statloc, options)`	Waits for a child to terminate; writes the exit status into `statloc`.
`s = execve(name, argv, envp)`	Replaces the process's memory image with a new program; does not return if successful.
`exit(status)`	Terminates the process and passes the exit status to the parent.

Table 2.4: POSIX calls for process management.

`fork` is the only way to create a new process on UNIX. The call creates a new process with a copy of the parent's memory image, open files, and registers. After the call there are two processes, and both continue from the same point where `fork` returned. The only way to tell which of the two processes is which is the return value: in the parent `fork` returns the child's PID, and in the child it returns 0. If the call fails, −1 returns to the parent and no child is created. This is why it is said that `fork` "is called once and returns twice".

The child is often wanted to do a different job from the parent. For this the child calls `execve`. `execve` replaces the process's code, data, and stack with the image of the new program on disk. The process is the same process and its PID does not change; it is simply running another program now. A successful `execve` does not return, because the code that called it is no longer in memory. The line after `execve` runs only if the call fails.

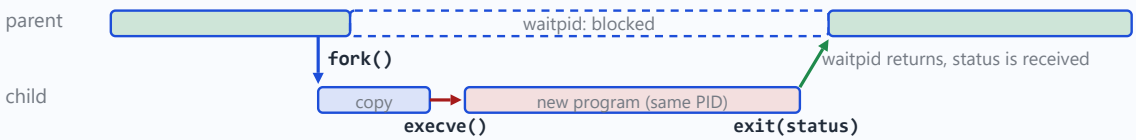


Figure 2.19: The process life cycle. The parent creates a copy with `fork()` and blocks with `waitpid`. The child moves to the new program with `execve()` and calls `exit(status)` when it has finished its work. At this moment the parent's `waitpid` call returns and the exit status is received.

Windows does not split this job into two calls: a single `CreateProcess` call both creates the new process and loads the new program into it. The benefit of UNIX's two-step approach is the short interval between `fork` and `execve`. In this interval the child can change its own open files; for example, it can redirect its standard output to a file or a pipe. The shell's `>` redirection and pipelines are set up in exactly this interval.

2.5.2 A simple shell

The best way to see how the process management calls work together is to write a **shell**. A shell is an ordinary user program that takes commands from the user and runs them. The skeleton given by Tanenbaum hides the details and shows only the essential structure:

```
#define TRUE 1

while (TRUE) {              /* forever */
    type_prompt();          /* print the prompt */
    read_command(command, parameters);

    if (fork() != 0) {      /* parent */
        waitpid(-1, &status, 0);
    } else {                /* child process */
        execve(command, parameters, 0);
        perror("execve");   /* on failure */
        _exit(127);
    }
}
```

The shell runs in an infinite loop. In each round it prints the prompt (`type_prompt`) and reads the line the user typed, splitting it into the command name and parameters (`read_command`). Then it calls `fork`. If the return value is not 0, this code is running in the parent, that is, in the shell itself; the shell waits for the child to finish with `waitpid`. If the return value is 0, the code is running in the child; the child replaces itself with the requested program using `execve`. Because a successful `execve` does not return, the next two lines run only if the call fails: the child prints the error and terminates with `_exit(127)` (127 is the exit status shells use for "command not found").

Example 2.8

In the shell with PID 812 the user types `ls`. The working directory contains the files `notes.txt` and `hw.c`. One round of the shell is shown in the table below.

Step	Line	Event	Processes	Screen
1	4	The prompt is printed and the shell waits for the user.	shell (812)	`$`
2	5	The line is parsed: `command = "/bin/ls"`, `parameters = ["ls", NULL]`.	shell (812)	`$ ls`
3	7	`fork`: returns 1045 in the parent and 0 in the child.	shell (812), child (1045)	`$ ls`
4	8	The parent blocks with `waitpid(-1, …)`.	shell (waiting), child (1045)	`$ ls`
5	10	The child becomes `ls` with `execve`; it lists the directory.	shell (waiting), ls (1045)	`notes.txt hw.c`
6	8	`ls` ends with `exit(0)`; `waitpid` returns, `status = 0`.	shell (812)	
7	4	The loop returns to the start and a new prompt is printed.	shell (812)	`$`

In step 3, after `fork` the same code runs in two processes at once. In the shell the condition `fork() != 0` is true (1045 ≠ 0) and the shell goes to line 8. In the child the condition is false (0) and the child goes to line 10 in the `else` branch. In step 5 the child's PID remains 1045; only the program it runs has changed. Because the first parameter of `waitpid` is −1, the shell waits for the first child to finish, whichever child it is.

Real shells go far beyond this skeleton. They search for the command name in the directories in the `PATH` environment variable, manage jobs running in the background with `&`, and set up redirections and pipelines. But at the heart of all of them is the same `fork` – `execve` – `waitpid` trio.

Common mistake

Forgetting to terminate the child when `execve` fails. If the user types a nonexistent command, `execve` returns −1 and the child goes on running with the shell's code. If there is no `exit` after `execve` in the child, the child returns to the top of the loop and starts behaving like a shell itself. The user now faces two shells. This is why the line `_exit(127)` is in the skeleton.

2.5.3 File management

Call	Description
`fd = open(file, how, …)`	Opens the file for reading, writing, or both; creates it if necessary.
`s = close(fd)`	Closes an open file.
`n = read(fd, buffer, nbytes)`	Reads at most `nbytes` bytes from the file into the buffer.
`n = write(fd, buffer, nbytes)`	Writes data from the buffer to the file.
`position = lseek(fd, offset, whence)`	Moves the file position pointer (`SEEK_SET`, `SEEK_CUR`, `SEEK_END`).
`s = stat(name, &buf)`	Gets the file's status information (size, permissions, times).

Table 2.5: POSIX calls for file management.

For each open file the kernel keeps a **position pointer**. This pointer determines from which byte of the file the next `read` or `write` call will start. Every read and write advances the pointer by the number of bytes processed. This is how, in the copy example, the file can be read in order from beginning to end: the program specifies no position anywhere; the kernel tracks the position itself.

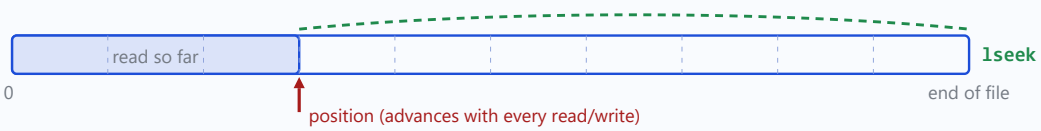


Figure 2.20: The file position pointer. The part already read is on the left, and the pointer advances to the right with every `read` or `write`. `lseek` moves the pointer to any desired point in the file.

`lseek` is used to go to a particular place in the file. The `whence` parameter determines what `offset` is measured from: `SEEK_SET` from the beginning of the file, `SEEK_CUR` from the pointer's current place, `SEEK_END` from the end of the file. The call returns the pointer's new position in bytes from the start of the file. For example, `lseek(fd, 0, SEEK_END)` moves the pointer to the end of the file and the return value equals the file's size.

Example 2.9

A file of 1000 bytes is opened with `fd = open(…, O_RDONLY)`. Then the calls `read(fd, buf, 300)`, `lseek(fd, 100, SEEK_CUR)`, `read(fd, buf, 300)`, `lseek(fd, -50, SEEK_END)`, and `read(fd, buf, 300)` are made. What does each call return?

Solution. The pointer is 0 at open. The first `read` reads 300 bytes and returns 300; the pointer reaches 300. `lseek(fd, 100, SEEK_CUR)` moves the pointer 100 forward and returns 400. The second `read` reads bytes 400–699 and returns 300; the pointer becomes 700. `lseek(fd, -50, SEEK_END)` moves the pointer 50 bytes back from the end, to 950, and returns 950. The last `read` asks for 300 bytes, but only 50 bytes remain in the file; it returns 50.

`stat` fills a structure with information about a file without touching its contents: its size, owner, permission bits, and last access and modification times. Alongside these basic calls, current Linux offers extended versions such as `openat`, `pread` / `pwrite` (calls that read and write at a given position without changing the position pointer), and `statx`.

2.5.4 Directory management and other calls

Call	Description
`s = mkdir(name, mode)`	Creates a new directory.
`s = rmdir(name)`	Deletes an empty directory.
`s = link(name1, name2)`	Creates a new directory entry named `name2` for the file that `name1` points to.
`s = unlink(name)`	Deletes a directory entry.
`s = mount(special, name, flag)`	Mounts a file system (its form on Linux: `mount(source, target, type, flags, data)`).
`s = umount(special)`	Unmounts a file system.
`s = chdir(dirname)`	Changes the working directory.
`s = chmod(name, mode)`	Changes the file's protection (permission) bits.
`s = kill(pid, signal)`	Sends a signal to a process; despite its name, it is not used only to kill processes.
`seconds = time(&seconds)`	Gives the number of seconds elapsed since 00:00 UTC on 1 January 1970 (the Unix epoch).

Table 2.6: Directory management and other POSIX calls.

The last call in the table is the source of a well-known problem in the software world. In old systems the time value was kept in a 32-bit signed integer (`time_t`). The largest value such an integer can hold is $2^{31} - 1$ = 2,147,483,647. This many seconds, counted from the start of 1970, corresponds to 19 January 2038. After that moment the counter overflows and the date goes back to 1901. This problem is known as the “year 2038 problem”. Current systems remove the problem by keeping time in a 64-bit `time_t` .

Note that the `chdir` call changes a property of the process. The working directory belongs to the process; even if a child process changes its own working directory, the parent's working directory stays the same. This is why the shell's `cd` command is implemented not as a separate program but as a built-in command inside the shell itself.

2.5.5 link and unlink

On UNIX, a file's name and the file itself are separate things. The file's real information (its owner, permissions, size, and the disk blocks of its data) is kept in a structure called an **inode**. A directory is just a table made of (inode number, name) pairs. A name is nothing more than a directory entry that points to an inode.

This separation has a natural consequence: there can be several directory entries that point to the same inode. A file then has several names. Each of these names is called a **hard link**. The inode keeps the number of names pointing to it in a counter called the **link count**.

Example 2.10

In the directory `/usr/jim` there is a file named `memo` that points to inode number 70. First `link("/usr/jim/memo", "/usr/ast/note")` and then `unlink("/usr/jim/memo")` are called.

1. **Initial state.** The directory `/usr/jim` has the entries (31, bin), (70, memo), (59, f.c.), and (38, prog1). The link count of inode 70 is 1.
2. `link` **.** The entry (70, note) is added to the directory `/usr/ast` . No data is copied; the new entry points to the same inode. The link count becomes 2. Now `/usr/jim/memo` and `/usr/ast/note` are the same file: a change made through one is seen through the other.
3. `unlink` **.** The entry (70, memo) in `/usr/jim` is deleted. The file itself is not deleted; it lives on under the name `note` . The link count drops to 1.

When the link count drops to 0 and no process keeps the file open, the kernel releases the inode and the file's data blocks. This is why UNIX has no separate call for “deleting a file”; the `rm` command actually calls `unlink` .

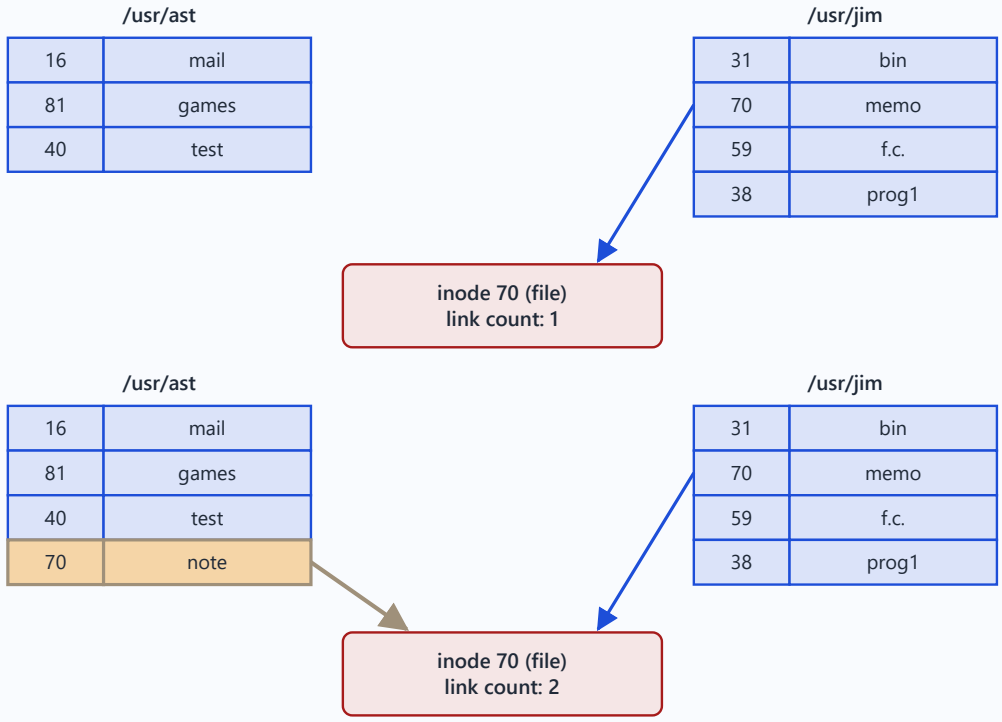


Figure 2.21: The `link` call. Top, the initial state: `/usr/jim/memo` points to inode number 70 and the link count is 1. Bottom, after `link`, `/usr/ast` also has the entry `note` pointing to 70; the link count is 2.

Whether a file is open is tracked by a counter separate from the link count. When a process opens a file with `open`, the kernel resolves the path name and reaches the inode. From then on the process accesses the file not through its name but through the descriptor, going directly to the inode. Even if the name is deleted, the open descriptor keeps working. The following code traces this situation:

```
int fd = open("/usr/jim/memo", O_RDONLY);
link("/usr/jim/memo", "/usr/ast/note");
unlink("/usr/jim/memo");
unlink("/usr/ast/note");
n = read(fd, buf, 100);    /* still readable */
close(fd);
```

Example 2.11

`memo` is a file of 85 bytes, and only the descriptors 0, 1, and 2 of the process are open. The table gives the link count and open count of inode 70 after each line.

Line	Call	Link count	Open	Note
—	start	1	0	only the name `memo` exists
1	`open`	1	1	`fd = 3`
2	`link`	2	1	`note` added

Line	Call	Link count	Open	Note
3	`unlink(memo)`	1	1	only `note` remains
4	`unlink(note)`	0	1	no path left, but the file is open
5	`read`	0	1	`n = 85`
6	`close`	0	0	inode and blocks freed

After line 4 the file has no name in the directory tree at all. Even so, because descriptor 3 keeps it open, the kernel does not delete the inode and the blocks. The `read` in line 5 goes through the descriptor directly to the inode; it needs no name. It asks for 100 bytes and, because the file has 85 bytes, returns 85. When the last descriptor is closed in line 6, both the link count and the open count become 0, and only at this moment does the kernel release the inode and the data blocks.

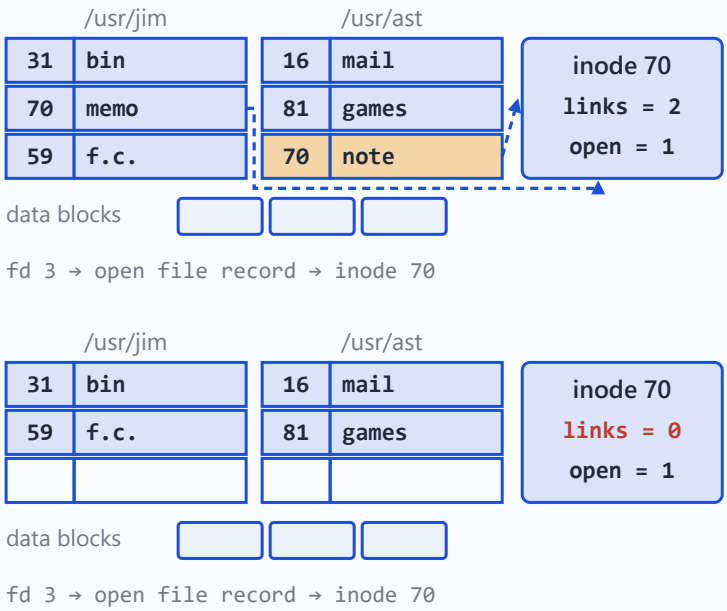


Figure 2.22: Two moments of Example 2.11. In the first frame, after `link` : `/usr/jim/memo` and `/usr/ast/note` point to the same inode, and the link count is 2. In the second frame, after the two `unlink` calls: the directories no longer hold a name for the file, but the inode lives on because the descriptor is open.

Programs make deliberate use of this behavior. A program that creates a temporary file deletes its name with `unlink` right after opening the file. The file can be used through the descriptor as long as the program runs, but it is not visible in the directory tree. Whether the program ends normally or crashes, when the last descriptor is closed the kernel cleans up the file automatically.

2.5.6 Win32 API equivalents

Windows's programming interface is the Win32 API. This interface contains thousands of functions, and most of the basic calls in POSIX have a counterpart in it. Table 2.7 gives Tanenbaum's list with current counterparts.

UNIX	Win32	UNIX	Win32
`fork`	`CreateProcess`	`mkdir`	`CreateDirectory`
`waitpid`	`WaitForSingleObject`	`rmdir`	`RemoveDirectory`
`execve`	(none: `CreateProcess`)	`link`	`CreateHardLink`
`exit`	`ExitProcess`	`unlink`	`DeleteFile`
`open`	`CreateFile`	`mount`	`SetVolumeMountPoint`
`close`	`CloseHandle`	`umount`	`DeleteVolumeMountPoint`
`read`	`ReadFile`	`chdir`	`SetCurrentDirectory`
`write`	`WriteFile`	`chmod`	(none: ACL, `SetFileSecurity`)
`lseek`	`SetFilePointerEx`	`kill`	(none: `TerminateProcess`)
`stat`	`GetFileAttributesEx`	`time`	`GetSystemTime`

Table 2.7: Win32 API counterparts of UNIX calls.

The “none” marks in the table come from design differences. `execve` has no counterpart, because Windows combines process creation and program loading in the single `CreateProcess` call. `chmod` has no counterpart, because Windows manages file permissions with access control lists (ACLs) instead of UNIX's nine permission bits. `kill` has no counterpart, because Windows has no signal mechanism like UNIX's; `TerminateProcess` is used to terminate a process. `link` and `mount`, which had no counterpart in old editions of Tanenbaum's book, are possible today with `CreateHardLink` and `SetVolumeMountPoint`.

Win32 functions do not move into the kernel directly. The real kernel transition is made by the functions beginning with `Nt…` in the `ntdll.dll` library; for example, `CreateFile` ends up calling `NtCreateFile`. The call numbers of these functions inside the kernel change from one Windows version to another and are not documented. Programs therefore always use the API; a program that relies directly on the numbers can stop working at the next Windows update.

2.6 Kernel structures

System calls are the outward-facing side of the operating system. The subject of this section is the inside: how are the components that serve a call organized? The answers to this question have, over decades, gathered around a few main structures. Each structure chooses a different point on the trade-off triangle of 2.1.1.

2.6.1 Overview

A useful way to order the structures is to look at how much work is done in kernel mode. Figure 2.23 arranges the structures on a line by this criterion.

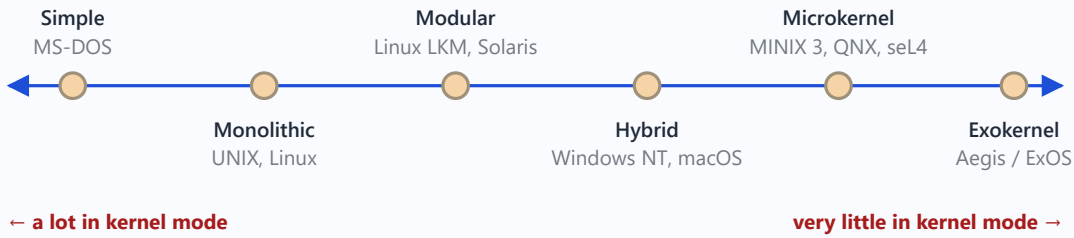


Figure 2.23: Structures by the amount of functionality in the kernel. From left to right: simple structure (MS-DOS), monolithic (UNIX, Linux), modular (Linux LKM, Solaris), hybrid (Windows NT, macOS), microkernel (MINIX 3, QNX, seL4), and exokernel (Aegis/ExOS). At the left end there is a lot in kernel mode, and at the right end very little.

Two concepts do not fit this line neatly. **Layered structure** is not a placement but an organizing principle: components are divided into hierarchical layers. The THE system and MULTICS's protection rings are examples of this principle, and the principle can be applied to a system at any point on the line. **Virtual machines** are a separate axis: the hardware itself is virtualized and several operating systems run on a single machine.

2.6.2 Simple structure: MS-DOS

Some operating systems have no clear structure. These systems were written for small, simple, and limited hardware and grew over time beyond what was expected. MS-DOS is the best-known example.

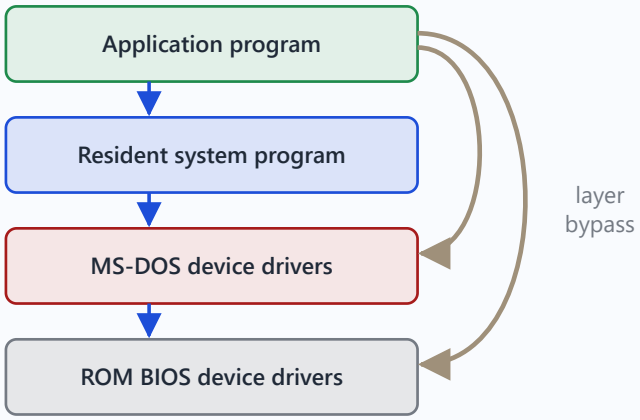


Figure 2.24: The layers of MS-DOS: the application program, the resident system program, the MS-DOS device drivers, and the ROM BIOS device drivers. The orange arrows show that applications can bypass the layers in

between and reach the lower layers directly.

In MS-DOS the interfaces and levels of functionality are not well separated. An application can call BIOS routines directly instead of using the services MS-DOS offers, and can even write directly to screen memory or device ports. The basic reason is hardware: the Intel 8088 processor for which MS-DOS was written had no dual mode. Because there was no hardware protection, a single faulty program could crash the whole system. There were also no clear boundaries between modules and no **data hiding**: every component could touch the data of the others. Silberschatz treats such systems under the heading “simple structure”, separately from monolithic kernels.

2.6.3 The monolithic system

The most common structure is the one in which the whole operating system is linked as a single program that runs in kernel mode.

Definition 2.10

In a **monolithic kernel**, all operating system services run in a single address space in kernel mode. Every procedure in the kernel can call any other directly.

The word “monolithic” does not mean disorder. Tanenbaum describes the basic organization that even a monolithic system has, in three layers:

1. **The main program.** It looks at the number of the system call and starts the requested service procedure.
2. **Service procedures.** Each does the work of one system call.
3. **Utility procedures.** They do the jobs that several services need in common; for example, fetching data from the user program's memory.

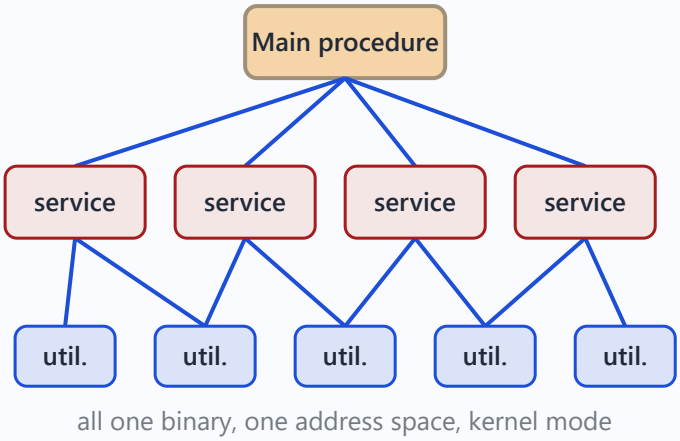


Figure 2.25: The internal organization of a monolithic system. The main procedure calls the service procedures; the service procedures share the utility procedures. All of it is in a single binary file, in a single address space, and in kernel mode.

Traditional UNIX and Linux are monolithic kernels. Figure 2.26 shows the layers of traditional UNIX. Users and system programs (shells, commands, compilers, libraries) are above the system call interface. The kernel below this interface hosts signals, terminal handling, the file system, swapping, disk drivers, CPU scheduling, and virtual memory as a single whole. Below the kernel are the interface between the kernel and the hardware and the device controllers.

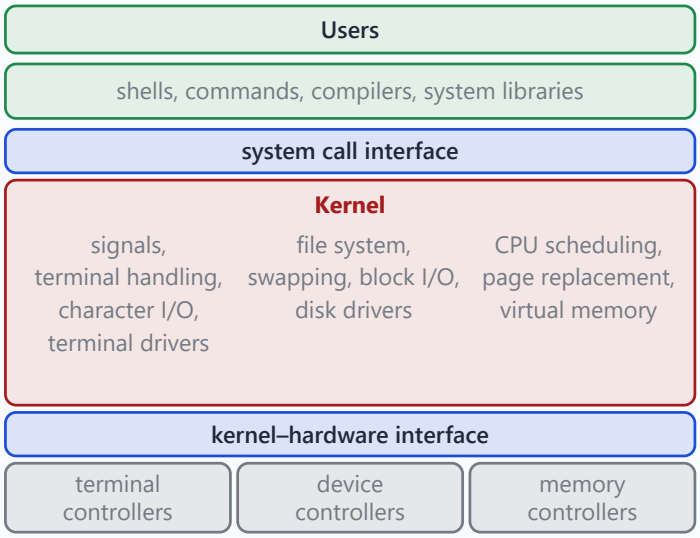


Figure 2.26: The structure of traditional UNIX. Everything between the system call interface and the kernel–hardware interface is a single kernel.

The strengths of the monolithic structure are clear. Components use each other by direct function calls; there is no message preparation, copying, or mode switching in between. That is why monolithic kernels are fast. Their design and implementation are also relatively simple. Their

weaknesses come from the same closeness. Because all components are in the same address space, a faulty pointer in a single driver can corrupt any structure of the kernel and crash the whole system. The code base is also huge: in Linux, which has passed 40 million lines, finding a bug and doing maintenance is not easy. Most monolithic kernels today also accept drivers and file systems as modules that can be loaded at run time; this added flexibility is touched on in 2.6.9.

2.6.4 Layered structure

The classic way to make a large system manageable is to divide it into layers.

> **Definition 2.11**
>
> In a **layered system**, the operating system is divided into layers. The bottom layer is closest to the hardware and the top layer closest to the user. Each layer uses only the services of the layers below it.

The first example of this structure is the THE system that Edsger Dijkstra developed with his students in 1968. Figure 2.27 shows the six layers of THE.

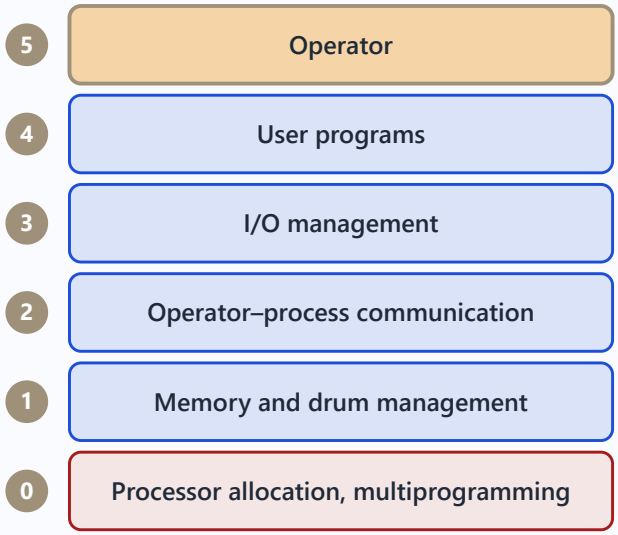


Figure 2.27: The layers of the THE system. 0: processor allocation and multiprogramming; 1: memory and drum management; 2: operator–process communication; 3: I/O management; 4: user programs; 5: the operator.

- **Layer 0** switches among processes when interrupts arrive and when the timer expires. Those above this layer do not have to think about several processes sharing a single processor; each process is written as if it had its own processor.

- **Layer 1** allocates memory to processes and moves pages between main memory and the drum, the secondary memory of that era. The upper layers do not know whether a page is in memory or on the drum.
- **Layer 2** manages the communication between each process and the operator console. Each process above behaves as if it had its own console.
- **Layer 3** manages the I/O devices. The upper layers work with abstract devices instead of the details of the real ones.
- **Layer 4** is the user programs. These programs do not deal with the details of memory, the console, or I/O.
- **Layer 5** is the system operator.

The greatest benefit of the layered structure is that debugging and verification can be done layer by layer. Layer 0 is tested alone and its correctness is made certain. Then layer 1 is added; if an error appears, then since the layer below is correct, the error is in layer 1. Abstraction and data hiding are easy too: each layer hides its own data structures from the upper layers.

The structure has two important difficulties. The first is defining the layers correctly. The dependencies among components are often two-way. For example, the memory manager needs the I/O driver to write pages to disk; the I/O driver needs the memory manager to allocate buffers. It is hard to decide which will be below. The second is performance: with every layer crossing, a request incurs extra work such as parameter preparation and checking. For this reason purely layered operating systems have not become widespread. The idea lives on today in network protocol stacks: a network packet passes in turn through the application, transport, network, and link layers.

2.6.5 The microkernel

The biggest problem of the monolithic structure is that a bug in one component can bring down the whole system. The microkernel approach gives a radical answer to this problem: only the things that really have to be there stay in the kernel.

Definition 2.12

In a **microkernel**, only interprocess communication (message passing), basic scheduling, address space management, and interrupt handling stay in kernel mode. Drivers, file systems, and the network stack run in user mode as **server processes** and communicate with each other by messages.

Adding a new service in a microkernel means adding a new server in user space; the kernel itself does not change. Mach, MINIX 3, QNX Neutrino (common in the automotive sector), the L4 family, the formally verified seL4, and Google Fuchsia's Zircon kernel are examples of this approach. macOS, although derived from Mach, is not a pure microkernel but a hybrid kernel.

A principle often cited in microkernel design is the **separation of mechanism and policy**. Mechanism determines *how* a job is done, and policy *what* is done. For example, the kernel provides the mechanism for running the highest-priority ready process. Which priorities are given to processes is decided by a component in user mode. In this way, when the policy is to be changed, the kernel need not be touched.

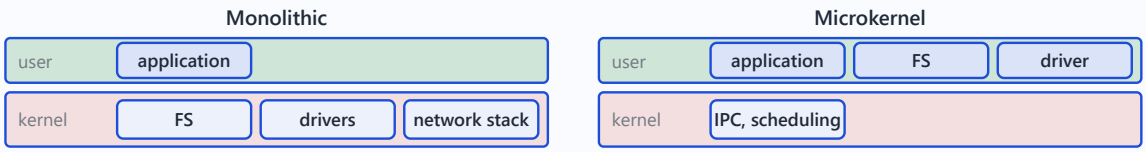
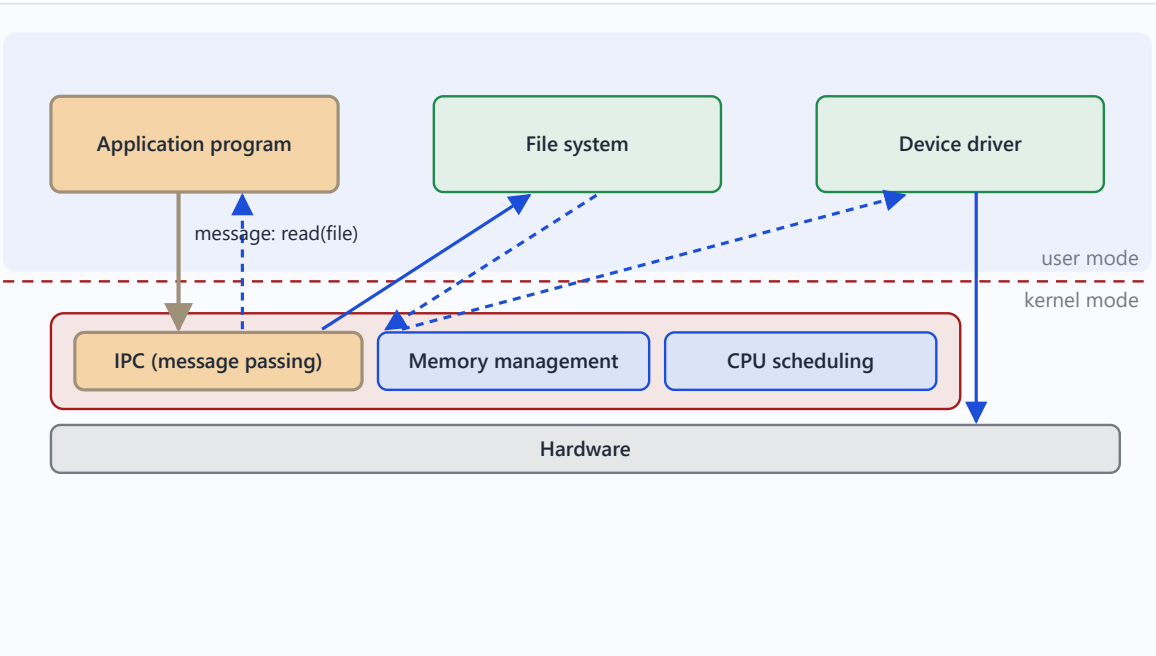


Figure 2.28: A comparison of a monolithic kernel and a microkernel. On the left, the file system, drivers, and network stack are in kernel mode. On the right, the file system and the driver are separate processes in user mode; only IPC and scheduling remain in the kernel.

The advantages of the microkernel are these. A crashed driver does not bring down the system; only that driver's process dies and it can be restarted. Because the kernel is small, it is easy to test, to verify, and to port to other hardware; the correctness of seL4's kernel of a few thousand lines has been proved mathematically. Because there is little privileged code, the surface an attacker can target is also small. The disadvantage is performance: every request between components needs a message and several mode switches. The designs of the L4 family have greatly reduced this overhead by making message passing extremely fast.

2.6.6 Message passing in a microkernel

In a microkernel, an application reading from a file follows a much longer path than in a monolithic kernel. Figure 2.29 shows two moments of this path.



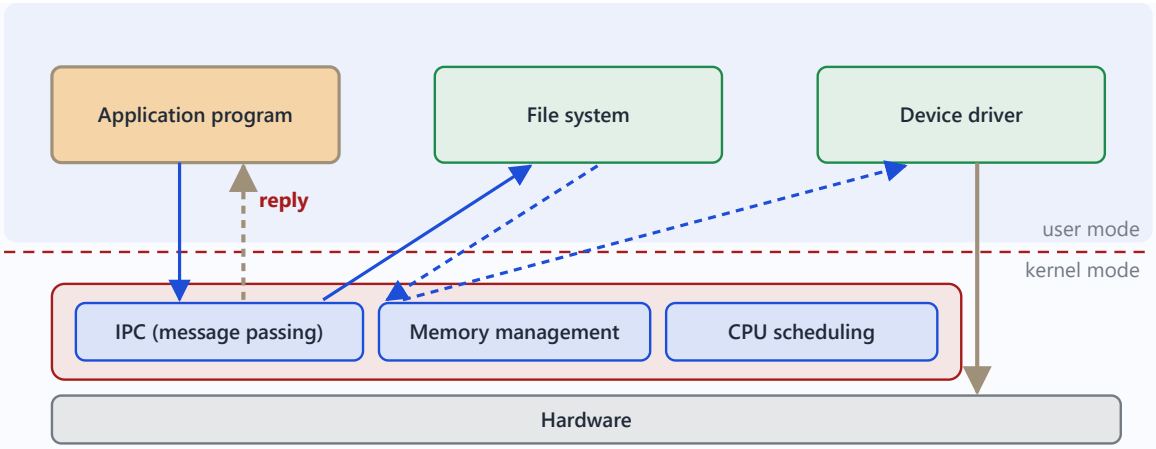


Figure 2.29: Reading a file in a microkernel. At the top, the application sends a message to the file system server; the message passes through the kernel's IPC mechanism. At the bottom, the device driver has reached the hardware with the I/O the kernel permits, and the reply is returning to the application along the same path.

The application hands its request to the kernel as a message; the kernel delivers the message to the file system server in user mode. The file system finds which disk block is to be read, but it cannot access the disk itself; so it sends a message to the disk driver, again through the kernel. The driver reads the block with the I/O operations the kernel permits. The replies follow the same path in reverse. In a monolithic kernel, the same job is a chain of functions calling one another inside a single system call.

To see the path in more detail, let us trace the code of the client and the file system server (FS). The code below is written with two primitives named `send` and `receive`. `send(destination, &m)` sends a message to the destination; `receive(source, &m)` waits until a message arrives from the specified source (or from anyone, if `ANY`).

```
/* client */
m.type = READ; m.file = 7; m.offset = 0;
send(FS, &m);
receive(FS, &m);       /* wait for reply */
/* file-system server (FS) */
while (1) {
  receive(ANY, &m);
  block = find_block(m.file, m.offset);
  send(DISK, &block);
  receive(DISK, &data);
  m.result = data;
  send(m.source, &m);
}
```

Example 2.12

The client wants to read from byte 0 of file number 7. The file system knows that this position is in block 120. At the start, FS and the disk driver are waiting for work in `receive`.

1. **The request is prepared.** The client writes its request into the message structure: `m.type = READ`, `m.file = 7`, `m.offset = 0`.
2. **Message 1.** `send(FS, &m)` is a trap into the kernel. The kernel copies the message into FS's buffer and wakes FS up. The client blocks in `receive(FS, &m)` waiting for the reply.
3. **FS gets the job.** FS's `receive(ANY, &m)` call returns. Who the message came from is in the field `m.source`. With `find_block`, FS finds that the requested data is in block 120.
4. **Message 2.** FS cannot access the disk directly either. With `send(DISK, &block)` it sends the second message to the driver and waits for the reply with `receive(DISK, &data)`.
5. **Message 3.** The driver reads the block with the I/O the kernel permits and sends the data to FS in the third message. FS prepares the reply with `m.result = data`.
6. **Message 4.** `send(m.source, &m)` carries the reply to the client; the client becomes ready.
7. **FS returns to the loop.** FS starts waiting for a new job in `receive(ANY, &m)` at the top of the `while` loop.
8. **The client continues.** The client's `receive` call returns; the data is in the field `m.result`.

Four messages were sent for a single read. Each message involves at least one user → kernel and one kernel → user transition. In a monolithic kernel, the same job finishes inside a single system call, with two mode switches and a few function calls.

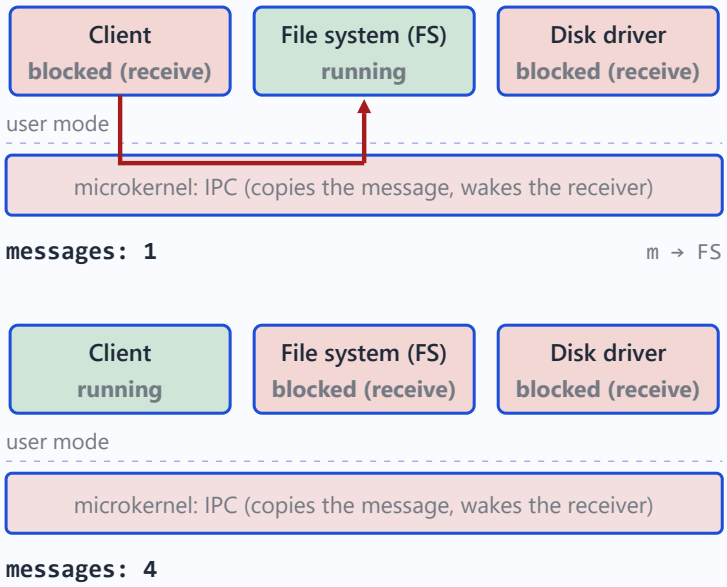


Figure 2.30: Two moments of Example 2.12. In the first frame, the first message: the client has sent its request and is waiting for the reply, and FS has woken up. In the second frame, the result: four messages have been completed and the client has received the data.

Common mistake

Measuring the overhead of a microkernel only by the number of messages. Besides two mode switches, each message usually needs a context switch too. The processor moves from the client process to the server process, and the server's address space becomes active. The data in the caches also becomes largely useless. The real source of the overhead is often these context switches.

2.6.7 MINIX 3 and the client–server model

MINIX 3 is a microkernel operating system developed by Andrew Tanenbaum and his team that puts reliability in the foreground. Figure 2.31 shows its structure.

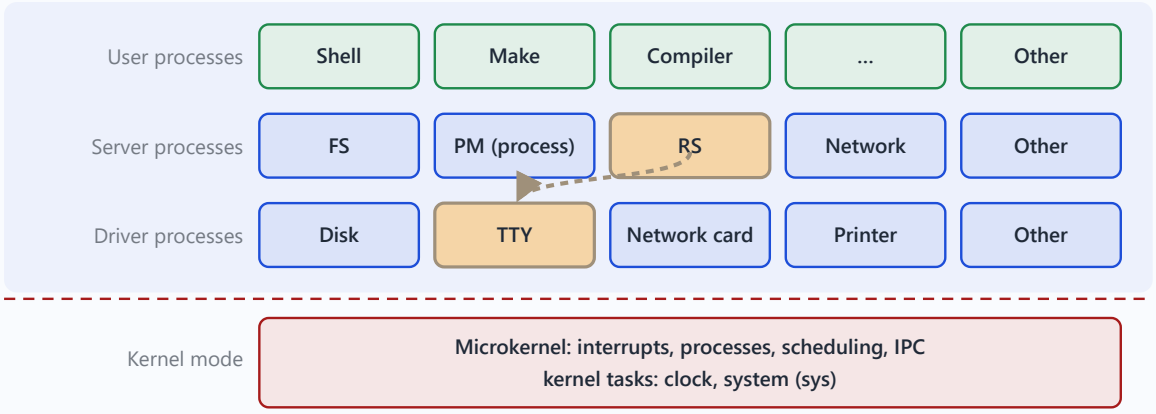
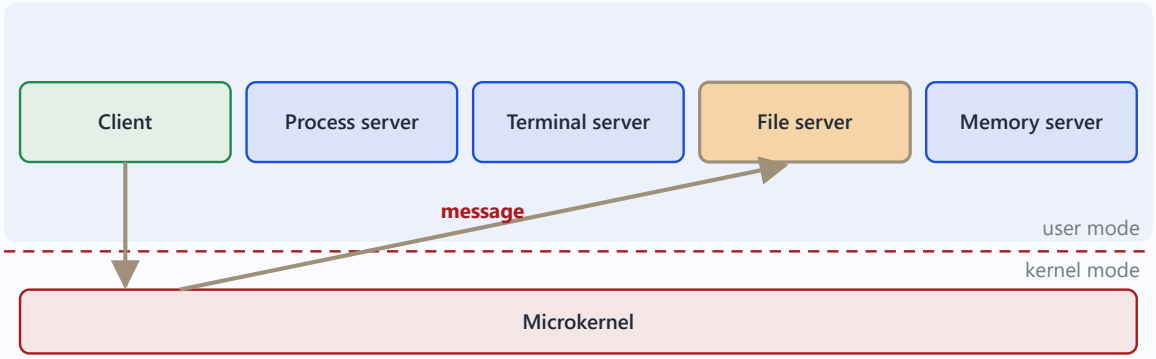


Figure 2.31: The structure of MINIX 3. User processes (shell, make, compiler), server processes (FS, the process manager PM, the reincarnation server RS, network), and driver processes (disk, TTY, network card, printer) are in user mode. In kernel mode there are only the microkernel, which handles interrupts, processes, scheduling, and IPC, and the clock and system tasks. The dashed arrow shows RS restarting the TTY driver.

Two features of MINIX 3 stand out. The first is the **reincarnation server** (RS). RS constantly checks the health of the drivers. If a driver, for example the terminal driver (TTY), crashes or stops responding, RS restarts it without the user noticing. The system repairs itself. The second is **split privilege**. Drivers run in user mode and can reach the I/O ports only through kernel calls, to the extent they are allowed. A faulty driver can corrupt its own memory, but it cannot touch the memory of the kernel or of another process.

It is natural to divide the processes running on a microkernel into two classes: the **clients** that request service and the **servers** that provide it. Communication takes place only by messages. This arrangement is called the **client–server model**.



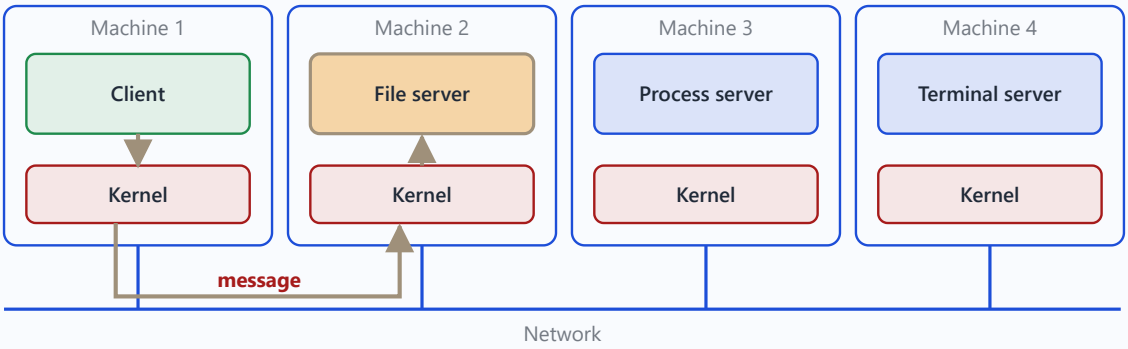


Figure 2.32: The client–server model. At the top, on a single machine the client and the file, process, and terminal servers run on a microkernel and messages pass through the kernel. At the bottom, each process is on a separate machine; the client's message goes over the network to the file server's machine.

The model has an important consequence. The client does not have to know whether the message is handled on its own machine or on another machine in the network: it sends a message and waits for the reply. For this reason the model extends naturally to **distributed systems**. Today's web applications and microservice architectures are large-scale applications of the same idea. Even so, message passing is not secure in itself. Security depends on the kernel's checking which process may send messages to which server.

2.6.8 The Tanenbaum–Torvalds debate

In 1991 Linus Torvalds released the first version of the Linux kernel. A few months later, in early 1992, MINIX's author Andrew Tanenbaum wrote a message in a newsgroup titled “LINUX is obsolete”. This message started one of the most frequently cited debates in the history of operating systems.

Tanenbaum's claim was this: the monolithic design is an outdated approach of the 1970s. A microkernel is more modular, more flexible, and easier to port; a bug in one component does limited damage. As hardware gets faster, the overhead of message passing will become negligible. Torvalds replied as follows: the monolithic kernel is simpler to design and implement, and direct function calls are faster. There is no need for message passing between processes; what really matters is code that works in practice. Besides, the open-source development model would make Linux develop quickly.

In the thirty years since, both sides have turned out partly right. Linux stayed monolithic and became one of the most widespread kernels in the world. But for the sake of flexibility it also borrowed ideas from microkernels. Loadable modules, and file systems (FUSE) and drivers (VFIO) running in user space, are examples. eBPF, which makes it possible to load small programs into the kernel safely, points in the same direction. Microkernels, in turn, have become widespread in safety-critical and embedded systems (QNX, seL4), where reliability matters more than performance.

2.6.9 Modular structure

Most of today's operating systems follow a modular approach, to gain flexibility while keeping the speed of the monolithic kernel.

Definition 2.13

In a **modular structure**, the kernel holds only the core components; other services are added to the kernel at boot or at run time as **loadable kernel modules** (LKMs). Modules run in kernel mode and in the kernel's address space.

Linux, FreeBSD, and Solaris use this structure. The modular structure resembles two separate structures but differs from both. It resembles the layered structure, because each module has a defined interface; it is more flexible than that, because any module can call any other. It resembles the microkernel, because the core of the kernel is small and the services are in separate pieces; it differs from it in that the modules communicate by direct function calls without using messages and run in kernel mode.

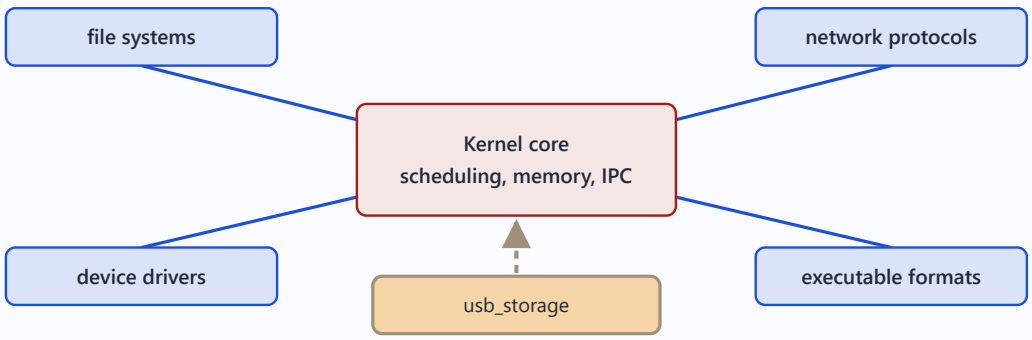


Figure 2.33: Loadable kernel modules. The kernel core in the middle holds scheduling, memory, and IPC; file systems, network protocols, device drivers, and executable file formats are added as modules. When a USB drive is plugged in, the `usb_storage` module is loaded; the `vfat` module is added when the drive is mounted.

The life cycle of a module can be followed on Linux as follows. At boot, the kernel and the necessary modules in the initial RAM disk (initramfs) are loaded; the `lsmod` command lists the loaded modules. The user plugs in a USB drive. The kernel generates an event. The `udev` service sees this event and has the `usb-storage` module loaded, together with its dependencies, with `modprobe`. The `vfat` module needed to read the FAT file system on the drive, however, is loaded not at this time but when the drive is mounted (when the `mount` call asks for the file system type). Once loaded, a module registers itself with the kernel and from then on works through direct function calls. When no one is using the module any longer, that is, when its use count drops to 0, the `modprobe -r` or `rmmod` command removes it from memory.

The price of the modular structure is the lack of isolation. Because a module runs in kernel mode, a faulty module can crash the kernel; a malicious module can take over complete control of the system. Moreover, modules must be compiled to be compatible with the version of the kernel they are loaded into.

Common mistake

Counting Linux as a microkernel because it has loadable modules. What defines a microkernel is not that components are separate pieces, but that they run as separate processes in user mode and communicate by messages. Linux modules run in kernel mode, in the kernel's address space, and communicate by function calls. Linux is therefore a modular monolithic kernel.

2.6.10 Hybrid kernels

Some systems set out with microkernel ideas and, out of performance concerns, moved many services back into kernel mode. These systems are called **hybrid kernels**. Windows and macOS are the two best-known examples.

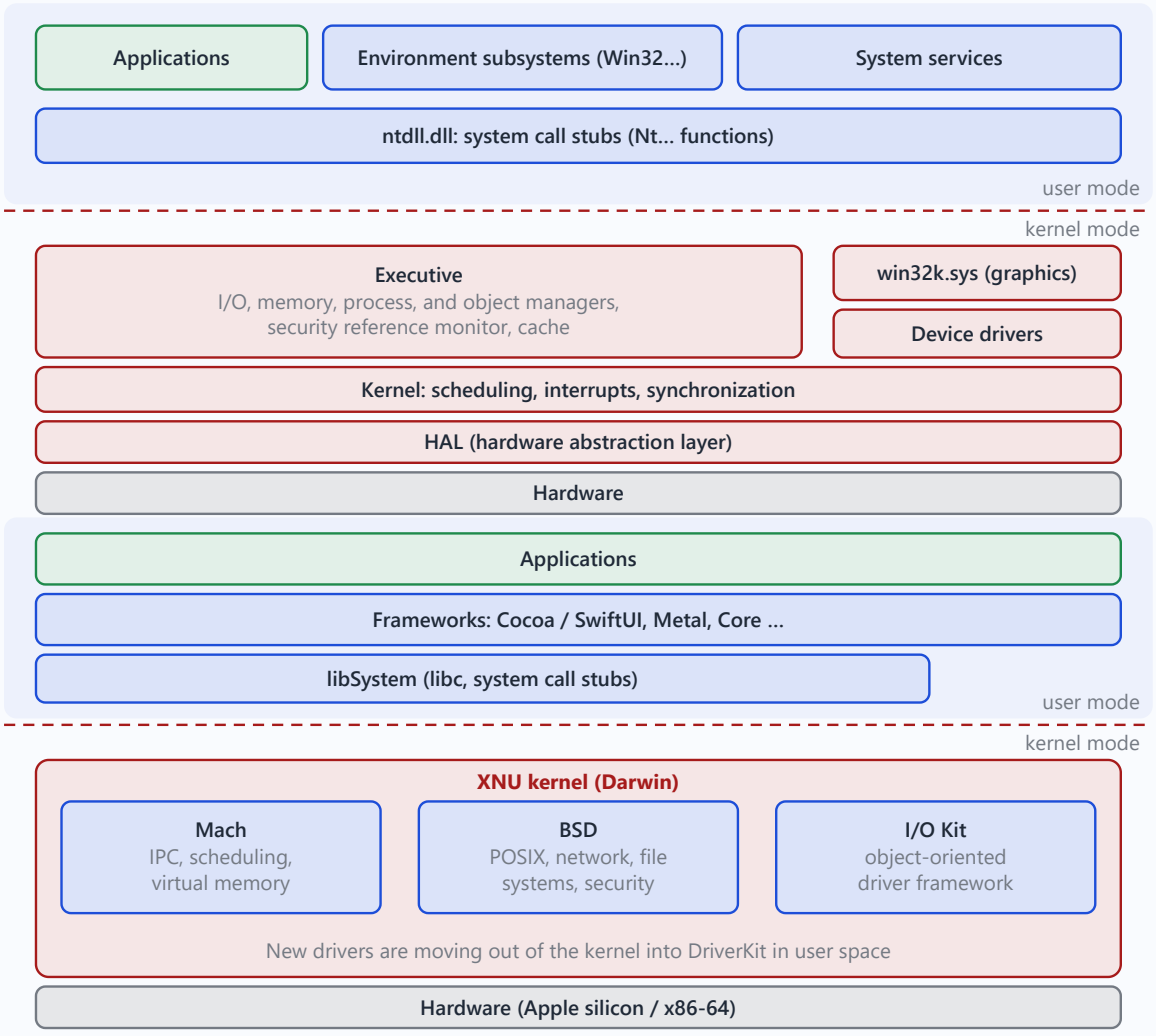


Figure 2.34: Two hybrid kernels. At the top, Windows NT: in user mode, applications, environment subsystems, system services, and `ntdll.dll`; in kernel mode, the executive, `win32k.sys`, device drivers, the kernel, and

the HAL. At the bottom, the XNU kernel of macOS and iOS: Mach, BSD, and I/O Kit are merged in the same kernel address space.

The design of Windows NT bears traces of microkernel ideas: subsystems, an object- and message-based structure, and a layer that abstracts the hardware (the hardware abstraction layer, HAL). But for performance, the executive, which includes the I/O, memory, process, and object managers, the graphics subsystem, and the device drivers were moved into kernel mode. XNU, the kernel of macOS and iOS, combines the Mach microkernel and a BSD UNIX layer in the same kernel address space. Mach provides IPC, scheduling, and virtual memory; BSD provides the POSIX interface, networking, and file systems; I/O Kit provides the object-oriented driver framework. The aim is to bring the modularity of the microkernel together with the speed of the monolithic kernel. In recent years Apple has been moving new drivers out of the kernel into the DriverKit framework in user space.

2.6.11 The exokernel

At the far end of the spectrum is the exokernel. This approach is a research architecture proposed in 1995 by Dawson Engler and Frans Kaashoek at MIT. The kernel is called Aegis, and the library operating system that works with it is ExOS. The exokernel is not a commercial operating system but an idea.

Definition 2.14

An **exokernel** is a kernel that only distributes hardware resources in a protected way and tracks which resource belongs to whom. Abstractions such as files and virtual memory are provided not by the kernel but by **library operating systems** (LibOS) that run in user mode together with the applications.

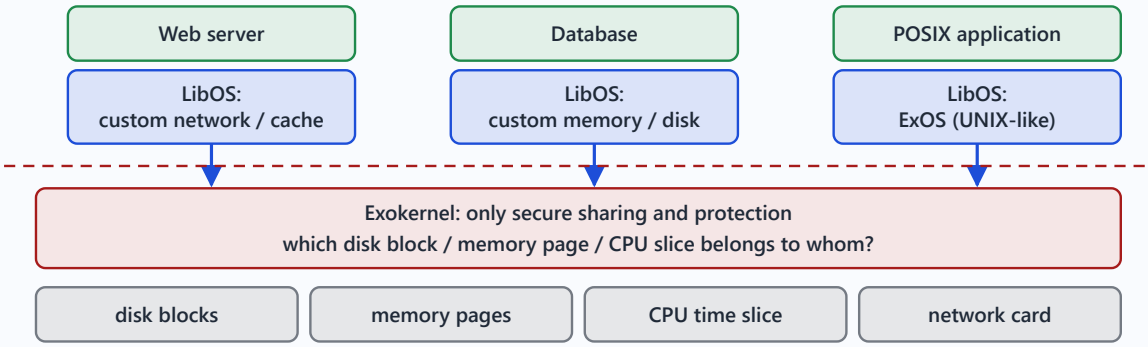


Figure 2.35: Exokernel structure. The web server, the database, and the POSIX application run with their own LibOSes (custom network and cache, custom memory and disk, UNIX-like ExOS). The exokernel only tracks to whom disk blocks, memory pages, processor time slices, and the network card belong.

Comparing the exokernel with the virtual machine approach makes the idea easier to grasp. A virtual machine monitor gives each virtual machine a copy of the whole machine. The virtual machine thinks it starts at block 0 of its own disk; the monitor has to map this block to a block on the real disk. An exokernel, on the other hand, gives each application a part of the real resource and only checks ownership. The application knows which real disk blocks it has received; no remapping layer in between is needed.

The slogan of the approach is **to separate management from protection**. The kernel only protects: an application cannot touch someone else's block or page. How the resources are managed is left to the application. A database can set up its own caching and disk layout strategy, and a web server its own network buffers. This application-specific management gives high performance; its price is that application development becomes more complicated. The exokernel idea lives on today in two forms. The first is **unikernels**, which compile a single application together with its own small operating system. The second is libraries such as DPDK and SPDK, which bypass the kernel in network and storage I/O and reach the device from user space.

2.6.12 Comparing the structures

Structure	What is in the kernel?	Strength	Weakness	Example
Simple	everything, unrestricted	small, simple	no protection	MS-DOS
Monolithic	all services	speed	no fault isolation	UNIX, Linux
Layered	hierarchical layers	easy to verify	overhead, layers hard to define	THE, MULTICS
Microkernel	IPC, scheduling, memory	reliability	message overhead	MINIX 3, QNX, seL4
Modular	kernel core + modules	flexibility, speed	module = full privilege	Linux, FreeBSD
Hybrid	microkernel + selected services	balance	complexity	Windows, macOS
Exokernel	protection only	application-specific speed	hard to develop for	Aegis/ExOS

Table 2.8: Comparison of operating system structures.

The table gives the structures in their pure forms. Real systems are seldom pure. Linux is monolithic and modular; Windows and macOS are hybrid; nearly all are layered to some degree. Instead of placing a system in a single box, it is more instructive to ask which decisions it made and why.

2.7 Virtual machines and multiprocessor systems

Kernel structures describe how an operating system organizes its own inside. The two topics of this section concern the hardware beneath the operating system: the duplication of hardware in software, and several processors sharing the same memory.

2.7.1 Virtual machines

Definition 2.15

A **hypervisor**, or virtual machine monitor (VMM), is a layer of software that creates **virtual machines** (VMs) on a single piece of physical hardware, each looking like an independent computer. The operating system running inside a virtual machine is called a **guest operating system**.

Each guest operating system believes it has its own processor, memory, disk, and I/O devices. For this reason different operating systems can run on the same hardware at the same time: on the same server, Linux can run in one virtual machine and Windows in another. Figure 2.36 compares a nonvirtual machine with virtual machines.

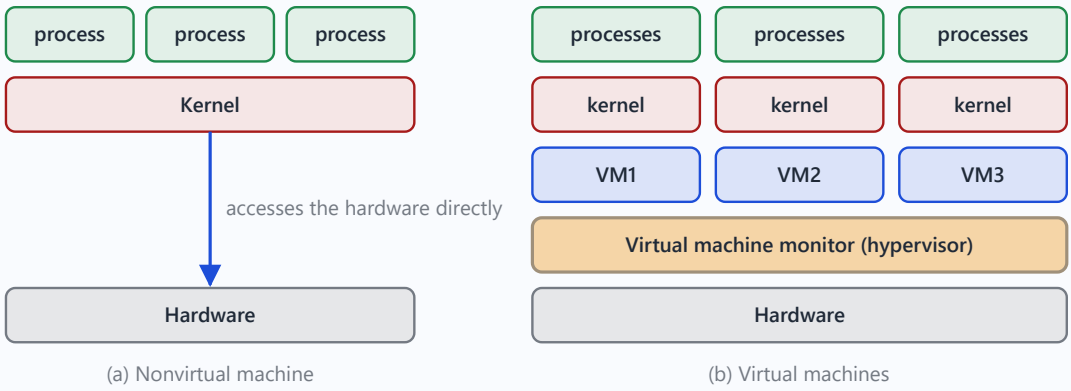


Figure 2.36: (a) A nonvirtual machine: processes run on a single kernel, and the kernel accesses the hardware directly. (b) Virtual machines: the hypervisor creates three virtual machines (VM1, VM2, VM3) on top of the hardware; each has its own kernel and its own processes.

The idea is not new; it goes back to IBM's VM/370 system, released in 1972. Today it is the foundation of cloud computing. The main benefits of virtualization are these:

- **Server consolidation.** Many lightly used servers are gathered on a single powerful machine.
- **Isolation.** A problem in one virtual machine does not spread to the others.
- **Snapshots.** The current state of a virtual machine is saved, and if needed one can return to this state.
- **Live migration.** A running virtual machine is moved to another physical machine without being shut down.

Hypervisors fall into two types according to where they run. A **Type 1** hypervisor runs directly on the hardware; there is no other operating system beneath it. The hypervisors used in data centers are of this kind. A **Type 2** hypervisor runs like an application on top of an ordinary operating system; the

programs that run virtual machines on a desktop computer are of this kind.

A concept often confused with the virtual machine is the **container**. A container does not carry its own kernel. It isolates a group of processes that share the same kernel from other processes through the kernel's namespaces and resource-limiting mechanisms. A container is therefore much lighter than a virtual machine and can be started in under a second. But because it shares the same kernel, only programs compatible with that kernel can run inside it, and its isolation is not as strong as a virtual machine's.

Common mistake

Thinking that an exokernel and a hypervisor are the same thing. Both share resources among several parties, but in different ways. A hypervisor gives each virtual machine a complete copy of the machine and maps virtual resources to real resources itself. An exokernel distributes pieces of the real resources and only checks ownership; the application's LibOS builds the abstractions.

2.7.2 Symmetric multiprocessing and multicore architecture

Almost every computer today has several processors or processor cores. This affects the structure of the operating system too.

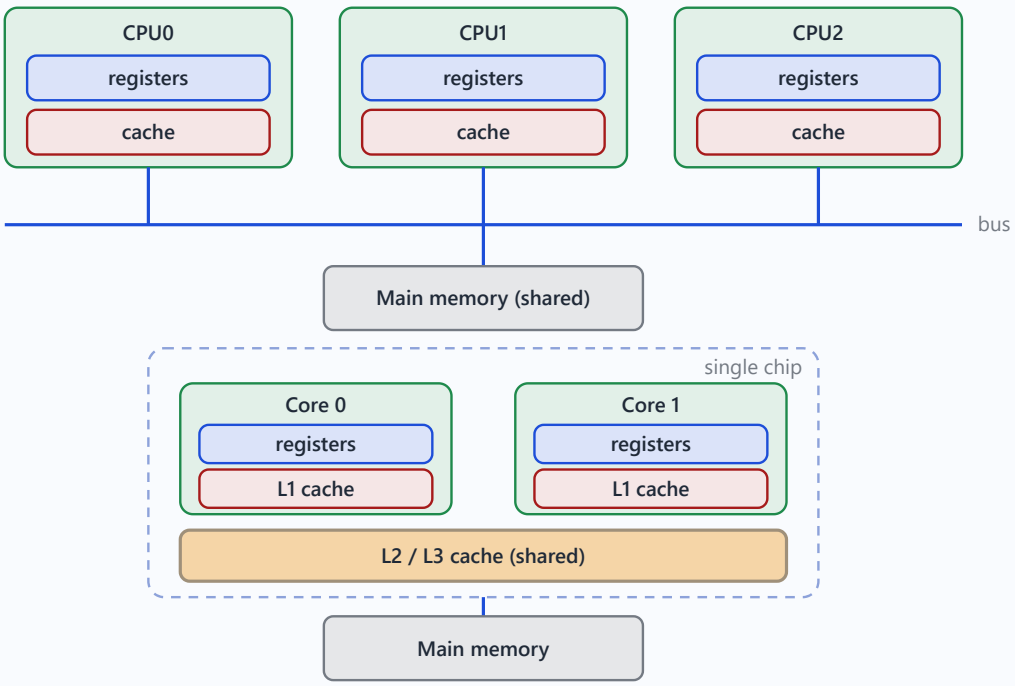


Figure 2.37: Top, the symmetric multiprocessor (SMP) architecture: each processor has its own registers and cache, and main memory is shared over a bus. Bottom, a multicore chip: two cores are on the same chip; each has its own L1 cache, and the L2/L3 cache is shared.

In a **symmetric multiprocessor** (SMP) system, each processor has its own registers and cache, but main memory is common. The word “symmetric” says that all processors have equal rights: all of them can do every job, including running operating system code. This has an important consequence. Because two processors can run kernel code at the same time, the kernel's shared data structures must be protected with locks.

Example 2.13

The kernel has a linked list of ready processes, and the pointer `head` points to the head of the list. At the start the list consists only of task A. Two processors add tasks to the list at the same time: CPU0 adds task X and CPU1 adds task Y. Insertion is two steps: (1) `t->next = head;` (2) `head = t;`. What happens if the steps are executed in the order CPU0(1), CPU1(1), CPU0(2), CPU1(2)?

Solution. CPU0(1): `X.next = A`. CPU1(1): `Y.next = A`. CPU0(2): `head = X`; the list becomes X → A. CPU1(2): `head = Y`; the list becomes Y → A. Task X has vanished from the list: it cannot be reached from any pointer and is never scheduled. The result depends on the order in which the steps are executed; this is called a **race condition**. The solution is to protect the insertion with a lock: acquire the lock, do (1) and (2), release the lock.

The first multiprocessor Linux versions solved this problem in a crude way: they protected the whole kernel with a single lock, the “big kernel lock”. Only one processor at a time could run kernel code. Over the years this lock was replaced by finer-grained locks and was removed completely in 2011.

In a **multicore** chip, several cores are on the same chip. Because the cores share nearby caches, they communicate faster and with less energy than processors on separate chips. A design that is common in 2026 is to have two kinds of cores on the same chip: performance (P) cores that are fast but consume energy, and efficiency (E) cores that are slow but frugal. This design gives the scheduler a new task: deciding which job runs on which kind of core.

2.8 Linking, loading, and booting

The last two topics are how a program and an operating system start running. The source code the programmer writes goes through several stages before it becomes a process. In the same way, there is a series of steps from the moment the power button is pressed until the login screen appears.

2.8.1 Compiling, linking, and loading

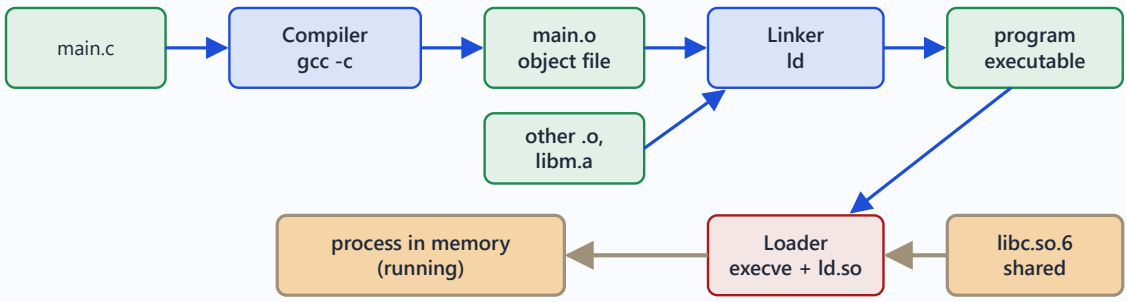


Figure 2.38: The compile, link, and load pipeline. `main.c` is translated by the compiler (`gcc -c`) into the object file `main.o` . The linker (`ld`) combines this with other object files and static libraries (`libm.a`) and produces the executable program. The loader (`execve` and `ld.so`) brings the program into memory together with the shared `libc.so.6` , and the process starts running.

1. **Compiling.** The compiler translates the source code into a **relocatable object file**. The addresses in this file are not yet final, because where the code will be placed in memory is not known. Symbols that the program uses but that are defined in another file, for example `printf` , are also unresolved.
2. **Linking.** The **linker** combines the object files and static libraries. It finds in which file each symbol is defined, fixes the addresses, and produces an **executable file**.
3. **Loading.** The shell calls `execve` . The **loader** in the kernel reads the file's header and maps the code and data segments into the process's address space. If the program uses shared libraries, it starts the dynamic linker (`ld-linux.so`).
4. **Dynamic linking. Shared libraries**, for example `libc.so.6` , are linked at run time. The library's code stays in memory as a single copy, and all processes that use it share this copy.

Example 2.14

On a server, 50 different programs that use the C library are running at the same time, and the library's code takes 3 MiB. How much memory is needed for the library code if the library is linked statically, and if it is linked dynamically?

Solution. In static linking, the library code that is used is copied into each executable file. Each process loads its own copy into memory: 50 × 3 MiB = 150 MiB is needed. In dynamic linking, because the library's code pages are read-only, a single copy is mapped into the address spaces of all processes: about 3 MiB is enough. Each process additionally uses a small private area for the library's writable data. Dynamic linking has one more benefit: when a security hole in the library is closed, only `libc.so` is updated. With static linking, all 50 programs would have to be relinked.

The format of the executable file is specific to the operating system: Linux uses ELF (Executable and Linkable Format), Windows uses PE (Portable Executable), and macOS uses the Mach-O format. This is one reason a program cannot be moved directly from one operating system to another. Even on the same processor, one system's executable file is in a format that the other system's loader does not understand, and the system calls of the two systems differ too.

2.8.2 The boot process

> **Definition 2.16**
>
> **Booting** is the process from the moment power is applied to the computer until the kernel is loaded into memory and starts the first user process.

On old PCs, booting was handled by the BIOS (firmware running in 16-bit real mode), in use since 1981. The BIOS loaded the master boot record (MBR) in the first sector of the disk into memory and ran it. Today's computers use UEFI. UEFI loads not a sector but a file (a program with the `.efi` extension). Figure 2.39 shows the boot steps of a Linux computer that uses UEFI.

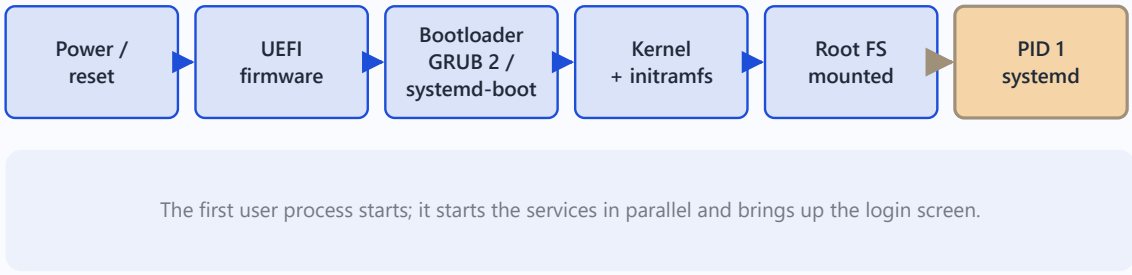


Figure 2.39: The booting of the operating system: power-on and reset, the UEFI firmware, the bootloader (GRUB 2 or systemd-boot), the kernel and initramfs, mounting of the root file system, and PID 1 (`systemd`).

1. **Power and reset.** When power is applied, the processor is reset and starts executing the firmware that sits in flash memory on the motherboard, from a fixed address determined by the hardware. At this moment there is no operating system in memory yet.
2. **UEFI.** The firmware first performs the power-on self-test (POST): it tests memory and finds devices. Then it looks at the boot order recorded in its persistent settings (NVRAM) and tries, in turn, the NVMe disk, USB, or network boot (PXE) options. It loads the bootloader from the EFI system partition (a FAT32 file system) of the GPT-formatted disk. If Secure Boot is on, the bootloader's digital signature is verified; an unsigned or modified bootloader is not run.
3. **The bootloader.** The **bootloader**, GRUB 2 or systemd-boot, presents the user with a menu. It places the selected kernel and the initial RAM disk (initramfs) into memory and hands control to the kernel.

4. **The kernel.** The kernel starts memory management, the scheduler, and the drivers. The scripts in initramfs load the modules needed to reach the real root file system (for example, disk controller or encrypted disk support).
5. **The root file system.** The real root file system is mounted, and initramfs, no longer needed, is released from memory.
6. **PID 1.** The kernel starts the first user process, `systemd`, whose PID is 1. `systemd` starts the services, the network, and the login screen as much in parallel as possible.

PID 1 has a special place. Every process created in the system from then on is its descendant; it is the root of the process tree. Processes whose parent has died are also adopted by PID 1, which collects their exit statuses. On Linux, the `systemd-analyze` command shows how long each stage of booting took. On Windows, the counterpart of the same chain is `winload.efi` → `ntoskrnl.exe` → `smss.exe` → `winlogon.exe`.

2.9 Summary

- The **operating system structure** is the strategy for where the components run (kernel or user mode) and how they communicate (function call or message). Each structure chooses a trade-off among performance, reliability, and flexibility.
- The operating system offers the user and programs the services of an interface, program execution, I/O, a file system, communication, and error detection; and the system as a whole the services of resource allocation, accounting, protection, and security. The services are reached through system calls.
- A **process** is a running program; its information is kept in the PCB in the process table. Processes form a tree through `fork()`. The **address space** consists of the text, data, BSS, heap, and stack segments.
- **Files** live in a directory tree; open files are referred to by small integers (descriptors), and a new file gets the smallest free number. **Mounting** attaches a file system to a directory in the tree. A **pipe** is a one-way byte channel connecting two processes.
- **Dual-mode operation** reserves privileged instructions for kernel mode; the **timer** makes sure the processor always returns to the operating system. The kernel is entered only by system call, interrupt, and exception.
- In a **system call**, the libc wrapper puts the call number and parameters into registers and executes the trap instruction; the kernel checks the number and finds the handler in the system call table. On error the call returns −1 and `errno` is set.
- `printf` is a library function and makes the `write` system call when it flushes its buffer. The buffer size determines the number of system calls made.
- The shell works with the `fork`, `execve`, and `waitpid` trio. `link` adds a new name to an inode, and `unlink` removes a name; a file disappears only when both the link count and the open count are 0.

- A **monolithic** kernel is fast but has no fault isolation. A **layered** structure makes verification easier but adds overhead. A **microkernel** is reliable but carries message overhead. A **modular** structure combines flexibility with speed, and **hybrid** kernels blend these ideas. An **exokernel** only protects and leaves management to the application.
- A **hypervisor** virtualizes the hardware and runs several operating systems on the same machine; a **container** isolates processes that share the same kernel. In SMP systems, kernel data structures must be protected with locks.
- Source code becomes a process by passing through the compiler, the linker, and the loader. At boot, the firmware, the bootloader, and the kernel run in turn, and the kernel starts PID 1, the first user process.

2.10 Key terms

operating system services process process control block (PCB) parent / child process address space interprocess communication (IPC) file descriptor path name block / character device mount pipe deadlock resource allocation graph dual-mode operation kernel mode / user mode trap timer system call application programming interface shell i-node hard link monolithic kernel layered system microkernel message passing policy / mechanism client–server model loadable kernel module (LKM) hybrid kernel exokernel virtual machine (VM) hypervisor guest operating system container symmetric multiprocessor (SMP) race condition executable file shared library booting bootloader

---

2.11 Test Yourself

1. In a newly started process only 0, 1, and 2 are open. The following run in order: `a = open("x", O_RDONLY)`, `b = open("y", O_RDONLY)`, `close(a)`, `c = open("z", O_RDONLY)`, `close(2)`, and `d = dup(b)`. What are `a`, `b`, `c`, and `d`?

   Answer

   Each call gives the smallest free number. `a = 3`, `b = 4`. `close(a)` frees 3; `c = 3`. `close(2)` closes standard error and the smallest free number becomes 2; `d = dup(4) = 2`. Result: 3, 4, 3, 2. The error messages in this process are now directed to file `y`; but since the file was opened read-only, an attempt to write there fails.

2. The copy code traced in these notes copies a 9000-byte file with a 4096-byte buffer. How many `read` and how many `write` calls are made, and what does each `read` return? How many system

calls are made in total, together with `open`, `creat`, the two `close` calls, and `printf`?

Answer

`read` returns 4096, 4096, 808, and 0 in turn: 4 `read` calls. Because the last call reports end of file, no `write` follows it: 3 `write` calls. The total is 1 + 1 + 4 + 3 + 2 + 1 = 12 system calls. The program prints `9000 bytes`.

3. In the call `write(1, "Günaydın\n", n)`, what must `n` be for the whole text to be written? What appears on the screen if `n` is given as the number of characters?

Answer

The text has 9 characters (G, ü, n, a, y, d, ı, n, \n). Because ü and ı take two bytes each in UTF-8, the number of bytes is 11; `n = 11` must be given. If `n = 9` is given, the last two bytes, that is, the final letter “n” and the newline, are not sent, and “Günaydı” appears on the screen; the cursor stays on the same line.

4. On x86-64 Linux, in which registers are the six parameters and the call number of `mmap(addr, length, prot, flags, fd, offset)` placed? Why is the fourth parameter not placed in `rcx`?

Answer

The call number goes into `rax`; `addr` into `rdi`, `length` into `rsi`, `prot` into `rdx`, `flags` into `r10`, `fd` into `r8`, and `offset` into `r9`. Because the `syscall` instruction saves the return address in `rcx` and the flags register in `r11`, a parameter in `rcx` would be erased before reaching the kernel; that is why `r10` is used instead of the `rcx` of the C convention.

5. A timer's counter decreases by 1 on every tick of a 1 MHz clock. The kernel wants 100 ticks per second. What value must be loaded into the counter? If handling each tick interrupt takes 5 µs, what percentage of the processor's time goes to ticks?

Answer

1,000,000 / 100 = 10,000; 10,000 is loaded into the counter and the time between two ticks is 10 ms. The overhead is 100 × 5 µs = 500 µs per second; this is 0.05% of the processor's time.

6. A process creates a file named `f` (link count 1). Then `link("f", "g")`, `link("g", "h")`, `unlink("f")`, `fd = open("h", O_RDONLY)`, `unlink("g")`, and `unlink("h")` are called. What is

the link count after each step? When are the file's blocks freed?

Answer

The link count becomes 2, 3, 2, 2, 1, and 0 in turn. `link("g", "h")` adds a third name to the inode that `g` points to, that is, to the same file. `open` does not change the link count; it makes the open count 1. After the last `unlink` the file has no name left, but because the descriptor is open the blocks are not freed. The blocks are freed only when `close(fd)` is called (or the process terminates).

7. Suppose the `perror` and `_exit(127)` lines in the child have been deleted from the simple shell skeleton: the child does not call `exit` after `execve`. What happens if the user types a nonexistent command such as `lss`?

Answer

`execve` cannot find the program and returns −1. The child, running the shell's code, leaves the `else` block and returns to the top of the `while` loop: it prints the prompt and starts waiting for a command. The parent shell, meanwhile, is still waiting in `waitpid` for this child to finish. Without noticing, the user talks to a second shell; only when that "shell" is exited does the parent wake up.

8. In a microkernel system, the file block the client wants is found in the file system server's cache. With how many messages and at least how many mode switches does the read complete? How many mode switches occur in a monolithic kernel?

Answer

There is no need to go to the disk driver. Two messages suffice: the client's request and the server's reply. Because each message needs one user → kernel and one kernel → user transition, there are at least 4 mode switches. In a monolithic kernel the read is a single system call: one transition on entry and one on return, 2 in total.

9. Which of the following operations are completed with library code alone, and which need a system call: `strlen`, `getpid`, `sqrt`, `fork`, `printf("a")` (no newline, output on the terminal)?

Answer

`strlen` and `sqrt` work only with the process's own data; they are library code. `getpid` and `fork` touch process information in the kernel; they are system calls. `printf("a")` is a library function, and because it is line-buffered on a terminal, it does not enter the kernel before a newline arrives; the "a" waits in the buffer. When the buffer is flushed later (at a newline, `fflush`, or the end of the program), `write` is called.

10. Suppose a driver corrupts memory through a faulty pointer. What can this bug harm in MINIX 3, in Linux (with the driver as a loadable module), and in an exokernel-based system (with the driver in the application's LibOS)?

    Answer

    In MINIX 3 the driver is a separate process in user mode; it can corrupt only its own memory, it crashes, and the reincarnation server restarts it. In Linux the module runs in kernel mode in the kernel's address space; it can corrupt any structure of the kernel and crash the whole system. In an exokernel the LibOS is in the same address space as the application; the bug corrupts that application, but because the exokernel checks ownership, it cannot touch the resources of the other applications.

2.12 Further reading

- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th ed., 2018), Chapter 2: covers operating system services, system calls, the linker and loader, structures, and booting in the order these notes follow, with plenty of figures.
- A. S. Tanenbaum and H. Bos, *Modern Operating Systems* (4th ed., 2014), Sections 1.5–1.7: treats the concepts of process, address space, and file, the POSIX calls, the eleven steps of `read`, and the monolithic, layered, microkernel, client–server, and exokernel structures in detail.
- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (version 1.10), the chapters “Process API” and “Limited Direct Execution”: explain the `fork` / `exec` / `wait` trio and the protection based on traps and the timer with short, clear examples; free and online.
- D. R. Engler, M. F. Kaashoek, and J. O'Toole, “Exokernel: An Operating System Architecture for Application-Level Resource Management” (SOSP 1995): the original paper that put forward the exokernel idea; it defends the principle of separating management from protection with measurements on Aegis and ExOS.

2.13 References

1. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, version 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
2. D. R. Engler, M. F. Kaashoek, and J. O'Toole, “Exokernel: An Operating System Architecture for Application-Level Resource Management”, *SOSP*, 1995.
3. Linux man pages: `syscalls(2)`, `syscall(2)`, `fork(2)`, `pipe(7)`, `mount(8)`.
4. A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapters 2 and 18).
5. A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014 (Sections 1.5–1.7 and 7).

PART II

Process Management

Chapters 3–7

PART II · CHAPTER 3

Processes

A computer typically runs a browser, a music player, a compiler, and dozens of background services all at once. The operating system sees each of them as a separate **process**. Every process has its own memory, its own register values, and its own open files. The process is the basic unit of the operating system's division of labor: the processor is handed out to processes, memory is allocated to processes, and files are opened on behalf of processes. These notes begin with the question of what a process is and how it is laid out in memory. They then use code traces to show how the UNIX calls `fork`, `exec`, `wait`, and `exit` create and end processes. Process states, the process control block, queues, and context switching show how the kernel keeps track of processes. The last sections cover how processes communicate with each other and why mobile systems restrict background processes.

LEARNING OUTCOMES

Process model
Explain the difference between a program and a process and describe the memory layout of a process; say in which region each variable of a C program lives.

Life cycle
Define the process states and the transitions between them; explain the contents and purpose of the process control block (PCB).

UNIX API
Trace small programs that use `fork()`, `exec()`, `wait()`, `exit()`, and `pipe()` line by line; determine their output and the number of processes they create.

Scheduling infrastructure
Explain and compute with the ready and wait queues, the effect of multiprogramming on CPU utilization, and the cost of a context switch.

IPC
Compare shared memory, message passing, pipes, and sockets in terms of speed, coordination, and typical use.

Mobile multitasking
Explain why and how iOS and Android restrict background processes.

3.1 The process concept

The operating system hides the difficulty of using hardware directly behind a few abstractions. A file is an abstraction of disk blocks; an address space is an abstraction of physical memory. The **process** is the abstraction of the processor. Thanks to the process concept, even a single processor can be used as if every program had a processor of its own. This section defines the process, shows how it looks in memory, and explains how many processes make progress at the same time.

3.1.1 Program and process

A file such as `a.out` or `vim` sitting on disk is a **program**. A program is an **executable file** made up of machine instructions and initial data. By itself it does nothing; it waits on disk. A program is therefore a *passive* entity.

When a program is run, the operating system allocates memory for it, loads its instructions into memory, and tells the processor "start here." From that moment on, there is a running activity. It has a **program counter** (the address of the next instruction), register values, a stack, open files, and an identification number. This live entity is called a **process**.

Definition 3.1

A **process** is an instance of a program in execution. A process has three components: an address space holding the program's code and data, the current state of the processor (the program counter and registers), and resources. Resources are the open files, memory regions, and identity information that the operating system allocates to the process. Every process carries a unique **process identifier** (PID) within the system.

The relationship between a program and a process resembles the relationship between a recipe and a dish cooking in the kitchen. The recipe sits on paper and does not change. Cooking is a process: which step you are at, what is in the pot, whether the stove is on all change from moment to moment. Two dishes can be cooked from the same recipe in two different kitchens at the same time, and neither affects the other.

The same holds for programs. When `vim` is opened in two terminal windows, a single file on disk gives rise to two processes. Each has its own address space, its own open files, and its own cursor position. Text typed in one does not appear in the other. The read-only code of the program, however, can be shared between the two processes; keeping two copies of the same machine instructions in memory would be wasteful. Figure 3.1 shows this situation.

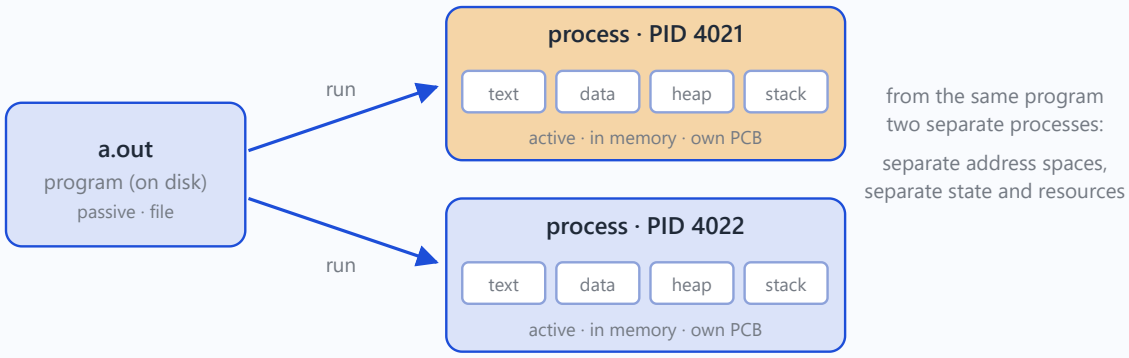


Figure 3.1: When the single program `a.out` on disk is run twice, two separate processes result (PIDs 4021 and 4022). Each process has its own text, data, heap, and stack regions and its own PCB.

Inside a process, instructions can run in a single sequence or in several sequences. A single-**threaded** process has one program counter and executes its instructions one after another. A multithreaded process has several program counters. Its threads share the same address space and, on a multicore machine, can truly make progress at the same time. Unless stated otherwise, the processes in these notes are single-threaded.

Common mistake

Using the words “program” and “process” interchangeably. A program is a file and consumes no resources by itself. A process consumes memory, processor time, and open files. A program may never run, may run once, or may run a hundred times at once; each run is a separate process.

3.1.2 Process layout in memory

Every process sees memory as if it were one contiguous array of its own. The addresses of this array make up the **address space**. Extending from 0 up to the highest address, this space is divided into several regions. Each region has its own purpose and its own access rights. Figure 3.2 shows the regions from bottom to top.

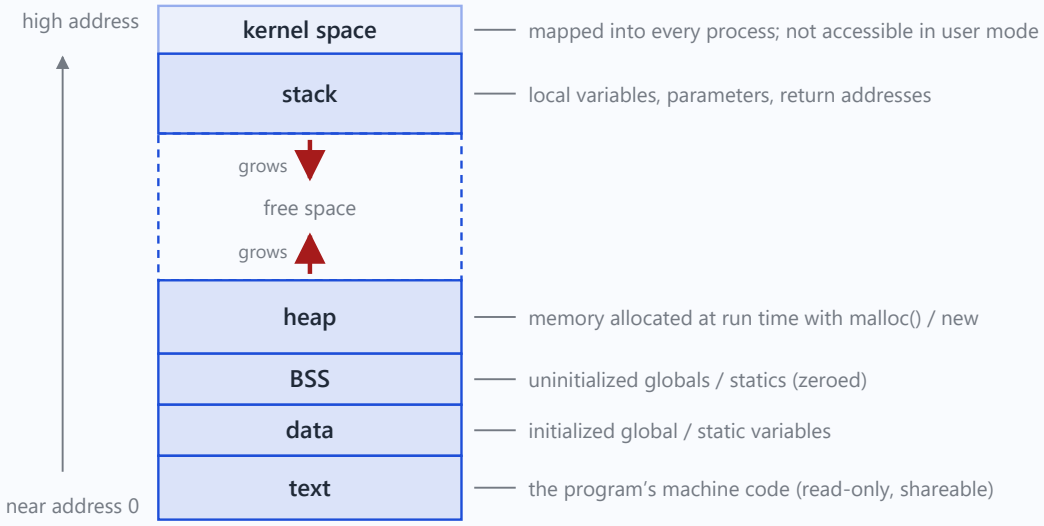


Figure 3.2: The memory layout of a process. At low addresses, the text, data, and BSS regions have fixed sizes. The heap grows upward and the stack grows downward; the free space between them leaves room for both to grow. The kernel space at the top is mapped into every process but is not accessible in user mode.

Let us take the regions one by one, starting from the lowest address:

- The **text** region holds the program's compiled machine code. It is read-only: a process cannot accidentally modify its own code. Processes created from the same program can share this region.
- The **data** region holds initialized global and `static` variables. These values are stored inside the executable file and copied into memory at load time.
- The **BSS** region holds uninitialized global and `static` variables. The name comes from an old assembler directive (*block started by symbol*). All of these variables start at zero. The file records only the size of the region; the loader allocates that much zeroed memory. A 4 MB array of zeros therefore does not make the executable 4 MB larger.
- The **heap** satisfies memory requested at run time with `malloc()` (or `new` in C++ and Java). As the program asks for memory, the heap grows upward; space released with `free()` is reused.
- The **stack** tracks function calls. Each call pushes a **frame** onto the stack: the parameters, local variables, and return address live in this frame. When the function returns, the frame is popped. The stack grows downward from high addresses.
- The **kernel space** is the topmost part of the address space. The kernel's code and data are mapped here, so the address space need not be switched during a system call. Code running in user mode cannot touch this region; if it tries, the hardware raises a fault.

That the heap and the stack grow toward each other is a deliberate design. It is not known in advance how much each region will grow: a recursive program uses a lot of stack, while a program that processes large data uses a lot of heap. Because they share the free space between them, memory can be divided flexibly between the two.

The program counter does not appear in this picture because it is a register inside the processor, not something in memory. The program counter points to an address in the text region: it is the process's "current position." The **stack pointer**, which points to the top of the stack, is also a register. When the process leaves the processor, the kernel saves these registers somewhere so that their values are not lost. That place is the process control block, discussed in 3.4.4.

In real systems the addresses of the regions are not the same on every run. **Address space layout randomization** (ASLR) shifts the starting addresses of the stack, the heap, and the shared libraries randomly on each run. Because an attacker cannot predict what is where in memory, many attacks become harder. The order of the regions and their directions of growth do not change.

3.1.3 Where do variables live?

The best way to make the regions concrete is to find where each name in a small C program lands. The program below is the same as the one in the slides:

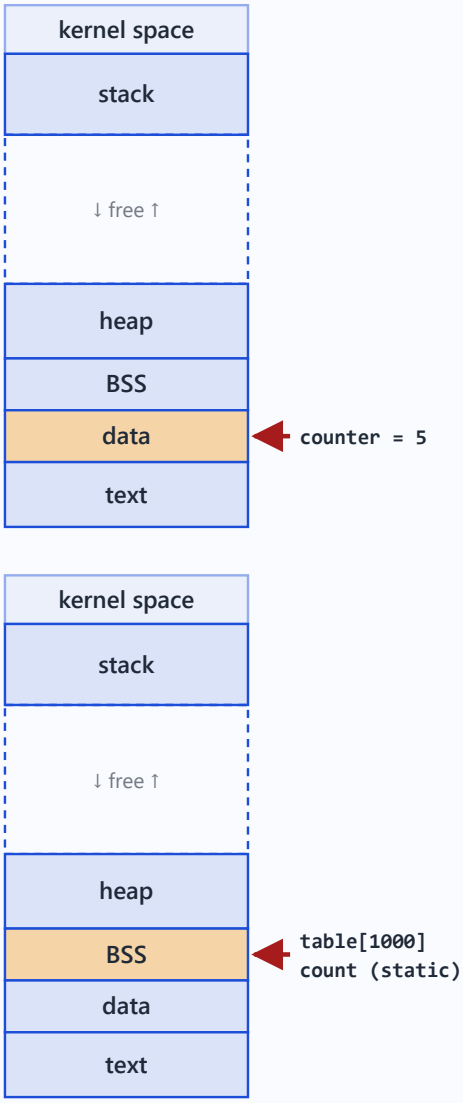
```c
#include <stdlib.h>
int counter = 5;    /* data */
int table[1000];  /* bss */

int main(int argc, char *argv[]) {
    int local = 3;
    static int count;
    int *p = malloc(4 * sizeof *p);
    p[0] = local + counter;
    free(p);
    return 0;
}
```

Let us find where each element of the program lives, one by one. Figure 3.3 shows the four regions other than text side by side.

Example 3.1

1. **Text.** The compiled instructions of the `main` function are in the text region. The code of `malloc` and `free` also sits in the text region of the C library. This region is read-only and can be shared among processes of the same program.
2. **Data.** `int counter = 5;` is global and initialized. The value 5 is stored in the executable file and copied into the data region at load time.
3. **BSS.** `int table[1000];` is global but uninitialized; its 4000 bytes start at zero and take no space in the file. The `static int count;` inside `main` is in BSS as well. Being declared inside a function does not put it on the stack; the keyword `static` makes the variable live for the whole program run.

4. **Stack.** When `main` is called, a frame is opened on the stack. `argc`, `argv`, the local variable `local = 3`, the pointer `p` itself, and the return address are in this frame. When `main` returns, the frame disappears.
5. **Heap.** `malloc(4 * sizeof *p)` allocates $4 \cdot 4 = 16$ bytes from the heap (assuming a 4-byte `int`). `p` is on the stack, but the 16 bytes it points to are on the heap. The line `p[0] = local + counter` writes $3 + 5 = 8$ into the first cell on the heap. `free(p)` returns this space; the variable `p` itself stays on the stack.



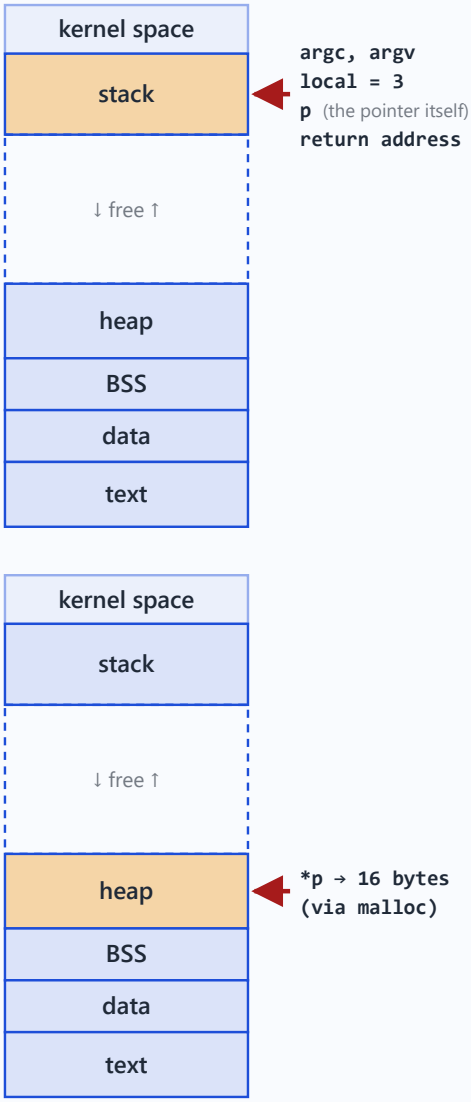


Figure 3.3: Where the names in the example program live. (a) Data: `counter = 5`. (b) BSS: `table[1000]` and `static count`. (c) Stack: the frame of `main`; `argc`, `argv`, `local`, `p`, and the return address. (d) Heap: the 16 bytes allocated by `malloc`.

On Linux this layout can also be seen from the command line. The command `size a.out` prints the sizes of the text, data, and BSS regions. All regions of a running process are listed by `cat /proc/PID/maps`. In that listing you see lines named `[heap]` and `[stack]` along with the ranges where the program and its libraries are mapped.

Common mistake

Saying “the pointer lives on the heap.” In the line `int *p = malloc(16);` there are two separate pieces of memory. The variable `p` is local, so it is on the stack and holds an 8-byte address. The 16 bytes that this address points to are on the heap. When the function returns, `p` disappears, but if `free` was not called, the space on the heap remains allocated. This is exactly what a memory leak is.

3.1.4 Pseudoparallelism and multiprogramming

A single-core processor can execute only one instruction stream at any instant. Even so, a user can listen to music and write a document at the same time. This is possible because the processor switches among processes very quickly. Each process runs for a few milliseconds, and then the processor moves on to the next. Because there are hundreds of switches per second, the user believes that all the processes run at once. This illusion is called **pseudoparallelism**.

> **Definition 3.2**
>
> **Multiprogramming** is keeping several processes in memory at the same time and sharing the processor among them. The goal is not to leave the processor idle while a process waits (for example, for data to arrive from disk) but to keep it busy with another process. The **scheduler** decides which process gets the processor.

Figure 3.4 shows this situation from three viewpoints. Part (a) shows the physical reality: a single program counter moves in turn among processes A, B, C, and D. Part (b) is the conceptual model: each process is thought of as an independent activity with its own logical program counter. When a process leaves the processor, the value of the physical program counter is saved into that process's logical counter. Part (c) is the time axis: over a long interval all four processes have made progress, but at any instant only one of them runs.

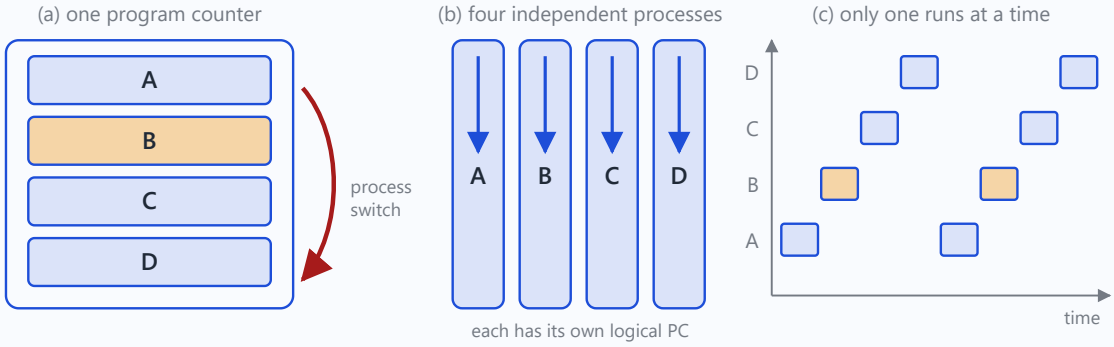


Figure 3.4: Multiprogramming. (a) A single program counter switches among four processes. (b) Conceptually there are four independent processes, each with its own logical program counter. (c) On the time axis every process makes progress, but only one runs at any instant.

The length that determines how often switches occur is called the **time slice** (or **quantum**). Typical values are a few milliseconds. In EEVDF (Earliest Eligible Virtual Deadline First), the scheduler of current Linux kernels, the base slice is about 3 ms. In the older Completely Fair Scheduler (CFS), the target time for every ready process to run once was between 6 and 24 ms, depending on the number of cores.

Pseudoparallelism is convenient for the user but carries a warning for the programmer: it cannot be known in advance when a process will be interrupted. An interruption can occur in the middle of an instruction sequence, between two lines, even between the read and the write of a variable. The speed of a process is not constant either; the same loop may take 2 ms in one run and 20 ms in another. A correct program therefore makes no assumptions about timing. Thoughts such as “the child process has surely finished by now” or “these two lines run without interruption” can be guaranteed only by an explicit coordination mechanism.

3.1.5 Concurrency and parallelism

Two words that are used interchangeably in everyday language describe different things in operating systems.

Definition 3.3

Concurrency is the progress of several tasks in the same time interval. Tasks can run interleaved on a single core; at any instant only one may be running. **Parallelism** is the physical execution of several tasks at the same time and requires several cores or processors.

Figure 3.5 shows the difference clearly. In the top row, a single core runs tasks A, B, and C piece by piece in turn: all three tasks progress in the same interval, but at no moment do two run together. The lower part has two cores; A and C run at the same time. A parallel system is always concurrent, but a concurrent system need not be parallel.

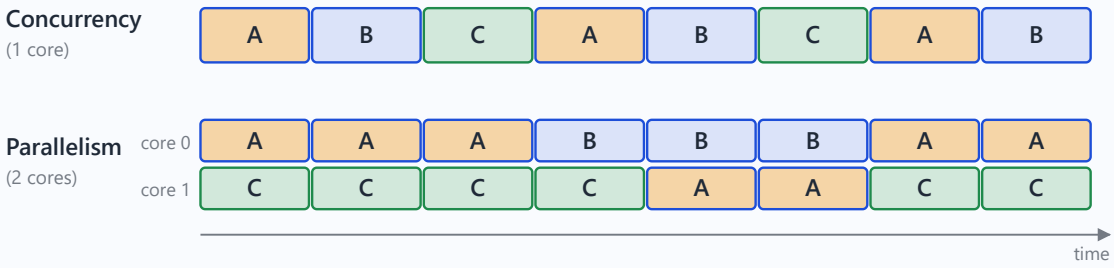


Figure 3.5: Concurrency on one core and parallelism on two cores. At the top the tasks interleave; at the bottom two cores run different tasks at the same time.

Multicore processors are now everywhere; as of 2026 even phones have 6 to 10 cores. This does not mean that switching is no longer needed. Even an ordinary laptop has hundreds of processes, and the number of cores is far below that. Multiprogramming and switching between processes therefore live on in every system, whatever the number of cores.

3.1.6 Example: Chrome's multiprocess architecture

One of the examples that best shows the practical value of the process concept is the Chrome browser. Older browsers ran as a single process. A faulty page or plugin in one tab could crash the entire browser. Chrome instead splits the work among many processes (Figure 3.6).

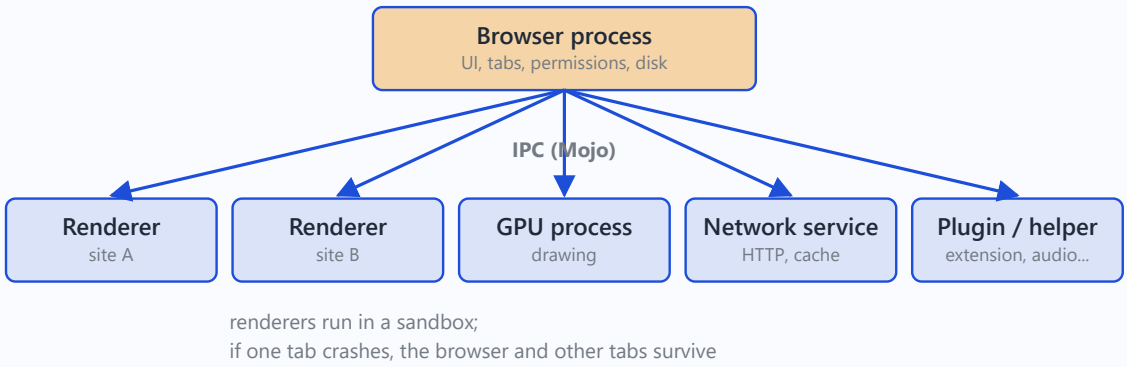


Figure 3.6: Chrome's multiprocess architecture. The privileged browser process, the sandboxed renderer processes, and the GPU, network, and helper processes communicate through an IPC system called Mojo.

- The **browser process** is unique and privileged. It manages the user interface, the tabs, disk access, and permissions.
- The **renderer** processes handle HTML, CSS, and JavaScript. Thanks to **site isolation**, each site runs in a separate renderer process. These processes run in a **sandbox**: they cannot access the disk or the network directly and pass every request on to the browser process.
- The **GPU, network, and helper processes** run drawing, network traffic, and extensions in separate processes. Legacy plugin support has been removed: NPAPI plugins in 2015, Flash at the end of 2020.

This design has three benefits. The first is *stability*: if a tab crashes, only its renderer process dies, and the browser and the other tabs keep working. The second is *security*: even if a malicious page takes over its own renderer process, it cannot escape the sandbox or read the data of other sites. The third is *parallelism*: different tabs can run in parallel on different cores. The price is memory; every process carries its own address space and data structures. A Chrome with many tabs therefore uses more memory than a single-process browser.

3.2 Process creation

Every process in the system is born from another process. This section describes the events and system calls through which processes are created. The emphasis is on the two basic UNIX calls: `fork()`, which makes a copy of the process, and `exec()`, which replaces the program in the process. Used together, they also explain how the shell runs a command and how it redirects output to a file.

3.2.1 Events and system calls that create processes

A new process comes into existence through four kinds of events:

1. **System boot.** After the kernel is loaded, it starts the first user process. On Linux this is usually `systemd`, which in turn starts the services that run continuously in the background. These services, which do not interact directly with the user, are called **daemons**; the web server, the print service, and the logger are of this kind.
2. **A request from a running process.** A process can create new processes to divide its work. For example, a build tool starts a separate compiler process for each source file.
3. **A user starting a command or an application.** Typing `ls` in the shell or clicking an icon on the desktop creates a new process.
4. **Starting a batch job.** On large servers and compute clusters, queued jobs are started as processes when resources become free.

These four cases look different from the outside, but technically the same thing happens: an existing process makes a system call that creates a new process. Even at boot, the kernel itself prepares the first process; every later process is born from a call by an earlier one.

System	Call
UNIX / Linux	`fork()` + `exec()`; also `posix_spawn()`, and `clone()` on Linux
Windows	`CreateProcess()`: creates the process and loads the program in a single call

Table 3.1: System calls that create processes.

UNIX splits the job into two steps. `fork()` creates a copy of the calling process; `exec()` replaces the program inside a process with another program. Windows' `CreateProcess()` does both at once: it creates the new process and loads the given program into it. `posix_spawn()` offers a one-step route on UNIX as well. Linux's `clone()` call lets the caller choose individually what the new task shares with its parent (memory, files, signal handlers); threads are created with this call too.

The process that creates a process is called the **parent**, and the created process is called the **child**. A child can create children of its own. Every process carries a unique PID and knows the PID of its parent.

3.2.2 The process tree

Parent–child relationships form a tree. On Linux, the root of the tree is `systemd` (or another `init` program), which starts at boot with PID 1. `kthreadd`, with PID 2, is the parent of the kernel threads; in-kernel workers such as `kworker` sit under it. Figure 3.7 shows a slice of the tree of a typical desktop system. The PID values in the figure are examples; real values change on every boot.

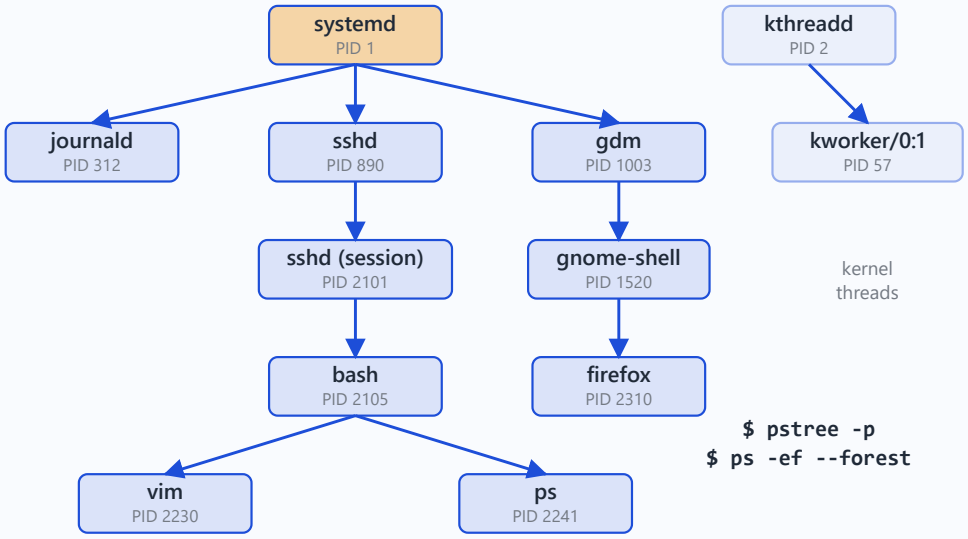


Figure 3.7: A slice of the Linux process tree. `systemd` (PID 1) is the root of the user processes and `kthreadd` (PID 2) is the root of the kernel threads. The tree can be seen with `pstree -p` or `ps -ef --forest`.

The trace of a shell session is easy to read in the tree. When a user connects over SSH, `sshd` (PID 890) creates a new `sshd` (PID 2101) for the session. That one starts the user's shell (`bash`, PID 2105). Every command typed in the shell, such as `vim` or `ps`, becomes a child of the shell. In the `ps` output, each process is listed with its parent's PID (PPID); this column gives the branches of the tree.

3.2.3 The parent–child relationship

When a new process is created, the operating system answers four questions:

- **Resource sharing:** The child may share all, some, or none of the parent's resources. In UNIX, the child inherits the parent's open file descriptors; it reaches the same files under the same numbers.
- **Execution:** The parent and child can run concurrently. Alternatively, the parent can wait for the child to finish with `wait()`.
- **Address space:** In UNIX, after `fork()` the child gets a *copy* of the parent's address space. They do not share the same memory. The child can then load a new program into itself with `exec()`.
- **Identifiers:** `getpid()` returns the process's own PID, and `getppid()` returns the PID of its parent.

Copying the address space looks expensive: if the parent has hundreds of megabytes of memory, is all of it copied? No. The child's address space is logically separate from the parent's, but the physical pages are shared until one of them writes. This technique is called **copy-on-write** and is the subject of a later subsection.

3.2.4 fork(): one call, two returns

`fork()` is unlike ordinary functions: it is called once but returns twice, once in the parent and once in the newly created child. Both processes continue from the line right after the call. The only way to tell which is which is the return value.

Definition 3.4

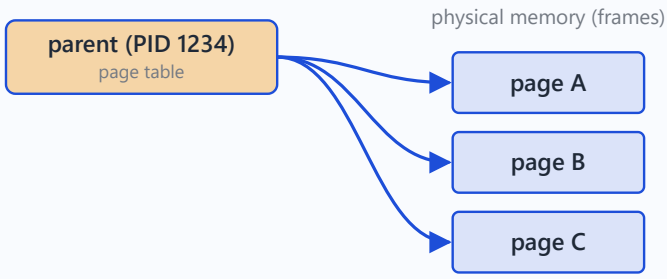
`pid_t fork(void)` creates a new process that is a copy of the calling process. On success it returns the child's PID (a positive number) to the parent and 0 to the child. On failure no child is created and −1 is returned to the parent only; the reason is stored in the variable `errno`.

These three cases determine the pattern written after every `fork`:

```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* child: fork returned 0 */ }
else                { /* parent: pid = child's PID */ }
```

Why does the child get 0 instead of its own PID? Because the child can always learn its own PID with `getpid()` and its parent's with `getppid()`. The parent, however, has no other way to learn the child's PID. A parent may have many children and needs their PIDs to wait for them or to send them signals. The value 0 is also a safe marker because no user process can have PID 0.

In the kernel, `fork` does the following: it allocates a new PCB and copies the parent's into it; assigns the new PID; copies the open file table; and copies the page table. It does *not* copy the pages themselves. Figure 3.8 shows how this works in three stages.



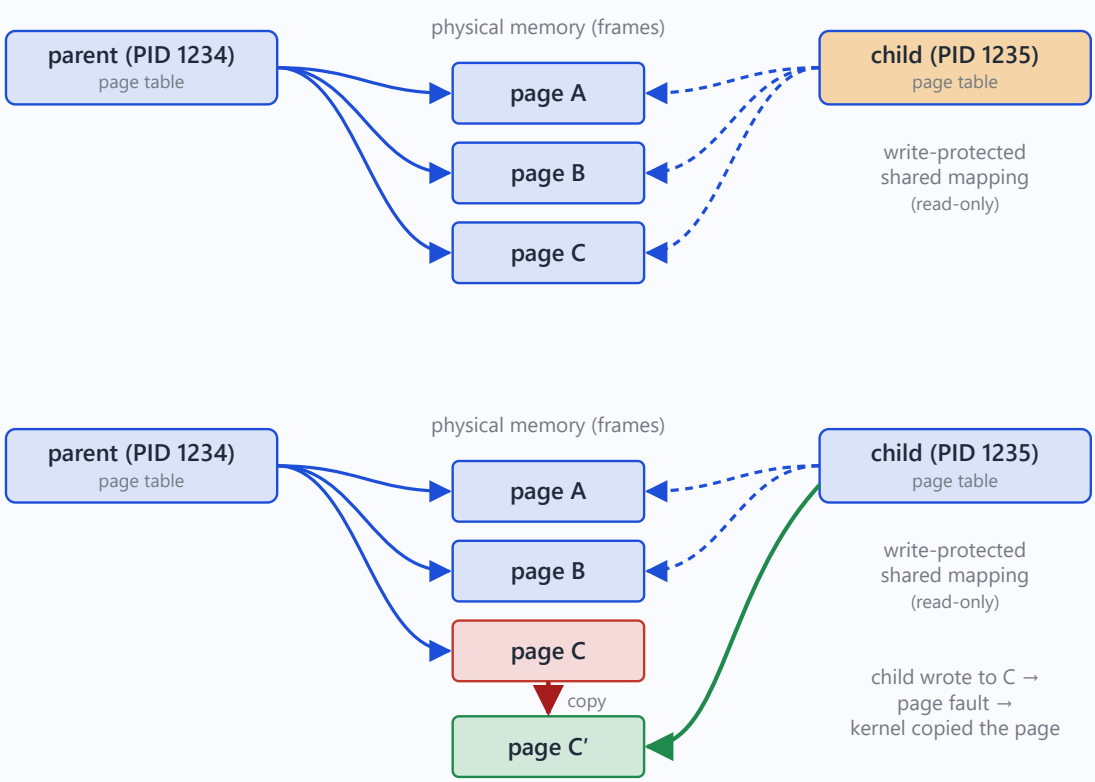


Figure 3.8: `fork()` and copy-on-write. (a) The parent's page table points to physical pages A, B, and C. (b) After `fork`, the child's page table points to the same pages; the pages are mapped read-only on both sides. (c) When the child writes to C, a page fault occurs; the kernel copies only that page (C′) and redirects the child's table to the new copy.

Definition 3.5

Copy-on-write (COW) means that two processes share the same physical page read-only, and the page is copied only when one of the processes tries to write to it. The write attempt raises a **page fault**; the kernel copies the page, points the writing process's page table at the new copy, and enables write permission.

Thanks to copy-on-write, the cost of `fork` is proportional to the size of the page table, not to the size of the address space. More importantly, if the child is going to call `exec` right away (which is almost always the case in a shell), almost no pages are copied. The pages that would be copied are discarded during `exec` anyway. The details of COW belong to virtual memory; here only its consequence matters: logically, the child is a complete copy of the parent, and neither side can change the other's variables.

Common mistake

Assuming that the parent and child share variables after `fork`. A global variable, an array on the heap, or a local variable on the stack becomes two separate copies at the moment of `fork`. An assignment made by the child does not affect the parent's copy. For processes to share data, you must explicitly set up shared memory, a pipe, or another IPC mechanism.

3.2.5 Code trace: fork() and wait()

The following program shows that after `fork` the two processes work with separate variables, and how `wait` fixes the order of the output:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    int x = 10;
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); exit(1); }
    if (pid == 0) {
        x = x + 5;
        printf("child: x=%d\n", x);
        return 0;
    }
    wait(NULL);
    printf("parent: x=%d\n", x);
    return 0;
}
```

The program starts with `x = 10` on line 7 and calls `fork` on line 8. If `fork` fails, it returns −1; in that case line 9 prints the error and ends the program. After that there are two processes, each with its own `x`. In the child, `pid == 0`, so the `if` block is entered; the parent skips both `if`s and reaches `wait(NULL)`. `wait` blocks the calling process until one of its children finishes and returns the PID of the finished child. The `NULL` argument says that the child's exit status is of no interest. Let the parent's PID be 1234 and the child's 1235. Table 3.2 gives the steps of one run.

Step	Line	Parent (1234)	Child (1235)	Output
1	7	running; `x = 10`	—	
2	8	`pid = 1235`, `x = 10`	created; `pid = 0`, `x = 10`	
3	15	`wait(NULL)` → waiting	ready	
4	10	waiting	running; `pid == 0` is true	
5	11	waiting; `x = 10`	`x = 15`	

Step	Line	Parent (1234)	Child (1235)	Output
6	12	waiting	`printf`	`child: x=15`
7	13	woken up (SIGCHLD)	`return 0` → zombie	
8	15	`wait` returns 1235	reaped (entry deleted)	
9	16	`printf` ; `x` is still 10	—	`parent: x=10`
10	17	terminated	—	

Table 3.2: Trace of the `fork()` + `wait()` program (parent 1234, child 1235).

Three points of the trace are important. First, in step 5 the child sets `x` to 15, but the parent's `x` stays at 10. There are two separate variables with the same name; in step 9 the parent prints 10. Second, the order of steps 3 and 4 in the table can vary. After `fork`, the child may run first; then the child prints and exits, and when the parent reaches `wait` the child has already finished, so `wait` returns immediately. In either order the output is the same:

```
child: x=15
parent: x=10
```

Third, in step 7, when the child returns from `main`, it does not vanish at once. The C runtime calls `exit(0)` with the return value of `main`. The kernel reclaims the child's memory but keeps its exit status until the parent reads it. During this short interval the child is a **zombie**. The kernel sends the parent a `SIGCHLD` signal and wakes it up. In step 8, `wait` collects the child's status and deletes its entry from the process table. This operation is called **reaping**. Zombies and reaping are treated in detail in 3.3.4.

3.2.6 Without wait(), the output order is undefined

What happens if `wait` is removed from the previous program? Consider the following short fragment:

```
pid_t pid = fork();
if (pid == 0) printf("child\n");
else          printf("parent\n");
```

After `fork`, both processes are ready, and the scheduler decides which runs first. The program has no way to influence this decision. Figure 3.9 shows two possible schedules on a single core: in one the parent prints first, in the other the child. On a multicore machine the two processes can even run at the same time. On Linux, which one runs first has varied with the version and the configuration; the programmer cannot rely on it.

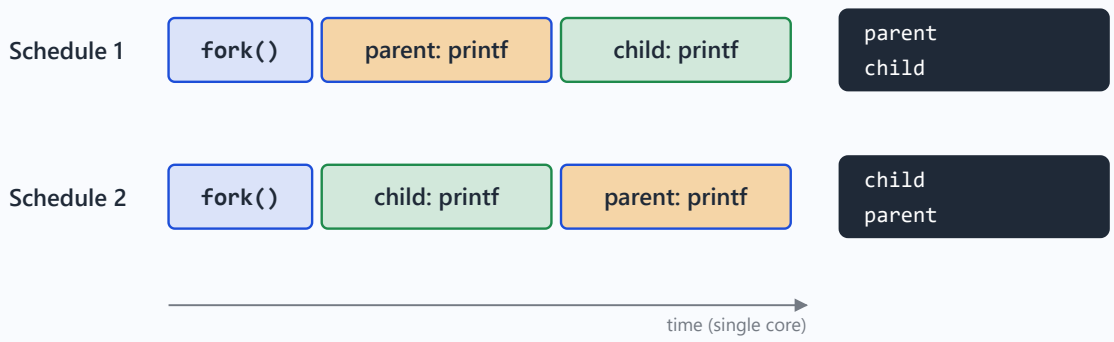


Figure 3.9: Two different schedules and two different output orders after `fork`. At the top the parent runs first, at the bottom the child; the terminal boxes show the output seen on screen.

If the correctness of the program depends on this order (for example, if the parent is going to read a file written by the child), the order must be established explicitly. The simplest way is `wait()` or `waitpid()`. For more complex cases there are pipes, semaphores, and other coordination tools. That the output order changes from run to run is the first sign of the most fundamental problem in concurrent programs: the result must not depend on timing that cannot be controlled.

3.2.7 How many processes are created? Buffered output and fork()

The behavior of `fork` that doubles the number of processes at every call is a staple of exam questions. Getting the rule right once makes even the most tangled-looking question easy. The rule is this: if k processes reach a `fork` line at that moment, there are $2k$ processes after the line. Each one produces a child, and the child continues from the same line.

```
fork();
fork();
printf("X\n");          /* how many times? */

for (int i = 0; i < 3; i++)
    fork();
printf("Y\n");          /* and this one? */
```

Example 3.2

1. **First fragment.** At the start there is a single process. After the first `fork` there are 2. Both run the second `fork`; then there are 4. All four reach `printf`: 4 lines of `X` are printed. The number of newly created processes is 3.
2. **Fragment with a loop.** The loop runs three times, for `i = 0`, 1, 2. In each round the number of processes doubles: 1 → 2 → 4 → 8. The children also carry a copy of the loop variable and continue the loop from where they were. In the end $2^3 = 8$ processes print `Y`; 7 of them are new.

3. **Generalization.** If n unconditional `fork` s run one after another, 2^n processes result and $2^n - 1$ of them are new. With conditional `fork` s (for example `if (fork() && fork())`) this rule cannot be applied directly; the return value must be followed separately for each process.

Finding the number of processes is not enough; sometimes the content of the output is surprising as well. Consider these three lines:

```
printf("A");
fork();
printf("B\n");
```

The answer that comes to mind is "one `A` and two `B` s." Yet the program prints two lines of `AB` . The reason is the C library's **output buffer**. `printf` does not send characters to the kernel immediately; it first writes them into a buffer in the process's own memory. The buffer is flushed only in certain situations, and its contents go to the kernel with the `write` system call. When writing to a terminal, the buffer is *line-buffered*: it is flushed when a newline (`\n`) arrives. When writing to a file or a pipe, it is *fully buffered*: it is flushed when it fills up or when the process exits.

The `"A"` in the first line contains no newline, so it is still in the buffer at the time of `fork` . Because the buffer lives in the process's memory, `fork` copies it too. Now both processes have `A` in their buffers. Each process appends `"B\n"` and prints `AB` with the newline. Whether the output goes to the terminal or is redirected to a file, `AB` appears twice.

If the first line were `printf("A\n")` , the result would depend on where the output goes. On a terminal, line buffering flushes `A` before the `fork` , and the output is `A` , `B` , `B` . If the output is redirected to a file with `./prog > output.txt` , full buffering keeps holding `A\n` . In that case both processes print `A` , `B` at exit; the file contains four lines: `A` , `B` , `A` , `B` .

Common mistake

Assuming that the text has been written to the screen or the file by the time `printf` returns. If the buffer is not flushed before `fork` or `exec` , unwritten data is either duplicated or lost. The safe way is to call `fflush(stdout)` before `fork` . For error messages, `stderr` is preferred because `stderr` is unbuffered.

3.2.8 The exec() family

`fork` produces only a copy; if the shell did nothing but copy itself on every command, it could never run `ls` . A second call is needed to run another program. The functions in the `exec` family replace the program inside the calling process with another program from disk.

Definition 3.6

The `exec` family replaces the calling process's memory image (code, data, heap, and stack) with a new image loaded from the given executable file. No new process is created: the PID, the parent relationship, and the open file descriptors that were not closed are preserved. A successful `exec` does not return to the calling code; if it does return, an error has occurred and the return value is −1.

Figure 3.10: `exec` replaces the process image. On the left, the code and data of `a.out` go away; in their place come the code of `ls`, new data, an empty heap, and a new stack carrying `argv`. The PID, PPID, open files, working directory, and identities are preserved.

The six members of the family do the same job; they differ only in how the arguments are passed. The letters in their names encode the difference. **l** (list): the arguments are given one by one, as a list ending with `NULL`. **v** (vector): the arguments are given as an array. **p** (path): the program is searched for in the directories of the `PATH` environment variable. **e** (environment): the environment variables are given explicitly.

Call	Arguments	PATH search	Environment
`execl` / `execv`	list / array	no	inherited
`execlp` / `execvp`	list / array	yes	inherited
`execle` / `execve`	list / array	no	given explicitly

Table 3.3: The `exec` family.

All of these functions are defined in the C library and end up in a single system call: `execve()`. For example, `execlp("ls", "ls", "-l", NULL)` first searches for `ls` in `PATH` and finds `/usr/bin/ls`. It then turns the arguments into the array `{"ls", "-l", NULL}` and calls `execve`. Repeating the program's name in the first argument is a convention: this value becomes `argv[0]` in the new program.

Common mistake

Forgetting to end the argument list with `NULL` or leaving out `argv[0]`. In the call `execlp("ls", "-l", NULL)`, `-l` is taken as the program's name (`argv[0]`) and `ls` runs without any options. If the `NULL` is forgotten, the library cannot find the end of the list and takes random values in memory for arguments.

3.2.9 What does exec do in the kernel?

From the outside, `exec` is a single step: the process is one program and then becomes another. Inside the kernel, six steps are carried out in order.

1. **Open the file; check permission and format.** If the file does not exist, lacks execute permission, or has an unrecognized format, `exec` returns −1 and the old program continues where it left off. Linux recognizes two formats: binaries that begin with an ELF header and scripts that begin with a `#!` line. For a script, the kernel runs the interpreter named after `#!` (for example `/bin/sh`) and passes the script's name to it as an argument.
2. **Discard the old address space.** The code, data, heap, and stack mappings are deleted; any other threads in the process are terminated. From this point on there is no program to return to. That is why all the checks are done in the first step.
3. **Close descriptors marked close-on-exec; reset signal handlers.** Files opened with `O_CLOEXEC` are closed; the others stay open. Caught signals revert to their default action, because the code of the handler function is no longer in memory. Ignored signals remain ignored.
4. **Map the code and data segments from the file.** Nothing is copied. When a page is used for the first time, a page fault occurs and that page is brought in from disk. Thanks to this **lazy loading**, the parts of a large program that are never used never come into memory.
5. **Allocate zeroed memory for BSS, heap, and stack.** At the top of the stack the kernel places the `argv` and `envp` strings, the arrays of pointers to them, and `argc`.
6. **Set the program counter to the entry point; return to user mode.** The entry point is not `main` but the C runtime's `_start` routine. `_start` sets up the runtime and calls `main(argc, argv, envp)`. When `main` returns, it calls `exit` with the return value. Throughout this whole operation the PID never changes.

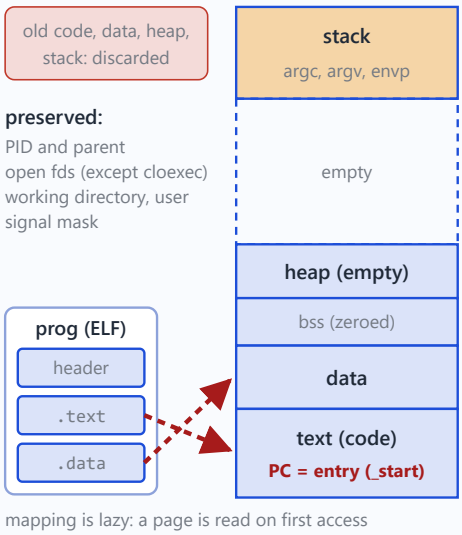


Figure 3.11: The new address space after `exec`. Text and data are mapped lazily from the file; BSS, heap, and stack come from zeroed memory; `argc`, `argv`, and `envp` sit at the top of the stack. The old code, data, heap, and stack are gone; the PID, parent, open descriptors (except close-on-exec ones), working directory, and user ID are preserved.

An important consequence follows from this order: the line after `exec` runs only if `exec` fails. There is therefore no need to write an `if (… == -1)` check after `exec`; if the following line is reached, there is already an error. The right pattern is to print the error and terminate the process right after `exec`.

3.2.10 Code trace: fork, exec, and waitpid

The following program is the complete pattern for running another program on UNIX: copy, replace the program in the child, wait in the parent.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid = fork();
  if (pid < 0) { perror("fork"); exit(1); }
  if (pid == 0) {
    execlp("ls", "ls", "-l", NULL);
    perror("execlp");  /* error */
    exit(1);
  }
  int st;
  if (waitpid(pid, &st, 0) < 0) { perror("waitpid"); exit(1); }
  if (WIFEXITED(st)) printf("done: %d\n", WEXITSTATUS(st));
  return 0;
}
```

Here `waitpid` is used instead of `wait`. `waitpid(pid, &st, 0)` waits for a specific child and writes its exit status into `st`. The third argument carries options; 0 means "wait until it finishes." `st` encodes not only the exit code but also how the process ended. `WIFEXITED(st)` tells whether it was a normal exit, `WEXITSTATUS(st)` gives the exit code, and `WIFSIGNALED(st)` tells whether it was killed by a signal. `WEXITSTATUS` is meaningful only if `WIFEXITED` is true; that is why line 16 tests it first. Lines 8 and 15 also catch the error case in which `fork` and `waitpid` return −1.

Step	Line	Parent (1234)	Child (1235)	Output
1	7	`pid = 1235`	`pid = 0`; the same `a.out` image	
2	15	`waitpid(1235, …)` → waiting	ready	
3	10	waiting	`execlp`: `ls` is found in `PATH`; the image becomes `/usr/bin/ls`, the PID is still 1235	
4	—	waiting	`ls -l` runs; writes to the `stdout` it inherited	(output of `ls -l`)
5	15	woken up; `waitpid` returns, `st` = encoded 0	`exit(0)` → reaped	
6	16	`WIFEXITED(st)` is true; prints `WEXITSTATUS(st) = 0`	—	`done: 0`

Table 3.4: Trace of fork + exec + waitpid (parent 1234, child 1235).

Step 3 is the heart of the program. Because `execlp` succeeds, lines 11 and 12 in the child never run: the code containing those lines is no longer in memory. From then on the child process is the `ls` program. `ls` does not know who started it; it simply writes to file descriptor 1. That descriptor was inherited from the parent and points to the terminal. This is why the listing appears on the screen.

If the file name were mistyped, for example `execlp("lss", …)`, `execlp` would return −1. The child would print the message `execlp: No such file or directory` with `perror` on line 11 and exit with `exit(1)`. The parent would then print `done: 1`. What if the `exit(1)` line were forgotten? The child would leave the `if` block and continue at line 14, executing the parent's code. Because `pid` is 0 in the child, `waitpid(0, …)` is called; since the child has nothing to wait for, it returns −1, and the child prints `waitpid: No child processes` and exits. Without error checking, the child might even print a second "done" line with a meaningless `st`. This is one of the most common mistakes in programs that use `fork`.

3.2.11 The shell loop: fork, exec, wait

The command-line shell is a program that runs the pattern of the previous subsection in an infinite loop. Each round of the loop consists of six steps. Figure 3.12 shows one round.

1. The shell (`bash`, PID 2105) shows the prompt and reads the command `ls` that the user typed.
2. It creates a copy of itself (PID 2300) with `fork()`. The parent waits with `waitpid(2300, …)`.
3. The child calls `execvp("ls", argv)`; its image is replaced by `ls`.
4. `ls` runs and writes to the terminal `stdout` it inherited.
5. `ls` exits. The kernel notifies the parent with `SIGCHLD` and `waitpid` reaps the child.
6. The shell shows the new prompt and the loop starts over.

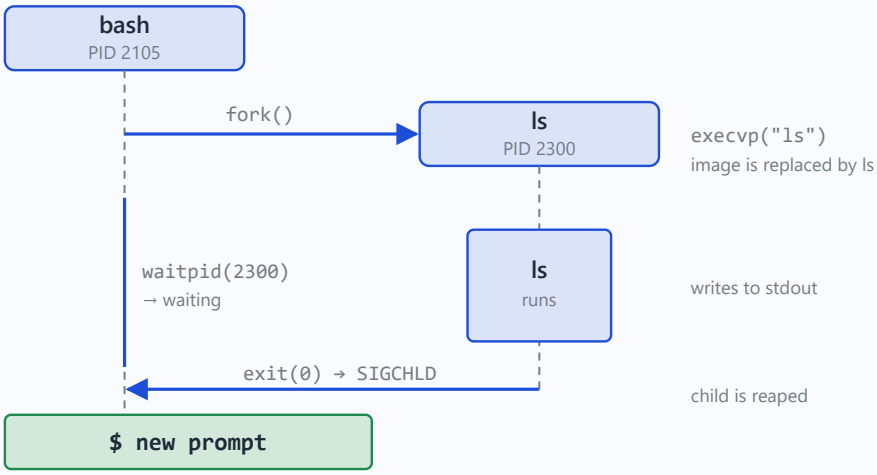


Figure 3.12: One round of the shell loop. `bash` forks, the child turns into `ls` and writes to the terminal, `SIGCHLD` arrives on exit, and the shell reaps the child and shows the new prompt.

If `&` is added to the end of the command (`sleep 60 &`), the shell does not call `waitpid` in step 2; it shows the new prompt immediately. This is a **background job**. A background child must also be reaped when it finishes. The shell does this when `SIGCHLD` arrives, or by collecting the waiting children before the next prompt.

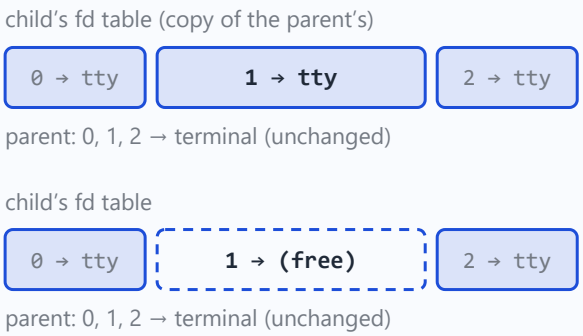
3.2.12 Why are fork and exec separate? Output redirection

While Windows makes do with a single `CreateProcess` call, why does UNIX use two calls? The answer lies in the short interval between them. After `fork` and before `exec`, the child is still running the shell's code. Meanwhile it can change its own environment as it likes: it can redirect file descriptors, change the working directory or the priority, and ignore some signals. Then it moves on to the new program with `exec`. The new program inherits these changes but knows nothing about them.

This is what happens when `wc -l notes.txt > count.txt` is typed in the shell. The following program does the same job:

```
int main(void) {
  pid_t pid = fork();
  if (pid == 0) {    /* child */
    close(1);        /* stdout freed */
    int fd = open("count.txt",
             O_WRONLY | O_CREAT | O_TRUNC, 0644);
    /* lowest free number: fd == 1 */
    execlp("wc", "wc", "-l", "notes.txt", NULL);
    perror("execlp");
    exit(1);
  }
  waitpid(pid, NULL, 0);
  printf("done\n");
  return 0;
}
```

The key to the program is that `open` always returns the **lowest free descriptor number**. The child first closes `stdout` with `close(1)`; number 1 becomes free. The `count.txt` opened right afterward therefore receives number 1. Figure 3.13 shows how the child's file descriptor table changes.



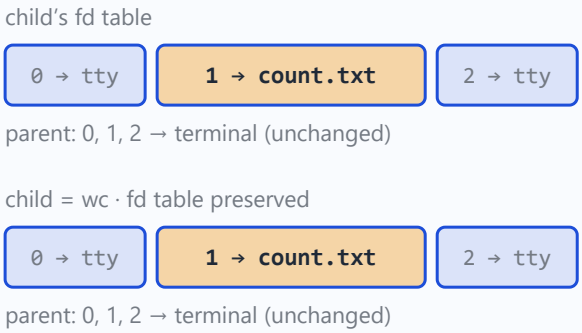


Figure 3.13: The child's file descriptor table. (a) After `fork`: 0, 1, and 2 point to the terminal. (b) `close(1)`: number 1 becomes free. (c) `open` returns the lowest free number, 1; `stdout` is now the file. (d) `exec` replaces the code and preserves the table: `wc` writes to the file without knowing it. The parent's table is unchanged throughout.

Step	Line	Child's fd 1	Event	Screen
1	2	terminal	`fork`: the child copies the parent's fd table	
2	12	terminal	the parent waits with `waitpid`	
3	4	(free)	the child calls `close(1)`	
4	5–6	`count.txt`	`open` returns 1: `fd == 1`	
5	8	`count.txt`	`exec`: the child is now `wc -l`; it writes its result to fd 1	
6	13	—	`wc` finishes; the parent writes to its own fd 1, the terminal	`done`

Table 3.5: Trace of output redirection (parent 1500, child 1501).

Only `done` appears on the screen. The output of `wc` (for example `42 notes.txt`) has gone to `count.txt`; it can be seen with `cat count.txt`. There is not a single line about redirection in the source code of `wc`. The program simply writes to number 1; what that number points to was set by the shell before `exec`.

This program assumes that no other file is opened between `close` and `open`. Real shells do the same job more safely with `dup2(fd, 1)`: they open the file under any number and then redirect number 1 to it. Windows' `CreateProcess` asks for the same settings (standard input and output, working directory, priority) as parameters. That is why the call has ten parameters. On UNIX, each setting is a separate small system call and can be made in any desired order between `fork` and `exec`.

3.3 Process termination

Every process ends one day. The end may come at the process's own wish, or through an error, or through the intervention of another process. However it ends, the kernel reclaims the process's resources. Only one small piece, the exit status, is kept until the parent reads it. This section describes

the kinds of termination, signals, and the zombie and orphan processes that arise from this small piece.

3.3.1 Reasons for termination, exit(), and abort()

A process terminates for one of four reasons. The first two are *voluntary*: the process decides to end itself. The last two are *involuntary*: the process is stopped from outside.

Reason	Type	Example
Normal exit	voluntary	`return 0`, `exit(0)`; the compiler finished its job
Exit on error	voluntary	file not found → `exit(1)`
Fatal error	involuntary	invalid memory access (`SIGSEGV`), division by zero (`SIGFPE`)
Intervention by another process	involuntary	`kill(pid, SIGTERM)`; `TerminateProcess()` on Windows

Table 3.6: Reasons for process termination.

The usual way of a voluntary exit is `exit(status)`. Returning from `main` with `return` has the same effect, because the C runtime calls `exit` with the return value of `main`. `exit` is a C library function and does three things: it flushes the `stdio` buffers, runs the handlers registered with `atexit`, and then enters the kernel with the `_exit` system call. If `_exit` is called directly, the buffers are not flushed. This difference is used after `fork` to keep the child from writing the buffers a second time.

The exit status is an integer, but only its low 8 bits reach the parent. Meaningful values are therefore in the range 0–255. By convention 0 means success and every nonzero value denotes a kind of error. In the shell, the exit status of the last command is shown by `$?`, and operators such as `&&` and `||` work according to this value.

`abort()`, by contrast, is an abnormal exit. The calling process sends itself the `SIGABRT` signal and dies from it. Depending on the system configuration, a **core dump** file may be produced. The `assert` macro calls `abort` when it fails. To terminate another process, the `kill()` system call is used.

However a process ends, the kernel does the same cleanup: it frees the process's memory, closes its open files, and reclaims its other resources. Only the entry in the process table remains: a few pieces of information such as the PID, the exit status, and the CPU time used. This information is kept until the parent calls `wait`.

What happens to the children if the parent exits first varies from system to system. Some systems terminate all the children along with the parent; this is called **cascading termination**. On UNIX, by contrast, the children keep living and are attached to a new parent.

3.3.2 Fatal errors

When a process performs an invalid operation, the processor detects it as an **exception** and hands control to the kernel. Depending on the type of exception, the kernel sends the process a **signal**. A signal is a small asynchronous message that tells a process an event has occurred. If the process has

not installed a handler for that signal, the default action is applied; for fatal errors this action is to terminate the process.

Error	Signal
Invalid or unauthorized memory access	`SIGSEGV`
Integer division by zero	`SIGFPE`
Illegal instruction	`SIGILL`
Alignment or bus error	`SIGBUS`
Writing to a closed pipe	`SIGPIPE`

Table 3.7: Fatal errors and signals.

Figure 3.14 shows the best-known example step by step. The program tries to write `*p = 5` while `p = NULL`. Address 0 is not mapped to any page; the memory management unit (MMU) raises a page fault. The kernel sees that the address is invalid and sends the process `SIGSEGV`. If the process has not installed a handler for this signal, it terminates, and a core dump may be written.

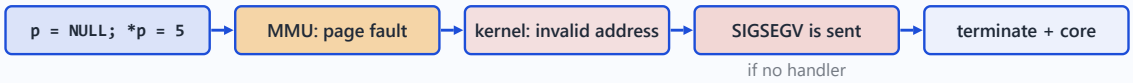


Figure 3.14: The flow of a fatal error: writing through a `NULL` pointer → MMU page fault → the kernel sees that the address is invalid → `SIGSEGV` → if there is no handler, the process terminates and a core dump is taken.

A core dump is a snapshot of the process's memory and registers at the moment it died. When a debugger opens it (`gdb a.out core`; `coredumpctl` on distributions that use systemd), you can see on which line and through which call chain the program died. This kind of examination is called *post-mortem crash analysis*.

Common mistake

Counting a memory leak as a fatal error. Memory that is not `free`d or files that are not closed do not kill the process by themselves; the process keeps running, and its resources are reclaimed anyway when it ends. In a long-lived server, however, a leak gradually exhausts memory. On Linux, if memory runs out, the **OOM killer** (out-of-memory killer) can pick one of the processes holding the most memory and terminate it with `SIGKILL`. Even then the process dies not because of the leak but because of this signal.

3.3.3 Termination by signals

A process can terminate another process with the `kill(pid, signal)` system call. Its name is misleading: `kill` only sends a signal; the result of the signal need not be that the process dies. The command-line commands `kill`, `pkill`, and `killall` are user interfaces to this call.

Signal	No.	Meaning
`SIGTERM`	15	terminate politely; the process can catch it and clean up (the default of `kill`)
`SIGKILL`	9	terminate by force; cannot be caught or ignored, no cleanup possible
`SIGINT`	2	Ctrl+C in the terminal
`SIGTSTP`	20	Ctrl+Z in the terminal (suspend); `fg` or `bg` resumes
`SIGSTOP` / `SIGCONT`	19 / 18	stop (cannot be caught) / resume

Table 3.8: Signals used in process control (numbers are for x86 and ARM Linux).

Signal numbers can vary from architecture to architecture; the values in the table are for x86 and ARM Linux. Portable code therefore uses names rather than numbers: `kill(pid, SIGTERM)` instead of `kill(pid, 15)`. A typical use on the command line looks like this:

```
$ ps -e | grep writer
 1234 ?  00:00:03 writer
$ kill 1234       # SIGTERM
$ kill -9 1234    # SIGKILL
$ pkill writer    # by name
$ killall writer
```

A good habit is to send `SIGTERM` first and to move on to `SIGKILL` if the process does not respond. A program that catches `SIGTERM` deletes its temporary files, finishes interrupted writes, and closes its connections cleanly. `SIGKILL` gives no such chance: the process is stopped where it stands. Killing a database with `kill -9` may require a recovery operation at the next startup.

Not every process can send a signal to every process. The sending process must belong to the same user as the target or be a privileged user (root). Otherwise `kill` returns with an “operation not permitted” error. This rule prevents one user from terminating another user’s processes.

3.3.4 Zombie and orphan processes

The order in which the parent and child end gives rise to two special cases. Both are related to the `wait` call.

Definition 3.7

A **zombie process** is a process that has finished but whose parent has not yet collected its exit status with `wait`. Its memory and resources have been released; only its PID and exit status remain in the process table. An **orphan process** is a process that is still running whose parent terminated before it.

Figure 3.15 puts the two scenarios side by side. In the first scenario the parent (PID 100) and the child (PID 101) are running. The child ends with `exit`, but the parent has not yet called `wait`. The kernel cannot discard the child's exit status, because the parent may ask for it later. The child becomes a zombie: in the `ps` output its state is `Z` and `<defunct>` appears next to its name. When the parent calls `wait`, it receives the status and the kernel deletes the entry. This step is called **reaping**; after it, no zombie remains.

In the second scenario the parent ends before the child. The child (PID 101) is still running and is now an orphan. The kernel does not leave the orphan without an owner: it attaches it to `init` or `systemd` (PID 1). If a **subreaper** process has been defined in the system, the orphan is attached to it; container managers and some session managers use this feature. If the child now calls `getppid()`, it gets 1 (or the PID of the subreaper). When the orphan ends, its new parent reaps it with `wait`. Because `init` does this job regularly, no zombies accumulate from orphans.

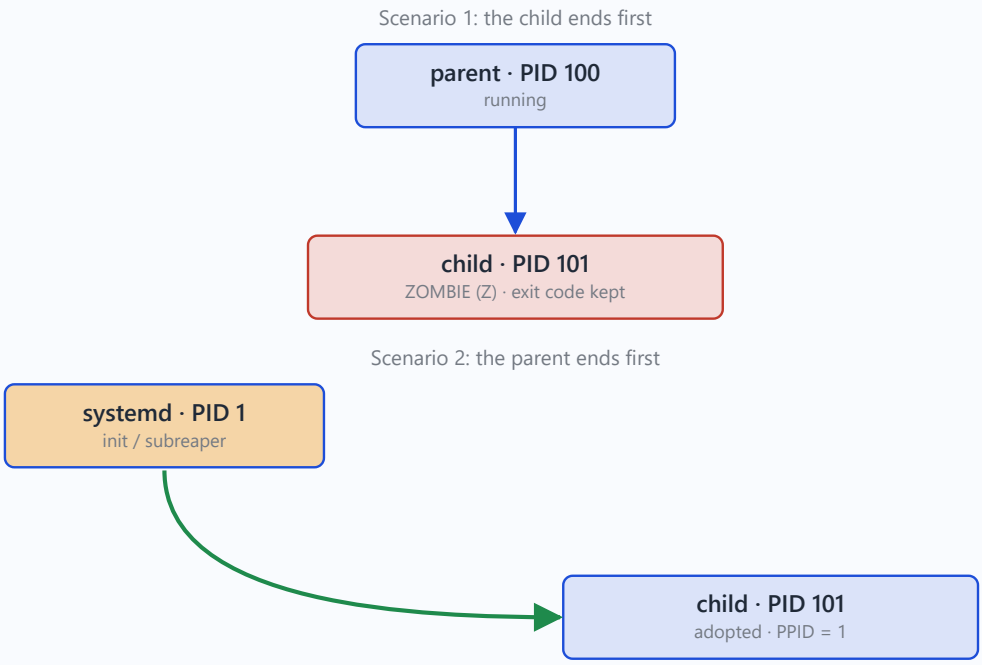


Figure 3.15: Zombie and orphan. (a) Scenario 1: the child (101) has finished but the parent (100) has not yet called `wait`; the child is a zombie and its exit code is kept. (b) Scenario 2: the parent finished first; the orphaned child was attached to `systemd` (PID 1), and its PPID is now 1.

Zombies use no memory or processor time, but they consume a limited resource: the PID. The number of PIDs on Linux is limited (the default upper limit is 32,768 or a larger value, depending on the system). A long-lived server that never calls `wait` and creates a child for every request accumulates thousands of zombies. Eventually no new process can be created and `fork` fails. Killing a zombie with `kill` does not help either, because it is already dead. The solution is either for the parent to call `wait` or for the parent to be terminated: then the zombies pass to `init` and are reaped.

A server that creates children usually catches the `SIGCHLD` signal in order to reap the children as they finish. The following skeleton shows this pattern:

```
#include <signal.h>
#include <sys/wait.h>

void reap(int sig) {
    (void)sig;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        ;                       /* collect all finished children */
}

int main(void) {
    signal(SIGCHLD, reap);      /* run reap() when a child ends */
    /* ... server loop: fork() for each request ... */
    return 0;
}
```

The first argument −1 of `waitpid` means “any child.” With the `WNOHANG` option, `waitpid` returns 0 without waiting if there is no finished child. The loop collects all of several children that finish at the same time. This matters because if several signals of the same kind arrive one after another, they may be delivered as a single signal.

Common mistake

Confusing zombies with orphans, or assuming that the two are equally harmful. A zombie is a process that has *finished* but has not been reaped; an orphan is a process that is *running* but whose parent has died. Orphans are harmless, because `init` adopts them and reaps them when they end. Accumulating zombies, on the other hand, can lead to PID exhaustion; the source of the problem is a parent that does not call `wait`.

3.4 Process states and the PCB

A process does not run continuously during its lifetime. Most of the time it waits for the processor or waits for an event to happen. The kernel must know what state each process is in at the moment and must keep everything about it in one place. This section describes the process states and the data structure in which this information is kept, the process control block.

3.4.1 The five-state model

Definition 3.8

The **process state** describes the current activity of the process. In the five-state model, the states are as follows:

- **New**: the process is being created.
- **Ready**: the process is ready to run and is waiting for the processor.
- **Running**: its instructions are being executed on the processor.
- **Waiting** (blocked): it is waiting for an event (the completion of I/O, a signal, the termination of a child).
- **Terminated**: its execution has finished.

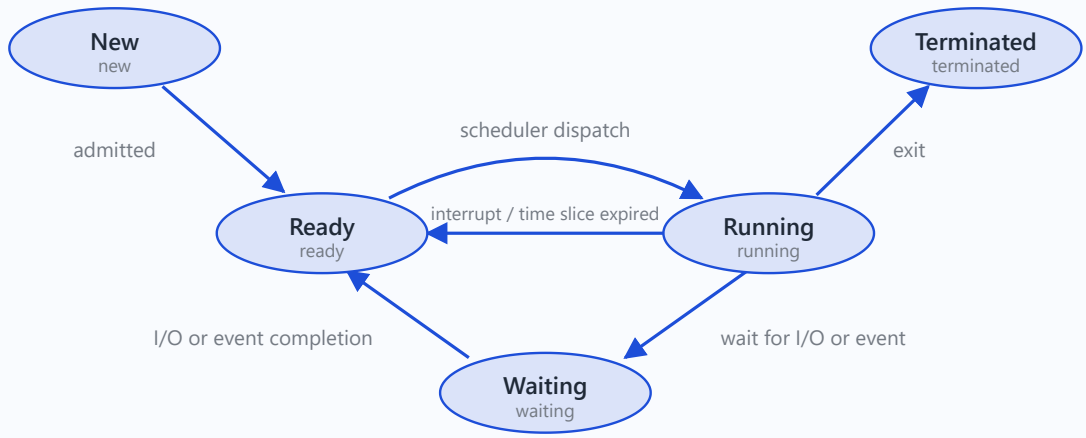


Figure 3.16: The five-state process model and its six transitions: admitted, scheduler dispatch, interrupt or time slice expired, wait for I/O or event, I/O or event completion, exit.

The arrows in Figure 3.16 are the transitions between states. Each transition has a cause:

- **New → Ready (admitted):** The kernel has set up the structures the process needs and has put the process in the ready queue.

- **Ready → Running (scheduler dispatch):** The scheduler chooses this one among the ready processes and loads its context onto the processor. This step is called **dispatch**.
- **Running → Ready (interrupt):** The time slice has expired or a higher-priority process has become ready. The process waits for nothing; its turn is simply over.
- **Running → Waiting (wait for event):** The process has to wait for an event such as reading from disk, input from the keyboard, or the termination of a child. It gives up the processor on its own.
- **Waiting → Ready (event completion):** The awaited event has occurred. The process does not run directly; it returns to the ready queue.
- **Running → Terminated (exit):** The process has called `exit` or has been killed.

On a single core, only one process can be in the *running* state at any instant. On n cores, at most n processes can run. Most of the hundreds of processes in the system are ready or waiting. In reality, the bulk of processes are waiting: an editor waits for a key press, a server waits for a network request.

3.4.2 State transitions step by step

To make the model concrete, let us follow a process's lifetime from start to finish. Let the process read data from disk once and exhaust its time slice once.

Example 3.3

1. **New.** The user started the program. The kernel is creating the PCB and allocating memory.
2. **New → Ready.** The kernel admitted the process and put it in the ready queue.
3. **Ready → Running.** The scheduler chose this process; its context was loaded.
4. **Running → Waiting.** The process requested a `read()` from disk. Until the data arrives it has nothing to do, and it gives up the processor.
5. **Waiting → Ready.** The disk interrupt arrived: the data is ready. The process does not run directly; it returns to the ready queue. At that moment another process may be running on the processor.
6. **Ready → Running.** Chosen again.
7. **Running → Ready.** The timer interrupt arrived: the time slice expired. This is **preemptive** scheduling; because the process is not waiting for anything, it goes to the ready state, not the waiting state.
8. **Ready → Running.** Chosen again.
9. **Running → Terminated.** The process called `exit()`. Until the parent calls `wait`, it stays in the process table as a zombie.

During this lifetime the process entered the ready queue 3 times (in steps 2, 5, and 7), waited once, and was chosen for the processor 3 times.

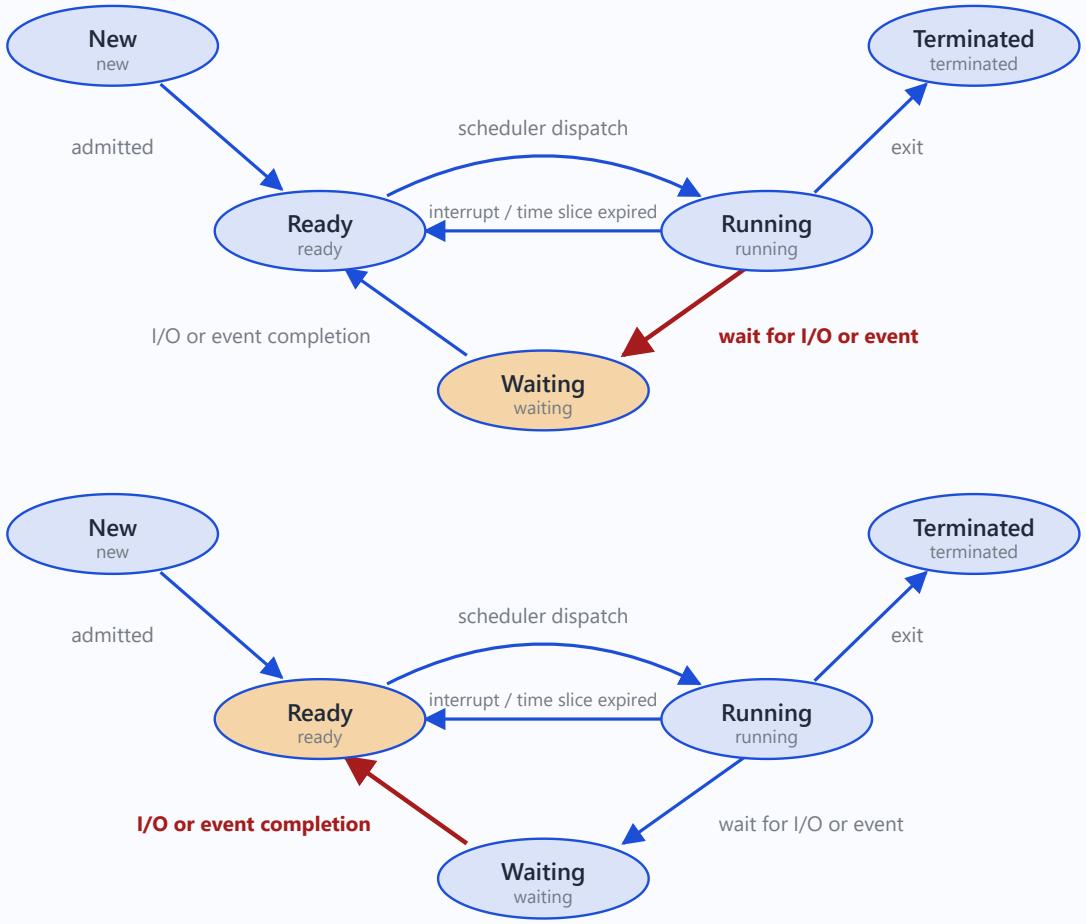


Figure 3.17: Two moments of the trace. (a) Step 4: the process called `read()` and moved to the waiting state. (b) Step 5: the event completed with the disk interrupt; the process returned to the ready state.

Common mistake

Thinking that a process whose awaited event has occurred moves directly to running. The model has no *Waiting* → *Running* arrow. When the event occurs, the processor most likely belongs to another process. The awakened process first enters the ready queue; the scheduler decides whether and when it gets the processor. Another common mistake is to put a process whose time slice has expired in the waiting state. A process whose time slice has expired waits for nothing; its turn is simply over, and it returns to the ready state.

3.4.3 Process states in Linux

The five-state model is conceptual; real kernels merge some states and split others. On Linux the states are shown as a single letter in the `STAT` column of the `ps` command:

STAT	Meaning	Counterpart in the model
R	running or ready to run	Running + Ready
S	interruptible sleep (waiting for an event)	Waiting
D	uninterruptible sleep (usually disk I/O)	Waiting
T	stopped (SIGSTOP , debugger)	suspended
Z	zombie	Terminated (not reaped)
I	idle kernel thread	—

Table 3.9: Linux process states and their counterparts in the five-state model.

```
$ ps -eo pid,stat,comm
  PID STAT COMMAND
    1 Ss   systemd
  890 Ss   sshd
 2105 Ss   bash
 2241 R+   ps
 2307 Z    writer <defunct>
```

In this output `systemd`, `sshd`, and `bash` are in state `S`: all of them are waiting for an event. The `s` at the end indicates that the process is a **session leader**. `ps` itself is `R+`: it is the only process running at that moment, and `+` indicates that it is in the foreground process group. The process named `writer` is in state `Z`; it is an unreaped zombie.

Linux does not distinguish between the ready and running states; both are shown as `R`. The waiting state, however, is split in two. A process in state `S` can be woken by a signal; for example, a program waiting for the keyboard ends at once with Ctrl+C. A process in state `D` does not respond to signals. This state is used for short disk operations whose interruption halfway would cause inconsistency. Processes stuck in `D` for a long time usually point to an unresponsive disk or network file system; they cannot be terminated even with `kill -9`.

3.4.4 The process control block

When a process leaves the processor, the values in its registers must be saved somewhere; otherwise the process cannot continue where it left off. There are other things the kernel needs to know about a process: its state, priority, memory, open files. All of these are gathered in a single record.

Definition 3.9

The **process control block** (PCB) is the data structure that the kernel keeps for each process and that contains everything that needs to be known about that process. The structure formed by all the PCBs is called the **process table**. The part of the PCB that is saved when the process leaves the processor and loaded when it returns (the program counter and registers) is called the **context**.

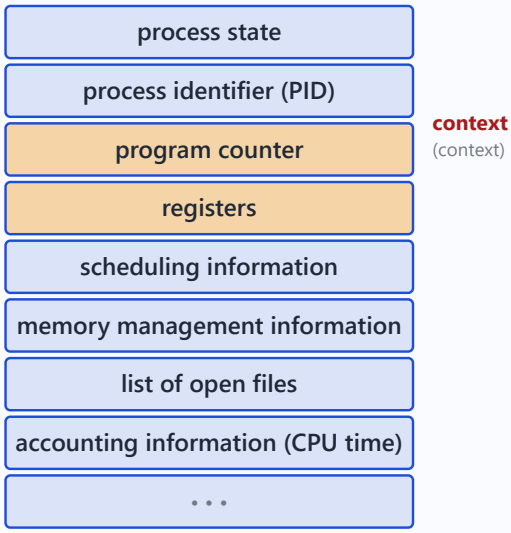


Figure 3.18: The main fields of a PCB: process state, PID, program counter, registers, scheduling information, memory management information, list of open files, and accounting information. The program counter and registers make up the process's context.

PCBs live in kernel memory. User processes cannot access them directly; they can read or change an item in their own PCB only through system calls. `getpid()` reads the PID from the PCB; `nice()` changes the priority; `open()` adds an entry to the list of open files. This restriction is the basis of security: a process cannot raise its own priority at will or reach another process's files.

3.4.5 PCB fields

Tanenbaum divides the fields in a process table entry into three groups according to the management task they serve. Table 3.10 gives this grouping.

Process management	Memory management	File management
registers · program counter · program status word (PSW) · stack pointer · process state · priority · scheduling parameters · PID · parent process · process group · signals · start time · CPU time used · children's CPU time · next alarm time	pointers to the text, data, and stack segments; today, the page table pointer and the list of memory regions	root directory · working directory · file descriptors · user ID (UID) · group ID (GID)

Table 3.10: PCB fields according to Tanenbaum.

Three fields in the table deserve a separate explanation:

- **Program status word** (PSW): the register that holds the processor's status flags (zero, overflow, carry ...), the operating mode (user or kernel), and the interrupt permissions. Because the result of a comparison sits in these flags, the PSW is also part of the context. Even if the process is interrupted between a comparison and a conditional jump, it must still take the correct branch.
- **Process group**: processes that are managed together. In the shell, a pipeline such as `ls | grep x | wc -l` forms a single group; when Ctrl+C is pressed, `SIGINT` goes to the whole group.
- **Next alarm time**: if a process wants to be alerted after some time with `alarm(seconds)`, the time at which the `SIGALRM` signal will be delivered is kept here.

The phrase "pointers to the text, data, and stack segments" in the memory management column reflects the old, segmented memory layout. Today the PCB points to the process's page table and the list of memory regions. Figure 3.19 shows that the process table consists of entries accessed by PID and that each entry carries information from these three groups.

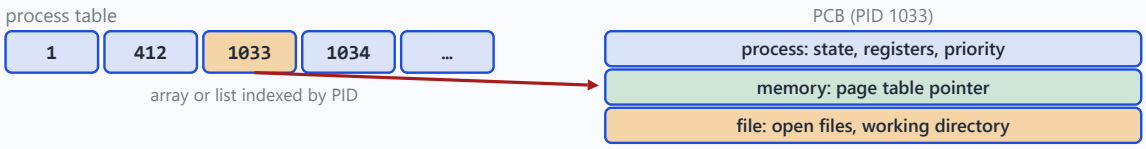


Figure 3.19: The process table: each entry is a PCB. The entry for PID 1033 holds process information (state, registers, priority), memory information (page table pointer), and file information (open files, working directory).

3.4.6 The PCB in Linux: task_struct

In Linux the PCB is called `task_struct` and is defined in the header file `<linux/sched.h>`. Linux represents processes and threads with the same structure; both are a "task" in the kernel's eyes. The structure has several hundred fields; the most important ones are these:

```
struct task_struct {          /* <linux/sched.h> */
    pid_t pid;                    /* process ID */
    unsigned int __state;         /* process state */
    int prio;                     /* priority */
    struct sched_entity se;       /* CFS/EEVDF info */
    struct task_struct *parent;   /* parent process */
    struct list_head children;    /* children */
    struct files_struct *files;   /* open files */
    struct mm_struct *mm;         /* address space */
    /* ... hundreds more fields ... */
};
```

Each of the fields corresponds to a group in Table 3.10. `pid` carries the identifier, `__state` the state, and `prio` the priority. `se` is the information the scheduler keeps per process: how much CPU time the process has used and its place in the scheduler's tree. `parent` and `children` build the process tree. `files` points to the open file table and `mm` to the address space (the page table and memory regions). It matters that `mm` and `files` are pointers: threads in the same process point to the same `mm` and `files` structures, and so share memory and files.

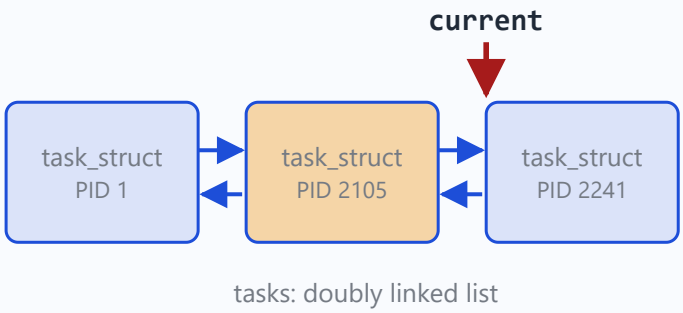


Figure 3.20: All `task_struct`s are on a doubly linked list. `current` points to the task running on that core.

The kernel keeps all tasks on a doubly linked list (Figure 3.20). On each core, a pointer called `current` points to the `task_struct` of the task running at that moment. When a system call arrives, the kernel reaches the caller's information through `current`. For example, `getpid()` roughly returns the value of `current->pid` (in reality it returns the ID of the thread group).

Kernel code changes over time, and the field names in textbooks can become outdated. In older kernels the state field was called `long state` and there was a separate field called `time_slice`. In current kernels the state is named `__state`. The time slice information has also been moved into the `sched_entity` structure.

3.5 Scheduling infrastructure

The goal of multiprogramming is to keep the processor continuously busy. For this, the kernel must know which processes are waiting for the processor and which are waiting for an event, and must be able to hand the processor from one process to another. The *policies* that decide which process is chosen (first come first served, shortest job first, round robin ...) are part of CPU scheduling. This section describes the *mechanism* on which policies are built: the queues, a simple model that measures the benefit of multiprogramming, and the context switch that transfers the processor between processes.

3.5.1 Ready and wait queues

The **scheduler** chooses, among the ready processes, the one that will move to the processor. It has two goals: not to leave the processor idle, and to give the user fast responses. To make the choice, the processes must be kept in an orderly way. The kernel uses queues for this.

Definition 3.10

The **ready queue** is the list of processes that are in memory and waiting for the processor. **Wait queues** hold the processes that are waiting for a particular event (a disk finishing I/O, a child terminating, a packet arriving from the network). There is a separate wait queue for each type of event or each device.

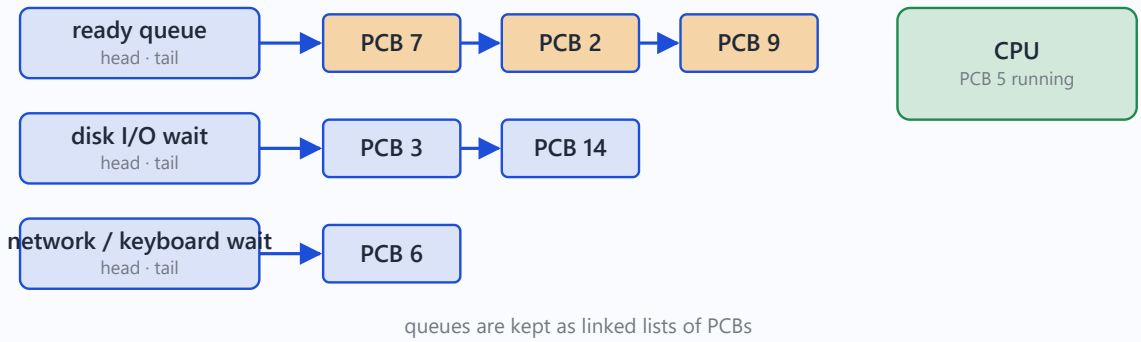


Figure 3.21: Queues are kept as linked lists of PCBs. The ready queue holds PCBs 7, 2, and 9; the disk I/O wait queue holds PCBs 3 and 14; the network/keyboard wait queue holds PCB 6. PCB 5 is running on the processor.

Queues are not a separate data structure but links between PCBs. Each queue has two pointers to its head and tail, and each PCB has a field pointing to the next PCB. A process moving from one queue to another means that its PCB is removed from one list and added to another; the PCB itself is not copied. A process is in only one queue at a time, and the running process is in no queue.

Real systems use smarter structures than a simple FIFO list. Linux assigns a separate **runqueue** to each processor core. This way, cores do not wait for each other when choosing a ready process. The CFS scheduler and its successor EEVDF keep this queue as a red-black tree: finding the next process is fast, and inserting and removing a process takes $O(\log n)$.

3.5.2 The queueing diagram

The lifetime of a process can also be drawn as a journey among queues. In the **queueing diagram** of Figure 3.22, rectangles stand for queues, the circle for the processor, and the arrows for transitions.

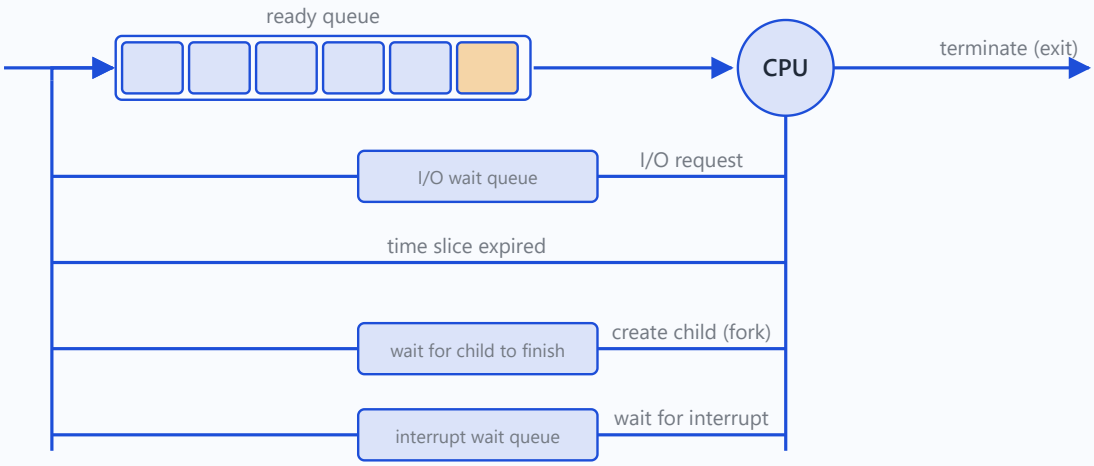


Figure 3.22: The queueing diagram of process scheduling. The process moves from the ready queue to the processor; from there it leaves by one of four routes: an I/O request, expiry of the time slice, creating a child and waiting, or waiting for an interrupt. In the end it leaves the system with `exit`.

The journey proceeds as follows. A new process is added to the end of the ready queue and waits its turn. When the scheduler chooses it, it runs on the processor. A running process can leave the processor by one of four routes:

- It makes an **I/O request** and moves to the relevant I/O wait queue. When the I/O completes, it returns to the ready queue.
- Its **time slice expires**; the timer interrupt takes it off the processor and it returns directly to the ready queue. On this route there is no wait queue.
- It **creates a child** and calls `wait()`; it waits until the child ends.
- It **waits for an interrupt or event**; for example, the expiry of a `sleep()` time or the arrival of a network packet.

The process repeats this cycle until it finishes. In the end it leaves the system with `exit()`; its PCB is deleted when the parent calls `wait`. The diagram is the state model of Figure 3.16 drawn with queues: the ready queue corresponds to the “Ready” state, the wait queues to the “Waiting” state, and the processor to the “Running” state.

3.5.3 CPU-bound and I/O-bound processes

Almost every process alternates between two kinds of activity: it computes on the processor, then waits for I/O, then computes again. Intervals of uninterrupted computation are called **CPU bursts**. Processes fall into two groups according to the length of these bursts (Figure 3.23).

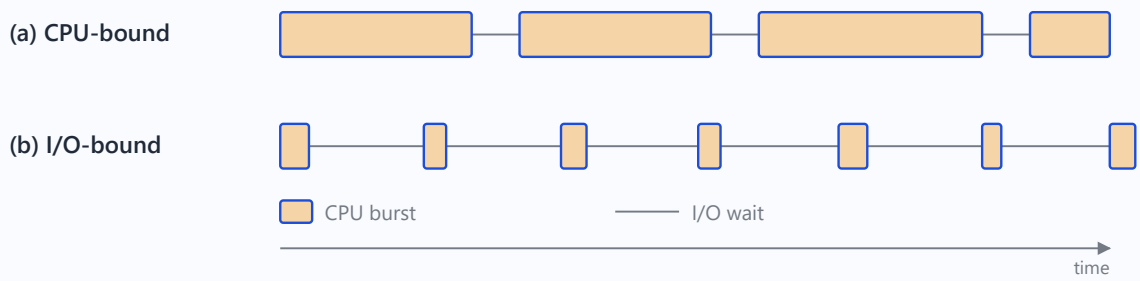


Figure 3.23: (a) CPU-bound process: long CPU bursts, rare I/O. (b) I/O-bound process: short CPU bursts, frequent I/O waits.

Definition 3.11

A **CPU-bound** process spends most of its time computing; it has long CPU bursts and rare I/O requests. Compiling, video encoding, and scientific computation are of this kind. An **I/O-bound** process spends most of its time waiting for I/O; its CPU bursts are short. A text editor, a web server, and a database client are of this kind.

The length of the bursts, not the duration of the I/O, determines which group a process belongs to. An I/O-bound process, after its data arrives, does a little computation and immediately asks for new I/O. Because processors have improved much faster than disks, more processes have become I/O-bound over time. The same computation takes less time each year, but waiting for the disk or the network does not shrink at the same rate.

This distinction matters for the scheduler. When an I/O-bound process becomes ready, it should get the processor at once; it finishes its short job and starts a new I/O, keeping the disk busy. If this process waits in the ready queue behind a long CPU-bound process, the disk sits idle and the user gets a slow response. A good system mixes the two kinds in balance: CPU-bound processes keep the processor busy, and I/O-bound processes keep the devices busy.

3.5.4 Modeling multiprogramming

How many processes should be kept in memory? A simple probabilistic model gives a rough but instructive answer to this question.

Theorem 3.1 (Multiprogramming model)

Let each process spend a fraction p of its time waiting for I/O, and let the processes be independent. If there are n processes in memory, the CPU utilization is

$$\text{CPU utilization} = 1 - p^n$$

Here n is called the **degree of multiprogramming**.

Proof. The processor is idle only when all the processes in memory are waiting for I/O at the same time; if at least one is ready, the processor runs. The probability that a process is waiting at any moment is p. Because the processes are independent, the probability that all n are waiting at once is the product of the probabilities: $p \cdot p \cdot \ldots \cdot p = p^n$. Since the probability that the processor is idle is p^n, the probability that it is busy is $1 - p^n$.

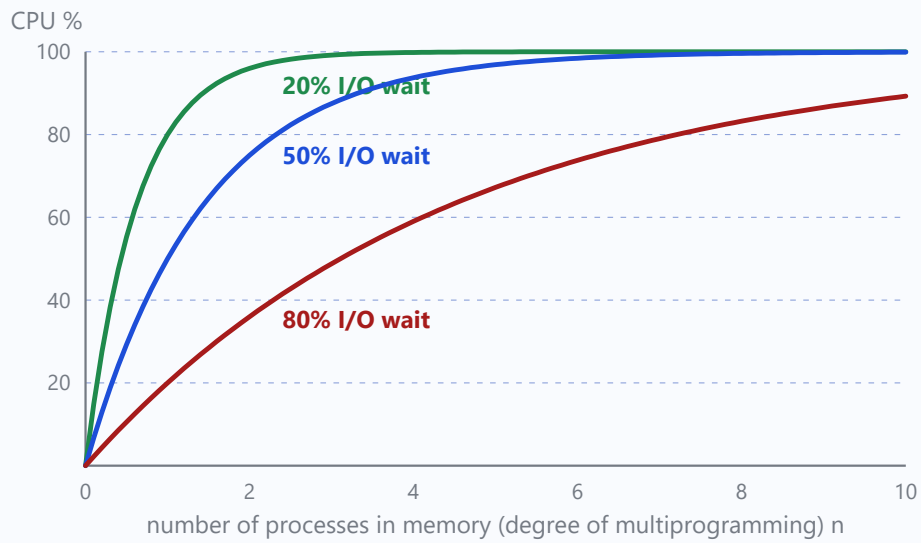


Figure 3.24: CPU utilization as a function of the degree of multiprogramming; three curves for 20%, 50%, and 80% I/O wait. The higher the wait fraction, the more processes are needed to reach the same utilization.

The curves in Figure 3.24 show two lessons of the model. First, for I/O-bound processes (p large), a single-programmed system is wasteful. With $p = 0.8$, utilization with a single process is only 20%; with 4 processes it rises to 59%, and with 10 processes to 89%. Second, the curve gradually flattens. Each new process gains less than the one before, because utilization never reaches 100%.

The model is crude. It assumes that processes are independent; yet processes waiting for a single disk are served in turn, and their waiting is correlated. Still, the model is useful for memory planning: when memory is increased, one can quickly estimate how many more processes will fit and how much this raises utilization. For example, if each process uses 2 GB and $p = 0.8$, adding 8 GB to an 8 GB machine (4 processes) raises utilization from 59% to 83%. The next 8 GB raises it only to 93%.

Example 3.4

Processes spend 60% of their time waiting for I/O. (a) What is the CPU utilization with 3 processes? (b) How many processes must be in memory for utilization to be at least 95%?

1. **(a)** $p = 0.6$ and $n = 3$: $0.6^3 = 0.216$. Utilization is $1 - 0.216 = 0.784$, that is, 78.4%.
2. **(b) Inequality.** We need $1 - 0.6^n \geq 0.95$, which means $0.6^n \leq 0.05$.
3. **Trial.** $0.6^5 = 0.07776$ → utilization 92.2%; not enough. $0.6^6 = 0.046656$ → utilization 95.3%; enough.
4. **Check with logarithms.** $n \geq \ln 0.05 / \ln 0.6 \approx 5.86$. Because n is an integer, it is rounded up: at least 6 processes are needed.

Common mistake

Rounding down the value found with logarithms. Rounding the result ln 0.05 / ln 0.6 ≈ 5.86 down to 5 leaves utilization at 92.2% and the condition is not met. A similar trap is misreading p: p is the I/O wait fraction, not the CPU utilization fraction. If it is said that "processes use the processor 25% of the time," take $p = 0.75$.

3.5.5 Code trace: CPU utilization $1 - p^n$

It is easy to produce the values of the model as a table with a loop. The program in the slides computes the values for $n = 1 \ldots 5$ with $p = 0.8$:

```
double p = 0.8;      /* I/O fraction */
double pn = 1.0;
for (int n = 1; n <= 5; n++) {
  pn = pn * p;       /* all waiting */
  double util = 1 - pn;
  printf("n=%d %.0f%%\n", n, util * 100);
}
```

The trick of the program is that it does not recompute p^n from scratch in every round. The variable `pn` holds the value from the previous round and is multiplied by `p` once per round. Thus in round n, `pn` = p^n. The `%%` in `printf` prints a percent sign; `%.0f` prints the number with no decimals, rounding it.

n	pn	util	Output line	Increase (points)
1	0,8	0,2	`n=1 20%`	—
2	0,64	0,36	`n=2 36%`	16
3	0,512	0,488	`n=3 49%`	12,8

n	pn	util	Output line	Increase (points)
4	0,4096	0,5904	`n=4 59%`	10,24
5	0,32768	0,67232	`n=5 67%`	8,19

Table 3.11: Trace of the loop ($p = 0.8$).

The last column of the table shows the model's "diminishing returns" property with numbers. The second process raises utilization by 16 points; the fifth only by 8.19 points. In fact each increase is exactly 0.8 times the previous one: in going from n to $n + 1$, the increase is $p^n - p^{n+1} = p^n(1 - p)$. That 48.8 is printed as `49%` and 67.232 as `67%` also comes from the rounding of `%.0f`.

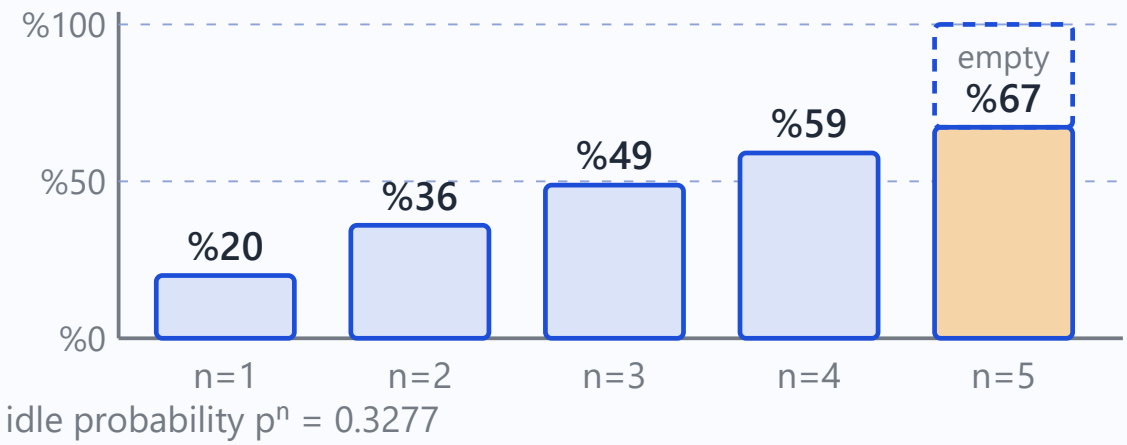


Figure 3.25: CPU utilization for $n = 1…5$ when the loop ends: 20%, 36%, 49%, 59%, 67%. The upper part of each bar is the fraction in which the processor stays idle (p^n).

3.5.6 Context switching

When the scheduler chooses a process, the processor must switch to that process. This switch requires saving the state of the process currently running on the processor somewhere and loading the state of the chosen process into the processor.

Definition 3.12

A **context switch** is the operation of transferring the processor from one process to another. The kernel saves the running process's context (program counter, registers, PSW) into its PCB, loads the chosen process's context from its own PCB, and switches the address space to that of the new process.

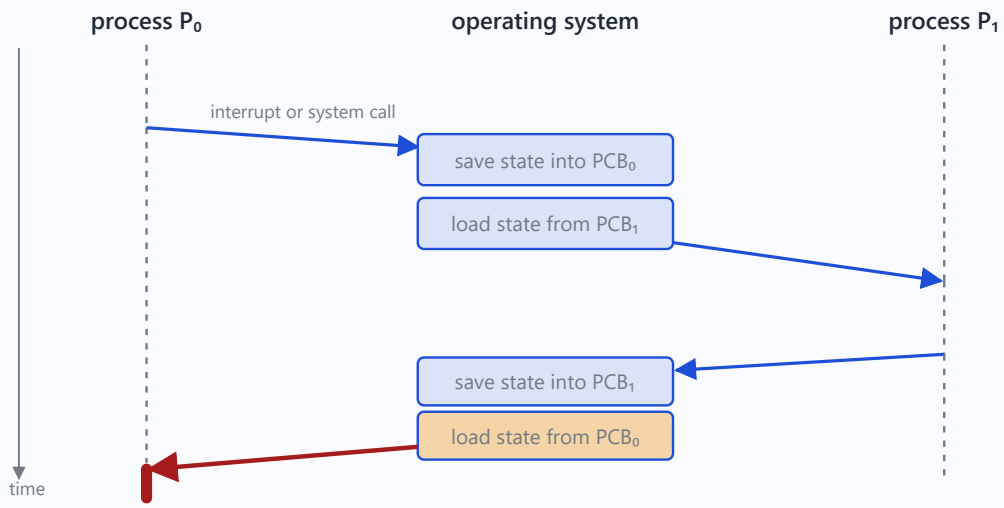


Figure 3.26: Context switch between two processes. While P_0 runs, an interrupt or system call arrives; the kernel saves P_0's state into PCB_0 and loads P_1's from PCB_1. Later the same work is done in the opposite direction. No user work is done during the gray intervals.

Figure 3.26 shows two switches on a time axis. While P_0 runs, an interrupt or system call arrives. The processor switches to kernel mode, and P_0's program counter and registers are saved into PCB_0. The scheduler chooses P_1; P_1's context is loaded from PCB_1 and the address space changes. While P_1 runs, P_0 is idle: it is in the ready or waiting state. At the next interrupt, P_1's state is saved into PCB_1, the contents of PCB_0 are restored, and P_0 resumes where it left off.

The gray intervals in the figure are the cost of the context switch. During that time the processor is working, but no user program makes progress. A context switch is therefore pure **overhead**.

3.5.7 Code trace: context switching

In real kernels the context switch is written in assembly, because the registers must be accessed directly. Its logic, however, can be modeled in a few lines of C. In the program below, the array `P` is the process table; each PCB has three registers (`pc` , `sp` , `r1`) and a state letter. The global variables `pc` , `sp` , and `r1` stand for the processor's registers.

```
struct pcb { int pc, sp, r1; char state; };
struct pcb P[2];   /* process table */
int pc, sp, r1;    /* CPU registers */
void context_switch(int from, int to) {
  P[from].pc = pc;  P[from].sp = sp;  /* save */
  P[from].r1 = r1;  P[from].state = 'R';
  pc = P[to].pc;  sp = P[to].sp;  /* load */
  r1 = P[to].r1;  P[to].state = 'X';
}
/* interrupt */  context_switch(0, 1);
/* ... P1 runs ... */
/* interrupt */  context_switch(1, 0);
```

`context_switch(from, to)` consists of two halves. The first half saves the registers of the old process (`from`) into its PCB and sets its state to `'R'` (ready). The second half loads the registers of the new process (`to`) into the processor and sets its state to `'X'` (running). Initially P_0 is running: `pc = 104`, `sp = 896`, `r1 = 8`. The last state of P_1 is stored in PCB_1: `pc = 340`, `sp = 500`, `r1 = 2`.

Step	Line	CPU	PCB_0	PCB_1	Explanation
1	10	104/896/8 (P_0)	—/—/— X	340/500/2 R	timer interrupt; the kernel decides to switch to P_1
2	5	104/896/8	104/896/— X	340/500/2 R	P_0's `pc` and `sp` have been saved
3	6	104/896/8	104/896/8 R	340/500/2 R	`r1` has been saved too; P_0 is ready
4	7	340/500/8	104/896/8 R	340/500/2 R	P_1's `pc` and `sp` have been loaded
5	8	340/500/2 (P_1)	104/896/8 R	340/500/2 X	`r1` has been loaded; P_1 is running
6	11	352/492/5	104/896/8 R	340/500/2 X	P_1 has run a few instructions; PCB_1 is no longer current
7	12	352/492/5	104/896/8 R	340/500/2 X	new interrupt; the switch back to P_0 is about to happen
8	5	352/492/5	104/896/8 R	352/492/2 X	P_1's `pc` and `sp` have been saved
9	6	352/492/5	104/896/8 R	352/492/5 R	`r1` has been saved; P_1 is ready
10	7	104/896/5	104/896/8 R	352/492/5 R	`pc` and `sp` have been loaded from PCB_0
11	8	104/896/8 (P_0)	104/896/8 X	352/492/5 R	`r1 = 8`; P_0 continues where it was interrupted

Table 3.12: Trace of the context switch. Values are written in the order `pc` / `sp` / `r1`; "—" marks a field that has not yet been saved.

Three intermediate states in the trace deserve attention. In step 4 the processor's registers are mixed: `pc` and `sp` belong to P_1, while `r1` still belongs to P_0. This half-done state shows that a context switch must be carried out without interruption. Real kernels disable interrupts during this time. In step 6, the values in PCB_1 (340/500/2) are out of date; P_1's real state is only in the processor. The PCB is updated not while the process runs but when it leaves the processor. In step 11, P_0 continues from exactly the state in which it was interrupted, from `pc = 104`, and does not notice that it was interrupted.

The model simplifies reality. A real processor has dozens of general-purpose registers, floating-point and vector registers, and the PSW. The address space also changes: the processor's page table register is loaded so that it points to the new process's table.

3.5.8 Limited direct execution

The context switch leaves one question unanswered: while the processor is running a user program, how does the kernel get control back? The kernel is also a program, and the processor runs only one thing at a time. If the user program enters an infinite loop, the kernel's code may never run.

One solution could be to run the user program step by step in an interpreter, but that is far too slow. The route operating systems take is **limited direct execution**. The program runs directly on the processor at full speed. But it has two limits: it cannot do privileged work itself and must ask the kernel; and it cannot hold the processor as long as it likes, because the kernel takes it back by force. Figure 3.27 shows this arrangement in five steps.

1. **At boot (in kernel mode).** The operating system tells the hardware which code to run on which event: it sets up the **trap** and interrupt table. It then starts the **timer**, which will raise an interrupt every few milliseconds. Both operations use privileged instructions; the user program cannot change them.
2. **A is running.** Process A runs directly on the processor in user mode; there is no interpreter in between. For a privileged job such as opening a file, it must trap into the kernel with a system call.
3. **Timer interrupt.** Even if A is in an infinite loop, the interrupt arrives. The hardware saves A's registers on the kernel stack, switches to kernel mode, and jumps to the operating system's interrupt handler. Control is now with the operating system.
4. **The scheduler chooses B.** Because A's time slice has expired, the operating system decides to switch to B. The `switch()` routine writes A's context into PCB(A), loads B's, and moves to B's kernel stack.
5. **Return from trap.** The `return-from-trap` instruction loads B's registers as if B were returning from where it was interrupted, and switches to user mode. Now B runs.

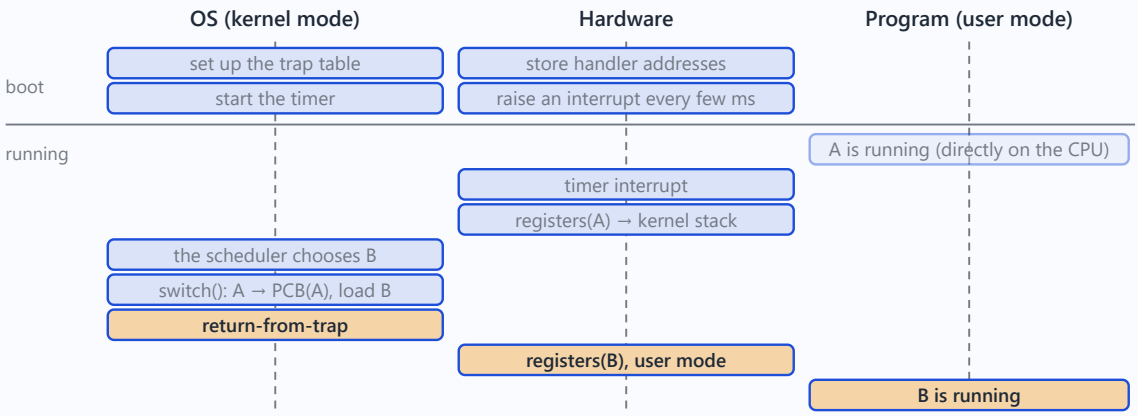


Figure 3.27: Limited direct execution. At boot the operating system sets up the trap table and the timer. While A runs directly on the processor, the timer interrupt gives control to the operating system; `switch()` moves from A to B, and with the return from trap B starts to run.

The timer interrupt is the backbone of this arrangement. Without the interrupt, the operating system could take control back only when the process made a system call of its own accord. This approach is called **cooperative multitasking**, and Windows 3.x and old versions of Mac OS worked this way. On these systems, a single program that entered an infinite loop would lock up the entire machine; the remedy was to restart it.

3.5.9 The cost of a context switch

A context switch does not come free. While moving to another process, the processor must save the old process's state and load the new one's, and no useful work is done during this time. The larger the PCB and the more complex the operating system, the longer the switch takes. The cost is of two kinds:

- **Direct cost:** saving and loading the registers and running the scheduler code. On current Linux it is typically a few microseconds.
- **Indirect cost:** the "cooling" of the cache and the TLB (translation lookaside buffer, a small cache that holds address translations). In its first moments the new process cannot find its data in the cache and suffers many cache misses. This indirect cost is often larger than the direct cost.

Hardware helps to reduce this cost. Some processors have several register sets; a switch amounts to changing a pointer that selects the set. An address-space tag added to TLB entries (ASID on ARM, PCID on x86) removes the need to flush the TLB at every switch: the entries of different processes are told apart by their tags.

The cost of a context switch plays a direct role in choosing the time slice. If the slice is chosen too short, most of the processor's time goes to switches; if too long, processes wait a long time for their turn.

Example 3.5

A context switch takes s = 0.05 ms. There are 3 CPU-bound processes running in turn in the ready queue, and each uses its full slice of q = 4 ms. (a) What percentage of processor time goes to switches? (b) At most how long does a process wait between two of its slices?

1. **Identify one cycle.** Each slice is followed by a switch. So in every interval of $q + s$ = 4.05 ms, 0.05 ms is overhead.
2. **(a)** The overhead fraction is $s / (q + s) = 0.05 / 4.05 \approx 0.0123$, that is, about 1.23%.
3. **(b)** When process A finishes its slice, a switch happens and B runs; one more switch and C runs; another switch and A starts again. A's wait is 3 switches and 2 slices: $3 \cdot 0.05 + 2 \cdot 4 = 8.15$ ms.
4. **Generalization.** For k processes the wait is $k \cdot s + (k - 1) \cdot q$. As the slice gets smaller, the wait gets shorter but the overhead fraction grows.

3.5.10 Interrupt handling

Most context switches begin with an interrupt. Tanenbaum summarizes in eight steps what happens at the lowest layer of the operating system when an I/O interrupt arrives:

1. **The hardware saves the program counter, the PSW, and a few registers on the stack.** The interrupt request is accepted not in the middle of an instruction but at an instruction boundary. The processor switches to kernel mode.
2. **The hardware loads the new program counter from the interrupt vector.** The **interrupt vector** is the table that holds the handler's address for each type of interrupt. The operating system fills in this table at boot.
3. **An assembly routine saves the registers.** The registers that the hardware did not save are written to the current process's PCB or the kernel stack.
4. **An assembly routine sets up a new stack.** The interrupt handler runs not on the user's stack but on the kernel stack.
5. **The interrupt service routine written in C runs.** It typically reads the data coming from the device and places it in a buffer. If a process is waiting for this data, it moves that process to the ready state.
6. **The scheduler chooses the next process.** If the awakened process has a higher priority than the interrupted one, it may be chosen. In that case the interrupt turns into a context switch.
7. **The C routine returns to the assembly code.**
8. **An assembly routine starts the chosen process.** The process's registers and memory mapping are loaded and the processor returns to user mode.

The division of labor among the steps is remarkable. The hardware does the first two steps, because at that moment there is no safe environment in which any software could run. Steps 3, 4, and 8, which touch the registers directly, are written in assembly. The real work is done in portable C code. This path, which starts with an interrupt, lets the interrupted process continue without noticing it was interrupted; or, if the scheduler so decides, hands the processor to another process.

3.6 Interprocess communication

Separate address spaces prevent one process's error from spreading to another. But the same separation makes it hard for processes to work together: one process cannot read another's variable. Cooperating processes use the facilities that the operating system provides to exchange data. This section describes the two basic models of these facilities, shared memory and message passing, and then UNIX pipes and sockets.

3.6.1 Independent and cooperating processes

A process is **independent** if it does not affect other processes and is not affected by them. Processes that share data or affect one another's work are **cooperating**. There are four reasons for processes to cooperate:

- **Information sharing:** Several applications may want to access the same data, for example text copied to a clipboard.
- **Speedup:** A large job is divided into pieces, and the pieces are processed in parallel on different cores.
- **Modularity:** The system is built from small processes that each do a single job.
- **Security and isolation:** Risky work is done in separate processes with restricted privileges; Chrome's renderer processes are an example.

Definition 3.13

Interprocess communication (IPC) is the general name for the mechanisms that let cooperating processes exchange data and coordinate their actions. It has two basic models: **shared memory** and **message passing**.

IPC must answer three questions. The first is obvious: how does one process pass information to another? The other two are subtler. The second is that processes must not get in each other's way. If two reservation processes both see the last free seat at the same time and both sell it, two tickets are issued for the same seat. The third is ensuring the correct order among dependencies. One must not fire before aiming; data must not be consumed before it is produced.

The last two questions require **synchronization**. Uncontrolled access to shared data makes the result depend on the order in which the processes run. This situation is called a **race condition**. The same questions apply to threads that share the same address space.

3.6.2 Shared memory and message passing

The two models differ in where the data passes between processes (Figure 3.28).

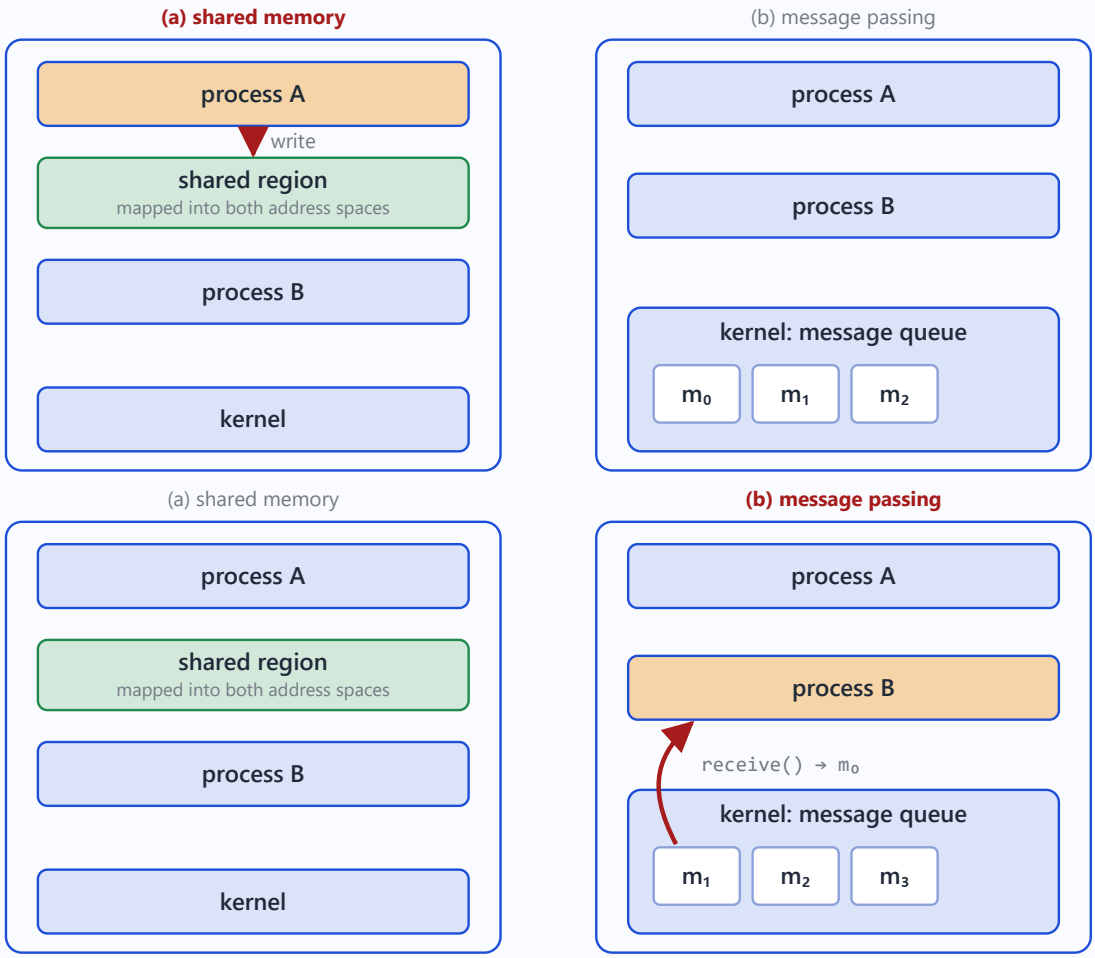


Figure 3.28: Two IPC models. (a) Shared memory: the kernel maps a region into the address spaces of two processes, and A writes to the region directly. (b) Message passing: the kernel keeps the messages in its own queue; B takes the next message (m_0) with `receive()`.

In **shared memory**, the kernel creates a memory region and maps it into the address spaces of two processes. A system call is needed only for the setup. After that, A writes to the region with an ordinary memory write, and B reads with an ordinary memory read. The kernel does not come into play in this exchange at all, which is why shared memory is very fast. The price is this: the kernel does not control who writes when. The processes must provide the coordination themselves, with tools such as locks or semaphores.

In **message passing**, A sends a message with a `send(m)` call; the kernel copies the message into its own queue. B takes the next message with a `receive()` call; this time the kernel copies the message into B's memory. Every message involves at least one system call and two copies; message passing is therefore slower than shared memory. Its gain is that the coordination lives in the kernel: a process that wants to read from an empty queue is made to wait, and a message is never read half-way. The same interface can also work over a network, whereas shared memory is possible only for processes on the same machine.

Criterion	Shared memory	Message passing
Role of the kernel	only at setup	on every message
Copying	none	the message is copied to the kernel and to the receiver
Coordination	the processes' responsibility	provided by the kernel
Suitable when	large data on the same machine	small messages; distributed systems

Table 3.13: Comparison of shared memory and message passing.

3.6.3 POSIX shared memory

POSIX sets up shared memory with an object that is named like a file. The code below contains the parts of two separate processes: the upper one is the producer, the lower one the consumer.

```
/* producer */
int fd = shm_open("/os03", O_CREAT|O_RDWR, 0600);
ftruncate(fd, 4096);
char *p = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd, 0);
strcpy(p, "hello");     /* direct write */

/* consumer (another process) */
int fd = shm_open("/os03", O_RDONLY, 0);
char *p = mmap(NULL, 4096, PROT_READ,
               MAP_SHARED, fd, 0);
printf("%s\n", p);
shm_unlink("/os03");    /* remove */
```

The setup consists of three steps. `shm_open` creates a shared memory object named `/os03` or opens the existing one, and returns a file descriptor. On Linux this object appears as a file under `/dev/shm`. `ftruncate` sets the size of the object to 4096 bytes. `mmap` maps the object into the process's address space and returns the starting address of the mapping. The `MAP_SHARED` flag is important: writes to this region are visible in every process that maps the same object.

After setup, shared memory is ordinary memory. The producer writes `"hello"` directly with `strcpy`; no system call is involved. The consumer opens the same object read-only, maps it, and reads it with `printf`. When finished, `shm_unlink` deletes the object's name; the object disappears when the last process that has mapped it lets go.

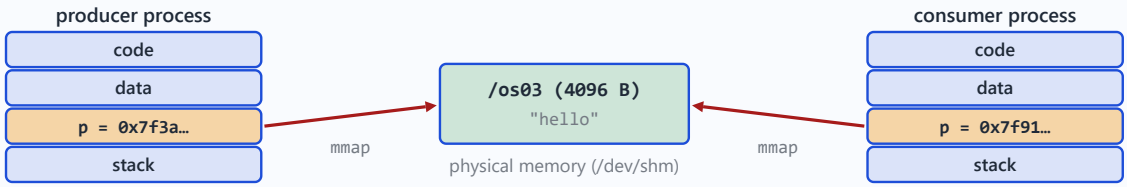


Figure 3.29: Two processes map the same physical region (`/os03` , 4096 bytes) to different virtual addresses in their own address spaces: `p = 0x7f3a…` in the producer and `p = 0x7f91…` in the consumer. Both see the same `"hello"` data.

Figure 3.29 also shows a subtle point: the values of `p` in the two processes are different. The region can be mapped to a different virtual address in each process's address space. Storing a pointer in a shared region is therefore dangerous; an address that is valid in one process may be meaningless in the other. Shared data structures use an offset from the start of the region instead of a pointer.

Common mistake

Assuming that shared memory provides coordination by itself. In the example code, if the consumer runs before the producer writes, it sees the region empty and prints an empty string. Worse, if the consumer reads while the producer is writing, it may see half-written data. In real code, access to the region must be regulated with a semaphore or a lock. There is also an older interface for this: System V's `shmget` and `shmat` calls. The logic is the same; only the objects are named with a numeric key instead of a file name.

3.6.4 Message-passing design

Message passing has two basic operations: `send(message)` and `receive(message)` . Before two processes can communicate, a **communication link** must be established between them. Four questions stand out in the design of this link.

Dimension	Options
Physical link	shared memory, hardware bus, network
Naming	direct (the process's name) · indirect (mailbox or port)
Synchronous / asynchronous	blocking (synchronous): the sender waits until the message is received, the receiver waits until a message arrives · nonblocking (asynchronous): the call returns immediately
Buffering	zero capacity (the sender waits: rendezvous) · bounded capacity (the sender waits if the queue is full) · unbounded capacity (the sender never waits)

Table 3.14: Dimensions of message-passing design.

Buffering is how many messages the link can hold. A **zero-capacity** link has no place to store a message. The sender waits until the receiver calls `receive` , and the message is transferred directly at the moment the two processes meet. In a **bounded-capacity** link, up to *N* messages can wait in the

queue; if the queue is full, the sender waits. **Unbounded capacity** is conceptual; in reality memory is always limited.

Definition 3.14

Rendezvous is message passing in which both `send` and `receive` are blocking. The sender waits until the receiver takes the message; the receiver waits until a message arrives. The transfer happens when both processes are ready at the same time.

Rendezvous solves the producer–consumer problem by itself. The producer cannot send a new message until the consumer has taken one; the consumer cannot take a message that has not been produced. The programmer need not keep a lock or a counter separately. Examples in real systems include POSIX message queues (`mq_send`, `mq_receive`), the ports of the Mach kernel underlying macOS, and Windows' ALPC (Advanced Local Procedure Call) mechanism.

Example 3.6

A message queue has a capacity of 2; `send` and `receive` are blocking. The consumer has not yet called `receive`, and the producer wants to send messages m_1, m_2, and m_3 in a row. What happens?

1. `send(`m_1`)`: The queue is empty; m_1 is copied into the queue and the call returns immediately. There is 1 message in the queue.
2. `send(`m_2`)`: There is room; m_2 is added. The queue is now full (2/2).
3. `send(`m_3`)`: There is no room in the queue; the producer moves to the waiting state.
4. **The consumer calls** `receive()`: It takes m_1 (the message queue is FIFO). One slot becomes free; the kernel wakes the waiting producer and m_3 is added to the queue. The queue holds m_2 and m_3.

If the capacity were zero, the producer would wait at the very first `send`, and when the consumer called `receive`, m_1 would be transferred directly.

3.6.5 Direct and indirect communication

The naming question is how the sender says whom it is sending the message to (Figure 3.30).

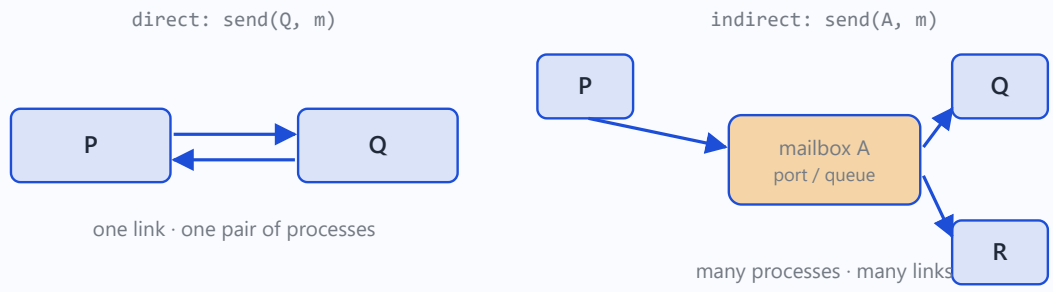


Figure 3.30: On the left, direct communication: P sends to Q by name; the link belongs to a single pair of processes. On the right, indirect communication: P, Q, and R communicate through mailbox A; one link can belong to many processes, and one pair can have many links.

In **direct communication**, each process names the other party: `send(P, m)` sends to P, and `receive(Q, m)` receives from Q. The link is established automatically and belongs to exactly one pair of processes. There is a single link between each pair, and the link is usually bidirectional. The weakness of this approach is that process names are embedded in the code: if a process restarts and gets a new identity, all the code that names it must change.

In **indirect communication**, messages are sent to and received from a **mailbox** or **port**: `send(A, m)`, `receive(A, m)`. Each mailbox has a unique identity; processes that share the same mailbox can communicate. A link can belong to more than two processes, and a pair of processes can communicate through several mailboxes. A link can be unidirectional or bidirectional. This flexibility raises a question too: when P puts a message in a mailbox and Q and R call `receive` at the same time, who gets the message? Systems solve this either by allowing only one receiver at a time or by choosing the receiver themselves.

3.6.6 Pipes

A **pipe** is UNIX's oldest and most widely used IPC tool. A pipe is a FIFO buffer kept in the kernel with two ends. The bytes written at one end are read from the other end in the same order (Figure 3.31).

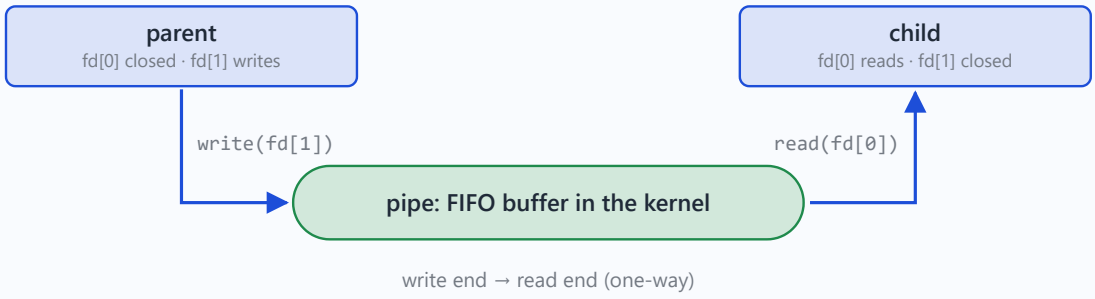


Figure 3.31: An ordinary pipe: the parent writes to the write end `fd[1]`, and the child reads from the read end `fd[0]`. The data passes through the FIFO buffer in the kernel and flows in one direction. Each process closes the end it does not use.

Four questions are asked when designing a pipe. Is the flow unidirectional (simplex) or bidirectional? If bidirectional, can both directions be used at the same time (full duplex) or only in turn (half duplex)? Is a parent–child relationship required between the communicating processes? Can the pipe be used over a network? The two kinds of pipe answer these questions differently.

	Ordinary (unnamed)	Named (FIFO)
Access	only the creating process and its children	any process that knows the file name
Direction	one-way	UNIX: half duplex; Windows: full duplex
Lifetime	ends when the processes close it	created with `mkfifo`; stays in the file system
Network	no	UNIX: no; Windows: yes

Table 3.15: Ordinary and named pipes.

An ordinary pipe has no name; it is accessed only through file descriptors. These descriptors can pass to other processes only by inheritance through `fork`. An ordinary pipe is therefore used between related processes. A named pipe, on the other hand, carries a name in the file system (`mkfifo /tmp/channel`). Two processes that do not know each other can communicate by opening this name. Although FIFOs on UNIX can be used in both directions, in practice a separate FIFO is opened for each direction, because if two processes write to the same FIFO the data gets mixed up.

3.6.7 Code trace: pipe() and fork()

The following program shows the parent sending `"welcome"` to the child over a pipe:

```
int main(void) {
  int fd[2];
  char buf[32];
  pipe(fd);
  pid_t pid = fork();
  if (pid == 0) {   /* child: reads */
    close(fd[1]);
    ssize_t n = read(fd[0], buf, sizeof buf - 1);
    buf[n] = '\0';
    printf("child got: %s\n", buf);
    close(fd[0]);
    return 0;
  }
  close(fd[0]);     /* parent: writes */
  write(fd[1], "welcome", 7);
  close(fd[1]);
  wait(NULL);
  return 0;
}
```

The call `pipe(fd)` creates a pipe and returns two descriptors: `fd[0]` is the read end and `fd[1]` is the write end. Because numbers 0, 1, and 2 belong to the standard input, output, and error streams, the lowest free numbers are 3 and 4. Then comes `fork`, and the child inherits both ends. Thus both ends of the pipe are open in both processes. Each process closes the end it will not use: the child the write end, the parent the read end. Let the parent's PID be 1234 and the child's 1235.

Step	Line	Event	Open ends	Buffer	Output
1	4	`pipe(fd)`: `fd[0] = 3`, `fd[1] = 4`	P: 3, 4	empty	
2	5	`fork`: the child inherits the ends	P: 3, 4 · C: 3, 4	empty	
3	7	the child closes the write end	P: 3, 4 · C: 3	empty	
4	8	the child calls `read`: the pipe is empty → waiting	P: 3, 4 · C: 3	empty	
5	14	the parent closes the read end	P: 4 · C: 3	empty	
6	15	the parent writes 7 bytes; the child wakes up	P: 4 · C: 3	`welcome`	
7	16–17	the parent closes the write end and waits with `wait`	C: 3	`welcome`	
8	8	`read` returns 7; `n = 7`	C: 3	empty	
9	9–10	`buf[7] = '\0'`; the child prints	C: 3	empty	`child got: welcome`
10	11–12, 17–18	the child closes the read end and exits; the parent returns from `wait` and ends	—	—	

Table 3.16: Trace of `pipe()` + `fork()`. In the "Open ends" column, P stands for the parent and C for the child.

The order of the steps in the trace may vary with scheduling. For example, the parent may write before the child calls `read`; then the child gets the data without waiting at all. The output is the same in every order, because only the child prints and the data flows along a single path.

Three details of the trace show the rules of pipes. First, a process that reads from an empty pipe is made to wait (step 4). Second, writes of up to `PIPE_BUF` bytes (4096 bytes on Linux) are **not split**: they are written either completely or not at all. This way the small messages of two processes writing to the same pipe do not get mixed together. Third, `read` returns at most as many bytes as requested but may return fewer. The program asked for 31 bytes (`sizeof buf - 1`), there are 7 bytes in the pipe, and `read` returns 7. The returned value is used to put `'\0'` at the end of the string.

Closing the write end matters too. `read` returns 0 when the pipe is empty and *no* process has a write end open; this means end of file (EOF). As long as a write end stays open somewhere, the reader waits, thinking "maybe more data will come."

Common mistake

Not closing unused pipe ends. Suppose the child uses a loop that reads until EOF and the parent does not call `close(fd[1])`. Then even when the pipe empties, a write end remains open and `read` never returns 0. The child waits forever, and the parent waits for the child in `wait`: the program deadlocks. The child not closing its own write end has the same result, because that end also counts as "an open writer." There is a rule in the opposite direction too: a process that writes to a pipe whose read end is completely closed receives a `SIGPIPE` signal and by default terminates.

3.6.8 Pipelines in the shell

When `ls | wc -l` is typed in the shell, the output of `ls` becomes the input of `wc`, and the number of files in the directory appears on the screen. The shell does this by combining the pipe with redirection:

```
int fd[2]; pipe(fd);
if (fork() == 0) { dup2(fd[1], 1); close(fd[0]); close(fd[1]);
                   execlp("ls", "ls", NULL); }
if (fork() == 0) { dup2(fd[0], 0); close(fd[0]); close(fd[1]);
                   execlp("wc", "wc", "-l", NULL); }
close(fd[0]); close(fd[1]); wait(NULL); wait(NULL);
```

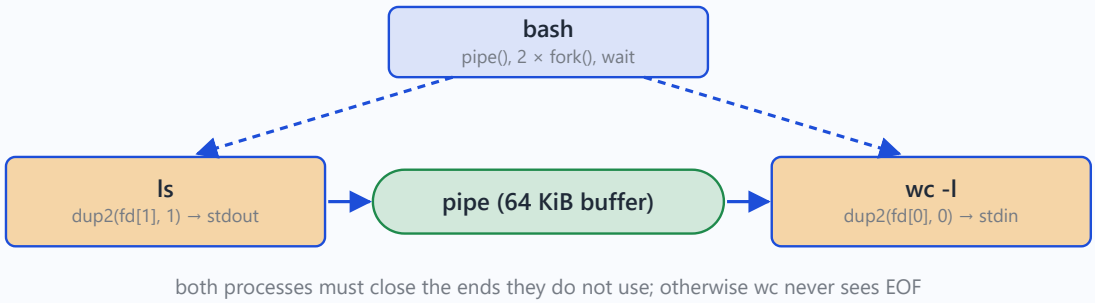


Figure 3.32: The `ls | wc -l` pipeline. The shell creates one pipe and two children; the standard output of `ls` is redirected to the write end of the pipe, and the standard input of `wc` to the read end. On Linux the pipe buffer is 64 KiB by default.

The shell first creates a pipe and then calls `fork` twice. The first child connects its standard output to the write end of the pipe with `dup2(fd[1], 1)`. `dup2(old, new)` makes the descriptor numbered `new` a copy of `old`; if `new` is open, it closes it first. The child then closes both pipe descriptors, because from now on it will do its job through number 1. Then it turns into `ls`. The second child does the same for the read end and standard input (0) and turns into `wc -l`. The shell closes its own ends and waits for the two children.

Here again the closing rule is decisive. If the shell or the second child kept the write end open, then even when `ls` finished and closed its own end, an open writer would remain on the pipe. `wc` would never see EOF and could never finish counting lines. Error checks have been left out of the code in the slides for brevity; a real shell checks the return value of every call.

The strength of this design is that programs do not notice the pipe. `ls` writes to standard output and `wc` reads from standard input; for them there is no difference between a terminal, a file, and a pipe. Thus small programs that know nothing of each other can be chained with `|` in the shell to do big jobs. The UNIX philosophy of "each program should do one thing well" rests on this infrastructure.

3.6.9 Sockets

Pipes connect processes on the same machine. Sockets are used for processes on different machines to communicate.

Definition 3.15

A **socket** is an endpoint of communication. A network socket is identified by the combination of an IP address and a port number; for example `161.25.19.8:80`. Communication is always between a pair of sockets.

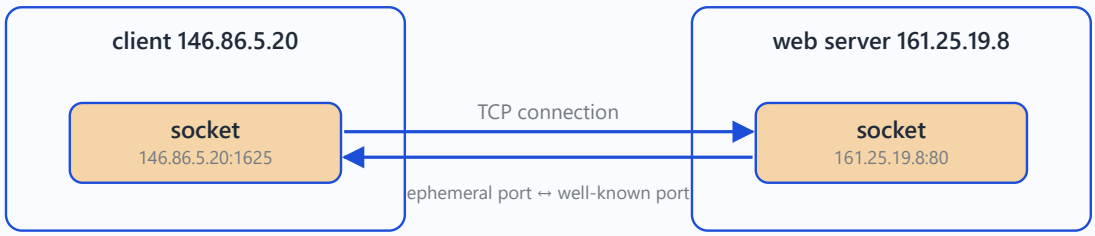


Figure 3.33: Communication with sockets. The client (146.86.5.20) establishes a TCP connection from an ephemeral port (1625) to the well-known port 80 of the web server (161.25.19.8).

The IP address distinguishes the machine, and the port distinguishes the service on the machine. A web server listens on port 80; when the client connects, the operating system assigns the client an ephemeral port (1625 in the figure). The connection is defined by the pair of sockets `146.86.5.20:1625` and `161.25.19.8:80`. A second connection opened from the same machine to the same server gets another ephemeral port, and the two connections do not get mixed up.

Ports below 1024 are reserved for **well-known** services: SSH 22, HTTP 80, HTTPS 443. Starting to listen (`bind`) on these ports as a server requires privilege; this prevents an ordinary user from setting up a fake SSH server. The address `127.0.0.1` (`::1` in IPv6) is the **loopback** address and points to the same machine; processes on the same machine can also communicate over network sockets.

Sockets offer two basic kinds of communication. **TCP** is connection-oriented: data is delivered as an ordered, reliable byte stream; lost packets are retransmitted. **UDP** is connectionless: each datagram is sent separately and delivery is not guaranteed; in return it carries less overhead. On the same machine, **UNIX domain sockets** (`AF_UNIX`) are faster, because they do not pass through the network stack; they are named by a path name in the file system. Docker, systemd, and the X/Wayland display servers communicate with processes over these sockets.

3.7 Multitasking on mobile systems

On a desktop, an application left open keeps running even when the user switches to another window. On phones the situation is different. Memory is limited, swap space is either absent or small, and above all the battery is limited. An application that uses the processor continuously in the background can drain the battery within hours. Mobile operating systems therefore restrict background processes strictly.

iOS and iPadOS. The first iPhone software (2007) allowed only a single user application to run; everything other than the foreground application was closed. iOS 4 (2010) introduced limited multitasking. Even today only certain kinds of work can continue in the background: audio playback, location tracking, VoIP (voice calls over the Internet), and tasks scheduled with `BGTaskScheduler`. An application that uses none of these permissions is **suspended** shortly after it is moved to the background: the process stays in memory but is given no processor time. When memory is needed, the system can terminate suspended applications without warning; when the user returns, the application rebuilds its state.

Android. Android has allowed background processes from the start. Long-running work is carried out with the **service** component; a music player keeps playing thanks to its service even while its user interface is suspended. Since Android 8 (2017), however, there have been background restrictions. A long job must either run as a **foreground service** that shows the user a notification, or be split into `WorkManager` tasks that the system runs at a suitable moment.

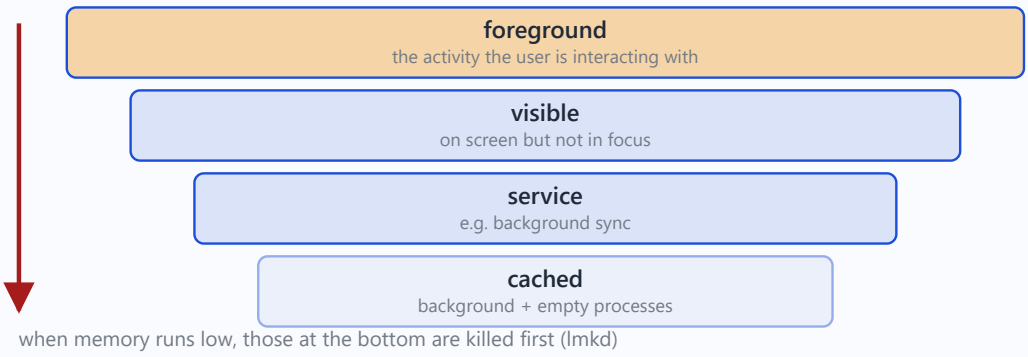


Figure 3.34: The importance order of processes on Android: foreground, visible, service, and cached processes. When memory runs low, `lmkd` kills processes starting from the lowest group.

When memory is tight, Android decides which process to kill by an importance ranking (Figure 3.34). At the top is the **foreground** process with which the user is interacting. It is followed by **visible** processes, which are on screen but not in focus, and **service** processes that do work in the background. At the bottom are the **cached** processes. The "background" and "empty" processes mentioned separately in older documents are now gathered in this group. When memory runs low, `lmkd` (low memory killer daemon) kills processes starting from the bottom. Killing a cached process is invisible to the user; when the application is opened again, it is started from scratch.

Although the two systems take different approaches, the reason is the same: limited memory and battery. The assumption on the desktop that "a process ends of its own accord" does not hold on mobile. Mobile applications are written on the assumption that they can be suspended or terminated at any moment, and they save their state regularly.

3.8 Summary

- A **process** is an instance of a program in execution. A program is a passive file sitting on disk; a process is an active entity with an address space, a processor context (program counter, registers), and resources (open files, identifiers). Many processes can arise from the same program.
- The address space of a process consists, from bottom to top, of the **text**, **data**, **BSS**, **heap**, and **stack** regions; at the very top is the kernel space, which user mode cannot access. The heap grows upward and the stack downward. Initialized global and `static` variables are in data, uninitialized ones in BSS, local variables on the stack, and memory allocated with `malloc` on the heap.
- A single core switches rapidly among processes and creates **pseudoparallelism**; this is called **multiprogramming**. **Concurrency** is tasks progressing in the same interval, and **parallelism** is tasks running at the same instant; it requires several cores.
- `fork()` creates a copy of the calling process; it returns the child's PID to the parent, 0 to the child, and −1 to the parent on error. The copying is done lazily with **copy-on-write**. n consecutive unconditional `fork`s produce 2^n processes. Unwritten data in the `stdio` buffer is duplicated by `fork`.
- The `exec` family replaces the process image with a new program; the PID, the parent, and the open file descriptors that were not closed are preserved. A successful `exec` does not return. All six functions end up in `execve()`.
- On every command the shell runs the **fork → exec → wait** loop. That `fork` and `exec` are separate lets the child prepare its own environment (redirection, directory, priority) before `exec`. `open` returns the lowest free descriptor number.
- A process terminates voluntarily (`exit`) or involuntarily (fatal error, signal). Only the low 8 bits of the exit status reach the parent. `SIGTERM` can be caught; `SIGKILL` cannot.
- A **zombie** is a finished process that has not been reaped; it consumes a PID. An **orphan** is a running process whose parent has died; `init` / `systemd` (or a subreaper) adopts it and reaps it when it ends. The remedy for accumulating zombies is `wait` / `waitpid` or a `SIGCHLD` handler.

- The **five-state model**: New → Ready ⇄ Running → Waiting → Ready … → Terminated. There is no Waiting → Running transition; a process whose time slice expires returns to Ready. Linux merges ready and running into `R` and splits waiting into `S` and `D`.
- The kernel tracks every process with a **PCB**; in Linux this structure is `task_struct`. The PCB carries process, memory, and file management information; the program counter and registers are the process's **context**. Queues are linked lists of PCBs.
- In the model of independent I/O waiting, CPU utilization is $\mathbf{1 - p^n}$; each additional process gains less than the one before.
- A **context switch** saves the old process's context and loads the new one's, and is pure overhead; its indirect cost (the cooling of the cache and the TLB) is often greater than its direct cost. In **limited direct execution**, the program runs on the processor at full speed; the timer interrupt returns control to the operating system.
- IPC has two models. **Shared memory** is very fast and involves no kernel after setup, but leaves coordination to the processes. **Message passing** goes through the kernel on every message, the kernel provides the coordination, and it also works over a network. Blocking `send` and `receive` form a **rendezvous**.
- A **pipe** is a one-way FIFO buffer in the kernel. Reading from an empty pipe blocks; when no open write end remains, `read` returns 0 (EOF). Unused ends must be closed. The shell builds pipelines with `pipe`, `fork`, `dup2`, and `exec`. A **socket** is an endpoint made of an IP address and a port.
- Because of limited memory and battery, mobile systems suspend background processes and terminate them if necessary; Android does this with `lmkd` according to an importance ranking.

Call	What it does	Return value
`fork()`	creates a copy of the calling process	the child's PID in the parent, 0 in the child, −1 on error
`exec…()`	replaces the program in the process	does not return on success; −1 on error
`wait()` / `waitpid()`	waits for a child to finish and reaps it	the PID of the finished child; 0 with `WNOHANG` if none has finished
`exit()`	flushes the buffers and terminates the process	does not return
`kill()`	sends a signal to a process	0 on success, −1 on error
`pipe()`	creates a pipe: `fd[0]` is the read end, `fd[1]` the write end	0 on success, −1 on error
`dup2(old, new)`	makes descriptor `new` a copy of `old`	`new`

Table 3.17: The basic UNIX calls of the chapter.

3.9 Key terms

process executable file address space program counter (PC) register thread multiprogramming concurrency parallelism parent / child process copy-on-write (COW) page fault shell

file descriptor system call signal zombie process orphan process process state
process control block (PCB) ready queue scheduler dispatcher
preemptive / nonpreemptive scheduling time slice (quantum) CPU-bound / I/O-bound
context switch interrupt interrupt vector trap timer user mode / kernel mode
interprocess communication (IPC) shared memory message passing race condition
synchronization synchronous / asynchronous communication buffer pipe socket

---

3.10 Test Yourself

1. When the following fragment is run, how many lines of `X` are printed and how many processes are created in total? What does the process tree look like?

```
if (fork() == 0)
    fork();
printf("X\n");
```

Answer

At the start there is P0. The first `fork` creates P1. In P0 the return value is the child's PID, that is, not 0; the condition is false and P0 goes straight to `printf`. In P1 the return is 0; the condition is true and P1 creates P2 with the second `fork`. P1 and P2 also reach `printf`. There are 3 processes in total (2 of them new) and 3 lines of `X`. The tree is a chain: P0 → P1 → P2.

2. What does the program print? Is the order of the output certain?

```
int x = 1;
pid_t pid = fork();
if (pid == 0) x = x * 3;
else { wait(NULL); x = x + 4; }
printf("%d\n", x);
```

Answer

The child computes `x = 1 · 3 = 3` in its own copy and prints `3`. The parent first waits for the child; then it computes `x = 1 + 4 = 5` in its own copy and prints `5`. Because the parent waits with `wait` for the child to finish before printing, the order is certain: first `3`, then `5`. The parent's `x` is never 3, because after `fork` there are two separate `x` variables.

3. The following program is run first in the terminal and then with `./prog > c.txt`. What is seen in the two cases? Why?

```
printf("first\n");
execlp("echo", "echo", "then", NULL);
printf("error\n");
```

Answer

In the terminal, `stdout` is line-buffered; `"first\n"` is written at once because of the newline. Then `exec` succeeds and `echo` prints `then`. Two lines are seen on the screen: `first`, `then`. With redirection to a file, `stdout` is fully buffered; `"first\n"` waits in the buffer in the process's memory. When `exec` discards the address space, the buffer is lost too. `c.txt` contains only `then`. `error` is not printed in either case, because `exec` succeeds. The remedy is to call `fflush(stdout)` before `exec`.

4. Processes spend half of their time waiting for I/O. How many processes must be in memory for CPU utilization to be at least 99%?

Answer

We need $p = 0.5$ and $1 - 0.5^n \geq 0.99$; that is, $0.5^n \leq 0.01$. $0.5^6 = 0.015625 \rightarrow$ utilization 98.44%; not enough. $0.5^7 = 0.0078125 \rightarrow$ utilization 99.22%; enough. With logarithms, $n \geq \ln 0.01 / \ln 0.5 \approx 6.64$; rounded up, 7 processes are needed.

5. A context switch takes 0.02 ms. There are 4 CPU-bound processes running in turn in the ready queue, and the time slice is 1 ms. What percentage of processor time goes to switches? At most how long does a process wait between two of its slices?

Answer

The overhead fraction is $0.02 / (1 + 0.02) = 0.02 / 1.02 \approx 1.96\%$. When a process's slice ends, 4 switches and the slices of the other 3 processes pass: $4 \cdot 0.02 + 3 \cdot 1 = 3.08$ ms.

6. In the `ps` output there are three processes whose `STAT` column is `S`, `D`, and `Z`. Which one can be terminated at once with Ctrl+C, which does not respond even to `kill -9`, and which is

pointless to kill? How is the last one cleaned up?

Answer

`S` is in interruptible sleep; a signal wakes it and it terminates on `SIGINT`. `D` is in uninterruptible sleep, usually waiting for a disk operation; even `SIGKILL` is not delivered until the operation finishes. `Z` is a zombie, that is, already finished; there is nothing left to kill. The zombie is reaped by its parent with `wait`. If the parent does not do this, the parent is terminated; the zombie passes to `init`, and `init` reaps it.

7. A child process exits with `exit(256)` in one run and with `exit(-1)` in another. Which values does the parent read with `WEXITSTATUS(st)`? Does the shell count the command as successful in the first case?

Answer

Only the low 8 bits reach the parent. 256 mod 256 = 0, so the parent reads 0. The low 8 bits of −1 are 1111 1111; the parent reads 255. In the first case the shell sees 0 and counts the command as successful, although the program meant to report an error. Portable programs therefore use only the range 0–255.

8. Before `exec`, a child process runs the following two lines. What is the value of `fd`? If `execlp("sort", "sort", NULL)` is then called, where does `sort` read its data from? Which shell notation does this correspond to?

```
close(0);
int fd = open("input.txt", O_RDONLY);
```

Answer

`close(0)` closes standard input and number 0 becomes free. Because `open` returns the lowest free number, `fd = 0`. `exec` preserves the file descriptor table; `sort` reads from standard input, that is, from number 0, which is now `input.txt`. This corresponds to the shell notation `sort < input.txt`.

9. A server creates a child with `fork` for every request but never calls `wait`. 50 requests arrive per second and every child finishes its work immediately. At most 32,000 processes can exist in the system at the same time, and there are 500 processes when the server starts. After how long does

`fork` begin to fail? What is the remedy?

Answer

Each child becomes a zombie when it finishes and keeps holding its PID. The free room is 32,000 − 500 = 31,500 processes. Because 50 zombies accumulate per second, no new process can be created after 31,500 / 50 = 630 seconds, that is, 10.5 minutes. The remedy is to reap the finished children in the `SIGCHLD` handler with a `waitpid(-1, NULL, WNOHANG)` loop.

3.11 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters "Processes," "Process API," and "Limited Direct Execution": they present the process abstraction, the `fork` / `exec` / `wait` pattern, and the regaining of control through the timer interrupt in short chapters. The book is online and free.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th edition, 2018), Chapter 3: treats process states, the PCB, and the IPC models more broadly, with Windows and Android examples.
- A. S. Tanenbaum and H. Bos, *Modern Operating Systems* (4th edition), Section 2.1: gives a detailed account of the process model, the fields of the process table, interrupt handling, and the multiprogramming model.
- M. Kerrisk, *The Linux Programming Interface* (2010), Chapters 24–28 and 44: a reference for `fork`, `exec`, `wait`, process termination, and pipes. It treats in detail Linux-specific pitfalls such as buffers, signals, and zombies.

3.12 References

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, pages.cs.wisc.edu/~remzi/OSTEP.
2. Chromium Projects, "Multi-process Architecture" and "Site Isolation" documents.
3. Android Developers, "Processes and app lifecycle".
4. M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010.
5. Linux manual pages: fork(2), execve(2), wait(2), pipe(7), signal(7), shm_overview(7).
6. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th edition, Wiley, 2018.
7. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th edition, Pearson, 2014 (5th edition 2023).

Threads

A process holds together all the resources of a running program: its address space, open files, and signal handlers. Often, though, several activities inside a single process must make progress at the same time. A browser draws the page while it waits for data from the network; a web server answers hundreds of requests at once. The thread is the answer to this need. Flows of execution that share the same address space but each run with their own program counter, registers, and stack provide concurrency without paying the cost of creating a process. These notes begin with the definition of a thread and the reason it is needed. They then use Amdahl's law to examine the limits of parallel programming on multicore processors. They compare user-level and kernel-level threads and the N:1, 1:1, and M:N mapping models. Thread creation with Pthreads and Java, thread pools, fork-join, and implicit approaches such as OpenMP and GCD are presented with code. The last section is devoted to the problems that threads bring with them: `fork()`, signals, cancellation, thread-local storage, reentrant code, and race conditions.

LEARNING OUTCOMES

Concept

Distinguish a thread from a process; say which state is shared by all threads and which is private to each thread.

Performance

Distinguish concurrency from parallelism, and use Amdahl's law to compute the speedup achievable with a given number of cores and its upper bound.

Models

Compare user-level and kernel-level threads and the N:1, 1:1, and M:N mapping models in terms of switching cost, blocking, and use of multiple cores.

Programming

Create threads with Pthreads and Java, pass arguments, obtain results, and wait for them; recognize implicit approaches such as thread pools, fork-join, OpenMP, and GCD.

Problems
Explain race conditions, non-reentrant code, thread-local storage, cancellation, signal delivery, and the behavior of `fork()` in a multithreaded process.

Real systems
Interpret the Linux `clone()` call, Java 21 virtual threads, and the Go runtime in terms of the mapping models.

4.1 Basic concepts

The process concept is the operating system's basic tool for isolating programs from one another. Every process has its own address space; one process can neither see nor corrupt another's memory. This isolation is valuable for security, but it becomes an obstacle for tasks that must work together on the same data. If two processes are to share the same data, a shared memory region must be set up or messages must be sent; both are laborious and slow. The thread leaves isolation at the process level and allows several flows of execution inside a single address space.

4.1.1 What is a thread?

The execution of a program can be thought of from two separate angles. The first is resources: the program occupies a place in memory, opens files, makes network connections, starts child processes. The second is execution: the processor carries out the program's instructions in order, a program counter points to the next instruction, registers hold intermediate values, and the stack keeps track of function calls. The classic process concept combines these two angles: every process has both its resources and a single flow of execution. The thread concept separates the two.

Definition 4.1

A **process** is a program in execution. The owner of resources such as the address space, open files, child processes, signal handlers, and identity information is the process.

A **thread** is a flow of execution within a process. It has its own program counter (PC), register set, stack, and state. The thread is the basic unit of processor use and scheduling.

Tanenbaum sums up this distinction in a single sentence: the process groups resources together, and the thread is the entity that is executed on the processor. A process contains one or more threads. All of them see the same address space; a value that one writes to a global variable can be read by another at once. On the other hand, each thread can be at a different place in the program, call different functions, and work with different local variables.

The operating system keeps a **process control block** (PCB) for every process. A similar structure is needed for threads: the **thread control block** (TCB). The TCB holds the thread's identifier (tid), its state, the saved PC and register values, the stack pointer (SP), its priority, and a pointer to the thread-local storage area. The resource information is not in the TCB but in the PCB of the process to which the thread belongs. Figure 4.1 shows this relationship.

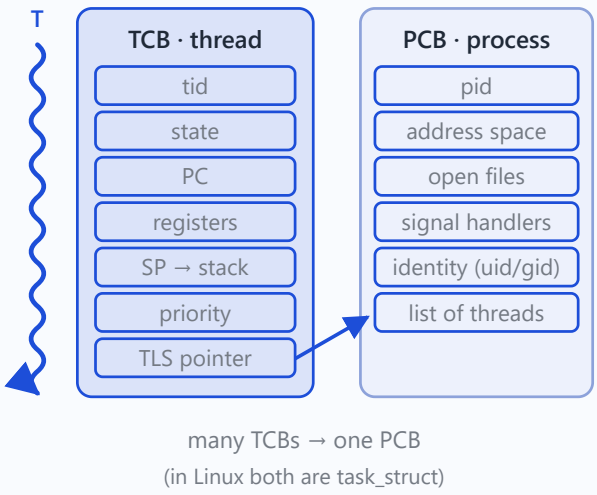


Figure 4.1: The relationship between the TCB and the PCB. Each thread's TCB holds the tid, state, PC, registers, stack pointer, priority, and TLS pointer, and points to the PCB of the process it belongs to. The PCB holds the pid, address space, open files, signal handlers, identity (uid/gid), and list of threads. Many TCBs point to a single PCB. In Linux both are represented by the same `task_struct` structure.

An old name for threads is **lightweight process** (LWP). The word "lightweight" points to the cost. Creating a new process requires setting up or copying the address space, preparing page tables, and duplicating the file table. A new thread needs only a stack and a TCB; the address space already exists. For the same reason, switching between two threads is also cheaper than switching between two processes.

Common mistake

Thinking of a thread as a "small process" and assuming that each thread has its own memory. Threads have no separate address space. Global variables, data allocated on the heap, and the code are common to all threads. Only the execution state is separate: the PC, registers, stack, and thread-local variables.

4.1.2 Why multiple threads?

Most of today's applications are multithreaded. A web browser, an integrated development environment (IDE), a game engine, a database server, or a web server runs with dozens of threads. There are several reasons for this.

- **Different tasks run in separate flows.** An application may have to update the screen, fetch data from the network, answer network requests, and check spelling at the same time. Each task is written in its own thread as sequential code, as if it were the only job in the program.
- **A waiting task does not stop the others.** While waiting for disk or network, that thread blocks, but the other threads in the same process keep making progress.
- **Threads are cheap.** Creating a process is expensive; creating a thread is much cheaper.
- **Threads simplify the program.** Each flow remains sequential code that can be read from start to finish. Doing the same job with a single flow forces the programmer to save by hand where each task left off.

The operating system kernel is multithreaded as well. The kernel threads whose names appear in square brackets when you look with the `ps` command on Linux are an example: `kthreadd` creates the other kernel threads, `kworker` threads carry out deferred work, and `ksoftirqd` handles software interrupts. Figure 4.2 shows three threads inside an application process and the kernel that schedules them.

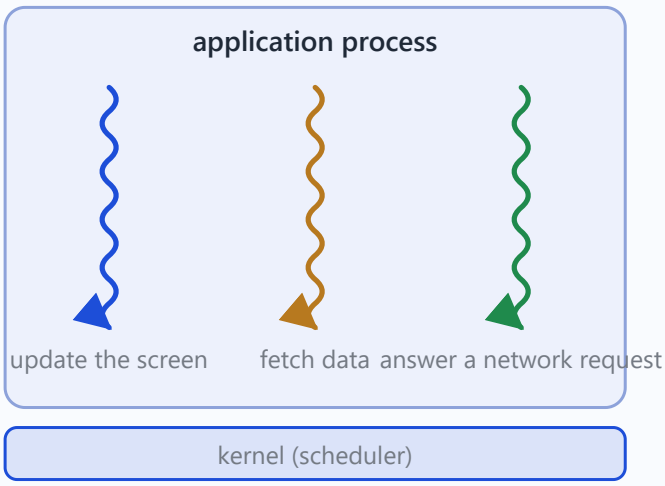


Figure 4.2: Three threads in an application process: one updates the screen, one fetches data, and one answers network requests. The scheduler in the kernel places all three on the processor.

4.1.3 Single-threaded and multithreaded processes

A single-threaded process has a single flow of execution: one PC, one register set, and one stack. The program can be in only one place at a time. In a multithreaded process the code, global data, and open files are still one each; but each flow has its own registers and its own stack. Figure 4.3 shows the two

cases side by side.

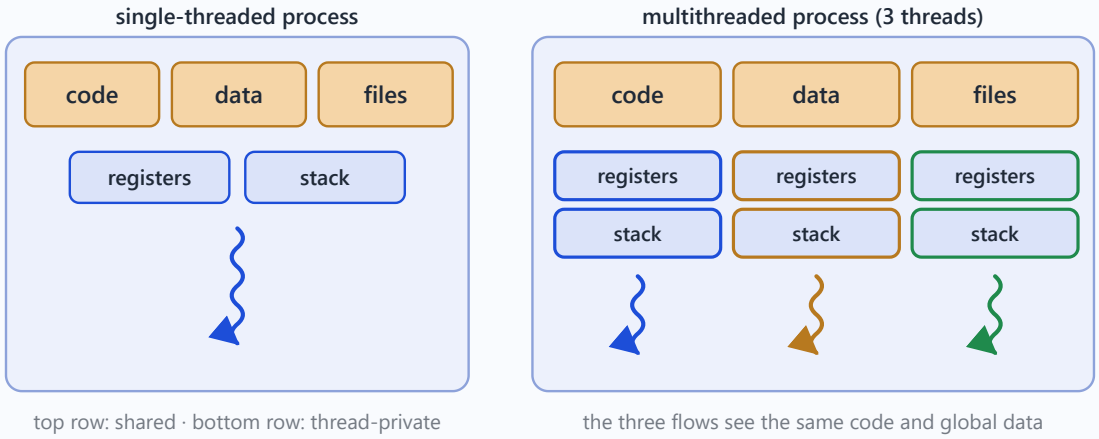


Figure 4.3: A single-threaded process on the left and a three-threaded process on the right. The orange boxes in the top row (code, data, files) are shared resources. The registers and stack in the bottom row are private to each thread. The wavy arrows show the flows of execution; the three flows see the same code and the same global data.

The distinction in the figure is the source of almost every problem with threads. Everything in the shared row is common data that all threads can access at the same time. What one thread writes to a global variable, the other threads see at once. This makes cooperation easy; but if two threads try to change the same variable at the same time, the result can be corrupted. The items in the private row, on the other hand, let each flow continue its own work without losing it.

4.1.4 Shared and private items

Knowing which item is kept per process and which per thread is the first step toward writing a multithreaded program correctly. Table 4.1 lists this distinction.

Per process (shared by all threads)	Per thread (private)
Address space: code, global variables, heap	Program counter (PC)
Open files, sockets	Registers
Child processes	Stack and stack pointer
Pending alarms (timers)	State: running, ready, waiting, terminated
Signal handlers	Thread ID, signal mask
Identity and accounting information	Thread-local storage (TLS), `errno`

Table 4.1: Items kept per process and per thread.

The left column of the table shows that the operating system manages resources at the process level. If one thread opens a file, the file descriptor is valid for the whole process; another thread can read and write with the same descriptor and can even close the file. Signal handlers are also process-wide:

a handler that one thread installs for `SIGINT` runs no matter which thread the signal is delivered to.

The items in the right column belong to the execution itself. Two threads can run the same function at the same time; since each one's local variables are on its own stack, they do not get mixed up. Variables such as `errno`, which "look global but are private to the thread," are provided by thread-local storage (4.6.5).

> **Common mistake**
>
> Thinking "the stack is private to the thread, so no other thread can touch it." The stack is private but **not protected**. All stacks are in the same address space; a faulty pointer can read or corrupt data on another thread's stack. The hardware does not prevent this, because the page table is the same for all threads.

A summary rule is enough for the rest of this chapter: **access to shared data requires synchronization**. Every global variable, every heap object, and every open file that several threads write is open to a race condition unless it is protected by a lock or an atomic operation (4.6.7).

4.1.5 Each thread has its own stack

Function calls and local variables must be kept separately for each flow. While thread A is inside a function `f`, thread B may call `g`. If both used the same stack, B's frame would overwrite A's frame, and when A returned from `f` it would jump to the wrong address. That is why every thread has its own stack. Figure 4.4 shows how the stacks are placed in the address space of a process with three threads.

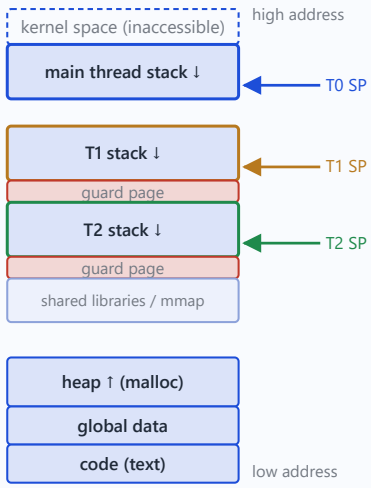


Figure 4.4: The address space of a process with three threads. At the very top is the inaccessible kernel space, below it the main thread's stack, and then the stacks of T1 and T2 with guard pages between them. In the

middle are the shared libraries and the `mmap` region; at the bottom are the heap growing upward, the global data, and the code. Each thread's SP points to its own stack.

On Linux, the glibc library places the stack of a new thread in an anonymous memory region allocated with `mmap`. The default stack size is usually the value of `ulimit -s`; on most distributions this is 8 MiB. Not all of this 8 MiB is placed in physical memory at once: thanks to virtual memory, only the pages that are touched actually consume memory.

Stacks grow downward, and if there is nothing between two stacks, an overflowing stack can overwrite its neighbor. To prevent this, a **guard page** is placed at the end of each stack. The guard page is marked inaccessible; the moment an overflowing stack touches this page, a page fault occurs and the process is terminated with the `SIGSEGV` signal. The error is thus noticed at once instead of silently corrupting the neighboring stack.

The stack size can be adjusted. In Pthreads, the desired size is given with `pthread_attr_setstacksize` before the thread is created. In Java, the `-Xss` option determines the stack size of every platform thread. For a thread that recurses deeply the stack may need to be enlarged, and for thousands of short-lived threads it may need to be reduced.

The **virtual threads** introduced in Java 21 change this situation. The stack frames of a virtual thread are kept not in a fixed-size memory region but in objects on the heap, and grow as needed. That a virtual thread can start with a stack of a few hundred bytes makes it possible for millions of virtual threads to exist at the same time (4.3.8).

Example 4.1

A server creates one platform thread for each connection; the default stack size is 8 MiB. For 10,000 connections, how many bytes of virtual address space are reserved for stacks alone?

1. One stack is 8 MiB = 8×2^{20} bytes.
2. 10,000 stacks: 10,000 × 8 MiB = 80,000 MiB ≈ 78.1 GiB.
3. A 64-bit address space can easily hold this, and pages that are not touched consume no physical memory. But at least a few pages of each stack are used; every thread also has a TCB and a kernel stack in the kernel. Tens of thousands of platform threads therefore spend noticeably more resources than virtual threads or an event-driven design.

4.1.6 Example: a three-threaded word processor

To show why threads are useful, Tanenbaum's classic example is a word processor. While editing a book of hundreds of pages, the user deletes a line on page one. Deleting this line requires all the following pages to be reformatted. At the same time, the user wants to keep typing, and the program must save the document to disk at regular intervals. Figure 4.5 assigns these three jobs to three threads.

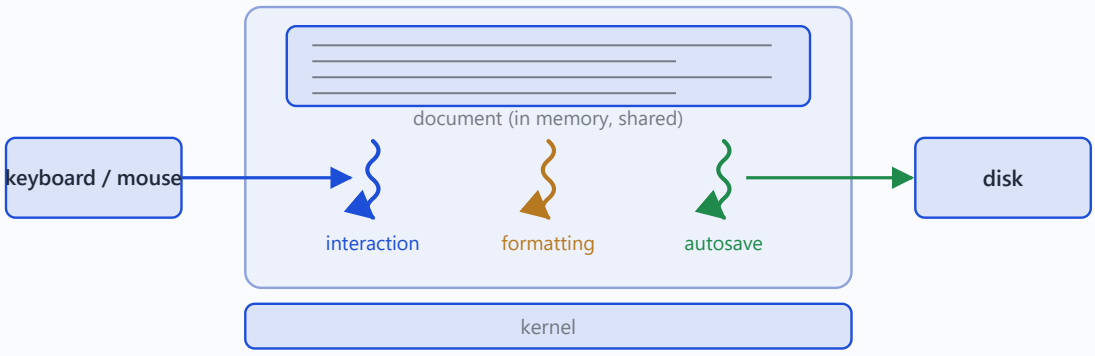


Figure 4.5: A three-threaded word processor. The document exists in memory as a single copy and the three threads share it: the interaction thread handles keyboard and mouse events, the formatting thread rearranges the pages, and the autosave thread writes the document to disk.

Interaction
: It handles key and mouse events immediately. No matter how much work there is in the background, the interface does not freeze.

Formatting
: When a line is deleted, it rearranges hundreds of pages in the background. When the user jumps to page 600, that page is most likely ready.

Autosave
: It writes the document to disk at regular intervals. It does not make the user wait while disk I/O is pending.

The same design could not be built with three separate processes. All three jobs must work on the **same document**; if the document were copied into three separate address spaces, every change would have to be passed on to the others. The shared address space is the key advantage here. A single-threaded design, on the other hand, would be left with two bad options: either keyboard input goes unanswered until formatting finishes, or the formatting code stops every few lines to poll the keyboard. The second option makes the code unreadable.

4.1.7 A multithreaded web server

The most familiar use of threads is in servers. A web server receives a large number of requests every second. Some of the requests are for frequently accessed pages, and these pages are ready in a **web page cache** in memory. For the other requests the page must be read from disk, which takes milliseconds. If the server handles every request in sequence, a request waiting for the disk makes all the requests behind it wait.

In Tanenbaum's design the server process has two kinds of threads. The **dispatcher** thread reads the requests coming from the network. For each request it picks an idle **worker** thread, hands the request over to it, and wakes it up. The worker first looks for the page in the cache; if it cannot find it,

it reads it from disk and sends the response. Figure 4.6 shows this arrangement at a moment when one of the workers is blocked on disk.

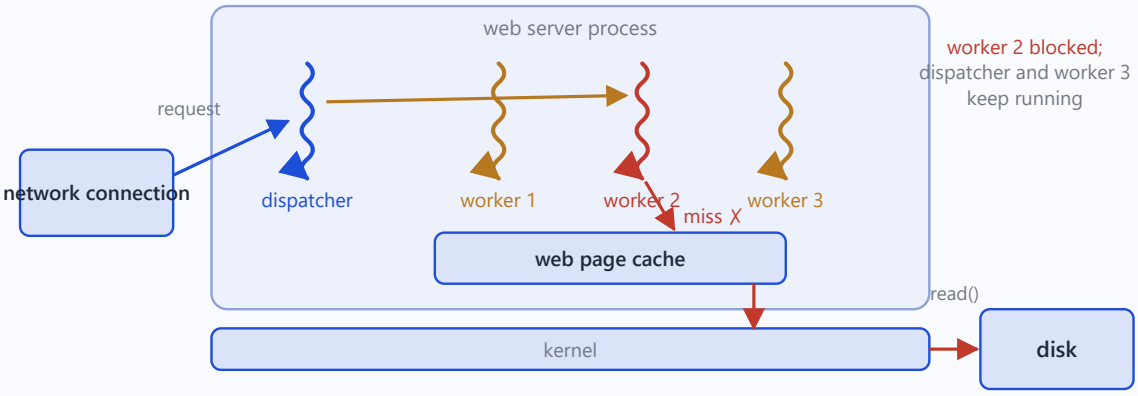


Figure 4.6: A multithreaded web server. The dispatcher has handed the request from the network to worker 2. Worker 2 did not find the page in the cache (✗) and is blocked in a `read()` call to read it from disk. Meanwhile the dispatcher and worker 3 keep running.

Four moments stand out in this design. A request arrives and the dispatcher reads it from the network. The dispatcher picks an idle worker, hands the request over to it, and wakes it up. If worker 1 finds the page in the cache, it sends the response at once; it does not go to disk. A new request goes to worker 2, and if the page is not in the cache, worker 2 blocks on disk. The last case is the important one: worker 2 blocking stops only worker 2. The dispatcher keeps accepting new requests, and worker 3 keeps handling another request.

The code of the two threads is extremely simple. The dispatcher reads the next request in an infinite loop and hands it to a worker:

```
while (TRUE) {
  get_next_request(&buf);  /* from net */
  handoff_work(&buf);      /* to worker */
}
```

`get_next_request(&buf)` blocks until a request arrives from the network and writes the request into the buffer `buf`. `handoff_work(&buf)` passes the request to an idle worker and wakes that worker. After that the dispatcher does not concern itself with what happens to the job; it goes straight back to the top of the loop. The worker's code is a similar loop:

```
while (TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

The worker blocks in `wait_for_work` until the dispatcher gives it a job. When a job arrives, it looks for the page in the cache (`look_for_page_in_cache`). If the page is not in the cache, it reads it from disk with `read_page_from_disk` ; this call blocks until the disk read finishes. Finally `return_page` sends the page to the client, and the worker goes back to the top of the loop to wait for a new job.

Both pieces of code are sequential and readable. The blocking `read` call stops only the worker that calls it. In a single-threaded server the situation is different: for a page that is not in the cache the whole process blocks, the processor sits idle while the disk is awaited, and requests that arrive in the meantime wait in a queue. In the multithreaded server the work is handed to a worker, and the server keeps accepting new requests.

4.1.8 Code trace: the dispatcher and the workers

Let us follow how the design works with a concrete scenario. The server process has three threads: the dispatcher and two workers, W1 and W2. Initially the dispatcher is listening on the network, and both workers are blocked in `wait_for_work` . Only the page `a.html` is in the cache. One request arrives first for `b.html` and then for `a.html` . The code being traced is the two loops put together:

```
/* dispatcher thread */
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
/* worker thread */
while (TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

Step	Event	Dispatcher	W1	W2	Disk	Cache
0	start	running	blocked (waits for work)	blocked (waits for work)	idle	a.html

Step	Event	Dispatcher	W1	W2	Disk	Cache
1	request 1 arrives: `b.html`	running	blocked (waits for work)	blocked (waits for work)	idle	a.html
2	`handoff_work`: request 1 → W1	running	ready	blocked (waits for work)	idle	a.html
3	W1 looks in the cache: miss	blocked (network)	running	blocked (waits for work)	idle	a.html
4	W1 `read_page_from_disk`	blocked (network)	blocked (disk)	blocked (waits for work)	reading	a.html
5	request 2 arrives: `a.html`	running	blocked (disk)	blocked (waits for work)	reading	a.html
6	W1 is busy: request 2 → W2	running	blocked (disk)	ready	reading	a.html
7	W2 finds `a.html` in the cache	blocked (network)	blocked (disk)	running	reading	a.html
8	W2 `return_page`: request 2 answered	blocked (network)	blocked (disk)	running	reading	a.html
9	disk finishes, W1 wakes up, request 1 answered	blocked (network)	running	blocked (waits for work)	idle	a.html, b.html
10	both workers wait for work again	blocked (network)	blocked (waits for work)	blocked (waits for work)	idle	a.html, b.html

Table 4.2: Trace of the dispatcher and the two workers. Each row shows the state after a step.

The key step in the trace is step 4. W1 blocks to read from disk, but only W1 stops; the other threads in the same process keep running. While the disk is awaited, the dispatcher takes the second request and, because W1 is busy, gives it to W2. W2 finds the page in the cache and answers. As a result **request 2 is answered before request 1**. When the disk read finishes, W1 wakes up, also puts the page into the cache, and sends it to the first client. Figure 4.7 shows steps 4 and 8 side by side.

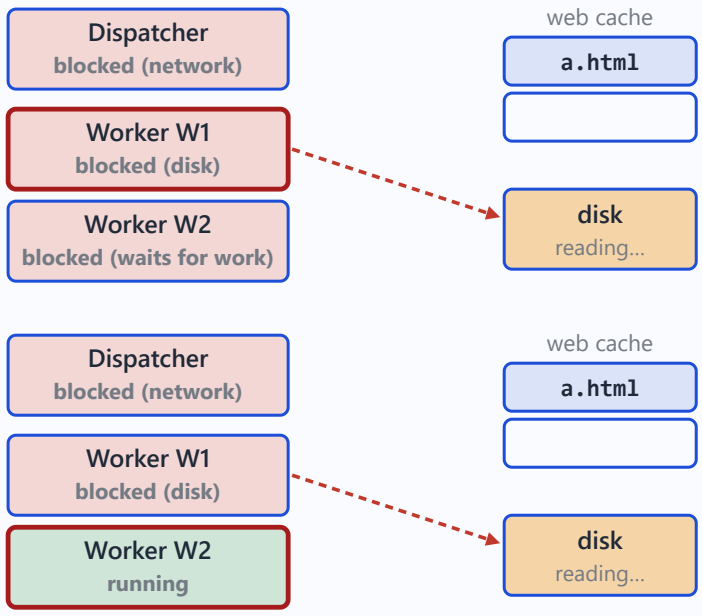


Figure 4.7: Two moments of the trace. (a) Step 4: W1 is blocked on disk, the disk is "reading," and the dispatcher waits on the network for a new request. (b) Step 8: while W1 is still blocked on disk, W2 returns `a.html` from the cache.

In a single-threaded server, request 2 would have waited until the disk read finished, even though its page was ready in memory. The multithreaded server uses this wasted time by overlapping waiting with computation. Each piece of code also stays sequential and readable.

Common mistake

Counting the answering of request 2 before request 1 as an error. In a multithreaded server, requests are completed not in the order they arrive but in the order they become ready. Because each request answers on its own connection, the change of order does not break correctness; on the contrary, it shortens the average response time.

4.1.9 Three ways to design a server

There are three basic options for how a server copes with blocking operations. Tanenbaum and Bos compare them as in Table 4.3.

Model	Parallelism / overlap	System calls	Programming
Multithreaded process	yes	blocking	sequential code, easy
Single-threaded process	no	blocking	easy but slow

Model	Parallelism / overlap	System calls	Programming
Finite-state machine (event-driven)	yes (I/O overlaps)	nonblocking + notification / interrupt	hard: the state of each request is saved by hand

Table 4.3: Three ways to design a server (the comparison in Figure 2-10 of Tanenbaum and Bos, *Modern Operating Systems*).

The third model needs some explanation. In an event-driven server there is a single thread, but this thread never blocks. When it needs to read from disk, it makes a nonblocking call and moves on to other work, knowing that it will receive a notification when the read finishes. Meanwhile it records in a table where the half-finished request stands, which page it is waiting for, and whom to answer. When the notification arrives, it finds that request in the table and continues where it left off. The program thus turns into a **finite-state machine**: each request is a small machine that moves from one state to another. I/O overlaps with computation, but the sequential code structure that the thread model provides is lost.

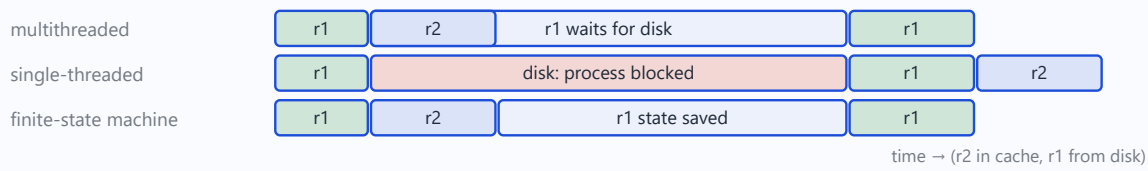


Figure 4.8: Timeline of the three models for two requests; r1 is served from disk and r2 from the cache. In the multithreaded server, r2 is handled while r1 waits for the disk. In the single-threaded server, the process blocks while the disk is awaited and r2 is left to the end. In the finite-state machine, r1's state is saved and r2 steps in.

Today the event-driven model is widespread: nginx, Node.js, and Redis use it. On Linux, `epoll` and `io_uring`, and on BSD and macOS `kqueue`, are interfaces for watching many connections with a single thread. Most real servers combine the two approaches: one event loop runs per core, and work that might block (e.g., disk access, compression) is sent to a thread pool.

Example 4.2

On a single-processor server, 3/4 of the requests are served from the cache and use 10 ms of processor time. The remaining 1/4 have a 50 ms disk wait in addition to 10 ms of processor time. The disk can serve many requests at once. How many requests per second does the server answer if it is single-threaded, and if it is multithreaded?

1. **Single thread.** While the disk is awaited, the process blocks and the processor sits idle. The average duration of a request is $3/4 \times 10 + 1/4 \times (10 + 50) = 7.5 + 15 = 22.5$ ms. $1000 / 22.5 \approx 44.4$ requests are answered per second.

2. **Multiple threads.** While the worker waiting for the disk is blocked, another worker uses the processor; the disk wait overlaps with computation. The bottleneck is the processor alone, and each request needs 10 ms of processor time. 1000 / 10 = 100 requests are answered per second.
3. **Comment.** Because there is a single processor, the gain does not come from parallelism; it comes from the overlap of blocking calls with computation. The same overlap can be achieved with a single event-driven thread, but the code becomes more complicated.

4.1.10 Benefits of multithreading

The examples so far bring out the four basic benefits of multithreaded programming.

Responsiveness
: The program keeps responding even if part of it blocks or performs a long computation. In user interfaces this property is decisive: if a long operation started by pressing a button runs in a separate thread, the window does not freeze.

Resource sharing
: Threads share the process's memory and files by default. For the same sharing between processes, a shared memory region or a message-passing mechanism must be set up; with threads nothing extra is done.

Economy
: Creating threads and switching between them is cheap compared with processes: the address space is not copied and the page tables do not change.

Scalability
: A single multithreaded process can run on several cores at the same time. A single-threaded process uses only one core, however many cores the machine has.

The figures often quoted in textbooks, that "creating a thread is 30 times and switching between threads 5 times faster than for processes," rest on an old Solaris measurement. These ratios do not hold on every system. On Linux, processes and threads are created with the same `clone()` call (4.6.9), so the difference between their creation costs is smaller. The real gain is that the memory mapping is not copied and the address translation cache is preserved during switching.

4.1.11 The cost of a context switch

The processor's move from one flow of execution to another is called a **context switch**. The difference between the two kinds of switch explains why threads are cheaper. Figure 4.9 compares the two kinds.

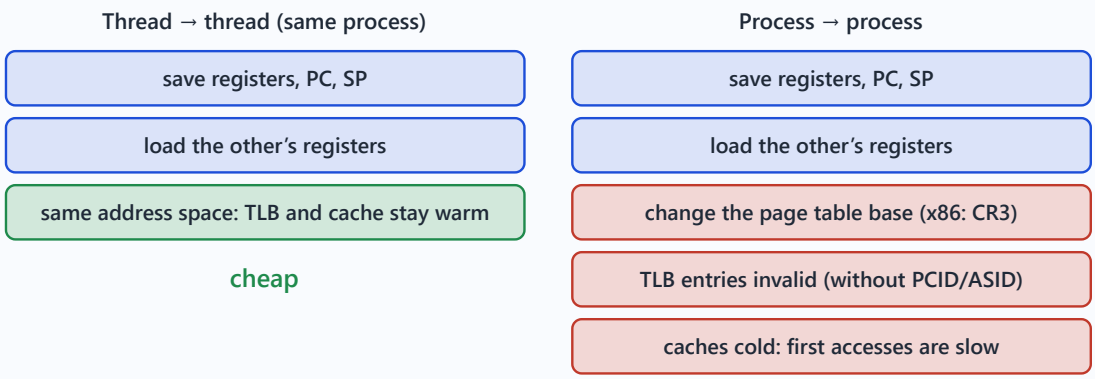


Figure 4.9: Two kinds of context switch. Between two threads in the same process, only the registers, PC, and SP are saved and loaded; because the address space stays the same, the TLB and cache stay warm. In a switch between processes the page table base also changes (the CR3 register on x86); without PCID/ASID support the TLB entries become invalid and the caches go cold.

In both switches the kernel saves the running flow's registers, PC, and SP and loads those of the new flow. In a switch between processes one more step is added: the base address of the page table is changed. On x86 processors this address is kept in the CR3 register. When the new address space is entered, the old entries in the **translation lookaside buffer** (TLB), which stores the records that translate virtual addresses into physical addresses, are no longer valid. If the processor cannot tag the entries with an address space identifier (PCID or ASID), the TLB is flushed. After that, the new process runs slowly until its first memory accesses refill the TLB and the data caches.

The cost has two components. The **direct cost** is the time to save and load the registers and to run the scheduler; on today's processors it is on the order of microseconds. The **indirect cost** comes from memory accesses that slow down after the switch because of cold caches and TLB misses. The indirect cost is often larger than the direct cost. In a switch between threads of the same process, most of the indirect cost disappears because the address space does not change.

For user-level threads (4.3.2), switching is cheaper still. The switch is done not by the kernel but by a library function in user space. There is no trap and no entry into the kernel; the switch comes down to an ordinary function call of a few instructions.

Common mistake

Thinking that the cost of a context switch is only the time to save registers. Saving the registers may take a few hundred nanoseconds, but the cache and TLB misses that follow can slow the program down for much longer. A program that runs more threads than necessary spends the processor on refilling caches instead of useful work because of frequent switches.

4.2 Multicore programming

Until about 2005, processors came to market at higher clock speeds every year, and single-threaded programs sped up on their own. When power consumption and heat stopped this increase, manufacturers turned to putting several cores on the same chip. Today a laptop has 8–16 cores and a server 64–128. The way to benefit from these cores is to split the program into parts that can run at the same time. Threads are the most direct tool for this.

4.2.1 Concurrency and parallelism

Two words that are used interchangeably in everyday language carry different meanings in operating systems.

Definition 4.2

Concurrency is the progress of several tasks in the same time interval. The tasks need not run at the same instant; they can also progress interleaved on a single core.

Parallelism is the execution of several tasks at the same instant. It requires at least two cores or processing units.

Consider four tasks on a single-core machine. The scheduler gives each a short time slice: first T1, then T2, T3, T4, and T1 again. Because the slices are short, to the human eye the four tasks appear to progress together. In reality only one runs at any instant. This is concurrency but not parallelism. If the same four tasks run on a two-core machine, T1 and T3 share the first core and T2 and T4 the second. Now at every moment two tasks truly run at the same time: this is parallel execution. Figure 4.10 compares the two cases.

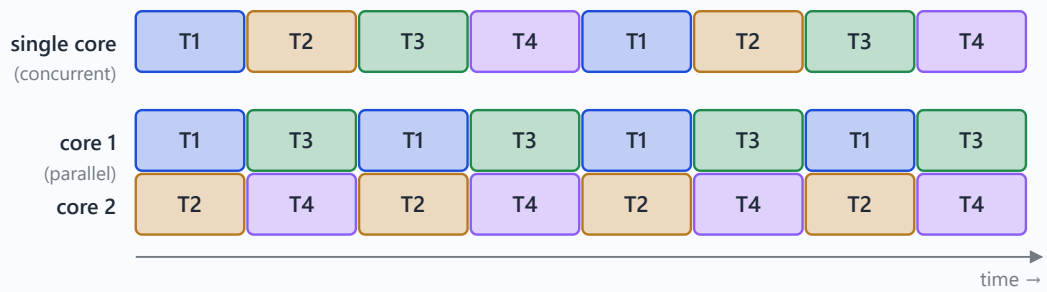


Figure 4.10: At the top, time sharing on a single core: T1–T4 run interleaved in turn; only one task runs at any instant (concurrent). At the bottom, two cores: T1/T3 on the first and T2/T4 on the second run at the same time (parallel).

Parallelism requires concurrency: two tasks running at the same time also progress in the same time interval. The converse is not true: a concurrent program can run on a single core without parallelism. The same program behaves differently on two different pieces of hardware. On a single core, threads only interleave; on multiple cores they truly run at the same time and can truly access shared data at the same time.

> **Common mistake**
>
> Thinking that using threads on a single core is pointless. Even without parallelism, concurrency is valuable: while one thread is blocked, another runs (the server in Example 4.2 sped up by more than a factor of two on a single processor). Likewise, it is wrong to think there are no race conditions on a single core: the time slice can end in the middle of a read-increment-write sequence.

4.2.2 The challenges of multicore programming

A larger number of cores does not speed up programs by itself. The programmer or the compiler must divide the work among the cores. Silberschatz groups the difficulties of this task under five headings; memory effects must be added to them.

Identifying tasks
: Finding activities in the application that are independent of one another and can run in parallel. Some algorithms divide naturally (the sum of an array); others proceed step by step and cannot be divided.

Balance
: Tasks should be of similar size. If one of four cores gets four times as much work as the others, that core determines the total time. Giving a separate core to a very small task does not pay either; the cost of starting the task exceeds the gain.

Data splitting
: The data that the tasks access must also be divided among the cores. If all the cores access the same data, parallelism turns into contention.

Data dependency
: If one task is waiting for the result of another, access must be synchronized. Dependencies reduce the part that can run in parallel.

Testing and debugging
: The order of execution may change in every run. A bug may appear once in a hundred runs and may never show up under a debugger.

Memory effects
: If different variables are on the same cache line and two cores write to them, the cache coherence mechanism keeps moving the line between the cores. Although logically nothing is shared, the cores slow each other down. This situation is called **false sharing**.

There are tools that help find races: ThreadSanitizer, enabled with the `-fsanitize=thread` option in GCC and Clang, Valgrind's Helgrind tool, and Java Flight Recorder. But no tool catches all races; tools examine only the accesses that occur in that run. Reducing sharing at the design stage is the most effective method.

4.2.3 Data and task parallelism

There are two basic ways of dividing work among cores. In **data parallelism**, the data is divided into pieces and each core performs the same operation on its own piece. In **task parallelism**, the work is divided into different tasks and each core performs a different operation; the tasks may use the same data or different data. Figure 4.11 shows the two ways on the same eight-element array.

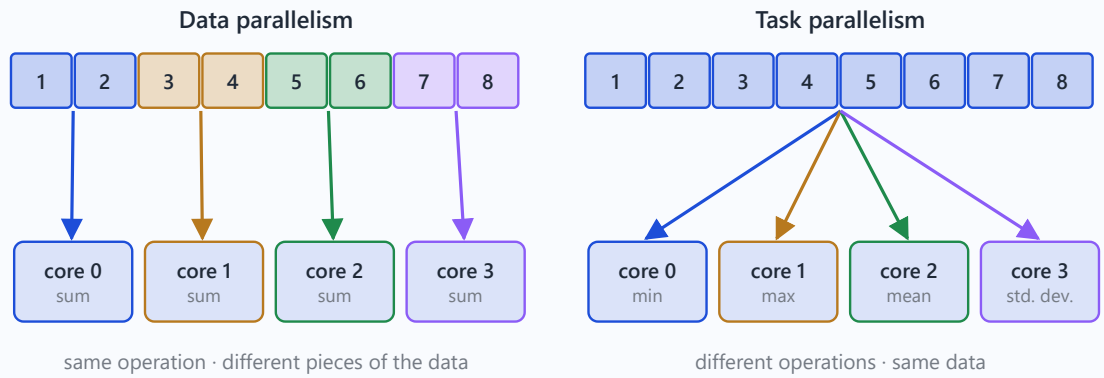


Figure 4.11: On the left, data parallelism: the array 1–8 is divided into four pieces and four cores perform the same operation (sum) on their own pieces. On the right, task parallelism: four cores perform different operations (min, max, mean, standard deviation) on the whole of the same array.

Summing an array of N elements with four cores is the typical example of data parallelism: each core sums N/4 elements, and in the end the four partial sums are combined. Graphics processors are the extreme case of this model; thousands of simple cores apply the same instruction to different pixels or matrix elements. A program that computes the statistics of an array is an example of task parallelism: one core finds the minimum, one the maximum, one the mean, and one the standard deviation. Real applications usually use both together.

4.2.4 Amdahl's law

How much can we speed up a program by increasing the number of cores? Almost every program has a part that cannot be parallelized: reading the input, combining the results, work done in sequence behind a lock. This part takes the same time no matter how many cores are added. Gene Amdahl's observation, made in 1967, is that this serial part sets a firm upper limit on the speedup.

Let the program's running time on a single core be 1 unit. A portion S of this time is serial, and $1 - S$ can be parallelized. On N cores the serial part still takes S; the parallel part is divided equally among N cores and takes $(1 - S) / N$. The **speedup** is the ratio of the old time to the new time.

Theorem 4.1 (Amdahl's law)

A program with serial fraction S speeds up on N cores by at most a factor of

$$\text{speedup} \le 1 / (S + (1 - S) / N)$$

As $N \to \infty$ the parallel term goes to zero, and the upper bound of the speedup becomes $1 / S$.

The inequality sign says that the formula is an optimistic bound. The formula assumes that the parallel part divides perfectly and that there is no communication or synchronization cost among threads. The real speedup falls below this bound. Another measure is **efficiency**: the ratio of the speedup to the number of cores, that is, how well each core is used.

Example 4.3

25% of a program's running time is serial ($S = 0.25$). What are the time and the speedup on 1, 2, and 4 cores? What is the upper bound of the speedup?

1. **N = 1:** time 0.25 + 0.75 / 1 = 1, that is, 100%. Speedup 1×.
2. **N = 2:** time 0.25 + 0.75 / 2 = 0.25 + 0.375 = 0.625, that is, 62.5%. Speedup 1 / 0.625 = 1.60×. The number of cores doubled, but the speedup is not 2×.
3. **N = 4:** time 0.25 + 0.75 / 4 = 0.25 + 0.1875 = 0.4375, that is, 43.75%. Speedup 1 / 0.4375 ≈ 2.29×. Efficiency 2.29 / 4 ≈ 57%.
4. **N → ∞:** the parallel part approaches zero and the serial 25% remains. The speedup is at most 1 / 0.25 = 4×.

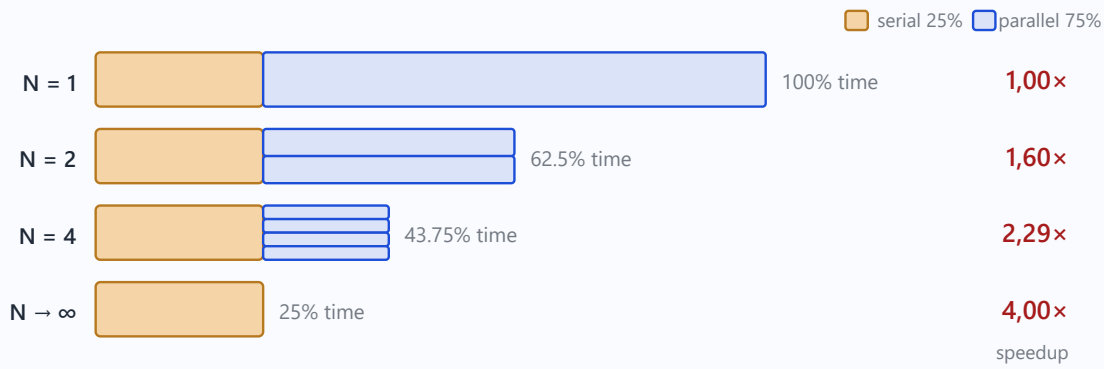


Figure 4.12: The time bars of Example 4.3. The orange part is the serial 25% and the blue part the parallel 75%. At N = 2 the parallel part is divided by two, and at N = 4 by four; the serial part never gets shorter. As N

→ ∞ only the serial part remains and the speedup stops at 4×.

Figure 4.12 gives the essence of the law visually. As cores are added, only the blue part gets shorter. After some point almost the whole bar consists of the orange part, and new cores hardly change the time. Even a serial part that looks small limits scaling noticeably.

Example 4.4

On an eight-core machine, a computation is to be sped up by at least a factor of 6. What is the largest the serial part can be?

1. Let us turn the speedup formula into an equation: 6 = 1 / (S + (1 − S) / 8). From this, S + (1 − S) / 8 = 1/6.
2. Multiply both sides by 8: 8S + 1 − S = 8/6, that is, 7S = 8/6 − 1 = 1/3.
3. S = 1/21 ≈ 0.0476. The serial part can be at most about 4.8%; at least 95.2% of the code must be parallel.
4. Check: 1 / (1/21 + (20/21) / 8) = 1 / (1/21 + 5/42) = 1 / (7/42) = 6.

Common mistake

Confusing the serial fraction with the parallel fraction. The *S* in the formula is the serial (non-parallelizable) fraction. If the question says “80% of the program can be parallelized,” then *S = 0.2*, not 0.8. Another common mistake is to confuse speedup with time: 0.625 is the time, and 1.60 is the speedup.

4.2.5 Amdahl curves and Gustafson’s law

The consequences of Amdahl’s law are even more striking when plotted for different serial fractions. Figure 4.13 shows the speedup on a logarithmic scale of core count for four programs whose parallel fractions are 50%, 75%, 90%, and 95%.

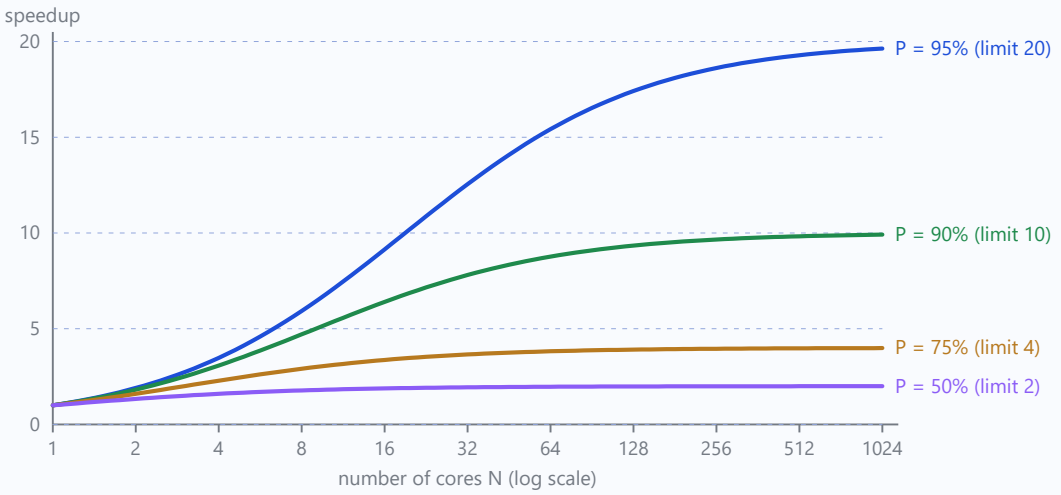


Figure 4.13: Amdahl curves. The horizontal axis is the number of cores (1 to 1024, logarithmic scale). For parallel fractions P = 95%, 90%, 75%, and 50%, the speedup approaches 20, 10, 4, and 2 respectively and flattens out there.

Serial fraction S	N = 2	N = 4	N = 8	N = 16	N = 64	N = 1024	Limit 1/S
0,50	1,33	1,60	1,78	1,88	1,97	2,00	2
0,25	1,60	2,29	2,91	3,37	3,82	3,99	4
0,10	1,82	3,08	4,71	6,40	8,77	9,91	10
0,05	1,90	3,48	5,93	9,14	15,42	19,64	20

Table 4.4: Speedup for different serial fractions (computed with Python, rounded to two decimals).

Two conclusions follow from the table. First, the limit is very sensitive to the serial fraction: reducing the serial part from 10% to 5% raises the limit from 10 to 20. Second, the return from adding cores diminishes quickly. Even if 95% of the program is parallel, the speedup on 1024 cores is only 19.6×; more than 98% of the cores are working in vain. In reality, because of synchronization and communication overhead, the speedup stays below even these values.

Amdahl's law assumes that the problem size stays fixed: the same job is done with more cores. In 1988 John Gustafson asked a different question: as the number of cores grows, the problem usually grows too. Someone doing weather forecasting who gets 1024 cores does not settle for making the same forecast faster; they make a more detailed forecast with a finer grid. In this case the parallel part grows with the cores, while the serial part stays roughly constant. With this assumption, **Gustafson's law** says that the **scaled speedup** is

> $\text{scaled speedup} = S + (1 - S) \cdot N$

Here *S* is the serial share of the running time of the enlarged problem on N cores. For example, for *S = 0.1* and N = 16 the scaled speedup is 0.1 + 0.9 · 16 = 14.5; Amdahl's law gives 6.4 with the same values. The two laws do not contradict each other; they answer different questions. Amdahl answers "how fast can I finish this job?"; Gustafson answers "how large a job can I do in the same time?"

4.2.6 Code trace: Amdahl speedup

The short C program below prints the speedup on 1, 2, 4, 8, and 16 cores for a program whose serial fraction is 10%, and at the end prints the upper bound:

```c
double S = 0.1;           /* serial fraction */
int N = 1;                /* cores */
while (N <= 16) {
  double sp = 1 / (S + (1 - S) / N);
  printf("N=%2d  %.2fx\n", N, sp);
  N = N * 2;
}
printf("limit %.0fx\n", 1 / S);
```

The program starts with `S = 0.1` and `N = 1`. The `while` loop runs as long as N does not exceed 16. In each round the speedup `sp` is computed with Amdahl's formula, printed with two decimal places by `printf`, and N is doubled. When N = 32 the condition is false, the loop ends, and the last line prints the limit `1 / S`. Table 4.5 traces each round of the loop.

Round	N	`S + (1 - S) / N`	sp	Output line	Increase over the previous round
1	1	0,1 + 0,9 = 1	1,00	`N= 1 1.00x`	—
2	2	0,1 + 0,45 = 0,55	1,82	`N= 2 1.82x`	×1,82
3	4	0,1 + 0,225 = 0,325	3,08	`N= 4 3.08x`	×1,69
4	8	0,1 + 0,1125 = 0,2125	4,71	`N= 8 4.71x`	×1,53 (%53)
5	16	0,1 + 0,05625 = 0,15625	6,40	`N=16 6.40x`	×1,36 (%36)
—	32	32 ≤ 16 is false: the loop ends; the last line is `limit 10x`			

Table 4.5: Trace of the Amdahl program (`S = 0.1`). The values were verified by recomputing them in Python.

The last column shows the diminishing returns with numbers. The number of cores doubles in each round, but the ratio of increase in speedup falls from 1.82 to 1.36. Going from 8 to 16 cores raises the speedup by only 36%. That the decimal separator in the output is a point comes from C's `printf` formatting.

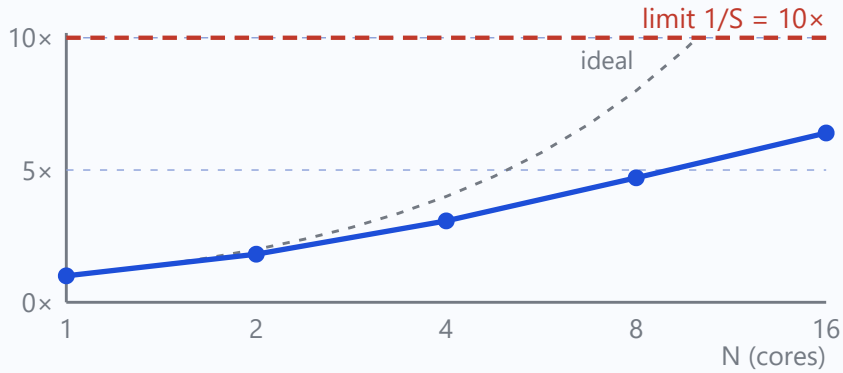


Figure 4.14: The speedup graph at the end of the trace. The blue curve shows the speedup on 1, 2, 4, 8, and 16 cores (1.00×, 1.82×, 3.08×, 4.71×, 6.40×). The dashed gray curve is the ideal speedup (sp = N); the red dashed line is the limit 1/S = 10×. The real curve approaches the limit but can never exceed it.

As N goes to infinity, the parallel part falls to zero, the serial 10% remains, and the speedup can never exceed 10×. Even on 16 cores the speedup is 6.4×; the efficiency is 6.4 / 16 = 40%. 60% of the cores are effectively spent waiting for the serial part.

4.3 Thread models

The question of who manages threads has two extreme answers. Either all management is done in a library in user space and the kernel knows nothing about threads, or the kernel recognizes every thread and schedules it itself. These two extremes and the hybrid solutions between them determine the cost of threads, their blocking behavior, and whether they can use multiple cores.

4.3.1 User-level and kernel-level threads

Definition 4.3

User-level threads are managed in a library that runs in user space, that is, in a **run-time system**. The kernel believes that each process has a single flow.

Kernel-level threads are threads that the kernel recognizes and schedules itself. Linux, Windows, and macOS use this approach.

Figure 4.15 shows where the thread table lives in the two approaches. At the user level, each process has its own run-time system and its own thread table; the kernel knows only the process table. At the kernel level, the thread table is in the kernel and contains the threads of all processes.

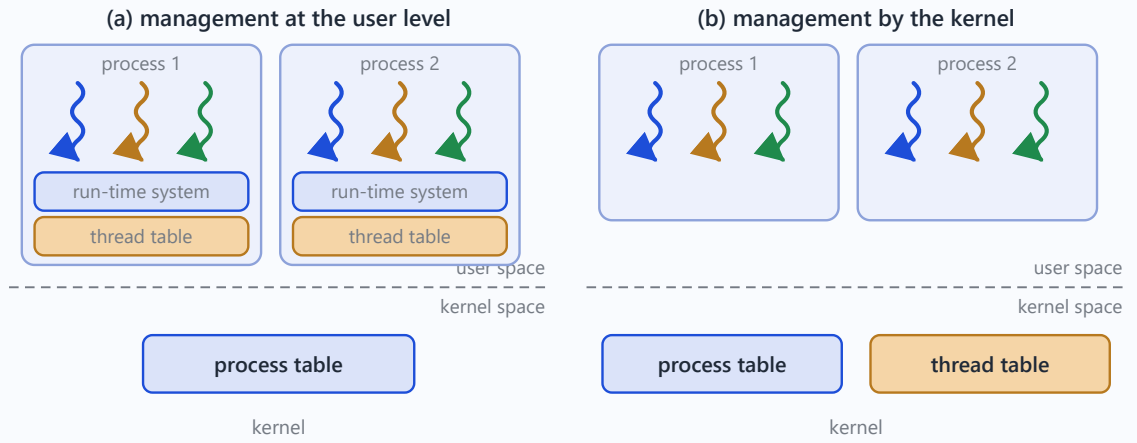


Figure 4.15: (a) Management at the user level: process 1 and process 2 each have a run-time system and a thread table; the kernel keeps only the process table. (b) Management in the kernel: both the process table and the thread table are in the kernel.

4.3.2 User-level threads

Managing threads entirely in user space may look odd at first, but this approach has serious advantages.

- **No kernel support is needed.** Even on a kernel with no thread support, threads can be implemented as a library. Many UNIX versions were like this until the 1990s.
- **Each process keeps its own table.** For each thread the table stores the PC, the registers, the SP, and the state. It is similar to the kernel's process table, but it lives in user memory.
- **Switching is very fast.** Switching from one thread to another needs no trap, no entry into the kernel, no change of memory mapping, and no TLB flush. A switch is a function call of a few instructions.
- **The scheduling algorithm can be chosen.** Each process can use a scheduling algorithm that suits it; for example, to run a garbage-collector thread at the most suitable moment.

The disadvantages are more fundamental, and they explain why user-level threads have been abandoned at the operating-system level.

- **A blocking system call stops the whole process.** If a thread calls `read` and blocks on disk, the kernel blocks the whole process, because for the kernel the process has a single flow. The other threads cannot run even if they are ready. The solutions are not elegant: using wrappers (jackets) that first test the call with `select` and switch to another thread if there is no data to read, or replacing all calls with their nonblocking versions.
- **A page fault also blocks the whole process.** If a page a thread accesses is not in memory, the kernel holds the process while it brings the page from disk; the library does not even know about it.

- **The clock interrupt does not reach the library.** The kernel can interrupt the process when its time slice expires, but which thread runs within the process is decided by the library. Because the library has no clock interrupt, if a thread does not give up the CPU voluntarily (`yield`), the other threads never run.
- **A single process cannot use several cores.** Because the kernel sees the process as a single flow, it places it on a single core at a time.

4.3.3 Code trace: `thread_yield` at the user level

To see how simple a user-level switch is, let us follow the `thread_yield` function of a library step by step. The code below is a simplified version of a real library. `PC` and `SP` are pseudo-variables standing for the processor's program counter and stack pointer; a real library reads and writes them with a few lines of assembly.

```
enum { READY, RUNNING, WAITING };
struct tcb { int pc, sp, state; } T[3];
int cur = 0;               /* running */
void thread_yield(void) {  /* in library */
  T[cur].pc = PC;  T[cur].sp = SP;
  T[cur].state = READY;
  do cur = (cur + 1) % 3;
  while (T[cur].state != READY);
  T[cur].state = RUNNING;
  PC = T[cur].pc;  SP = T[cur].sp;
}
```

`struct tcb` holds each thread's saved PC, SP, and state; `T[3]` is a table for three threads. `cur` is the index of the thread running at that moment. `thread_yield` first saves the running thread's PC and SP values into the table and makes it READY. The `do … while` loop then advances `cur` circularly and keeps going until it finds a READY thread. The thread found is made RUNNING, and its saved PC and SP values are loaded into the processor. The function's return instruction now returns on the new thread's stack; execution continues where the new thread left off.

Initially T0 is running; in the processor PC = 0x124 and SP = 0x8FF0. The T0 row of the table holds old values (0x100, 0x9000); these are left over from the last time T0 was saved. T1 is waiting for a lock in the library (WAITING), and T2 is ready and last gave up the processor at PC = 0x480, SP = 0xB000. T0 calls `thread_yield()` voluntarily.

Step	Line executing	T0 (pc, sp, state)	T1	T2	`cur`	CPU
0	(start)	0x100, 0x9000, X	0x300, 0xA000, W	0x480, 0xB000, R	0	PC=0x124 SP=0x8FF0

Step	Line executing	T0 (pc, sp, state)	T1	T2	cur	CPU
1	`T[cur].pc = PC; T[cur].sp = SP;`	0x124, 0x8FF0, X	same	same	0	same
2	`T[cur].state = READY;`	0x124, 0x8FF0, R	same	same	0	same
3	`cur = (cur + 1) % 3;`	same	same	same	1	same
4	`while (T[1].state != READY)` : W, the loop continues	same	same	same	1	same
5	`cur = (cur + 1) % 3;`	same	same	same	2	same
6	`while (T[2].state != READY)` : R, the loop ends	same	same	same	2	same
7	`T[cur].state = RUNNING; PC = …; SP = …;`	0x124, 0x8FF0, R	0x300, 0xA000, W	0x480, 0xB000, X	2	PC=0x480 SP=0xB000
8	T2 runs for a while and calls `thread_yield()` at PC = 0x48C: save, READY, `cur` = (2 + 1) % 3 = 0, T0 is READY	0x124, 0x8FF0, R	same	0x48C, 0xAFF8, R	0	PC=0x48C SP=0xAFF8
9	T0 is made RUNNING, its registers are loaded	0x124, 0x8FF0, X	same	0x48C, 0xAFF8, R	0	PC=0x124 SP=0x8FF0

Table 4.6: Trace of `thread_yield`. States: X = RUNNING, R = READY, W = WAITING.

In steps 1 and 2 T0's state is saved. This is an ordinary function call: there is no trap and no entry into the kernel. In steps 3–6 the table is scanned; T1 is skipped because it is waiting, and T2 is chosen because it is ready. In step 7 T2's registers are loaded. The whole switch ends in a few instructions. In step 8 T2 also gives up the processor voluntarily, and the turn comes around to T0 again. In step 9 T0 continues from address 0x124, that is, where it left off. Figure 4.16 shows the start and step 7.

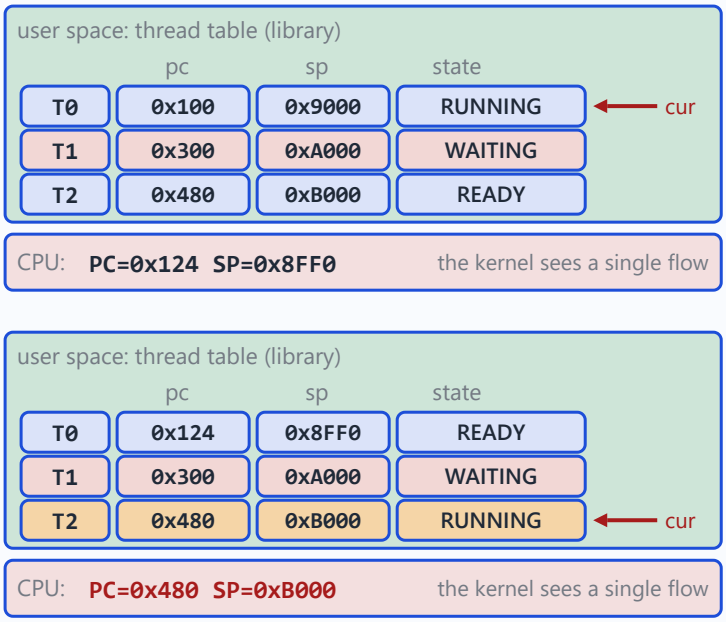


Figure 4.16: The thread table in user space. (a) Start: T0 is running, `cur` points to T0, and the processor has PC = 0x124. (b) Step 7: T0's values have been saved and it is READY, T1 is still WAITING, T2 is RUNNING, and the processor has been loaded with PC = 0x480, SP = 0xB000. The kernel sees a single flow throughout.

Common mistake

Assuming that user-level threads are also interrupted by the time slice. If T2 entered an infinite computation loop and never called `thread_yield()`, T0 would wait forever. The clock interrupt goes to the kernel; the kernel can interrupt the process and run another process, but when the process runs again, T2 continues where it left off. The library has no chance to step in.

4.3.4 The blocking call: N:1 and 1:1

The most serious problem of user-level threads is blocking system calls. To make the problem visible, let us put two models side by side. In the **many-to-one** (N:1) model, all the user threads of a process run on a single kernel thread. In the **one-to-one** (1:1) model, every user thread has its own kernel thread.

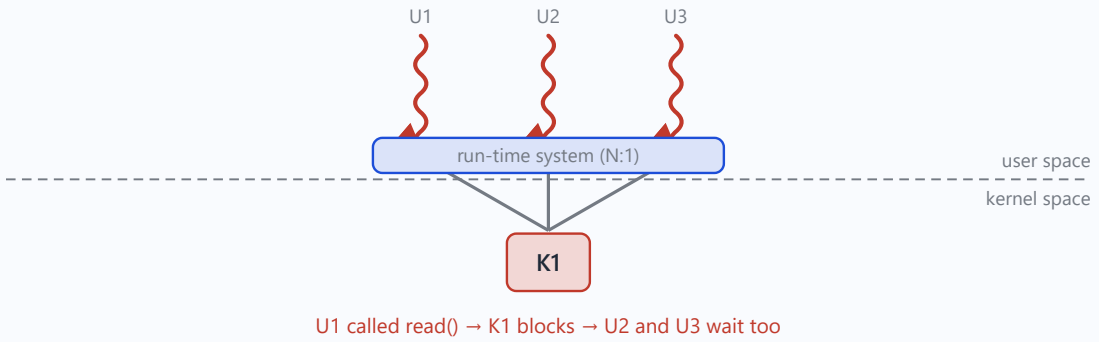


Figure 4.17: The many-to-one model. The user threads U1, U2, and U3 are mapped onto a single kernel thread (K1) through the run-time system. U1 calls the blocking `read()`; the kernel sees only K1 and blocks it, so U2 and U3 wait too.

In the N:1 model, when U1's `read()` call reaches the kernel, the kernel sees a single schedulable entity belonging to this process: K1. It blocks K1 until the read finishes. Even if U2 and U3 are ready, there is no kernel thread to run them; the whole process stops. In the 1:1 model the same call blocks only U1's kernel thread, K1. U2 and U3 have their own kernel threads (K2 and K3); the scheduler keeps running them in parallel on two cores.

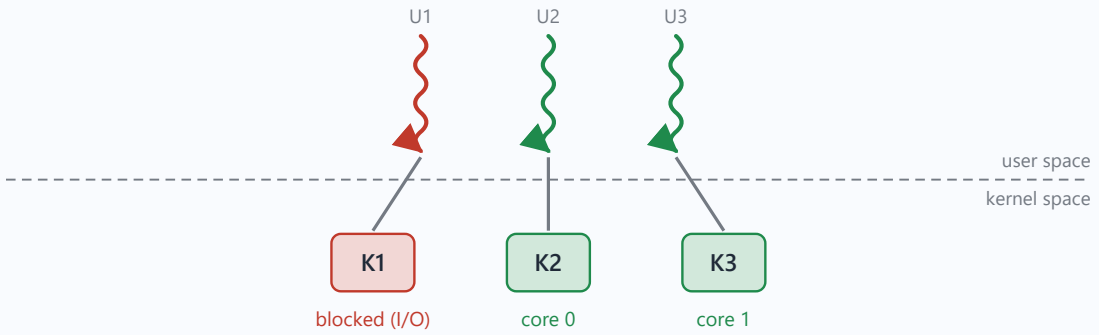


Figure 4.18: The one-to-one model. When U1 blocks on I/O, only K1 blocks; U2 and U3 keep running on core 0 and core 1 through K2 and K3.

4.3.5 Kernel-level threads

With kernel-level threads, the kernel keeps a thread table that contains the same information as the user-level table (registers, state, and the like). The consequences of this arrangement can be listed as follows.

- Creating and terminating a thread is a system call. It is therefore more expensive than at the user level.
- If a thread blocks, the kernel picks another. The chosen thread need not be from the same process.

- A page fault holds only the thread that caused it.
- The threads of a process truly run in parallel on different cores.
- Every thread takes up space in kernel memory: the TCB and the kernel stack. Their number is therefore not unlimited.

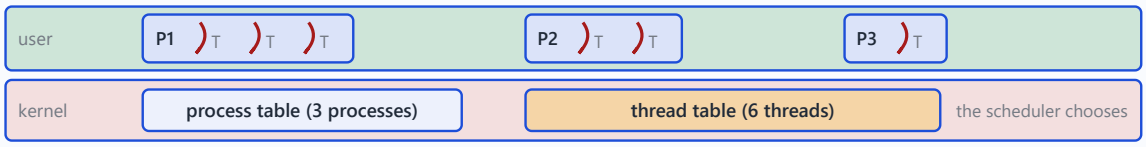
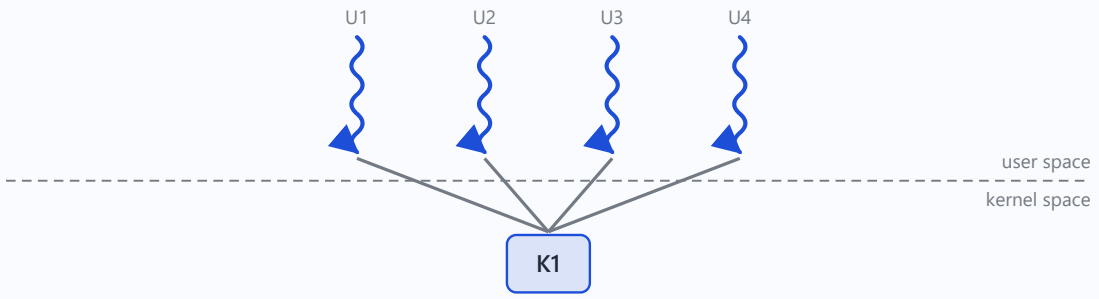


Figure 4.19: Kernel-level threads. In user space, P1 has three threads, P2 has two, and P3 has one. In the kernel, the process table holds three processes and the thread table holds six threads; the scheduler chooses among the threads.

Today Linux (NPTL, the Native POSIX Thread Library), Windows, and macOS use the 1:1 model. Because the cost of creating a kernel thread has fallen over the years, tens of thousands of kernel threads are not a problem. On Linux the number of threads is determined by three limits: the system-wide `/proc/sys/kernel/threads-max`, `ulimit -u` per user, and `pids.max` per control group. To observe the threads of a process, use `ps -eLf` (one line per thread), `top -H`, or the directory `/proc/<pid>/task/`; this directory contains a subdirectory for each thread.

4.3.6 Mapping models

How user threads are mapped to kernel threads falls under four models. Figure 4.20 shows all four side by side.



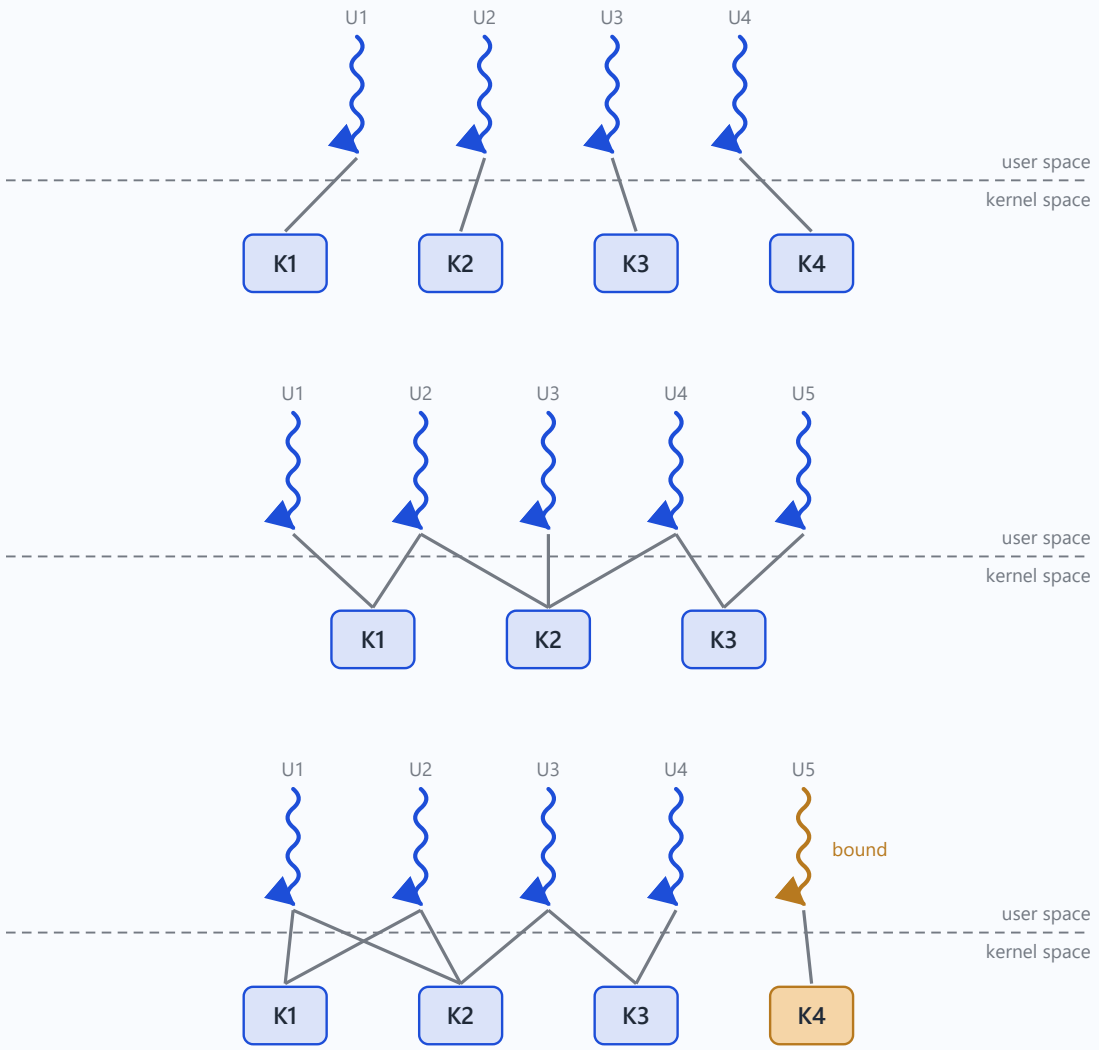


Figure 4.20: Mapping models. (a) Many-to-one: four user threads are mapped to a single kernel thread. (b) One-to-one: each user thread has a separate kernel thread. (c) Many-to-many: five user threads are multiplexed onto three kernel threads. (d) Two-level: in addition to the many-to-many mapping, U5 is bound directly to K4.

Many-to-one (N:1)

Switching is fast, but a blocking call stops the whole process and there is no parallelism. Examples: Solaris green threads and GNU Portable Threads; both are historical today.

One-to-one (1:1)

There is true parallelism and each thread can block independently. The price is that every thread uses kernel resources. Linux, Windows, and macOS use this model.

Many-to-many (M:N)

Many user threads are **multiplexed** onto a smaller number of kernel threads. Creating a user thread is cheap, several cores are used, and when a thread blocks the run-time system can run the others on other kernel threads. The problem is the difficulty of implementation: two

separate schedulers, one in user space and one in the kernel, make decisions without knowing about each other.

Two-level (hybrid)

In addition to M:N, a user thread can be permanently **bound** to a single kernel thread. A latency-sensitive thread thus gets a kernel thread of its own.

4.3.7 Comparing the models

Criterion	N:1	1:1	M:N
Switching cost	very low	medium (kernel)	low
Blocking call	the whole process stops	only that thread	only that thread*
Multiple cores	cannot be used	can be used	can be used
Number of threads	very many	limited by kernel memory	very many
Implementation	easy (library)	easy	hard (two schedulers)

Table 4.7: Comparison of the mapping models.

The starred cell depends on a condition: in the M:N model, a thread blocking affects the others only if the run-time system does not notice the blocking call and move the other user threads to another kernel thread.

At the operating-system level, the M:N model has been abandoned. Solaris 9 and FreeBSD 7 moved to the 1:1 model because of the coordination problems between the two schedulers and the complexity of the implementation. The M:N model, however, has returned in language run-times. Go's lightweight threads (goroutines), Erlang processes, and Java 21 virtual threads implement the M:N model not in the kernel but in the language's run-time. Because the run-time itself manages the calls that could block, the biggest problem of the old M:N implementations disappears.

Example 4.5

During its run, a program performs 500,000 thread switches. A user-level (N:1) switch takes 50 ns and a kernel-level (1:1) switch takes 2 µs. How much total time goes to switching in the two models? What happens when one of the threads performs a 10 ms disk read?

1. **N:1:** 500,000 × 50 ns = 25,000,000 ns = 0.025 s.
2. **1:1:** 500,000 × 2 µs = 1,000,000 µs = 1 s. The user-level switch is 40 times cheaper, because there is no trap and no entry into the kernel.
3. **Disk read, N:1:** The kernel sees a single kernel thread and blocks it. The whole process waits 10 ms; this time equals 200,000 user-level switches.
4. **Disk read, 1:1:** Only the reading thread waits. The other threads keep running during those 10 ms, even in parallel on other cores.

5. **Conclusion.** Switching cost alone does not decide the choice of model. In a program that does I/O, a single blocking call can wipe out most of the time that N:1 gains in switches. The M:N model tries to combine the two advantages: cheap switching and a run-time that does not block.

4.3.8 Current M:N: virtual threads and Go

The **virtual threads** that became permanent with Java 21 (Project Loom) are today's most widespread example of the M:N model. The JVM multiplexes many virtual threads onto a small number of platform threads. The platform thread that runs a virtual thread is called a **carrier**; carriers are ordinary kernel threads, and the operating system schedules them with the 1:1 model. Figure 4.21 shows these two layers.

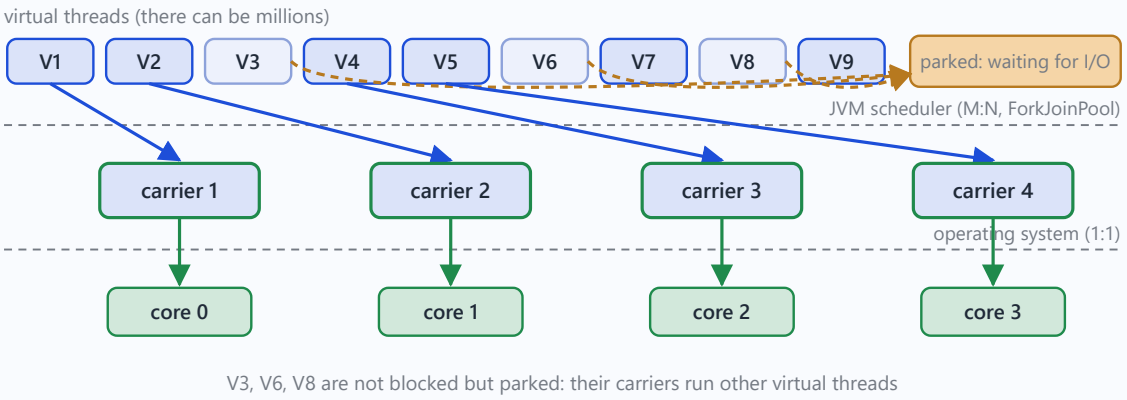


Figure 4.21: Java virtual threads. Some of the virtual threads V1–V9 are bound to four carriers; V3, V6, and V8, which are waiting for I/O, have been parked and moved to the heap. The JVM scheduler (M:N, ForkJoinPool) places virtual threads onto carriers; the carriers run on four cores in the operating system with the 1:1 model.

When a virtual thread makes a blocking I/O call, the JVM **unmounts** it from its carrier: it saves the stack frames to the heap and parks the thread. The carrier becomes free and runs another virtual thread. When the I/O finishes, the parked thread is mounted on a carrier again and continues where it left off. V3, V6, and V8 in the figure are not blocked but parked; their carriers run other virtual threads in the meantime. The kernel sees only the carriers. That is why creating millions of virtual threads is possible. A virtual thread is started with `Thread.ofVirtual().start(r)`.

The Go run-time describes the same idea with three letters. **G** is a Go thread (started with `go f()`). **M** is an operating-system thread (machine). **P** is a logical processor that represents the right to run Go code; its number is set by `GOMAXPROCS` and by default equals the number of cores. Each P has a queue of Gs waiting to be run. A P whose queue becomes empty **steals** work from another P's queue. If a G enters a blocking system call, the run-time detaches the P from that M and gives it to another M; the other Gs keep running.

4.3.9 Pop-up threads

In distributed systems, the traditional way of handling incoming messages is for a thread blocked in a `receive` call to wake up when a message arrives. The state of this thread, that is, its registers and stack, must be restored on every message. A faster option is to create a new thread to handle the message when it arrives. This thread is called a **pop-up thread**.

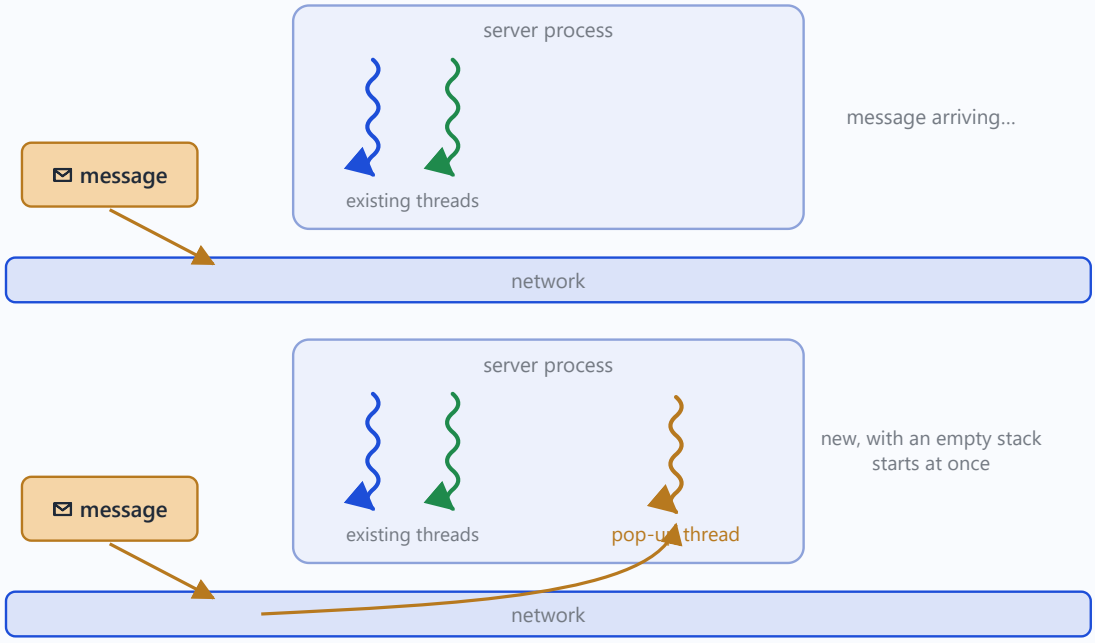


Figure 4.22: A pop-up thread. (a) Before the message arrives: there is no thread in the server process waiting for this message. (b) When the message arrives, the system creates a new thread to handle it; the thread starts at once with an empty stack.

A pop-up thread is new; it has no history, registers, or stack contents to restore. The delay between the arrival of the message and the start of its processing is therefore low. The same idea is used inside the kernel too: some kernels handle interrupts in short-lived threads created when the interrupt arrives. A pop-up thread can be created with Pthreads as follows:

```
/* on message arrival */
pthread_t t;
pthread_create(&t, NULL, handle, msg);
pthread_detach(t); /* no join */
```

`pthread_create` passes the message to the function `handle` as an argument. `pthread_detach` makes the thread **detached**: when the thread finishes, its resources are returned automatically and nobody needs to wait for it with `pthread_join`. Because it would be pointless for the server to wait one by one for the threads it creates for each incoming message, a detached thread is the natural choice here.

4.4 Thread libraries

The programmer creates and manages threads through a **thread library**. The library offers an application programming interface (API) for creating, waiting for, terminating, and synchronizing threads. The library can be entirely in user space; in that case calling a library function is an ordinary function call. Or the library relies on kernel support, and most of its functions result in a system call.

4.4.1 Libraries and two styles of use

POSIX Pthreads
: The C API of Linux, macOS, and BSD systems. Besides creating and waiting for threads, it also provides the mutual exclusion lock (mutex) and the condition variable.

Windows API
: Creates with `CreateThread` and waits with `WaitForSingleObject`. It is kernel-level and uses the 1:1 model.

Java
: Works with the classes `Thread`, `Runnable`, and `ExecutorService`. The Java virtual machine (JVM) uses the operating system's threads, and since Java 21 also virtual threads.

C and C++ standards
: C11 brought the `<threads.h>` header; C++11 brought `std::thread` and C++20 `std::jthread`. These replaced the Boost.Thread library, which was formerly widespread.

Python
: Provides the `threading` module. However, the global interpreter lock (GIL) in the standard interpreter allows only one thread at a time to run Python code; parallelism is therefore not achieved for processor-heavy work. The GIL-free build, which arrived experimentally in Python 3.13, became a supported option in 3.14.

Go
: The statement `go f()` starts a new Go thread. The run-time uses the M:N model; threads communicate through channels.

Threads are used in two different styles. In **asynchronous** use, the parent thread starts the child and continues with its own work; little data is shared between the two. The multithreaded web server is an example: the dispatcher gives the work to a worker and does not wait for its result. In **synchronous** use, the parent waits for its children to finish (`join`) and combines their results. This pattern is called **fork-join**: the work forks, the pieces run in parallel, and then they join again.

4.4.2 POSIX Pthreads

Pthreads is the thread creation and synchronization API defined by the IEEE 1003.1c (POSIX.1c) standard. An important detail is that Pthreads is a **specification**, not an implementation. The standard says how the library should behave; it leaves it to the implementer whether this is done at

user level or kernel level. On Linux, NPTL inside glibc implements Pthreads with the 1:1 model; on macOS and the BSDs too, Pthreads runs on kernel threads. A program that uses Pthreads is compiled with the command `gcc prog.c -pthread`.

Call	Description
`pthread_create`	Creates a new thread and runs the given function in it.
`pthread_exit`	Terminates the calling thread.
`pthread_join`	Waits for a given thread to finish.
`pthread_detach`	Makes the thread detached: when it finishes, its resources are returned automatically; it is not joined.
`pthread_self`	Returns the calling thread's own identifier.
`sched_yield`	Gives up the processor. (`pthread_yield` is nonstandard and was deprecated in glibc 2.34.)
`pthread_attr_init` / `_destroy`	Initializes or destroys the attribute structure that holds settings such as stack size and detached state.

Table 4.8: The basic Pthreads calls.

The signature of `pthread_create` shows the general style of Pthreads: `int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void *arg)`. The identifier of the new thread is written to the first parameter. The second parameter gives the attributes; `NULL` means default settings. The third parameter is the function the thread will run; this function takes a single `void *` and returns a `void *`. The fourth parameter is the argument to be passed to that function. The call returns 0 on success and an error code on failure.

4.4.3 Pthreads: create and wait

The first example is adapted from the program in Figure 4.11 of Silberschatz. The program adds up the integers up to an upper bound, which would come from the command line, in a separate thread; here the upper bound is given directly as the string `"5"`. The lines `#include <pthread.h>`, `<stdio.h>`, and `<stdlib.h>` are not shown.

```
int sum;              /* shared */
void *runner(void *param) {
  int upper = atoi(param);
  sum = 0;
  for (int i = 1; i <= upper; i++)
    sum += i;
  pthread_exit(0);
}
int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL, runner, "5");
  pthread_join(tid, NULL);
  printf("sum = %d\n", sum);
  return 0;
}
```

`sum` is a global variable; both threads therefore see it. `runner` is the function that the thread will run. It converts its parameter to an integer with `atoi`, puts it into the variable `upper` on its own stack, adds the numbers from 1 to `upper` to `sum`, and terminates with `pthread_exit(0)`. `main` starts the new thread with `pthread_create`, waits for it to finish with `pthread_join`, and prints the result.

Step	main	runner	sum	Description
1	running	not yet	0	The process starts with a single thread; the global `sum` is zero-initialized.
2	`pthread_create`	ready	0	A new thread is created; `runner("5")` is a candidate to run.
3	`pthread_join` → waiting	running	0	The scheduler decides when `runner` starts.
4	waiting	`upper = atoi("5")` = 5	0	`upper` is on runner's own stack.
5	waiting	loop: 1 + 2 + 3 + 4 + 5	15	The shared `sum` is updated.
6	waking up	`pthread_exit(0)` → done	15	runner ends; join returns.
7	`printf`	done	15	Output: `sum = 15`.

Table 4.9: Trace of the create-and-wait program.

The output of the program is the same on every run, because `main` reads `sum` only after `pthread_join` returns. Join guarantees that runner has finished all its writes. Without join, `main` could read `sum` before runner has even started and print 0. Worse, returning from `main` terminates the whole process; runner would die before it could finish its work. The observation in step 3 is important too: when `pthread_create` returns, there is no guarantee whatsoever that the new thread has started running. The scheduler decides which runs first.

Figure 4.23: Step 3 of the trace: `main` is waiting in `pthread_join`, and runner is running.

4.4.4 Multiple threads

To show data parallelism with Pthreads, let us divide the sum of the numbers from 1 to 1000 among four threads. Each thread adds 250 numbers.

```
#define N 4
long partial[N];         /* one cell each */

void *add_range(void *arg) {
  long k = (long)arg;    /* 0 .. N-1 */
  for (long i = k*250 + 1; i <= (k+1)*250; i++)
    partial[k] += i;
  return NULL;
}
int main(void) {
  pthread_t t[N];
  for (long k = 0; k < N; k++)
    pthread_create(&t[k], NULL, add_range, (void *)k);
  long total = 0;
  for (int k = 0; k < N; k++) {
    pthread_join(t[k], NULL);
    total += partial[k];
  }
  printf("%ld\n", total);  /* 500500 */
}
```

The thread number `k` is passed not as a pointer but as a **value**: the expression `(void *)k` places the number directly into the pointer field, and `(long)arg` in `add_range` takes it back out. Because `long` and a pointer have the same width, this conversion is safe. Thread number `k` adds the numbers from *250k + 1* to *250(k + 1)* into its own cell `partial[k]`. `main` creates the four threads, then joins each in turn and adds its partial sum to the main total.

k	Range	`partial[k]`
0	1 … 250	31.375
1	251 … 500	93.875
2	501 … 750	156.375
3	751 … 1000	218.875

k	Range	`partial[k]`
Sum		500.500

Table 4.10: The ranges and partial sums of the four threads (verified with Python).

The program has no race condition, because each thread writes only to its own cell and `main` reads a cell only after the owner of that cell has finished. Whatever order the threads run in, the result is always 500,500.

Common mistake

Passing the address of the loop variable, `&k`, as the argument. All four threads receive a pointer to the same variable `k`. Because the loop keeps incrementing `k` in the meantime, by the time a thread reads `*arg`, `k` may already be 2 or 4; two threads add the same range and one range is never added. When the loop ends, the lifetime of `k` ends too. The solution is to pass the value directly or to give each thread a separate memory cell.

There is also a performance trap in the code. The four cells of the `partial` array are next to each other in memory and fall on the same cache line. Because four cores write to this line continuously, the line goes back and forth between the cores: **false sharing**. If each thread accumulates the sum in a local variable first and writes it to `partial[k]` once at the end, the problem disappears.

4.4.5 Passing arguments and getting results

A thread function has a single `void *` parameter and a single `void *` return value. To pass or receive several values, the values are put in a structure and the address of the structure is passed. The program below passes the lower and upper bounds of a range in a structure and gets back the total and the count in a structure as well.

```c
typedef struct { int lo, hi; } range_t;
typedef struct { long total; int count; } result_t;

void *add_range(void *arg) {
  range_t *a = arg;           /* void* → type */
  result_t *s = malloc(sizeof *s); /* on heap */
  s->total = 0; s->count = 0;
  for (int i = a->lo; i <= a->hi; i++) {
    s->total += i; s->count++;
  }
  return s;         /* = pthread_exit(s) */
}
int main(void) {
  range_t a = { 1, 100 };
  pthread_t t;  void *ret;
  pthread_create(&t, NULL, add_range, &a);
  pthread_join(t, &ret);          /* ret = s */
  result_t *s = ret;
  printf("%d numbers, total %ld\n", s->count, s->total);
  free(s);
  return 0;
}
```

`main` creates the structure `a` on its own stack and passes its address as the argument. `add_range` converts the `void *` to its real type, `range_t *`. For the result, it allocates a `result_t` on the heap with `malloc`, zeroes its fields, and counts the numbers while adding those from 1 to 100 in the loop. The line `return s;` has the same effect as `pthread_exit(s)`: the pointer is delivered to whoever waits for this thread with join. The `pthread_join(t, &ret)` in `main` writes this pointer into `ret` through the second parameter of type `void **`.

Step	main stack	T stack	Heap
1	`a = {1, 100}`	no	—
2	`a` ; `pthread_create`	`arg` → `a`	—
3	`a`	`a = arg` , `s` → heap	`total = 0, count = 0`
4	`a`	`i = 101` (loop ended)	`total = 5050, count = 100`
5	`a`	frame removed	`total = 5050, count = 100`
6	`ret` → heap, `s = ret`	T finished	`total = 5050, count = 100`
7	`printf` , `free(s)`	T finished	(freed)

Table 4.11: Trace of the argument-and-result program.

The program prints `100 numbers, total 5050`. Two lifetime questions stand out in the trace. The first is the lifetime of the argument: `a` is on `main`'s stack, and this frame lives until `main` joins T. It is therefore safe to pass its address. The second is the lifetime of the result: T's stack disappears when T

finishes, but the object on the heap goes on living. Ownership passes to `main` through join, and returning the memory (`free`) is now `main` 's job. Figure 4.24 shows two important moments.

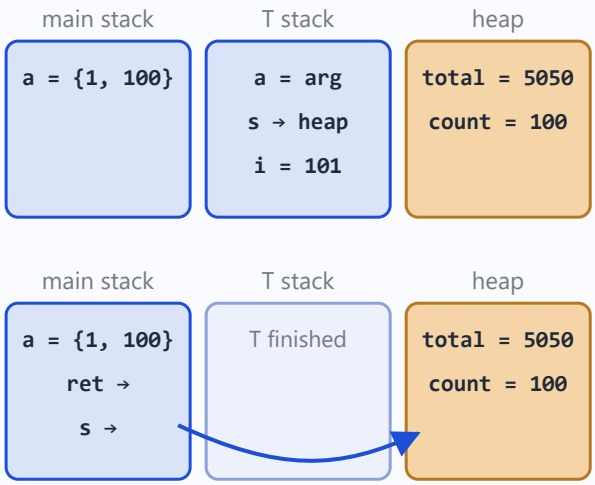


Figure 4.24: (a) Step 4: on T's stack the pointers `a` and `s` ; in the structure on the heap `total = 5050` and `count = 100` . (b) Step 6: T has finished and its stack is gone; `main` 's pointers `ret` and `s` point to the structure on the heap.

4.4.6 A common mistake: returning a result from the stack

Instead of allocating the result on the heap, keeping it on the thread's own stack and returning its address is a frequent mistake in Pthreads programs:

```
void *add_range(void *arg) {
  result_t s = { 0, 0 };  /* on T's stack */
  /* ... fill in s ... */
  return &s;              /* BUG */
}
```

In the faulty version the result is the local variable `s` on T's stack, and `&s` is returned. When the function returns, its frame is deleted. When the thread ends, its whole stack is released or may be given to another thread created later. When `main` reads the address it got from join, the result is undefined: a garbage value, `SIGSEGV` , or, worst of all, a value that sometimes looks correct. The last is the most dangerous, because the bug does not show in tests and appears in use. The correct version uses the heap:

```c
void *add_range(void *arg) {
  result_t *s = malloc(sizeof *s);
  if (s == NULL) return NULL;
  /* ... fill in s ... */
  return s;    /* on heap: lives on */
}
```

The heap version is correct, but it brings two responsibilities. If `malloc` finds no memory, it returns NULL; the result is therefore checked before use, and the thread returns NULL. The owner of the result is now `main`: after using the pointer it got with `pthread_join`, it must `free` it, or the memory leaks.

```c
void *ret;
pthread_join(t, &ret);
if (ret != NULL) {
  printf("total = %ld\n", ((result_t *) ret)->total);
  free(ret);
}
```

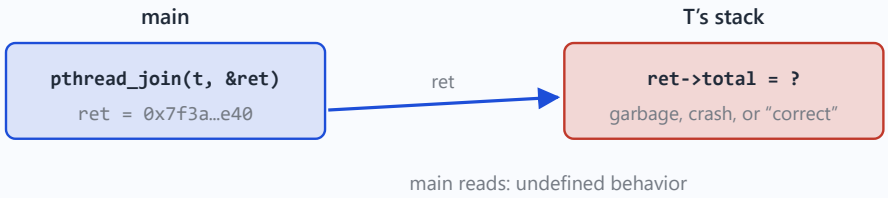


Figure 4.25: The end of the faulty version. T has returned, its stack frame has been deleted, and the stack has been released. `ret` in `main` still holds the old address (0x7f3a...e40); reading `ret->total` is undefined behavior.

The same rule applies to the argument: the address given to a thread must stay alive for as long as the thread uses it. In most cases, GCC gives the warning `-Wreturn-local-addr` for functions that return the address of a local variable; this warning should be treated as an error.

4.4.7 Java: `Thread`, `start`, and `join`

In Java every program starts with at least one thread, the main thread, which runs the `main` method. The shortest way to create a new thread is to give the `Thread` constructor the code to be run as a lambda expression. The program below does the sum of the Pthreads example in Java.

```
public class Sum {
  static int total = 0;
  public static void main(String[] a)
      throws InterruptedException {
    Thread t = new Thread(() -> {
      for (int i = 1; i <= 5; i++) total += i;
    });
    t.start();
    System.out.println("main: waiting");
    t.join();
    System.out.println("total = " + total);
  }
}
```

`new Thread(lambda)` only creates an object; no thread is running yet and the object's state is `NEW`. `t.start()` starts a new operating-system thread, and the lambda runs in that thread. Meanwhile `main` keeps printing; the two threads can run at the same time. At `t.join()`, `main` waits in the `WAITING` state until `t` finishes. When `t` finishes the loop, `total` is 15 and `t`'s state becomes `TERMINATED`.

Line	main	t
`new Thread(…)`	RUNNABLE	NEW (created, not started)
`t.start()`	RUNNABLE	RUNNABLE
`println("main: waiting")`	RUNNABLE	RUNNABLE
`t.join()`	WAITING (join)	RUNNABLE
(t finishes the loop)	waking up	TERMINATED
`println("total = " + total)`	RUNNABLE	TERMINATED

Table 4.12: `Thread.State` values of the threads in the Java program.

The program was compiled and run with the `javac` and `java` commands of JDK 21. Its two-line output is the same every time: first `main: waiting`, then `total = 15`. When two lines that print `t.getState()` were added to the program to observe the states, `NEW` was seen before `start` and `TERMINATED` after join.

The variable `total` is neither `volatile` nor protected by a lock; even so, it is guaranteed that `main` sees 15. The Java memory model says that all the writes a thread makes are visible to the thread that waits for it with join. This relationship is called the **happens-before** relationship. Besides waiting, join also provides visibility.

Common mistake

Calling `run()` instead of `start()`. `t.run()` is an ordinary method call: the lambda runs not in a new thread but inside the calling thread, that is, `main`. The program gives the right result but there is no concurrency at all. The difference is seen at once when `Thread.currentThread().getName()` is printed: inside `run()` the name is `main`. A `Thread` object can also be started only once; a second `start()` call throws `IllegalThreadStateException` (tried with JDK 21).

4.4.8 Java 21: virtual threads

To show why virtual threads matter, let us write a program that creates 10,000 tasks. Each task sleeps for one second in its own virtual thread; the sleep stands in for waiting for a network response.

```
import java.time.Duration;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class Virtual {
  public static void main(String[] args) {
    try (var ex = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, 10_000).forEach(i ->
        ex.submit(() -> {
          Thread.sleep(Duration.ofSeconds(1));
          return i;
        }));
    }  // close(): waits for all tasks
    System.out.println("done");
  }
}
```

`Executors.newVirtualThreadPerTaskExecutor()` returns an executor that creates a new virtual thread for each submitted task. `IntStream.range(0, 10_000)` generates the numbers from 0 to 9999, and for each of them `ex.submit` submits a task. The task sleeps for one second with `Thread.sleep` and returns its number. The `try` block is written in try-with-resources style: when the block ends, the executor's `close()` method is called automatically, and this method waits until all the submitted tasks have finished. Finally `done` is printed.

If the program ran sequentially it would take 10,000 seconds, that is, about 2 hours 47 minutes. In measurements made with JDK 21 on two different machines, all the tasks finished in about 2 seconds and in 1.2 seconds. The second machine had 12 logical cores, and the tasks ran on 12 carrier threads. The measured time varies from machine to machine, but the order of magnitude does not: each of the ten thousand tasks releases its carrier while waiting, and a few carriers suffice for all the tasks.

Running the same program with 10,000 platform threads means 10,000 kernel threads and 10,000 stacks. This works on most systems, but it is expensive in memory and kernel resources; when the number of tasks reaches the hundreds of thousands, the system limits are hit. The area where virtual threads are suitable is tasks that spend much of their time waiting for I/O: servers that serve thousands of concurrent connections. There is no gain for processor-heavy work, because in that case the bottleneck is the number of cores. Virtual threads can also be created the classic way: `Thread.ofVirtual().start(r)` starts a virtual thread, while `Thread.ofPlatform()` returns a builder that creates platform threads.

4.5 Implicit threading

4.5.1 Specifying the task and leaving the threads to the run-time

When the number of threads reaches hundreds or thousands, creating, waiting for, and synchronizing each one by hand becomes both harder and error-prone. **Implicit threading** takes this burden off the programmer. The programmer specifies only the **tasks**, that is, the pieces of work that can run in parallel. The compiler and run-time libraries create the threads, distribute the tasks to them, and manage them. Table 4.13 lists the common approaches.

Approach	Where	Idea
Thread pool	Java `ExecutorService`, the Windows thread pool	Ready threads take the tasks in the queue.
Fork-join	Java `ForkJoinPool`, parallel streams	Divide the task, solve the pieces in parallel, combine the results.
OpenMP	C, C++, Fortran	Parallel regions through compiler directives.
Grand Central Dispatch	macOS, iOS (libdispatch)	Blocks are put on dispatch queues.
oneTBB (formerly Intel TBB)	C++ template library	`parallel_for`, task scheduling, work stealing.

Table 4.13: Implicit threading approaches.

4.5.2 Thread pools

Suppose a multithreaded web server creates a new thread for every request. This design has two problems. The first is the cost of creating and then destroying a thread on every request; if the work is short, this cost can be larger than the work itself. The second is that there is no limit on the number of concurrent threads: a sudden burst of requests can create thousands of threads and exhaust memory and the processor.

Definition 4.4

A **thread pool** is a fixed (or bounded) number of threads that are created at the start and that take the tasks placed in a queue and run them in turn. A thread that finishes its task does not die; it takes the next task in the queue.

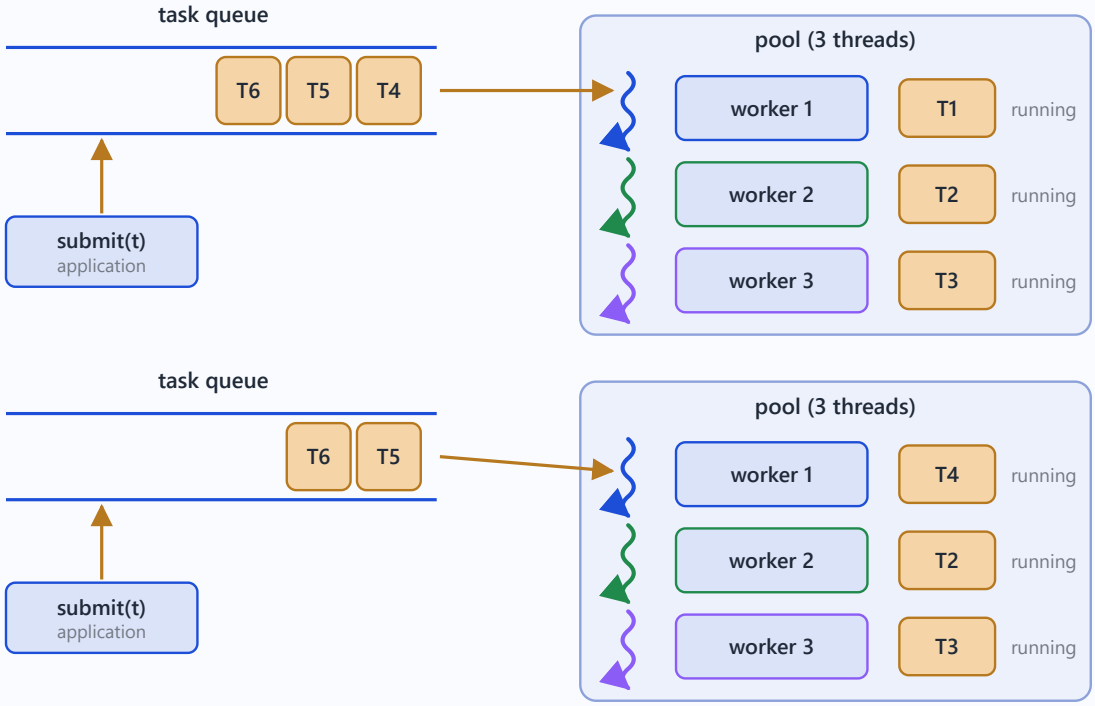


Figure 4.26: A pool with three threads. (a) The workers have taken T1, T2, and T3 from the queue and are running them; T4, T5, and T6 wait in the queue. (b) When worker 1 finishes T1, it has not died; it has taken the next task, T4.

The pool creates a fixed number of threads at the start, and the application puts tasks in the queue. Idle workers take tasks from the queue; the cost of creating a new thread is not paid. When a worker finishes a task, it immediately takes the next. When the queue empties, the workers wait until the next task arrives. The number of concurrently running threads never exceeds the pool size. The benefits of a pool can be summed up in three points:

- **Faster service.** A ready thread takes the task at once; the creation cost has been paid once.
- **It protects the system.** The pool size puts an upper bound on the number of threads that can exist at the same time. A burst of requests increases not the number of threads but the length of the queue.
- **It separates the task from the way it is run.** The task says only what is to be done. Whether it runs delayed or periodically, and with what priority, is decided by the pool.

A rough rule is used for choosing the pool size. For processor-heavy tasks the size should equal the number of cores; in Java this number is found with `Runtime.getRuntime().availableProcessors()`. More than that gains no parallelism and only adds context switching and cache contention. For I/O-heavy tasks a larger pool is needed, because most threads are waiting. If a task's wait time is W and its compute time is C, then about $N_{threads} \approx N_{cores} \times (1 + W / C)$ threads keep the cores busy.

Example 4.6

A task running on a four-core machine computes for 10 ms and then waits 30 ms for the response of a remote service. How many threads should the pool have?

1. The wait/compute ratio is W / C = 30 / 10 = 3.
2. $N_{threads} \approx 4 \times (1 + 3) = 16$.
3. Interpretation: there are four threads per core. While one thread computes for 10 ms and then waits 30 ms, the other three use the same core in turn, and the core stays busy all the time. This number is a starting point; the real value is tuned by measurement. A pool that is too large consumes memory, and one that is too small leaves the processor idle.

4.5.3 Java `ExecutorService`

Java's `java.util.concurrent` package offers pools through the `ExecutorService` interface. The code fragment below gives four tasks to a pool of two threads; each task computes the square of its own number.

```
ExecutorService pool = Executors.newFixedThreadPool(2);
List<Future<Integer>> results = new ArrayList<>();
for (int g = 1; g <= 4; g++) {
  final int n = g;
  results.add(pool.submit(() -> n * n));
}
int total = 0;
for (Future<Integer> f : results)
  total += f.get();       // waits for result
System.out.println("total = " + total);
pool.shutdown();
```

`Executors.newFixedThreadPool(2)` creates a fixed pool of two threads. The loop submits one task for each of `g` = 1, 2, 3, 4. Inside the lambda the loop variable cannot be used directly, because Java requires that the local variables captured by lambdas not change afterward; that is why a new copy is taken in every round with `final int n = g;`. `submit` puts the task in the queue and immediately returns a `Future` object. A `Future` is a placeholder for a result that may not yet be ready. In the

second loop, `f.get()` obtains the result of each task, waiting until it is ready if it is not. The `shutdown()` in the last line stops the pool from accepting new tasks; the running tasks finish and the threads terminate.

Task	`n`	`f.get()`	`sum`
1	1	1	1
2	2	4	5
3	3	9	14
4	4	16	30

Table 4.14: Collecting the `Future` results. Which worker runs the tasks and in what order may vary; the order of collection does not.

When run with JDK 21, the program prints the following:

```
total = 30
```

The result is deterministic: however the tasks are distributed to the two workers, the `get()` calls are made in the order of the list, and each returns the result of its own task. The `Executors` class has other factory methods too. `newCachedThreadPool` creates new threads as needed and closes idle ones after a while. `newScheduledThreadPool` is used for delayed and periodic tasks. `newVirtualThreadPerTaskExecutor` creates a new virtual thread for each task (4.4.8).

4.5.4 Fork-join

Fork-join is the parallel counterpart of divide-and-conquer algorithms. If a task is larger than a threshold, it is split in two; the pieces are forked as separate tasks and can run on different threads of the pool. Small tasks that have reached the threshold are computed directly, sequentially. The parent task waits for its children (join) and combines their results. Figure 4.27 computes the sum of eight numbers with this method.

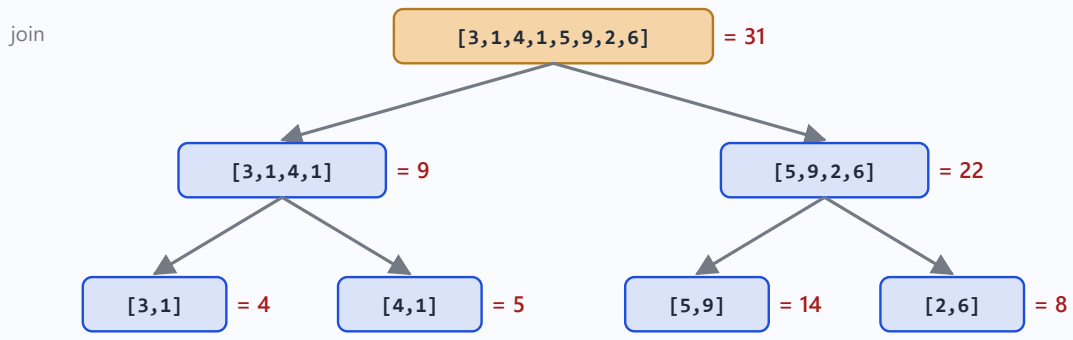


Figure 4.27: Summing the array [3, 1, 4, 1, 5, 9, 2, 6] with fork-join. The root task forks into [3,1,4,1] and [5,9,2,6]; these are in turn split into leaves of two elements each. The leaves are computed directly (4, 5, 14, 8), the join steps give 4 + 5 = 9 and 14 + 8 = 22, and the root gives 9 + 22 = 31.

The three stages of the method can be read from the figure. First the large task is split, because it is larger than the threshold. In the fork step two subtasks are created; they can run on different threads of the pool. The subtasks are split again, and the leaves that reach the threshold (two elements) are computed directly: [3,1] = 4, [4,1] = 5, [5,9] = 14, [2,6] = 8. In the join step each parent waits for its children and combines them: 4 + 5 = 9, 14 + 8 = 22. At the root the result is 9 + 22 = 31. Fork-join is the typical example of synchronous thread use: every parent waits for the result of its children.

4.5.5 `ForkJoinPool` and work stealing

For fork-join, Java offers the `ForkJoinPool` pool and the `RecursiveTask<V>` class in the `java.util.concurrent` package. A task that returns a result is derived from `RecursiveTask` and writes the `compute()` method. The class below sums the range `[lo, hi)` of an array; the fields `a`, `lo`, and `hi` and the constructor are not shown, for brevity.

```
class SumTask extends RecursiveTask<Long> {
  protected Long compute() {
    if (hi - lo <= LIMIT) {      // small: do it
      long s = 0;
      for (int i = lo; i < hi; i++) s += a[i];
      return s;
    }
    int mid = (lo + hi) / 2;
    SumTask left = new SumTask(a, lo, mid);
    SumTask right = new SumTask(a, mid, hi);
    left.fork();                  // start async
    long rightSum = right.compute();
    return left.join() + rightSum;
  }
}
long s = ForkJoinPool.commonPool()
           .invoke(new SumTask(a, 0, a.length));
```

`compute()` first compares the length of the range with the threshold. If the range is small, it sums it with an ordinary loop and returns. If it is large, it splits it in the middle and creates two subtasks. `left.fork()` puts the left half in the queue of the worker that is running at that moment and returns at once; the task has not run yet. The right half is computed in the same thread, directly with `right.compute()`. In the last line, `left.join()` waits for the result of the left half and the two results are added. `ForkJoinPool.commonPool()` is the JVM's shared pool; `invoke` starts the root task and waits for its result.

Running the right half directly in the same thread instead of forking it is a deliberate choice. If both halves were forked, the thread would wait for the two of them without doing anything. This way only one task enters the queue at each split, and the thread does not sit idle.

For an array filled with the numbers 1 to 100,000, the program prints `total = 5000050000` on JDK 21. This value equals *100,000 × 100,001 / 2*. The result is the same in a trial with a threshold of 1000 elements and a common pool of 11 threads.

How the pool balances the load matters too. In `ForkJoinPool`, each worker has its own **double-ended queue** (deque). A worker puts the tasks it forks at one end of its own queue and takes from that same end; this makes it continue with the most recently split, smallest task. A worker whose queue has become empty does not wait idle; it takes a task from the **other end** of another worker's queue. This is called **work stealing**. At the other end there are usually the oldest, largest tasks; a single steal therefore carries a lot of work, and the chance of colliding with the owner is low.

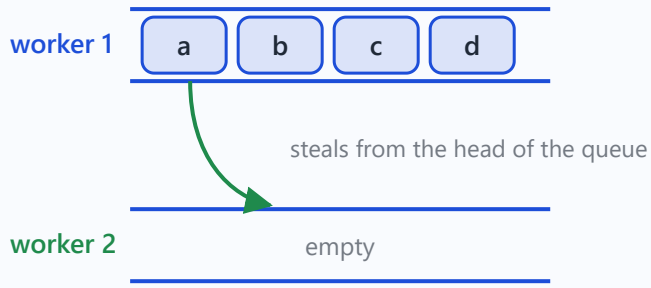


Figure 4.28: Work stealing. Worker 1's queue holds tasks a, b, c, d; the owner takes from its own end (the tail). Worker 2's queue is empty; it steals a task from the head of worker 1's queue.

4.5.6 Code trace: a sum with fork-join

To see exactly what fork and join do, let us trace a small array in a pool with a single worker: `a = {2, 7, 1, 8, 2, 8, 1, 8}`, `LIMIT = 2`, and the call `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`. Choosing a single worker makes the trace deterministic: nobody steals tasks. The method being traced is this:

```
protected Long compute() {    // LIMIT = 2
  if (hi - lo <= LIMIT) {
    long s = 0;
    for (int i = lo; i < hi; i++)
      s += a[i];
    return s;
  }
  int mid = (lo + hi) / 2;
  SumTask left = new SumTask(a, lo, mid);
  SumTask right = new SumTask(a, mid, hi);
  left.fork();                  // to deque
  long rightSum = right.compute();
  return left.join() + rightSum;
}
```

Step	Event	Queue	Computed
1	Root [0,8): 8 − 0 = 8 > LIMIT, it splits	empty	—
2	`left.fork()` : [0,4) goes to the queue; does not run at once	[0,4)	—
3	`right.compute()` : [4,8) in the same thread	[0,4)	—
4	[4,8) splits: [4,6) goes to the queue	[0,4) [4,6)	—

Step	Event	Queue	Computed
5	[6,8): 2 elements, at the limit; summed directly	[0,4) [4,6)	[6,8) = 1 + 8 = 9
6	`left.join()` : [4,6) is at the end of the queue and nobody has stolen it; the worker removes it and runs it itself	[0,4)	[4,6) = 2 + 8 = 10
7	[4,8) combines and returns to the root	[0,4)	[4,8) = 10 + 9 = 19
8	Root `left.join()` : [0,4) leaves the queue and runs; it splits, [0,2) goes to the queue	[0,2)	—
9	[2,4) leaf	[0,2)	[2,4) = 1 + 8 = 9
10	join runs the leaf [0,2); [0,4) combines	empty	[0,2) = 2 + 7 = 9; [0,4) = 9 + 9 = 18
11	The root combines	empty	[0,8) = 18 + 19 = 37

Table 4.15: Trace of fork-join in a single-worker pool. The queue is the worker's double-ended queue; the item on the right is the end of the queue nearest the worker.

The program was run with JDK 21 and printed `total = 37`. When a trace line that prints the range and the result was added at the finish of each task, the tasks finished in exactly the order of the table: [6,8) = 9, [4,6) = 10, [4,8) = 19, [2,4) = 9, [0,2) = 9, [0,4) = 18, [0,8) = 37. Seven tasks are created in total; four of them are leaves.

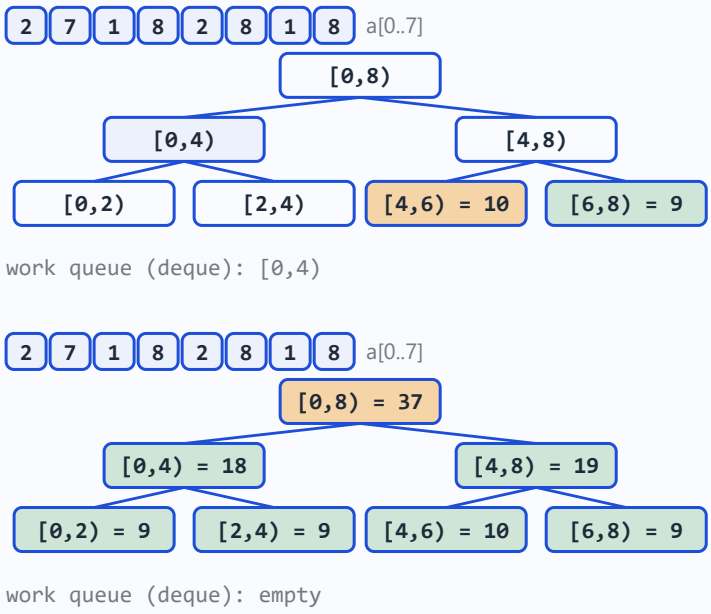


Figure 4.29: Two moments of the fork-join trace. (a) Step 6: [6,8) = 9 and [4,6) = 10 have been computed; only [0,4) remains in the queue. (b) End: all nodes have been computed, the root [0,8) = 37, the queue is empty.

If there were one more idle worker in the pool, it could have stolen the task [0,4) that entered the queue in step 2. In that case [0,4) and [4,8) would be computed truly in parallel, and the order in which tasks finish would change. The result would not change: every node is still the sum of its two children. This distinction comes up often in multithreaded programs: **the execution order may not be deterministic, but the result of a correctly synchronized program is deterministic**.

4.5.7 OpenMP

OpenMP is a set of compiler directives and libraries that makes parallel programming easier on shared-memory systems for C, C++, and Fortran. The programmer marks the regions that can run in parallel with `#pragma omp`; the threads are created by the compiler and the OpenMP run-time.

```
#include <omp.h>
#include <stdio.h>

int main(void) {
  #pragma omp parallel
  printf("hello from %d\n", omp_get_thread_num());

  double a[1000], b[1000], c[1000], s = 0;
  /* ... fill in a and b ... */
  #pragma omp parallel for reduction(+:s)
  for (int i = 0; i < 1000; i++) {
    c[i] = a[i] + b[i];
    s += c[i];
  }
  return 0;
}
```

`#pragma omp parallel` opens a **parallel region**: the statement that follows the directive runs separately in each member of a team of threads. The number of threads in the team defaults to the number of cores; it is changed with the `OMP_NUM_THREADS` environment variable or with `num_threads(n)` added to the directive. In this example the region is a single `printf` statement; each thread gets its own number with `omp_get_thread_num()` and prints a “hello” line. The order of the lines may change on every run. At the end of the region there is an implicit barrier: the program does not continue until all threads have finished.

The second directive, `#pragma omp parallel for`, divides the iterations of the `for` loop that follows among the threads. With four threads, each gets about 250 iterations. The line `s += c[i]` in the loop, however, is a problem: `s` is a single variable shared by all threads, and several threads adding to it at the same time creates a race condition. The `reduction(+:s)` clause solves this: each thread is given a private copy of `s`, and when the loop ends the copies are summed and added to the real `s`. The program is compiled with `gcc -fopenmp`; if this option is not given, the directives are ignored and the program runs sequentially.

4.5.8 Grand Central Dispatch and oneTBB

Grand Central Dispatch (GCD) is the combination of a run-time library (libdispatch) developed by Apple for macOS and iOS and a language feature added to C and Objective-C. The language feature is **blocks**: self-contained units of work that begin with a caret and are written inside curly braces, like `^{ printf("I am a block"); }`.

```
dispatch_queue_t q = dispatch_get_global_queue(
                        QOS_CLASS_USER_INITIATED, 0);
dispatch_async(q, ^{ puts("I am a block"); });
```

Blocks are put on **dispatch queues**. A serial queue runs blocks in order, one at a time; every application's main queue is serial, and user-interface work is done there. A concurrent (global) queue runs blocks in parallel. Global queues are divided into **quality of service** (QoS) classes; `QOS_CLASS_USER_INITIATED` in the example is for work that the user started and is waiting for. `dispatch_async` puts the block in the queue and returns at once. A block that leaves the queue is assigned to an idle thread in a system-managed pool. Thread management is entirely in GCD; the programmer creates no threads.

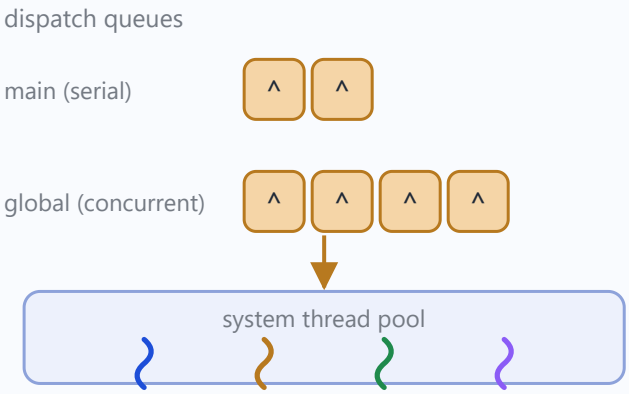


Figure 4.30: Grand Central Dispatch. Blocks (^) are put on the main (serial) queue or on the global (concurrent) queue; the threads in the system thread pool take blocks from the queues.

oneTBB is the open-source continuation of the Intel Threading Building Blocks library and comes as C++ templates. A call such as `tbb::parallel_for(0, n, [&](int i) { f(v[i]); });` divides the range of the loop into pieces and distributes the pieces to threads with its own scheduler, using work stealing. The programmer only says "the iterations of this loop are independent of one another"; the library decides how many threads to use and how large the pieces are. In Apple's Swift language, the language's own `async` / `await` mechanism has largely taken the place of GCD.

4.6 Thread issues

Many interfaces of the operating system were designed, before threads, on the assumption that every process has a single flow. `fork()`, signals, the global `errno` variable, and library functions that use static buffers are examples. When a process has several flows, a new question arises for each of these interfaces.

4.6.1 `fork()` and `exec()` in a threaded process

In a process with three threads, if one of the threads calls `fork()`, how many threads should the child process have? Some UNIX systems gave two answers to this question and offered two versions of `fork`: one copied all the threads, the other only the caller. POSIX gives a single answer: **only the thread that calls `fork` is copied**. The whole address space is copied, but only one of the threads.

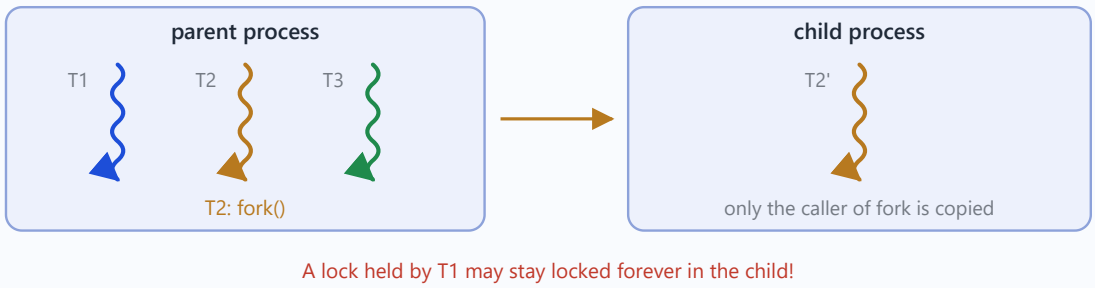


Figure 4.31: In a parent process with three threads, T2 calls `fork()`. According to POSIX, the child process contains only a copy of T2 (T2′). A lock that T1 held at that moment may stay locked forever in the child.

If `exec` is going to be called right after `fork`, this behavior is right: the new program replaces the whole address space anyway and starts with a single thread. Copying the other threads would be wasted work. The problem is the case where the child keeps running without calling `exec`. In the copied address space, the locks that the parent's other threads held at the time of `fork` are also locked. For example, if T1 holds `malloc`'s internal lock at that moment, there is no T1 in the child to release this lock, and the child's first `malloc` call waits forever.

For this reason, in the child of a multithreaded process POSIX allows only **async-signal-safe** functions (functions that can be called safely from a signal handler and use no internal locks) to be called until `exec`. `pthread_atfork` registers handlers that run before and after `fork`, which lets libraries bring their locks into an orderly state. If the goal is to start a new program, `posix_spawn`, which does `fork` and `exec` in a single step, is often the better choice.

4.6.2 Signal handling

A **signal** is a software notification that tells a process an event has occurred. In a single-threaded process it is clear where the signal goes. In a multithreaded process, the question is: to which thread should the signal be delivered? There are four options, depending on the type of signal.

Option	Example
To the thread the signal concerns	synchronous signals: `SIGSEGV`, `SIGFPE`
To every thread in the process	some termination cases
To any thread that does not block the signal	POSIX process signal: `SIGINT`
To a special thread that receives all signals	a `sigwait` loop

Table 4.16: Signal delivery options in a multithreaded process.

Synchronous signals arise from an operation that a thread itself performs: an invalid memory access produces `SIGSEGV`, and division by zero produces `SIGFPE`. These signals naturally go to the thread that made the error. Asynchronous signals, on the other hand, come from outside: when the user presses Ctrl+C, the process receives `SIGINT`. POSIX delivers such a process signal to any thread that does not block that signal.

Two distinctions matter here. Signal **handlers** are process-wide: which function runs for a signal is the same for all threads. The signal **mask**, on the other hand, is private to the thread: each thread determines for itself which signals it blocks with `pthread_sigmask`. To send a signal to a specific thread, `pthread_kill(tid, sig)` is used.

Figure 4.32: `SIGINT` (Ctrl+C) arrives at the process. T1 and T2 have blocked the signal with their masks; the signal is delivered only to T3, which does not block it and waits with `sigwait()`.

Good practice in multithreaded programs is the arrangement in the figure. All threads block process signals; a single thread waits for these signals in a loop with `sigwait`. The signal is thus handled as an ordinary function return instead of an asynchronous handler, and the restrictions of the handler (only async-signal-safe functions) are avoided. None of these subtleties is possible with user-level threads: the kernel recognizes only the process and cannot address the right thread. Windows has no concept of signals; a similar task is performed by asynchronous procedure calls (APC).

4.6.3 Thread cancellation

It may be necessary to terminate a thread before it finishes its work. When one of several threads searching in parallel in a database finds the result, the others become unnecessary. When the “stop” button is pressed in a browser, the threads that load the page must stop. The thread to be terminated

is called the **target thread**. Cancellation takes two forms.

Asynchronous cancellation
: The target is terminated immediately. The target may be holding a lock at that moment or may have half-updated a data structure; the lock is never released and the data is left inconsistent. It is therefore dangerous.

Deferred cancellation (the default)
: The target checks for a cancellation request at certain **cancellation points** and, if there is a request, terminates in an orderly way.

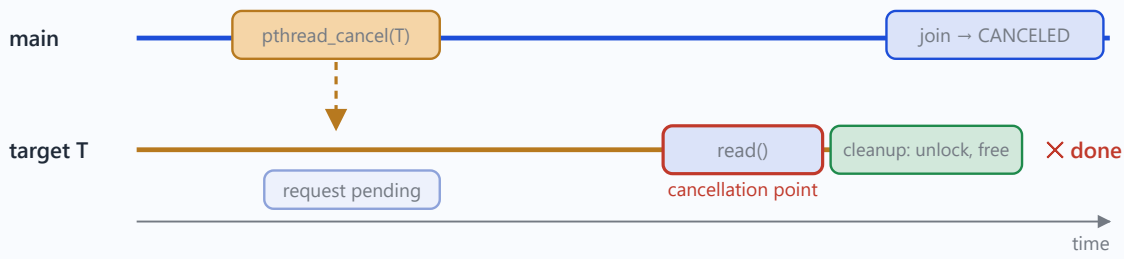


Figure 4.33: Deferred cancellation. `main` calls `pthread_cancel(T)`; the request is held. When T reaches a cancellation point (`read()`), the cleanup handlers run (the lock is released, memory is returned), T terminates, and join returns `PTHREAD_CANCELED`.

In deferred mode, `pthread_cancel(T)` is only a request; T keeps running. When T reaches a cancellation point, the request is processed. Cancellation points are calls specified by POSIX that can generally block: `read`, `sleep`, `pthread_cond_wait`, and the like. If a long computation loop never passes through a cancellation point, the programmer puts an artificial cancellation point in the loop by adding a `pthread_testcancel()` call. When the cancellation is processed, first the cleanup handlers registered with `pthread_cleanup_push` run, in the reverse order of registration: locks are released and allocated memory is returned. Then T terminates, and the thread waiting for it with join receives `PTHREAD_CANCELED` as the return value.

In Java the counterpart of cancellation is the cooperative `interrupt()` mechanism. `t.interrupt()` sets an interrupt flag on the target; the target checks the flag with `Thread.interrupted()` or leaves a blocking call with `InterruptedException`. `Thread.stop()`, the Java counterpart of asynchronous cancellation, has not worked since JDK 20 and throws an exception when called.

4.6.4 Global variable conflicts

Global variables designed for single-threaded programs conflict between threads. The best-known example is UNIX's `errno` variable. When a system call fails, it writes the error code into `errno`; after the call, the program reads this variable to learn the cause of the error.

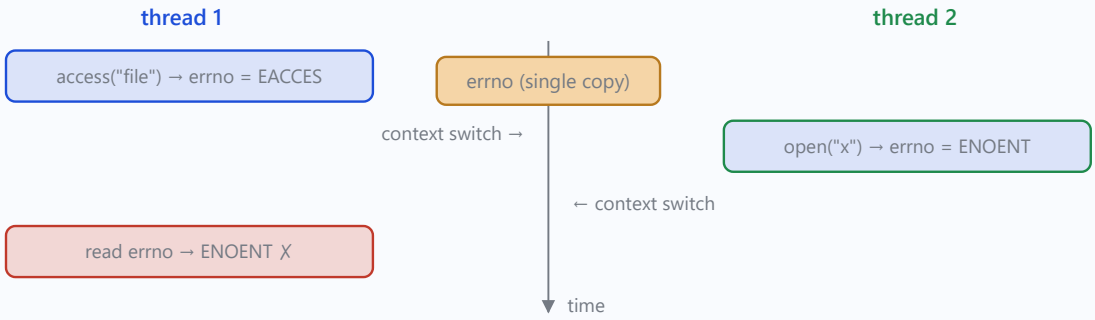


Figure 4.34: The global `errno` conflict. Thread 1 calls `access("file")`, and `errno = EACCES`. The context changes; thread 2 calls `open("x")` and overwrites `errno` with `ENOENT`. When thread 1 runs again, it reads the wrong value (`ENOENT`).

The problem has two solutions: forbid the variable or give each thread its own copy. The first is incompatible with old code; the second was therefore chosen. In today's C libraries, `errno` is a macro and corresponds to a thread-local variable. Each thread sees its own `errno`.

4.6.5 Thread-local storage

Definition 4.5

Thread-local storage (TLS) refers to variables that are accessed by the same name but have a separate copy in each thread. What one thread writes to a TLS variable, the other threads do not see.

The difference between TLS and a local variable is lifetime. A local variable lives only while the function in which it is defined runs, and other functions cannot access it. A TLS variable lives for the lifetime of the thread, across function calls: it is like a global variable, but private to the thread. It is defined with the keyword `_Thread_local` in C11, `thread_local` in C23 and C++, and `__thread` in GCC. Pthreads does the same job with function calls: `pthread_key_create` creates a key, and `pthread_setspecific` and `pthread_getspecific` write and read the value of that key in the calling thread. Typical uses are `errno`, the identifier of an operation, and a per-thread random number generator.

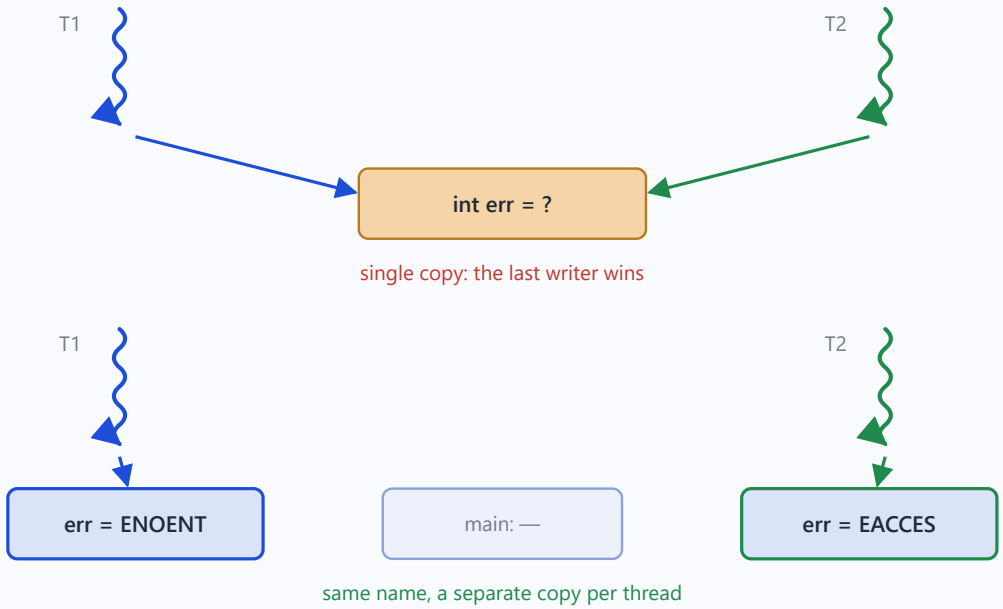


Figure 4.35: (a) Plain global `int err;` : T1 and T2 share a single copy; the last writer wins. (b) `_Thread_local int err;` : the same name, a separate copy per thread; T1's `ENOENT` and T2's `EACCES` do not get mixed up.

In Java, TLS is provided by the `ThreadLocal` class. The program below writes different values to the same `ThreadLocal` variable in two threads and at the end reads `main`'s own copy:

```
public class Local {
  static ThreadLocal<Integer> id =
    ThreadLocal.withInitial(() -> 0);
  public static void main(String[] args) throws Exception {
    Thread a = new Thread(() -> { id.set(1); System.out.println("A: " + id.get()); });
    Thread b = new Thread(() -> { id.set(2); System.out.println("B: " + id.get()); });
    a.start(); a.join();
    b.start(); b.join();
    System.out.println("main: " + id.get());
  }
}
```

`ThreadLocal.withInitial(() -> 0)` makes each thread's copy start at 0 when it is first read. Thread A writes 1 to its own copy and thread B writes 2 to its own copy. When run with JDK 21, the output is `A: 1` , `B: 2` , `main: 0` . Because `main` writes nothing, it sees the initial value. Because A and B are run here in turn with join, the order of the lines is also the same every time; if the two threads were started together, the order of the A and B lines could vary, but the values would not.

4.6.6 Reentrant and thread-safe code

Some library routines keep their internal state in static variables. Such a routine is called **non-reentrant**. If, while one thread puts a message into the library's shared buffer, a context switch occurs and another thread writes to the same buffer, the first message is lost. `malloc` was historically like this too: while updating the free-memory list, it left the list briefly inconsistent, and an intervening thread could receive an invalid pointer. Today's `malloc` implementations use internal locks and per-thread memory areas (arenas).

Definition 4.6

Thread-safe code can be called from several threads at the same time; it protects shared state with a lock or keeps no state at all. **Reentrant** code uses no shared state at all; all the state it needs is in the parameters given by the caller and in local variables.

Many functions of the classic C library have an unsafe version and a safe version added later. The safe versions usually end with the suffix `_r` (reentrant) and keep their state not in a static variable but in a pointer given by the caller.

Unsafe	Safe version
`strtok`	`strtok_r`
`localtime`	`localtime_r`
`rand`	`rand_r` or a local generator
`asctime`, `ctime`	`strftime`
Java `SimpleDateFormat`	`DateTimeFormatter`

Table 4.17: Unsafe functions and their safe counterparts.

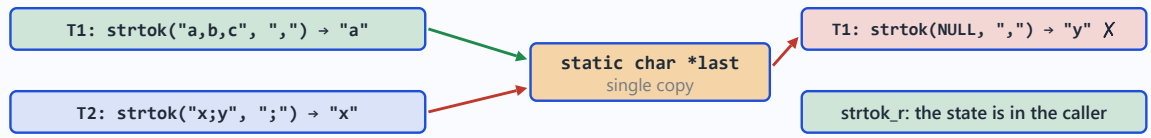


Figure 4.36: Threads getting mixed up because of `strtok`'s static pointer. T1 starts splitting `"a,b,c"` and gets `"a"`. T2 calls it with `"x;y"` and points the single-copy `static char *last` at its own string. T1's next call `strtok(NULL, ",")` returns `"y"` instead of `"b"`. In `strtok_r` this state is kept in the caller.

`strtok` splits a string into pieces according to delimiters. The string is given on the first call; on later calls `NULL` is given and the function remembers where it left off from the static `last` pointer. In the scenario in the figure, T2's call moves this pointer to its own string; T1 now gets not the rest of its own string but the `"y"` left over from T2's string. `strtok_r` keeps the same pointer in a `char **` parameter given by the caller; because each thread has its own pointer, there is no mix-up.

Common mistake

Treating the concepts of thread-safe and reentrant as synonyms. A function protected by a lock is thread-safe but not reentrant: if, while the same thread is inside the function, a signal handler calls the same function again, it may wait forever, because the thread itself already holds the lock. A reentrant function, which uses no shared state, is safe in both cases.

4.6.7 Race condition: `counter++`

The simplest example of why access to shared data requires synchronization is incrementing a counter. In C or Java, `counter++` is a single line, but on the machine it consists of three steps:

```
LOAD  R, counter
ADD   R, 1
STORE counter, R
```

The value is read from memory into a register (`LOAD`), incremented in the register (`ADD`), and written back to memory (`STORE`). If two threads execute these three steps interleaved, an increment can be lost.

Definition 4.7

A **race condition** is an erroneous situation in which several threads access shared data and the result depends on the timing of the accesses.

Let `counter = 5` initially; T1 and T2 each increment the counter once. The expected result is 7. Table 4.18 traces an ordering in which an increment is lost.

Step	T1	T2	R1	R2	`counter`
1	`LOAD R1 ← counter`		5	—	5
2		`LOAD R2 ← counter`	5	5	5
3		`ADD R2, 1` ; `STORE`	5	6	6
4	`ADD R1, 1` ; `STORE`		6	6	**6**

Table 4.18: Lost update. T1 is on core 0 and T2 on core 1.

T1 reads the value into its register: R1 = 5. T2 steps in (through a context switch on a single core, or by running at the same time on multiple cores) and also reads 5. T2 increments and writes: `counter` = 6. T1 continues from its old value and writes 5 + 1 = 6. T2's increment is lost; this bug is called a

lost update. The four steps of the two threads can interleave in $C(4, 2) = 6$ different orders. When all the orderings are tried with Python, the result is 7 in two of them and 6 in four: when a thread gets in between the other's read and write, the increment is lost.

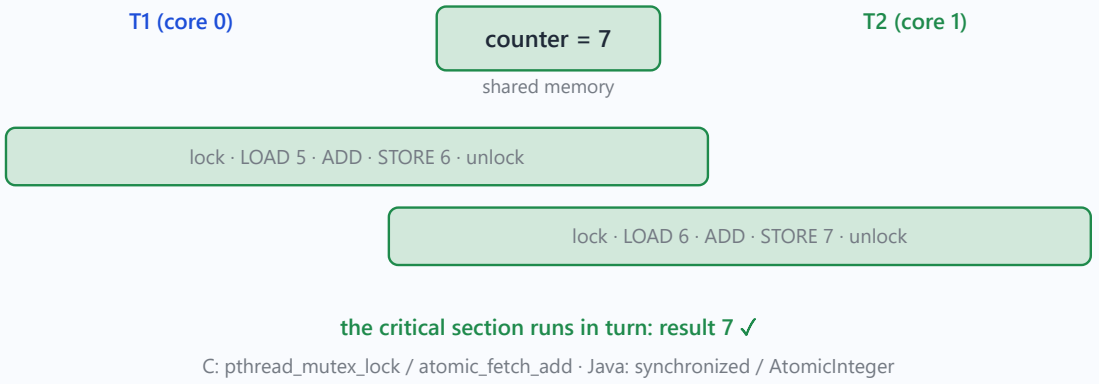


Figure 4.37: The solution: the read-increment-write sequence is protected with a lock. T1 takes the lock, reads 5, writes 6, and releases the lock; then T2 takes the lock, reads 6, and writes 7. The critical section runs in turn and the result is 7. In C, `pthread_mutex_lock` or `atomic_fetch_add`, and in Java, `synchronized` or `AtomicInteger` is used.

The solution is to make the read-increment-write sequence **atomic**: either the sequence is protected with a lock so that only one thread executes it at a time, or the hardware's atomic operations that do read-modify-write in a single instruction are used. Locks, atomic instructions, and the critical-section problem are the subject of synchronization; what matters here is to see that the problem arises from the shared address space of threads.

Example 4.7

The shared `counter` is 0. Three threads increment the counter once each; each increment has two steps: read (`r = counter`) and write (`counter = r + 1`). What are the possible final values?

1. **Largest value:** If the threads run one after another, each reads what the previous one wrote: 0 → 1 → 2 → 3.
2. **Smallest value:** If all three read before any writes, all three read 0 and all three write 1. The result is 1.
3. **Intermediate value:** If two read the same value and the third reads after one of them has written, the result is 2.
4. **Counting.** There are $6! / (2! \cdot 2! \cdot 2!) = 90$ orderings in which the six steps can interleave. When all are tried with Python, 6 of them give 3, 36 give 2, and 48 give 1. The correct result, 3, is obtained in only 6.7% of the orderings.

4.6.8 Observing the race

The lost update is not a theoretical possibility; it can be observed with a simple Java program. In the program below, two threads each increment the shared `counter` one million times.

```
public class Race {
  static int counter = 0;
  public static void main(String[] a) throws Exception {
    Runnable job = () -> {
      for (int i = 0; i < 1_000_000; i++) counter++;
    };
    Thread t1 = new Thread(job), t2 = new Thread(job);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println("counter = " + counter);
  }
}
```

The expected result is 2,000,000. When the program was run three times with JDK 21, the following results were obtained:

```
counter = 1456465
counter = 1544210
counter = 1910165
```

When the same program was run five more times on another machine with JDK 21.0.3, the results were 1,024,058, 1,047,114, 1,929,112, 1,055,456, and 1,079,458. **These numbers change on every run**; even on the same machine, two runs do not give the same result. The only thing that does not change is that the result is smaller than 2,000,000 every time: some of the increments are lost.

The result is not deterministic: it depends on scheduling, the number of cores, and how the just-in-time (JIT) compiler compiles the code. The JIT can compile the loop so that the counter is kept in a register and the writes become sparser; this changes the number of lost increments. The output of such a program therefore cannot be traced step by step; only the range of possible values can be stated.

There are three ways to fix it. The increment can be put inside a `synchronized` block, `AtomicInteger.incrementAndGet()` can be used, or, for very heavy updates, `LongAdder` can be chosen. When the first two were tried with JDK 21, with two threads each doing one million increments, the output was every time:

```
synchronized: 2000000  atomic: 2000000
```

The best solution is not to share at all: each thread counts in a local variable, and the results are combined at the end. This is the idea of fork-join and of the `partial[k]` example; the lock cost disappears too.

4.6.9 Threads in Linux: `clone()`

The Linux kernel makes no distinction between a "process" and a "thread." Both are a **task** and are represented by the same `task_struct` structure. A new task is created with the `clone()` system call; the flags given to `clone()` determine what the new task shares with its parent. If no sharing flag is given, the result is like `fork`: a copy of everything is made.

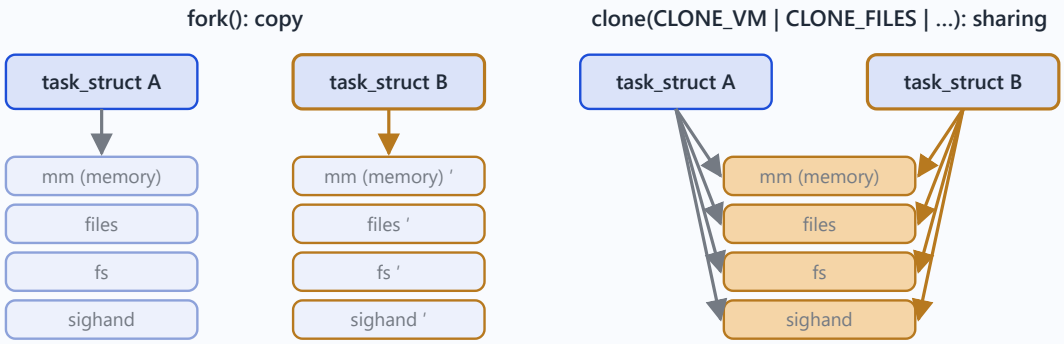


Figure 4.38: On the left, `fork()`: task B gets copies of the memory (`mm`), `files`, `fs`, and `sighand` structures. On the right, `clone(CLONE_VM | CLONE_FILES | ...)`: A and B point to the same structures.

`CLONE_VM` shares the address space, `CLONE_FS` the file-system information such as the working directory and root directory, `CLONE_FILES` the open file table, and `CLONE_SIGHAND` the signal handlers. `pthread_create` corresponds roughly to the call `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | ...)`. `CLONE_THREAD` puts the new task in the same **thread group** as its parent. The group's identifier (TGID, thread group ID) appears to the user as the process ID: `getpid()` returns the same value for everyone in the group. `gettid()` returns each task's own identifier, that is, the thread ID. This flexibility lets Linux support processes, threads, and hybrid cases between the two (e.g., tasks that share memory but have separate file tables) with a single mechanism.

4.7 Process or thread?

The advantages of threads are summarized as follows:

- **Responsiveness:** when one blocks, another handles user I/O.
- **Resource sharing:** memory, open files, and sockets are shared by default.
- **Speed:** creation and switching are cheaper than for processes.
- **Hardware parallelism:** a single process uses many cores.

- **Easy programming model:** parallelism is achieved even with blocking calls (the web server); each flow is written as sequential code.

The disadvantages are the other side of the same sharing:

- **Synchronization:** access to shared data must be controlled; there is a risk of races and deadlock.
- **Dependence:** threads in the same process are not independent; if one crashes, the whole process crashes.
- **No memory protection:** one thread can overwrite another's stack.
- **Debugging and testing are hard:** bugs depend on timing and do not appear on every run.

The choice is made according to the balance between isolation and sharing. If isolation is needed, for example if browser tabs must not be affected by each other's crashes or a security boundary must be drawn, a process is chosen. If tight data sharing and low cost are needed, a thread is chosen. Chrome uses both together: each tab and extension runs in a separate process, and inside each process there are several threads for drawing, networking, and script execution.

4.8 Summary

- The **process** owns resources; the **thread** is a flow of execution inside a process and the basic unit of scheduling. The code, global data, heap, open files, and signal handlers are shared; the PC, registers, stack, state, signal mask, and TLS are private to each thread. The stack is private but not protected.
- Every thread has its own stack; on Linux the default size is usually 8 MiB, and the guard pages between stacks catch overflow with `SIGSEGV`.
- In a multithreaded **web server**, the dispatcher gives requests to workers; a worker that blocks on disk stops only itself. The gain comes, even on a single processor, from the overlap of waiting and computation. The event-driven design achieves the same overlap with a single thread but requires saving state by hand.
- Benefits: responsiveness, resource sharing, economy, scalability. In a switch between threads the address space does not change, and the TLB and cache stay warm.
- **Concurrency** is tasks progressing in the same time interval; **parallelism** is their execution at the same instant and requires at least two cores. Parallelism requires concurrency; the converse is not true.
- **Amdahl's law:** speedup $\le 1 / (S + (1 - S) / N)$; upper bound $1 / S$. For $S = 0.1$ the speedup on 16 cores is 6.4× and the limit is 10×. **Gustafson's law** gives the scaled speedup, when the problem grows, as $S + (1 - S) \cdot N$.
- With **user-level** threads, switching is very cheap, but a blocking call and a page fault stop the whole process, the clock interrupt does not reach the library, and a single process cannot use several cores. **Kernel-level** threads block independently and run in parallel, but each uses kernel resources.

- **N:1** is cheap but has no parallelism; **1:1** (Linux, Windows, macOS) provides true parallelism; **M:N** was abandoned in operating systems and has returned in language run-times (Go, Erlang, Java 21 virtual threads).
- **Pthreads:** `pthread_create`, `pthread_join`, `pthread_exit`, `pthread_detach`. Arguments and results are passed as `void *`; the address given to a thread must live long enough, and the result must be returned on the heap. **Java:** `start()` starts a new thread, `run()` does not; `join` both waits and provides visibility.
- **Implicit threading:** pools pay the creation cost once and bound the number of threads (for I/O-heavy work, $N_{cores} \times (1 + W / C)$); fork-join splits the task and balances the load with **work stealing**; OpenMP works with compiler directives, GCD with blocks and queues, and oneTBB with templates.
- In POSIX, `fork()` copies only the calling thread. Signal handlers are process-wide and the signal mask is private to the thread. With **deferred cancellation**, a thread terminates safely at cancellation points. Variables such as `errno` are made thread-private with **TLS**.
- `counter++` is the three steps read-increment-write; if unprotected, increments are lost and the result changes on every run. The solution is a lock, an atomic operation, or a design without sharing. On Linux, processes and threads are created with the same `clone()` call, with different sharing flags.

4.9 Key terms

thread multithreaded process context switch concurrency parallelism false sharing

Amdahl's law user-level / kernel-level thread thread pool work stealing signal

thread-local storage reentrant / thread-safe race condition atomic instruction

---

4.10 Test Yourself

1. Which of the following items are shared and which are private to each of two threads in the same process: the stack pointer, the open file table, a `static` variable, a `_Thread_local` variable, an array allocated on the heap with `malloc`?

   Answer

   Shared: the open file table, the `static` variable, and the array on the heap; all three belong to the process's address space or resources. Private: the stack pointer (each thread has its own stack) and the `_Thread_local` variable (each thread has a separate copy).

2. 70% of a program can be parallelized. What are the speedup on four cores and the theoretical

upper bound?

Answer

S = 0.3. On four cores the time is 0.3 + 0.7 / 4 = 0.3 + 0.175 = 0.475; the speedup is 1 / 0.475 ≈ 2.11×. The upper bound is 1 / 0.3 ≈ 3.33×.

3. Can a multithreaded program running on a single-core machine have a race condition? Why?

Answer

Yes. Even without parallelism there is concurrency. A thread's time slice can end between `LOAD` and `STORE`; the scheduler runs the other thread, which reads the same old value. A lost update also occurs on a single core; it is just seen less often.

4. One of the threads of a program running in the N:1 model accesses a page that is not in memory. What happens? How does the same situation turn out in the 1:1 model?

Answer

In the N:1 model the page fault goes to the kernel, and while the kernel brings the page from disk it holds the process's single kernel thread; the library cannot switch to another thread and the whole process stops. In the 1:1 model only the thread that caused the fault waits; the others keep running.

5. The array `a = {4, 1, 3, 5, 2, 6}` is summed with the fork-join `SumTask` task with `LIMIT = 2`. Find the task tree and the result of each node.

Answer

The root [0,6) splits: mid = 3. The left [0,3) has three elements and is larger than the limit; with mid = 1 it splits into [0,1) = 4 and [1,3) = 1 + 3 = 4, total 8. The right [3,6) splits too: [3,4) = 5 and [4,6) = 2 + 6 = 8, total 13. The root is 8 + 13 = 21. Seven tasks are created; four of them are leaves. (Run with a single-worker `ForkJoinPool` on JDK 21: 21.)

6. On a four-core machine, each task computes for 10 ms and waits 30 ms for a database response. What does the rough rule give for the pool size? What would happen if the tasks only computed?

Answer

$N_{threads} \approx 4 \times (1 + 30 / 10) = 16$. For tasks that only compute, the size should equal the number of cores, that is, 4; anything more only adds context switching and cache contention.

7. In a process with four threads, one thread calls `fork()` and the child immediately calls

`exec("/bin/ls")`. How many threads does the child have before and after `exec`?

Answer

One before `exec`: in POSIX only the thread that calls `fork` is copied. One after `exec` as well: the new program replaces the whole address space and starts with a single thread.

8. Three threads each increment a counter that starts at 0 once, with an unprotected `counter++`. What are the possible final values?

Answer

1, 2, or 3. If all three read 0 before any writes, the result is 1; if two read the same value, 2; if there is no overlap at all, 3. Of the 90 possible orderings, 48 give 1, 36 give 2, and only 6 give 3.

9. A Java program, after the line `Thread t = new Thread(r);`, calls `t.run();` and never calls `start()`. How many threads run, and what does `t.getState()` return?

Answer

Only the `main` thread runs; the code of `r` executes inside `main` like an ordinary method. Because `t` was never started, `getState()` returns `NEW`.

4.11 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters “Concurrency: An Introduction” and “Interlude: Thread API”: they present race conditions and the Pthreads API with small, runnable C examples; free at ostep.org.
- A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, Section 2.2: treats in detail the word processor, the web server, user-level and kernel-level threads, pop-up threads, and the problems of making single-threaded code multithreaded, which are the source of the examples in these notes.
- OpenJDK, *JEP 444: Virtual Threads*: the official document explains why virtual threads were designed, the mounting and unmounting mechanism, and in which cases they bring no gain.
- M. D. Hill and M. R. Marty, “Amdahl’s Law in the Multicore Era,” *IEEE Computer*, 41(7), 2008: extends Amdahl’s law to different core designs and shows how the serial part affects the design of multicore chips.

4.12 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th edition, Wiley, 2018 — Chapter 4: Threads & Concurrency.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th edition (2014) / 5th edition (2023), Pearson — Section 2.2: Threads.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Concurrency: An Introduction", "Thread API".
4. POSIX.1-2017 (IEEE Std 1003.1); the Linux manual pages `pthreads(7)` and `clone(2)`.
5. JEP 444: Virtual Threads (Java 21); the OpenMP 5.2 specification.
6. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *AFIPS Spring Joint Computer Conference*, 1967, pp. 483–485.
7. J. L. Gustafson, "Reevaluating Amdahl's Law", *Communications of the ACM*, 31(5), 1988, pp. 532–533.
8. M. D. Hill, M. R. Marty, "Amdahl's Law in the Multicore Era", *IEEE Computer*, 41(7), 2008, pp. 33–38.

CPU Scheduling

Memory holds dozens of processes that are ready to run, but a core can run only one at a time. Each time the CPU becomes free, the operating system's scheduler decides which process runs next. That decision determines how long a user waits after pressing a key, how many jobs a server finishes per hour, and whether a brake controller misses its deadline. These notes begin with the moments at which the decision is made and the criteria by which a good decision is judged. FCFS, SJF, SRTF, priority, and round-robin scheduling are then run step by step on the same set of four processes, so their results can be compared directly. Multilevel queues, proportional-share methods such as lottery and stride scheduling, load balancing on multiprocessors, and RM and EDF for real-time systems follow. The last sections cover the Linux CFS and EEVDF schedulers, Windows priority levels, and methods for evaluating algorithms. Every Gantt chart and average in the chapter was checked by actually running the algorithms.

LEARNING OUTCOMES

Concepts
Explain the CPU–I/O burst cycle, the four situations in which a scheduling decision is made, preemptive and nonpreemptive scheduling, and the job of the dispatcher.

Criteria
Compute waiting, turnaround, and response times from a Gantt chart; weigh throughput, CPU utilization, and fairness goals against one another.

Algorithms
Apply FCFS, SJF, SRTF, priority, round-robin, MLQ, and MLFQ scheduling step by step to the same set of processes and compare the results.

Prediction and fairness
Predict the next CPU burst with an exponential average; prevent starvation with aging, lottery, stride, and fair-share scheduling.

Multiprocessor and real time
Explain load balancing and processor affinity; draw RM and EDF schedules and test schedulability with the utilization bound.

Real systems
Interpret Linux CFS and its successor EEVDF, Windows priority classes, and dynamic priority boosts.

5.1 Basic concepts

An old single-task system has no scheduling problem: there is one program in memory and the CPU runs it. When the program waits for data from the disk, the CPU waits too. Multiprogramming systems fill this gap by keeping many processes in memory at once; when one process starts waiting, the CPU is given to another. The idea is simple, but it immediately raises a question: which of the waiting ready processes should be chosen? That question is the subject of this chapter.

5.1.1 What is scheduling?

Definition 5.1

The **short-term scheduler** (CPU scheduler) is the kernel component that, when the CPU becomes free, selects which of the processes in the ready queue will run. The **ready queue** holds the processes that are ready to run and are waiting only for a CPU. Processes waiting for I/O or some other event sit in **wait queues**.

The scheduler has two goals. The first is to keep the CPU busy: it should never sit idle while a ready process exists. The second is to improve the system's objective. The objective depends on the environment: fast response on a desktop, high throughput on a server, fairness on a shared machine, meeting deadlines on an embedded controller. The same algorithm can be good for one objective and bad for another; this trade-off will come up again and again throughout the chapter.

During its lifetime a process moves among queues (Figure 5.1). A new process enters the ready queue. When the scheduler picks it, it runs on the CPU. If its time slice expires, a clock interrupt stops it and it returns to the ready queue. If it requests I/O, it moves to a wait queue, and when the I/O completes it becomes ready again. When its work is done it leaves the system. Each core runs at most one process at a time; every process other than the running ones sits in some queue.

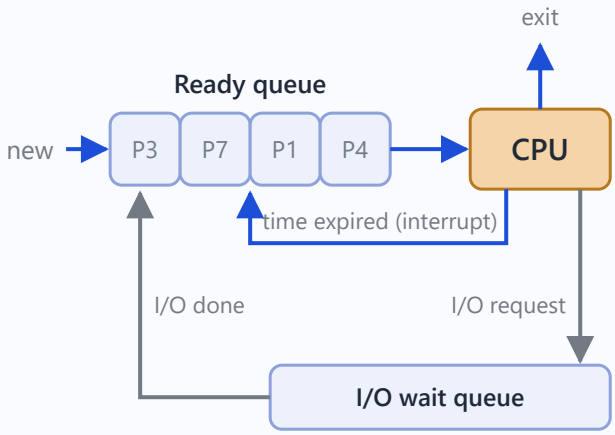


Figure 5.1: The queueing diagram. A new process enters the ready queue and is selected for the CPU. When its time expires it returns to the ready queue; when it requests I/O it moves to the I/O wait queue; when the I/O completes it becomes ready again.

Despite its name, the ready queue need not be FIFO. Depending on the algorithm it can be a priority queue, a balanced tree, or an unordered linked list. The Linux fair scheduler, for example, uses a red-black tree. The entries in the queue are not the processes themselves but their process control blocks (PCBs).

One more detail: modern kernels that support threads actually schedule kernel threads, not processes. In a multithreaded process each thread is a separate entry in the ready queue. Textbooks still use the phrase "process scheduling," and in these notes the word process means "the unit being scheduled."

5.1.2 The CPU–I/O burst cycle

The success of scheduling rests on an observed property of processes: execution alternates between CPU work and I/O waiting. A process computes for a while, then reads a file and waits, then computes again.

Definition 5.2

A **CPU burst** is the time a process uses the CPU continuously between two I/O requests. An **I/O burst** is the time the process waits for an I/O operation to finish. A process begins with a CPU burst and ends with a final CPU burst that issues the termination call (Figure 5.2).

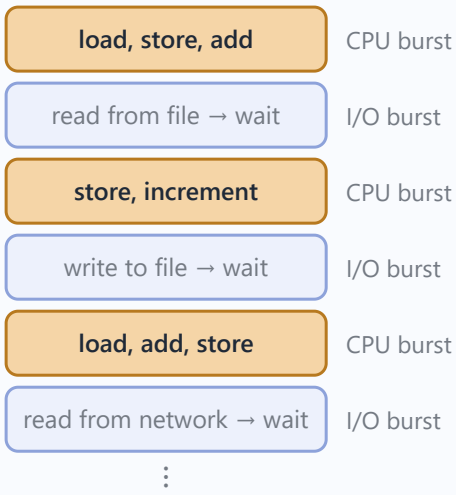


Figure 5.2: The alternating sequence of CPU and I/O bursts. A CPU burst made up of instructions such as `load`, `store`, and `add` is followed by a file read and a wait; then a new CPU burst begins.

Measured CPU burst durations on real systems always have the same shape (Figure 5.3). There are many short bursts and few long ones. The graph is exponential or similar, a curve with a long tail to the right. This observation explains why algorithms that favor processes with short bursts work well.

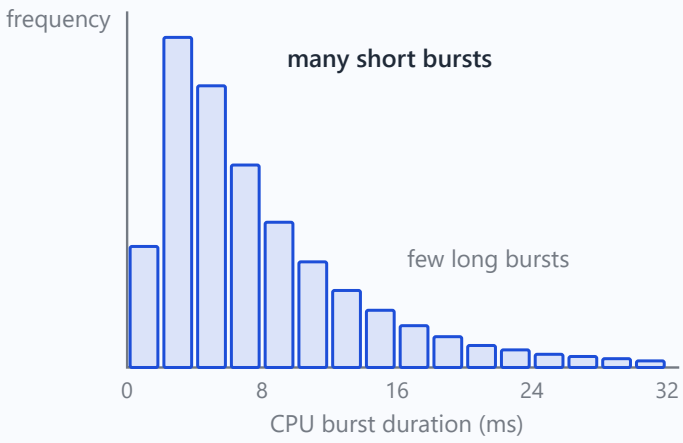


Figure 5.3: Histogram of CPU burst durations. Short bursts are very frequent; long bursts are rare.

Processes fall into two groups by burst structure (Figure 5.4). A **CPU-bound** process has long CPU bursts and infrequent I/O; video encoding, compilation, and scientific computing are examples. An **I/O-bound** process has short CPU bursts followed by long I/O waits; a text editor, a shell, or a database client is typical. A process can move from one group to the other over time: a program that first reads files and then does a long computation lives through both phases.

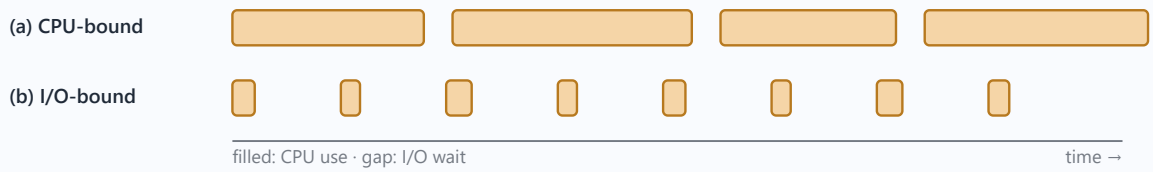


Figure 5.4: (a) A CPU-bound process has long CPU bursts. (b) An I/O-bound process waits a long time after short bursts. Filled boxes show CPU use; the gaps between them show I/O waiting.

What matters for scheduling is not how long the I/O wait is but how short the CPU burst is. If an I/O-bound process is run as soon as it becomes ready, it issues a new I/O request within a few milliseconds, which keeps the disk or network card busy too. If the process is kept waiting behind a long CPU job, the device sits idle. Because disk technology has improved much more slowly than CPUs, most processes have become I/O-bound over time. Not keeping I/O-bound processes waiting is therefore a common design principle of modern schedulers.

5.1.3 Overlapping with I/O: treating bursts as separate jobs

Treating each CPU burst of a process as a separate short job gives the scheduler a chance to overlap I/O with computation. This idea lies behind most of the algorithms in the chapter, and it is best understood through an example.

Example 5.1

Processes A and B each need 50 ms of CPU time. A runs 10 ms and then waits 10 ms for the disk; this cycle repeats five times. B does no I/O. Let us compare two schedules.

1. **Schedule without overlap.** A's 50 ms of work counts as a single job, and B starts only when A ends. With five CPU bursts and four I/O waits, A finishes at 5 × 10 + 4 × 10 = 90. While A is at the disk the CPU is idle, because the next job is still A. B runs from 90 to 140. The average turnaround time is (90 + 140) / 2 = 115 ms, and CPU utilization is 100 / 140 ≈ 71%.
2. **Schedule with overlap.** Each 10 ms burst of A counts as a separate short job. When A goes to the disk, B runs. When A's I/O completes, the new A job, with 10 ms remaining, preempts B, whose remaining time is longer. This is the behavior of the rule that picks the shortest remaining time (STCF or SRTF). The schedule goes A 0–10, B 10–20, A 20–30, and so on; A still finishes at 90, but B finishes at 100. The average turnaround time is (90 + 100) / 2 = 95 ms, and CPU utilization is 100 / 100 = 100%.
3. **Comment.** A's finishing time did not change, but B finished 40 ms earlier and the CPU was never idle. The gain came from overlapping A's disk waits with B's computation (Figure 5.5).

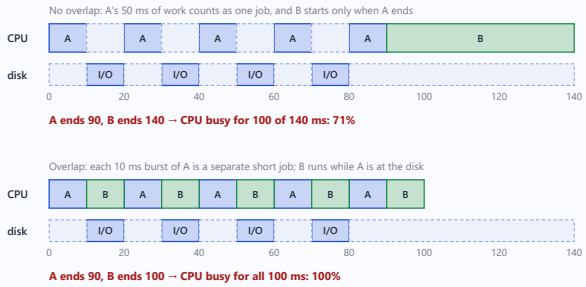


Figure 5.5: Top: the schedule without overlap; the CPU is idle during A's I/O waits and B finishes at 140. Bottom: the schedule with overlap; B runs while A is at the disk, and both processes finish within 100 ms.

A real system does not know burst lengths in advance, so the rule "put the short burst first" can be applied only indirectly. The multilevel feedback queue does exactly this: it observes the behavior of processes and keeps those with short bursts in the upper queues.

5.1.4 When are scheduling decisions made?

The scheduler is not a loop that runs continuously; it is invoked at particular events. In the process state diagram, four transitions may require a decision (Figure 5.6):

1. **Running → waiting.** The process requests I/O or waits for a child with `wait()`. The CPU is free, and a new process must be chosen.
2. **Running → ready.** A clock interrupt arrives and the process's time slice has expired. Stopping the running process and choosing another is optional.
3. **Waiting → ready.** An I/O operation completes and the waiting process wakes up. If the woken process is more important than the one currently running, it may preempt it.
4. **Termination.** The process ends with `exit()`. The CPU is free, and a choice is again required.

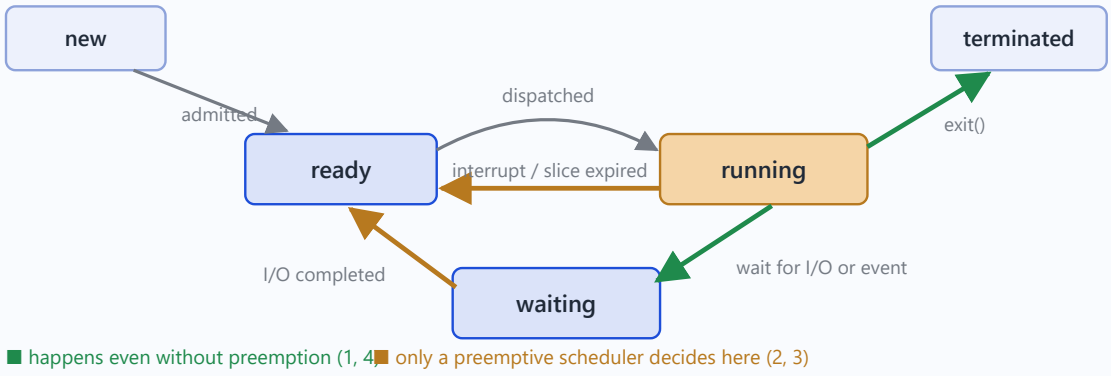


Figure 5.6: The four transitions in the process state diagram at which a scheduling decision is made. At the green transitions (1 and 4) every scheduler decides; at the orange transitions (2 and 3) only a preemptive scheduler decides.

In situations 1 and 4 there is no other option: the CPU is idle and must be given to someone. In situations 2 and 3 the CPU is still running a process, and the scheduler may choose to let it keep running. Schedulers are divided into two kinds according to whether they decide in these two situations.

5.1.5 Preemptive and nonpreemptive scheduling

Definition 5.3

In **nonpreemptive** (or cooperative) scheduling, decisions are made only in situations 1 and 4. The chosen process runs until it blocks, terminates, or voluntarily gives up the CPU (`yield`). In **preemptive** scheduling, decisions are also made in situations 2 and 3. A process can be suspended when its time slice expires or when a more important process becomes ready.

Nonpreemptive scheduling needs no clock interrupt and involves few context switches. Its price is that the system depends on the good will of every program. Windows 3.x and classic Mac OS (9 and earlier) used cooperative multitasking. A single program that entered an infinite loop never released the CPU and could lock up the whole system. Preemptive scheduling prevents this with the clock interrupt: the kernel takes the CPU back at regular intervals. Today all general-purpose operating systems are preemptive (Figure 5.7).

Figure 5.7: The same two processes. Nonpreemptively, A runs until it requests I/O and B waits for A to give up the CPU voluntarily. Preemptively, the clock interrupt cuts A off at the end of every q = 4 units, and A and B alternate.

Preemption has a price too. If two processes share data, one can be interrupted while updating it, and the other may read the half-finished, inconsistent state. This risk of a **race condition** makes synchronization tools mandatory in preemptive systems. Moreover, every preemption means a context switch, which also consumes CPU time.

The same question arises for the kernel's own code. Can a process be preempted while running kernel code in the middle of a system call? Old UNIX kernels waited for the call to finish. Since Linux 2.6, kernel code can be preempted too; preemption happens only if the running code holds no lock. The kernel tracks this with the `preempt_count` counter it keeps for every task: if the counter is zero, preempting is safe. The PREEMPT_RT patches, developed for real-time use, entered the mainline kernel in Linux 6.12 in 2024 after about twenty years of work. In this configuration interrupt handlers also run as threads and can be preempted; worst-case latency drops to the microsecond range.

Which approach to choose depends on the environment. For batch jobs that run overnight, nonpreemptive or long-slice preemptive algorithms suffice, since nobody is waiting at a screen. In interactive systems and on servers preemption is essential: one process monopolizing the CPU can keep other users' requests waiting for minutes.

Common mistake

Believing that "a preemptive scheduler does not decide at an I/O request." A preemptive scheduler decides in all four situations; it differs from a nonpreemptive one in that it also decides in situations 2 and 3. The completion of a disk read (waiting → ready) can stop the running process only under a preemptive scheduler. Under a nonpreemptive scheduler the woken process joins the ready queue and waits its turn.

5.1.6 The dispatcher and the context switch

Choosing and running the chosen process are two separate jobs. The scheduler makes the choice; that is a policy. Placing the chosen process on the CPU is a mechanism, and the dispatcher takes care of it.

Definition 5.4

The **dispatcher** is the kernel module that gives the CPU to the process the scheduler selected. It does three things: it switches context, it switches to user mode, and it jumps to the right address in the program. The time the dispatcher takes to stop one process and start another is called **dispatch latency** (Figure 5.8).

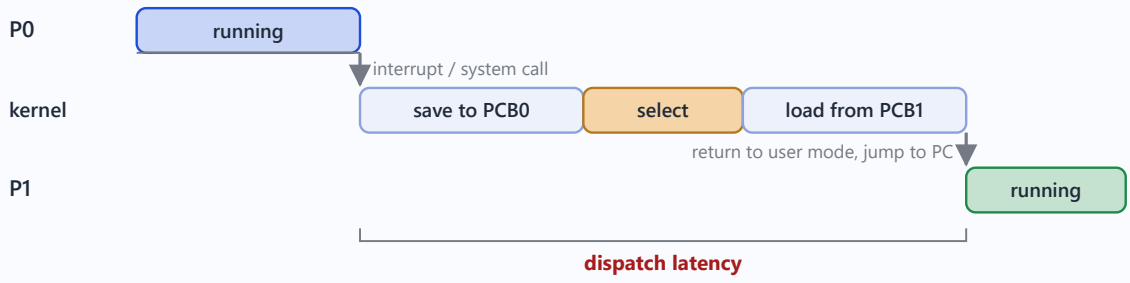


Figure 5.8: The dispatcher's job. The kernel is entered through an interrupt or a system call. The dispatcher saves P0's state into PCB0 and the scheduler selects P1. The dispatcher loads P1's state from PCB1, returns to user mode, and jumps to where P1 left off. The time in between is the dispatch latency.

The dispatcher runs at every context switch, so it must be very fast. On today's processors the direct cost is a few microseconds. The indirect cost can be larger: the new process's data is not in the cache, and its first accesses are slowed by misses. Table 5.1 summarizes what changes in a switch between processes.

Component	On a process switch
Registers, PC, SP	saved to the old process's PCB, loaded from the new one's
Memory map	the page-table base changes (the CR3 register on x86)
TLB	not flushed if the processor supports PCID/ASID tags
Caches	not flushed, but cold for the new process

Table 5.1: The components of hardware state in a context switch between processes.

On Linux, context switches can be measured directly. The `vmstat 1` command prints one line per second; its `cs` column is the total number of context switches in that second. For a single process, the file `/proc/<pid>/status` keeps two counters. `voluntary_ctxt_switches` counts the switches in which the process gave up the CPU itself by blocking. `nonvoluntary_ctxt_switches` counts the switches in which the scheduler forcibly preempted the process. For an I/O-bound process the first counter grows; for a CPU-bound process, the second.

5.1.7 Goals by environment

There is no single "best" scheduler, because systems want different things. Tanenbaum groups the goals by environment (Table 5.2). Some goals apply to every system. **Fairness** means that similar processes get similar shares of the CPU. **Policy enforcement** means obeying the rule the system sets; for example, security-audit processes run first. **Balance** means keeping the CPU and the I/O devices busy together.

Environment	Main goals	Suitable approach
All systems	Fairness, policy enforcement, balance	—
Batch	Throughput (jobs/hour), short turnaround time, high CPU utilization	Nonpreemptive or long-slice preemptive
Interactive	Short response time, proportionality (delays that match the user's expectations)	Preemptive, short time slice
Real-time	Meeting deadlines (no data loss), predictability (no drop in audio or video quality)	Priority- or deadline-based; tasks run briefly and block

Table 5.2: Scheduling goals by environment and suitable approaches.

Proportionality is an interesting goal. A user knows that compressing a large file will take a long time and is willing to wait. But the same user will not tolerate closing a window taking several seconds. A good interactive scheduler keeps delay proportional to the perceived size of the task.

Batch jobs are not a relic of the past. Banks' overnight payroll and account-closing runs, scientific computations, training of AI models, and continuous-integration (CI) builds are still batch jobs today. In high-performance computing centers they are queued by job managers such as Slurm. Goals often

conflict: switching context often to shorten response time lowers throughput because of the cost of each switch.

5.1.8 Scheduling criteria

To compare algorithms, the goals have to be turned into measurable quantities. Three moments in a process's life are enough to define all of these criteria: its **arrival** time A in the system, the time F at which it **first runs** on the CPU, and its **completion** time C (Figure 5.9).

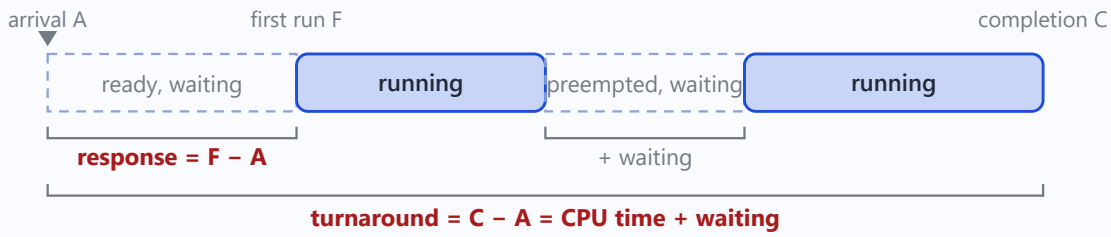


Figure 5.9: The arrival (A), first-run (F), and completion (C) times on a process's timeline. Response time is F − A and turnaround time is C − A; turnaround time is the CPU time plus the waiting times spent in the ready queue.

Definition 5.5

CPU utilization is the fraction of time the CPU is busy. **Throughput** is the number of processes completed per unit time. **Turnaround time** is the time from a process's submission to its completion: C − A. **Waiting time** is the total time a process spends in the ready queue; if the process does no I/O, it equals turnaround time minus CPU time. **Response time** is the time from submission to the first time the process gets the CPU: F − A.

The first two criteria should be large and the last three small. In real systems CPU utilization is around 40% on a lightly loaded machine and around 90% on a busy server. None of the criteria is sufficient on its own; the differences between algorithms show up most clearly in the average waiting, average turnaround, and average response times.

One of these criteria is more fundamental than the others. A process's CPU time and I/O time belong to the program itself; they do not change whichever algorithm is chosen. The only thing the scheduler can influence is the time a process waits in the ready queue. Since turnaround time = CPU time + I/O time + waiting time, the difference in turnaround between two algorithms is exactly the difference in waiting.

Response time and waiting time are easy to confuse. Response measures only the first wait: when does the user see the first reaction on screen after typing a command? Waiting time is the sum of all the intervals the process spends in the queue during its life. Under a preemptive algorithm a process

may be preempted and returned to the queue many times; its response time can stay small while its waiting time grows large.

In interactive systems the **variance** of response time matters more than the average. If most commands are answered in 0.1 seconds and one now and then in 5 seconds, users find that more annoying than all of them being answered in 0.5 seconds. Finally, **weighted** (normalized) **turnaround time** is the ratio of turnaround time to CPU time. Its ideal value is 1, and it shows how heavily short jobs are penalized.

Example 5.2

Suppose P1 (arrival 0, CPU 5), P2 (arrival 1, CPU 4), and P3 (arrival 2, CPU 2) are scheduled with this Gantt chart: P1 0–2, P2 2–4, P3 4–6, P1 6–8, P2 8–10, P1 10–11. Let us find the criteria for the three processes.

1. **Completion times.** The last segment of each process in the Gantt chart: P1 finishes at 11, P2 at 10, P3 at 6.
2. **Turnaround = completion − arrival.** P1: 11 − 0 = 11, P2: 10 − 1 = 9, P3: 6 − 2 = 4. The average is (11 + 9 + 4) / 3 = 8.
3. **Waiting = turnaround − CPU.** P1: 11 − 5 = 6, P2: 9 − 4 = 5, P3: 4 − 2 = 2; the average is 13 / 3 ≈ 4.33. P1's waiting can also be counted directly: it is in the queue during the intervals 2–6 and 8–10, so 4 + 2 = 6.
4. **Response = first run − arrival.** P1: 0 − 0 = 0, P2: 2 − 1 = 1, P3: 4 − 2 = 2; the average is 1.

Because P1 starts immediately, its response time is 0, but because it waited in the queue twice, its waiting time is 6. Response measures the first wait; waiting measures all waits.

Common mistake

Computing waiting time as "completion − CPU time." The formula is *turnaround* − CPU time, and turnaround is completion − arrival. If the arrival time is not zero, the two give different results. In Example 5.2, completion − CPU for P2 gives 6, while the correct waiting is 10 − 1 − 4 = 5.

5.2 Basic algorithms

The best way to understand algorithms is to run them by hand. The algorithms in this section are traced step by step on the same four processes; at the end of each, the average waiting, turnaround, and response times and the number of context switches are given. This way you can see in numbers what an algorithm improves and what it makes worse.

5.2.1 The common process set

Table 5.3 gives the common set used throughout the chapter. The four processes arrive one millisecond apart; their CPU times range from 4 to 9 ms. The priority column is used only in priority scheduling (Figure 5.10).

Process	Arrival	CPU time	Priority
P1	0	8	2
P2	1	4	3
P3	2	9	1
P4	3	5	4

Table 5.3: The common process set (times in ms).

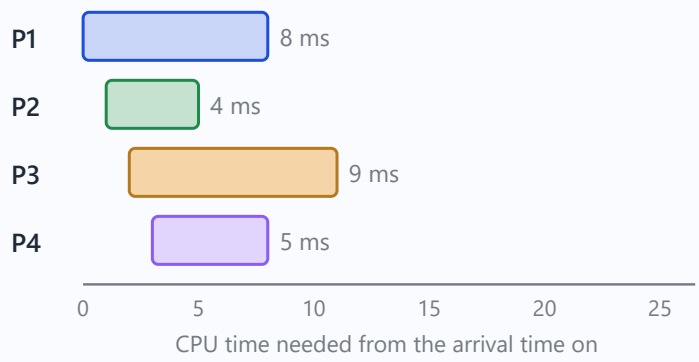


Figure 5.10: The timeline of the common set: each bar starts at the process's arrival time and extends for the CPU time it needs.

Four rules apply in the calculations. A smaller priority number means a higher priority. In a tie, the process that arrived first wins. In round robin, if a process arrives at the same instant another process's slice expires, the newly arrived process enters the queue first. Context-switch time is taken to be zero, and the processes do no I/O. These assumptions are not realistic, but they show the differences between the algorithms plainly.

Context switches are counted as follows: if two consecutive segments in the Gantt chart belong to different processes, there is one switch. A process's first appearance on the CPU is not counted as a switch. If a process is alone in the queue at the end of its slice, moving on to a new slice is not a switch either.

5.2.2 FCFS

Definition 5.6

In **first-come, first-served** (FCFS) scheduling, the ready queue is FIFO. A process that gets the CPU keeps it until it finishes or blocks. FCFS is nonpreemptive.

FCFS is the easiest algorithm to implement: a new process is appended to the end of the queue, and when the CPU becomes free the head is removed. Both operations are O(1). The trace on the common set goes as follows:

- **t = 0.** Only P1 is in the queue. P1 gets the CPU and does not release it for 8 ms. During this time P2, P3, and P4 arrive and join the queue in order.
- **t = 8.** P1 finishes. P2, at the head of the queue, runs. P2 had arrived at 1, so it waited 8 − 1 = 7 ms.
- **t = 12.** P2 finishes and P3 runs; it waited 12 − 2 = 10 ms.
- **t = 21.** P3 finishes and P4 runs; it waited 21 − 3 = 18 ms. P4 finishes at 26 (Figure 5.11; Table 5.4).

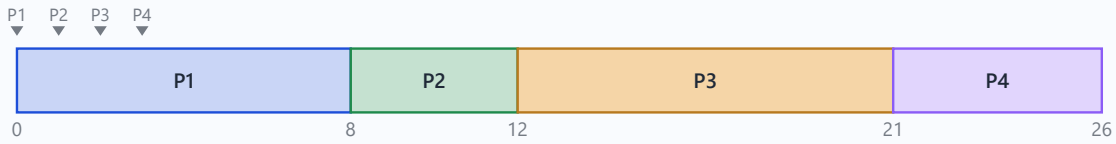


Figure 5.11: FCFS Gantt chart: the processes run in arrival order without interruption. P1 0–8, P2 8–12, P3 12–21, P4 21–26.

Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	12	21	19	10	10
P4	3	5	21	26	23	18	18
Avg.					15,25	8,75	8,75

Table 5.4: Criteria of the common set under FCFS.

The average waiting time is 35 / 4 = 8.75 ms and the average turnaround time is 61 / 4 = 15.25 ms. In nonpreemptive algorithms, once a process first starts running it continues until it finishes, so response time equals waiting time. The Gantt chart has three switches: P1→P2, P2→P3, P3→P4.

5.2.3 The convoy effect

FCFS's average waiting time is not constant; it is extremely sensitive to the arrival order of the processes. Silberschatz's classic example shows this clearly.

Example 5.3

P1 (24 ms), P2 (3 ms), and P3 (3 ms) are ready at t = 0. What is the average waiting time under FCFS for two different arrival orders?

1. **Order P1, P2, P3.** P1 0–24, P2 24–27, P3 27–30. The waiting times are 0, 24, 27; the average is (0 + 24 + 27) / 3 = 17 ms.
2. **Order P2, P3, P1.** P2 0–3, P3 3–6, P1 6–30. The waiting times are P1 6, P2 0, P3 3; the average is (6 + 0 + 3) / 3 = 3 ms.
3. **Comment.** The jobs are the same and the total time is the same (30 ms); only the order changed, and the average waiting time fell from 17 to 3. If a long job is in front, all the short jobs wait for as long as it runs (Figure 5.12).

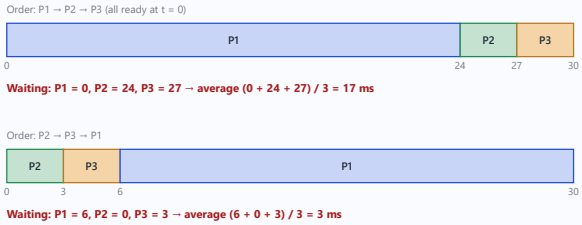


Figure 5.12: Two orders of the same three jobs. With the long P1 in front the average waiting time is 17 ms; with the short jobs in front it is 3 ms.

The problem grows worse in a dynamic environment where processes do I/O. Consider one CPU-bound process C and several I/O-bound processes (I1, I2, I3). While C holds the CPU with its long burst, the I/O of the I/O-bound processes completes and they all line up in the ready queue behind C. Meanwhile the disk is idle. When C finally goes to I/O, I1, I2, and I3 quickly finish their short bursts and go to the disk. This time the CPU is idle. Then C comes back and the cycle repeats.

Definition 5.7

The **convoy effect** is the accumulation of processes with short bursts behind a single process with a long burst. As a result the CPU and the I/O devices are idle in turn, and their utilization drops (Figure 5.13).

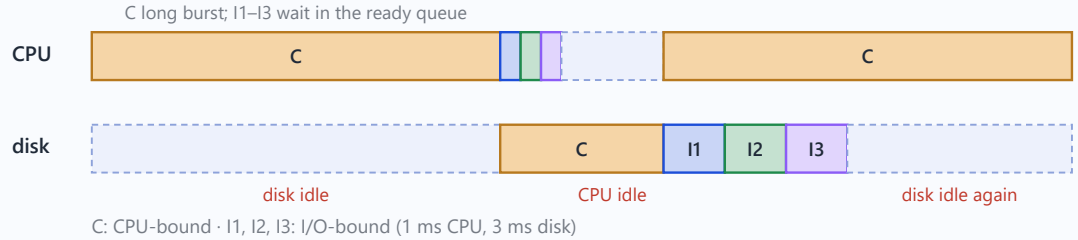


Figure 5.13: The convoy effect. While C holds the CPU with its long burst, I1–I3 wait in the ready queue and the disk is idle. When C moves to the disk, the I/O-bound processes finish their short bursts and queue up at the disk, and this time the CPU is idle.

There is no starvation in FCFS: every process runs once those ahead of it are done. But because it is nonpreemptive, a single long job can hold the system for a long time. This is why FCFS is not suitable for interactive systems. Still, it lives on inside many algorithms: to break ties among processes of equal priority, arrival order is usually used.

5.2.4 Code trace: FCFS waiting time

FCFS's waiting-time calculation can be done with a single loop. The following C function takes an array of processes sorted by arrival time, prints each process's waiting time, and prints the average.

```c
/* p[] sorted by arrival time */
void fcfs(Proc p[], int n) {
  int t = 0, sum = 0;
  for (int i = 0; i < n; i++) {
    if (t < p[i].arrival) t = p[i].arrival;
    int wait = t - p[i].arrival;
    t += p[i].burst;          /* completion */
    sum += wait;
    printf("%s: %d\n", p[i].name, wait);
  }
  printf("avg = %.2f\n", (double) sum / n);
}
```

`t` is the clock holding the moment the CPU becomes free; `sum` is the total of the waiting times. For each process the loop first checks whether the clock is behind the process's arrival. If it is, the CPU sat idle, and the clock is advanced to the arrival time. On the common set this line never runs, because the processes arrive before the CPU becomes free, but it is necessary for sets with gaps. Then the waiting time is computed as `t - p[i].arrival`: the time from the process's arrival until it gets the CPU. `t += p[i].burst` moves the clock to the process's completion. At the end the total is divided by the number of processes; the `(double)` cast prevents integer division (Table 5.5).

i	Process	`t` (at start)	wait	`t` (at end)	`sum`	Output
0	P1 (0, 8)	0	0 − 0 = 0	8	0	`P1: 0`

i	Process	`t` (at start)	wait	`t` (at end)	`sum`	Output
1	P2 (1, 4)	8	8 − 1 = 7	12	7	`P2: 7`
2	P3 (2, 9)	12	12 − 2 = 10	21	17	`P3: 10`
3	P4 (3, 5)	21	21 − 3 = 18	26	35	`P4: 18`
—						`avg = 8.75`

Table 5.5: `fcfs()` on the common set. Each row shows the state at the end of a loop step.

The function runs in O(n) time and uses no extra memory. The array must be sorted; sorting takes O(n log n). The decimal separator in the program's output is a point because `printf` works with the C locale.

5.2.5 SJF: shortest job first

The convoy example offers a hint: putting short jobs first lowers the average waiting time. SJF turns this idea into a rule.

Definition 5.8

In **shortest-job-first** (SJF) scheduling, when the CPU becomes free, the ready process whose *next CPU burst* is shortest is chosen; ties are broken by FCFS. In these notes the name SJF denotes the nonpreemptive version; the preemptive version is called SRTF.

The name of the algorithm is slightly misleading. The choice is determined not by the total length of the job but by the length of the next CPU burst. A more accurate name would be "shortest next CPU burst."

Theorem 5.1 (Optimality of SJF)

If all jobs are ready at the same time and their lengths are known, the SJF order minimizes the average waiting time (and the average turnaround time).

The reasoning is as follows. Let four jobs have lengths a, b, c, d and run in that order. The first finishes at time a, the second at a + b, the third at a + b + c, and the fourth at a + b + c + d. The average turnaround is (4a + 3b + 2c + d) / 4. The first job's length is added to all four turnaround times, the last one's to only one. The largest coefficient belongs to a; to make the sum small, the shortest job must go first. If two adjacent jobs are swapped so that the shorter one comes first, the sum either decreases or stays the same. These swaps eventually lead the order to the SJF order (Figure 5.14).

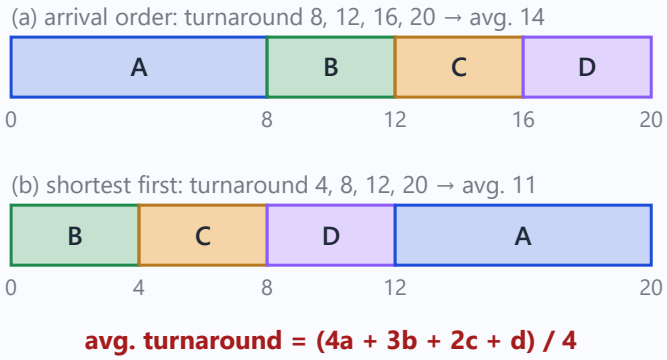


Figure 5.14: Four jobs: A is 8 units; B, C, and D are 4 units each. (a) In arrival order the turnaround times are 8, 12, 16, 20, average 14. (b) In shortest-job-first order they are 4, 8, 12, 20, average 11. The average turnaround is (4a + 3b + 2c + d) / 4.

The theorem has two conditions, and both matter. The first is that the jobs are ready at the same time. If the jobs arrive at different times, nonpreemptive SJF is not optimal: if a short job arrives right after a long job has started, SJF must make it wait. In that case the optimal version is the preemptive one, SRTF. The second condition is that the lengths are known. In batch systems the user gives a time limit when submitting a job; the `--time` option in Slurm is an example. A job that gives a short limit has a better chance of running first, but if the time is exceeded the job is terminated. In short-term scheduling the next burst can never be known; it can only be predicted from past bursts.

SJF also has a starvation problem. If short jobs keep arriving, a long job may never be chosen. SJF aims to minimize average waiting and turnaround time; it does not increase CPU utilization. In every algorithm in this chapter the CPU is already busy as long as there is a ready job.

Example 5.4

Four processes are ready at t = 0: P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Let us find the schedule and averages under SJF and compare with FCFS.

1. **t = 0.** All are ready; P4 (3), with the shortest CPU time, is chosen.
2. **t = 3.** P4 is done. Ready: P1 (6), P2 (8), P3 (7). The shortest is P1 (6); it runs 3–9.
3. **t = 9.** Ready: P3 (7), P2 (8). P3 runs 9–16.
4. **t = 16.** Only P2 is left; it runs 16–24.
5. **Averages.** Waiting times are P1 3, P2 16, P3 9, P4 0; the average is 28 / 4 = 7 ms. Turnaround times are 9, 24, 16, 3; the average is 13 ms. Had the same set run under FCFS (in the order P1, P2, P3, P4), the waiting times would be 0, 6, 14, 21 and the average 41 / 4 = 10.25 ms (Figure 5.15).

Figure 5.15: SJF when all processes are ready at time 0: P4 0–3, P1 3–9, P3 9–16, P2 16–24.

5.2.6 SJF on the common set and weighted turnaround

On the common set the processes arrive at different times, and the weakness of nonpreemptive SJF shows. At t = 0 only P1 is present and it gets the CPU. Short jobs that arrive later cannot stop P1.

- **t = 0.** Only P1 is ready; it runs 0–8.
- **t = 8.** P1 is done. Ready: P2 (4), P3 (9), P4 (5). The shortest is P2; it runs 8–12.
- **t = 12.** Ready: P3 (9), P4 (5). The shortest is P4; it runs 12–17.
- **t = 17.** Only P3 is left; it runs 17–26 (Figure 5.16; Table 5.6).

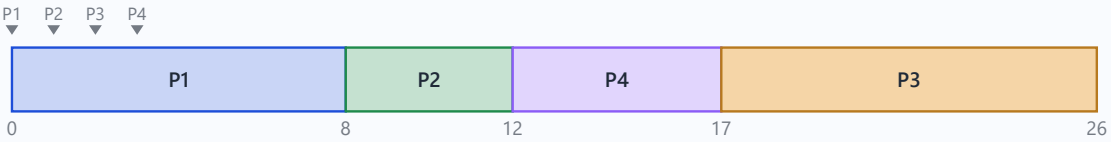


Figure 5.16: Nonpreemptive SJF on the common set: P1 is not interrupted; then the shortest jobs run in turn. P1 0–8, P2 8–12, P4 12–17, P3 17–26.

Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	17	26	24	15	15
P4	3	5	12	17	14	9	9
Avg.					14,25	7,75	7,75

Table 5.6: Criteria of the common set under nonpreemptive SJF.

Compared with FCFS, SJF lowered the average waiting time from 8.75 to 7.75. The first two processes ran the same as under FCFS; the difference comes from P4 getting ahead of P3. P4's waiting dropped from 18 to 9 and P3's rose from 10 to 15: the 9 ms gained for the short job is more than the 5 ms taken from the long one.

Weighted turnaround time makes this trade more visible. Table 5.7 compares two algorithms on another set of five jobs.

		A	B	C	D	E	Avg.
	Arrival	0	1	2	3	4	
	CPU	4	3	5	2	4	
SJF	Completion	4	9	18	6	13	
	Turnaround	4	8	16	3	9	8
	Weighted	1	2,67	3,2	1,5	2,25	2,12
FCFS	Completion	4	7	12	14	18	
	Turnaround	4	6	10	11	14	9
	Weighted	1	2	2	5,5	3,5	2,8

Table 5.7: Completion, turnaround, and weighted turnaround (turnaround / CPU time) under FCFS and SJF.

Under SJF, A runs 0–4. At t = 4, B (3), C (5), D (2), and E (4) are ready; the shortest, D, runs 4–6. Then come B 6–9, E 9–13, and last C 13–18. C's weighted turnaround is 16 / 5 = 3.2. The average for SJF is 2.12 and for FCFS 2.8. Under FCFS the largest penalty falls on the 2 ms job D: its turnaround of 11 ms is 5.5 times its CPU time.

FCFS's penalizing of short jobs is more striking on an extreme set. In the set A (arrival 0, CPU 1), B (1, 100), C (2, 1), D (3, 100), the 1 ms job C waits 99 ms behind B. C's turnaround is 100 and its weighted turnaround is 100 / 1 = 100. D, which suffers the same wait, has a weighted turnaround of only 199 / 100 = 1.99. A 99 ms wait is negligible for a long job and a disaster for a short one.

5.2.7 Predicting the next burst: the exponential average

The only way to use SJF in short-term scheduling is to predict the next burst. The usual method is the exponential average of past bursts.

Definition 5.9

The **exponential average** prediction of the next burst is

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n,\ 0 \le \alpha \le 1$$

Here t_n is the measured actual length of the n-th burst; τ_n is the prediction made for that burst; and τ_0 is a constant chosen at the start (for example, the system average).

The formula builds the new prediction as a mixture of two pieces of information: the latest measurement and the old prediction, which summarizes the past. α sets their weights. If $\alpha = 0$, the latest measurement is ignored and the prediction stays at τ_0. If $\alpha = 1$, only the last burst counts; the prediction equals the last measured burst. A common choice is $\alpha = ½$: the latest measurement and the past get equal weight.

The name becomes clear when the formula is expanded:

> $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + (1-\alpha)^2\,\alpha\, t_{n-2} + \ldots + (1-\alpha)^{n+1}\,\tau_0$

Because $1-\alpha$ is less than one, each older measurement has a weight that is $(1-\alpha)$ times that of the one before it. For $\alpha = ½$ the latest measurement has weight ½, the one before it ¼, and the one before that ⅛. Tanenbaum describes the same idea under the name **aging**: the estimates go T_0, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, and so on. This "aging" should not be confused with aging in priority scheduling; the latter is a method for preventing starvation.

The following program processes a sequence of seven bursts with $\alpha = 0.5$ and $\tau_0 = 10$.

```
double tau = 10.0;      /* τ0 */
const double a = 0.5;  /* α */
int t[] = { 6, 4, 6, 4, 13, 13, 13 };
for (int n = 0; n < 7; n++) {
  printf("%.1f %d\n", tau, t[n]);
  tau = a * t[n] + (1 - a) * tau;
}
printf("next: %.1f\n", tau);
```

At each step the loop first prints the prediction made for that burst and the actual value, then updates the prediction. The update line corresponds exactly to the formula: `a * t[n]` is the share of the latest measurement and `(1 - a) * tau` is the share of the old prediction. When the loop ends, the prediction that has seen all seven measurements is printed as "next" (Table 5.8).

n	prediction τ_n	actual t_n	computation	new τ_{n+1}
0	10	6	0,5 · 6 + 0,5 · 10	8
1	8	4	0,5 · 4 + 0,5 · 8	6
2	6	6	0,5 · 6 + 0,5 · 6	6
3	6	4	0,5 · 4 + 0,5 · 6	5
4	5	13	0,5 · 13 + 0,5 · 5	9
5	9	13	0,5 · 13 + 0,5 · 9	11
6	11	13	0,5 · 13 + 0,5 · 11	12

Table 5.8: Trace of the exponential average ($\alpha = 0.5$, $\tau_0 = 10$).

The program's output is the seven lines `10.0 6`, `8.0 4`, `6.0 6`, `6.0 4`, `5.0 13`, `9.0 13`, `11.0 13` and finally `next: 12.0`. In the first half of the sequence the bursts move back and forth between 4 and 6, and the prediction settles into that range. When the bursts jump to 13, the prediction catches up in a few steps, 5 → 9 → 11 → 12, closing half the remaining gap at each step. That the prediction follows reality with a delay is clearly visible in Figure 5.17.

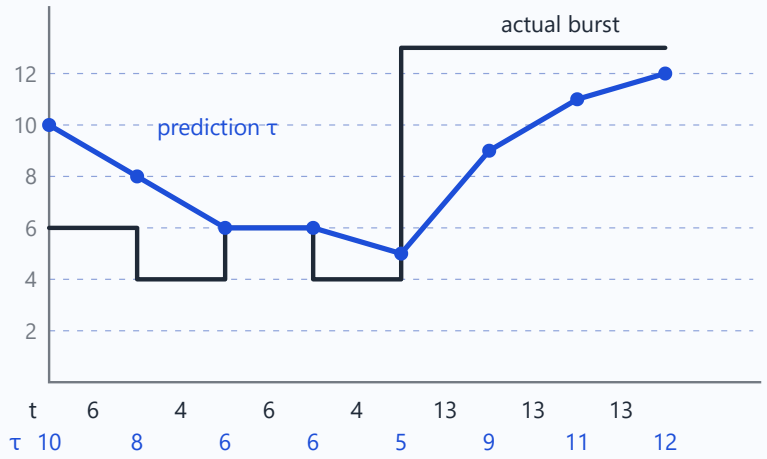


Figure 5.17: Exponential average ($\alpha = 0.5$, $\tau_0 = 10$). The stepped line shows the actual bursts (6, 4, 6, 4, 13, 13, 13) and the dotted curve the predictions (10, 8, 6, 6, 5, 9, 11, 12).

The choice of α is a trade-off. If the same sequence is run with $\alpha = 0.75$, the final prediction is ≈ 12.9 and the jump is caught faster; but a single outlying measurement also shakes the prediction that much more. With $\alpha = 0.25$ the final prediction stays at ≈ 10.3: the prediction is stable but adapts slowly to change.

Example 5.5

Let $\alpha = 0.5$ and $\tau_0 = 12$ ms. The first three measured bursts are 8, 6, and 10 ms. What are τ_1, τ_2, and τ_3? What would τ_3 be if $\alpha = 1$?

1. $\tau_1 = 0.5 \cdot 8 + 0.5 \cdot 12 = 4 + 6 = 10$ ms.
2. $\tau_2 = 0.5 \cdot 6 + 0.5 \cdot 10 = 3 + 5 = 8$ ms.
3. $\tau_3 = 0.5 \cdot 10 + 0.5 \cdot 8 = 5 + 4 = 9$ ms.
4. If $\alpha = 1$, only the last measurement counts: $\tau_3 = t_2 = 10$ ms. If α were 0, the prediction would never change and would stay at 12.

Common mistake

Confusing t_n with τ_n in the formula, or always computing the new prediction from τ_0. At each step the *result* of the previous step is used. In Example 5.5, $\tau_1 = 10$ is used to compute τ_2, not 12.

5.2.8 SRTF: shortest remaining time first

Definition 5.10

Shortest-remaining-time-first (SRTF; also called SRTN or STCF) is the preemptive version of SJF. When a new process arrives, its CPU time is compared with the *remaining* time of the running process; if the newcomer is shorter, the running process is preempted.

The SRTF trace on the common set:

- **t = 0.** Only P1 is present and it starts running.
- **t = 1.** P2 arrives (4 ms). P1's remainder is 8 − 1 = 7. Since 4 < 7, P1 is preempted and P2 runs.
- **t = 2.** P3 arrives (9). P2's remainder is 3, and 3 ≤ 9; no preemption.
- **t = 3.** P4 arrives (5). P2's remainder is 2, and 2 ≤ 5; no preemption.
- **t = 5.** P2 finishes. The remaining times are P1 7, P3 9, P4 5; the shortest, P4, is chosen and runs 5–10.
- **t = 10.** P4 finishes. The remaining are P1 7 and P3 9; P1 resumes where it left off, 10–17.
- **t = 17.** P1 finishes and P3 runs 17–26 (Figure 5.18; Table 5.9).

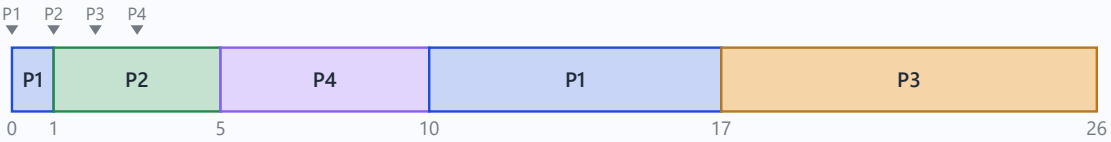


Figure 5.18: SRTF on the common set: P1 0–1, P2 1–5, P4 5–10, P1 10–17, P3 17–26. P1 is preempted at 1 and resumes where it left off at 10.

Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	17	17	9	0
P2	1	4	1	5	4	0	0
P3	2	9	17	26	24	15	15
P4	3	5	5	10	7	2	2
Avg.					13	6,5	4,25

Table 5.9: Criteria of the common set under SRTF.

On this set SRTF lowers the average waiting time to 6.5 ms; it is the best of the algorithms we have examined. The price is four context switches (P1→P2, P2→P4, P4→P1, P1→P3). In preemptive algorithms, response and waiting separate: P1's response time is 0, but its waiting time is 9, because it waited in the queue from 1 to 10.

SRTF is optimal for average waiting time when jobs arrive at different times, but it inherits both problems of SJF. The remaining times must be known, and long jobs may starve. Also, because a comparison is made at every arrival, the number of switches can grow.

Example 5.6

P1 (arrival 0, CPU 7), P2 (2, 4), P3 (4, 1), P4 (5, 4) are scheduled with SRTF. What are the Gantt chart and the average waiting time?

1. **t = 0–2.** Only P1 is ready; it runs 2 ms, leaving 5.
2. **t = 2.** P2 arrives: 4 < 5, so P1 is preempted. P2 runs 2–4, leaving 2.
3. **t = 4.** P3 arrives: 1 < 2, so P2 is preempted. P3 runs 4–5 and finishes.
4. **t = 5.** P4 arrives (4). The remaining are P1 5, P2 2, P4 4; the shortest, P2, finishes at 7 (running 5–7).
5. **t = 7.** The remaining are P1 5 and P4 4; P4 runs 7–11. Finally P1 runs 11–16.
6. **Waiting times.** P1: 16 − 0 − 7 = 9, P2: 7 − 2 − 4 = 1, P3: 0, P4: 11 − 5 − 4 = 2. The average is 12 / 4 = 3 ms.

5.2.9 Priority scheduling

Definition 5.11

In **priority scheduling** each process is given a priority number, and the ready process with the highest priority runs. In these notes a small number means high priority.

The direction of the number varies from system to system, and this is a common source of confusion. In Linux's `nice` value and in the kernel's internal priority scale, a small number means high priority. In Windows, a large number means high priority: 31 is the highest level.

Priorities come from internal or external factors. Internal priorities rest on measurable quantities: memory requirements, the number of open files, the measured ratio of average I/O burst to CPU burst. External priorities are set outside the operating system: the importance of the job, the fee paid, the organization's policy. SJF is in fact a special case of priority scheduling; the priority is determined by the predicted next burst. If all priorities are equal, the algorithm reverts to FCFS.

Priority scheduling can be preemptive or nonpreemptive. In the preemptive version, a newly arrived process preempts the running one if it has higher priority. In the nonpreemptive version the new process is put at the head of the queue but waits for the running one to finish. Among processes at the same priority level RR can be applied; Figure 5.19(b) shows this combination.

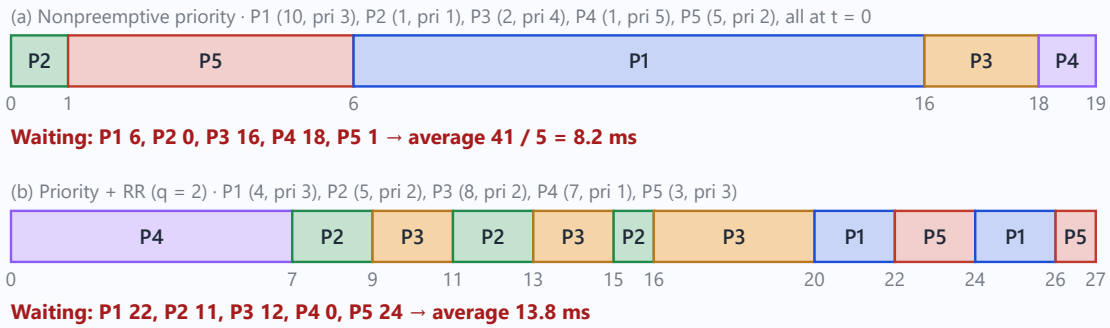


Figure 5.19: (a) Nonpreemptive priority, all ready at t = 0: the order is P2, P5, P1, P3, P4; the average waiting time is 41 / 5 = 8.2 ms. (b) Priority with RR (q = 2) among equal priorities: first P4, the highest priority, alone; then P2 and P3 alternating; last P1 and P5 alternating; the average waiting time is 69 / 5 = 13.8 ms.

In case (b), P4 at priority 1 runs alone from 0 to 7. At priority 2, P2 (5) and P3 (8) alternate in two-millisecond slices: P2 7–9, P3 9–11, P2 11–13, P3 13–15, P2 finishes at 16 after 15–16, and P3 finishes at 20 after 16–20. At priority 3, P1 (4) and P5 (3) share 20–27 in the same way. The waiting times are P1 22, P2 11, P3 12, P4 0, P5 24.

An important side effect of priority scheduling is **priority inversion**. Suppose a high-priority process H waits for a lock held by a low-priority process L. L must run to release the lock, but medium-priority processes M keep getting ahead of L. As a result H ends up waiting for all the M processes, which have lower priority than itself. The solution is **priority inheritance**: the process L that holds the lock temporarily takes over the priority of the waiter until the lock is released.

On the common set, preemptive priority scheduling proceeds as follows (priorities P1 2, P2 3, P3 1, P4 4):

- **t = 0.** Only P1 is present and it runs.
- **t = 1.** P2 arrives (priority 3). The running P1's priority (2) is higher; no preemption.
- **t = 2.** P3 arrives (priority 1). It is higher than P1; P1 is preempted and P3 runs.
- **t = 3.** P4 arrives (priority 4). It is lower than P3; no preemption.
- **t = 11.** P3 finishes. Ready: P1 (2), P2 (3), P4 (4). P1 runs its remaining 6 ms, 11–17.
- **t = 17.** P2 runs 17–21, then P4 runs 21–26 (Figure 5.20; Table 5.10).

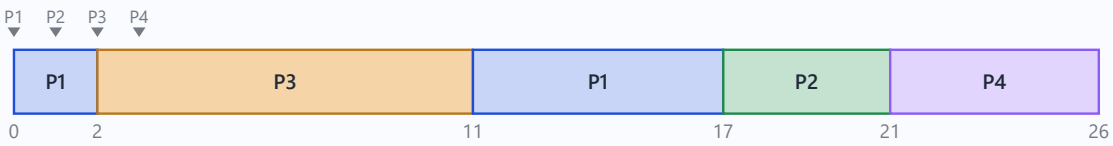


Figure 5.20: Preemptive priority on the common set: P1 0–2, P3 2–11, P1 11–17, P2 17–21, P4 21–26. P1 is preempted when the high-priority P3 arrives.

Process	Arrival	CPU	Priority	First run	Completion	Turnaround	Waiting	Response
P1	0	8	2	0	17	17	9	0
P2	1	4	3	17	21	20	16	16
P3	2	9	1	2	11	9	0	0
P4	3	5	4	21	26	23	18	18
Avg.						17,25	10,75	8,5

Table 5.10: Criteria of the common set under preemptive priority.

With an average waiting time of 10.75 ms it is even worse than FCFS. This is not surprising: the priority algorithm aims not at waiting time but at not keeping important jobs waiting. The most important process, P3, never waited; the least important, P4, waited 18 ms.

Example 5.7

P1 (arrival 0, CPU 5, priority 3), P2 (2, 4, 1), P3 (3, 2, 4), P4 (5, 3, 2), P5 (6, 1, 5). What are the Gantt chart and the averages under preemptive priority?

1. **t = 0–2.** P1 runs. At t = 2, the arriving P2 (priority 1) preempts P1.
2. **t = 2–6.** P2 runs. P3 (4) and P4 (2) arrive but are lower than P2; P2 finishes at 6.
3. **t = 6.** Ready: P1 (3), P3 (4), P4 (2), P5 (5). P4 runs 6–9, P1 9–12, P3 12–14, P5 14–15.
4. **Criteria.** Completion times are P1 12, P2 6, P3 14, P4 9, P5 15. Turnaround times are 12, 4, 11, 4, 9; the average is 40 / 5 = 8 ms. Waiting times are 7, 0, 9, 1, 8; the average is 25 / 5 = 5 ms.

If the same set is run with nonpreemptive priority, P1 finishes uninterrupted at 0–5, followed by P2 5–9, P4 9–12, P3 12–14, and P5 14–15. The average waiting time is 24 / 5 = 4.8 ms, which is better than the preemptive result. Preemption does not always improve the average; it only ensures that the high-priority job is not kept waiting.

5.2.10 Starvation and aging

Definition 5.12

Starvation is a process continually losing out to others and never getting to run. **Aging** is a method of preventing starvation by gradually raising the priority of a process that waits in the ready queue.

Starvation is a real problem in priority scheduling. If high-priority jobs keep arriving, a low-priority job can wait forever. According to a rumor, when the IBM 7094 at MIT was shut down in 1973, a low-priority job submitted in 1967 that had still not run was found. Aging puts an upper bound on this

wait.

Example 5.8

Nonpreemptive priority scheduling is used. The rule: for every 4 ms spent in the ready queue, the priority number drops by 1. L (3 ms, priority 7) arrives at t = 0; a new job H (4 ms, priority 2) arrives every 4 ms. When does L run?

1. **t = 0.** L's priority is 7, the newly arrived H1's is 2. H1 runs. When L has waited 4 ms its priority becomes 6.
2. **t = 4, 8, 12, 16.** L's priority at these moments is 6, 5, 4, and 3 in turn. Each time, the newly arrived H (priority 2) is higher and runs; L's priority number drops by one after each round.
3. **t = 20.** L's priority has fallen to 2, equal to that of the newly arrived H6. In a tie the one that arrived first wins: L runs and finishes at 23. L waited 20 ms.

In the worst case L waits priority difference × 4 ms = (7 − 2) × 4 = 20 ms. Without aging, L's priority would stay at 7 as long as the H stream continued, and L would never run (Figure 5.21).

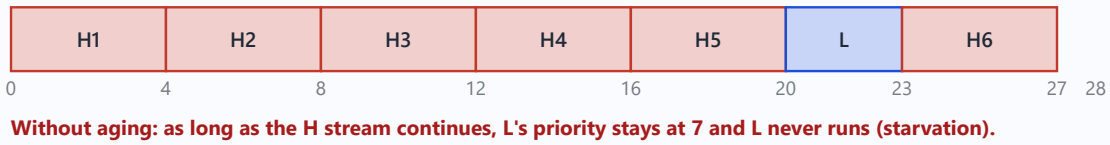


Figure 5.21: Thanks to aging, L runs at ms 20: H1–H5 run 4 ms each, L 20–23, H6 23–27.

Aging does not eliminate starvation, but it turns an infinite wait into a bounded one. The bound is proportional to the priority range and the time between two increments. Suppose priorities in a system range from 0 to 63 and are raised every 10 seconds. A process with priority 63 reaches the highest priority in at most 63 × 10 = 630 seconds (10.5 minutes), even if it never runs.

Common mistake

Counting priority values when computing the aging time. Going from 63 down to 0 takes 63 steps, not 64 values. The waiting time is the number of steps times the step interval: 63 × 10 s = 630 s, not 640 s.

5.3 Round robin

FCFS is fair but nonpreemptive; SJF and SRTF are good on average but need to know the times. Interactive systems, on the other hand, need an algorithm that gives everyone a quick response without time information. Round robin is the oldest and most common answer to that need.

5.3.1 Cyclic order and the time quantum

Definition 5.13

Round robin (RR) is FCFS with preemption added. The ready queue is FIFO; each process runs for at most one **time slice** (time quantum) q. When the slice expires, a clock interrupt arrives and the process goes to the end of the queue. If the process finishes or blocks before its slice is up, it releases the CPU at once (Figure 5.22).

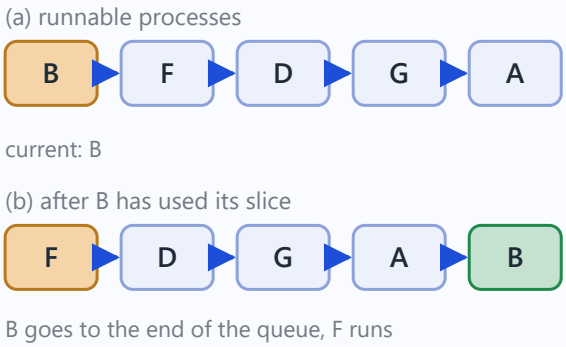


Figure 5.22: Round robin. (a) The list of runnable processes; the current process is B. (b) After B has used its slice it goes to the end of the list, and the next process, F, runs.

The most valuable property of RR is that it gives a definite upper bound on waiting. If there are n ready processes in the queue and the slice is q, no process waits longer than $(n - 1) \cdot q$ for its next slice, because each process ahead of it uses at most one slice. This bound holds when the number of ready processes n stays constant and context-switch time is neglected; if a new process enters the queue, or each switch takes s, the wait grows accordingly. For example, with 5 processes and a 20 ms slice, every process gets onto the CPU at least once every 80 ms. Response time is bounded and there is no starvation.

The price is paid in turnaround time. Because RR runs the processes in turn, they all stretch out to the very end of the work; turnaround time is generally worse than under SJF. Response time, in contrast, is much better. Silberschatz's classic example shows the difference in numbers (Figure 5.23).

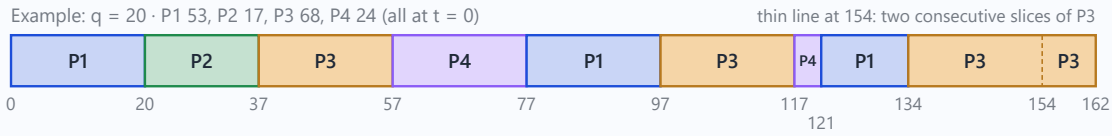


Figure 5.23: RR (q = 20) with P1 53, P2 17, P3 68, P4 24 ms, all ready at t = 0. The order is P1(0–20), P2(20–37), P3(37–57), P4(57–77), P1(77–97), P3(97–117), P4(117–121), P1(121–134), P3(134–162). The last P3 block consists of two consecutive slices (134–154, 154–162); since no other process is ready in between, P3 continues without interruption, and the thin line at 154 marks the slice boundary. The average waiting time is 73 ms and the average response time 28.5 ms; had the same set run under SJF, the average waiting time would be 38 ms.

In this example P2 finishes its 17 ms job in the first round. P4 uses its remaining 4 ms in the second round at 117–121; this is the narrow piece in the chart. The completion times are P1 134, P2 37, P3 162, P4 121. The waiting times are P1 134 − 53 = 81, P2 20, P3 162 − 68 = 94, P4 97; the average is 292 / 4 = 73 ms. The SJF order (P2, P4, P1, P3) gives waiting times 0, 17, 41, 94, and the average drops to 38 ms. But under SJF P3 gets onto the CPU for the first time at ms 94; under RR, at ms 37.

Another weakness of RR is its attitude toward I/O-bound processes. An I/O-bound process uses a small part of its slice and blocks for I/O (moves to a wait queue), and when the I/O completes it returns to the end of the ready queue. CPU-bound processes, on the other hand, use the whole slice every time. As a result CPU-bound processes get a larger share of the CPU. This imbalance is the starting point for improvements such as MLFQ and "virtual RR" (a separate, priority auxiliary queue for a process returning from I/O).

5.3.2 q = 2 and q = 5 on the common set

Let us trace RR on the common set with two different slices. Recall the rule: if new processes arrive while a process's slice is expiring, the new ones enter the queue first and the process whose slice expired enters after them.

Example 5.9

The common set, q = 2 ms. What are the Gantt chart and the criteria?

1. **t = 0–2.** Only P1 is present and it runs 2 ms (6 left). P2 and P3, arriving meanwhile, enter the queue before P1: the queue is P2, P3, P1.
2. **t = 2–4.** P2 runs (2 left). P4, arriving at t = 3, enters before it: the queue is P3, P1, P4, P2.
3. **t = 4–6.** P3 (7 left). **t = 6–8.** P1 (4 left). **t = 8–10.** P4 (3 left).
4. **t = 10–12.** P2 uses its last 2 ms and finishes at 12.
5. **t = 12–20.** P3 12–14 (5 left), P1 14–16 (2 left), P4 16–18 (1 left), P3 18–20 (3 left).

6. **t = 20–26.** P1 finishes at 20–22, P4 finishes at 22–23. P3 runs 23–25, and since no other process is in the queue for the remaining 1 ms, it immediately gets a new slice and finishes at 26 (Figure 5.24; Table 5.11).

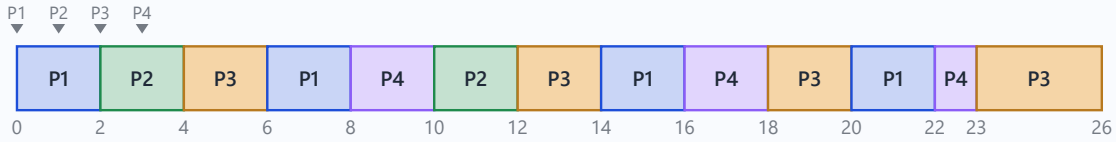


Figure 5.24: RR with q = 2 on the common set. The processes swap places often; everyone gets onto the CPU for the first time quickly.

Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	22	22	14	0
P2	1	4	2	12	11	7	1
P3	2	9	4	26	24	15	2
P4	3	5	8	23	20	15	5
Avg.					19,25	12,75	2

Table 5.11: Criteria of the common set under RR (q = 2).

The average response time fell to 2 ms; this is the best value so far. But with an average waiting time of 12.75 and an average turnaround of 19.25 ms, it is the worst. The Gantt chart has 13 segments and 12 context switches. In a real system each switch takes a few microseconds; although it is taken as zero in this small example, the overhead grows as the slice shrinks.

When the same set is run with q = 5, the table changes. P1 runs 0–5 (3 left); P2, P3, and P4, arriving meanwhile, enter the queue first. P2 finishes at 5–9 before its slice expires. P3 runs 9–14 (4 left) and P4 finishes at 14–19. P1 finishes at 19–22 and P3 at 22–26 (Figure 5.25; Table 5.12).

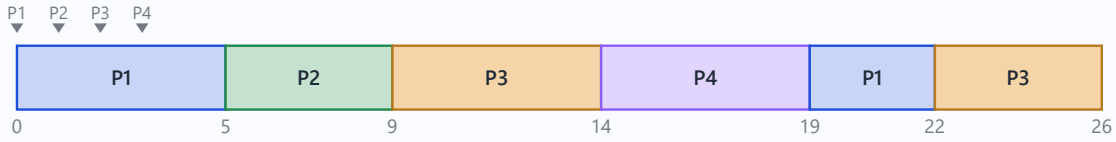


Figure 5.25: RR with q = 5 on the common set: P1 0–5, P2 5–9, P3 9–14, P4 14–19, P1 19–22, P3 22–26.

Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P1	0	8	0	22	22	14	0
P2	1	4	5	9	8	4	4
P3	2	9	9	26	24	15	7

Process	Arrival	CPU	First run	Completion	Turnaround	Waiting	Response
P4	3	5	14	19	16	11	11
Avg.					17,5	11	5,5

Table 5.12: Criteria of the common set under RR (q = 5).

With the larger slice the number of switches drops from 12 to 5; the average waiting time improves to 11 and the turnaround to 17.5. In return the average response rises from 2 to 5.5 ms. As the slice grows, RR approaches FCFS.

5.3.3 Code trace: `rr()`

The core of RR is a loop of a few lines. The following function takes the array sorted by arrival time and prints each process's completion time. The helper function `add_arrivals(p, n, &k, t)` adds, in order, the processes that have arrived by time `t` but have not yet entered the queue. `k` is the index of the next process in the array to be added.

```
/* p[] sorted by arrival time */
void rr(Proc p[], int n, int q) {
  int t = 0, k = 0;     /* k: next arrival */
  add_arrivals(p, n, &k, t);
  while (!empty() || k < n) {
    if (empty()) {       /* CPU idle */
      t = p[k].arrival;  /* next arrival */
      add_arrivals(p, n, &k, t);
    }
    Proc *s = dequeue(); /* head */
    int slice = min(q, s->left);
    t += slice; s->left -= slice;
    add_arrivals(p, n, &k, t); /* first */
    if (s->left > 0) enqueue(s);   /* then */
    else printf("%s %d\n", s->name, t);
  }
}
```

The loop continues as long as processes that have not yet arrived remain (`k < n`), even when the queue is empty. When the queue is empty the CPU is idle; the code advances the clock directly to the arrival time of the next process and adds that process to the queue. Without these lines the loop would end early and processes arriving later would never run. On the common set the CPU is never idle, so this branch does not run. Each round of the loop is one dispatch. The process at the head of the queue is removed and run for `min(q, s->left)` : if the remaining time is shorter than the slice, the process finishes without using up the slice. The clock is advanced and the remaining time reduced. The order of the next two lines determines RR's tie-breaking rule. First the new processes that arrived during this slice are added, and then the process that is not finished goes to the end of

the queue. If the two lines were swapped, the process whose slice expired would get ahead of the newcomers and the schedule would be different. A process that finishes does not return to the queue; its name and completion time are printed (Table 5.13).

Round	Running	`slice`	`t`	Left	Queue (head → tail)	Output
initially	—	—	0	—	P1	
1	P1	4	4	4	P2, P3, P4, P1	
2	P2	4	8	0	P3, P4, P1	`P2 8`
3	P3	4	12	5	P4, P1, P3	
4	P4	4	16	1	P1, P3, P4	
5	P1	4	20	0	P3, P4	`P1 20`
6	P3	4	24	1	P4, P3	
7	P4	1	25	0	P3	`P4 25`
8	P3	1	26	0	empty	`P3 26`

Table 5.13: `rr()` on the common set, q = 4. Each row shows the state at the end of a loop round.

When the queue is empty and no process is left to arrive, the loop ends: 8 dispatches and 7 context switches. The waiting times are P1 20 − 8 = 12, P2 8 − 1 − 4 = 3, P3 26 − 2 − 9 = 15, P4 25 − 3 − 5 = 17; the average is 11.75 ms. The average turnaround is 18.25 and the average response is (0 + 3 + 6 + 9) / 4 = 4.5 ms (Figure 5.26).

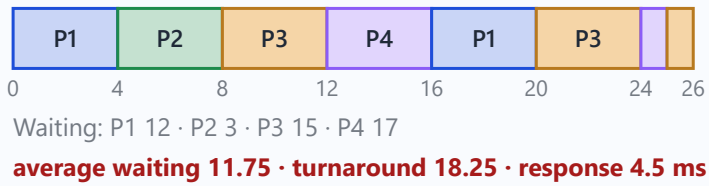


Figure 5.26: The result of the `rr()` trace (q = 4): P1 0–4, P2 4–8, P3 8–12, P4 12–16, P1 16–20, P3 20–24, P4 24–25, P3 25–26.

Each round runs in O(1) time: removing from and adding to the queue take constant time. The total number of rounds is the sum, over the processes, of each CPU time divided by q and rounded up. In this example it is 2 + 1 + 3 + 2 = 8. Consequently, as q shrinks the number of dispatches grows in inverse proportion.

Common mistake

Putting the process whose slice expired ahead of new processes arriving at the same moment. In these notes and in the slides, new arrivals enter first; in the code this is ensured by the `add_arrivals` line coming before the `enqueue(s)` line. Textbooks may use different rules on

this point; in an RR question, the rule must be settled first.

5.3.4 Choosing the time quantum

RR's only parameter is q, and its behavior depends largely on it. When the common set is run with different slices, Table 5.14 results (Figure 5.27).

Time quantum	Waiting	Turnaround	Response	Switches
q = 2	12,75	19,25	2	12
q = 3	13,5	20	3	9
q = 4	11,75	18,25	4,5	7
q = 5	11	17,5	5,5	5
q = 6	12	18,5	6,5	5
q ≥ 9 (FCFS)	8,75	15,25	8,75	3

Table 5.14: Averages (ms) and number of context switches by time quantum on the common set.

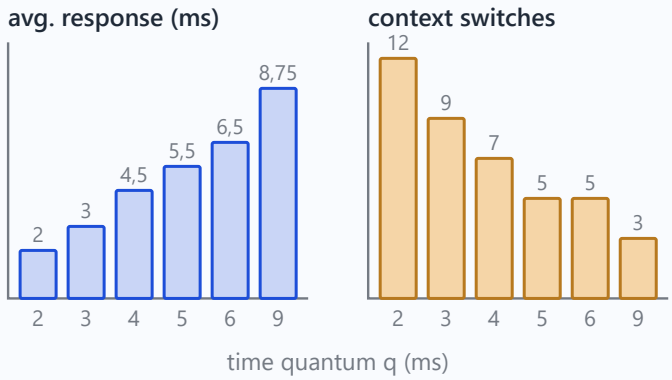


Figure 5.27: As the time quantum grows, average response time increases and the number of context switches decreases.

Three conclusions follow from the table. First, as q shrinks, response improves but the number of switches grows. Second, average waiting and turnaround do not vary monotonically with q: q = 3 is even worse than q = 2, and q = 5 is better than q = 6. Third, when q is equal to or larger than the longest burst (here P3's 9 ms), no process is preempted and RR becomes exactly FCFS.

That turnaround does not vary monotonically with q can be seen on another set as well. For P1 6, P2 3, P3 1, P4 7 ms (all at t = 0), the average turnaround for q = 1 to 7 is 11; 11.5; 10.75; 11.5; 12.25; 10.5; 10.5 (Figure 5.28, right). The general rule is this: if most processes can finish their work in a single slice, the average turnaround improves.

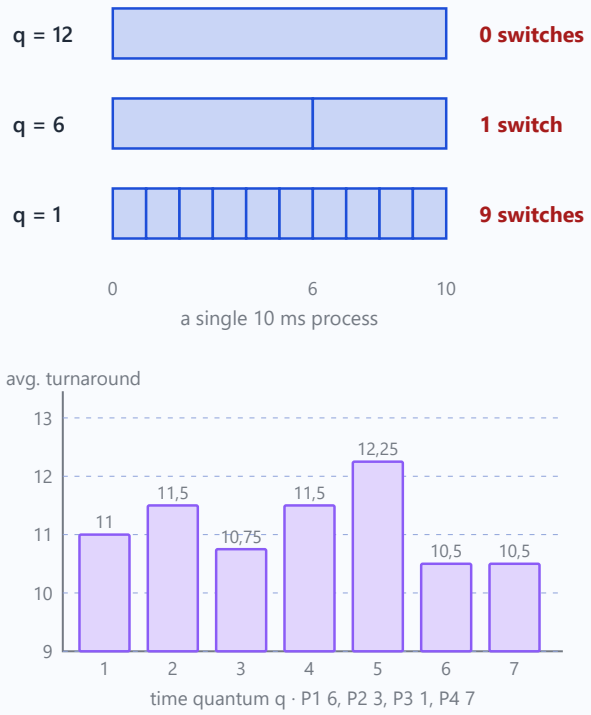


Figure 5.28: Left: a single 10 ms process; with q = 12 there is no switch, with q = 6 there is 1, with q = 1 there are 9. Right: the average turnaround time as a function of q for the set P1 6, P2 3, P3 1, P4 7.

If each context switch in a real system takes time s, the fraction of CPU time wasted with processes that use their slices to the end is

> overhead fraction = s / (q + s)

This formula assumes that every process uses its whole slice and that there is a switch after every slice; for processes that finish before their slice expires or block for I/O, the useful work is shorter than q and the fraction differs. Every round contains q of useful work and s of switching. If q = 4 ms and s = 1 ms, the overhead is 1 / 5 = 20%: one-fifth of CPU time goes only to switching processes. At q = 100 ms the overhead falls to ≈ 1%, but in a queue of ten processes each process waits nearly a second and the system responds sluggishly.

Common mistake

Computing the overhead fraction as s / q. The denominator is the whole round, that is, the sum of useful work and the switch: q + s. For q = 4, s = 1 this gives s / q = 25%; the correct value is 20%.

Today a context switch takes a few microseconds, and typical slices are between 1 and 100 ms. The direct switching overhead is therefore small; the real cost is the cooling of the cache. In Linux's EEVDF scheduler the base slice is computed with the formula 0.75 ms × (1 + $\log_2$ of the CPU count). The CPU count in the formula is capped at 8, so on 8 or more cores the slice is 3 ms. In Windows client versions the slice is about two clock ticks. Silberschatz's rule of thumb is this: about 80% of CPU bursts should be shorter than q. That way most processes go to I/O without using up their slice, and the slice limits only the long CPU jobs.

An example shows how a well-chosen slice breaks the convoy effect. In the set P1 24, P2 3, P3 3 ms, the FCFS average waiting time was 17 ms. With q = 4, P1 runs 0–4, P2 4–7, P3 7–10, and P1 10–30. The waiting times are 6, 4, and 7, an average of 17 / 3 ≈ 5.67 ms.

Example 5.10

A, B, C, D arrive at t = 0 in that order; each needs 4 ms of CPU. What are the average turnaround and response times under FCFS and under RR with q = 1?

1. **FCFS.** Completion times are 4, 8, 12, 16; the average turnaround is 40 / 4 = 10 ms. Response is (0 + 4 + 8 + 12) / 4 = 6 ms. Three switches.
2. **RR (q = 1).** The processes swap places every millisecond. The last round is at 12–16; completion times are 13, 14, 15, 16 and the average turnaround is 58 / 4 = 14.5 ms. Response is (0 + 1 + 2 + 3) / 4 = 1.5 ms. Fifteen switches.

For jobs of equal length, RR is among the best options in response and the worst in turnaround: everyone is kept waiting until the end of the work.

5.4 Multilevel queues

In a real system the processes are not homogeneous. A process playing audio, a text editor the user is typing in, and a script making a nightly backup should not compete fairly in the same queue. Multilevel queues divide processes into classes and give each class its own queue and its own rule.

5.4.1 Multilevel queue (MLQ)

Definition 5.14

In **multilevel queue** (MLQ) scheduling the ready queue is divided into several separate queues. Each process is assigned *permanently* to one queue according to its type (for example foreground or background). Each queue has its own algorithm, and there is also a separate scheduling among the queues (Figure 5.29).

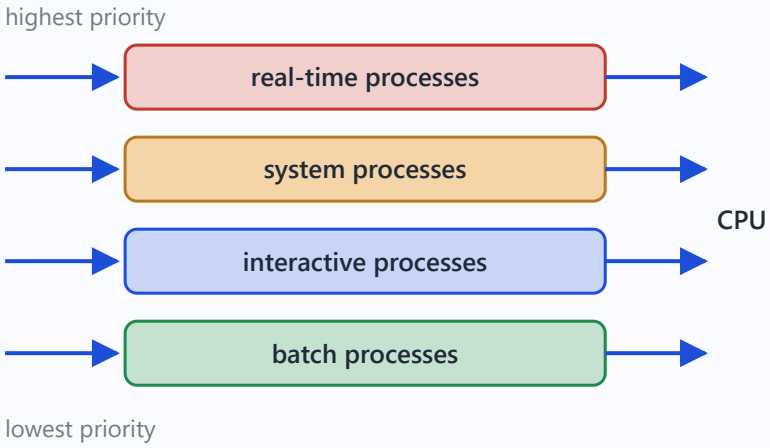


Figure 5.29: Multilevel queue: separate queues for real-time, system, interactive, and batch processes; priority decreases from top to bottom.

In a typical arrangement, interactive foreground processes are scheduled with short-slice RR and background batch processes with FCFS. There are two ways to choose among the queues. With **fixed priority**, a lower queue never runs until the upper queue is empty. A batch job waits as long as there are processes in the interactive queue and may starve. With **time slicing**, each queue is given a fixed fraction of the CPU; for example 80% to the foreground and 20% to the background.

MLQ has low overhead: a process is assigned to a queue once and never moved again. But it is not flexible. Even if a process changes its behavior over time, its queue does not change. A process whose class was misjudged either gets a priority it does not deserve or is deprived of the one it deserves. This fixed-assignment structure is MLQ; the structure in which processes change queues according to their behavior is called a feedback queue.

5.4.2 Multilevel feedback queue (MLFQ) and its rules

MLFQ gives a practical answer to the question asked at the very beginning of the chapter: how do we put short jobs first without knowing burst lengths? The answer is to observe the processes. Every new job is first assumed to be short and is put in the top queue. If it is really short, it finishes there quickly. If it is long, it uses up its slice and drops to a lower queue. In this way the algorithm learns from the past and approaches SJF.

Definition 5.15

A **multilevel feedback queue** (MLFQ) is a multilevel queue in which processes are moved among queues according to their observed behavior. In OSTEP's formulation it has five rules:

1. If Priority(A) > Priority(B), A runs.
2. If priorities are equal, A and B run in RR.
3. A new job enters the top queue.
4. A job that uses up its allotment at a level moves down one level, no matter how many times it has given up the CPU.
5. After some time period S, all jobs are moved to the top queue (**priority boost**; Figure 5.30).

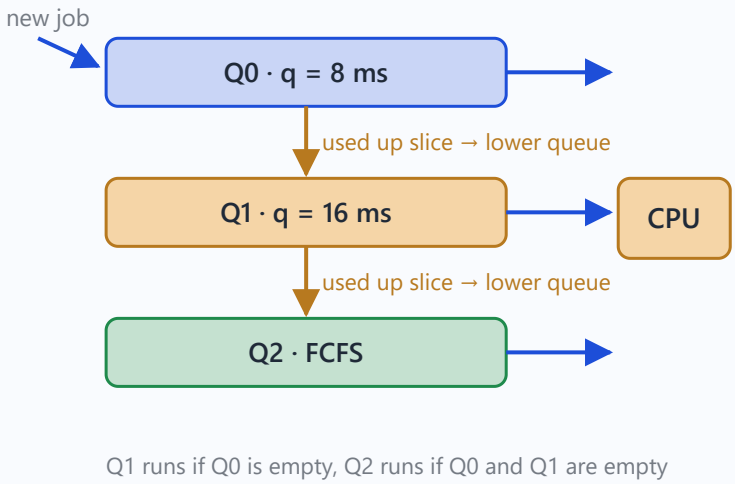


Figure 5.30: MLFQ: a new job enters Q0; a job that uses up its slice drops one queue lower. If Q0 is empty, Q1 runs; if Q0 and Q1 are empty, Q2 runs.

Each of the rules solves a problem. The first two set up the basic schedule. The third is the advantage given to new jobs: a short command runs at once and finishes. The fourth pushes long jobs down; counting the allotment as *total* usage at the level prevents cheating. The fifth prevents starvation and gives a job whose behavior has changed a new chance.

The parameters that define an MLFQ are: the number of queues, the algorithm and slice of each queue, when a job moves to a lower or higher queue, and which queue a new job enters. Usually the slices are short in the upper queues and long in the lower ones: at the top, short jobs cycle quickly; at the bottom, long jobs run less often but for longer. This flexibility makes MLFQ the most general algorithm and at the same time the hardest to tune.

5.4.3 MLFQ step by step

Example 5.11

Three queues: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). Jobs (arrival, CPU): A (0, 30), B (2, 6), C (4, 20), D (25, 4). When a job arrives in a higher queue, the preempted job returns to the head of its own queue and completes its slice.

1. **t = 0.** A arrives, enters Q0, and runs with an 8 ms slice. Meanwhile B (t = 2) and C (t = 4) are added to Q0.
2. **t = 8.** A has used up its slice and drops to Q1 (22 left). Next in Q0, B runs.
3. **t = 14.** B finishes in Q0 after 6 ms, before its slice is used up. C runs 8 ms in Q0.
4. **t = 22.** C also drops to Q1 (12 left). Q0 is empty; A, at the head of Q1, starts running with a 16 ms slice.
5. **t = 25.** D arrives and enters Q0. Because Q0 is above Q1, A is preempted and returns to the head of Q1; it has used 3 ms of its slice and 13 ms remain.
6. **t = 29.** D finishes after 4 ms; it never waited. A uses the remaining 13 ms of its slice in Q1.
7. **t = 42.** A has used up its Q1 slice too and drops to Q2 (6 left). C runs in Q1; its remaining 12 ms fit in the 16 ms slice and it finishes at 54.
8. **t = 54.** Q0 and Q1 are empty. A runs its remaining 6 ms in Q2 under FCFS and finishes at 60 (Figure 5.31; Table 5.15).

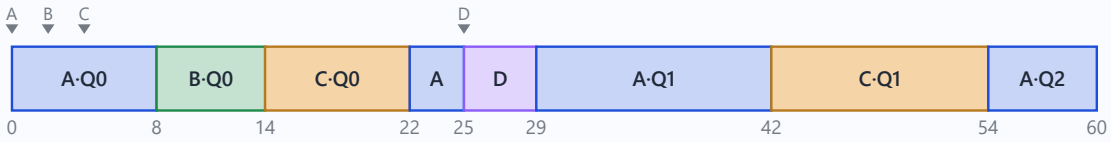


Figure 5.31: Gantt chart of the MLFQ trace; the label on each segment shows the job and the queue it ran in: A·Q0 0–8, B·Q0 8–14, C·Q0 14–22, A 22–25, D 25–29, A·Q1 29–42, C·Q1 42–54, A·Q2 54–60.

Job	Arrival	CPU	Completion	Turnaround	Waiting	Response
A	0	30	60	60	30	0
B	2	6	14	12	6	6
C	4	20	54	50	30	10
D	25	4	29	4	0	0
Avg.				31,5	16,5	4

Table 5.15: Criteria of the MLFQ example.

The short jobs B and D finish quickly in the top queue; their response times are 6 and 0. The long job A filters downward and finishes last. The average response is 4 ms. MLFQ has put short jobs first without knowing any process's length: all it does is count how much CPU each job uses.

Example 5.12

With the same rules, take Q0 (q = 4), Q1 (q = 8), Q2 (FCFS) and A (0, 14), B (1, 3), C (3, 9), D (16, 2). When do the jobs finish?

1. A uses up its slice at 0–4 and drops to Q1. B finishes at 4–7 before its slice is used up (response 3). C uses up its slice at 7–11 and drops to Q1.
2. Q0 is empty; A, at the head of Q1, starts at 11 with an 8 ms slice. At t = 16 D arrives; A, having used 5 ms of its slice, is preempted and returns to the head of Q1.
3. D finishes at 16–18. A uses the remaining 3 ms of its slice at 18–21; it has run 4 + 8 = 12 ms in total in Q0 and Q1, and drops to Q2.
4. C finishes its remaining 5 ms in Q1 at 21–26. A runs its remaining 2 ms in Q2 at 26–28.

Completion times: B 7, D 18, C 26, A 28. The average response is (0 + 3 + 4 + 0) / 4 = 1.75 ms.

5.4.4 MLFQ in practice

Why MLFQ's rules are the way they are becomes clear from the problems that arise when they are not.

Gaming. Had the first form of the rule been "a job that uses up its slice moves down; a job that gives up the CPU before the slice ends stays at the same level," a malicious program could exploit it. The program does a short I/O just before the slice ends, for example using 9.9 ms of a 10 ms slice. It never drops to a lower queue and monopolizes most of the CPU. The solution is to count the allotment as total usage at the level (rule 4). However many pieces the job uses the CPU in, it drops when the total reaches 10 ms (Figure 5.32).

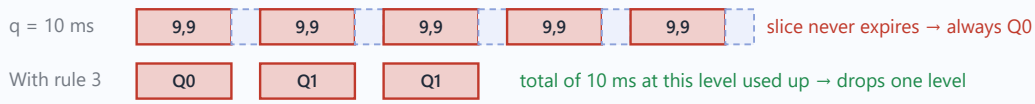


Figure 5.32: Fooling the MLFQ. Top: the job uses 9.9 ms of every slice and then does I/O, and always stays in Q0. Bottom: with rule 4, when the total of 10 ms at the level is used up, it drops one level.

Starvation and change of behavior. If there are many interactive jobs, the upper queues never empty and the long jobs in the lower queue starve. Also, if a long computation later becomes interactive, this goes unnoticed because it stays in the lower queue. Periodic priority boosting (rule 5)

solves both problems together. Choosing S is hard. If it is too large, long jobs still starve; if it is too small, interactive jobs do not get enough CPU. OSTEP calls such values "voodoo constants."

History. The MLFQ idea was first implemented by Corbató and his team in 1962 in the CTSS system at MIT. In CTSS the priority classes ran for 1, 2, 4, and 8 slices; a job that finished its slices dropped to a lower class. When Enter was pressed at the terminal, the process was moved to the top class, because the system assumed it had become interactive. Users soon noticed this and pressed Enter now and then to speed up their long jobs. This is one of the first examples of the gaming problem.

Today. The MLFQ idea lives on in many real systems. Solaris's time-sharing (TS) class defines each priority level's slice and its move-down and move-up rule with a table. FreeBSD's ULE scheduler, Windows's dynamic priorities, and macOS use the same idea. Linux, since version 2.6.23, has relied on fair sharing instead of such priority heuristics: first with CFS, then with EEVDF.

Common mistake

Believing that a priority boost prevents fooling MLFQ. A priority boost prevents starvation; the rule that stops a job from always using 9.9 ms of its slice and staying on top is the counting of total usage at the level. Sending every job that does I/O straight to the bottom is not a solution either: jobs that are truly interactive get punished.

5.5 Proportional share

The algorithms so far tried to improve turnaround or response time. Proportional-share schedulers set a different goal: each process or user should receive a predetermined share of the CPU. On a cloud server the customer wants the share they paid for; on a shared machine every user wants a fair share.

5.5.1 Guaranteed and fair-share scheduling

Definition 5.16

In **guaranteed scheduling**, if there are n users (or processes), each is promised 1/n of the CPU. The scheduler computes for each the value *ratio = CPU received / CPU entitled to* and runs the one with the lowest ratio. In **fair-share scheduling** the share is given not to processes but to users or groups.

In guaranteed scheduling a process whose ratio is 0.5 has received only half its share and goes to the front. A process whose ratio is 2.0 has received twice its share and yields to those with lower ratios. For the computation it is enough for the scheduler to keep track of how long each process has existed and how much CPU it has used since. Guaranteed scheduling in Tanenbaum's sense is this share guarantee; it does not mean running to a predetermined time schedule.

Per-process fairness rewards a user who starts many processes. Let user 1 run four processes (A, B, C, D) and user 2 a single process (E). With per-process RR the five processes get equal shares, and user 1 gets 80% of the CPU. In fair share the share is first divided among users: 50% to each user. User 1's share is divided among their four processes, and user 2's goes to E. The order takes the form A, E, B, E, C, E, D, E. Shares can also be set by policy; if user 1 is given a 2/3 share, two of every three slices go to their processes (Figure 5.33).

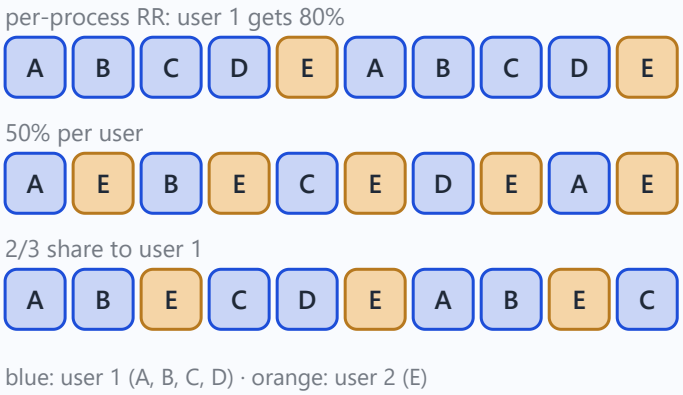


Figure 5.33: Fair share. User 1 runs four processes (A–D) and user 2 one process (E). With per-process RR user 1 gets 80%; with per-user sharing the two get 50% each; if policy requires, user 1 can be given a 2/3 share.

Today fair share on Linux is done with control groups (cgroups). In cgroup v2 each group has a `cpu.weight` value, and the CPU is divided according to the group weights. systemd puts each user in a group called `user-UID.slice`; so a user starting a hundred processes does not reduce another user's share. Kubernetes likewise translates the CPU requests given to containers into these weights.

5.5.2 Lottery scheduling

Definition 5.17

In **lottery scheduling** each process is given tickets. In every slice a ticket is drawn at random and the holder of the ticket runs. In the long run a process's CPU share is the ratio of its tickets to the total number of tickets (Figure 5.34).

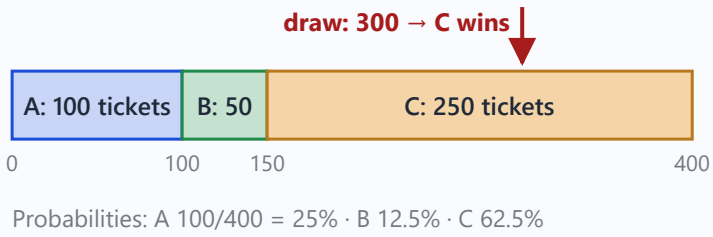


Figure 5.34: Lottery. A holds tickets numbered 0–99, B 100–149, C 150–399; the drawn number 300 falls in C's range. The probabilities are A 100/400 = 25%, B 12.5%, C 62.5%.

The most attractive aspect of the method proposed by Waldspurger and Weihl in 1994 is its simplicity. All that is needed is a random number generator and a running total over the list of processes. Expressing priority is easy too: a more important process gets more tickets. A newly arrived process can win as soon as it receives tickets; in this sense the algorithm responds at once.

Tickets also solve the hard parts of fair share. In **ticket transfer**, a client waiting for a reply from a server temporarily hands its tickets to the server; the server thus speeds up the work of the client that waits most. **Ticket currency** lets a user divide their own tickets among their own processes in any unit they like; the system converts it into a global value. In an environment with mutually trusting processes, **inflation** can also be used: a process that needs more CPU temporarily increases its own number of tickets.

The weakness of lottery is that it is probabilistic. In the short run it can be unfair: two jobs with equal tickets may split 10 slices 7–3. The share approaches the ticket ratio only as the number of draws grows. The method that achieves the same ratio deterministically is stride scheduling.

Example 5.13

A has 50 tickets, B has 30, and C has 20. If C is given 20 more tickets, what is the expected number of slices for C in 240 slices?

1. The total number of tickets becomes 50 + 30 + 40 = 120.
2. C's share is 40 / 120 = 1/3; the expected value in 240 slices is 240 / 3 = 80.

Computing with the old share (20%) gives 48; mistakenly taking the total to be 100 gives 40% and 96. 80 is an expected value; in a single run 75 or 86 may also be seen.

5.5.3 Code trace: the drawing loop

OSTEP gives lottery's decision loop as follows. The processes are in a linked list together with their ticket counts; `getrandom(0, totaltickets)` returns a number between 0 and the total number of tickets. OSTEP gives this function that name; it is pseudocode and not the Linux system call of the same name.

```
/* one draw per time slice */
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
  counter = counter + current->tickets;
  if (counter > winner)
    break;          /* found the winner */
  current = current->next;
}
/* current: process that gets the CPU */
run(current);
```

The loop scans the list from the head and keeps in the variable `counter` the total of the tickets seen so far. At each node the node's tickets are added to the total. The moment the total exceeds the drawn number, the drawn ticket lies in this node's range and the loop breaks. If it does not, the loop moves to the next node. For the trace, the drawn numbers are fixed as 47, 35, and 12 in turn; the ticket list is A 30, B 10, C 60 (total 100). A holds tickets 0–29, B 30–39, C 40–99 (Table 5.16).

Draw	Node	`counter`	`counter > winner` ?	Result
47	A	30	no	next node
47	B	40	no	next node
47	C	100	yes	C wins
35	A	30	no	next node
35	B	40	yes	B wins
12	A	30	yes	A wins

Table 5.16: The decision loop in three draws (ticket list A 30, B 10, C 60).

At each new draw the counter starts from zero and the list is scanned from the head. In three slices A, B, and C each ran once. Yet the expected shares are 30%, 10%, and 60%; in the short run the share is not proportional to the tickets, and only in the long run does it approach 30 : 10 : 60.

The cost of a draw is O(n) in the worst case, because the winner may be at the end of the list. If the list is sorted by decreasing ticket count, the processes with the most tickets come first and the scan gets shorter on average. In systems with many processes the selection can be reduced to O(log n) by keeping the running totals in a tree.

5.5.4 Stride scheduling

Definition 5.18

In **stride scheduling** each process has a **stride**: stride = a large constant / the number of tickets. Each process also keeps a **pass** counter, which starts at zero. In every slice the process with the smallest pass value runs and adds its own stride to its pass value.

A process with many tickets has a small stride; its pass value grows slowly and it is chosen often. A process with few tickets has a large stride; once it has run, it stays behind for a long time. Waldspurger proposed this method in 1995 as the deterministic counterpart of lottery (Table 5.17; Figure 5.35).

Example 5.14

A has 100 tickets, B has 50, and C has 250; the constant is 10,000. The strides are A 10,000 / 100 = 100, B 200, C 40. In a tie the order A, B, C applies. What is the order of execution over the first 8 slices?

Slice	pass A	pass B	pass C	Running	Update
1	0	0	0	A	pass(A) = 100
2	100	0	0	B	pass(B) = 200
3	100	200	0	C	pass(C) = 40
4	100	200	40	C	pass(C) = 80
5	100	200	80	C	pass(C) = 120
6	100	200	120	A	pass(A) = 200
7	200	200	120	C	pass(C) = 160
8	200	200	160	C	pass(C) = 200

Table 5.17: Stride trace. The pass values are those at the start of the slice, that is, at the moment of selection.

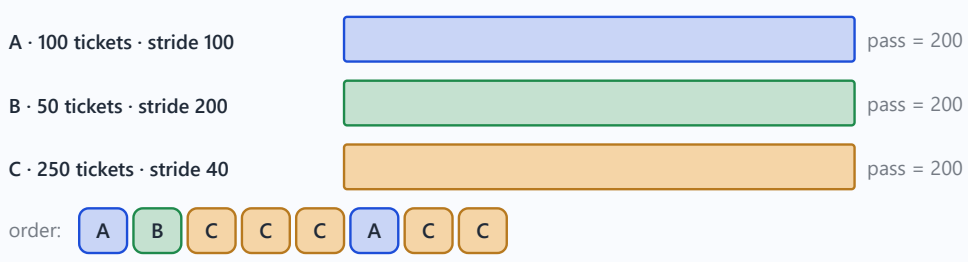


Figure 5.35: The end of 8 slices in stride scheduling. The order is A, B, C, C, C, A, C, C; the pass values of all

three processes have become equal to 200.

In 8 slices A ran 2 times, B 1, and C 5. The ratio is exactly 2 : 1 : 5, which is the ticket ratio 100 : 50 : 250 itself. At the end of the eighth slice all three pass values are 200: one full cycle has been completed and the same pattern starts again. In lottery the same ratio holds only in the long run and only approximately.

One disadvantage of stride is global state. When a new process enters the system, what should its pass value be? If it is given zero, the other processes' pass values are in the hundreds, so the new process would monopolize the CPU for a long time. The new process is therefore started with the smallest pass value at that moment. Lottery has no such problem; a new process just brings its tickets and gets a fair chance at once. Linux CFS's virtual-runtime idea is very close to stride: `vruntime` is a pass counter, and the inverse of the weight is the stride.

Example 5.15

A has 50, B has 100, C has 25 tickets; the strides are A 200, B 100, C 400. What is the order of the first 7 slices?

1. Slice 1: pass (0, 0, 0) → A; A becomes 200. Slice 2: (200, 0, 0) → B; B becomes 100. Slice 3: (200, 100, 0) → C; C becomes 400.
2. Slice 4: (200, 100, 400) → B; B becomes 200. Slice 5: (200, 200, 400) → tie, A; A becomes 400.
3. Slice 6: (400, 200, 400) → B; B becomes 300. Slice 7: (400, 300, 400) → B; B becomes 400.

The order is A, B, C, B, A, B, B. A got 2 slices, B 4, C 1: the ratio 2 : 4 : 1 = 50 : 100 : 25. At the end all three pass values are 400.

5.6 Multiprocessor scheduling

Today even a phone has eight cores. With several CPUs the scheduler must answer two questions: which job should run, and on which CPU should it run? The second question is harder than it looks, because of caches and the memory layout.

5.6.1 Thread scheduling: PCS and SCS

The kernel schedules not processes but kernel threads. User threads are managed by a library or the language runtime. This two-level structure gives rise to two separate contention scopes.

Definition 5.19

In **process-contention scope** (PCS) the thread library makes the user threads of the same process compete on lightweight processes (LWPs) within the process. This scope appears in the N:1 and M:N models. In **system-contention scope** (SCS) the kernel chooses among all the threads in the system (Figure 5.36).

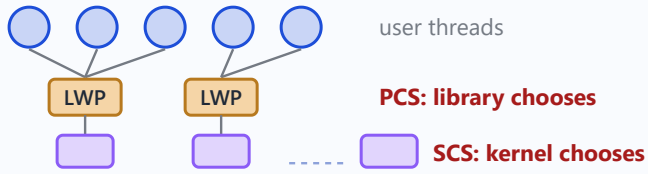


Figure 5.36: PCS and SCS. The library maps user threads to LWPs (PCS); the kernel maps kernel threads to CPUs (SCS).

Pthreads lets a program query and set the scope through an attribute:

```
pthread_attr_t attr;
int scope;
pthread_attr_init(&attr);
pthread_attr_getscope(&attr, &scope);
if (scope == PTHREAD_SCOPE_PROCESS)
  printf("PCS\n");
/* Linux, macOS: SYSTEM only */
pthread_attr_setscope(&attr,
                      PTHREAD_SCOPE_SYSTEM);
pthread_create(&tid, &attr, runner, NULL);
```

The program creates an attribute object and reads the default scope. If the scope is `PTHREAD_SCOPE_PROCESS`, it prints `PCS`. It then sets the scope to system and creates the thread with this attribute. Linux, Windows, and macOS use the 1:1 model, so they support only SCS. On Linux the default scope is already `PTHREAD_SCOPE_SYSTEM` and the program prints nothing; asking for PCS fails with the `ENOTSUP` error.

Although the PCS idea was abandoned in operating systems, it has returned in language runtimes. Java 21's virtual threads and Go's goroutines are mapped by the runtime's own scheduler onto a small number of carrier threads. The kernel sees only the carriers; the runtime decides which virtual thread runs on which carrier.

5.6.2 Common queue and per-core queues

Multiprocessor scheduling has two basic organizations. In **asymmetric multiprocessing** (AMP) a single master processor makes all the scheduling decisions and handles the I/O; the others run only user code. It is simple, because only one processor touches the kernel data structures, but the master becomes a bottleneck. In **symmetric multiprocessing** (SMP) each processor makes its own scheduling decisions. Most modern general-purpose systems are SMP; AMP-like organizations are still used in embedded and heterogeneous systems (for example on chips where a master processor hands out work to auxiliary cores) (Figure 5.37).

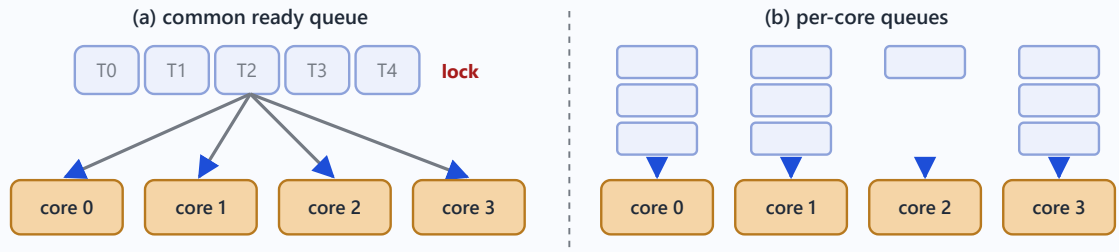


Figure 5.37: (a) A single common ready queue protected by a lock: all four cores take their work from the same queue. (b) Each core has its own ready queue.

In SMP there are two ways to hold the ready processes. In a **common queue** all the cores take work from the same queue. The load balances itself, because each core that becomes free takes the next job in the queue. But every access to the queue needs a lock, and lock contention grows with the number of cores. Also, a job may run on a different core each time; the data in its cache is useless.

With **per-core queues** each core has its own ready queue. This is also called two-level scheduling: a job is first assigned to a core and then runs from that core's queue. A lock is almost never needed, and because the job stays on the same core its cache stays warm. Linux, Windows, and FreeBSD use this organization. Its price is that the queues may become unbalanced over time: one core may have five jobs waiting while another sits idle. Load balancing is therefore needed.

5.6.3 Multicore and hardware threads

In modern processors a core may spend most of its time waiting for memory. On a cache miss, hundreds of cycles pass until the data arrives from main memory. Silberschatz notes that this **memory stall** can take up to 50% of core time (Figure 5.38).

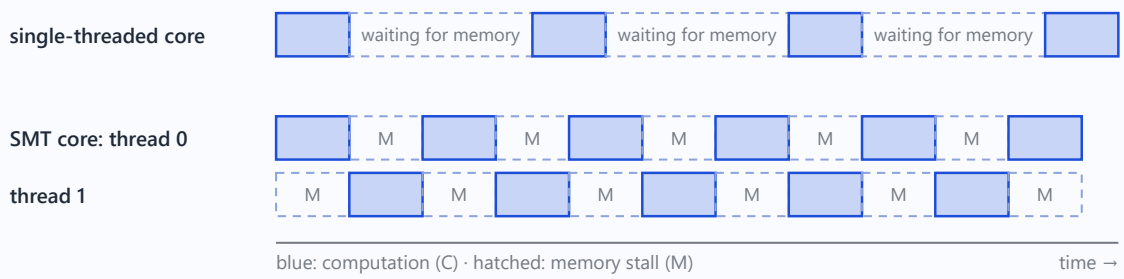


Figure 5.38: Memory stall. A single-threaded core waits for a significant part of its time (C: computation, M: memory stall). In SMT two hardware threads fill each other's waits.

Simultaneous multithreading (SMT; Hyper-Threading at Intel) fills this gap. A core has two hardware threads; each has its own set of registers, while the execution units are shared. While one waits for memory, the other executes instructions. The operating system sees each hardware thread as a separate **logical CPU**; a processor with four cores and SMT shows eight logical CPUs.

This gives rise to two-level scheduling. The operating system chooses which software thread runs on which logical CPU. The core's hardware chooses which hardware thread executes at a given moment. The operating system must know about this distinction. Sibling logical CPUs share execution units. So when an idle physical core exists, putting two busy jobs on the two logical CPUs of the same core is a bad decision; both slow down.

5.6.4 Load balancing

Definition 5.20

Load balancing is the task of distributing jobs evenly among cores. In **push migration** a periodic checker compares the loads and pushes jobs from an overloaded core to less loaded ones. In **pull migration** an idle core pulls a job from the queue of a loaded core (Figure 5.39).

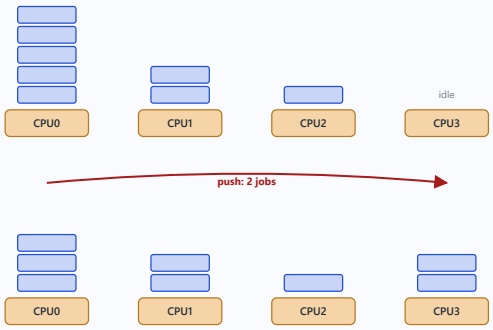


Figure 5.39: Left, the initial state: CPU0's queue has five jobs and CPU3 is idle. Right, push migration: the periodic balancer pushes two jobs from CPU0 to CPU3.

With per-core queues the imbalance does not fix itself; someone has to intervene. The two methods complement each other, and Linux uses both. A periodic balancer runs every few milliseconds and removes large differences. When a core's queue empties, pull migration is done without waiting: the idle core steals a job from the busiest queue (Figure 5.40).

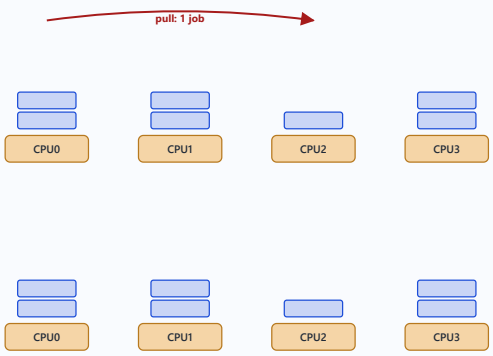


Figure 5.40: Left, pull migration: when the only job on CPU2 finishes, the idle CPU2 steals a job from the busiest queue. Right, the result: the queues are balanced.

Even when balance is achieved, it has a cost. The migrated job's cache is cold on the new core, and its first accesses start with misses. Linux therefore balances in layers. These layers are called scheduling domains. It balances first among the SMT siblings of the same physical core, then among cores that share the last-level cache. Balancing across NUMA nodes is done less often and with a larger imbalance threshold, because migration is most expensive there.

5.6.5 Code trace: `balance()`

The essence of pull balancing is a short function. `q[c].n` is the queue length of core number c; `migrate(src, dst)` moves the job at the tail of the source queue to the destination queue.

```
void balance(int me) {    /* idle CPU */
  int max = me;
  for (int c = 0; c < NCPU; c++)
    if (q[c].n > q[max].n) /* busier */
      max = c;
  int t = (q[max].n - q[me].n) / 2;
  for (int k = 0; k < t; k++)
    migrate(&q[max], &q[me]); /* from tail */
}
```

The function is called with the number of the core whose queue has emptied (`me`). The first loop finds the busiest queue: `max` starts as `me` and is updated whenever a longer queue is seen. Then the number of jobs to migrate is computed as half the difference. Had the whole difference been moved, the imbalance would merely have changed direction; moving half brings the two queues closer. The last loop takes that many jobs from the tail of the source (Table 5.18; Figure 5.41).

Step	Event	`max`	Queue lengths
initially	`max = me = 1`	1	3, 0, 5, 2
c = 0	3 > 0, busier	0	3, 0, 5, 2
c = 1	0 > 3 is false	0	3, 0, 5, 2
c = 2	5 > 3, busier	2	3, 0, 5, 2
c = 3	2 > 5 is false	2	3, 0, 5, 2
`t`	(5 − 0) / 2 = 2 (integer division)	2	3, 0, 5, 2
k = 0	H migrates from CPU2 to CPU1	2	3, 1, 4, 2
k = 1	G migrates from CPU2 to CPU1	2	3, 2, 3, 2

Table 5.18: `balance(1)`; initially the queues are CPU0 3 (A, B, C), CPU1 0, CPU2 5 (D–H), CPU3 2 (I, J).

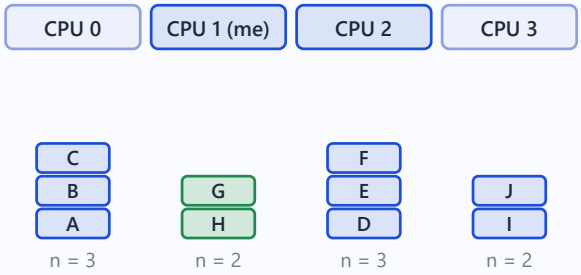


Figure 5.41: The queues after `balance(1)`: H and G have migrated to CPU1; the lengths are 3, 2, 3, 2.

There is a reason jobs are taken from the tail of the queue. The ones at the tail are not those that have waited longest but those that arrived last; they have loaded the least data into the cache, that is, their caches are the coldest. Moving them minimizes the loss of affinity. The function's time is O(NCPU + t). Real kernels look not only at queue length but also at the jobs' weights and their recent CPU usage.

5.6.6 Affinity, NUMA, and heterogeneous cores

Definition 5.21

Processor affinity is the tendency to keep a thread on the CPU whose cache and TLB are warm. With **soft affinity** the scheduler tries to keep the job on the same CPU but does not guarantee it. With **hard affinity** the set of CPUs on which the thread may run is specified explicitly.

Linux offers hard affinity through the `sched_setaffinity()` system call; in the shell, the command `taskset -c 0-3 ./prog` restricts the program to CPUs 0–3. Databases and real-time applications use this often.

The affinity question grows larger in NUMA (non-uniform memory access) architectures. In multi-socket servers each processor has memory close to it. Access to local memory is fast; access to another node's memory over the interconnect is slow. A good scheduler runs a thread on the node where its memory is. Linux's automatic NUMA balancing samples page accesses and moves tasks closer to their data or pages closer to their tasks (Figure 5.42).

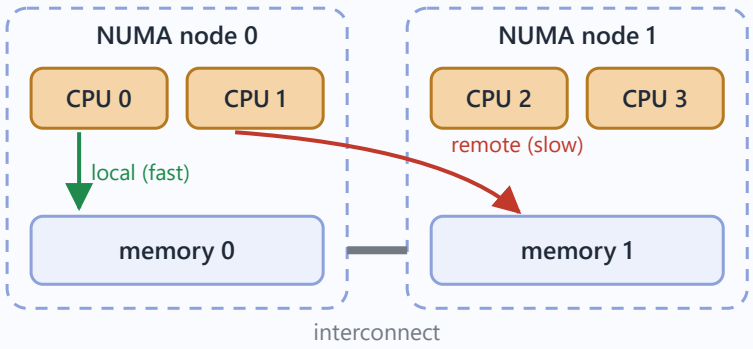


Figure 5.42: NUMA: each node's CPUs access their own local memory quickly and the other node's memory slowly, over the interconnect.

Load balancing and affinity conflict with each other. Migration achieves balance but breaks cache and memory locality. Affinity preserves locality, but some cores may sit idle while others are overloaded. Real schedulers strike a balance between the two with thresholds: if the imbalance is small, jobs stay where they are; if it is large, they are moved.

In recent years cores have stopped being uniform as well. **Heterogeneous multiprocessor** chips combine high-performance but power-hungry cores with slow but efficient ones. ARM's big.LITTLE and DynamIQ designs, Intel's P and E cores since the 12th generation, and the Apple M series are examples. Intel's Thread Director hardware gives the operating system hints about which kind of core suits which thread. Linux's Energy Aware Scheduling (EAS) puts each job on the core that meets the performance target and uses the least energy. Background jobs go to the efficient cores, and jobs the user is waiting for go to the powerful cores.

Common mistake

Believing that migration is free. Load balancing evens out the jobs, but a migrated job finds the cache and TLB cold on the new core; if it was moved between NUMA nodes, it also accesses its memory remotely. An idle core pulling a job (pull migration) also breaks affinity; under hard affinity, however, the job is never moved.

5.7 Real-time scheduling

In a general-purpose system a job finishing late is annoying but not wrong. In a real-time system, a correct result that arrives late counts as wrong. An airbag controller noticing a collision 40 ms late is the same as never noticing it. In these systems the goal of scheduling is not to improve the average but to guarantee every job's deadline.

5.7.1 Hard and soft real time, latency

Definition 5.22

In a **hard real-time** system, missing a deadline means the system has failed. In a **soft real-time** system, critical tasks have priority but there is no guarantee; missing a deadline only lowers the quality of service.

ABS braking systems, airbags, industrial robot control, and flight control are hard real time. Video streaming, voice over the internet (VoIP), games, and audio playback are soft real time: if a frame arrives late the picture stutters for a moment, but the system keeps working.

The quantity that measures a real-time system's responsiveness is **event latency**: the time from the occurrence of an event until the process that serves it starts running. This time is the sum of several components (Figure 5.43). **Interrupt latency** is the time from the arrival of the interrupt until the interrupt service routine (ISR) starts. The ISR then runs. Finally comes **dispatch latency**: the scheduler's stopping the running process and starting the real-time process.

Figure 5.43: Event latency: the sum of interrupt latency, the interrupt service routine (ISR), and dispatch latency, which consists of the conflict and dispatch phases.

In the conflict phase of dispatch latency two things are done. Any work running in the kernel is preempted, and a resource that a low-priority process holds and the real-time process needs is released. For this phase to be short, the kernel must be preemptive; if kernel code could not be preempted, a long system call would have to be waited out. This is why real-time systems require a preemptive, priority-based kernel. Even this alone is not enough for a hard guarantee. Before accepting a new task, the system must test whether all deadlines can still be met when the task is added. This test is called **admission control**.

Real-time algorithms can be static or dynamic. In static algorithms the priorities are computed once before the system runs; rate-monotonic scheduling is like this. In dynamic algorithms the decisions are made at run time; earliest deadline first is like this.

5.7.2 Periodic tasks and rate-monotonic scheduling

Most real-time tasks are periodic. A sensor is read every 10 ms; a control loop computes a new output every 50 ms.

Definition 5.23

A **periodic task** requires t of CPU time once every p units of time (the period) and must finish each job by its deadline d; $0 \le t \le d \le p$. The task's **utilization** is $U = t / p$. In these notes the deadline is the end of the period: $d = p$ (Figure 5.44).

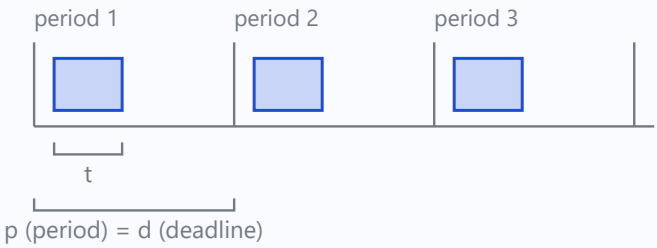


Figure 5.44: A periodic task: it requires t of CPU time every p and must finish by the deadline d (here, the end of the period).

Definition 5.24

In **rate-monotonic** (RM) **scheduling** each periodic task is given a static priority inversely proportional to its period: the shorter the period, the higher the priority. It is preemptive; when a new job of a higher-priority task arrives, the running task is preempted.

The intuition behind RM is simple: a task that must run more often is more urgent. RM is optimal among static-priority algorithms. If a task set can be scheduled by any static priority assignment, it can be scheduled by RM as well. A set that RM cannot schedule cannot be scheduled by any other static priority assignment.

Theorem 5.2 (Liu–Layland bound)

Let n independent periodic tasks have their deadlines at the end of their periods. If the total utilization satisfies

$$\Sigma U \le n\,(2^{1/n} - 1)$$

then RM meets all deadlines.

The bound decreases as n grows. Table 5.19 gives a few values; as n goes to infinity the bound approaches ln 2 ≈ 0.693. So whatever the number of tasks, if the total utilization does not exceed 69.3%, RM's success is guaranteed.

n	1	2	3	4	5	10	∞
bound	1	0,8284	0,7798	0,7568	0,7435	0,7177	ln 2 ≈ 0.6931

Table 5.19: The Liu–Layland bound $n(2^{1/n} - 1)$.

The direction of the theorem matters: the condition is **sufficient, not necessary**. If the utilization is below the bound, RM certainly succeeds. If it is above the bound, RM may or may not succeed; the test gives no conclusion. To decide, the schedule must be drawn over one hyperperiod (the least common multiple of the periods). If the total utilization exceeds 1, no algorithm can schedule the set: more than one second of work per second is being demanded of the CPU.

Example 5.16

P1 (p = 50, t = 20) and P2 (p = 100, t = 35). Can they be scheduled with RM?

1. **Test.** U = 20/50 + 35/100 = 0.40 + 0.35 = 0.75. For n = 2 the bound is 0.828; since 0.75 ≤ 0.828, RM meets all deadlines.
2. **Schedule.** P1 has the short period and the high priority. P1 runs 0–20, P2 20–50 (30 ms). At t = 50 a new job of P1 arrives and preempts P2: P1 50–70. P2 finishes its remaining 5 ms at 70–75; its deadline is 100. From 75 to 100 the CPU is idle.
3. **The second hyperperiod.** From 100 to 200 the same pattern repeats. No deadline is missed (Figure 5.45).

Figure 5.45: Rate-monotonic scheduling: P1 (p = 50, t = 20) runs first every 50 ms, P2 (p = 100, t = 35) uses the remaining gaps; all deadlines are met.

5.7.3 RM and EDF on the same set

Definition 5.25

In **earliest-deadline-first** (EDF) scheduling the priority is dynamic: among the ready jobs, the one with the nearest deadline runs. A job's priority is redetermined in every new period, according to its new deadline.

EDF's schedulability condition is much more generous: for preemptive, independent tasks whose deadlines are at the end of their periods, $\Sigma U \le 1$ is both sufficient and necessary. EDF can in theory use the CPU up to 100%. Let us see the difference between the two algorithms on a set that RM fails to pass the test on.

Example 5.17

P1 ($p = 50$, $t = 25$) and P2 ($p = 80$, $t = 35$). $U = 25/50 + 35/80 = 0.5 + 0.4375 = 0.9375 > 0.828$: there is no guarantee for RM; but since $U \le 1$, EDF suffices.

1. **RM.** P1 (short period) runs first at 0–25 and P2 at 25–50; P2 has completed 25 ms and has 10 ms left. At t = 50 a new job of P1 arrives and, having the higher static priority, preempts P2; P1 runs 50–75. P2 runs 5 ms more at 75–80, but its time runs out at t = 80: P2 misses its deadline with 5 ms short.
2. **EDF, t = 50.** The new job of P1 has deadline 100 and that of P2 has deadline 80. P2, with the nearer deadline, continues and finishes at 60. P1 runs 60–85.
3. **EDF, t = 80.** A new job of P2 arrives; its deadline is 160. Since P1's is 100, P1 continues until 85. P2 runs 85–100. At t = 100 the deadline of P1's new job is 150 < 160 and P1 goes ahead: P1 100–125, P2 finishes at 125–145. From 145 to 150 the CPU is idle.
4. **Result.** Over the 400 ms hyperperiod (the least common multiple of 50 and 80), EDF misses no deadline (Figure 5.46).

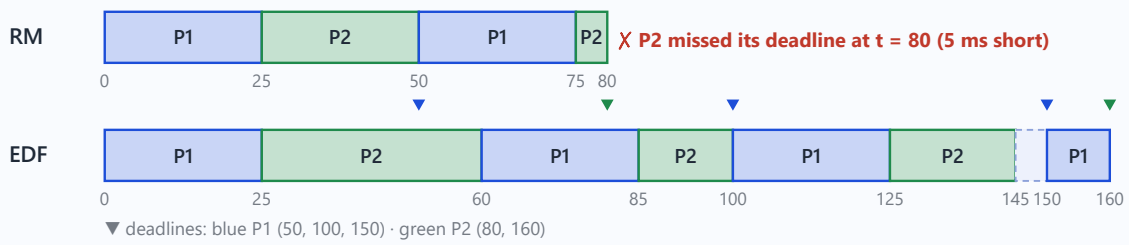


Figure 5.46: P1 (50, 25) and P2 (80, 35). Top, RM: P2 misses its deadline at t = 80 by 5 ms. Bottom, EDF: P1 0–25, P2 25–60, P1 60–85, P2 85–100, P1 100–125, P2 125–145, P1 150–; all deadlines are met.

EDF's superiority has a price. Because priorities are dynamic, deadlines have to be compared at every decision. More importantly, there is its behavior under overload. If total utilization temporarily exceeds 1, in RM only the low-priority tasks miss, and which ones miss is predictable. In EDF the misses can spread like a chain, and it is hard to say in advance which task will be affected.

An idea close to EDF is **proportional-share** real-time scheduling. A total of T shares is distributed among the applications; an application receiving N shares gets N/T of the CPU. Admission control rejects a request that would exceed the total. Linux's `SCHED_DEADLINE` policy (since version 3.14) combines EDF with constant-bandwidth-server (CBS) protection. Each task declares a runtime, a deadline, and a period. If a task tries to run longer than it declared, it is stopped until the next period; so a misbehaving task cannot break the guarantees of the others.

Example 5.18

T1 (runtime 2, period 5) and T2 (runtime 4, period 7). What is the total utilization? Are the first deadlines met under RM and under EDF?

1. $U = 2/5 + 4/7 = 34/35 \approx 0.97$. The RM bound of 0.828 is exceeded; for EDF, $U \le 1$ suffices.
2. **RM.** T1 runs 0–2, T2 2–5 (3 units). At t = 5 the new job of T1 preempts T2 by its static priority: T1 5–7. T2's deadline is 7 and it is 1 unit short; the deadline is missed.
3. **EDF.** At t = 5 the new deadline of T1 is 10 and that of T2 is 7; T2 continues and finishes at 6. Over the 35-unit hyperperiod EDF misses no deadline.

5.7.4 Code trace: the Liu–Layland test

RM's sufficiency test is a short function. The function returns three values: 1 (RM certainly suffices), 0 (the test cannot decide), and −1 (since utilization exceeds 1, no algorithm can schedule the set).

```
int rm_test(Task g[], int n) {
  double U = 0;
  for (int i = 0; i < n; i++)
    U += (double) g[i].t / g[i].p;
  double bound = n * (pow(2, 1.0/n) - 1);
  if (U <= bound) return 1; /* enough */
  if (U <= 1.0) return 0;   /* unknown */
  return -1;            /* impossible */
}
```

The loop adds up each task's utilization; the `(double)` cast prevents integer division. Then the bound is computed. The `1.0` in `pow(2, 1.0/n)` is important: had `1/n` been written, integer division would give 0 for $n \ge 2$ and the bound would come out as 0. The last two comparisons separate the three outcomes (Table 5.20).

Set	Step	Added	U	bound	Turnaround
T1 (20, 5), T2 (50, 10), T3 (100, 30)	i = 0	5 / 20 = 0,25	0,25		
	i = 1	10 / 50 = 0,2	0,45		
	i = 2	30 / 100 = 0,3	0,75		
	bound		0,75	$3(2^{1/3} - 1) = 0{,}7798$	1
P1 (50, 25), P2 (80, 35)	i = 0	25 / 50 = 0,5	0,5		
	i = 1	35 / 80 = 0,4375	0,9375		
	bound		0,9375	0,8284	0

Table 5.20: `rm_test()` on two task sets.

In the first set, 0.75 ≤ 0.7798, so the function returns 1 and RM meets all deadlines. The second set is the one in Example 5.17. The utilization exceeds the bound but is below 1; the function returns 0. In this set RM really does miss a deadline, while EDF meets them. A result of 0 does not mean "RM fails": the set P1 (p = 50, t = 25) and P2 (p = 100, t = 50) has a utilization of 0.5 + 0.5 = 1 > 0.828 and the test gives 0, but because the periods are multiples of one another, RM meets all the deadlines (P1 0–25, P2 25–50, P1 50–75, P2 75–100). Another set that slightly exceeds the bound may also be schedulable with RM; only the schedule itself shows this. The function's time is O(n).

5.7.5 Code trace: the EDF scheduler

EDF's decision rule can be applied at every time step. The following program schedules three periodic tasks for 12 ms, making a decision every millisecond.

```
/* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
for (int t = 0; t < 12; t++) {
    for (int i = 0; i < n; i++)
        if (t % P[i] == 0) {      /* new job */
            left[i] = C[i];
            due[i] = t + P[i];
        }
    int s = -1;
    for (int i = 0; i < n; i++)
        if (left[i] > 0 &&
            (s < 0 || due[i] < due[s]))
            s = i;
    if (s >= 0) left[s]--;     /* 1 ms */
}
```

`P[i]` is the period, `C[i]` the run time, `left[i]` the remaining time of the current job, and `due[i]` the deadline of the current job. At the start of every millisecond the jobs of tasks whose period begins are renewed: the remaining time is set to `C[i]` and the deadline to `t + P[i]`. The second loop picks, among the tasks that still have work, the one with the smallest deadline. Because the

comparison is strictly less than (`<`), in a tie the first task in the array wins. The chosen task's remaining time drops by 1. If there is no work at all, `s` stays −1 and the CPU is idle for that millisecond.

The tasks are A (P = 3, C = 1), B (P = 4, C = 1), and C (P = 6, C = 2). The total utilization is 1/3 + 1/4 + 2/6 = 11/12 ≈ 0.917 ≤ 1. Since the least common multiple of the periods is 12, a 12 ms window shows the whole pattern (Table 5.21; Figure 5.47).

`t`	New job	Ready (deadline)	Running
0	A, B, C	A 3, B 4, C 6	A
1	—	B 4, C 6	B
2	—	C 6	C
3	A	A 6, C 6	A (first in a tie)
4	B	B 8, C 6	C
5	—	B 8	B
6	A, C	A 9, C 12	A
7	—	C 12	C
8	B	B 12, C 12	B (first in a tie)
9	A	A 12, C 12	A (first in a tie)
10	—	C 12	C
11	—	none	empty

Table 5.21: EDF code trace. The "Ready" column shows the tasks that still have work, with their deadlines.

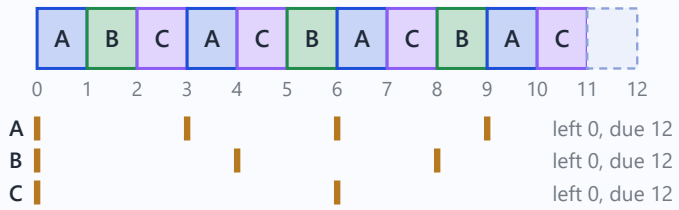


Figure 5.47: EDF Gantt chart: A, B, C, A, C, B, A, C, B, A, C, idle. In each millisecond the ready job with the nearest deadline runs.

In 12 ms, 11 ms of work is done and 1 ms of slack remains; that is 1 − 11/12. No job misses its deadline. The moment t = 4 separates EDF from RM: B's period is shorter than C's, and under RM B would run. EDF instead looks at the deadlines: C's job must finish by 6 and B's by 8; C is chosen. This set is also scheduled without misses under RM, but because its utilization (0.917) exceeds the three-task Liu–Layland bound (0.780), only the schedule itself can show RM's success.

The program's time is O(n) per step. Real EDF implementations do not decide every millisecond; they decide only when a job arrives or finishes, and keep the ready jobs in a priority queue ordered by deadline. Each decision then takes O(log n).

5.7.6 POSIX real-time scheduling

POSIX's real-time extension (POSIX.1b) makes it possible to assign scheduling policies to threads. Table 5.22 summarizes the policies available on Linux.

Policy	Behavior
`SCHED_FIFO`	Fixed priority (1–99), no time slice; the thread runs until it blocks or yields the CPU.
`SCHED_RR`	Same as `SCHED_FIFO`, but a time slice is applied among equal priorities.
`SCHED_OTHER`	The default time-sharing policy (EEVDF on Linux); also `SCHED_BATCH` and `SCHED_IDLE`.
`SCHED_DEADLINE`	Linux only: EDF with runtime, deadline, and period.

Table 5.22: POSIX and Linux scheduling policies.

```
pthread_attr_t attr;
struct sched_param sp;
int policy;
pthread_attr_init(&attr);
pthread_attr_getschedpolicy(&attr, &policy);
/* policy is ignored unless inheritance is off */
pthread_attr_setinheritsched(&attr,
                     PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
sp.sched_priority = 50;          /* 1..99 */
pthread_attr_setschedparam(&attr, &sp);
pthread_create(&tid, &attr, runner, NULL);
```

The program first prepares the attribute object and reads the default policy. Then comes an important line: inheritance is turned off with `pthread_attr_setinheritsched`. By default a new thread inherits its policy from the thread that creates it; if inheritance is not turned off, the policy set in the attribute is silently ignored. Then the policy is set to `SCHED_FIFO`, the priority to 50, and the thread is created.

Real-time policies require privilege: on Linux the `CAP_SYS_NICE` capability is needed; in an unprivileged process `pthread_create` returns `EPERM`. In the shell the same thing is done with the command `chrt -f 50 ./prog`. A real-time thread that enters an infinite loop can lock up the system. As a safeguard Linux by default gives real-time jobs at most 0.95 seconds of every second; the remaining 0.05 seconds is left to normal processes. This limit can be changed with the `sched_rt_runtime_us` setting. One detail in the function names is also often misspelled: the names are `pthread_attr_getschedpolicy` and `pthread_attr_setschedpolicy`; there is no underscore between "sched" and "policy."

5.8 Operating-system examples

Real schedulers are combinations of the ideas in this chapter. Linux uses priority queues for real-time jobs and proportional share for normal jobs. Windows works with dynamic priorities that resemble an MLFQ.

5.8.1 Linux scheduling classes

The Linux scheduler is modular: each task belongs to a **scheduling class**, and the classes are consulted in a fixed priority order. The scheduler first asks the top class "do you have a task to run?"; if not, it moves to the one below. While the upper class has a ready task, those below never run (Table 5.23).

Class	Policies	Selection
`stop`	— (internal to the kernel)	CPU stop, task migration; preempts everything
`deadline`	`SCHED_DEADLINE`	EDF + CBS
`rt`	`SCHED_FIFO`, `SCHED_RR`	array of 100 priority levels and a bitmap, O(1) selection
`fair`	`SCHED_NORMAL`, `SCHED_BATCH`, `SCHED_IDLE`	CFS (2.6.23–6.5), EEVDF (6.6 and later)
`ext`	`SCHED_EXT`	Scheduler written in BPF (sched_ext, 6.12 and later)
`idle`	—	idle loop when there is no work

Table 5.23: Linux scheduling classes (in priority order).

This order directly determines which of three ready tasks on the same CPU runs first. A `SCHED_DEADLINE` task comes before a `SCHED_FIFO` task at priority 50. That in turn comes before a normal task with nice value −20. The nice value affects only the share within the fair class; it cannot put a normal task ahead of a real-time task (Figure 5.48).

Figure 5.48: Linux kernel priorities: 0–99 real-time tasks, 100–139 normal tasks (nice −20 to +19). A small number means high priority.

Inside the kernel a single priority scale is used, and a small number means high priority. The range 0–99 belongs to real-time tasks; the `rt_priority` (1–99) given by the user is converted to the kernel priority as 99 − `rt_priority`. The range 100–139 belongs to normal tasks: nice −20 → 100, nice 0 → 120, nice 19 → 139. The default slice of the `SCHED_RR` policy is 100 ms.

5.8.2 CFS: virtual runtime

From Linux 2.6.23 to 6.5, normal tasks were scheduled by the Completely Fair Scheduler (CFS). CFS's goal is to imitate an ideal multitasking processor. On the ideal processor, with n tasks, each would run at the same time at 1/n of the CPU's speed. Because a real CPU can run only one task at a time, CFS tracks how far each task has fallen behind this ideal and runs the one that has fallen furthest behind.

Definition 5.26

The **virtual runtime** (`vruntime`) is the CPU time a task has received, scaled by its weight. When a task runs for Δ,

> vruntime += Δ × 1024 / weight

CFS always runs the task with the smallest `vruntime` .

The weight comes from the nice value. The weight of nice 0 is 1024; each nice step changes the weight by about a factor of 1.25. The weight of nice 5 is 335 and that of nice −5 is 3121. A factor of about 1.25 corresponds to a difference of roughly 10% in CPU share between two CPU-bound tasks. For a nice 0 task, vruntime grows at the same rate as real time. A task with a smaller weight (larger nice) has its vruntime grow faster; so it is chosen less often.

Consider two CPU-bound tasks: A with nice 0 (weight 1024), B with nice 5 (weight 335). In the long run both vruntimes must advance at the same rate; this requires their shares to be proportional to their weights. A gets 1024 / (1024 + 335) = 1024 / 1359 ≈ 75% and B ≈ 25%; the ratio is about 3 : 1. When B runs 10 ms its vruntime grows by 10 × 1024 / 335 ≈ 30.6 ms.

I/O-bound tasks are rewarded automatically in this scheme. A sleeping task uses no CPU, so its vruntime does not grow; when it wakes up it has fallen behind the CPU-bound tasks and immediately moves to the front. It is also undesirable for a task that has slept for a long time to monopolize the CPU when it returns. CFS therefore pulls the woken task's vruntime to slightly behind the minimum vruntime; the advantage given to a task that has fallen behind is limited (Figure 5.49).

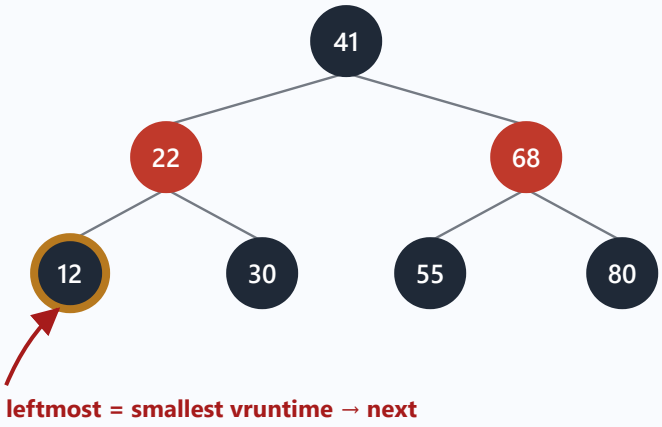


Figure 5.49: CFS keeps ready tasks in a red-black tree ordered by vruntime. The leftmost node is the task that has received the least CPU and is the next choice.

Ready tasks are kept in a red-black tree keyed by vruntime. Insertion and deletion take O(log n). The leftmost node, that is, the task with the smallest vruntime, is also cached in a pointer; so selection is O(1).

5.8.3 Code trace: `vruntime`

The following code shows CFS's selection rule as a simplified tick handler. At each tick the leftmost task is run for 3 ms (3000 µs), its vruntime is updated, and it is reinserted into the tree.

```
#define DELTA 3000  /* 3 ms, in µs */
/* nice 0 → 1024, nice 5 → 335 */
void tick(void) {
  Task *g = leftmost();   /* leftmost */
  run(g, DELTA);
  g->vruntime += DELTA * 1024 / g->weight;
  tree_insert(g);     /* re-sort */
}
```

`leftmost()` returns the task with the smallest vruntime; in a tie the task inserted into the tree first is on the left. `run` runs the task for 3 ms. The update line uses integer arithmetic: for A, 3000 × 1024 / 1024 = 3000; for B, 3,072,000 / 335 = 9170 (the remainder is discarded). The last line puts the task back into its place in the tree according to its new vruntime. Initially both tasks' vruntime is 0 (Table 5.24).

Tick	A.vruntime	B.vruntime	Chosen	Added
1	0	0	A (first inserted in a tie)	3000
2	3.000	0	B	9170
3	3.000	9.170	A	3000

Tick	A.vruntime	B.vruntime	Chosen	Added
4	6.000	9.170	A	3000
5	9.000	9.170	A	3000
6	12.000	9.170	B	9170
7	12.000	18.340	A	3000
8	15.000	18.340	A	3000
9	18.000	18.340	A	3000
end	21.000	18.340		

Table 5.24: Trace of `tick()` ; A is nice 0 (weight 1024), B is nice 5 (weight 335). The vruntime values are in µs.

In 9 ticks A ran 7 times and B 2 times. In the short run the ratio is 7 : 2; in the long run it approaches the weight ratio 1024 : 335, that is, ≈ 75% – 25%. When the same rule is run for 10,000 ticks, A gets 7535 and B 2465 ticks. CFS has no fixed time slice; the share comes directly from the weight. Real CFS measures time in nanoseconds and, instead of fixed ticks, divides a "scheduling latency" window among the tasks according to their weights; but the selection rule is the same. This rule is the same idea as the pass counter of stride scheduling: `vruntime` is the pass, and 1024 / weight is the stride.

5.8.4 EEVDF

CFS provided fair share well, but it needed many heuristic tunings for latency-sensitive tasks. In Linux 6.6 the scheduler for normal tasks was replaced by EEVDF. EEVDF (Earliest Eligible Virtual Deadline First) is an algorithm proposed by Stoica and Abdel-Wahab in 1995. It replaces the heuristics in CFS with two clear criteria: eligibility and virtual deadline.

Definition 5.27

In EEVDF, V is the (weighted) average of the tasks' virtual runtimes. A task's **lag** shows how far behind its share it is. A task is **eligible** if it is behind its share or exactly at it: lag ≥ 0, that is, v ≤ V. A task's **virtual deadline** is vd = v + slice / weight. Among the eligible tasks, the one with the earliest vd runs.

The eligibility condition governs fairness, and the deadline governs latency. A task that has received more than its share (v > V) is not eligible and waits; so no one can exceed its share. Among eligible tasks, a task that asks for a short slice has a nearer deadline and runs first. A task asking for a short slice runs more often but for shorter periods; its share does not change in the long run.

Example 5.19

Three tasks of equal weight: A (v = 12, slice 3), B (v = 12, slice 1), C (v = 15, slice 3). Since the weights are equal, vd = v + slice is used. Let us trace five selections.

1. V = (12 + 12 + 15) / 3 = 13. C is ahead (v = 15 > V): not eligible. A and B are eligible; vd is 15 for A and 13 for B → B is chosen.
2. B ran 1 ms: v = 13, new vd = 14. V = 13.33; A and B are eligible; the earliest vd is B's (14) → B.
3. B advanced to 14; V = 13.67. B and C are ahead of the average; only A is eligible → A runs its 3 ms slice.
4. A: v = 15, new vd = 18. V = 14.67; only B (v = 14) is eligible → B.
5. B: v = 15, vd = 16. V = 15; all three are eligible. vd is A 18, B 16, C 18 → B (Figure 5.50).

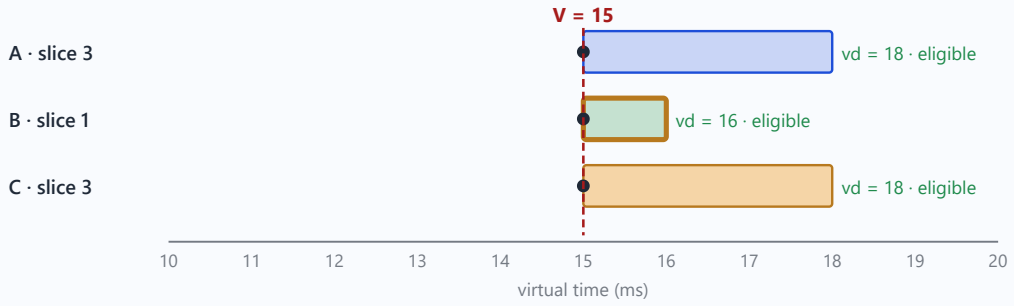


Figure 5.50: The fifth selection moment: V = 15 and all three tasks are eligible; B, with the earliest virtual deadline (vd = 16), is chosen.

In the example B, which asks for a short slice, got four of the five selections, but each time it ran only 1 ms. A ran once, for 3 ms. The eligibility condition limits B's running ahead: when B moved ahead of the average (the third step) it lost its eligibility and gave way to A. Since Linux 6.12 a latency-sensitive task can ask for a short slice for itself with the `sched_setattr()` call. The weights are computed from nice in the same way as in CFS.

5.8.5 Windows priorities

Windows uses a preemptive, priority-based scheduler. Of the ready threads, the one with the highest priority runs. It stays on the CPU until it blocks, its slice ends, or a higher-priority thread becomes ready. There are 32 priority levels, and a large number means high priority. Levels 1–15 are the variable class and 16–31 the real-time class. Level 0 is reserved for the system thread that zeroes free memory pages (the zero-page thread). Each level has its own queue; if there is no ready thread, the idle thread runs.

A thread's numeric priority comes from two pieces of information: its process's priority class and the thread's relative priority within that class (Table 5.25).

Relative priority	REALTIME	HIGH	ABOVE_NORMAL	NORMAL	BELOW_NORMAL	IDLE
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6

Relative priority	REALTIME	HIGH	ABOVE_NORMAL	NORMAL	BELOW_NORMAL	IDLE
above normal	25	14	11	9	7	5
normal	**24**	**13**	**10**	**8**	**6**	**4**
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Table 5.25: Windows priority classes (columns) and relative priorities (rows). The bold row is each class's base priority.

The priority of threads in the variable class changes over time; this is Windows's counterpart of MLFQ. When an I/O completes, the thread's priority is temporarily boosted. The boost is larger for interactive devices such as the keyboard and mouse than for disk I/O. Threads of the foreground window and threads woken by GUI events also get a boost. A boosted priority decreases by 1 each slice and returns to the base, and never exceeds 15. There is a safeguard against starvation as well: a thread that has been ready for about 4 seconds without running has its priority raised to 15 for one slice. The slice is about 2 clock ticks on client versions and 12 on server versions; on a server the long slice reduces context switches and increases throughput (Figure 5.51).

Figure 5.51: Windows dynamic priority boost: a thread with base priority 8 rises to 10 when an I/O completes and returns to the base, dropping 1 per slice.

Common mistake

Believing that a task with nice −20 on Linux will get ahead of a `SCHED_FIFO` task. nice changes only the weight within the fair class; while a ready task exists in the real-time class, no task in the fair class runs. On Windows, too, dynamic boosts are limited to the variable class: a thread cannot be boosted past 15 into the real-time class.

5.9 Evaluating algorithms

Among so many algorithms, which should be chosen for a particular system? First the criterion is set: for example, "the longest response time must not exceed 300 ms, and subject to that, throughput should be maximized." Then the algorithms are evaluated against this criterion. There are four ways of evaluating.

5.9.1 Four methods and Little's law

Deterministic modeling. For a particular workload given in advance, each algorithm's result is computed. All the Gantt charts in this chapter are deterministic modeling. The method is exact and simple, and the results are comparable numbers. Its weakness is that the result is valid only for that input: on another set the ranking may change.

Queueing models. In a real system the workload changes every day, but the distributions of the processes' arrival and service times can be measured. Queueing theory computes quantities such as average queue length, average waiting time, and utilization from these distributions. The most basic tool is Little's law.

Theorem 5.3 (Little's law)

In a system in steady state, the average number of processes n is the product of the average arrival rate λ and the average time W a process spends in the system:

$$n = \lambda \times W$$

In steady state the number of processes entering the system equals the number leaving. If λ processes arrive per second and each stays W seconds, there are on average λ × W processes inside at any moment. If 7 processes arrive per second and each stays 2 seconds on average, there are 14 processes in the system on average. The law is independent of the arrival and service distributions; that is why it is used in so wide an area. If any two of the quantities are known, the third can be found. If 6 requests per second arrive at a server and the system holds 18 requests on average, each request stays W = 18 / 6 = 3 seconds on average (Figure 5.52).

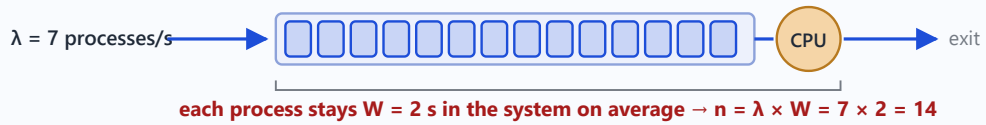


Figure 5.52: Little's law: λ = 7 processes arrive per second and each process stays W = 2 s in the system on average; the system holds n = 7 × 2 = 14 processes on average.

The limit of queueing models is the assumptions made so that they can be solved mathematically. Assumptions such as arrivals following a Poisson distribution do not hold exactly for real workloads; and the model of a complex algorithm often cannot be solved.

Simulation. A programmed model of the system is built: the clock, the queues, the scheduler. The model is run with randomly generated data or with trace data recorded from a real system. Because trace data preserves the order of real events, the results are close to reality. The price is time: a detailed simulation can take hours, and the recordings take up a lot of space.

Implementation. The most exact way is to put the algorithm into a real kernel and try it. This used to be expensive: the kernel had to be modified, compiled, and rebooted. Today Linux's `sched_ext` infrastructure allows loading schedulers written in BPF without recompiling the kernel; large companies compare different schedulers in production with A/B tests. Implementation has a subtle problem as well: users and programmers adapt to the algorithm. Users pressing Enter in CTSS were the first example of this; the measured behavior can change because of the algorithm.

5.9.2 Comparison on the common set

The results of all the algorithms examined on the common set are brought together in Table 5.26 (Figure 5.53).

Algorithm	Waiting	Turnaround	Response	Switches
FCFS	8,75	15,25	8,75	3
SJF	7,75	14,25	7,75	3
SRTF	6,5	13	4,25	4
Priority (P)	10,75	17,25	8,5	4
RR q = 2	12,75	19,25	2	12
RR q = 5	11	17,5	5,5	5

Table 5.26: Averages (ms) and numbers of context switches of the algorithms on the common set. P: preemptive.

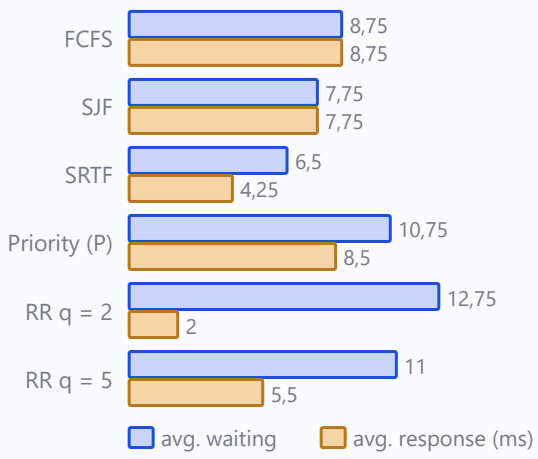


Figure 5.53: Average waiting and average response times of the algorithms on the common set.

SRTF is best in average waiting (6.5), but it requires the lengths to be known. RR with q = 2 is best in response (2), but it makes the most switches and is worst in waiting. Preemptive priority optimizes importance, not waiting time; its average is worse than FCFS, but the most important process, P3,

never waits. There is no single "best" algorithm. The choice depends on the goal: response, throughput, fairness, or deadlines. This is why real systems use hybrid structures such as MLFQ or EEVDF: while putting short jobs first, they try not to starve anyone and to keep response bounded.

5.9.3 Three exercises on turnaround, response, and fairness

Example 5.20

A, B, and C all arrive at t = 0 and each needs 5 ms of CPU. What are the average turnaround and average response times under SJF and under RR (q = 1)?

1. **SJF.** Since the lengths are equal, SJF is the same as FCFS here. Completion times are 5, 10, 15; the average turnaround is 10 ms. Response is (0 + 5 + 10) / 3 = 5 ms.
2. **RR (q = 1).** A, B, and C swap places every millisecond; the last round is at 12–15 and the completion times are 13, 14, 15. The average turnaround is 14 ms and the response is (0 + 1 + 2) / 3 = 1 ms.

Example 5.21

A (arrival 0, length 100), B (10, 10), C (10, 10). What are the average turnaround and response times under nonpreemptive SJF and under STCF (SRTF)?

1. **SJF.** At t = 0 only A is present and it cannot be preempted: A 0–100, B 100–110, C 110–120. Turnaround times are 100, 100, 110; the average is 310 / 3 ≈ 103.3 ms. Response times are 0, 90, 100; the average is ≈ 63.3 ms.
2. **STCF.** At t = 10 B arrives; its length is 10 and A's remainder is 90, so A is preempted. B 10–20, C 20–30, A 30–120. Turnaround times are 120, 10, 20; the average is 150 / 3 = 50 ms. Response times are 0, 0, 10; the average is ≈ 3.3 ms.

Preemption removed the short jobs' waiting behind the long job, that is, the convoy effect.

Example 5.22

Let the unfairness measure U be the ratio of the completion time of the job that finishes first to that of the job that finishes second. Two jobs with equal tickets each want L slices. What is U under RR? How does U change as L grows under lottery scheduling?

1. **RR (and stride).** The jobs run alternately; one finishes at 2L − 1 and the other at 2L. U = (2L − 1) / 2L; for L = 10 it is 19 / 20 = 0.95. As the job gets longer U approaches 1.
2. **Lottery.** In each slice the two jobs win with equal probability; when one reaches L wins it finishes, and the other runs its remaining slices alone and finishes at 2L. If the expected value of the first job's completion time is computed, the expected U comes out as ≈ 0.82

for L = 10 and ≈ 0.94 for L = 100.

Lottery creates a marked unfairness for short jobs; as the job gets longer, the effect of chance diminishes and U approaches 1. Deterministic stride does not have this problem.

5.10 Summary

- The **short-term scheduler** selects a process from the ready queue when the CPU becomes free; the **dispatcher** switches context, switches to user mode, and jumps into the program. The time in between is the dispatch latency.
- Processes alternate between **CPU and I/O bursts**. Most bursts are short; not keeping I/O-bound processes waiting keeps the devices busy. Treating each burst as a separate job overlaps I/O with computation.
- A decision is made in four situations: running → waiting, running → ready, waiting → ready, and termination. A **nonpreemptive** scheduler decides only in situations 1 and 4; a **preemptive** one decides in all four.
- Criteria: CPU utilization and throughput should be large; turnaround (C − A), waiting (turnaround − CPU), and response (F − A) times should be small. The scheduler affects only the waiting time.
- **FCFS** is simple and starvation-free, but it is sensitive to arrival order and causes the convoy effect. **SJF** minimizes the average waiting time when all jobs are ready at the same time; the next burst is predicted with the **exponential average**: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$. **SRTF** is the preemptive version and gives the lowest waiting time (6.5 ms) on the common set.
- In **priority** scheduling, starvation is solved with **aging** and priority inversion with priority inheritance.
- **RR** gives each process at most q of time; when n ready processes is constant and the switching cost is neglected, no process waits longer than $(n - 1) \cdot q$. As q shrinks, response improves and switches increase; if slices are fully used, the overhead fraction is $s / (q + s)$. When q exceeds the longest burst, RR becomes FCFS.
- **MLQ** divides processes into permanent classes. **MLFQ** moves them according to behavior: a new job starts at the top, a job that uses up its allotment at a level moves down, and at regular intervals everyone goes back to the top.
- Proportional share: **guaranteed** and **fair-share** scheduling (Linux cgroup `cpu.weight`), probabilistic **lottery**, and deterministic **stride** (stride = constant / tickets, the smallest pass runs).
- On multiprocessors most modern general-purpose systems use SMP and **per-core queues**; imbalance is corrected by push and pull migration, and migration in turn breaks **affinity**. SMT cores, NUMA, and heterogeneous cores complicate the decision.
- In real time, **RM** has static priorities; $\Sigma U \le n(2^{1/n} - 1)$ is a sufficient condition. **EDF** is dynamic; $\Sigma U \le 1$ is both sufficient and necessary. Linux `SCHED_DEADLINE` combines EDF with CBS.

- The Linux classes are consulted in order: stop, deadline, rt, fair, ext, idle. **CFS** chooses the smallest `vruntime` (vruntime += Δ × 1024 / weight); **EEVDF** chooses the earliest virtual deadline among the eligible tasks. Windows uses 32 priority levels and dynamic boosts.
- Evaluation methods: deterministic modeling, queueing models (**Little's law** n = λ × W), simulation, and implementation.

5.11 Key terms

scheduler ready queue CPU burst preemptive / nonpreemptive scheduling dispatcher context switch throughput turnaround time response time convoy effect shortest-job-first exponential average starvation aging priority inversion time slice (quantum) multilevel feedback queue lottery scheduling load balancing processor affinity rate-monotonic scheduling earliest deadline first

---

5.12 Test Yourself

1. P1 (arrival 0, CPU 7), P2 (1, 3), P3 (3, 2), P4 (6, 1). What are the average waiting times under nonpreemptive SJF and under SRTF?

   Answer

   SJF: P1 runs 0–7 uninterrupted; at t = 7 the ready processes are P2 (3), P3 (2), P4 (1); the order is P4 7–8, P3 8–10, P2 10–13. Waiting times are 0, 9, 5, 1; the average is 15 / 4 = 3.75 ms. SRTF: at t = 1, P2 (3) < P1's remainder of 6, so P1 is preempted; P2 1–4, P3 4–6, P4 6–7, P1 7–13. Waiting times are 6, 0, 1, 0; the average is 7 / 4 = 1.75 ms.

2. A (arrival 0, CPU 5), B (1, 4), C (2, 3); RR, q = 2. A newly arrived process enters the queue before the process whose slice has ended. What are the Gantt chart and the average waiting time?

   Answer

   A 0–2, B 2–4, C 4–6, A 6–8, B 8–10 (finishes), C 10–11 (finishes), A 11–12 (finishes). Completion times are A 12, B 10, C 11. Waiting times are A 12 − 0 − 5 = 7, B 10 − 1 − 4 = 5, C 11 − 2 − 3 = 6; the average is 6 ms. The average response is (0 + 1 + 2) / 3 = 1 ms.

3. α = 0.25 and τ_0 = 10 ms. The measured bursts are 4, 4, and 16 ms. What is τ_3? What is the effect of

choosing α = 0.25 on this sequence?

Answer

$\tau_1 = 0.25 \cdot 4 + 0.75 \cdot 10 = 8.5$; $\tau_2 = 0.25 \cdot 4 + 0.75 \cdot 8.5 = 7.375$; $\tau_3 = 0.25 \cdot 16 + 0.75 \cdot 7.375 \approx 9.53$ ms. A small α gives a lot of weight to the past: after two short bursts the prediction is still above 7, and it catches up slowly with the burst that jumps to 16.

4. MLFQ: Q0 (q = 3), Q1 (q = 6), Q2 (FCFS). A (arrival 0, CPU 12), B (2, 2), C (5, 6). When do the jobs finish?

Answer

A uses up its slice at 0–3 and drops to Q1. B finishes in Q0 at 3–5. C, arriving at t = 5, uses up its Q0 slice at 5–8 and drops to Q1. A runs in Q1 at 8–14, uses up its slice, and drops to Q2 (3 left). C finishes in Q1 at 14–17. A finishes in Q2 at 17–20. Completion times are B 5, C 17, A 20.

5. A system uses a nonpreemptive scheduler. The running process starts a file read, and at that moment a disk read that was started earlier completes. In which of these two events is the scheduler invoked?

Answer

Only in the first. When the running process requests I/O it moves to the waiting state and the CPU becomes free (running → waiting); a choice is mandatory. The completion of the disk read moves a process from waiting to ready; the nonpreemptive scheduler does not stop the running process, and the woken process joins the ready queue.

6. In stride scheduling A has 40, B has 20, and C has 80 tickets; stride = 10,000 / tickets. In a tie the order is A, B, C. What is the order of execution in the first 7 slices?

Answer

The strides are A 250, B 500, C 125. The order is A, B, C, C, A, C, C. A gets 2, B 1, and C 4 slices; the ratio is 2 : 1 : 4 = 40 : 20 : 80. At the end of the seventh slice all three pass values are 500.

7. Three periodic tasks: T1 (p = 4, t = 1), T2 (p = 5, t = 2), T3 (p = 10, t = 3). What does the Liu–

Layland test say? Does RM really miss a deadline?

Answer

U = 0.25 + 0.4 + 0.3 = 0.95. For three tasks the bound is ≈ 0.780; the test cannot decide (the result is 0). If the RM schedule is drawn over the 20 ms hyperperiod (T1 0–1, T2 1–3, T3 3–4, T1 4–5, T2 5–7, T3 7–8, T1 8–9, T3 9–10, …), no deadline is missed. The test is a sufficient condition, not a necessary one.

8. On Linux CFS there are two CPU-bound tasks: A with nice −5 (weight 3121) and B with nice 0 (weight 1024). What are their approximate shares? Whose vruntime grows faster?

Answer

A gets 3121 / 4145 ≈ 75% and B ≈ 25%. Since vruntime += Δ × 1024 / weight, B's vruntime grows about 3 times faster for the same running time; B is chosen less often.

9. Why is a single common queue not used in a system with per-core queues? What is the price of per-core queues?

Answer

In a common queue every access needs a lock; lock contention grows with the number of cores, and because a job may run on a different core each time, its cache stays cold. With per-core queues no lock is needed and affinity is preserved; but the queues may become unbalanced and load balancing (push and pull migration) is needed. A migrated job finds its cache cold on the new core.

10. In RR, q = 9 ms and each context switch takes 1 ms. The processes use their slices to the end. What percentage of CPU time goes to switching? What changes compared with q = 4?

Answer

1 / (9 + 1) = 10%. At q = 4 the overhead was 1 / 5 = 20%. As the slice grows the overhead falls, but each process in the queue waits longer for its next slice and response time gets worse.

5.13 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters "Scheduling: Introduction," "The Multi-Level Feedback Queue," "Lottery Scheduling," and "Multiprocessor Scheduling": builds the MLFQ rules starting from the problems, and presents lottery and stride scheduling with short C code; free at ostep.org.

- C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, 20(1), 1973: the foundational paper that proves the optimality and the utilization bound of RM and the ΣU ≤ 1 condition of EDF.
- Linux kernel documentation, `Documentation/scheduler/sched-eevdf.rst` and `sched-deadline.rst`: describe EEVDF's eligibility and virtual-deadline rules and the admission control of `SCHED_DEADLINE` in the kernel's own documentation.
- P. Yosifovich et al., *Windows Internals*, 7th ed., Part 1, Chapter 4: gives a detailed account of Windows's priority levels, slice computation, dynamic boosts, and multiprocessor thread selection.

5.14 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 5: CPU Scheduling.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Section 2.4: Scheduling.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — Chapters 7–10: Scheduling, MLFQ, Lottery, Multiprocessor.
4. P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7th ed., Part 1, 2017 — Chapter 4: Thread scheduling.
5. Linux kernel documentation: `sched-eevdf.rst`, `sched-design-CFS.rst`, `sched-deadline.rst`, `sched-ext.rst`; the `sched(7)` manual page.
6. I. Stoica, H. Abdel-Wahab, “Earliest Eligible Virtual Deadline First”, 1995.
7. C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, 20(1), 1973, pp. 46–61.
8. C. A. Waldspurger, W. E. Weihl, “Lottery Scheduling: Flexible Proportional-Share Resource Management”, *OSDI*, 1994.

PART II · CHAPTER 6

Synchronization

Threads that share an address space, or processes that use a common region of memory, can read and write the same data without knowing about one another. The scheduler can preempt a thread after any instruction; on a multicore machine two threads really do run at the same time. Two pieces of code that are each correct on their own can therefore give a wrong result when run together. Synchronization is the discipline of regulating access to shared data so as to prevent these errors. These notes begin with the problem itself: the race condition and the critical section. They then examine software solutions built from nothing but reads and writes, the memory model of modern processors, and the atomic instructions that the hardware provides. On this foundation spinlocks, mutexes, semaphores, and monitors are built. The classic problems put these tools to the test. The last sections cover the POSIX and Java interfaces, priority inversion, and the basics of lock-free data structures.

LEARNING OUTCOMES

Problem
Define the race condition and the critical section; explain the three conditions a solution must satisfy (mutual exclusion, progress, bounded waiting).

Software and the memory model
Evaluate the lock variable, strict alternation, and Peterson's solution against these three conditions; explain the effect of reordering on modern processors and the role of memory barriers.

Hardware support
Build locks and counters with `test_and_set`, `compare_and_swap`, XCHG, and atomic variables; trace an interleaving step by step.

Tools
Compare spinlocks, mutexes, semaphores, monitors, and condition variables; apply the Mesa-style `while` rule.

Classic problems
Solve the bounded-buffer, readers–writers, and dining-philosophers problems; recognize the risks of starvation and deadlock.

Real systems
Use the POSIX and Java interfaces; interpret priority inversion (Mars Pathfinder) and the basics of lock-free structures.

6.1 Race conditions

When processes and threads run alone, their behavior is easy to predict: the same input gives the same result. When they share data, this ease is lost. The result now depends not only on the code but also on the order in which the instructions of the two flows are interleaved. This section shows where the problem comes from through three examples: a counter in the kernel, a printer queue, and a single-line `count++`.

6.1.1 Why synchronization?

> **Definition 6.1**
>
> **Synchronization** is the coordination of how concurrently running processes or threads access shared data, so that the data remains consistent.

A process can be interrupted after any of its instructions. The timer interrupt arrives, the scheduler picks another process, and the interrupted process leaves the update it was making half done. If another process reads the half-finished update in the meantime, it sees an inconsistent value. On multicore systems not even an interrupt is needed: two cores can truly write the same memory word at the same time. Figure 6.1 shows this situation. Two threads carry out the "read, modify, write" steps on the same `count` variable; a context switch in the middle leaves the update half done.

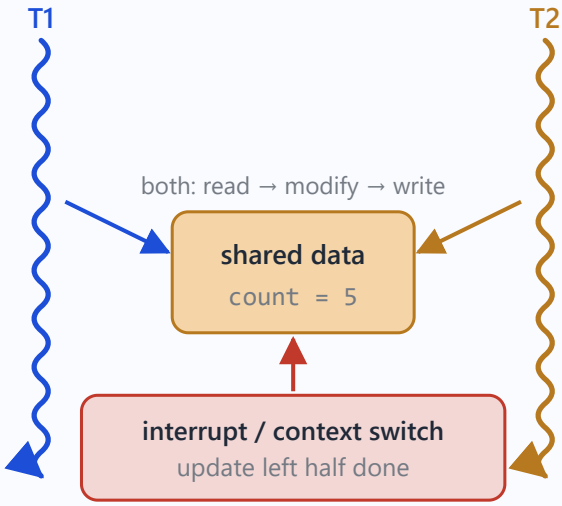


Figure 6.1: Two threads read and write the same shared variable. An interrupt or context switch in the middle can leave the update half done.

The symptoms of synchronization errors are varied:

- **Inconsistent data and lost updates.** One of two increments is lost; a node of a linked list is torn out of the list.
- **An overwritten shared buffer.** When a non-reentrant function, one that uses a common static buffer, is called from two threads, one overwrites the other's result.
- **Deadlock.** Two threads wait forever for each other to release something.
- **Starvation.** A thread never gets its turn; the others always get ahead of it.

What these errors have in common is that they appear *occasionally*. The program runs correctly a hundred times and gives a wrong result on the hundred-and-first. Stepping through it with a debugger changes the timing, so the error often disappears. Synchronization errors are therefore prevented not by testing but by getting the code right from the start.

6.1.2 A race in the kernel: fork() and the pid

Races are not seen only in user programs; the kernel itself is full of shared data. In Silberschatz's example the kernel keeps the next free process identifier in a variable called `next_available_pid`. Its value is 2615. Processes P0 and P1 call `fork()` at almost the same moment. On each call the kernel code does three things: it reads the value, gives it to the child, and increments it and writes it back.

1. **P0 reads.** P0's kernel code reads `next_available_pid`: 2615. It is interrupted before it can increment and write.
2. **P1 reads and writes.** P1 also reads 2615, gives 2615 to its child, and writes 2616 to the variable.

3. **P0 continues.** P0 resumes where it left off. With the old value it holds, it gives its child 2615 as well, and it also writes 2616.

Figure 6.2: A race inside `fork()`. Both processes read the value 2615; both children get the identifier 2615 and one of the two increments is lost (adapted from Silberschatz Figure 6.2).

As a result two different processes have been given the same identifier and the counter has grown by one instead of two (Figure 6.2). Had the operations run in another order, for example if P0 had finished its three steps without being interrupted, the children would have received the identifiers 2615 and 2616. That the result depends on the order of the operations is the very essence of the basic concept of this chapter.

6.1.3 The printer spooler directory

Tanenbaum explains the same problem with a printer queue. The names of files to be printed are written in turn into numbered slots of a **spooler directory**. The printer process scans the directory and prints the files. There are two shared variables: `out` points to the next slot to be printed and `in` to the first free slot. Initially slots 4, 5, and 6 are full; `out = 4` and `in = 7`.

Processes A and B each want to print a file. Each reads `in` into a local variable `next_free`, writes its file name into that slot, and increments `in`. Figure 6.3 shows the following interleaving:

1. A reads `in` and sets `next_free = 7`. Right then its time slice ends.
2. B also reads `in`: 7 again.
3. B writes the name `b.txt` into slot 7 and sets `in = 8`.
4. A resumes where it left off. Because its local variable is still 7, it writes `a.txt` into slot 7, overwrites B's entry, and sets `in = 8`.

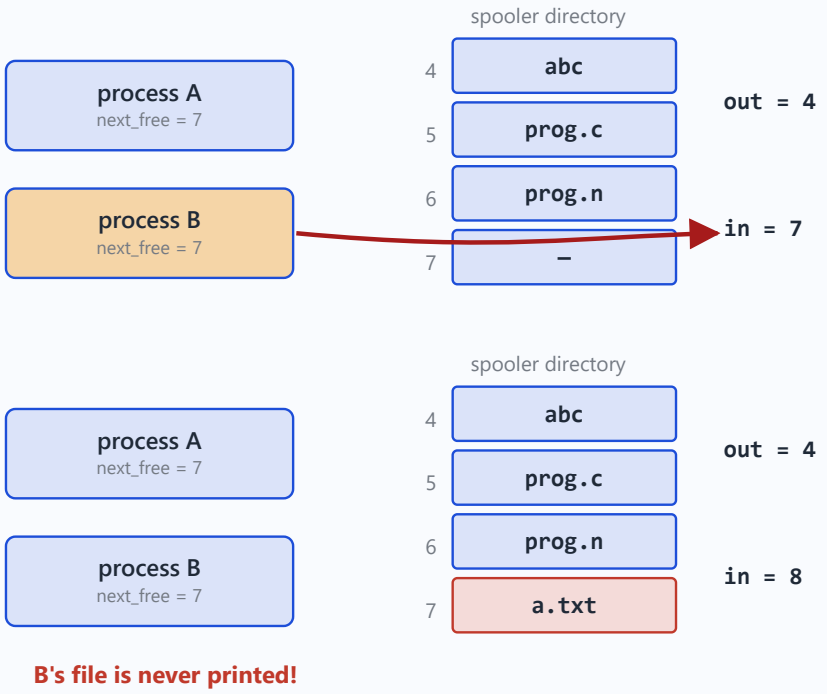


Figure 6.3: A race in the printer spooler directory. (a) A and B have read `in` and the `next_free` of both is 7. (b) First B writes `b.txt` into slot 7, then A writes `a.txt` into the same slot. The directory looks consistent, but B's file is never printed.

Seen from outside, the directory is consistent: `in = 8` and all the slots are full. The printer process notices no inconsistency. But B's output is lost, and B's user waits at the printer in vain.

Definition 6.2

A **race condition** arises when two or more processes access shared data and the final result depends on the order in which the accesses are executed.

6.1.4 One line, three instructions: count++

A race condition does not need long code. A single `count++` line in C compiles to three instructions at the machine level:

1. `reg = count` : load the value from memory into a register,
2. `reg = reg + 1` : increment the register,
3. `count = reg` : write the register back to memory.

`count--` also consists of the same three steps; only the middle operation is a subtraction. Each thread has its own register set. When a thread is preempted, its registers are saved into its PCB; the intermediate value in the register is not reflected in memory.

Example 6.1

Initially `count = 5`. A producer executes `count++` and a consumer executes `count--`. The correct result is 5. What is the result in the following interleaving?

Time	Who	Instruction	reg_1	reg_2	count
T_0	producer	`reg1 = count`	5	–	5
T_1	producer	`reg1 = reg1 + 1`	6	–	5
T_2	consumer	`reg2 = count`	6	5	5
T_3	consumer	`reg2 = reg2 - 1`	6	4	5
T_4	producer	`count = reg1`	6	4	6
T_5	consumer	`count = reg2`	6	4	**4**

Table 6.1: One interleaving of the `count++` and `count--` instructions.

1. **T_0–T_1.** The producer loads the value into its register and increments it. The register holds 6, but memory still holds 5. It is interrupted at exactly this moment.
2. **T_2–T_3.** The consumer reads 5 from memory and decrements it to 4 in its own register.
3. **T_4.** The producer runs again and writes 6 to memory.
4. **T_5.** The consumer writes the 4 in its own register to memory and overwrites the producer's 6. The result is 4; the correct value was 5.

Had T_4 and T_5 been swapped, the producer would have written last and the result would have been 6. The six instructions of the two threads can be interleaved in 20 different ways while each thread's own order is preserved. When all 20 interleavings are tried one by one, only three results appear: 4, 5, and 6. The correct result, 5, is obtained only if the three instructions of one thread complete before the other thread's read.

The **interleaving** mentioned here is the arrangement of the instructions of concurrent flows into a single sequence, with the order inside each flow preserved. On a single core the scheduler determines this sequence. On multiple cores, the order in which the instructions on different cores reach memory determines it. For two flows of m and n instructions the number of possible interleavings is $C(m + n, m)$; for two flows of three instructions each it is $C(6, 3) = 20$. This number grows very quickly as the flows get longer, and trying them all by hand becomes impossible.

Another face of the race is that the same program produces a different result on every run. The two processes below use a shared counter called `n`: the first process keeps incrementing `n`, and the second occasionally prints `n` and resets it to zero. The aim is to report how many events were counted in a given period.

```
/* Process 1 */     /* Process 2 */
repeat            repeat
  n = n + 1;        print(n);
                    n = 0;
```

With `n = 5`, the steps of the two processes can be interleaved in three different ways:

- `n = n + 1; print(n); n = 0` → 6 is printed and the counter becomes 0. This is the correct behavior.
- `print(n); n = 0; n = n + 1` → 5 is printed and the counter becomes 1. This is also correct: the increment is left for the next report.
- `print(n); n = n + 1; n = 0` → 5 is printed and the counter becomes 0. The increment appears in no report: it has been *lost*.

Such an execution is called a **non-reproducible** execution. The faulty result appears only under a particular timing, and rerunning with the same input usually does not repeat it.

The same race is easy to see on a real machine. In the following Java program two threads each increment an unprotected counter 100,000 times. The correct result is 200,000.

```
public class Race {
  static int counter = 0;                 /* unprotected */
  public static void main(String[] a) throws InterruptedException {
    Runnable task = () -> { for (int n = 0; n < 100_000; n++) counter++; };
    Thread t1 = new Thread(task), t2 = new Thread(task);
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("counter = " + counter);
  }
}
```

When the program was run three times on the same machine with JDK 21, it gave these outputs:

```
counter = 170429
counter = 150258
counter = 135129
```

These numbers change on every run; on another machine, under another load, very different values may come out. The only thing that does not change is that the result is almost never 200,000. Each lost increment is the result of an interleaving like the one in Table 6.1.

Common mistake

Believing that a one-line C expression is indivisible. Expressions such as `count++`, `x += 2`, or `a[i] = a[i] + 1` are a single line in the source code but a read–modify–write sequence on the machine, and can be interrupted between any two of its steps. Another misconception is "a very short operation, nothing will get in between." The probability of a race may be low; but in code that runs millions of times per second, a low-probability event happens often.

6.2 The critical-section problem

Race conditions have a common structure: in a certain part of its code each process touches shared data. To solve the problem it is not necessary to protect the whole program; it is enough to prevent these parts from running at the same time. This section defines that part and what is expected of a solution.

6.2.1 Entry, critical, exit, and remainder sections

Let the system have n processes: $\{P_0, P_1, \ldots, P_{n-1}\}$. Each has a section of code in which it accesses shared data: writing a global variable, updating a table, writing a file.

Definition 6.3

A **critical section** is a piece of code in which a process accesses shared data and in which only one process should be at a time. **Mutual exclusion** is the condition that while one process is in its critical section, no other process is in its own critical section.

Before entering its critical section, a process asks for permission; the code that makes this request is called the **entry section**. When leaving the critical section it announces this to the others; this is the **exit section**. The rest of the code is the **remainder section**. A typical process repeats these four parts in a loop (Figure 6.4):

```
while (true) {
    /* entry section             */
        /* critical section      */
    /* exit section              */
        /* remainder section       */
}
```

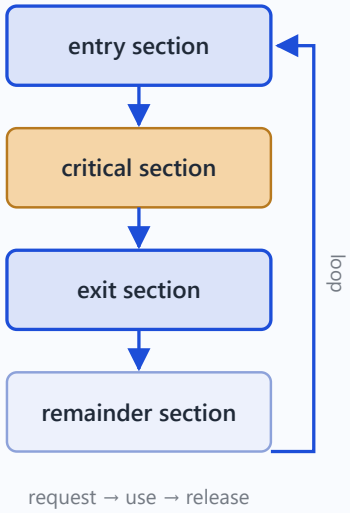


Figure 6.4: The general structure of a process: request permission (entry), use (critical section), release (exit), and the remainder section. The loop repeats continuously.

The critical-section problem is to design an entry and exit protocol that lets the processes cooperate. The contents of the critical section and the remainder section do not concern the protocol. What matters is that the entry and exit sections work correctly whatever the timing.

6.2.2 The three requirements of a solution

Mutual exclusion alone is not enough. A protocol that lets no one into the critical section also "provides" exclusion but is useless. Silberschatz therefore asks three requirements of a solution.

Definition 6.4

1. **Mutual exclusion.** While P_i is in its critical section, no other process can execute in its own critical section.
2. **Progress.** If the critical section is empty and some processes want to enter, only the processes that are not in their remainder sections take part in deciding who enters next. This decision cannot be postponed indefinitely.
3. **Bounded waiting.** After a process has made its request to enter, there is an upper bound on how many times the other processes can enter the critical section before its request is granted.

The first requirement is a **safety** property: "nothing bad ever happens." The second and third are **liveness** properties: "something good eventually happens." The progress requirement ensures that the system as a whole does not get stuck, and bounded waiting ensures that every individual process eventually gets a turn. If bounded waiting holds, no process falls into **starvation**.

None of the three requirements makes an assumption about the speed of the processes. It is assumed only that each process proceeds at a speed greater than zero; the relative speeds of the processes are unknown. Tanenbaum writes the same requirements as four items. Table 6.2 shows how these items match the three requirements.

Tanenbaum's four conditions	Counterpart
No two processes may be in their critical sections at the same time.	Mutual exclusion
No assumptions about CPU speed or count.	Each process proceeds at a speed greater than zero; relative speeds are unknown.
No process outside its critical section may block others.	Progress
No process should wait forever.	Bounded waiting

Table 6.2: Tanenbaum's four conditions and their counterparts (Silberschatz 10e §6.2; Tanenbaum MOS §2.3.2).

Common mistake

Confusing progress with bounded waiting. Progress means that, *when the critical section is empty*, one of those wanting to enter can do so in finite time. Bounded waiting is the limit on how many times way is given to others for *a particular process*. A lock can satisfy progress and still violate bounded waiting: each time someone enters, but the same unlucky process is always left outside.

6.2.3 A critical-section timeline

The behavior of a good solution is clearly visible in Tanenbaum's timeline (Figure 6.5):

1. **T1.** A enters its critical section.
2. **T2.** B also wants to enter. Because the section is occupied, B blocks or busy-waits.
3. **T3.** A leaves and the waiting B enters at once.
4. **T4.** B leaves.

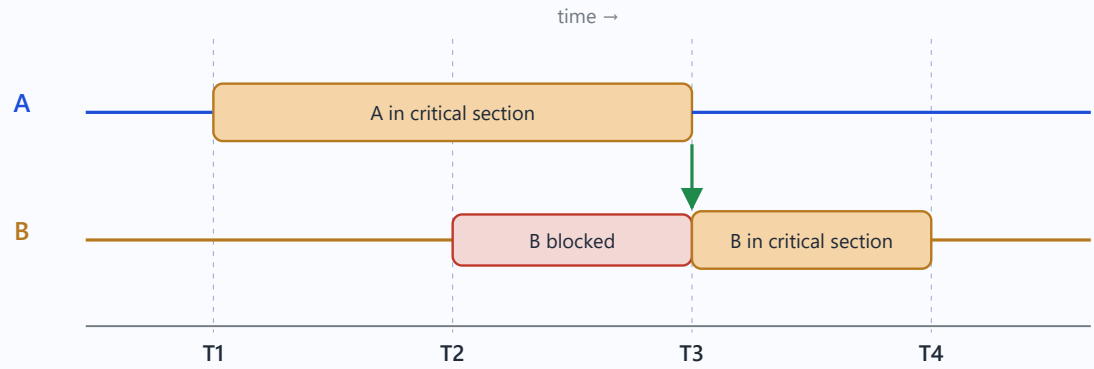


Figure 6.5: Mutual exclusion with a critical section. A is in the critical section from T1 to T3; B arrives at T2 and waits until T3, then stays in the critical section until T4 (adapted from Tanenbaum MOS Figure 2-22).

In the timeline, at no moment are two processes in the critical section together. A good solution must guarantee this behavior not only under this timing but under all possible timings. A single interleaving that violates one of the three requirements is enough to invalidate a solution.

6.3 Software approaches

The first attempts at a solution use no special hardware support; they are built only from ordinary read and write instructions. Most of these attempts fail, but seeing why they fail is the best way to understand the three requirements. The section begins with an approach specific to operating systems, disabling interrupts, and ends with Peterson's algorithm.

6.3.1 Disabling interrupts

On a single-core machine a process loses the processor only because of an interrupt. If a process disables interrupts before entering its critical section and re-enables them when it leaves, no timer interrupt arrives and no context switch takes place. On x86 this is done with the `cli` (clear interrupt flag) and `sti` (set interrupt flag) instructions. The idea is simple but unsuitable for user programs:

- A process that forgets to re-enable interrupts, or that enters an infinite loop in its critical section, locks up the whole system. The operating system can never take back control.
- While interrupts are disabled, I/O completion and timer interrupts wait. A long critical section leaves the system unresponsive.
- It does not work on a multicore machine. `cli` disables interrupts only on the core that executes it; the other cores continue to access the shared data (Figure 6.6).
- `cli` and `sti` are privileged instructions. A program in user mode cannot execute them; if it tries, the processor raises a trap.

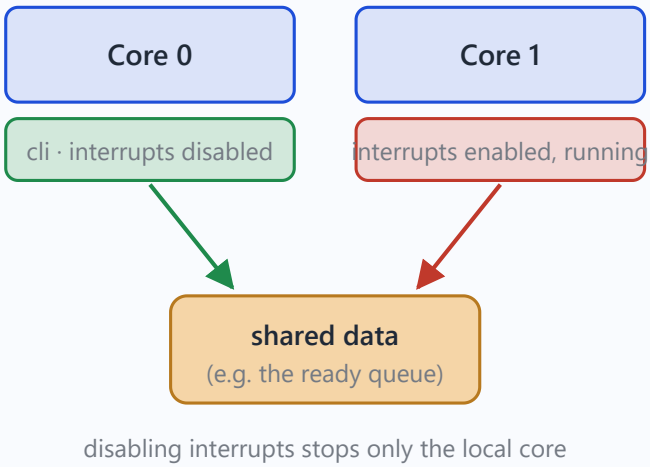


Figure 6.6: Disabling interrupts stops only the local core. While interrupts are disabled on Core 0, Core 1 keeps running and accessing the shared data (for example the ready queue).

The method is common inside the kernel, but it is used briefly and locally. In Linux, `local_irq_save()` disables interrupts on the local core and saves the old state. For data shared with an interrupt handler, `spin_lock_irqsave()` both disables interrupts and takes a spinlock against the other cores. Both measures are needed together: disabling interrupts protects against the handler on the same core, and the spinlock protects against the other cores.

6.3.2 The lock variable

The first software solution that comes to mind is a shared **lock variable**. If the value of `lock` is 0, the section is empty; if 1, someone is inside. A process entering waits until the value is 0 and then writes 1; on leaving it writes 0 again.

```
int lock = 0;           /* shared */

void enter(void) {
  while (lock != 0)
    ;                   /* busy wait */
  lock = 1;
}
void leave(void) {
  lock = 0;
}

/* each process:  enter(); critical(); leave(); */
```

The flaw in the code is that the "test" and "set" steps are separate. A process can be interrupted between the test on line 4 and the assignment on line 6. The following interleaving breaks mutual exclusion:

1. **Start.** `lock = 0`; the section is empty.
2. **P0 tests.** P0 reads `lock`: 0. It leaves the loop.
3. **P0 is interrupted.** P0 is interrupted before performing the assignment `lock = 1` on line 6. P1 runs and also reads 0.
4. **P1 enters.** P1 sets the lock to 1 and enters the critical section.
5. **P0 enters.** P0 resumes where it left off. Since it has already done the test, it does not read the value again; it also writes 1 and enters.

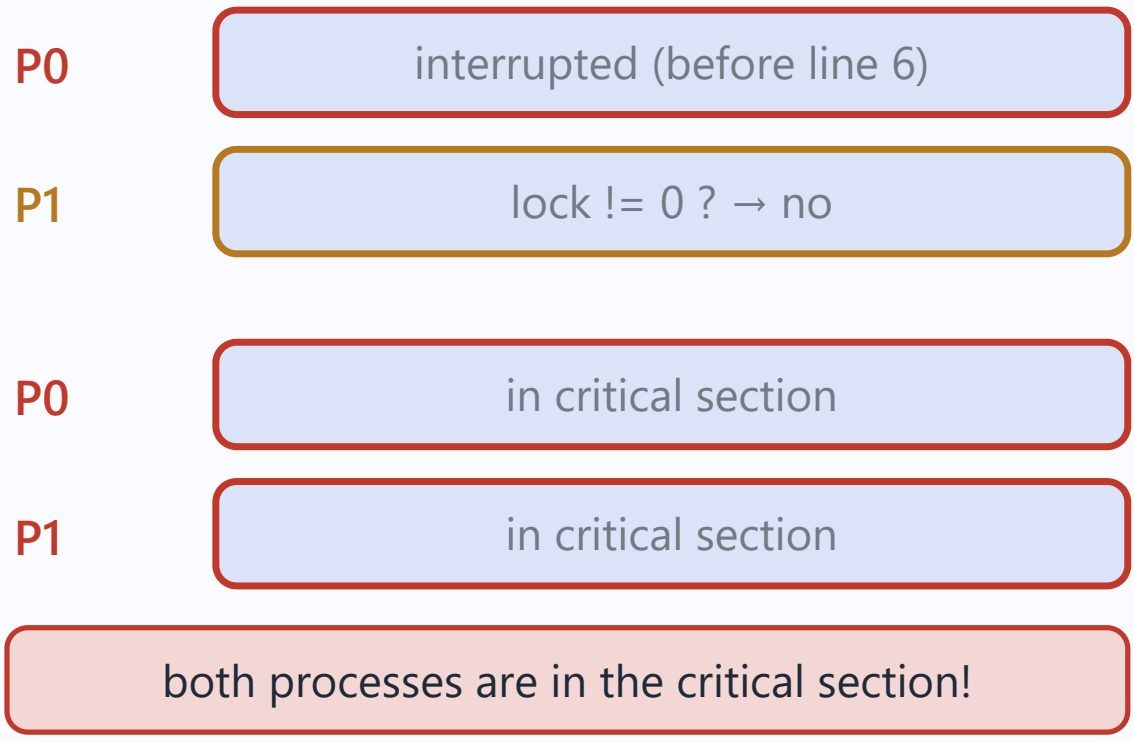


Figure 6.7: The failure of the lock variable. (a) P0 has passed the test but has been interrupted before line 6; P1 also reads `lock` as 0. (b) Both processes are in the critical section.

When the entire state space is explored with a program, the same result comes out: a state in which two processes are in the critical section at the same time is reachable. The problem is circular at heart. The lock itself is a shared variable, and testing and changing it safely would need yet another lock. So a lock variable built from ordinary reads and writes is not enough by itself. The solution is to combine the test and the assignment into a single indivisible instruction; the hardware provides this instruction. That one can go further with reads and writes alone is shown by strict alternation and Peterson's algorithm.

6.3.3 Strict alternation

The second attempt uses a turn variable instead of a lock. The shared variable `turn` shows whose turn it is to enter the critical section. P0 enters only when `turn == 0`, P1 only when `turn == 1`. The process that leaves gives the turn to the other.

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* wait */
  critical_section();
  turn = 1;
  noncritical();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* wait */
  critical_section();
  turn = 0;
  noncritical();
}
```

This solution provides mutual exclusion: `turn` cannot be both 0 and 1 at the same time, and only the process that leaves the critical section changes it. Bounded waiting is also provided: the turn passes to the other process at every exit, so a process waits for at most one entry. The problem is in the progress requirement. The following scenario shows this clearly (Figure 6.8):

1. With `turn = 0`, P0 enters; P1 busy-waits.
2. On leaving, P0 sets `turn = 1`; P1 enters.
3. On leaving, P1 sets `turn = 0` and starts a long noncritical job. P0 enters and leaves once more and sets `turn = 1`.
4. P0 wants to enter again but it is P1's turn. The section is empty and P1 is not interested in the critical section; yet P0 cannot enter.

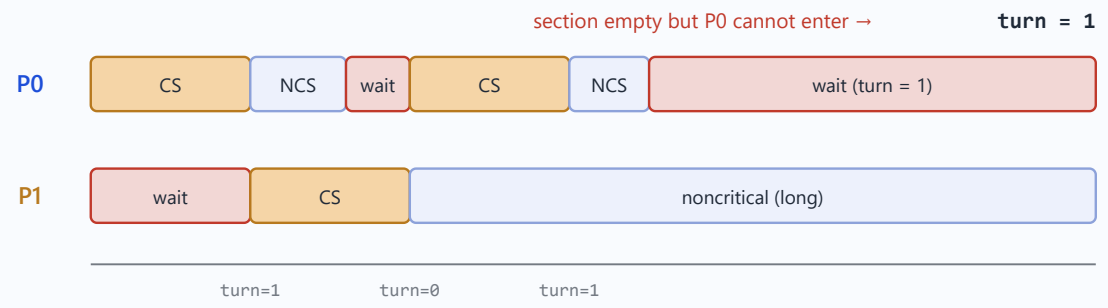


Figure 6.8: Violation of the progress requirement in strict alternation. CS is the critical section and NCS the noncritical (remainder) section. In the last stage, although the section is empty, P1, which is outside the critical section, blocks P0.

Strict alternation makes the processes run *in forced turns*. Of two processes with different speeds, the faster has to wait for the slower to finish its remainder section. A process that gets stuck in its remainder section, or never returns, keeps the other out forever. Also, the waiting process wastes processor time in the `while (turn != i) ;` loop.

Definition 6.5

Busy waiting is waiting by continuously testing a condition in a loop. The waiting process does not block; it keeps using the processor. A lock that busy-waits is called a **spinlock**.

6.3.4 Peterson's algorithm

In 1981 G. L. Peterson gave a short solution that works correctly for two processes using only reads and writes. The algorithm combines the lock-variable idea with the strict-alternation idea. Each process has a `flag`: `flag[i] = true` means "P_i wants to enter." The shared `turn` variable decides who enters first when both want to.

```
/* shared */
bool flag[2] = { false, false };
int  turn;

/* P_i  (j = 1 - i) */
while (true) {
  flag[i] = true;    /* I want in      */
  turn = j;          /* after you      */
  while (flag[j] && turn == j)
    ;                /* busy wait      */
  /* critical section */
  flag[i] = false;
  /* remainder section */
}
```

The entry section is two lines. The process first announces its intention (`flag[i] = true`), then politely gives the turn to the other (`turn = j`, that is, "after you"). Then it waits as long as the other process wants to enter and it is the other's turn. On exit it merely lowers its own flag.

How the algorithm satisfies the three requirements can be shown in a few sentences:

- **Mutual exclusion.** If both processes are in the critical section, `flag[0]` and `flag[1]` must both be `true`. In that case each can leave the loop only when `turn` is in its favor. But `turn` cannot be both 0 and 1 at once. The process that wrote `turn` last has left the turn to the other and it is its job to wait.
- **Progress.** If P_j does not want to enter, `flag[j] = false`, and P_i's loop condition is immediately false: P_i enters without waiting. A process in its remainder section does not block the other.
- **Bounded waiting.** If P_j leaves while P_i is waiting and immediately wants to enter again, it must write `turn = i` in its entry section. This assignment clears P_i's way. P_i waits for at most one entry.

Peterson's algorithm is for two processes only. For *n* processes there is Lamport's bakery algorithm: each process takes a ticket number and the process with the smallest number enters. These algorithms have two important costs. The first is busy waiting; the waiting process burns processor time. The second is more important: the correctness proof assumes that all processors see the memory operations in a single common order. This assumption is called **sequential consistency**, and modern hardware does not provide it on its own.

Common mistake

Thinking that the order of the two assignments in the entry section does not matter. If `turn = j` is written first and `flag[i] = true` second, the algorithm breaks. P0 writes `turn = 1` and is interrupted. P1 writes `turn = 0` and `flag[1] = true`; since `flag[0]` is still `false`, it enters. P0 continues and writes `flag[0] = true`; since `turn` is 0, the condition is false and P0 enters too. When the entire state space is explored, it is verified that this order lets both processes into the critical section together; with the correct order, no such state is ever reachable.

6.3.5 Code trace: Peterson

The behavior of the algorithm is best shown by tracing an interleaving step by step. The following code separates the entry and exit sections into the functions `enter(i)` and `leave(i)`. Memory is assumed to be sequentially consistent: all memory operations can be explained by a single common order that preserves each thread's program order.

```
bool flag[2] = { false, false };
int turn;
void enter(int i) {
  int j = 1 - i;
  flag[i] = true;   /* I want in */
  turn = j;         /* after you */
  while (flag[j] && turn == j)
    ;               /* busy wait */
}
void leave(int i) {
  flag[i] = false;
}
```

Table 6.3 traces a scenario in which two processes want to enter almost at the same time. Each row gives the shared variables and the states of the processes after a step. "Entering" means being in the entry section, "spinning" means being in the waiting loop, and CS means being in the critical section.

Step	Event	flag[0]	flag[1]	turn	P0	P1
1	initial	false	false	?	remainder	remainder
2	P0: `flag[0] = true`	true	false	?	entering	remainder
3	P1: `flag[1] = true`	true	true	?	entering	entering
4	P0: `turn = 1`	true	true	1	entering	entering
5	P1: `turn = 0`	true	true	0	entering	entering
6	P0 tests: `turn == 1` is false	true	true	0	CS	entering
7	P1 tests: condition true	true	true	0	CS	spinning
8	P0: `flag[0] = false`	false	true	0	remainder	spinning
9	P1 tests: `flag[0]` is false	false	true	0	remainder	CS
10	P0: `flag[0] = true`, `turn = 1`	true	true	1	entering	CS
11	P0 tests: condition true	true	true	1	spinning	CS
12	P1: `flag[1] = false`	true	false	1	spinning	remainder
13	P0 tests: `flag[1]` is false	true	false	1	CS	remainder

Table 6.3: Step-by-step trace of Peterson's algorithm (sequentially consistent memory).

Steps 4 and 5 are the key point of the trace. Both processes give the turn to the other; but `turn` is a single variable and the value of the last writer stays. Because P1 wrote last, `turn = 0` and it falls to P1 to wait. In step 6 P0's condition `flag[1] && turn == 1` is `true && false`; P0 enters. In step 7 P1's condition `flag[0] && turn == 0` is true and P1 spins. Mutual exclusion is preserved.

In step 8 P0 leaves and lowers its flag. P1's condition turns false and P1 enters (step 9): progress is satisfied. In step 10 P0 wants to enter again; this time P0 is the last to write `turn` and it falls to P0 to wait. When P1 leaves (step 12), P0 enters. Had P1 wanted to enter again at once, it would have written `turn = 0` and waited. So P0 waits for at most one entry: bounded waiting is satisfied too.

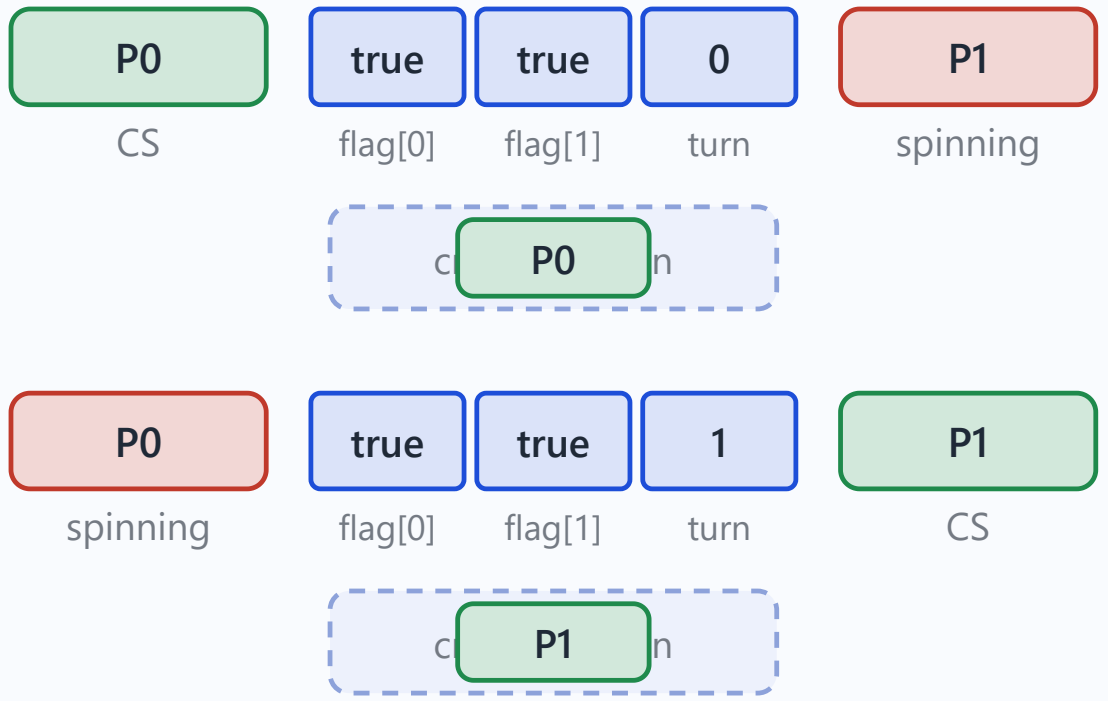


Figure 6.9: Two moments from the Peterson trace. (a) Step 7: `turn = 0`, P0 is in the critical section, P1 is spinning. (b) Step 11: `turn = 1`, the roles have changed; P1 is in the critical section, P0 is spinning.

This trace is a single interleaving; correctness has to hold for all of them. The small state space of two-process Peterson was explored completely with a program. In sequentially consistent memory neither a violation of exclusion nor a deadlock was found.

6.4 The memory model

The proof of Peterson's algorithm assumes that when a process writes to a variable, the write is visible to everyone immediately and in the same order. On a modern multicore processor this assumption is false. Both the processor and the compiler may change the order of memory accesses as long as the result of a single thread does not change. This section explains how this reordering breaks Peterson and how it is prevented.

6.4.1 Why does Peterson break on modern processors?

Each core has a **store buffer**. When a core executes a store instruction, the value first goes into this buffer. The core moves on to the following instructions without waiting for the store to reach memory; the buffer is drained to memory in the background. This arrangement makes stores much faster. If the same core then reads that address, it gets the value from its own buffer. The other cores, however, see the old value until the buffer drains.

```
/* P_i  (i = 0 or 1, j = 1 - i) */
flag[i] = true;
turn = j;
while (flag[j] && turn == j)
  ;
/* critical section */
flag[i] = false;
```

In Peterson's entry section each process first does two writes (`flag[i]` and `turn`) and then one read (`flag[j]`). With a store buffer, the following execution is possible (Figure 6.10):

1. **Start.** In memory both flags are `false` .
2. **P0 writes.** P0 writes `flag[0] = true` and `turn = 1` . Both writes wait in Core 0's buffer; they are not yet visible in memory.
3. **P1 writes.** P1 does the same; its writes are in Core 1's buffer.
4. **P0 reads.** P0 reads `flag[1]` from memory: `false` . It leaves the loop and enters the critical section.
5. **P1 reads.** P1 also reads `flag[0]` from memory as `false` and enters. Both processes are in the critical section.
6. **The buffers drain.** Memory finally gets `flag[0] = flag[1] = true` . But it is too late.

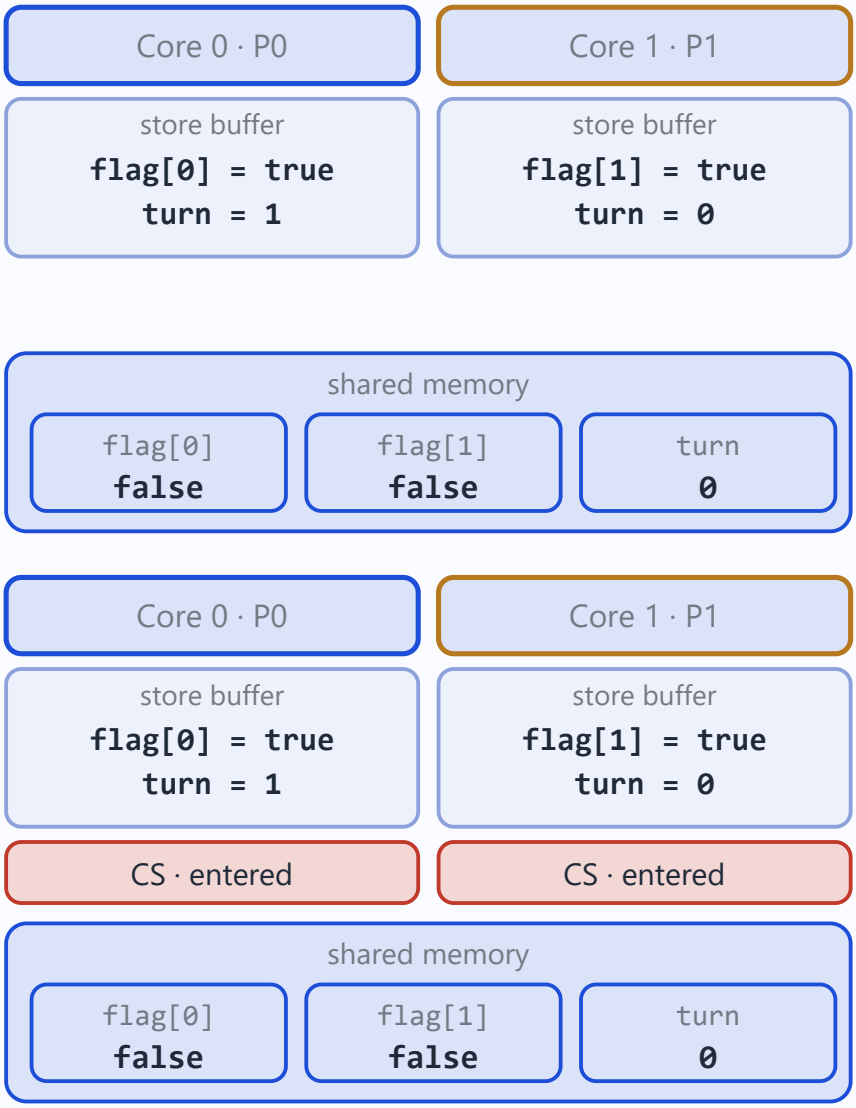


Figure 6.10: The store buffer breaks Peterson. (a) The writes of the two processes wait in their own buffers; in memory the flags are still `false`. (b) Both processes read the other's flag from memory as `false` and enter the critical section.

No instruction here has run "wrongly." Each core has executed the program in order from its own point of view. The problem is that one core's write reaches the other core *after* that core's next read. For an outside observer the effect is as if "the write slipped past the following read." The compiler can create the same effect: it can move the read of `flag[j]` ahead of the writes, or read a value that does not change in the loop once and keep it in a register. For these notes, all executions of Peterson were explored in a model built with store buffers. An execution in which both processes entered together was found in the version without barriers; none was found in the version with full barriers.

6.4.2 Memory models and reordering

Which reorderings a processor allows is determined by its **memory model**. The experiment that shows the difference most briefly is called the store-buffer litmus test:

```
/* x = y = 0 */
/* T1 */        /* T2 */
x = 1;          y = 1;
r1 = y;         r2 = x;

/* r1 == 0 && r2 == 0 ?  */
```

Example 6.2

Initially `x = y = 0`. T1 and T2 execute the two lines above. Which (`r1`, `r2`) results are possible?

1. **Number of interleavings.** Each thread has two instructions. With each one's own order preserved, C(4, 2) = 6 interleavings arise.
2. **Sequential consistency.** All six orders are tried. Since in each thread the write comes before the read, at least one write has completed by the time the first read is made. The results are (0, 1), (1, 0), and (1, 1); (0, 0) comes out in no order.
3. **A processor with a store buffer.** While T1's write `x = 1` waits in the buffer, T1 can read `y` from memory as 0. The same holds for T2. So on x86, ARM, and RISC-V, `r1 = r2 = 0` can also be observed.
4. **With a full barrier.** If a full memory barrier is placed between the write and the read in each thread, (0, 0) becomes impossible again. When all executions are explored in the store-buffer model, only the three results of SC remain.

Definition 6.6

Sequential consistency (SC) means that the memory operations of all threads can be arranged in a single common order. In this order each thread's own program order is preserved and every core sees the same order (Lamport, 1979).

Model	Reordering allowed
Sequential consistency (SC)	None; all cores see a single common order.
x86-64 (TSO)	Only "write → later read"; writes are ordered among themselves.
ARMv8, RISC-V (weak)	Reads and writes can pass one another if there is no dependency between them.
Compiler	May move, keep in a register, or delete any access if equivalent for a single thread.

Table 6.4: Memory models and the reorderings they allow.

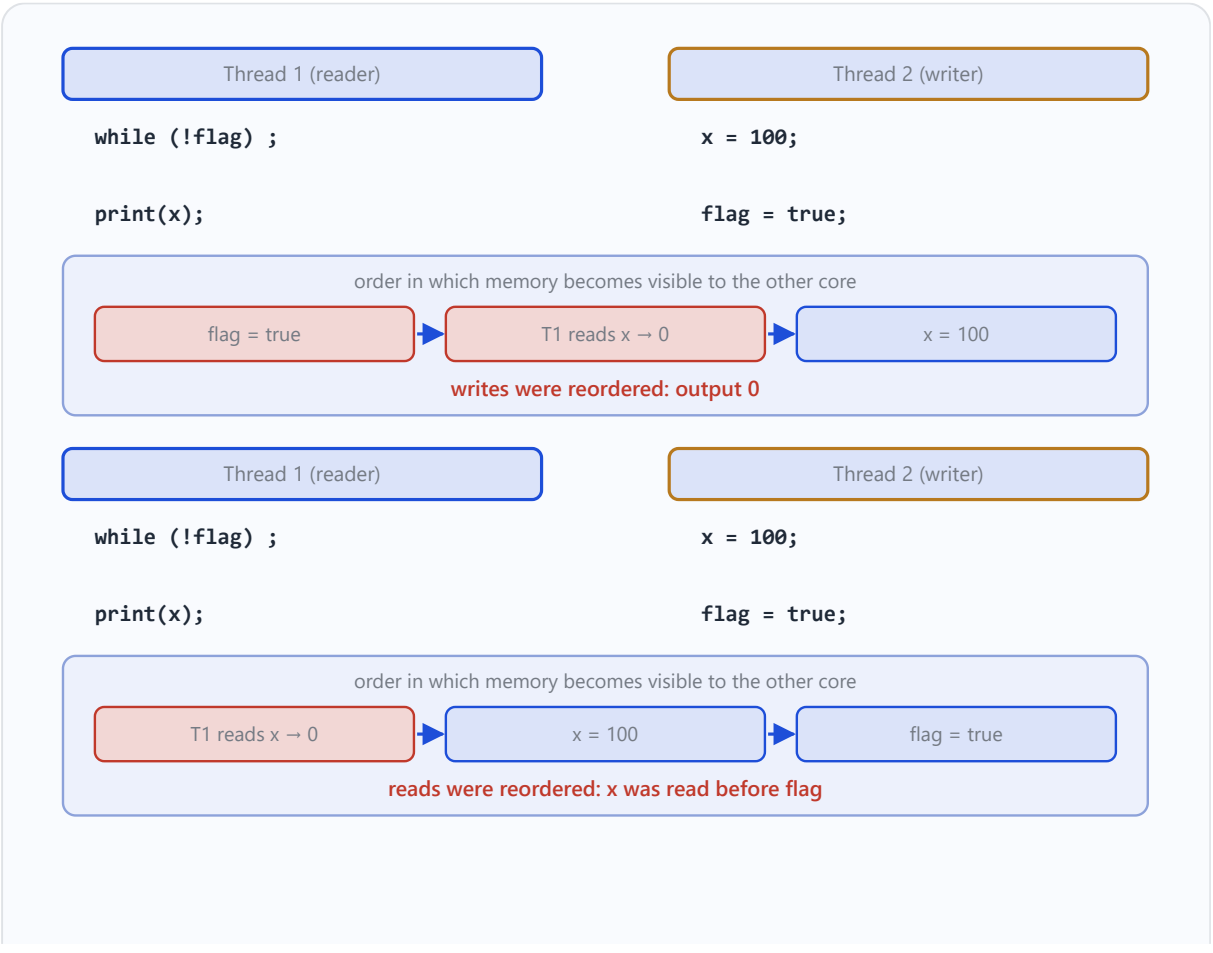
x86's model is called **total store order** (TSO). This model is quite close to SC; it allows a single relaxation, the one arising from the store buffer. This is exactly the relaxation that breaks Peterson. In weak models such as ARM and RISC-V more reordering is possible. For example, two writes may appear to another core in the reverse of the order in which they were written.

Because of these differences, concurrent code is written not against "what the processor does" but against the language's memory model. The `volatile` keyword in C is not a tool for this job. `volatile` merely asks the compiler to actually perform every access to that variable. For threads it provides neither atomicity nor ordering. The right tools are C11's `<stdatomic.h>` header, Java's `volatile` fields and the `java.util.concurrent` package, and locks.

6.4.3 Memory barriers

In Silberschatz's example two threads communicate through a flag. Thread 2 first writes `x = 100` and then `flag = true`. Thread 1 waits until `flag` is true and then prints `x`. The programmer expects 100 to be printed. On a weak memory model, two things spoil this expectation (Figure 6.11):

- **Writes are reordered.** Thread 2's writes reach the other core in reverse order. Thread 1 sees `flag` as `true` but `x` is still 0.
- **Reads are reordered.** Thread 1 itself may read `x` before `flag`. When it leaves the loop, the `x` it holds is the old value.



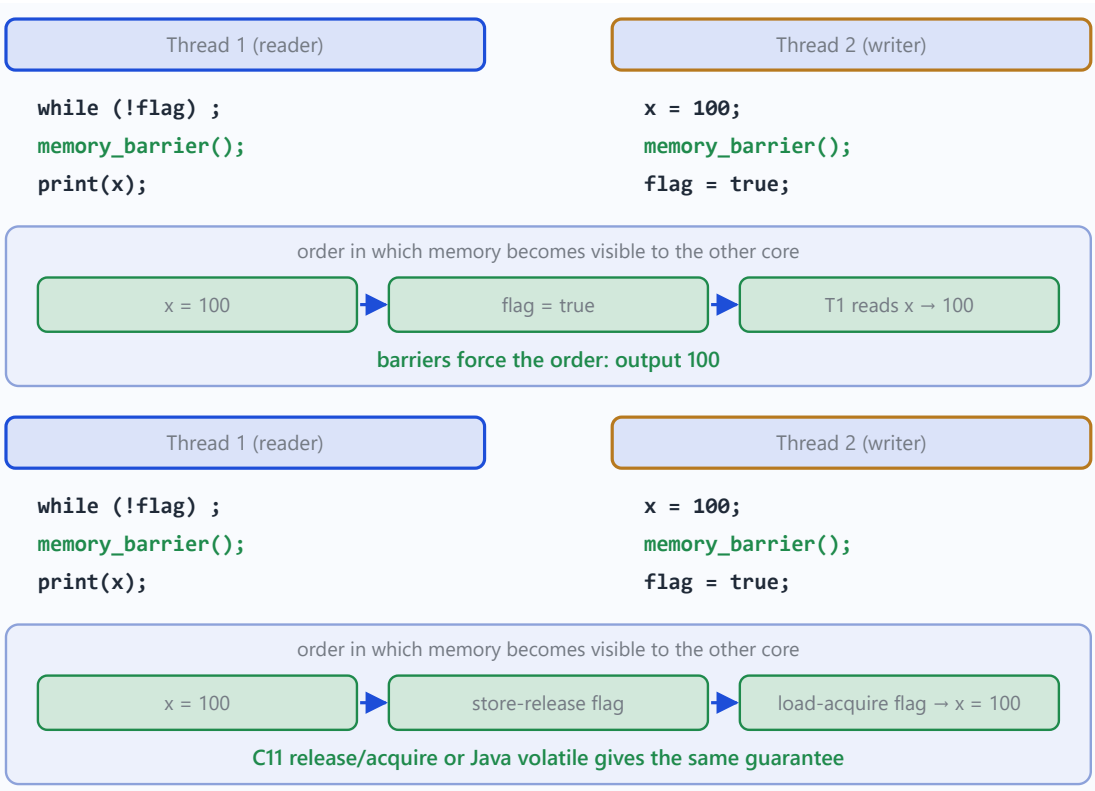


Figure 6.11: Memory barriers. (a) Without a barrier the writes are reordered and the output is 0. (b) Reads are reordered: `x` is read before `flag`. (c) With a barrier on both sides the order is enforced and the output is 100. (d) C11 release/acquire atomics or Java `volatile` give the same guarantee more cheaply.

Definition 6.7

A **memory barrier** (or **fence**) is an instruction that forces all memory accesses before it to complete, and become visible to the other cores, before those after it.

In the example thread 2 puts a barrier between the two writes: the write of `x` is certainly visible before `flag`. Thread 1 also puts a barrier after the loop: the read of `x` is certainly done after the read of `flag`. Together the two barriers guarantee that the output is 100. In real code the name of the barrier varies with the platform: `mfence` on x86, `dmb` on ARM, `smp_mb()` in the Linux kernel, `atomic_thread_fence` in C11.

A full barrier is an expensive instruction, because it waits for the store buffer to drain. In most cases a lighter guarantee suffices. The writing side writes `flag` with **release** semantics: all earlier accesses are visible before this write. The reading side reads `flag` with **acquire** semantics: later accesses are not performed before this read. Implementations of locks and semaphores already include these barriers. So in code protected by a lock there is no need to write barriers separately. Barriers are the tool of those who write locks and atomic variables.

Common mistake

Writing Peterson's algorithm in C with `volatile` variables and assuming it will work correctly on x86. `volatile` keeps the compiler from holding the read in a register but does not affect the processor's store buffer. The program seems to work correctly for a long time, and then occasionally two threads enter the critical section together. Adding `usleep()` to the wait loop is not a cure either; it only changes the timing and makes the bug rarer. The cure is to put a full barrier between the write `flag[i] = true` and the read `flag[j]`, or to declare the variables as sequentially consistent atomics (`_Atomic`, default `memory_order_seq_cst`).

6.4.4 Peterson's algorithm written correctly in Java

In the Java memory model, reads and writes of `volatile` fields are sequentially consistent. The compiler and the JVM add the barriers needed to preserve the order of these accesses themselves. So Peterson's algorithm works correctly in Java with separate `volatile` fields:

```
public class Peterson {
  static volatile boolean flag0, flag1;
  static volatile int turn;
  static int counter = 0;

  static void lock(int i) {
    if (i == 0) { flag0 = true; turn = 1;
      while (flag1 && turn == 1) Thread.onSpinWait(); }
    else        { flag1 = true; turn = 0;
      while (flag0 && turn == 0) Thread.onSpinWait(); }
  }
  static void unlock(int i) {
    if (i == 0) flag0 = false; else flag1 = false;
  }
  /* main: two threads, i = 0 and i = 1,
     each 100_000 times: lock(i); counter++; unlock(i);
     join both, print counter */
}
```

In the code the two flags are separate fields, and `turn` is also `volatile`. `Thread.onSpinWait()` (Java 9 and later) gives the processor the hint "I am busy-waiting"; on x86 it becomes the `pause` instruction. This instruction reduces the loop's power consumption and the harm it does to the hardware thread sharing the same core. The `main` part of the code in the slides was summarized as a comment; the full runnable version is as follows:

```
public class Peterson {
  static volatile boolean flag0, flag1;
  static volatile int turn;
  static int counter = 0;

  static void lock(int i) {
    if (i == 0) { flag0 = true; turn = 1;
      while (flag1 && turn == 1) Thread.onSpinWait(); }
    else        { flag1 = true; turn = 0;
      while (flag0 && turn == 0) Thread.onSpinWait(); }
  }
  static void unlock(int i) {
    if (i == 0) flag0 = false; else flag1 = false;
  }
  public static void main(String[] a) throws InterruptedException {
    Thread[] t = new Thread[2];
    for (int k = 0; k < 2; k++) {
      final int i = k;
      t[k] = new Thread(() -> {
        for (int n = 0; n < 100_000; n++) { lock(i); counter++; unlock(i); }
      });
    }
    for (Thread x : t) x.start();
    for (Thread x : t) x.join();
    System.out.println("counter = " + counter);
  }
}
```

The program was run three times with JDK 21 and gave this output every time:

```
counter = 200000
counter = 200000
counter = 200000
```

This time the counter is 200,000 on every run: the critical section is protected. The unprotected version of the same counter gave different and smaller values on every run. That this program works correctly under one particular timing is not a proof; the guarantee of correctness is the sequential consistency that the Java memory model gives for `volatile`.

Common mistake

When writing Peterson in Java, three pitfalls are common. The first is the declaration `volatile boolean[] flag`: this makes only the array *reference* volatile, not the elements. Reads and writes of the elements remain ordinary accesses. The second is generating the process number with `getId() % N`; two threads may get the same value of `i` and both use

the same flag. The third is putting `System.out.println` in the critical section. Printing uses its own lock internally and changes the timing so much that the measurement loses its meaning.

6.4.5 The synchronization barrier

The word "barrier" names two different concepts in synchronization. A memory barrier enforces the order of a single core's memory accesses. A **synchronization barrier** makes a group of processes meet at a certain point.

Parallel computations divided into phases need this meeting. In a grid simulation each process computes a slice of the grid. The next phase uses the values of the neighboring slices from this phase. So no process should move on to the next phase before the others have finished this one. A process that has finished the phase calls `barrier_wait()` and blocks. When the N-th process arrives, the barrier opens and all move on to the next phase together (Figure 6.12).

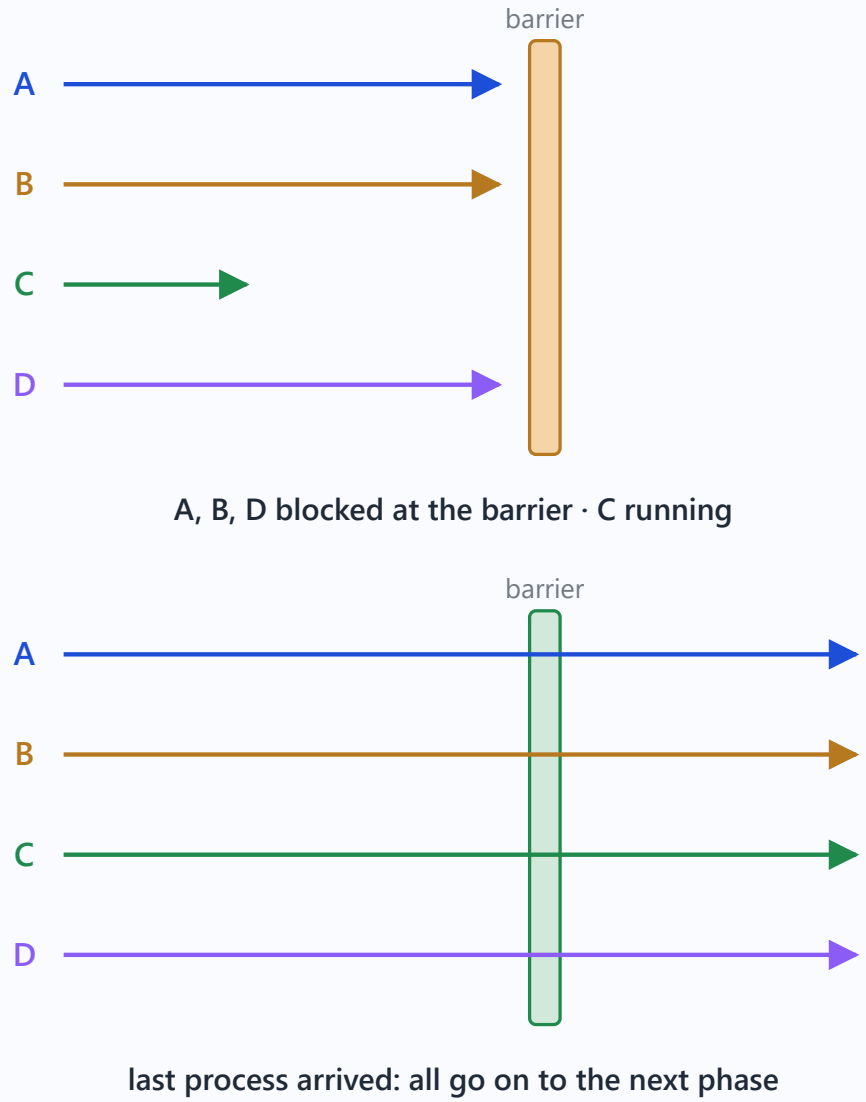


Figure 6.12: The synchronization barrier. (a) A, B, and D have finished their phases and are blocked at the barrier; the slowest process, C, is awaited. (b) When the last process arrives the barrier opens and all move on to the next phase.

In POSIX there is `pthread_barrier_wait` for this job, and in Java `CyclicBarrier`. In the Java fragment below three threads compute two phases. The second argument given to the barrier is an action; it runs once when the last thread arrives, before the barrier opens.

```
var b = new CyclicBarrier(3,
  () -> System.out.println(
          "phase " + phase[0]++ + " done"));
/* 3 threads, each:
   for (f = 0; f < 2; f++) {
     compute(f); b.await();
   } */
```

The full program was run with JDK 21, with each thread sleeping a random time between phases:

```
import java.util.concurrent.CyclicBarrier;
public class Barrier {
  public static void main(String[] a) throws Exception {
    int[] phase = {0};
    var b = new CyclicBarrier(3,
      () -> System.out.println(
              "phase " + phase[0]++ + " done"));
    Thread[] t = new Thread[3];
    for (int k = 0; k < 3; k++) {
      t[k] = new Thread(() -> {
        try { for (int f = 0; f < 2; f++) { Thread.sleep((long) (Math.random() * 20));
b.await(); } }
        catch (Exception e) { }
      });
      t[k].start();
    }
    for (Thread x : t) x.join();
  }
}
```

The output is the same on every run, because the action runs exactly once in each phase and only after all the threads have arrived:

```
phase 0 done
phase 1 done
```

The order in which the threads reach the barrier changes on every run. Because the program's output does not depend on this order, the variability is not visible from outside.

6.5 Hardware support

The common problem of software solutions is that reading a value and writing to it are two separate steps. Anything can happen in between. To close this gap the hardware offers instructions that test and change a memory word without interruption. These instructions are the foundation of all locks, semaphores, and lock-free data structures.

6.5.1 Atomic hardware instructions

Definition 6.8

An **atomic instruction** is a hardware instruction that reads, modifies, and writes a memory word and performs these three steps without interruption. While the instruction runs, no other core can access that word; the effect of the instruction is seen either in full or not at all.

Silberschatz defines the behavior of these instructions with a C-like function. The three lines of the `test_and_set` function below are in reality a single machine instruction:

```
/* runs as ONE atomic instruction */
bool test_and_set(bool *target) {
  bool old = *target;
  *target = true;
  return old;
}

/* usage: a simple spinlock */
bool lock = false;
while (test_and_set(&lock))
  ;            /* busy wait */
/* critical section */
lock = false;
```

The function returns the old value of the target and in every case writes `true` to it. If the lock is free (`false`), the caller gets `false` and the lock is now `true` : the lock has been acquired. If the lock is taken, the caller gets `true` and nothing changes: the loop tries again. The "window between the test and the assignment" of the lock variable has disappeared.

Atomicity holds on multiple cores as well. Old processors used to lock the memory data path for the duration of the instruction. Today the cache-coherence protocol reserves the relevant cache line for a single core for the duration of the instruction; the other cores cannot access that line in the meantime. Table 6.5 gives the counterparts on common architectures.

Architecture	Atomic read-modify-write
x86-64	`XCHG` , `LOCK CMPXCHG` , `LOCK XADD` , `LOCK BTS`
ARMv8	`LDXR` / `STXR` (LL/SC); with v8.1 LSE, `CAS` , `LDADD` , `SWP`
RISC-V "A" extension	`LR` / `SC` , `AMOSWAP` , `AMOADD`
Tanenbaum's virtual processor	`TSL REGISTER,LOCK`

Table 6.5: Atomic read-modify-write instructions by architecture.

The LL/SC pair on ARM and RISC-V reaches the same result by a different route. `LDXR` (load-linked) reads the value and starts monitoring the address. `STXR` (store-conditional) writes only if no other write has intervened; if one has, it fails and the code tries again.

6.5.2 A critical section with TSL

Tanenbaum explains the same idea with an instruction called `TSL` (test and set lock). `TSL REGISTER,LOCK` reads the word `LOCK` into a register and writes 1 to `LOCK`; it does both in a single indivisible step. Entering and leaving the critical section are written with a few lines of assembly:

```
enter_region:
  TSL REGISTER,LOCK  | R = LOCK; LOCK = 1
  CMP REGISTER,#0    | was LOCK 0?
  JNE enter_region   | no: try again
  RET                | entered critical section

leave_region:
  MOVE LOCK,#0       | LOCK = 0
  RET
```

`enter_region` tests the old value of `LOCK`. If the value was 0, the lock was free and now belongs to this core; the function returns. If the value is not 0, the lock is taken and the loop tries again. `leave_region` sets the lock to 0 with an ordinary write. Table 6.6 traces a scenario in which two processors try to enter at the same time.

Step	Event	LOCK	R0	R1
1	lock free	0	–	–
2	CPU 0: `TSL` (old 0 into the register, 1 into LOCK)	1	0	–
3	CPU 1: `TSL` (old 1 into the register)	1	0	1
4	CPU 0: `CMP` → 0, in the critical section; CPU 1: `JNE` → loop	1	0	1
5	CPU 0: `MOVE LOCK,#0`, leaves	0	0	1
6	CPU 1: `TSL` → sees 0, entering	1	0	0

Table 6.6: Trace of a critical section with TSL (R0, R1: the registers of the two processors).

The third step is important. CPU 1's `TSL` also writes 1 to `LOCK`; but since the value is already 1, nothing changes. The 1 that arrives in CPU 1's register is the information "the lock was taken." At no moment are the two processors inside together. TSL works on multiple cores too: unlike disabling interrupts, it also prevents the other processors from accessing that word (Tanenbaum MOS Figure 2-25).

6.5.3 The XCHG instruction

`XCHG` swaps the contents of two operands in a single step. First 1 is put into the register, then the register is swapped with `LOCK`. After the swap `LOCK` is in every case 1 and the register holds the old value of `LOCK` (Figure 6.13). The effect is the same as TSL.

```
enter_region:
  MOVE REGISTER,#1    | put 1 in register
  XCHG REGISTER,LOCK  | swap register and LOCK
  CMP REGISTER,#0     | was LOCK 0?
  JNE enter_region    | no: loop
  RET                 | entered critical section
leave_region:
  MOVE LOCK,#0        | LOCK = 0
  RET
```

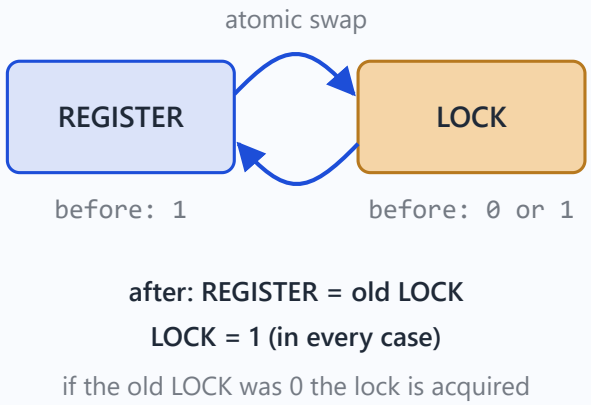


Figure 6.13: Atomic swap with `XCHG`. Afterwards the register holds the old value of `LOCK` and `LOCK` holds 1. If the old value was 0, the lock has been acquired.

This is the basic instruction used for locking on Intel x86 processors. `XCHG` with a memory operand is implicitly treated as `LOCK`-prefixed, that is, it is always atomic. Its counterpart in C11 is the call `atomic_exchange(&lock, 1)`.

6.5.4 A lock with compare_and_swap

The most flexible atomic instruction is **compare-and-swap** (CAS). The instruction takes three operands: an address, an expected value, and a new value. If the value at the address equals the expected one, it writes the new value. In every case it returns the old value at the address.

```
int compare_and_swap(int *v, int exp, int nv) {
  int old = *v;           /* these three lines */
  if (old == exp)         /* are ONE atomic op */
    *v = nv;
  return old;
}

int lock = 0;
void enter(void) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ;                     /* busy wait */
}
void leave(void) { lock = 0; }
```

When used as a lock, the expected value is 0 and the new value is 1. If the lock is free, CAS writes 1 and returns 0: the loop ends. If the lock is taken, CAS writes nothing and returns 1: the loop continues. The trace goes as follows:

1. **The lock is free.** `lock = 0`.
2. **T0 wins.** T0's CAS finds `lock` to be 0, writes 1, and returns 0. T0 leaves the loop and enters the critical section.
3. **T1 loses.** T1's CAS finds 1. This is not the expected value; it writes nothing and returns 1.
4. **T1 waits.** T1 busy-waits; each attempt loops without changing the memory word.
5. **T0 leaves.** T0 sets the lock to 0. In real code this write is done with release ordering.
6. **T1 enters.** T1's next CAS succeeds.

On x86 the counterpart of CAS is `LOCK CMPXCHG`. In C11 `atomic_compare_exchange_strong` and in Java `AtomicInteger.compareAndSet` do the same job (Silberschatz 10e §6.4.2).

Common mistake

Thinking that a simple spinlock built with an atomic instruction satisfies all three requirements. Mutual exclusion is satisfied thanks to atomicity. Progress is satisfied too: when the lock is released, one of the attempts certainly succeeds. But which thread's CAS runs first when the lock is released is undetermined. An unlucky thread can lose every race; **bounded waiting** is not guaranteed.

6.5.5 Atomic increment with CAS

Beyond building locks, CAS also makes it possible to update without using a lock. To increment a shared counter, a thread first reads the value. Then with CAS it says, "if the value is still as I read it, write one more." If someone has changed the value in the meantime, CAS fails and the loop tries again with the current value.

```c
void increment(int *v) {
  int temp;
  do {
    temp = *v;              /* read              */
  } while (temp != compare_and_swap(v, temp, temp + 1));
}                           /* write if unchanged */

/* C11: if the counter is atomic_int, one line: atomic_fetch_add(v, 1); */
```

Example 6.3

Initially `v = 5`. T1 and T2 call `increment(&v)` at the same time. The order is as follows: T1 reads, T2 reads, T2 does CAS, T1 does CAS, T1 retries. What are `v` and the local values at each step?

Step	Event	v	T1.temp	T2.temp
1	T1 reads	5	5	–
2	T2 reads	5	5	5
3	T2: CAS(v, 5, 6) → returns 5, succeeds	6	5	5
4	T1: CAS(v, 5, 6) → returns 6, fails	6	5	–
5	T1 reads again	6	6	–
6	T1: CAS(v, 6, 7) → returns 6, succeeds	7	6	–

Table 6.7: Trace of increment with CAS.

1. **Steps 1–2.** Both threads read 5. With a plain `v++` an increment would be lost here.
2. **Step 3.** T2's CAS finds `v` still 5 and writes 6. The 5 it returns equals `temp`; T2's loop ends.
3. **Step 4.** T1's CAS finds `v` to be 6. The expected value, 5, does not match; CAS does not write and returns 6. Since 6 ≠ 5, the loop repeats.
4. **Steps 5–6.** T1 reads the current value (6) and its CAS succeeds this time: `v = 7`.

No increment was lost and nobody held a lock. When all interleavings of the two threads are explored with a program, the final value is always 7.

This "read → compute → CAS → retry if it fails" loop is the basic pattern of lock-free data structures. If the counter is declared as `atomic_int`, C11 does the same increment in one line: `atomic_fetch_add(v, 1)`. On x86 this is executed with a single `LOCK XADD` instruction without a loop.

6.5.6 A CAS lock with bounded waiting

What the simple CAS lock lacked was bounded waiting. Silberschatz fills this gap with a `waiting[n]` array. Each waiting process raises its own flag. The process that leaves the critical section does not release the lock directly; it first scans the processes in ring order, starting with the one after itself. If it finds the first waiter, it hands the lock over to it.

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* no one waiting */
  else
    waiting[j] = false;  /* hand over turn */
  /* remainder section */
}
```

The subtlest point of the code is the hand-over step. When the leaving process finds a waiting P_j, it does not touch `lock`; it merely sets `waiting[j] = false`. Since P_j's loop condition is `waiting[j] && key == 1`, P_j leaves the loop and enters. During this the lock stays at 1, so no one else can slip in. The lock is really released (`lock = 0`) only when no one is waiting.

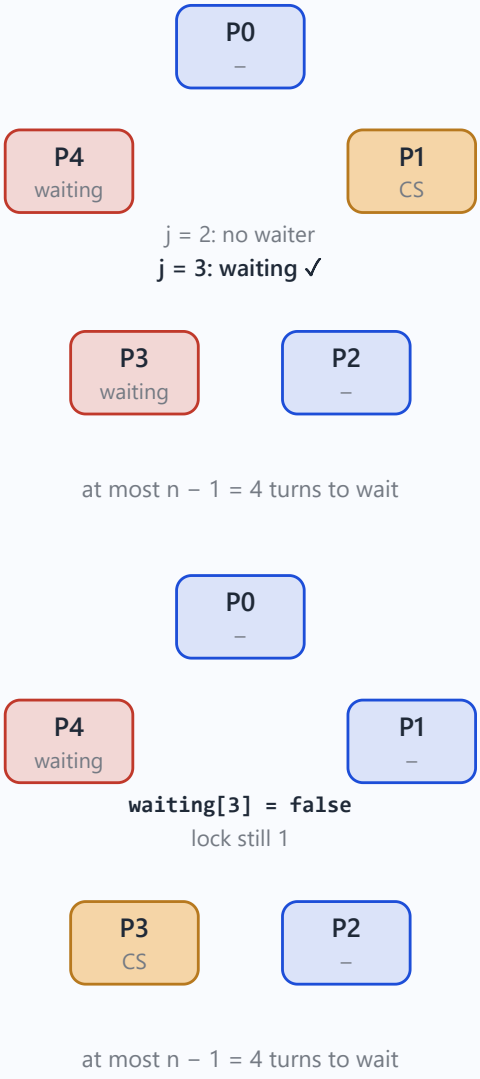


Figure 6.14: The cyclic scan. (a) On leaving, P1 scans in the order `j = 2, 3, …`; the first waiter is P3. (b) P1 does not release the lock; it sets `waiting[3] = false`; P3 enters and `lock` is still 1. With five processes, at most $n - 1 = 4$ turns have to be waited.

The ring scan guarantees bounded waiting. A waiting process waits for at most $n - 1$ entries, those of the processes that will be scanned before it. Table 6.8 traces a scenario with four processes. The line numbers in this code are as follows: 2 raising the flag, 3–4 the waiting loop, 5 lowering the flag, 7–9 the scan, 10–13 releasing or handing over.

```
/* P_i; n = 4 processes */
waiting[i] = true; key = 1;
while (waiting[i] && key == 1)
  key = compare_and_swap(&lock, 0, 1);
waiting[i] = false;
/* critical section */
j = (i + 1) % n;
while ((j != i) && !waiting[j])
  j = (j + 1) % n;
if (j == i)
  lock = 0;            /* no one waiting */
else
  waiting[j] = false;  /* hand over turn */
```

Step	Event	lock	key	j	waiting	CS
1	lock free, no waiters	0	–	–	F F F F	–
2	P2 arrives; CAS returns 0	1	0	–	F F T F	–
3	P2 `waiting[2] = false`, enters	1	0	–	F F F F	P2
4	P0 arrives; CAS returns 1, spins	1	1	–	T F F F	P2
5	P3 arrives and spins	1	1	–	T F F T	P2
6	P2 leaves; the scan stops at `j = 3`	1	–	3	T F F T	P2
7	P2 `waiting[3] = false`; P3 enters	1	–	3	T F F F	P3
8	P3 leaves; `j = (3 + 1) % 4 = 0`	1	–	0	T F F F	P3
9	the turn is handed to P0	1	–	0	F F F F	P0
10	P0 leaves; `j = 1, 2, 3` are empty, `j` back to 0	1	–	0	F F F F	P0
11	`j == i`: no one waiting, `lock = 0`	0	–	–	F F F F	–

Table 6.8: Trace of the CAS lock with bounded waiting ($n = 4$; the waiting array in the order P0 P1 P2 P3, T/F).

The ninth step is interesting. P0 had arrived before P3, but the ring order chose P3 first. This lock does not work in arrival order (FIFO); it merely ensures that there is an upper bound on the wait. Because when P2 leaves the scan starts from the one after itself, `j = 3`, P3 has got ahead. Still, P0 waits for at most $n - 1 = 3$ entries. The only moment the lock is really free is step 11.

6.6 Locks

Atomic instructions are impractical on their own; programmers use them not directly but through tools built on top of them. This section covers the most basic of these tools: atomic variables, spinlocks, and mutexes. It then looks at how data structures that use locks are scaled.

6.6.1 Atomic variables

Languages and libraries offer atomic instructions as **atomic variables**. In C11 every access to a variable of type `_Atomic` is atomic and, by default, sequentially consistent (`seq_cst`). With this type even the expression `counter++` is atomic; it compiles to `lock xadd` on x86 and `ldadd` on ARMv8.1.

```c
#include <stdatomic.h>
atomic_int counter = 0;   /* _Atomic int */

void *work(void *arg) {
  for (int n = 0; n < 100000; n++)
    atomic_fetch_add(&counter, 1);
  return NULL;
}
/* 2 threads + join → 200000 */
```

Java's counterpart is the `java.util.concurrent.atomic` package. `AtomicInteger.incrementAndGet()` is an atomic increment and `compareAndSet(expected, new)` is CAS directly:

```java
static AtomicInteger atom = new AtomicInteger();
/* two threads, each 100_000 times: */
atom.incrementAndGet();

var v = new AtomicInteger(5);
v.compareAndSet(5, 6);   /* true,  v = 6 */
v.compareAndSet(5, 7);   /* false, v = 6 */
```

The fragments in the slides were combined into a single program. The program also increments the same counter with a `synchronized` method:

```
import java.util.concurrent.atomic.AtomicInteger;
public class Atom {
  static AtomicInteger atom = new AtomicInteger();
  static int plain = 0;
  static synchronized void increment() { plain++; }
  public static void main(String[] a) throws InterruptedException {
    Runnable task = () -> { for (int n = 0; n < 100_000; n++) { atom.incrementAndGet();
increment(); } };
    Thread t1 = new Thread(task), t2 = new Thread(task);
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("AtomicInteger = " + atom.get());
    System.out.println("synchronized  = " + plain);
    var v = new AtomicInteger(5);
    System.out.println(v.compareAndSet(5, 6) + " " + v.get());
    System.out.println(v.compareAndSet(5, 7) + " " + v.get());
  }
}
```

When run three times with JDK 21, it gave the same output every time:

```
AtomicInteger = 200000
synchronized  = 200000
true 6
false 6
```

Both methods give the result 200,000. `compareAndSet(5, 6)` succeeds because the value is 5. `compareAndSet(5, 7)` fails because the value is now 6, and `v` stays at 6.

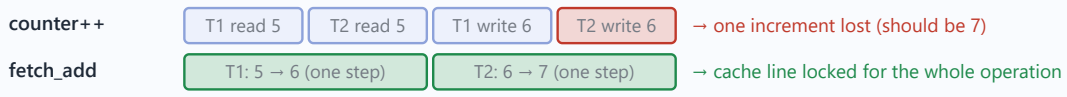


Figure 6.15: Plain `counter++` versus `fetch_add`. In the top row two reads see the same value and one increment is lost. In the bottom row each increment is a single step; the cache line is locked for the duration of the operation.

Common mistake

Believing that atomic variables protect several variables together. An atomic variable makes the update of only *one* variable indivisible. If a buffer array and the number of items in it have to be kept consistent together, making the counter atomic is not enough. The array may have been updated while the counter has not yet been incremented. For consistency across several variables a lock or monitor is still needed.

6.6.2 Spinlocks and mutexes

What a waiting thread does while a lock is taken gives rise to two kinds of locks. A **spinlock** busy-waits: it does not give up the processor and keeps testing the lock. As soon as the lock is released, it enters without a context switch. A **sleeping mutex**, on the other hand, blocks the waiter; the kernel gives the processor to another thread. When the lock is released, the waiter is woken.

Definition 6.9

A **mutex** (mutual exclusion lock) is a binary lock with an owner, used for mutual exclusion. `lock` acquires the lock or makes the caller wait until the lock is free; `unlock` releases the lock and can be called only by the lock's owner.

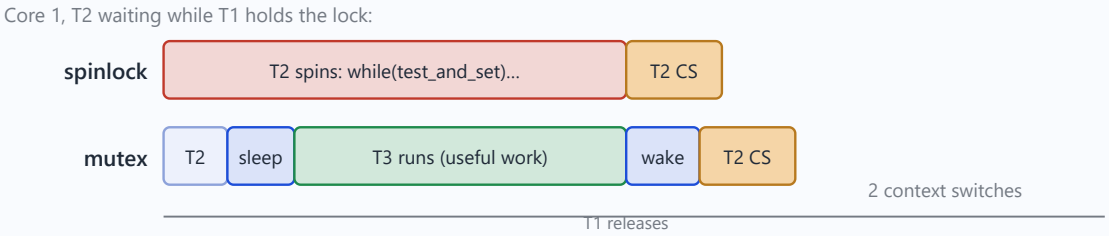


Figure 6.16: Spinlock and sleeping mutex. With a spinlock T2 spins until T1 releases the lock and enters at once. With a mutex T2 sleeps, the processor switches to T3, and useful work gets done; the price is two context switches.

The choice depends on how long the wait will be. If the critical section is shorter than two context switches, spinning is cheaper than sleeping. If the critical section is long or can block inside (for example a disk read), spinning burns the processor for nothing. On a single-core machine a spinlock is meaningless: the thread holding the lock cannot run, because the spinning thread is using the processor. The waiter burns its whole slice, and the lock can be released only at the next context switch.

	Spinlock	Mutex (sleeping)
Suitable	Multicore; critical section shorter than two context switches	Long critical section, or one that can block
Single core	Meaningless: the holder cannot run, the waiter burns its slice	Right choice
Example	Linux `spin_lock`, `pthread_spin_lock`	`pthread_mutex_t`, Java `synchronized`

Table 6.9: Comparison of spinlock and sleeping mutex.

Most real mutex implementations combine the two methods. They first spin for a short time and, if the lock is not released in that time, sleep.

6.6.3 How a mutex is written: yield and futex

Tanenbaum writes a mutex with TSL for user-level threads. The code is very similar to `enter_region` with TSL. Its only difference is that if the lock is taken it calls `thread_yield` instead of looping back:

```
mutex_lock:
  TSL REGISTER,MUTEX | get old, set 1
  CMP REGISTER,#0    | was it free?
  JZE ok             | yes: acquired
  CALL thread_yield  | no: give up CPU
  JMP mutex_lock     | then try again
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

With user-level threads, thread switching is not driven by the timer interrupt. A thread that busy-waits never gives up the processor, and the thread holding the lock can never run. `thread_yield` gives the processor to the other threads in the same process; the owner of the lock can run and release it (Tanenbaum Figure 2-29).

The solution in Linux is the **futex** (fast userspace mutex). The lock is an integer word in user space. If there is no contention, the lock is acquired and released with a single atomic instruction; the kernel is never entered. The kernel steps in only when a thread really needs to sleep. glibc's `pthread_mutex`, Java locks, and Go's concurrency tools are built on top of this mechanism. In the common design in Ulrich Drepper's paper "Futexes Are Tricky," the word has three values: 0 free, 1 locked with no waiters, 2 locked with possible waiters.

1. **No contention.** T1 turns the word from 0 to 1 with a single CAS. No system call is made.
2. **T2 arrives.** T2 finds the lock taken, sets the value to 2 ("there are waiters"), and sleeps in the kernel's wait queue with the `futex_wait` system call.
3. **T1 releases.** T1 sets the word to 0. Since the old value was 2, it calls `futex_wake`.
4. **T2 wakes.** T2 sets the word to 2 again with an atomic exchange (`xchg`). Since the old value returned by the exchange is 0, the lock now belongs to T2. T2 cannot know whether there are other waiters; so it cautiously sets the value to 2, not 1.

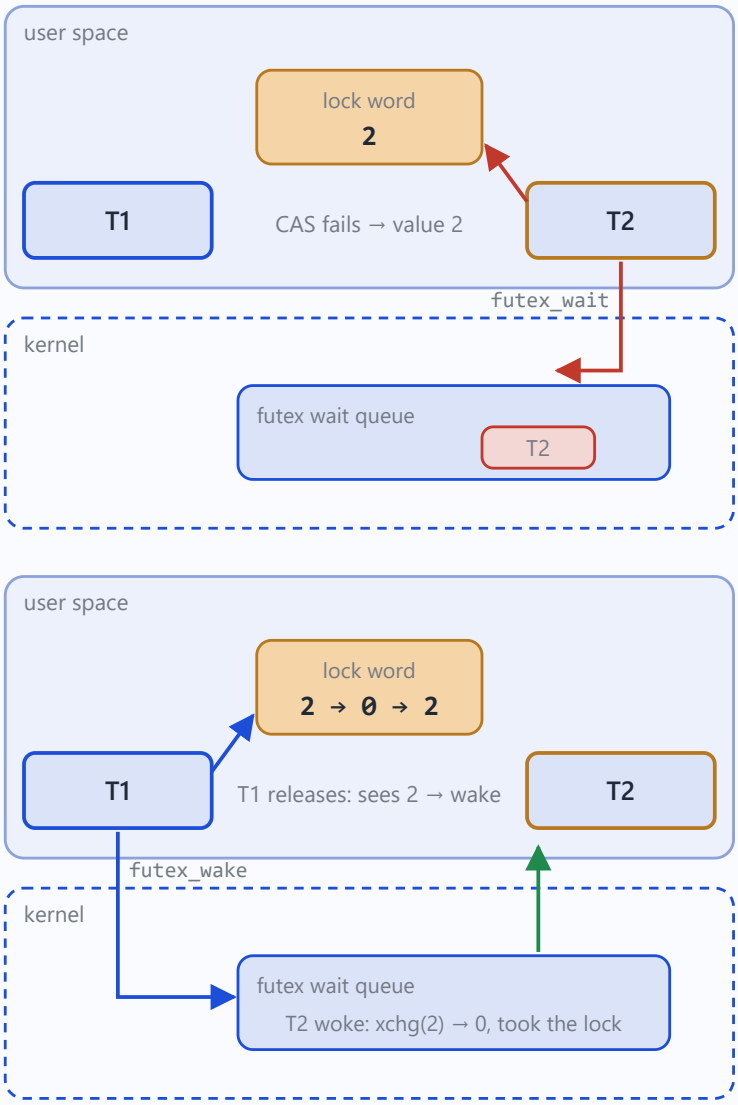


Figure 6.17: A mutex with futex. (a) T2 finds the lock taken, sets the word to 2, and sleeps in the futex queue in the kernel. (b) When T1 releases, it sees 2, sets the word to 0, and calls `futex_wake`; the woken T2 sets the word to 2 again and takes the lock.

The reason for separating the values 1 and 2 is to avoid unnecessary `futex_wake` calls. If the value is 1, nobody is sleeping and the releasing thread never enters the kernel. The woken thread's setting the value to 2 sometimes leads to an unnecessary `futex_wake` call; but it ensures that no waiter is forgotten. Thus in the uncontended case acquiring and releasing each consist of a single atomic instruction.

6.6.4 The scalable counter

A counter protected by a single lock is correct but does not scale. Every increment competes for the same lock, and the same cache line is moved between cores continually. As the number of cores grows, the total time does not shrink; it grows. The **approximate counter** of the Arpaci-Dusseaus

solves this problem with per-core counting. Each core has its own local counter and its own lock. When a local counter reaches a threshold `S`, its value is transferred to the global counter.

```
typedef struct {
  int global;      pthread_mutex_t glock;
  int local[NCPU]; pthread_mutex_t llock[NCPU];
  int S;                     /* transfer threshold */
} acount_t;

void add(acount_t *c, int cpu) {
  pthread_mutex_lock(&c->llock[cpu]);
  c->local[cpu]++;
  if (c->local[cpu] >= c->S) {        /* transfer */
    pthread_mutex_lock(&c->glock);
    c->global += c->local[cpu];
    pthread_mutex_unlock(&c->glock);
    c->local[cpu] = 0;
  }
  pthread_mutex_unlock(&c->llock[cpu]);
}
int get(acount_t *c) {             /* approximate */
  pthread_mutex_lock(&c->glock);
  int v = c->global;
  pthread_mutex_unlock(&c->glock);
  return v;
}
```

`add` takes only the local lock of the calling core. Because different cores use different locks, they do not wait for each other. The global lock is taken only during a transfer, that is, once every `S` increments. `get` returns only the global counter; so the result is *approximate*.

Example 6.4

There are four cores and `S = 4`. The cores call `add` in the order below. After each step, what are the local counters (L1–L4), the global counter G, and the true total?

t	Cores incrementing	L1	L2	L3	L4	G	true
0	–	0	0	0	0	0	0
1	1, 3	1	0	1	0	0	2
2	1, 2, 3	2	1	2	0	0	5
3	1, 3, 4	3	1	3	1	0	8
4	1, 2	4 → 0	2	3	1	4	10
5	3, 4	0	2	4 → 0	2	8	12

Table 6.10: Trace of the approximate counter (S = 4).

1. **t = 1–3.** Eight increments occur, but no local counter reaches 4. The global lock is never touched; G = 0.
2. **t = 4.** L1 reaches the threshold. Core 1 takes the global lock, adds 4 to G, and resets its local counter.
3. **t = 5.** L3 also reaches the threshold and transfers: G = 8.
4. **Reading.** `get()` returns 8. The true value is 12; the difference of 4 sits in L2 and L4.

In the worst case every local counter stops at `S - 1` and has never been transferred. So `get()` can lag the true value by at most NCPU × (S − 1); in the example it is 4 × 3 = 12. `S` sets the trade-off between accuracy and scalability. A large `S` goes to the global lock rarely and scales well, but the value read lags further behind. With `S = 1` every increment goes to the global counter: the result is exact but the counter is as slow as a single-lock counter. The `percpu_counter` in the Linux kernel is based on the same idea.

6.6.5 Lock ordering in concurrent data structures

The easiest way to make a data structure thread-safe is to wrap every operation in a single lock. This is correct but serializes all operations. Splitting the lock can increase parallelism; Table 6.11 summarizes common layouts.

Structure	Lock layout	Result
Linked list	Single mutex	Simple. `malloc` should be done outside the lock; `unlock` must not be forgotten on any return path.
Hand-over-hand locking	A lock per node: take the next one first, then release the previous one	Traversals proceed in parallel; but the cost of the lock on each node often eats up the gain.
Queue	Separate locks for head and tail, a dummy node	Enqueue and dequeue can happen at the same time.
Hash table	A locked list per bucket	Operations going to different buckets do not wait for one another; scales very well.

Table 6.11: Lock layouts in concurrent data structures.

The two-lock queue published by Michael and Scott in 1996 is a fine example of splitting a lock. There is always a **dummy node** at the head of the queue. Initially both the head and the tail point to this dummy. `enqueue` takes only `tail_lock` and advances the tail. `dequeue` takes only `head_lock`: it frees the old dummy, returns the value of the first real node, and that node becomes the new dummy (Figure 6.18). Thanks to the dummy node the two operations never modify the same node.

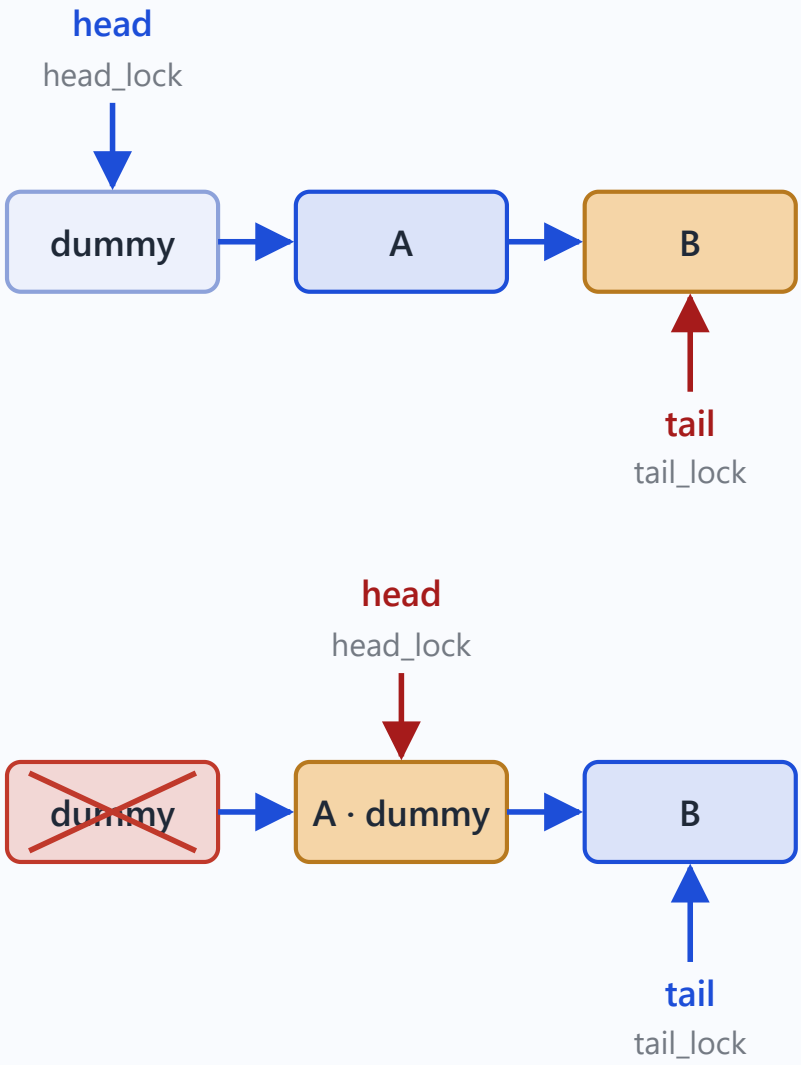


Figure 6.18: The two-lock queue. (a) `enqueue(A)` and `enqueue(B)` take only `tail_lock`. (b) `dequeue()` takes only `head_lock`; the old dummy is freed, A's value is returned, and A becomes the new dummy.

The practical rule is this: write the code correctly with a single lock first. If measurement shows that the lock is really a bottleneck, the lock is split. More locks does not always mean faster. Every lock brings an extra atomic instruction and a risk of deadlock.

6.7 Semaphores

A mutex says only "let one in." Many coordination problems ask for more: taking one of N identical resources, or waiting until an event occurs. Dijkstra's semaphore meets both needs with a single tool.

6.7.1 The semaphore

Definition 6.10

A **semaphore** is an integer that is accessed only through two atomic operations (Dijkstra, 1965):

- `wait(S)`, also P or `down`: if the value is 0, the calling process blocks (sleeps); otherwise the value is decremented by 1 and the process continues.
- `signal(S)`, also V or `up`: the value is incremented by 1; if some process is waiting, one is woken.

The value of a semaphore can be thought of as the number of available resources. If the value is 3, three more processes can pass without waiting. The letters P and V come from Dijkstra's Dutch words; Tanenbaum uses the names `down` and `up`, and Silberschatz `wait` and `signal`.

`wait` and `signal` are themselves short critical sections: they read and change the value. The operating system protects these two operations by disabling interrupts on a single core and with a spinlock on multiple cores. Busy waiting does not disappear completely this way; but it shrinks from the application's long critical section to a few instructions of the semaphore operation. In real systems a process that calls `down` when the value is 0 sleeps. The busy-waiting version seen in textbooks is only the first, conceptual definition.

6.7.2 A semaphore with a wait queue

In Silberschatz's implementation a semaphore consists of a value and a list of waiting processes:

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

```
void wait(semaphore *S) {      /* P / down */
  S->value--;
  if (S->value < 0) {
    add(S->list, this_proc);
    sleep();                    /* block */
  }
}
void signal(semaphore *S) {    /* V / up */
  S->value++;
  if (S->value <= 0) {
    P = remove(S->list);        /* FIFO */
    wakeup(P);
  }
}
```

In this implementation `wait` first decrements the value and then tests it. So the value can be negative. If the value is $-k$, k processes are waiting in the queue (Figure 6.19). `signal` increments the value; if the result is still 0 or negative, someone is waiting in the queue and the one at the head is woken.

Figure 6.19: A semaphore with value −2. The absolute value is the number of waiting processes: T2 and T3 sleep in the list.

Example 6.5

There is a single resource and the semaphore starts at 1 (a binary semaphore, working like a mutex). T1, T2, and T3 do `wait` in turn, then `signal` in the same order.

Event	S.value	S.list	In the critical section
initial	1	empty	–
T1 `wait`	0	empty	T1
T2 `wait`	−1	T2	T1
T3 `wait`	−2	T2, T3	T1
T1 `signal`	−1	T3	T2
T2 `signal`	0	empty	T3
T3 `signal`	1	empty	–

Table 6.12: Trace of a semaphore with a wait queue (Silberschatz's implementation).

1. **T1 wait.** The value becomes 0. Since it is not negative, T1 continues.
2. **T2 and T3 wait.** The value becomes −1 and then −2; both are added to the queue and sleep.
3. **T1 signal.** The value becomes −1. Since it is still ≤ 0, T2 at the head of the queue is woken and enters the critical section.
4. **T2 signal.** The value becomes 0; T3 is woken.
5. **T3 signal.** The value becomes 1 and nobody is waiting.

The order of the queue is not specified in the definition of a semaphore. If a FIFO queue is used, every waiter eventually gets its turn and the wait is bounded. Another order (for example LIFO or by priority) can lead to starvation. In Tanenbaum's definition the value is never negative: if someone is asleep, `up` wakes one directly without incrementing the value. The two definitions give the same behavior from outside; only the reading of the internal value differs.

6.7.3 Usage patterns

The initial value of a semaphore determines what it is used for:

- **Binary semaphore (like a mutex).** The initial value is 1. Each process does `wait` ... critical section ... `signal`.
- **Counting semaphore.** The initial value is N and it represents N identical resources: the connections of a connection pool or the empty slots of a buffer. For example, if at most 5 connections can be opened to a database, the semaphore starts at 5; the sixth request blocks.
- **Signaling or ordering.** The initial value is 0 and it is used to wait until an event happens. If P2's statement S2 must run after P1's statement S1, P1 does `signal(synch)` after S1 and P2 does `wait(synch)` before S2 (Figure 6.20).

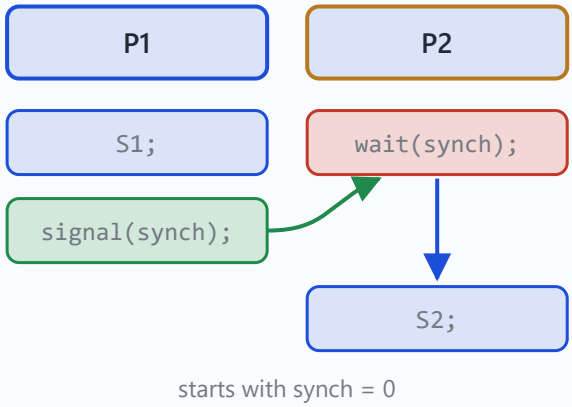


Figure 6.20: Ordering with a semaphore that starts at 0. P2's `wait(synch)` waits until P1 does `signal(synch)`; so S2 runs after S1.

In the ordering pattern it is no problem if the event occurs first. If P1 does `signal` first, the value becomes 1; when P2 then does `wait`, it passes without waiting. A semaphore stores the wakeup as a counter, and no signal is lost. This property is the most important difference that separates a semaphore from a condition variable.

The semaphore is a powerful but unstructured tool: correctness depends on writing every `wait` / `signal` pair correctly. Table 6.13 lists typical errors. A single forgotten line creates a bug that the compiler never warns about and that is rarely seen in testing.

Code	Result
`signal(m)` … CS … `wait(m)`	Reversed order: several processes enter the critical section.
`wait(m)` … CS … `wait(m)`	The process waits for itself: it blocks permanently.
`wait` or `signal` is forgotten	Exclusion breaks or deadlock occurs.
Two semaphores are taken in different orders	Deadlock.

Table 6.13: Common mistakes with semaphores.

6.7.4 Code trace: a barrier with semaphores

A synchronization barrier can be built with two semaphores and a counter. `mutex` protects the counter. `turnstile` starts at 0, that is, closed. The last thread to arrive opens the turnstile. Each thread that passes through the turnstile lets the one behind it through:

```
#define N 3
semaphore mutex = 1, turnstile = 0;
int count = 0;
void barrier(void) {
  down(&mutex);
  count = count + 1;
  if (count == N) up(&turnstile); /* last */
  up(&mutex);
  down(&turnstile);  /* pass through */
  up(&turnstile);    /* let next one in */
}
```

The following trace uses Tanenbaum's semaphore semantics: if someone is asleep, `up` wakes it without incrementing the value. Three threads A, B, and C arrive at the barrier in turn.

Step	Event	mutex	turnstile	count	asleep at the turnstile
1	all three computing	1	0	0	–
2	A arrives, takes the mutex, `count = 1`	0	0	1	–
3	1 ≠ 3: A releases the mutex, sleeps at the turnstile	1	0	1	A
4	B arrives, `count = 2`, sleeps at the turnstile	1	0	2	A, B
5	C arrives, `count = 3 = N`	0	0	3	A, B

Step	Event	mutex	turnstile	count	asleep at the turnstile
6	C `up(&turnstile)` : A wakes	0	0	3	B
7	C releases the mutex, sleeps at the turnstile	1	0	3	B, C
8	A passes, wakes B with `up`	1	0	3	C
9	B passes, wakes C with `up`	1	0	3	–
10	C passes, `up` : no one waiting	1	1	3	–

Table 6.14: Trace of a barrier with semaphores (N = 3).

In step six C opens the turnstile but, because someone is waiting, the value does not increase; A is woken. C then sleeps at the turnstile right behind it (step 7), because it can pass through the turnstile only if someone lets it through. The turnstile works like a one-person revolving door: each one who passes lets the next one through. The last one through, C's `up`, finds no one and the value becomes 1.

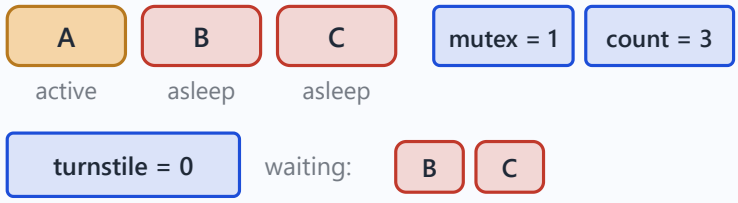


Figure 6.21: Step 7 of the barrier trace. A has been woken and is active; B and C sleep at the turnstile. `count = 3`, the turnstile is still 0.

All interleavings of the three threads were explored with a program. In every execution the threads passed through the turnstile only after `count = 3`, and no execution deadlocked. This barrier is single-use: in the end the turnstile stays at 1 and the counter is 3. To reuse it in a loop the turnstile must be closed again. A reusable version needs two turnstiles; Downey's book works this version out in detail.

Common mistake

Thinking that the `up(&turnstile)` line after passing through the turnstile is unnecessary. The last arriving thread's `up` wakes only a single sleeper. If each thread that passes does not let the one behind it through, only one gets through the turnstile and the other two sleep forever.

6.8 Classic problems

Synchronization tools are tested with a few classic problems that model many situations in real systems. When a new coordination mechanism is proposed, one looks at how cleanly it can solve these problems.

6.8.1 The producer–consumer problem

Two processes share a fixed-size buffer (N slots). The **producer** writes items into the buffer and the **consumer** reads items from it. If the buffer is full the producer must wait, and if it is empty the consumer must wait. This arrangement is called a **bounded buffer**. If the buffer were unbounded, only the consumer would wait. The buffer is a circular array: `in` points to the slot the producer will write, `out` to the slot the consumer will read, and both advance with `(i + 1) % N`.

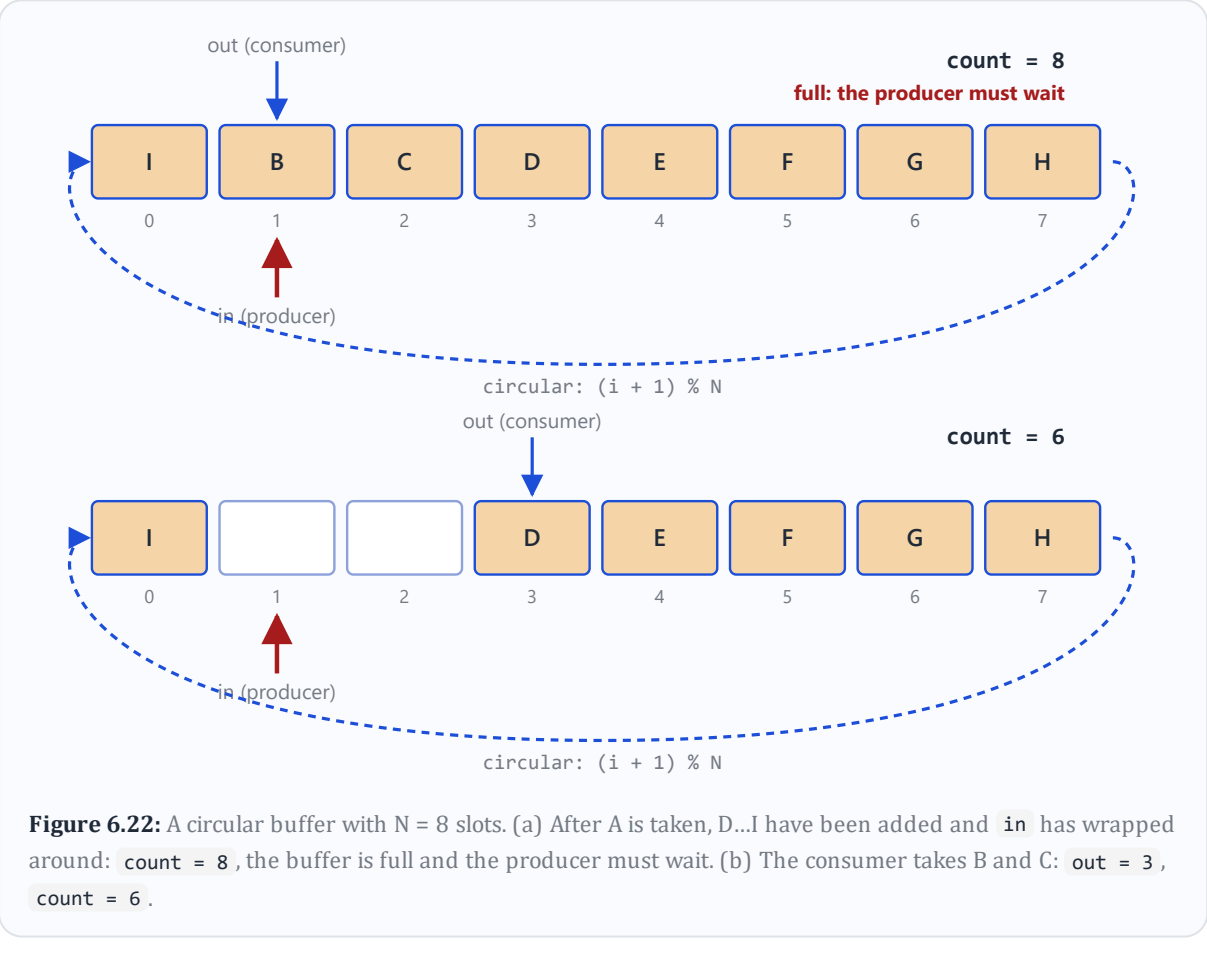


Figure 6.22: A circular buffer with N = 8 slots. (a) After A is taken, D…I have been added and `in` has wrapped around: `count = 8`, the buffer is full and the producer must wait. (b) The consumer takes B and C: `out = 3`, `count = 6`.

In the example the producer first puts A, B, and C into slots 0, 1, and 2; `in = 3`. The consumer takes A: `out = 1`, `count = 2`. The producer puts items D…I into slots 3–7 and 0; `in` wraps around to 1 and `count = 8 = N`. When the consumer takes B and C, `out = 3` and `count = 6`, and the producer can write again.

6.8.2 A fatal race: the lost wakeup

The first solution that comes to mind uses a shared `count` counter and the `sleep` and `wakeup` calls. `sleep()` blocks the caller; `wakeup(p)` makes the sleeping `p` ready. Tanenbaum's code is as follows:

```
#define N 100
int count = 0;
void producer(void) {
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
void consumer(void) {
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

The code looks innocent but contains a fatal race. In the following order both processes sleep forever:

1. The buffer is empty. The consumer reads `count`: 0. It is interrupted just as it is about to call `sleep()`.
2. The producer produces an item and puts it in the buffer; `count` becomes 1.
3. Because `count == 1`, the producer assumes the consumer is asleep and sends `wakeup(consumer)`. Since the consumer has not yet gone to sleep, the signal is *lost*.
4. The consumer continues. Following its old decision, it calls `sleep()` and sleeps.
5. The producer fills the buffer; since the consumer is asleep, nobody reads. When `count == N` the producer sleeps too.

The problem is that the test of `count` and the call to `sleep` are not indivisible. A wakeup that arrives in between is lost without a trace, since no one is asleep. A semaphore solves this problem: because it stores the wakeup as a counter, an `up` that comes first lets the following `down` pass without waiting (Tanenbaum MOS Figure 2-27).

6.8.3 A bounded buffer with semaphores

The correct solution uses three semaphores:

```
#define N 100
semaphore mutex = 1;  /* buffer access      */
semaphore empty = N;  /* empty slots        */
semaphore full  = 0;  /* full slots         */

void producer(void) {
  while (true) {
    item = produce_item();
    down(&empty);   /* wait for a slot  */
    down(&mutex);   /* enter CS         */
    insert_item(item);
    up(&mutex);     /* leave CS         */
    up(&full);      /* one more full    */
  }
}
```

```
void consumer(void) {
  while (true) {
    down(&full);    /* wait for an item */
    down(&mutex);
    item = remove_item();
    up(&mutex);
    up(&empty);     /* one more empty   */
    consume_item(item);
  }
}
```

- `mutex = 1` provides mutual exclusion on buffer access. Had it started at 0, no one could ever get in.
- `empty = N` counts the empty slots. When the buffer is full, the producer blocks at `down(&empty)`.
- `full = 0` counts the full slots. When the buffer is empty, the consumer blocks at `down(&full)`.

`mutex` does exclusion; `empty` and `full` do ordering (synchronization). The two jobs are done with separate semaphores.

Example 6.6

N = 2 and the buffer is empty. The order is as follows: the consumer (C) tries to take an item; the producer (P) puts A and B; P tries for C; C takes an item; P puts C; C takes one more item. The semaphores have Tanenbaum's semantics: if someone is asleep, `up` wakes one and the value does not change.

Event	mutex	empty	full	Buffer	Asleep
initial	1	2	0	–	–
C: `down(&full)`, full = 0 → sleeps	1	2	0	–	C

Event	mutex	empty	full	Buffer	Asleep
P: puts A; `up(&full)` wakes C	1	1	0	A	–
P: puts B	1	0	1	A B	–
P: `down(&empty)` for C → sleeps	1	0	1	A B	P
C: takes A; `up(&empty)` wakes P	1	0	1	B	–
P: puts C	1	0	2	B C	–
C: takes B	1	1	1	C	–

Table 6.15: Trace of the bounded buffer with semaphores (N = 2).

1. **The consumer runs first.** Since `full = 0`, it sleeps. There is no busy waiting.
2. **A is put.** `up(&full)` wakes the sleeping consumer. Because the woken process uses this "1," `full` stays at 0.
3. **B is put.** The consumer has not run yet; `empty = 0` and `full = 1`.
4. **No room for C.** The producer sleeps; the buffer does not overflow.
5. **A is taken.** The consumer's `up(&empty)` wakes the sleeping producer; `empty` stays at 0.
6. **Final state.** C is put and B is taken. The buffer holds C: `mutex = 1`, `empty = 1`, `full = 1`.

When no operation is in progress, `empty + full = N` always holds; in the middle of an operation the difference is the item "in transit."

All interleavings of a four-item scenario were explored with a program: the items were always taken in the order A, B, C, D, the buffer never overflowed, and there was no deadlock.

6.8.4 If the order of down changes

The correctness of the solution depends on the order of the two `down` calls. What happens if the order is reversed in the producer?

```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← first */
    down(&empty);  /* ← then  */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```

1. The buffer is completely full: `empty = 0`.

2. The producer first takes `mutex` and then sleeps at `down(&empty)`. It still holds the mutex.
3. The consumer passes `down(&full)` but sleeps at `down(&mutex)`. Yet it was the one that would have done `up(&empty)`.
4. Everyone waits for the other: **deadlock**.

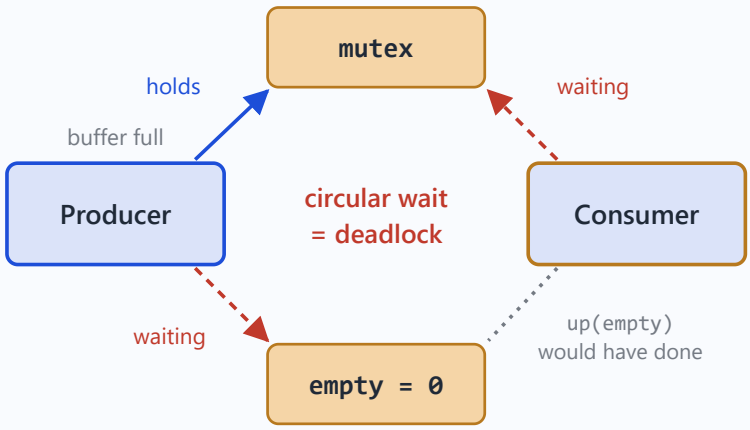


Figure 6.23: The circular wait created by the reversed `down`. The producer holds the mutex and waits for `empty`; the consumer, having passed `full`, waits for the mutex.

A state-space scan done with a program also finds a deadlocked state: `mutex = 0`, `empty = 0`, and two items in the buffer. The rule is this: wait first for the counting semaphore and then for the mutex; never sleep while holding the mutex. The order of the `up` calls, on the other hand, does not matter, because `up` never blocks.

6.8.5 The readers–writers problem

Many processes share a database. Readers only read the data; writers modify it. Readers do not have to exclude one another: ten readers can read at the same time. But while a writer is writing, neither another writer nor a reader should be inside. Tanenbaum's solution uses two semaphores and a counter:

```
semaphore mutex = 1;  /* guards rc          */
semaphore db = 1;     /* guards database    */
int rc = 0;           /* reader count       */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* first reader */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* last reader  */
    up(&mutex);
    use_data_read();
  }
}
```

`rc` is the number of readers inside, and `mutex` protects it. `db` protects the database. A writer takes `db` directly. Of the readers, only the *first* to enter takes `db` (`++rc == 1`) and only the *last* to leave releases it (`--rc == 0`). The readers in between come and go without ever touching `db`.

The solution provides exclusion correctly. When all interleavings of two readers and a writer were explored, no state arose in which the writer and a reader were inside together; the two readers, however, could read together. But the solution has a serious weakness (Figure 6.24):

1. R1 enters and takes `db`.
2. A writer arrives and waits at `down(&db)`.
3. A new reader R2 enters without touching `db`: `rc = 2`.
4. While R1 leaves, R3 arrives. `rc` drops to 1 and rises to 2 again; it never reaches 0.
5. As long as the stream of readers continues in overlapping fashion, `up(&db)` is never called and the writer waits indefinitely.

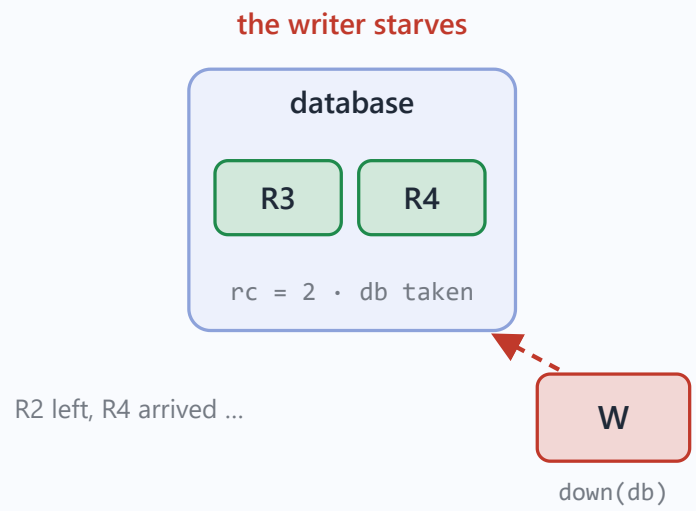


Figure 6.24: Writer starvation in the reader-priority solution. As readers arrive one after another, `rc` never becomes 0 and the writer waits at `down(db)`.

This solution is **reader-priority** and is open to writer starvation. The variants of the problem choose different priorities (Table 6.16). In Downey's starvation-free solution everyone who arrives first passes through a turnstile. A waiting writer holds the turnstile, and no new reader can enter behind it:

```
/* Downey: starvation-free */
/* writer */
wait(turnstile);
wait(roomEmpty);
  write();
signal(turnstile);
signal(roomEmpty);

/* reader */
wait(turnstile); signal(turnstile);
/* first reader takes roomEmpty,
   last reader releases it (via rc) */
  read();
```

Variant	Behavior
First readers–writers (reader priority)	A writer enters only when there are no readers; writers may starve.
Second readers–writers (writer priority)	If a writer is waiting, no new reader can enter; readers may starve.
Fair (turnstile)	Everyone who arrives first passes through the same turnstile; a waiting writer holds the turnstile.
`pthread_rwlock_t`	glibc's default is reader priority; writer priority can be configured.

Variant	Behavior
Java `ReentrantReadWriteLock`	`new ReentrantReadWriteLock(true)` gives a fair, near-FIFO order.
Linux seqlock, RCU	Read-mostly kernel structures in which readers never take a lock.

Table 6.16: Readers–writers variants and their counterparts in real systems.

6.8.6 The dining philosophers

Dijkstra posed the following problem in 1965. Five philosophers sit at a round table; there are five forks between them. Each philosopher thinks, and when hungry picks up the two forks on the right and the left and eats. The problem is a model of sharing limited resources among processes without deadlock and without starvation. The naive solution protects each fork with a semaphore:

```
semaphore fork[5];         /* all 1 */

void philosopher(int i) {
  while (true) {
    think();
    wait(&fork[i]);              /* left */
    wait(&fork[(i + 1) % 5]);    /* right */
    eat();
    signal(&fork[(i + 1) % 5]);
    signal(&fork[i]);
  }
}
```

If all become hungry at the same moment and take their left fork, every philosopher waits for the right fork and no one releases: deadlock (Figure 6.25). Even though each fork is protected by a semaphore, the solution is wrong. When all interleavings of the five philosophers were explored, exactly this state was found: all five have passed the first `wait` and all fork semaphores are 0.

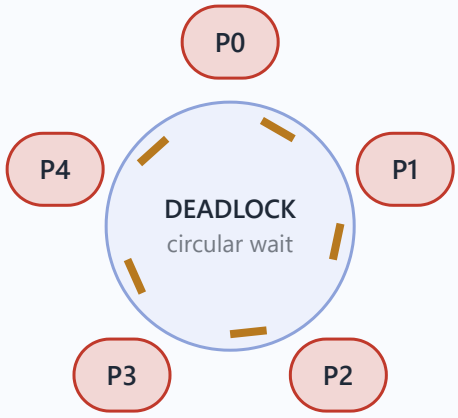


Figure 6.25: Deadlock in the naive solution. Each philosopher holds the left fork and waits for the right fork; a circular wait forms.

There are four ways to a solution:

- **At most four at the table.** A room semaphore starting at 4 lets at most four philosophers try to pick up forks at the same time. When five forks are distributed among four people, at least one reaches two forks and the circular wait is broken.
- **Resource ordering (asymmetry).** Each philosopher first takes the fork with the smaller number. P4's forks are 4 and 0; P4 asks for 0 first. The general rule: locks are always taken in the same global order.
- **Taking both forks together.** Tanenbaum's solution uses a state array and one semaphore per philosopher. A philosopher starts eating only if both neighbors are not eating.
- **A monitor.** Silberschatz's solution uses the procedures `pickup(i)` and `putdown(i)` and one condition variable per philosopher (`self[i]`); its logic is the same as Tanenbaum's.

Tanenbaum's solution is below. Each philosopher's state can be `THINKING`, `HUNGRY`, or `EATING`. The `s[i]` semaphores start at 0; `LEFT` and `RIGHT` are the numbers of the neighbors.

```
void take_forks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]);    /* block if no forks */
}
void put_forks(int i) {
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);     /* (i+4) % 5 */
  test(RIGHT);    /* (i+1) % 5 */
  up(&mutex);
}
void test(int i) {
  if (state[i] == HUNGRY && state[LEFT] != EATING
                         && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

`take_forks` sets the state to `HUNGRY` and calls `test(i)`. If the neighbors are not eating, `test` sets the state to `EATING` and gives the philosopher permission with `up(&s[i])`; the following `down(&s[i])` passes at once. If one of the neighbors is eating, permission is not given and the philosopher sleeps at `down(&s[i])`. `put_forks` sets the state to `THINKING` and tests the two neighbors: if a waiting neighbor can now eat, it wakes it.

Example 6.7

Initially everyone thinks (the state array is TTTTT; T is thinking, H hungry, E eating). P1, P2, and P3 become hungry in turn; then P1, followed by P3, finishes eating.

1. **P1 becomes hungry.** `test(1)`: P0 and P2 are not eating → P1 becomes `EATING`, `up(&s[1])`; `down(&s[1])` passes at once. State: TETTT.
2. **P2 becomes hungry.** The left neighbor P1 is eating → the test fails; P2 sleeps at `down(&s[2])`. State: TEHTT.
3. **P3 becomes hungry.** P2 is hungry but not eating, and P4 is thinking → P3 eats. State: TEHET. Two philosophers can eat at the same time.
4. **P1 puts down the forks.** `test(0)` and `test(2)` are done. `test(2)` fails, because P3 on the right is still eating. State: TTHET.
5. **P3 puts down the forks.** This time `test(2)` succeeds: P2 becomes `EATING` and `up(&s[2])` wakes it. State: TTETT.

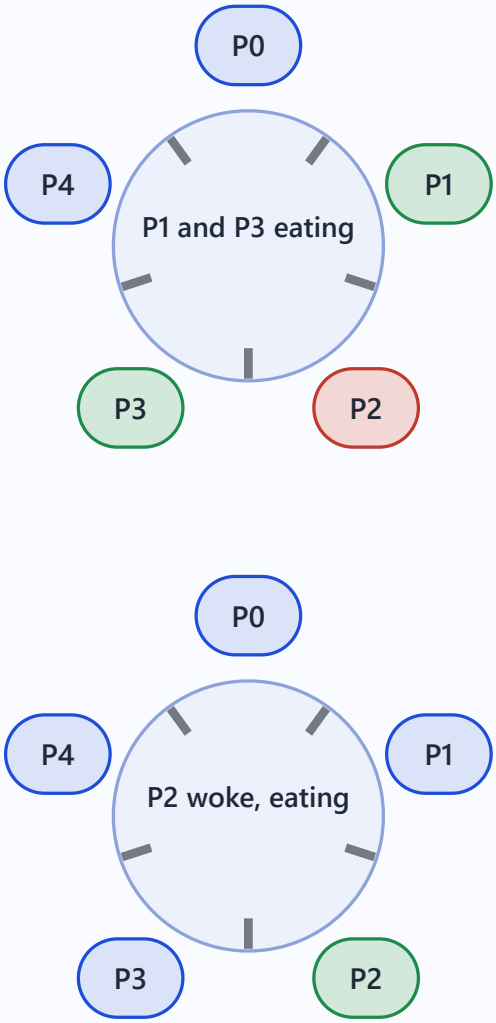


Figure 6.26: Two moments from Tanenbaum's solution. (a) P1 and P3 eat, and P2 is blocked and waits. (b) When P3 puts down its forks, P2 wakes and eats (Tanenbaum MOS Figure 2-47).

All interleavings of the five philosophers were explored with a program. In no execution was there a deadlock, two neighbors never ate together, and at most two philosophers ate at the same time. The number two is not random: each eating philosopher holds two forks, and two philosophers side by side cannot eat at the same time. At a table with n philosophers at most $\lfloor n/2 \rfloor$ can eat at once; for five it is 2, for seven 3.

Common mistake

Thinking that a deadlock-free solution is also starvation-free. In Tanenbaum's solution there is no deadlock. But if P1 and P3 keep taking turns eating, one of the two is always eating and P2's `test` never succeeds. P2 can in theory stay hungry forever. An additional rule is needed to prevent starvation: a ticket number, or raising priority with waiting time (aging).

6.9 Monitors

The correctness of code written with semaphores depends on every `wait` / `signal` pair, and even swapping two `down` calls creates a deadlock. The monitor was developed to reduce this class of errors at the programming-language level.

6.9.1 The monitor

> **Definition 6.11**
>
> A **monitor** is a language-level abstract data type that keeps the shared data, the procedures that operate on that data, and the initialization code together. Only one process can be active inside the monitor at a time (Hoare, 1974; Brinch Hansen, 1975).

```
monitor example
  integer i;
  condition c;
  procedure producer(); … end;
  procedure consumer(); … end;
end monitor;
```

Processes can call the monitor's procedures but cannot access its internal data directly. Mutual exclusion is provided not by the programmer but by the compiler: the compiler adds the entry lock at the start and end of each procedure. The programmer cannot forget to take or release the lock. If the monitor is occupied, the calling process waits in an **entry queue** (Figure 6.27).

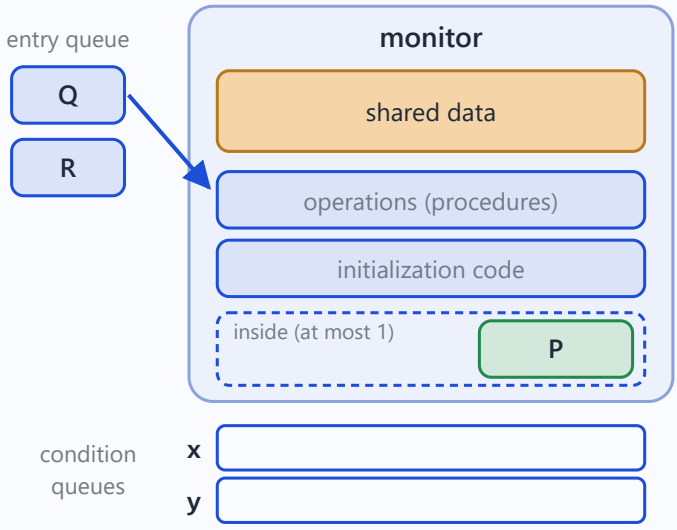


Figure 6.27: The structure of a monitor. At most one process (P) is active inside; Q and R wait in the entry queue. The condition variables `x` and `y` have queues of their own.

6.9.2 Condition variables

Exclusion is not enough; a process inside the monitor sometimes has to wait for a condition to hold. For example, the consumer has found the buffer empty. If it waits while holding the monitor, the producer can never get in. Monitors therefore offer **condition variables**.

Definition 6.12

A **condition variable** is a wait queue used to wait for a condition to occur. `x.wait()` suspends the caller and releases the monitor. `x.signal()` resumes one process waiting on `x` (if any). `broadcast` or `notifyAll` wakes all the waiters.

1. P is in the monitor; it cannot continue because the buffer is empty. Q is in the entry queue.
2. P calls `x.wait()`: it releases the monitor and sleeps in the `x` queue. Q can thus enter.
3. Q makes the condition true and calls `x.signal()`; a process wakes from the `x` queue. Who runs immediately is a semantic choice.
4. When no one is in the `x` queue, `x.signal()` has no effect and leaves no trace.

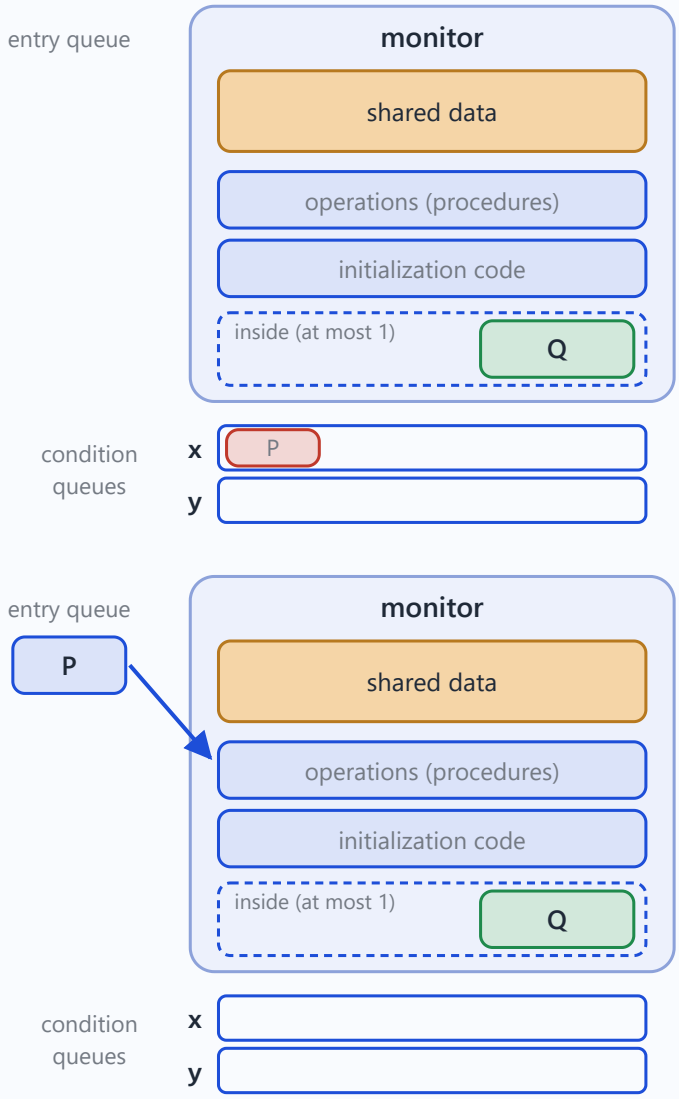


Figure 6.28: A condition variable. (a) P releases the monitor with `x.wait()` and sleeps in the `x` queue; Q goes in. (b) Q calls `x.signal()` and P wakes.

The last step is the basic difference between a condition variable and a semaphore. A semaphore's `signal` increments the value and the following `wait` uses that increment; the signal is remembered. A condition variable has no value; it is only a queue. If no one is waiting, the signal is lost. So the awaited condition (for example `count > 0`) is always kept separately in the shared data and tested before waiting.

6.9.3 Hoare and Mesa semantics

When Q wakes P with `x.signal()`, two processes in the monitor are ready to run. But only one can be active inside. Which one continues is determined by the monitor's **semantics**:

- **Signal-and-wait (Hoare).** The woken P runs at once; Q waits in an "urgent" queue. At the moment P wakes it is sure the condition is true: writing the wait with an `if` suffices.

- **Signal-and-continue (Mesa, 1980).** Q continues; P only becomes "ready" and waits to take the monitor again. Until P runs, another process may spoil the condition. When P wakes it must test the condition again.

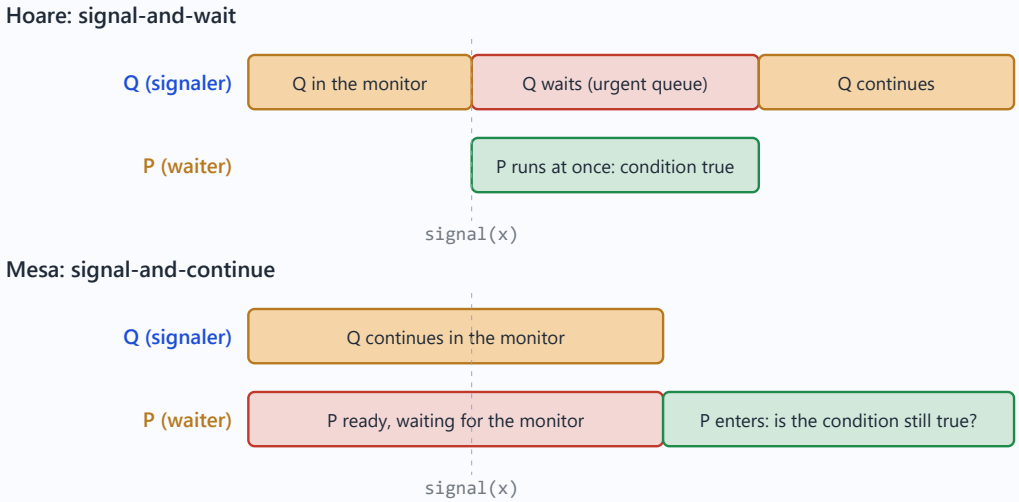


Figure 6.29: (a) Hoare semantics: after `signal` P runs at once and the condition is true; Q waits in the urgent queue. (b) Mesa semantics: Q continues in the monitor; P is ready and, when it enters, must test whether the condition is still true.

Semantics	Who runs at once?	Used by
Signal-and-wait (Hoare)	The woken process	Textbook monitors
Signal-and-exit (Brinch Hansen)	The woken process; `signal` must be the last instruction of the procedure	Concurrent Pascal
Signal-and-continue (Mesa)	The signaler	Pthreads, Java, C#, Go `sync.Cond`, the Linux kernel

Table 6.17: Monitor semantics.

Almost all real systems use Mesa semantics. Under Mesa semantics `signal` is only a "hint": the condition is perhaps true, and the woken thread tests it itself.

6.9.4 Why while and not if?

Under Mesa semantics, writing the wait with an `if` is a bug:

```
/* WRONG (under Mesa) */
lock(&m);
if (count == 0)
  cond_wait(&full, &m);
x = take();      /* may be empty */
unlock(&m);
```

```
/* RIGHT */
lock(&m);
while (count == 0)
  cond_wait(&full, &m);
x = take();      /* count > 0 for sure */
unlock(&m);
```

The way the bug arises is as follows (Figure 6.30):

1. The buffer is empty. Consumer C1 waits on the condition variable `full`.
2. The producer puts an item, sets `count = 1`, and calls `signal`. C1 wakes but must first reacquire the lock; it only becomes ready.
3. Meanwhile a newly arrived consumer C2 grabs the lock first (barging) and takes the item: `count = 0`.
4. C1 finally gets the lock. If written with `if`, it reads from the empty buffer without testing the condition again: a bug.
5. If written with `while`, C1 tests the condition again, sees `count == 0`, and waits again.

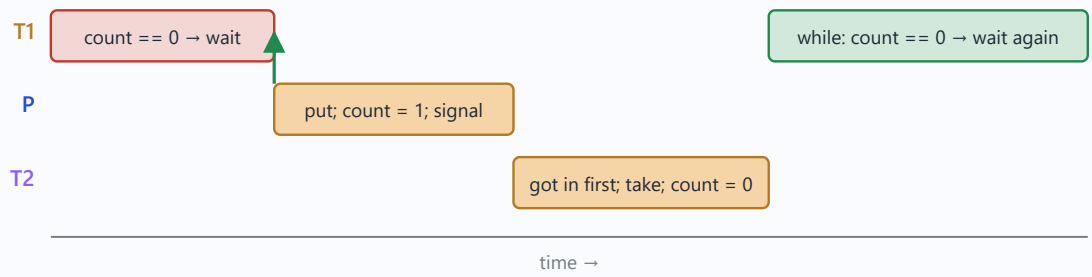


Figure 6.30: (a) C1, waiting with `if`, reads from the empty buffer after C2 has taken the item. (b) With `while`, C1 tests the condition again and waits again.

When all interleavings are explored with a model of a Mesa-style monitor, an execution that reads from an empty buffer is found in the version with `if`. There is no such execution in the version with `while`. POSIX also permits **spurious wakeups**: `pthread_cond_wait` may return even though no `signal` has arrived. So the `while` is mandatory in every case.

6.9.5 A single condition variable is not enough

`while` is necessary but not sufficient. In the Arpaci-Dusseaus' example the buffer has a single slot; there are two consumers (C1, C2) and one producer (P). All use the same condition variable `c`:

```
/* WRONG: one condition c */
C: while (count == 0) wait(&c, &m);
   x = get(); signal(&c);
P: while (count == 1) wait(&c, &m);
   put(x);    signal(&c);
```

1. The buffer is empty; C1 and C2 sleep on `c`.
2. P puts an item and wakes C1 with `signal(c)`.
3. P tries again; since the buffer is full it too sleeps on `c`. Now C2 and P are waiting on `c`.
4. C1 takes the item and does `signal(c)`. Which waiter will wake is not determined; this time C2 wakes.
5. C2 finds the buffer empty and sleeps again. C1 also sleeps in the next round. No one is left to wake the producer: the program stops.

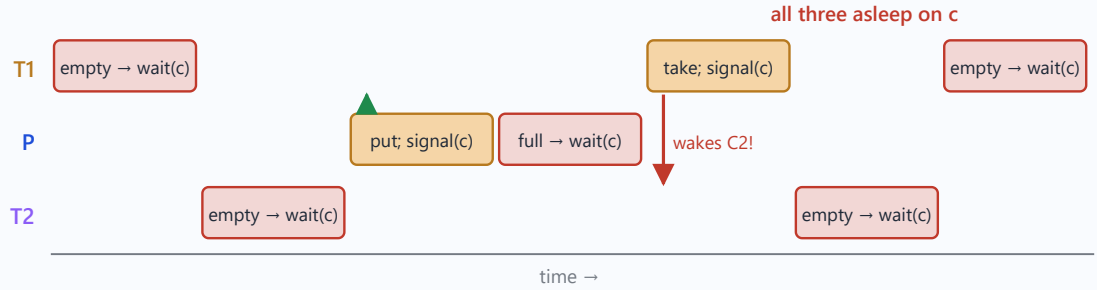


Figure 6.31: Deadlock with a single condition variable. C1's signal wakes C2, not the producer; in the end all three sleep on `c`.

The mistake is that the consumers and the producer wait in the same queue. A consumer should wake only the producer, and the producer only a consumer. The solution is two condition variables:

```
/* RIGHT: two conditions */
C: while (count == 0) wait(&full, &m);
   x = get(); signal(&empty);
P: while (count == 1) wait(&empty, &m);
   put(x);    signal(&full);
```

In a scan with the model, in the version with a single condition a state is reachable in which all three threads sleep despite the `while`. There is no such state in the version with two conditions.

6.9.6 signal and broadcast

Sometimes it cannot be known which waiter can make progress. In a memory allocator, threads ask for different amounts of memory:

```
int avail = 0;          /* free bytes */

void mem_alloc(int n) {
  lock(&m);
  while (avail < n)
    wait(&c, &m);
  avail -= n;  /* ... allocate */
  unlock(&m);
}
void mem_free(int n) {
  lock(&m);
  avail += n;
  broadcast(&c); /* signal won't do */
  unlock(&m);
}
```

Memory is exhausted. Ta wants 100 bytes and Tb wants 10; both sleep in the `while` loop. Tc frees 50 bytes. `signal` wakes only one. If that is Ta, since 50 < 100 it sleeps again; Tb keeps sleeping although it could have proceeded. `broadcast`, however, wakes everyone. Each tests its condition again: Tb takes its 10 bytes and `avail` becomes 40; Ta sees 40 < 100 and sleeps again.

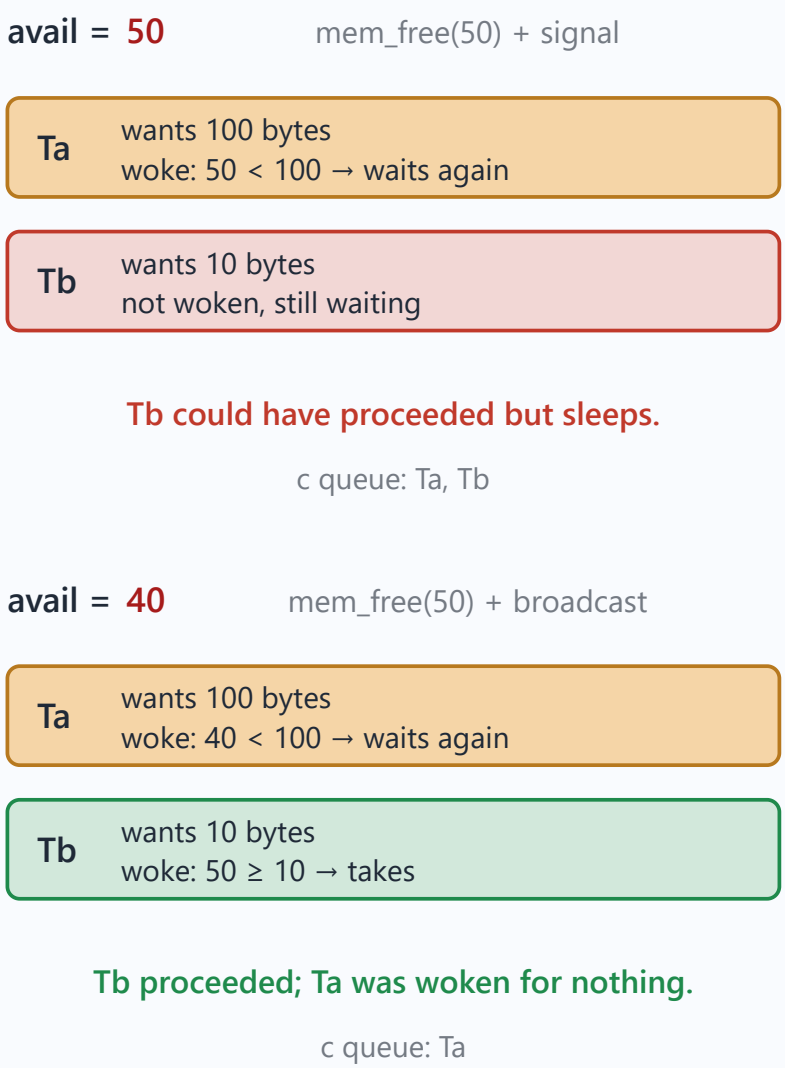


Figure 6.32: (a) `signal` wakes only Ta; Ta waits again and Tb's chance is wasted. (b) With `broadcast` Tb proceeds, and Ta has been woken for nothing.

The rule is this: if it is not known which waiter can make progress, use `pthread_cond_broadcast` (`notifyAll` in Java). This pattern is called a **covering condition**. Its price is unnecessary wakeups. If the waiters can be divided into separate conditions, `signal` is cheaper. In both cases the wait is done inside a `while`.

6.9.7 Producer–consumer with a monitor

The monitor Tanenbaum wrote in Pidgin Pascal solves the bounded buffer with two condition variables:

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove := remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;
```

Access to `count` is already mutually excluded thanks to the monitor. If the buffer is full, the producer blocks with `wait(full)` and releases the monitor. When the consumer takes an item, `count = N - 1` and `signal(full)` wakes the producer. What blocks the producer is `wait`; `signal` only wakes. Here `if` suffices, because Tanenbaum assumes Brinch Hansen semantics: `signal` is the last instruction of the procedure and the signaler leaves at once. In Pthreads or Java the same code must be written with `while`. There is also no lost wakeup: the test and the `wait` are done inside the same monitor, without interruption.

Example 6.8

N = 2 and the buffer is empty. The consumer (C) first calls `remove`; the producer (P) then puts the items a, b, c, d in turn; C calls `remove` twice in between.

Event	count	Buffer	In buffer	full	empty
initial	0	–	–	–	–
C `remove`: count = 0 → `wait(empty)`	0	–	–	–	C
P `insert(a)`	1	a	P	–	C
count = 1 → `signal(empty)`; P leaves, C continues	1	a	C	–	–

Event	count	Buffer	In buffer	full	empty
C takes a; 0 ≠ N − 1, no signal	0	–	–	–	–
P `insert(b)` ; `signal(empty)` goes to waste	1	b	–	–	–
P `insert(c)` ; the buffer is full	2	b c	–	–	–
P `insert(d)` : count = N → `wait(full)`	2	b c	–	P	–
C takes b; count = 1 = N − 1 → `signal(full)`	1	c	P	–	–
P returns from `wait` , puts d	2	c d	–	–	–

Table 6.18: Trace of producer–consumer with a monitor (N = 2, Brinch Hansen semantics).

When b is put, `count = 1` and `signal(empty)` is called; but no one is waiting and the signal is lost. This loss is harmless, because had anyone been waiting it would have been woken anyway. In the end no data is lost and no process sleeps forever.

6.10 POSIX and Java

Textbook monitors are a language feature and do not exist in C. In real programs the same structure is built by using a mutex and condition variables together. This section summarizes the tools in POSIX threads (Pthreads) and Java.

6.10.1 Pthreads: mutexes and condition variables

Mutex call	Purpose	Condition call	Purpose
`pthread_mutex_init`	creates a mutex	`pthread_cond_init`	creates a condition variable
`pthread_mutex_destroy`	destroys the mutex	`pthread_cond_destroy`	destroys the condition variable
`pthread_mutex_lock`	acquires the lock or blocks	`pthread_cond_wait`	atomically releases the mutex and sleeps; reacquires it on return
`pthread_mutex_trylock`	acquires, or returns `EBUSY` at once	`pthread_cond_signal`	wakes at least one of the waiters
`pthread_mutex_unlock`	releases the lock (only its owner)	`pthread_cond_broadcast`	wakes all the waiters

Table 6.19: Pthreads mutex and condition-variable calls.

The rule: `pthread_cond_wait` is always called while holding the mutex and inside a `while` loop. Releasing the mutex and going to sleep atomically closes the window for the lost wakeup. For static initialization there are `PTHREAD_MUTEX_INITIALIZER` and `PTHREAD_COND_INITIALIZER` . The mutex types are `NORMAL` , `ERRORCHECK` , and `RECURSIVE` ; priority inheritance is turned on with the `PTHREAD_PRIO_INHERIT` attribute. Additional interfaces include `pthread_rwlock_*` , `pthread_spin_*` , `pthread_barrier_*` , and C11's `mtx_t` / `cnd_t` types.

In the producer–consumer program that Tanenbaum and Bos wrote with Pthreads, the buffer is a single integer (0 means empty). The producer puts the values 1 and 2, and the consumer prints them:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
               condp = PTHREAD_COND_INITIALIZER;
int buffer = 0;                     /* 0 = empty */

void *producer(void *p) {
  for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
    pthread_mutex_lock(&m);
    while (buffer != 0) pthread_cond_wait(&condp, &m);
    buffer = i;
    pthread_cond_signal(&condc);
    pthread_mutex_unlock(&m);
  } return NULL;
}
void *consumer(void *p) {
  for (int i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&m);
    while (buffer == 0) pthread_cond_wait(&condc, &m);
    printf("%d\n", buffer); buffer = 0;
    pthread_cond_signal(&condp);
    pthread_mutex_unlock(&m);
  } return NULL;
}
```

1. The consumer runs first and takes the mutex. The buffer is empty; `cond_wait` atomically releases the mutex and sleeps.
2. The producer takes the mutex. Since the buffer is empty, it skips the loop and writes 1. `signal` wakes the consumer; but the consumer cannot return without reacquiring the mutex (Mesa).
3. The producer may have won the race and taken the mutex again. For `i = 2` it finds the buffer full, waits on `condp`, and releases the mutex.
4. The consumer takes the mutex, tests the `while` condition again ($1 \neq 0$), prints 1, and empties the buffer.
5. The consumer signals `condp`. The producer wakes, finds the buffer empty, and writes 2. In the second round the consumer prints 2.

All interleavings were explored with a Mesa-style model of the mutex and condition variable. In every execution the output was "1, 2" and none deadlocked. Even if the order changes, the output does not; what changes is only who sleeps when.

6.10.2 POSIX semaphores

```c
#include <semaphore.h>

/* unnamed: threads of one process */
sem_t empty, full;
sem_init(&empty, 0, N);  /* 0: in-process */
sem_init(&full,  0, 0);
sem_wait(&empty); /* … put … */  sem_post(&full);
sem_destroy(&empty);

/* named: between unrelated processes */
sem_t *s = sem_open("/os06", O_CREAT, 0644, 1);
sem_wait(s);  /* CS */  sem_post(s);
sem_close(s);
sem_unlink("/os06");
```

POSIX offers two kinds of semaphores. An **unnamed** semaphore is created as a variable with `sem_init`. If the second argument is 0, the semaphore is used among the threads of the same process. If it is 1, it is shared among processes; then the semaphore must reside in shared memory. A **named** semaphore is opened with a name using `sem_open`, and unrelated processes access it using the same name. `sem_wait` is the P operation and `sem_post` the V operation. `sem_post` can also be called safely from a signal handler. `sem_trywait` tries without waiting and `sem_timedwait` with a time limit; `sem_getvalue` gives the instantaneous value for information only.

macOS does not support unnamed semaphores; there `sem_init` is deprecated. A named semaphore or `dispatch_semaphore` is used. Java's counterpart is `java.util.concurrent.Semaphore`: `acquire()`, `release()`, and an optional fair mode.

6.10.3 Java: synchronized, wait, and notify

In Java every object is a monitor: it has an **intrinsic lock** and a single **wait set**. A `synchronized` method or block takes this lock and releases it on exit, even if an exception is thrown. The lock is reentrant: its owner can acquire the same lock a second time. `wait()` releases the lock and sleeps; `notify()` wakes one of the waiters and `notifyAll()` all of them. The semantics are Mesa; the wait is done inside a `while`.

```
public class OneSlot {
  private int slot = 0;            /* 0 = empty */

  synchronized void put(int x) throws InterruptedException {
    while (slot != 0) wait();      /* releases lock */
    slot = x;
    notifyAll();
  }
  synchronized int take() throws InterruptedException {
    while (slot == 0) wait();
    int x = slot;
    slot = 0;
    notifyAll();
    return x;
  }
  /* main: a producer puts 10, 20, 30, 40;
     main calls take() four times and prints */
}
```

The description of `main` in the slides was turned into a runnable program:

```
public class OneSlot {
  private int slot = 0;            /* 0 = empty */

  synchronized void put(int x) throws InterruptedException {
    while (slot != 0) wait();      /* releases lock */
    slot = x;
    notifyAll();
  }
  synchronized int take() throws InterruptedException {
    while (slot == 0) wait();
    int x = slot;
    slot = 0;
    notifyAll();
    return x;
  }
  public static void main(String[] a) throws InterruptedException {
    OneSlot s = new OneSlot();
    Thread p = new Thread(() -> {
      try { for (int x = 10; x <= 40; x += 10) s.put(x); } catch (InterruptedException e) {
}
    });
    p.start();
    for (int k = 0; k < 4; k++) System.out.println("took " + s.take());
    p.join();
  }
}
```

The program gave the same output on every run with JDK 21:

```
took 10
took 20
took 30
took 40
```

The producer and the consumer share the same wait set. That is why `notifyAll` is safer than `notify`: `notify` may wake a waiter of the wrong kind, and the deadlock seen with a single condition variable can occur.

6.10.4 Java: ReentrantLock and Condition

The `java.util.concurrent.locks` package offers a more flexible lock. Several `Condition` objects can be attached to one `ReentrantLock`. Producers and consumers can then wait in separate queues, and `signal` wakes the right group. The lock also offers `tryLock()`, timed and interruptible locking, and an optional fair mode. `unlock()` must always be called inside `finally`; otherwise an exception leaves the lock held forever.

```
class Buffer {
  private final int[] buf = new int[2];
  private int head = 0, tail = 0, count = 0;
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition notFull  = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();

  void put(int x) throws InterruptedException {
    lock.lock();
    try {
      while (count == buf.length) notFull.await();
      buf[tail] = x; tail = (tail + 1) % buf.length; count++;
      notEmpty.signal();
    } finally { lock.unlock(); }
  }
  /* take(): while (count == 0) notEmpty.await(); …
             notFull.signal();                      */
}
/* producer puts 1..5; main calls take() 5 times */
```

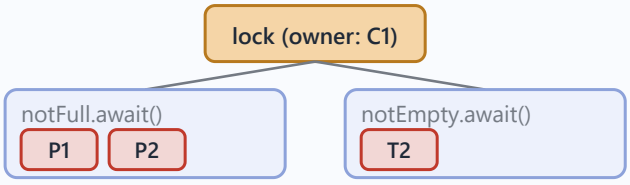


Figure 6.33: Two conditions attached to one lock. The producers (P1, P2) wait on `notFull` and the consumer C2 waits on `notEmpty`; the lock's owner is C1.

The class in the slides was completed with the `take()` method and run:

```
import java.util.concurrent.locks.*;
class Buffer {
  private final int[] buf = new int[2];
  private int head = 0, tail = 0, count = 0;
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition notFull  = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();

  void put(int x) throws InterruptedException {
    lock.lock();
    try {
      while (count == buf.length) notFull.await();
      buf[tail] = x; tail = (tail + 1) % buf.length; count++;
      notEmpty.signal();
    } finally { lock.unlock(); }
  }
  int take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0) notEmpty.await();
      int x = buf[head]; head = (head + 1) % buf.length; count--;
      notFull.signal();
      return x;
    } finally { lock.unlock(); }
  }
  public static void main(String[] a) throws InterruptedException {
    Buffer b = new Buffer();
    Thread p = new Thread(() -> {
      try { for (int x = 1; x <= 5; x++) b.put(x); } catch (InterruptedException e) { }
    });
    p.start();
    int sum = 0; StringBuilder sb = new StringBuilder();
    for (int k = 0; k < 5; k++) { int x = b.take(); sum += x; sb.append(x).append(' '); }
    p.join();
    System.out.println(sb + "| sum = " + sum);
  }
}
```

All three runs with JDK 21 gave the same output:

```
1 2 3 4 5 | sum = 15
1 2 3 4 5 | sum = 15
1 2 3 4 5 | sum = 15
```

With a two-slot buffer the producer can be at most two items ahead of the consumer; when the buffer is full it waits on `notFull`. The items are always taken in the order they were put. When the threads go to sleep varies from run to run; the output does not. A note on virtual threads: in Java 21, when a virtual thread blocked inside `synchronized`, it "pinned" the platform thread carrying it. For that reason `ReentrantLock` was recommended in that version. In JDK 24, JEP 491 fixed this problem.

6.11 Liveness

Correct synchronization first ensures safety: the data is not corrupted. But the system must also make progress. This section covers the liveness problems that can arise even when exclusion is correct.

6.11.1 Priority inversion

Under priority scheduling, a high-priority task runs the moment it is ready. Locks can break this rule in unexpected ways. Consider three tasks: high-priority H, medium-priority M, and low-priority L. H and L use a shared resource R; M has nothing to do with R. There is a single processor and times are in milliseconds.

1. **t = 0.** L takes R's lock. Its critical section will take 4 ms in total.
2. **t = 2.** H arrives and preempts L. At t = 3 it asks for R and blocks. A short wait is expected: when L finishes the remaining 2 ms, H will enter.
3. **t = 4.** M arrives. M has higher priority than L; it preempts L and runs until t = 10. L runs again only at t = 10 and releases R at t = 11.

H has waited 8 ms for R; 6 ms of that is M's running. The highest-priority task waits for a medium-priority task that has nothing to do with it. As M gets longer, H's wait grows without bound.

> **Definition 6.13**
>
> **Priority inversion** is a high-priority task waiting for a lock held by a low-priority task. In the meantime, because of the medium-priority tasks that step in, the high-priority task indirectly waits for them too. **Priority inheritance** is the lock-holding task temporarily taking over the priority of the highest-priority task waiting for that lock.

Priority inversion is not a solution but a problem. With priority inheritance, while H waits for R, L temporarily takes H's priority. Even if M arrives, it cannot preempt L. L runs from t = 3 to t = 5, releases R, and returns to its old priority. H runs at once and waits only 2 ms (Figure 6.34). Another solution is the **priority ceiling**: the task that acquires the lock is immediately raised to the priority of the highest-priority task that can use that lock.

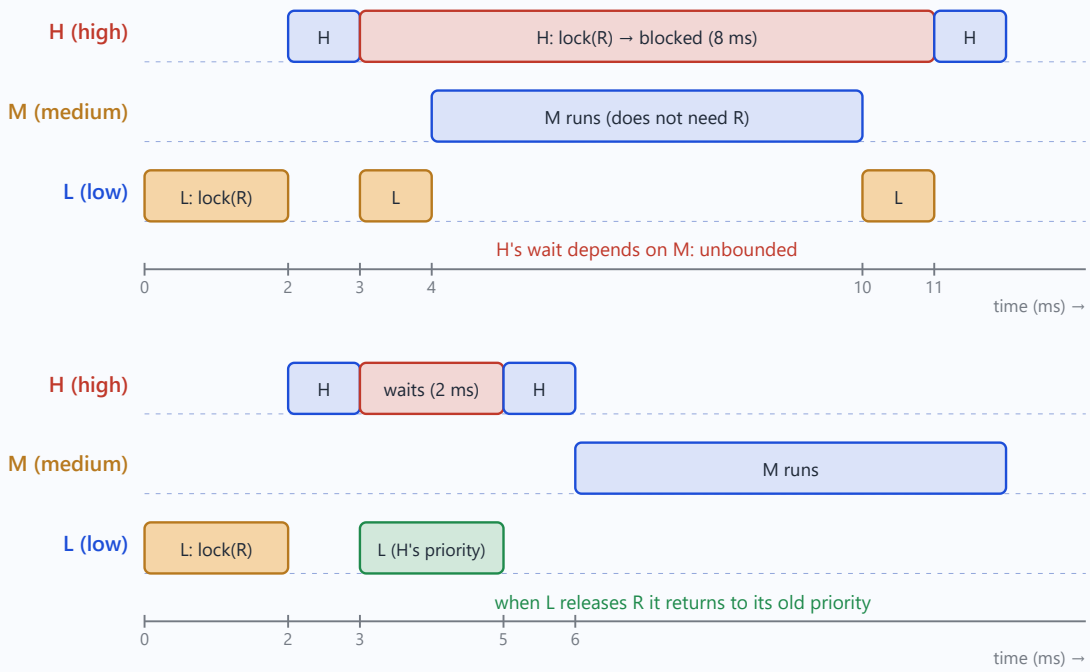


Figure 6.34: (a) Priority inversion: while H waits for R, M runs; H's wait depends on M's length. (b) Priority inheritance: L runs with H's priority, M cannot step in, and when L releases R, H runs at once.

These two scenarios were rerun, unit by unit, with a small model of a preemptive priority single-processor scheduler. H's wait for the lock came out as 8 without inheritance and 2 units with inheritance. If H waits for the lock with a spinlock, the result on a single core is worse. When scheduling is strict-priority and preemptive, the highest-priority ready process always runs; H, spinning in a loop without blocking, is always ready. H never gives up the processor, the lower-priority L can never run and can never release the lock: the system locks up.

Example 6.9

On a single CPU the priorities are H > M > L. L takes the lock at t = 0 and its critical section takes 4 units. H arrives at t = 1 and immediately asks for the same lock; once it gets the lock it runs for 2 units. M arrives at t = 2, runs for 5 units, and uses no lock. How many units does H wait for the lock, and when does it finish?

1. **No inheritance.** L runs from 0 to 1. H blocks at t = 1. L continues from 1 to 2. M arrives at t = 2 and, preempting L, runs from 2 to 7. L finishes its remaining 2 units from 7 to 9 and releases the lock. H runs from 9 to 11. H waits 8 units, from 1 to 9, and finishes at t = 11.
2. **With inheritance.** When H blocks at t = 1, L takes H's priority. Even though M arrives at t = 2, it cannot preempt L. L finishes its remaining 3 units from 1 to 4 and releases the lock. H runs from 4 to 6. H waits 3 units and finishes at t = 6; M runs from 6 to 11.

6.11.2 Mars Pathfinder (1997)

The most famous example of priority inversion happened on Mars. In July 1997 the Pathfinder lander on the Martian surface, running under the VxWorks real-time operating system, reset itself again and again. A low-priority meteorological task held a mutex that protected the information bus. The high-priority bus-management task blocked on this mutex. Meanwhile the medium-priority, long-running communications task stepped in (Figure 6.35).

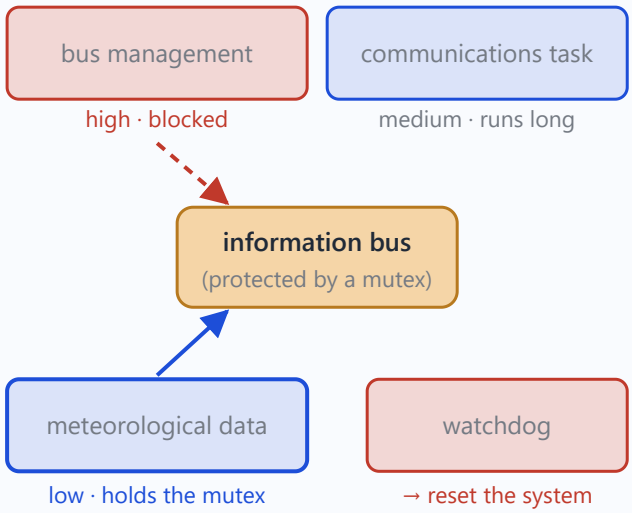


Figure 6.35: Priority inversion on Pathfinder. The meteorological task (low) holds the mutex, bus management (high) waits, the communications task (medium) runs for a long time, and the watchdog timer resets the system.

When the bus-management task could not run on time, a **watchdog timer** concluded "the system has hung" and did a full reset. JPL engineers reproduced the bug on the copy on the ground. The mutex's priority-inheritance option existed in VxWorks but was turned off; it was turned on with a small patch uploaded remotely, and the resets stopped. Today the same mechanism exists as the `PTHREAD_PRIO_INHERIT` attribute in POSIX and as priority-inheriting futexes and `rt_mutex` in Linux. The PREEMPT_RT real-time patch, which entered the mainline in Linux 6.12, also relies on this mechanism.

6.11.3 Deadlock, livelock, and starvation

Three liveness problems are often confused (Figure 6.36):

Definition 6.14

- **Deadlock**: every member of a set of processes waits for a resource held by another member of the set. The processes are blocked and the processor may sit idle. It occurs when four necessary conditions hold at the same time.
- **Livelock**: the processes are not blocked; they react to one another and keep repeating the same steps, but none makes progress.
- **Starvation**: the system as a whole makes progress, but one process keeps giving way to others and never gets a turn.

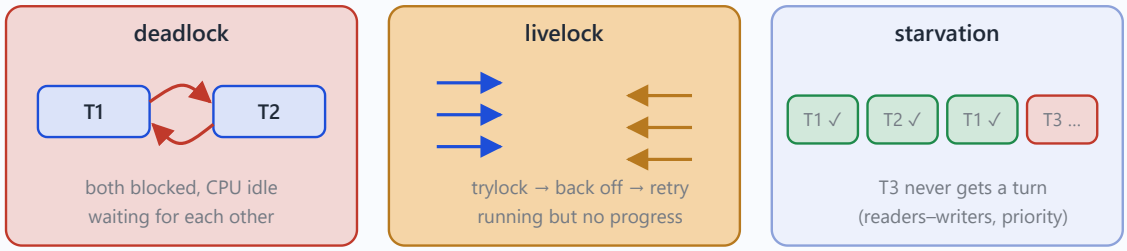


Figure 6.36: In deadlock everyone is blocked, in livelock everyone is running but there is no progress, and in starvation T3 never gets a turn while the others progress.

The typical example of livelock is two "polite" threads. Each wants two locks. A thread that cannot get the second lock with `trylock` releases the one it holds and retries after always the same interval. If the two threads work in exactly the same rhythm, they give way to each other forever. The cure is to back off for a random (usually exponentially growing) time; this is the method used for Ethernet collisions. The cure for starvation is FIFO queues, fair locks, and aging.

6.12 Introduction to lock-free structures

Every structure that uses locks has a weakness: if the thread holding the lock is preempted, waits for a page fault, or crashes, everyone stops. **Lock-free** structures use atomic instructions such as CAS directly instead of locks. Progress guarantees are defined at three levels:

Kind	Guarantee
Blocking	If the thread holding the lock stops, everyone stops.
Lock-free	The system as a whole always makes progress; some threads may fail repeatedly.
Wait-free	Every thread finishes its own operation in a bounded number of steps.

Table 6.20: Progress guarantees.

The simplest lock-free structure is the Treiber stack. The top of the stack is a single atomic pointer. `push` prepares the new node, links the node's `next` field to the current top, and tries to change the top with CAS:

```
void push(node_t *n) {
  node_t *old = atomic_load(&top);
  do {
    n->next = old;
  } while (!atomic_compare_exchange_weak(
              &top, &old, n));
}  /* on failure old = current top */
```

Example 6.10

The stack is top → A → B. T1 wants to add X and T2 wants to add Y.

1. **T1 prepares.** It reads `old = A` and sets `X.next = A`. It is interrupted at this point.
2. **T2 adds.** `CAS(top, A, Y)` succeeds; the stack becomes top → Y → A → B.
3. **T1's CAS fails.** `top` is now Y, not A. C11's `compare_exchange` function writes the current value into `old` on failure: `old = Y`. The loop sets `X.next = Y` and tries again.
4. **The second CAS succeeds.** The stack becomes top → X → Y → A → B.

No one held a lock. Even if T1 had stayed suspended forever after step 1, T2 could have finished its work.

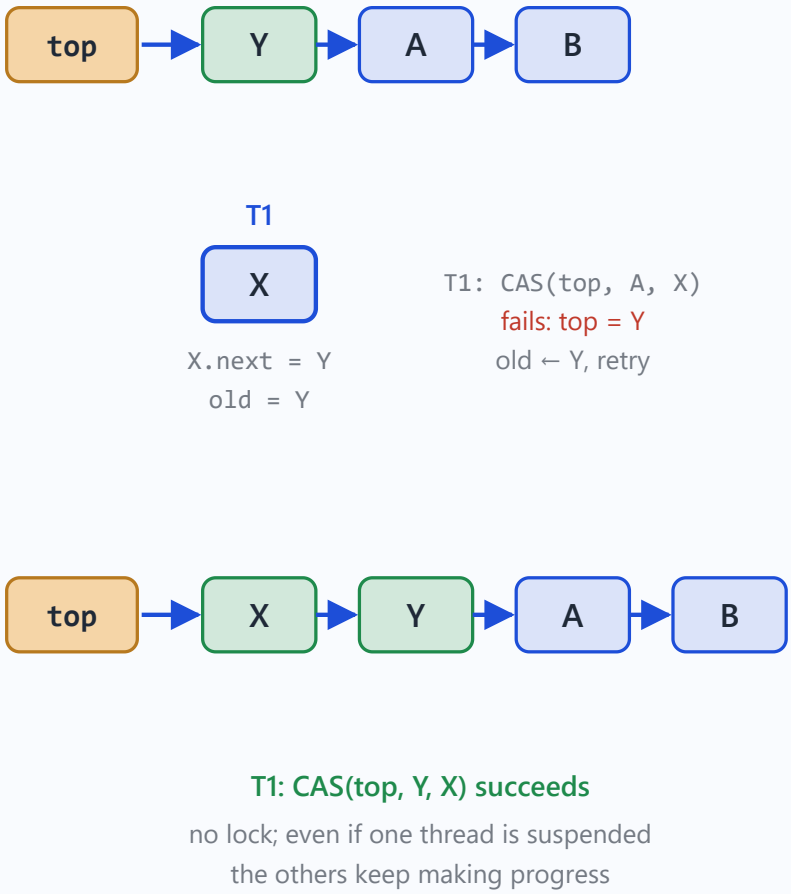


Figure 6.37: `push` on the Treiber stack. (a) After T2's insertion, T1's `CAS(top, A, X)` fails; T1 sets `old` to Y. (b) The second attempt succeeds.

The famous trap of lock-free programming is the **ABA problem**. CAS looks only at the equality of the value; it cannot know whether the value changed and changed back in between. In `pop`, let T1 read `top = A` and `A.next = B` and be interrupted. Meanwhile T2 pops A and B, frees B, and pushes A again. `top` is A once more. T1's `CAS(top, A, B)` succeeds and puts on top the node B, which is no longer on the stack and whose memory has been freed. The cure is to compare a version counter that increases with every change along with the pointer, or to delay freed nodes with safe memory reclamation (hazard pointers, epochs).

6.13 Summary

- A **race condition** is when the result of processes accessing shared data depends on the order of execution. Even a single `count++` is a read–modify–write sequence on the machine and can be interrupted in the middle. Such errors appear occasionally and are hard to catch with testing.
- A **critical section** is code that accesses shared data. A solution must satisfy three requirements: **mutual exclusion** (safety), **progress**, and **bounded waiting** (liveness).
- **Disabling interrupts** does not work on multiple cores and is not open to users. A **lock variable** breaks exclusion and **strict alternation** breaks progress. **Peterson's algorithm** satisfies the three requirements for two processes using only reads and writes, but assumes sequentially consistent memory.
- On modern processors the store buffer and compiler reordering break Peterson. x86 (TSO) allows only the "write → later read" reordering. A **memory barrier** or sequentially consistent atomics (C11 `_Atomic`, Java `volatile`) restore the right order; C's `volatile` does not.
- The hardware offers **atomic instructions**: `test_and_set`/TSL, XCHG, and CAS. A simple atomic spinlock does not guarantee bounded waiting; a `waiting[]` array and cyclic hand-over provide it. The CAS loop is the basis of lock-free updates.
- A **spinlock** busy-waits and is suitable on multiple cores for short critical sections. A **mutex** puts the waiter to sleep. On Linux, mutexes are built with the **futex**: if there is no contention, the kernel is not entered. A per-core counter scales by giving up a little accuracy.
- A **semaphore** is a counter used with `wait`/`signal`. With an initial value of 1 it is the exclusion pattern, with N resource counting, with 0 ordering. A semaphore stores the wakeup; the signal is not lost.
- The **bounded buffer** is solved with the semaphores `mutex`, `empty`, and `full`; if the order of `down` is reversed, deadlock occurs. The reader-priority **readers–writers** solution is open to writer starvation. The naive solution of the **dining philosophers** deadlocks; it is solved with a room semaphore, resource ordering, or a state array.
- A **monitor** provides exclusion at the language level; a **condition variable** is a wait queue with no value. Real systems use Mesa semantics: the wait is always inside a `while`, separate condition variables are used for separate groups, and `broadcast` is used if it is not known which waiter will proceed.
- In Pthreads `pthread_cond_wait` atomically releases the mutex and sleeps. In Java every object is a monitor (`synchronized`, `wait`, `notifyAll`); `ReentrantLock` offers several `Condition` objects.
- **Priority inversion** is a high-priority task indirectly waiting for a medium-priority one; it is solved by **priority inheritance** or the ceiling (Mars Pathfinder). Deadlock, livelock, and starvation are different liveness problems.
- **Lock-free** structures work with a CAS loop and are not affected by the lock holder stopping; but they need a version counter or safe memory reclamation against the ABA problem.

Tool	What it provides	When
Peterson, strict alternation	Exclusion with only reads/writes (assuming SC)	Conceptual; not used in real code
Memory barrier	Order and visibility of reads/writes	Inside lock and atomic implementations
TAS, XCHG, CAS	Atomic read-modify-write	The foundation of locks and lock-free structures
Atomic variable	Indivisible update of a single variable	Counters, flags, reference counting
Spinlock	Exclusion without sleeping	Multicore, very short critical sections, inside the kernel
Mutex	Exclusion, the waiter sleeps	General-purpose critical sections
Semaphore	Counting, exclusion, and ordering	N identical resources, event notification
Monitor + condition variable	Structured exclusion + waiting on a condition	Complex conditions (Java, Pthreads)
Readers–writer lock	Many readers, one writer	Read-mostly data

Table 6.21: Summary of the synchronization tools.

6.14 Key terms

synchronization race condition critical section mutual exclusion bounded waiting busy waiting Peterson's algorithm memory barrier atomic instruction spinlock mutex semaphore producer–consumer problem readers–writers problem dining philosophers problem monitor condition variable priority inversion priority inheritance deadlock livelock starvation

6.15 Test Yourself

1. The shared `count = 10`. T1 executes `count = count + 3` and T2 executes `count = count - 1`; each is three instructions: LOAD, ADD, STORE. What are all the possible final values?

   Answer

   12, 13, and 9. If one finishes before the other reads, the result is 12. If T2's update is lost the result is 13, and if T1's is lost it is 9. All 20 interleavings give one of these three values.

2. If the line `flag[i] = false` in the exit section of Peterson's algorithm is forgotten, which of the three requirements is violated? What happens to P1 after P0 enters and leaves once and stays in

its remainder section?

Answer

Progress is violated. `flag[0]` stays `true` forever. When P1 wants to enter it writes `turn = 0`; its condition `flag[0] && turn == 0` is true and P1 spins. While P0 sits in its remainder section, the only thing that can clear P1's way is P0's wanting to enter again and writing `turn = 1`. Although the critical section is empty, a process in its remainder section blocks the other.

3. On a semaphore with initial value 3 (Silberschatz implementation, FIFO queue), P1–P5 do `wait` in turn, followed by two `signal` calls. What are the value and the queue after the five `wait` calls and after the two `signal` calls?

Answer

After the five `wait` calls the value is 3 − 5 = −2; P1, P2, and P3 have passed, and P4 and P5 are in the queue. The first `signal` makes the value −1 and wakes P4; the second makes it 0 and wakes P5. In the final state the value is 0 and the queue is empty.

4. In the bounded-buffer solution with semaphores N = 3 and the consumer never runs. If the producer tries to put five items, how many items are put, what are `empty` and `full`, and at which item does the producer sleep?

Answer

Three items are put. Each time an item is put, `empty` decreases by one and `full` increases by one: at the end `empty = 0` and `full = 3`. For the fourth item `down(&empty)` finds the value 0 and the producer sleeps. The buffer does not overflow.

5. In the barrier code with semaphores, if the line `up(&turnstile)` after passing the turnstile is deleted, how many threads pass the barrier when N = 3?

Answer

Only one. The last arriving thread's `up` lets a single thread through the turnstile. Because the one that passes does not let the one behind it through, the other two sleep at the turnstile forever. When all interleavings were explored, the number that passed in every deadlocked execution was 1.

6. With `v = 0` three threads each increment once with the CAS loop (`increment`). What is the final

value? What are the minimum and maximum total numbers of CAS calls?

Answer

The final value is always 3. At least 3 CASes are done: if each finishes before the others read, there is no failure at all. At most 3 + 2 + 1 = 6 CASes are done. If all three read the same value, one succeeds and two retry. In the next round again one succeeds and one fails.

7. On a single CPU H > M > L. L takes the lock at t = 0 and its critical section is 3 units. H arrives at t = 1 and immediately asks for the same lock; once it gets the lock it runs for 1 unit. M arrives at t = 2, runs for 4 units, and uses no lock. How many units does H wait, and when does it finish, without and with priority inheritance?

   Answer

   Without inheritance the schedule is L(0–1), L(1–2), M(2–6), L(6–7), H(7–8). H waits 6 units, from 1 to 7, and finishes at t = 8. With inheritance L runs with H's priority from 1 to 3 and releases the lock; H runs from 3 to 4. H waits 2 units and finishes at t = 4; M runs from 4 to 8.

8. On a two-core machine an approximate counter with threshold `S = 5` is used. Core 0 calls `add()` twelve times and core 1 seven times. What does `get()` return at the end? What is the true value? What is the largest the difference can be in the worst case?

   Answer

   Core 0 transfers 5 twice and 2 remain locally; core 1 transfers 5 once and 2 remain locally. `get()` = G = 15 and the true value is 19. In the worst case the difference is 2 × (5 − 1) = 8.

9. At a table with six philosophers, at most how many can eat at the same time? Does the naive solution (left fork first, then right) deadlock with six philosophers too?

   Answer

   ⌊6/2⌋ = 3: for example P0, P2, and P4 can eat together. The naive solution deadlocks regardless of the number of philosophers: if all take their left fork, a circular wait forms.

10. In the store-buffer test T1 executes `x = 1; r1 = y;` and T2 executes `y = 1; r2 = x;`. A full memory barrier is placed only in T1, between the write and the read. On x86 (TSO) can `r1 = r2`

`= 0` still be observed?

Answer

Yes. T1's barrier only ensures that T1's own write is visible before its read. While T2's write `y = 1` waits in the buffer, T2 can read `x` from memory as 0; T1 can also read `y` from memory as 0. When all executions are explored in the store-buffer model, (0, 0) still comes out. The barrier must be placed on both sides.

6.16 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters "Locks," "Condition Variables," "Semaphores," and "Common Concurrency Problems": explain how locks are built from hardware, the approximate counter, and the pitfalls of condition variables, with short, measured examples.
- A. B. Downey, *The Little Book of Semaphores*: solves dozens of problems with semaphores, from the barrier with a turnstile to the starvation-free readers–writers; before each solution it leaves room for the reader to try it alone. The book is free.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th ed.), Chapters 6 and 7: treats the critical-section conditions, hardware support, and the tools in real systems in detail.
- B. Goetz et al., *Java Concurrency in Practice*: explains the Java memory model, the meaning of `volatile`, and the correct use of the `java.util.concurrent` tools, with practical examples.

6.17 References

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Locks," "Condition Variables," "Semaphores," "Common Concurrency Problems."
2. A. B. Downey, *The Little Book of Semaphores*, 2nd ed., Green Tea Press.
3. B. Goetz et al., *Java Concurrency in Practice*, Addison-Wesley, 2006; Java SE 21 API: `java.util.concurrent`; JEP 491.
4. U. Drepper, "Futexes Are Tricky," Red Hat technical report, version 1.6, 2011 (first version 2004), akkadia.org/drepper/futex.pdf.
5. M. Jones, “What really happened on Mars?”, 1997.
6. M. M. Michael, M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing* (PODC '96), pp. 267–275, 1996, doi:10.1145/248052.248106.
7. G. L. Peterson, "Myths About the Mutual Exclusion Problem," *Information Processing Letters*, 1981.
8. POSIX.1-2017 (IEEE Std 1003.1): `pthread_mutex_lock`, `pthread_cond_wait`, `sem_overview(7)`; the Linux `futex(2)` manual page.

9. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 6: Synchronization Tools, Chapter 7: Synchronization Examples.
10. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014 (5th ed. 2023) — §2.3 Interprocess Communication, §2.5 Classical IPC Problems.

PART II · CHAPTER 7

Deadlocks

If two threads try to acquire two locks in opposite orders, both may wait forever. Because neither runs, neither releases the lock it holds. This situation is called a **deadlock**. A deadlocked program raises no error and does not crash; it simply stops. Worse, it often appears only under one particular interleaving and can slip through testing. The chapter begins with the notion of a resource and the four conditions for deadlock. It then models deadlock with graphs and works through the four ways of coping with it in turn: ignoring it, detecting and recovering, avoiding, and preventing. Cycle search and a matrix-based algorithm are worked out step by step for detection, and safe states and the banker's algorithm for avoidance. The final sections cover livelock, starvation, concurrency bugs that are not deadlocks, and how deadlocks are found in Java programs.

LEARNING OUTCOMES

Definition and resources
Define deadlock; distinguish preemptable from nonpreemptable resources and describe the request–use–release cycle.

Four conditions
Demonstrate mutual exclusion, hold and wait, no preemption, and circular wait on an example.

Modeling
Draw resource-allocation and wait-for graphs, and say when a cycle in the graph means deadlock.

Detection and recovery
Execute the cycle-search and matrix-based detection algorithms by hand; compare recovery by preemption, rollback, and termination.

Avoidance
Tell safe states from unsafe ones; use the banker's algorithm to decide whether a request should be granted.

Prevention and practice
Recognize lock ordering, two-phase locking, livelock, and starvation; find a deadlock in Java with `jstack`.

7.1 Deadlock and resources

Processes that run concurrently share resources: a printer, a file, a database record, the lock that protects a data structure in memory. The usual way to regulate sharing is to make the other processes wait until the process using the resource has finished. Waiting is normally harmless; the waiting process resumes when its turn comes. But if the waiting forms a ring, the turn never comes. This section gives the definition of deadlock, the kinds of resources involved, and the smallest possible example.

7.1.1 What is deadlock?

Consider the simplest case. Process P1 holds resource R1 and waits for R2. Process P2 holds R2 and waits for R1. P1 cannot make progress because it cannot get R2, and it does not release R1. For the same reason P2 does not release R2. Both processes are blocked. The only event that could wake them is the other releasing its resource, and that event will never happen.

Definition 7.1

A set of processes is **deadlocked** if every process in the set is waiting for an event that only another process in the set can trigger. The awaited event is usually the release of a resource.

The phrase “only another process in the set” in the definition matters. If someone outside can rescue the waiting process, for example if the process holding the resource lies outside the set and will eventually finish, there is no deadlock, only a long wait. In a deadlock the rescuer is inside the set, and since everyone in the set is waiting, the rescuer never runs.

Everyday life offers similar situations. At an intersection that four cars enter at the same time, each car blocks the one on its right, and none can move (*gridlock*). Two trains approaching each other on a single track also wait for each other. Figure 7.1 shows the intersection example.

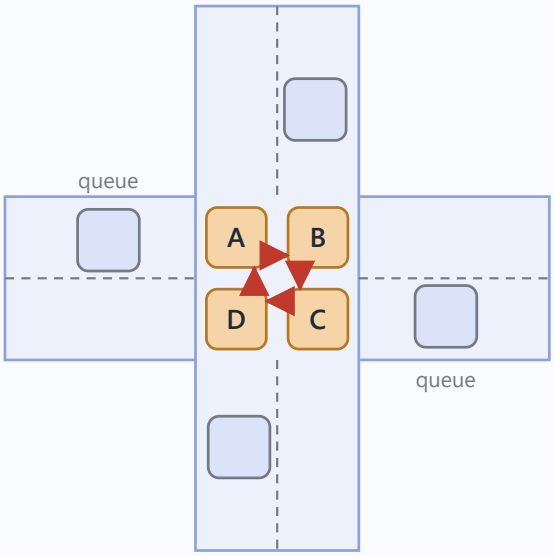


Figure 7.1: Deadlock at an intersection. Four cars hold the four squares of the intersection, and each waits for the square held by the next. The queues behind them mean that none can back up, either.

Deadlocked processes burn no CPU time, because they are blocked. The real damage is that the resources they hold stay unusable forever. Other processes that want those resources start to wait as well, and the deadlock spreads. The problem is not specific to operating systems: database transactions, multithreaded applications, and distributed systems also deadlock. In these notes the word “process” includes threads; everything said here holds for both.

7.1.2 Preemptable and nonpreemptable resources

Deadlock is about **resources**. A resource is anything a process must acquire in order to use it: a hardware device, a region of memory, a file, a lock, a database record. For deadlock purposes resources fall into two classes.

Definition 7.2

A **preemptable resource** can be taken away from its owner without harm and given back later. A **nonpreemptable resource**, in contrast, breaks the work in progress if it is taken from its owner; only the owner can release it, of its own accord.

The CPU is a preemptable resource. The scheduler stops the running process, saves its registers, and gives the CPU to another process; the process later continues where it left off. Main memory is preemptable too: a page can be written to disk and its frame given to another process, and the page is loaded back when needed. For such resources, even if a ring of waiting forms, the operating system can break it by reallocating the resource.

A Blu-ray disc that is half written is not preemptable: interrupting the burn and handing the drive to another process ruins the disc. A job being printed is similar; pages of another job would be interleaved with it. Data protected by a mutex or semaphore is left in an inconsistent intermediate state if the lock is taken by force. Deadlocks concern nonpreemptable resources of this kind. Table 7.1 lists common resources by type.

Resource	Number of instances	Type	Note
CPU core	many	preemptable	changes hands on a timer interrupt
Memory page	many	preemptable	reclaimed by swapping or paging
Mutex, semaphore, file lock	1	nonpreemptable	only the owner releases it
Database row lock	1	nonpreemptable*	*released if the transaction is rolled back

Table 7.1: Classification of resources from the deadlock point of view.

The "number of instances" column of the table also matters. A resource type may have several **identical instances**. If the system has two tape drives and a process asks for "a tape drive," it makes no difference which of the two it gets. Instances are identical if a request can be satisfied by any instance of the type. In general, resource type R_i has W_i instances. The number of instances directly affects how deadlock is detected.

> **Common mistake**
>
> Thinking that a database record lock is preemptable because the database resolves a deadlock by aborting a transaction. Aborting is not the same as forcibly taking the lock and giving it to someone else; all the changes of the transaction are undone and the transaction starts over. The resource is not preempted, work is lost. That is why the table marks it with an asterisk.

7.1.3 The resource-use cycle

A process always uses a resource in the same three steps:

1. **Request.** If the resource is free, it is given to the process. If not, the process blocks and is woken when the resource becomes free. In some systems the process instead receives an error code and retries a little later.
2. **Use.** The process operates on the resource: it prints on the printer, writes to the file, updates the protected data.
3. **Release.** The resource becomes free; one of the processes waiting for it is woken.

The operating system keeps a table recording which resource is assigned to whom and who is waiting for which resource. Deadlock detection works from the information in this table. Requesting and releasing are usually system calls or library functions; Table 7.2 shows the best-known pairs.

Request	Release
`open()`	`close()`
`malloc()` / `mmap()`	`free()` / `munmap()`
`pthread_mutex_lock()`	`pthread_mutex_unlock()`
`sem_wait()` (down)	`sem_post()` (up)

Table 7.2: Counterparts of the request and release operations.

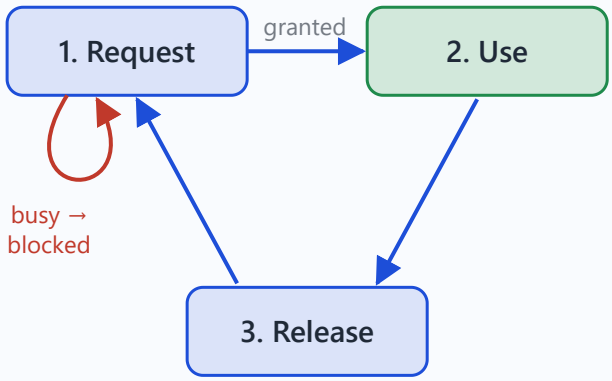


Figure 7.2: The resource-use cycle. If the resource is busy, the process blocks at the request step; when the resource is released, it wakes and moves on to the use step.

7.1.4 Same order and reverse order

Suppose user processes protect their resources with semaphores. Let each resource correspond to a binary semaphore: `down` before accessing the resource, `up` when finished. With one process or one resource there is no problem. With two processes and two resources, the order in which the resources are acquired decides the outcome. The following two processes acquire the resources in the same order:

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
```

This code does not deadlock. Both processes ask for `resource_1` first. One gets it and the other waits. The process that has `resource_1` acquires `resource_2` without trouble, because the only other process that could want it is still waiting in front of `resource_1`. When it finishes and releases both resources, the waiting process continues.

Now reverse the order in B:

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_2);  /* reverse order */
  down(&resource_1);
  use_both_resources();
  up(&resource_1);
  up(&resource_2);
}
```

In this version deadlock is possible. If A takes `resource_1` and B takes `resource_2`, each waits for what the other holds. The order of release does not matter; what counts is the order of acquisition, that is, of the `down` calls. From the programmer's point of view the two versions are almost identical. Swapping two lines turns a program that never deadlocks into one that deadlocks now and then.

7.1.5 Code trace: acquiring in reverse order

An abbreviated form of the reverse-order code follows. Both binary semaphores start at 1, so both resources are free.

```
semaphore r1 = 1, r2 = 1;
void A(void) {
  down(&r1);
  down(&r2);
  use_both();
  up(&r2); up(&r1);
}
void B(void) {
  down(&r2);
  down(&r1);
  use_both();
  up(&r1); up(&r2);
}
```

Suppose that on a single CPU the scheduler picks this sequence: A's first `down`, B's first `down`, A's second `down`, B's second `down`. Table 7.3 shows the state of the semaphores and processes at each step, and Figure 7.3 shows the graph for the same steps. In the graph a circle stands for a process and a square for a resource. A solid arrow from a resource to a process means “this resource belongs to this process,” and a dashed arrow from a process to a resource means “this process is waiting for this resource.”

Step	Event	`r1`	`r2`	A	B
0	start	1	1	ready	ready
1	A: `down(&r1)`	0	1	holds r1	ready
2	B: `down(&r2)`	0	0	holds r1	holds r2
3	B: `down(&r1)`	0	0	holds r1	blocked (r1)
4	A: `down(&r2)`	0	0	blocked (r2)	blocked (r1)

Table 7.3: Trace of acquisition in reverse order.

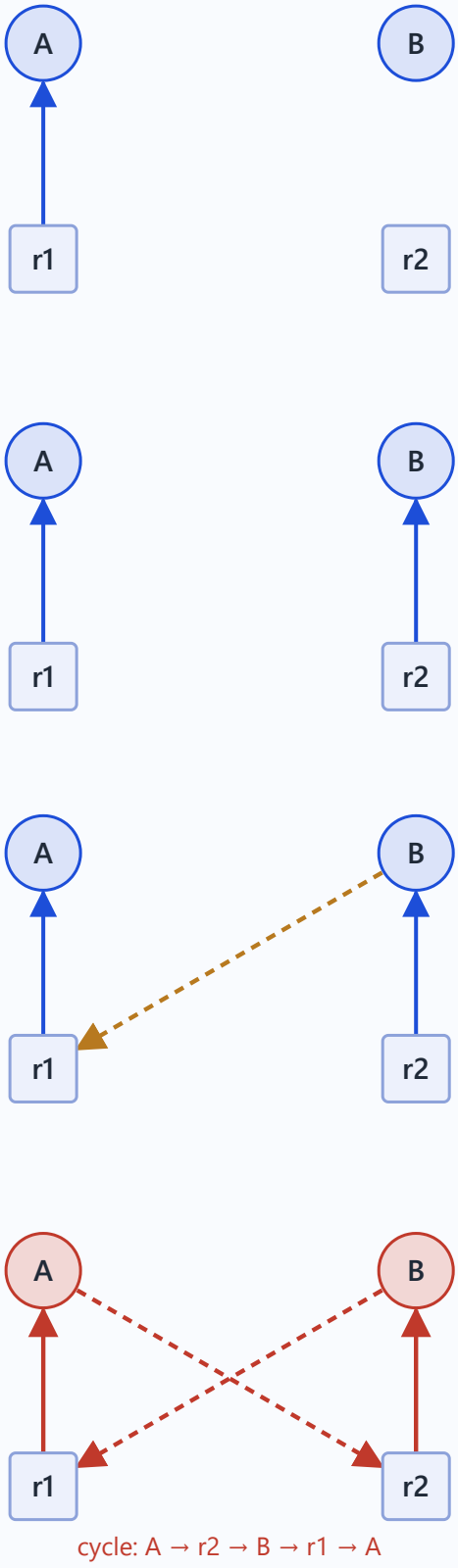


Figure 7.3: Acquisition in reverse order. (a) A takes `r1`. (b) The scheduler switches to B; B takes `r2`. (c) B requests `r1` and blocks. (d) A requests `r2` and blocks too: the cycle A → r2 → B → r1 → A has formed in the graph.

After the fourth step both processes are blocked, and neither can reach its `up` lines. The cycle in the graph is the visual mark of deadlock. The deadlock does not occur on every run. If A finishes both `down` calls before B starts, B waits at its first `down` and continues when A is done. Deadlock occurs only if both processes execute their first `down` before the other's second `down`. Because this window is narrow, the bug is rarely seen in testing and shows up when the system is under load.

Common mistake

Running the program a few times, seeing no deadlock, and concluding that the code is correct. A deadlock depends on a particular timing; it may not show up even once in a thousand runs. Correctness is shown by considering every possible ordering. In practice the most reliable approach is to acquire locks in the same order everywhere.

7.2 The four conditions for deadlock

What made deadlock possible in the reverse-order example? The resources could not be shared, the processes held one resource while waiting for another, nobody could take a resource back by force, and the waiting formed a ring. In 1971 Coffman, Elphick, and Shoshani showed that these four properties are necessary for deadlock.

7.2.1 The Coffman conditions

Theorem 7.1 (The four conditions for deadlock)

For deadlock to occur, the following four conditions must hold at the same time:

1. **Mutual exclusion**: each resource is either assigned to exactly one process or free; it cannot be shared at the same time.
2. **Hold and wait**: a process holding at least one resource requests new resources held by others and waits for them.
3. **No preemption**: a resource that has been granted cannot be taken back by force; only the process holding it releases it, of its own accord.
4. **Circular wait**: T_0 waits for a resource held by T_1; T_1 for one held by T_2; ...; and T_n for one held by T_0.

Figure 7.4: Circular wait. T0 holds R1 and waits for R2; T1 holds R2 and waits for R3; T2 holds R3 and waits for R1. The last arrow closes the ring.

That the conditions are "jointly necessary" means that deadlock is impossible in a system where even one of them is missing. If a resource can be shared, nobody waits for it. If no process waits while holding a resource, no ring can be built. If resources can be taken back, the ring can be broken. And if no ring ever forms, every waiting process eventually gets its turn. The **prevention** strategy rests on exactly this observation: make one of the four conditions structurally impossible.

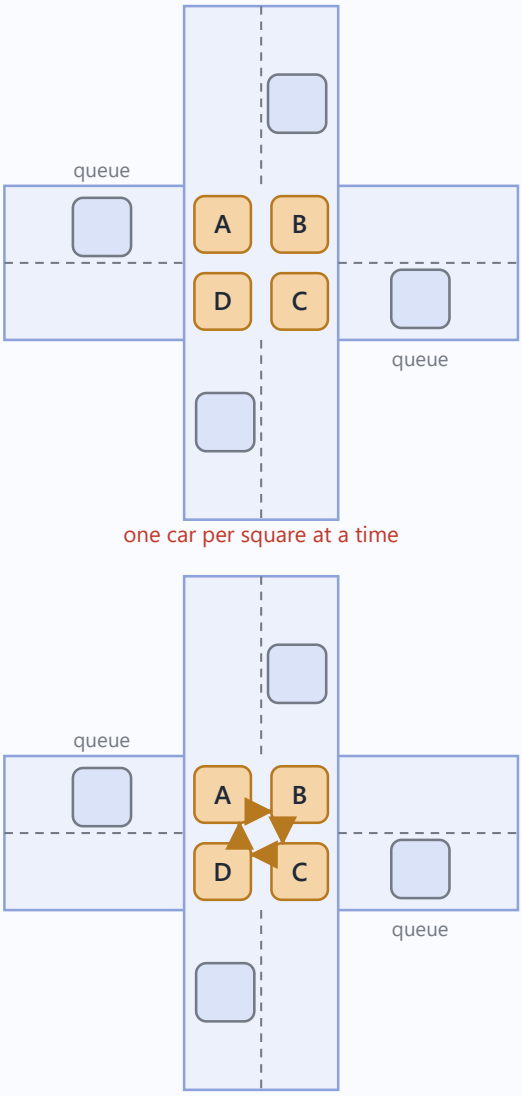
The four conditions are not of the same nature. The first three are the system's *policy*: whether resources are shared, how processes request resources, and whether resources can be taken back are fixed at design time. When these three policies hold, deadlock *can* occur, but need not. The fourth condition is the system's *current state*: depending on the order in which processes request resources, the ring either forms or does not. If there is only one instance of each resource type, the four conditions are also sufficient: if all four hold, the system is deadlocked. With multiple-instance resources the sufficiency disappears: another instance of a resource in the cycle may be held by a process outside the cycle.

Common mistake

Using the conditions interchangeably. "No preemption" means that a resource cannot be taken back by force; "hold and wait" means that a process holds one resource while waiting for another. Requiring a process to release what it holds before making a new request breaks hold and wait. Having the operating system take a held resource by force breaks no preemption.

7.2.2 The intersection example

To make the four conditions concrete, return to the intersection. Its four squares are four nonpreemptable resources, and the cars are the processes. Figure 7.5 marks the conditions one by one.


queue
A
B
D
C
queue
one car per square at a time
queue
A
B
D
C
queue

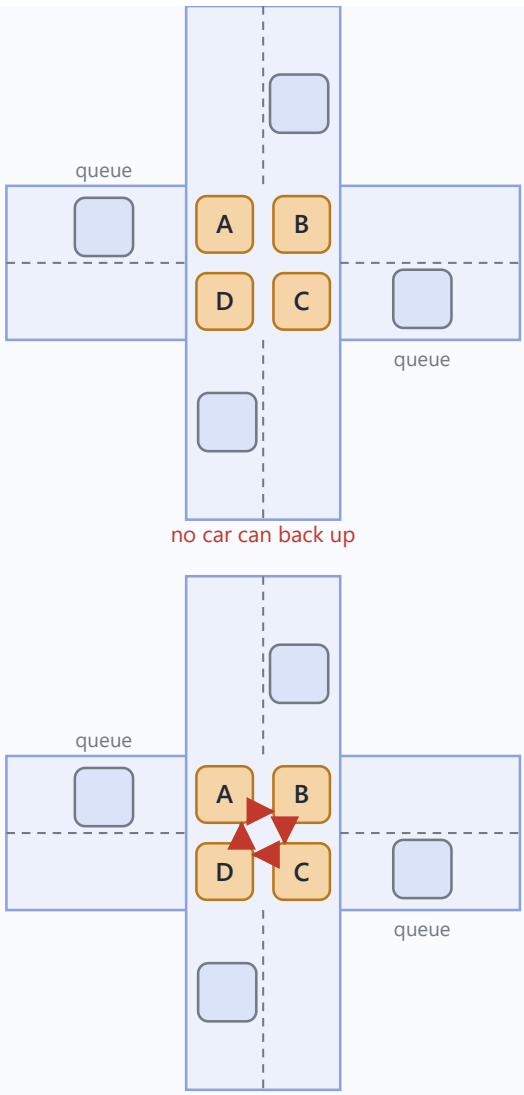


Figure 7.5: The four conditions at the intersection. (a) Mutual exclusion: only one car can be in a square at a time. (b) Hold and wait: each car holds its square while waiting for the square ahead. (c) No preemption: there is a queue behind, so no car can back up. (d) Circular wait: A → B → C → D → A.

1. **Mutual exclusion.** Two cars do not fit in the same square. A square is either empty or belongs to a single car.
2. **Hold and wait.** Each car holds the square it has entered and waits for the square ahead in order to move on.
3. **No preemption.** There is a queue behind each car; none of them can back up and vacate the square.
4. **Circular wait.** A waits for B's square, B for C's, C for D's, and D for A's. The ring is closed and the traffic is deadlocked.

The remedies in traffic also break one of the conditions. If a traffic officer makes a car back up, no preemption is broken. The rule “do not enter unless the intersection is clear” keeps a car from holding one square while waiting for another, so hold and wait disappears. A traffic light gives the

directions turns and keeps the ring from forming. The prevention methods in computer systems correspond to these three ideas.

7.3 Modeling deadlock

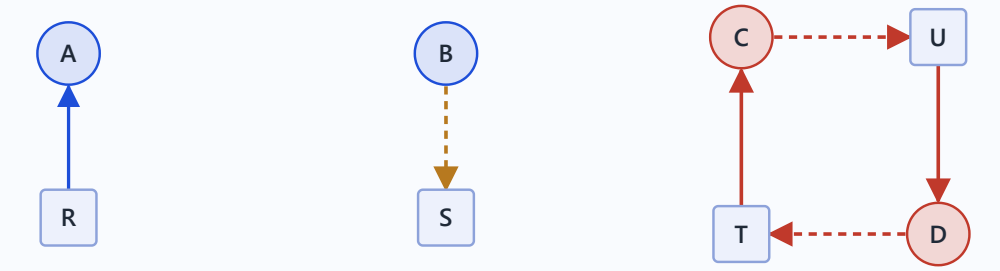
To talk about deadlock we need a tool that shows at a glance who holds what and who waits for what. The resource-allocation graph that Holt proposed in 1972 serves this purpose. The graph makes it easier to understand how deadlock arises, and it is also the data structure on which the detection algorithms work.

7.3.1 The resource-allocation graph

Definition 7.3

A **resource-allocation graph** is a directed graph with two kinds of nodes. Processes are drawn as circles and resources as squares. An **assignment edge** R → P from a resource to a process shows that R has been given to P. A **request edge** P → R from a process to a resource shows that P is waiting for R.

When a request is granted, the request edge turns into an assignment edge, that is, the direction of the arrow is reversed. When the resource is released, the assignment edge is deleted. For a resource type with several instances, one dot per instance is placed inside the square. An assignment edge starts at one of these dots; a request edge goes to the square itself, because the process waits not for a particular instance but for any instance of the type. In these notes request edges are drawn dashed and orange.



(a) A holds R (b) B requests S (c) Deadlock: C → U → D → T → C

Figure 7.6: The notation of the resource-allocation graph. (a) Assignment edge from R to A: A holds R. (b) Request edge from B to S: B wants S. (c) C waits for U; U is held by D; D waits for T; T is held by C. The cycle C → U → D → T → C in the graph is a deadlock.

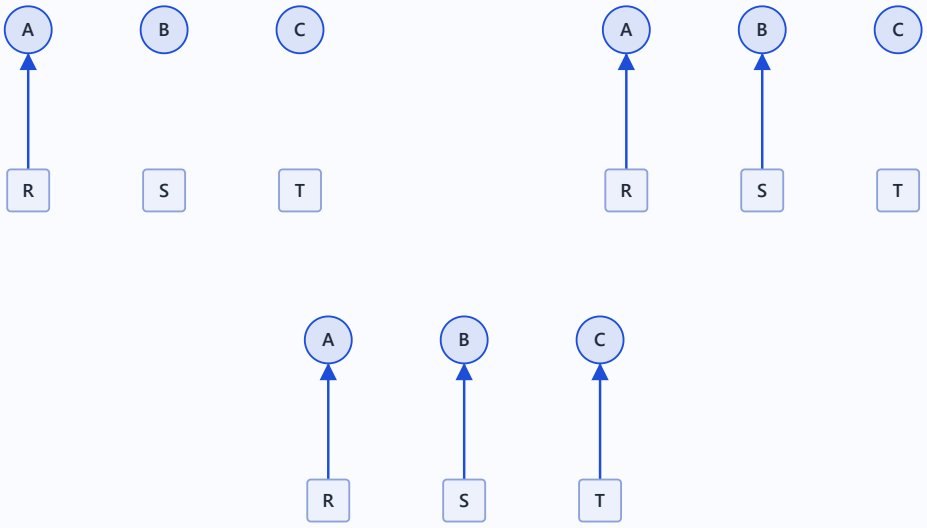
Tanenbaum draws the same model with process nodes. Silberschatz instead calls the nodes threads and labels them with the letter T. There is no difference in meaning between the two notations. In these notes, each example keeps the naming of the source it was taken from.

7.3.2 How deadlock arises

Consider three processes (A, B, C) and three resources (R, S, T). Each process requests two resources and releases both:

```
A          B          C
request R  request S  request T
request S  request T  request R
release R  release S  release T
release S  release T  release R
```

Suppose the operating system schedules the processes round robin and satisfies requests in the order they arrive. The first requests of the processes arrive in turn, and since all three resources are free, all are satisfied. When the second requests arrive, each process asks for the resource held by the next one. Figure 7.7 shows the six requests and the state of the graph at each step.



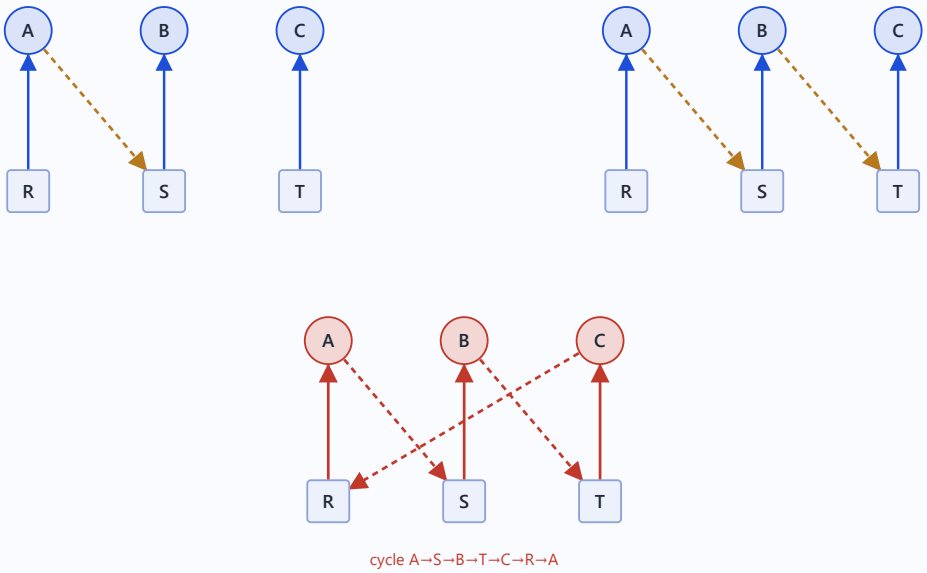


Figure 7.7: How a deadlock forms (Tanenbaum Figure 6-4). (a) A gets R. (b) B gets S. (c) C gets T. (d) A requests S and blocks. (e) B requests T and blocks. (f) C requests R and blocks: the cycle A → S → B → T → C → R → A closes.

1. **A requests R.** R is free; the assignment edge R → A is added.
2. **B requests S.** S is free; S → B is added.
3. **C requests T.** T is free; T → C is added.
4. **A requests S.** S is held by B. A blocks; the request edge A → S is added.
5. **B requests T.** T is held by C. B blocks; B → T is added.
6. **C requests R.** R is held by A. C blocks; C → R is added. The graph now contains the cycle A → S → B → T → C → R → A, and all three processes are deadlocked.

This outcome was not inevitable. The scheduler could have picked a different order. For example, if it had run A until A had made both requests and released its resources, then B, then C, no request would ever have waited. Parallelism would have been lost, but there would have been no deadlock. Deadlock depends on the order of the requests.

7.3.3 How deadlock can be prevented

If the operating system knew in advance that granting a request would lead to deadlock, it could prevent the deadlock by making the dangerous request wait. In the same example, suppose the operating system suspends B's requests for a while, that is, does not schedule B. Only A and C run. Figure 7.8 follows this ordering.

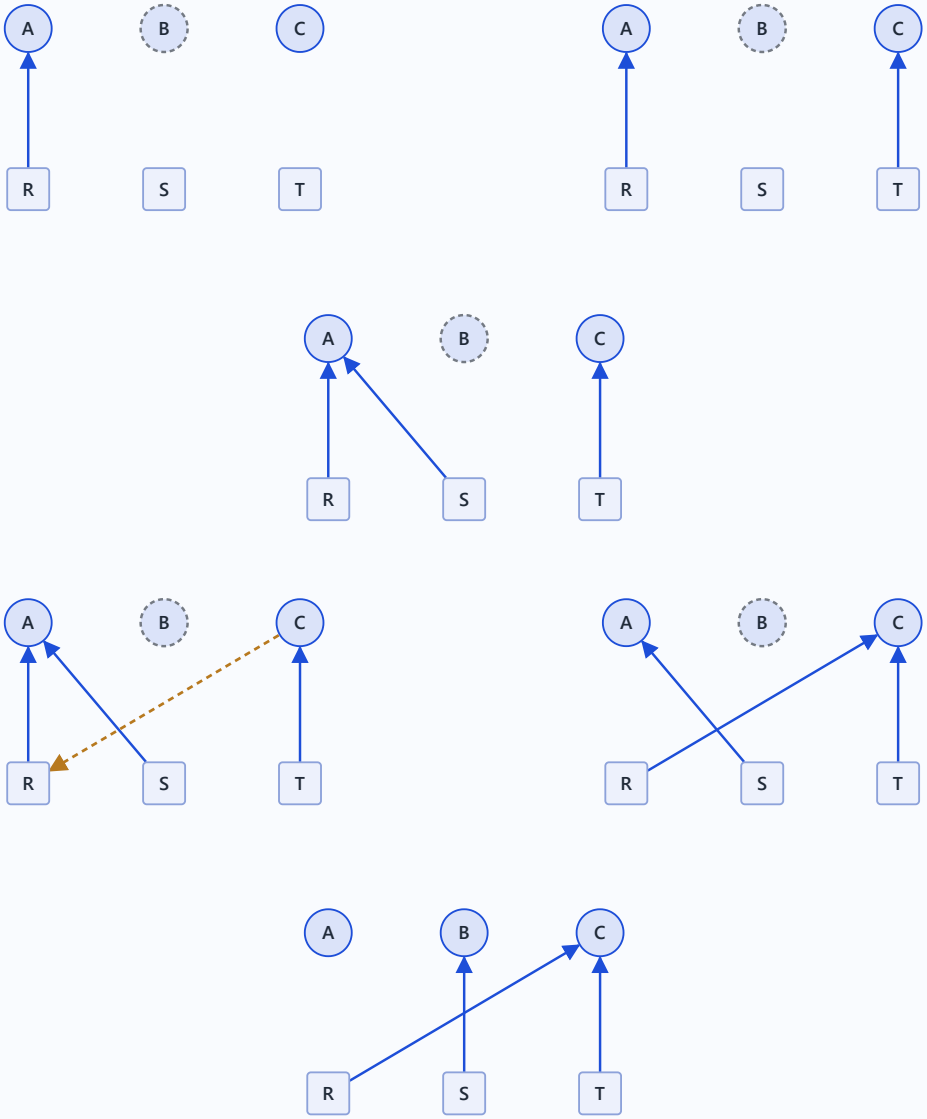


Figure 7.8: No deadlock when B is postponed (Tanenbaum Figure 6-4). (a) A gets R. (b) C gets T. (c) A gets S. (d) C requests R and blocks; there is no cycle. (e) A releases R; R is given to C. (f) A releases S and finishes; now B can run.

1. **A gets R; C gets T.** Both resources are free.
2. **A gets S.** S is free, because B has not requested it yet.
3. **C requests R.** R is held by A; C blocks. The graph has a request edge but no cycle. A is waiting for nothing and can proceed.
4. **A releases R.** R is given to the waiting C, and C proceeds.
5. **A releases S and finishes.** Now B is run. B gets S, and gets T when C finishes. There is no deadlock.

What the operating system does here is the essence of the avoidance strategy: which request should be postponed? To choose the request to postpone, one must know how many resources each process may ask for. The banker's algorithm uses this information.

7.3.4 Resources with multiple instances

In the examples so far each resource type had a single instance. Silberschatz's classic example also includes multiple-instance types. The resource sets are as follows: R1 has one instance, R2 two, R3 one, and R4 three. The state of the three threads is as follows:

- T1 holds one instance of R2 and waits for R1.
- T2 holds R1 and one instance of R2, and waits for R3.
- T3 holds R3.

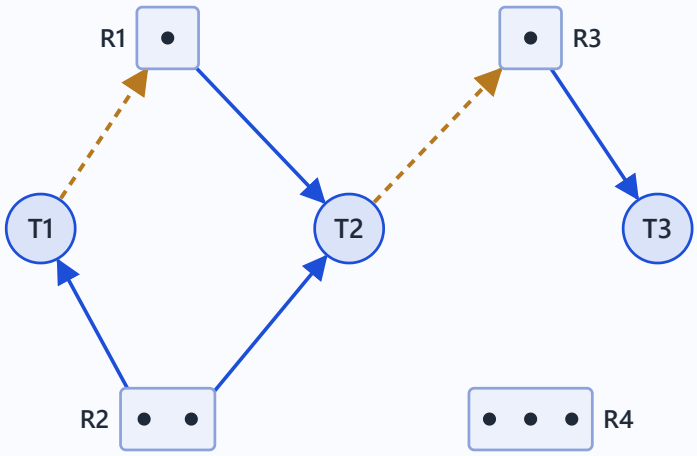


Figure 7.9: An allocation graph with multiple-instance resources (Silberschatz Figure 8.4). An assignment edge starts from a dot in the square, that is, from a particular instance; a request edge goes to the square itself. All three instances of R4 are idle.

The graph has no cycle. Since T3 waits for nothing, it finishes and releases R3. R3 is given to T2; T2 also finishes and releases R1 and its instance of R2. R1 is given to T1, and T1 finishes as well. The finishing order is T3, T2, T1. If there is no cycle, there is no deadlock; this rule is independent of the number of instances.

7.3.5 Cycles and deadlock

Add a single request to the same situation: T3 asks for an instance of R2. Both instances of R2 are in use (one by T1, one by T2), so T3 blocks. This one request edge creates two cycles in the graph:

- Cycle 1: T1 → R1 → T2 → R3 → T3 → R2 → T1.

- Cycle 2: T2 → R3 → T3 → R2 → T2.

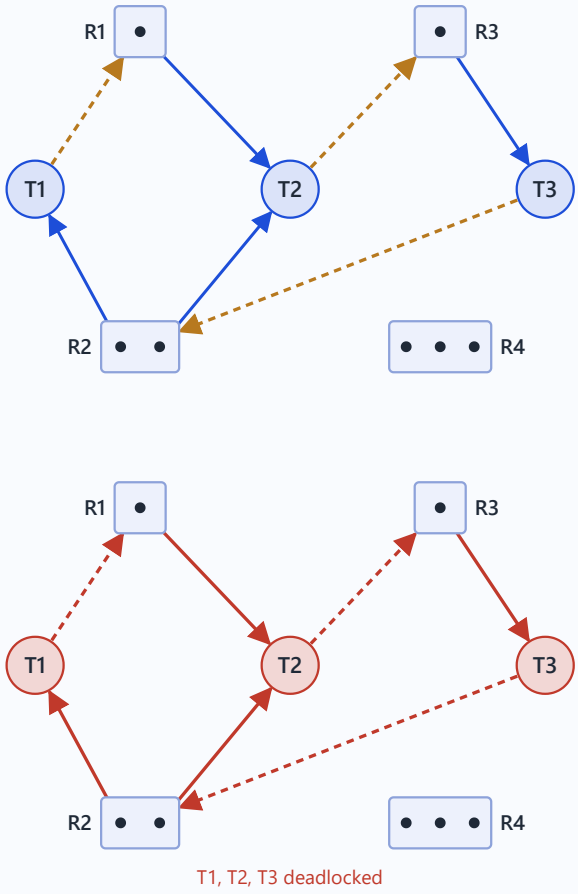


Figure 7.10: A deadlocked multiple-instance graph (Silberschatz Figure 8.5). (a) T3 requests an instance of R2; both instances are in use and T3 blocks. (b) Two cycles form; the owners of both instances of R2 are on the cycles. T1, T2, and T3 are deadlocked.

In this case there is no rescuer. T3 waits for one of the instances of R2 to be released, but the owners of both instances (T1 and T2) are on the cycle. T2 waits for R3, which T3 holds, and T1 waits for R1, which T2 holds. Every process on the cycle waits for another process on the cycle, and there is no one outside to release a resource. All three threads are deadlocked.

7.3.6 A cycle without deadlock

With multiple-instance resources a cycle does not always mean deadlock. In Silberschatz's second example, R1 and R2 have two instances each. The two instances of R1 are held by T2 and T3, and the two instances of R2 by T1 and T4. T1 requests an instance of R1, and T3 an instance of R2. The graph contains the cycle T1 → R1 → T3 → R2 → T1.

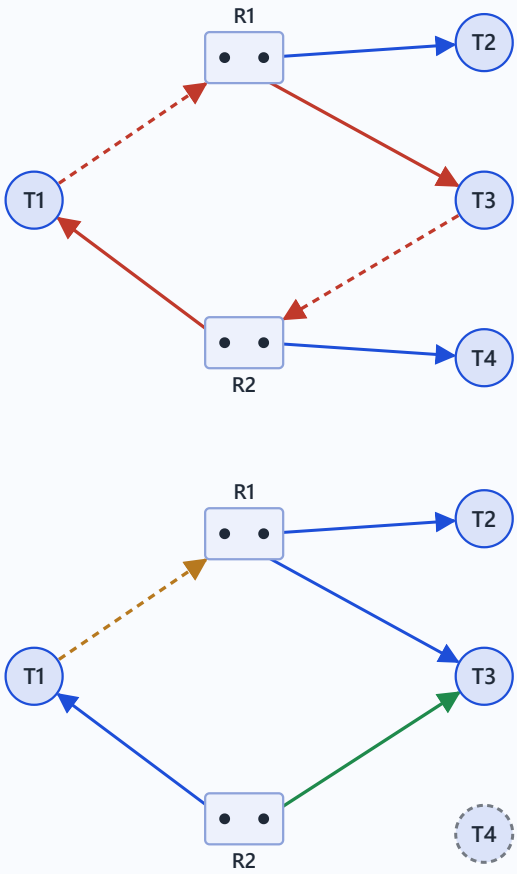


Figure 7.11: A cycle but no deadlock (Silberschatz Figure 8.6). (a) The cycle T1 → R1 → T3 → R2 → T1; T2 and T4 are outside the cycle and wait for nothing. (b) T4 finishes and releases its instance of R2; the instance is given to T3, and the cycle is broken.

The difference lies in the processes outside the cycle. T2 and T4 wait for nothing; they finish and release their instances. The instance of R2 released by T4 is given to T3, and the cycle is broken. The instance of R1 released by T2 can also go to T1. In the end everyone finishes; the system was never deadlocked.

Example 7.1

R1 has two instances, and R2 and R3 have one each. P1 holds R2 and requests R1. P2 holds one instance of R1 and requests R3. P3 holds R3 and requests R2. P4 holds the other instance of R1 and requests nothing. Is there a cycle in the graph? Is the system deadlocked?

1. **The cycle.** P1 → R1 → P2 → R3 → P3 → R2 → P1 is a cycle. If the resources had a single instance each, this would be a certain deadlock.
2. **An instance outside the cycle.** The second instance of R1 is held by P4, which is not on the cycle. P4 waits for nothing, finishes, and releases its instance of R1.

3. **A chain of completions.** P1's request for R1 is satisfied; P1 finishes and releases R1 and R2. P3 gets R2 and finishes, freeing R2 and R3. P2 gets R3 and finishes.
4. **Result.** The finishing order is P4, P1, P3, P2. Because everyone can finish, there is no deadlock.

7.3.7 The cycle rule

The conclusions from the examples are summarized in Table 7.4. If there is no cycle, there is no deadlock under any circumstances. If there is a cycle, the outcome depends on the number of instances.

Graph	One instance of every type	Several instances of some types
No cycle	no deadlock	no deadlock
Cycle	deadlock (cycle ⇔ deadlock)	deadlock possible; a cycle is necessary but not sufficient

Table 7.4: The relation between a cycle in the allocation graph and deadlock.

Theorem 7.2 (The cycle rule)

If the resource-allocation graph has no cycle, the system is not deadlocked. If every resource type has a single instance, a cycle is both a necessary and a sufficient condition for deadlock, and the processes on the cycle are deadlocked. If some types have several instances, a cycle is necessary but not sufficient.

Proof. In the single-instance case every resource on the cycle has exactly one holder, and that holder waits for another resource on the cycle. Since the only process that could release the resource is itself waiting, the resource never becomes free, and none of the processes on the cycle can proceed. In the multiple-instance case another instance of a resource on the cycle may be held by a process outside the cycle. When that process finishes and releases the instance, the request on the cycle is satisfied. That is why a cycle does not guarantee deadlock.

For single-instance resources, depth-first search is enough to find the cycle, and its cost is $O(V + E)$ (V nodes, E edges). For multiple-instance resources a cycle is only a warning. To decide for certain, one asks, using the allocation and request matrices, “can everyone finish in some order?” The theorem is also consistent with the four conditions: circular wait corresponds to a graph that contains a cycle.

7.3.8 The wait-for graph

If every resource type has a single instance, the allocation graph can be simplified further. Because a resource has a single holder, the information “P_i waits for R and R is held by P_j” simply means “P_i waits for P_j.”

Definition 7.4

A **wait-for graph** is a directed graph consisting only of process nodes. Every path $P_i \to R \to P_j$ in the allocation graph is collapsed into a single edge $P_i \to P_j$; the resource nodes are removed.

Figure 7.12 shows the transformation on an example with four processes. P1 holds R1 and requests R2. P2 holds R2 and requests R3. P3 holds R3 and requests R1. P4 holds R4 and requests R2.

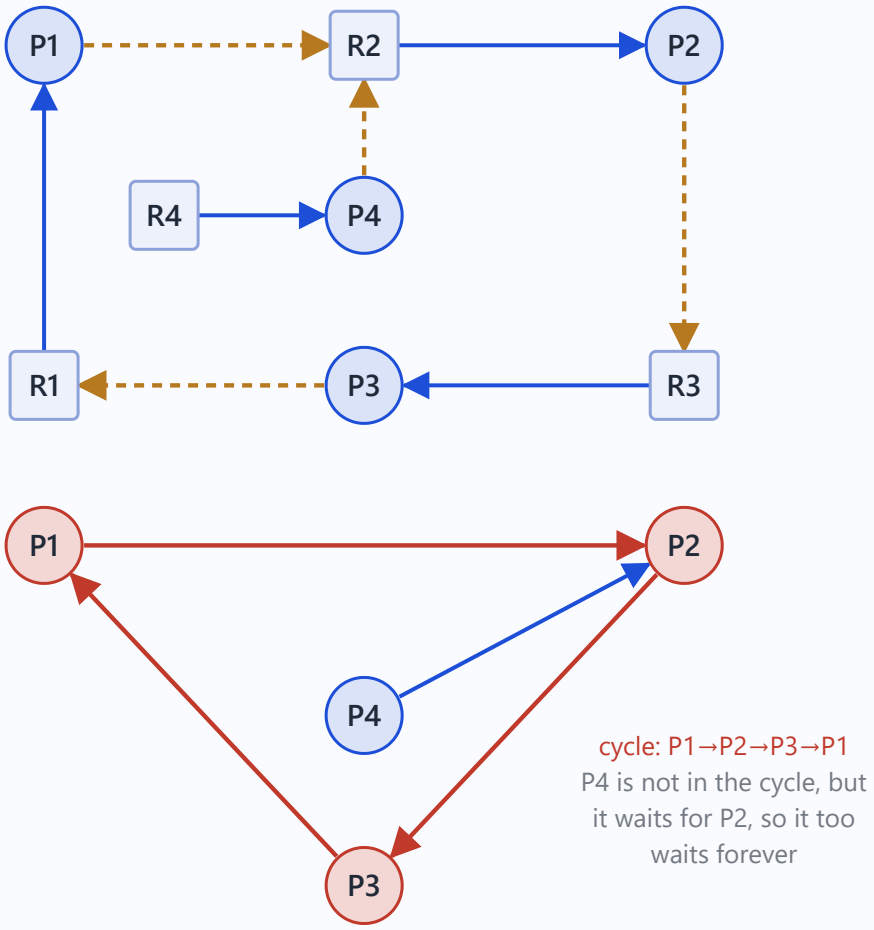


Figure 7.12: From the allocation graph to the wait-for graph. Top: the allocation graph with single-instance resources; bottom: the wait-for graph after the resource nodes are removed. The cycle P1 → P2 → P3 → P1 is a deadlock; P4 is not on the cycle, but because it waits for P2 it also waits forever.

The edges are obtained like this: P1 waits for R2 and R2 is held by P2, hence P1 → P2. In the same way P2 → P3, P3 → P1, and P4 → P2 are found. A cycle appears in the wait-for graph if and only if there is a deadlock. The operating system maintains this graph and runs a cycle-searching algorithm at intervals. Silberschatz gives the cost of this scan as $O(n^2)$ (n: number of processes).

The wait-for graph can be used only when every type has a single instance; for a multiple-instance type the “holder of R” is not a single process. Databases use this method widely: PostgreSQL and MySQL InnoDB maintain a wait-for graph for row locks and abort one of the transactions when they find a cycle.

Common mistake

Assuming that processes not on the cycle can always proceed. In the wait-for graph P4 is not on the cycle, but the P2 it waits for is. Since P2 will never finish, P4 can never proceed either. The deadlocked set is only the processes on the cycle; processes connected to the cycle by a path are indirect victims of the deadlock.

7.4 Strategies for coping

There are four ways to cope with deadlock. They differ in the following questions: Does the system allow deadlock at all? When is the decision made: at design time, at run time, or after the deadlock has occurred? What information does the decision need?

7.4.1 Four strategies

1. **Ignoring it.** Also called the “ostrich algorithm.” If deadlock is rare and prevention is expensive, the problem is ignored; if a deadlock occurs, the system or the process is restarted.
2. **Detection and recovery.** A reactive approach. Deadlock is allowed; the system checks at intervals whether a deadlock has occurred and, if it finds one, resolves it by preempting resources, rolling back, or terminating.
3. **Dynamic avoidance.** A proactive approach that decides at run time. A request that would put the system into an unsafe state is postponed. The banker's algorithm is an example. It requires that the maximum need of every process be known in advance.
4. **Prevention.** A proactive approach that decides at design time. At least one of the four conditions is made structurally impossible; lock ordering is the best-known example.

“Recovery” is not listed as a separate strategy because recovery is the second half of detection: first the deadlock is found, then it is resolved. The difference between avoidance and prevention is also often confused. Prevention removes one of the conditions for deadlock from the system altogether; since the condition can never be met, deadlock is impossible. Avoidance, on the other hand, allows all four conditions, but at every request it computes where the state is heading and makes dangerous requests wait.

7.4.2 The ostrich algorithm

The simplest strategy is to do nothing. It is named after the ostrich, which is believed to bury its head in the sand in the face of danger. If a deadlock occurs, the user or administrator kills the stuck process and, if necessary, restarts the system.

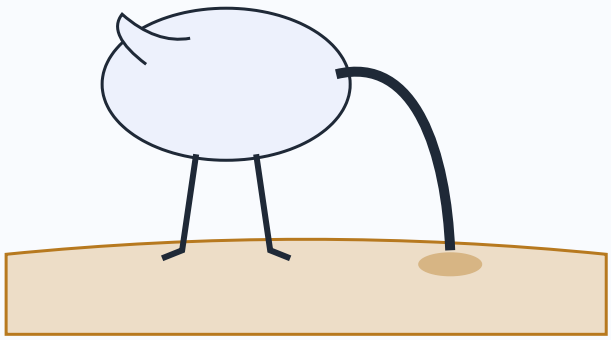


Figure 7.13: The ostrich algorithm: ignoring the problem.

Although it looks irresponsible at first sight, this choice is an engineering trade-off. If a deadlock happens once a year while crashes and hardware faults are far more frequent, costly and restrictive precautions are not worth it. An avoidance algorithm that runs at every request slows down the whole system, and rules imposed on resource requests make the programmer's job harder.

General-purpose operating systems such as Linux, Windows, and macOS apply this approach to user processes' locks, file locks, and pipe waits. The responsibility lies with the application developer. Inside itself, however, the operating system does take precautions against deadlock: locks in the kernel are acquired in a documented order, and the `lockdep` tool in Linux checks that order. In real-time and safety-critical systems the ostrich approach is unacceptable; flight-control software cannot be designed on the principle that it is "restarted now and then."

The ostrich approach is not a middle path between detection and avoidance. It does neither; it merely compares the probability and cost of deadlock with the cost of prevention.

7.5 Detection and recovery

In the detection-and-recovery strategy the system places no restriction on resource requests. Requests are satisfied as they arrive, and if the resource is not free the process is made to wait. From time to time the system runs an algorithm to check whether a deadlock has occurred. The algorithm differs depending on whether the resource types have one instance or several.

7.5.1 One resource of each type

If every resource type has a single instance, deadlock is the same thing as a cycle in the allocation graph. Tanenbaum's example has seven processes (A–G) and six resources (R–W):

- A holds R and requests S.
- B holds nothing but requests T.
- C holds nothing but requests S.
- D holds U and requests S and T.
- E holds T and requests V.
- F holds W and requests S.
- G holds V and requests U.

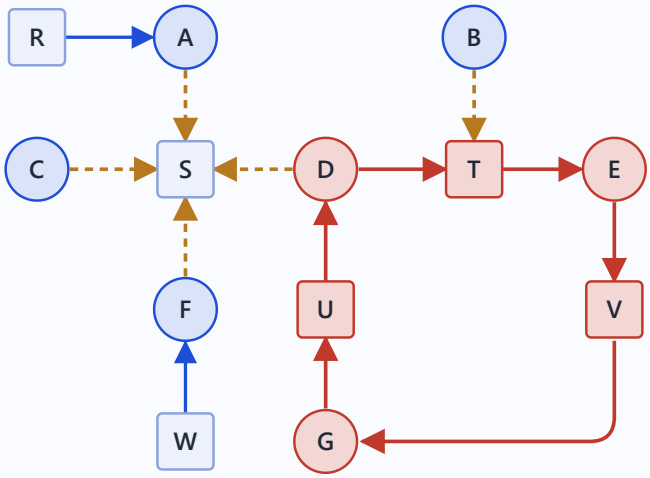


Figure 7.14: An allocation graph with seven processes and six resources (Tanenbaum Figure 6-5). The highlighted cycle D → T → E → V → G → U → D is the deadlock.

The question is: is the system deadlocked, and if so, which processes are involved? Once the graph is drawn, the answer can be found by eye. D → T → E → V → G → U → D is a cycle; D, E, and G are deadlocked. A, C, and F are not deadlocked. S is free; one of the three takes S, finishes and releases it, and then another takes it. B's case is different: B is not on the cycle, but because it waits for T, which is on the cycle, it cannot proceed either. In this small graph the eye is enough. For a graph with hundreds of nodes a systematic algorithm is needed.

7.5.2 The cycle-search algorithm

The algorithm given by Tanenbaum takes each node of the graph in turn as a starting point and proceeds depth-first from it. The path traversed is kept in a list *L*. If a node is added to *L* a second time, a cycle has been found.

1. For each node N in the graph, perform steps 2–6 starting from N.
2. Make L the empty list; treat all edges as unmarked.
3. Append the current node to the end of L. If the node now appears twice in L, the graph contains a cycle: report the cycle in L and stop.
4. Is there an unmarked edge leaving the current node? If so, go to step 5; if not, go to step 6.
5. Pick one of the unmarked edges, mark it, and move to the node it points to; return to step 3.
6. If we are at the starting node, there is no cycle reachable from N. If not, we are in a dead end: remove the node from L, go back to the previous node, and continue from step 4.

The essence of the algorithm is a depth-first search (DFS) from each node; the list L plays the role of the stack. Because this simple version restarts from every node, it takes $O(V \cdot (V + E))$ time in the worst case. The version that colors the nodes with three colors runs in a single pass in $O(V + E)$. In the output, the part of L after the repeated node is the cycle, and the processes in it are deadlocked.

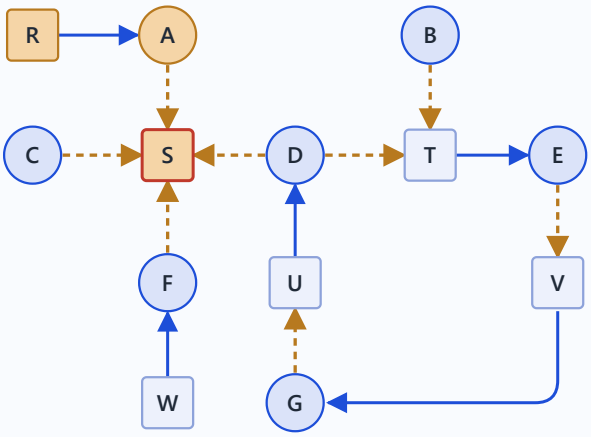
When step 6 says “go back to the previous node,” that node is already in L. That is why we continue from step 4, not step 3; going to step 3 would add the node a second time and report a false cycle. Tanenbaum's text says briefly “go to step 3” at this point; what is meant is to look at the node's outgoing edges without adding it again.

7.5.3 Cycle search step by step

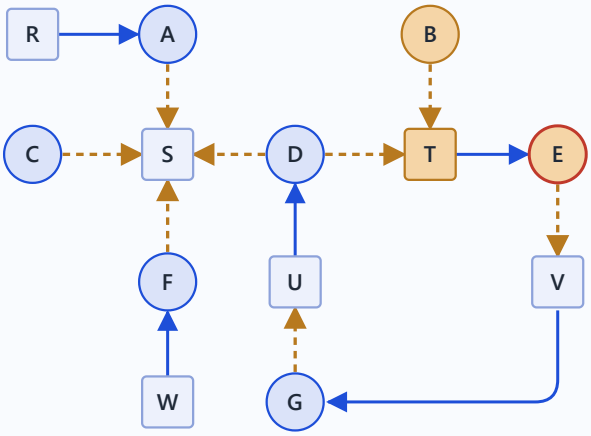
Let us run the algorithm on the graph of Figure 7.14. The starting nodes are chosen in the order of the figure, that is, left to right and top to bottom: R, A, B, C, S, D, T, E, F, U, V, W, G. If several edges leave a node, the choice is arbitrary; here, from D the edge to S is chosen first, then the edge to T. Table 7.5 gives the state of L at each step.

Start	Event	L
R	R is added; the only exit is R → A	[R]
R	A is added; A → S	[R, A]
R	S is added; no edge leaves S	[R, A, S]
R	dead end: S is removed, back to A; A has no unmarked edge	[R, A]
R	A is removed, back to R; R is the start: no cycle from R	[R] → []
A	A → S, S is a dead end; no cycle from A	[A] → [A, S] → []
B	B → T → E	[B, T, E]
B	E → V → G → U	[B, T, E, V, G, U]
B	U → D; S is tried before T from D	[B, T, E, V, G, U, D, S]
B	S is a dead end and is removed from L; back to D	[B, T, E, V, G, U, D]
B	D → T: T is added a second time → cycle	[B, T, E, V, G, U, D, T]

Table 7.5: Trace of the cycle-search algorithm.




```
L = [R, A, S]
```




```
L = [B, T, E]
```

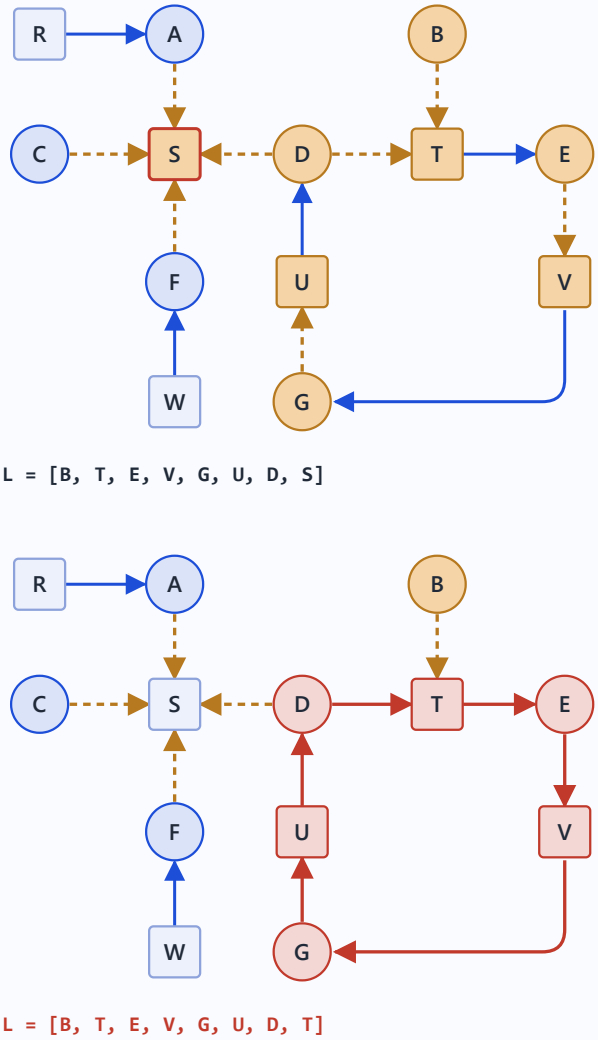


Figure 7.15: Four moments of the cycle search. The orange nodes are in the current list L, and the node with a red outline is the current node. (a) R → A → S reaches a dead end. (b) The search starts from B: B → T → E. (c) After U → D, S is tried. (d) With D → T, T is added a second time: the cycle is T, E, V, G, U, D.

The algorithm stops in the search that starts from B. In the list L = [B, T, E, V, G, U, D, T], T appears twice. The part after the first occurrence of the repeated node, that is, T, E, V, G, U, D, is the cycle. The processes on the cycle are D, E, and G; they are deadlocked. B is at the head of the list but not on the cycle: it lies on a path leading to the cycle. This is the same result we found by inspecting the graph by eye. These results were checked by rerunning the same algorithm in Python.

7.5.4 Code trace: colored depth-first search

Restarting from every node does unnecessary work. Colored DFS gives each node a color. A **white** node has not been visited yet. A **gray** node is currently on the stack, that is, on the path being followed. A **black** node has had all its neighbors processed and is finished. An edge to a gray node is called a **back edge** and shows that there is a cycle. The code below stores the wait-for graph as an adjacency matrix `waits[u][v]` : if `waits[u][v]` is 1, u waits for v.

```
enum { WHITE, GRAY, BLACK };
int color[N];          /* all WHITE */
bool cycle(int u) {
  color[u] = GRAY;     /* on path */
  for (int v = 0; v < N; v++) {
    if (!waits[u][v]) continue;
    if (color[v] == GRAY)
      return true;     /* back edge */
    if (color[v] == WHITE && cycle(v))
      return true;
  }
  color[u] = BLACK;    /* done */
  return false;
}
```

The logic of the code is as follows. When `cycle(u)` is called, u is colored gray. Then every v that u waits for is tried. If v is gray, the path has come back onto itself and `true` is returned. If v is white, `cycle(v)` is called recursively; if that call finds a cycle, `true` is passed up. Black neighbors are skipped, because it is already known that no cycle leads out of them. When all neighbors are done, u is colored black and `false` is returned. Each node is colored gray at most once and each edge is examined once, so the cost is $O(V + E)$. Because finding the edges with an adjacency matrix takes N steps at every node, the actual cost of this code is $O(N^2)$.

The graph has five processes, with edges P1 → P2, P2 → P3, P2 → P4, P4 → P5, and P5 → P2. The search starts from P1. Because the `for` loop tries neighbors in numerical order, from P2 it goes to P3 first and then to P4. Table 7.6 shows the colors at each step, and Figure 7.16 shows the important moments.

Step	Event	P1	P2	P3	P4	P5	Call stack
0	start	W	W	W	W	W	—
1	`cycle(P1)` : P1 gray	G	W	W	W	W	P1
2	P1 → P2 white: `cycle(P2)`	G	G	W	W	W	P1, P2
3	P2 → P3 white: `cycle(P3)`	G	G	G	W	W	P1, P2, P3
4	P3 has no neighbors: black, `false`	G	G	B	W	W	P1, P2
5	P2 → P4 white: `cycle(P4)`	G	G	B	G	W	P1, P2, P4
6	P4 → P5 white: `cycle(P5)`	G	G	B	G	G	P1, P2, P4, P5
7	P5 → P2: P2 is gray → back edge, `true`	G	G	B	G	G	P1, P2, P4, P5
8	`true` is passed up the stack	G	G	B	G	G	—

Table 7.6: Trace of colored DFS (W: white, G: gray, B: black).

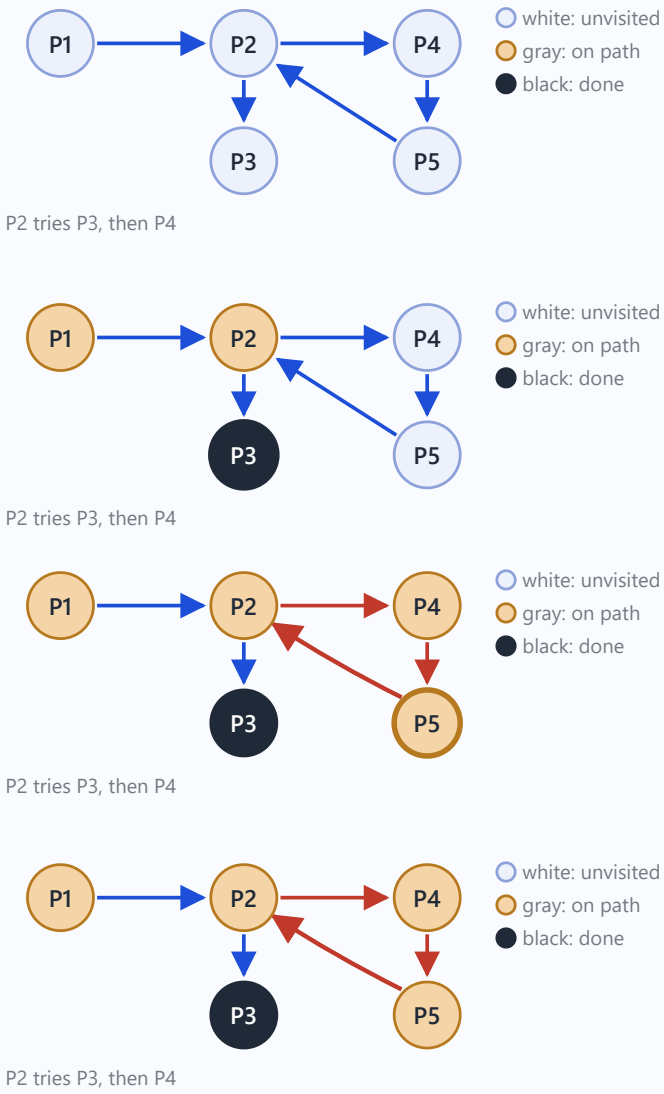


Figure 7.16: Colored DFS. (a) At the start all nodes are white. (b) P3 waits for no one and is colored black. (c) On the edge P5 → P2, P2 is gray: a back edge, that is, a cycle. (d) P2, P4, and P5 on the cycle are deadlocked; P1 is not on the cycle but is stuck because it waits for P2.

The cycle is P2 → P4 → P5 → P2, and these three processes are deadlocked. The gray nodes form the path on the stack at that moment; the cycle is the part from the gray node that the back edge points to (P2) up to the top of the stack (P5). P1 is gray too, but because the back edge goes to P2 and not to P1, P1 is not on the cycle. Since P1 waits for P2, it cannot proceed all the same.

Common mistake

Counting an edge to a black neighbor as a cycle too. A black node was finished earlier, and the paths leaving it have been examined and found to be acyclic. An edge to a black node only shows that two paths merge. For example, if there were also an edge P1 → P3, then while P3 is black this edge would not form a cycle. The sign of a cycle is only an edge to a gray node.

7.5.5 Several resources of each type: vectors and matrices

If resource types have several instances, searching for cycles is not enough. Instead the allocation and request information is kept in matrices, and the question asked is “can everyone finish in some order?” Let there be *n* processes and *m* resource types. The algorithm uses four data structures.

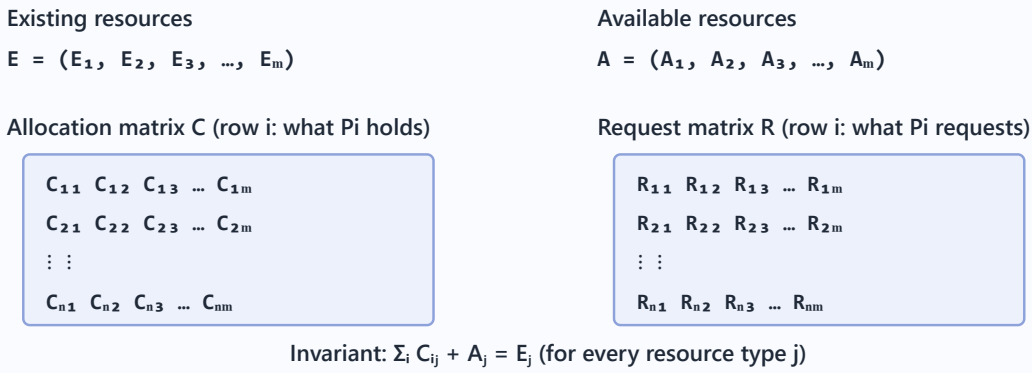


Figure 7.17: The data structures of the detection algorithm (Tanenbaum Figure 6-6): existing resources *E*, available resources *A*, the allocation matrix *C*, and the request matrix *R*. For every type *j*, the held and available instances add up to the number of existing instances.

- ***E*** (existing resources): E_j is the total number of instances of type *j* in the system.
- ***A*** (available resources): A_j is the number of instances of type *j* that are free right now.
- ***C*** (allocation matrix): row *i* of *C* holds the resources P_i currently holds.
- ***R*** (request matrix): row *i* of *R* holds the resources P_i currently requests and is waiting for.

There is an invariant relation among these structures: for every type *j*, $\Sigma_i\ C_{ij} + A_j = E_j$. A resource instance is either held by a process or free. Silberschatz gives the same structures under the names *Available*, *Allocation*, and *Request*.

The algorithm compares rows as vectors. For two vectors, $X \le Y$ holds if and only if $X_j \le Y_j$ for every *j*. For example, (1 0 1 0) ≤ (2 1 0 0) is false: the first two components match, but in the third component 1 > 0. For vectors, “smaller” must hold in every component separately; comparing the sums would give a wrong result.

7.5.6 The detection algorithm

The idea of the algorithm is simple. A process whose request can be satisfied by the currently free resources can run and finish. When it finishes it returns all the resources it holds to the free pool; because the pool has grown, another process's request may now be satisfiable. This repeats until no process is left or none of the remaining ones can be satisfied.

```
A ← available vector
all Pi unmarked
repeat:
  find an unmarked Pi with Ri ≤ A
  if none is found: exit
  A ← A + Ci   /* Pi ends, releases */
  mark Pi
unmarked Pi's are deadlocked
```

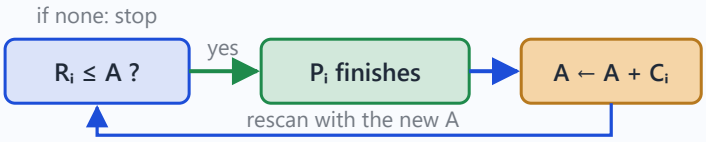


Figure 7.18: The detection loop. A process whose request can be met from the available vector finishes and adds what it holds to A; the scan is repeated with the new A. The algorithm stops when no row can be satisfied.

The line "$A \leftarrow A + C_i$" says that the resources held by the finished process return to the free pool. The processes that remain unmarked at the end are deadlocked. This result rests on an optimistic assumption: a process whose request can be satisfied is assumed to finish without asking for more resources. If the process makes a new request before finishing and that request leads to deadlock, the situation will be caught the next time the algorithm runs.

The cost is computed as follows. In each round at most n rows are scanned, and each row comparison has m components. Since each successful round marks one process, there are at most n rounds. The total cost is $O(m \cdot n^2)$.

7.5.7 A detection example

Example 7.2

There are four resource types: tape drive (Tp), plotter (Pl), scanner (Sc), and CD drive. The existing resources are E = (4 2 3 1) and the available ones A = (2 1 0 0). The allocation and request matrices are as in the figure below (Tanenbaum Figure 6-7). Is the system deadlocked?

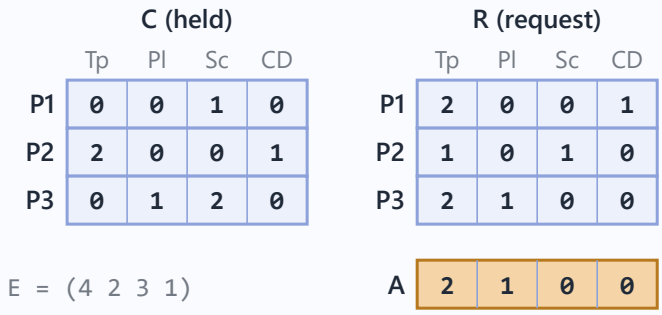


Figure 7.19: The initial state of the detection example: held resources C, requests R, and the available vector A.

1. **Check.** The column sums of C are (2 1 3 1); $E - (2\ 1\ 3\ 1) = (2\ 1\ 0\ 0) = A$. The data are consistent.
2. **P1.** $R_1 = (2\ 0\ 0\ 1) \le (2\ 1\ 0\ 0)$? In the CD component $1 > 0$: there is no free CD. P1 cannot finish for now.
3. **P2.** $R_2 = (1\ 0\ 1\ 0) \le (2\ 1\ 0\ 0)$? In the scanner component $1 > 0$: there is no free scanner.
4. **P3.** $R_3 = (2\ 1\ 0\ 0) \le (2\ 1\ 0\ 0)$ ✓. P3 can run and finish. When it finishes, $C_3 = (0\ 1\ 2\ 0)$ comes back: $A = (2\ 1\ 0\ 0) + (0\ 1\ 2\ 0) = (2\ 2\ 2\ 0)$. P3 is marked.
5. **New round, P1.** $(2\ 0\ 0\ 1) \le (2\ 2\ 2\ 0)$? CD is still 0; P1 waits again.
6. **P2.** $(1\ 0\ 1\ 0) \le (2\ 2\ 2\ 0)$ ✓. P2 finishes and $C_2 = (2\ 0\ 0\ 1)$ returns: $A = (4\ 2\ 2\ 1)$.
7. **New round, P1.** $(2\ 0\ 0\ 1) \le (4\ 2\ 2\ 1)$ ✓. P1 finishes and $C_1 = (0\ 0\ 1\ 0)$ returns: $A = (4\ 2\ 3\ 1) = E$.
8. **Result.** All processes are marked; the system is not deadlocked. The finishing order found is P3, P2, P1.

In the example the scan starts over from the beginning after each marking. So six comparisons are made in total: three in the first round, two in the second, one in the third. The order found depends on the order in which the algorithm scans the rows, but the deadlock verdict does not change. A process finishing can only enlarge the free pool, never shrink it. So a request that can be satisfied at one stage can still be satisfied at later stages. Whatever order is used for scanning, every process that can be marked is eventually marked. Table 7.7 summarizes how the vector A changes; all the values were recomputed with the algorithm itself in Python.

Round	Comparison	Result	A afterwards
1	P1: (2 0 0 1) ≤ (2 1 0 0)	✗ (CD)	(2 1 0 0)
1	P2: (1 0 1 0) ≤ (2 1 0 0)	✗ (scanner)	(2 1 0 0)
1	P3: (2 1 0 0) ≤ (2 1 0 0)	✓ P3 finishes	(2 2 2 0)
2	P1: (2 0 0 1) ≤ (2 2 2 0)	✗ (CD)	(2 2 2 0)

Round	Comparison	Result	*A* afterwards
2	P2: (1 0 1 0) ≤ (2 2 2 0)	✓ P2 finishes	(4 2 2 1)
3	P1: (2 0 0 1) ≤ (4 2 2 1)	✓ P1 finishes	(4 2 3 1)

Table 7.7: Change of the available vector in the detection example.

7.5.8 One request too many: deadlock

Change a single cell in the same situation. Let P3 request a CD as well, in addition to two tape drives and a plotter: R_3 = (2 1 0 1). The available vector is still A = (2 1 0 0).

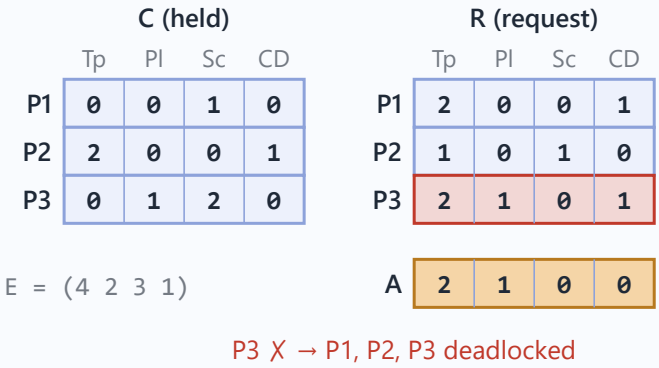


Figure 7.20: When P3's request is (2 1 0 1), no row can be satisfied with A = (2 1 0 0); P1, P2, and P3 are deadlocked.

1. **P1 ✗.** It needs a CD, and none is free.
2. **P2 ✗.** It needs a scanner, and none is free.
3. **P3 ✗.** It needs a CD. The only CD is held by P2, and since P2 cannot finish, it will never be freed.

In the first round no row can be satisfied and the algorithm ends immediately. All three processes remain unmarked; all three are deadlocked. Changing a single cell has completely changed the result. The algorithm does not search for cycles, but it yields the deadlocked processes directly; the waiting relation among these processes forms a cycle by itself. P3 waits for the CD held by P2; P2 waits for the scanner held by P1 or the scanners held by P3; and P1 waits for the CD held by P2.

Example 7.3

There are three resource types and E = (5 3 4). The allocations are P1 (1 0 1), P2 (2 1 0), P3 (0 1 2), P4 (1 1 0); the requests are P1 (0 1 0), P2 (1 0 1), P3 (2 0 1), P4 (0 0 2). The rows are scanned in order starting from P1, continuing from where the scan left off. Is the system deadlocked? What would happen if P2's request were (1 1 1)?

1. **Available vector.** The allocations sum to (4 3 3); A = (5 3 4) − (4 3 3) = (1 0 1).
2. **P1.** (0 1 0) ≤ (1 0 1) fails; there is no free instance of the second type.

3. **P2.** (1 0 1) ≤ (1 0 1) ✓ → A = (1 0 1) + (2 1 0) = (3 1 1).
4. **P3.** (2 0 1) ≤ (3 1 1) ✓ → A = (3 2 3).
5. **P4.** (0 0 2) ≤ (3 2 3) ✓ → A = (4 3 3).
6. **P1.** (0 1 0) ≤ (4 3 3) ✓ → A = (5 3 4) = E. Everyone is marked: there is no deadlock.
7. **The change.** If P2's request is (1 1 1), then with A = (1 0 1), P1 and P2 ask for the second type (0 free), P3 asks for 2 of the first type (1 available), and P4 asks for 2 of the third type (1 available). In the first round no row can be satisfied; all four processes are deadlocked.

7.5.9 When should detection run?

How often to run the detection algorithm is a separate design decision. There are three options:

- **At every request.** The algorithm runs after every request that is made to wait. A deadlock is found the moment it forms, together with the request that triggered it. But the CPU cost is high.
- **Periodically.** The algorithm runs every k minutes. It is cheap, but a deadlock can go unnoticed for a while, and in the meantime other processes may get caught in the cycle.
- **On a symptom.** It runs when CPU utilization drops below a threshold or when a lock wait grows long. Because deadlocked processes use no CPU, low utilization is a good symptom.

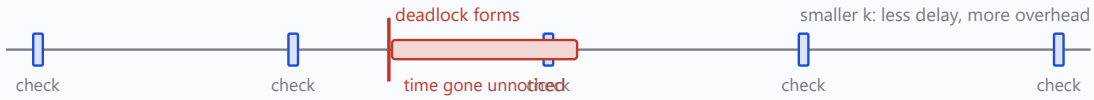


Figure 7.21: Periodic detection. If a deadlock forms between two checks, it goes unnoticed until the next check. A smaller period k reduces the delay but increases the load.

The right frequency depends on how likely deadlock is and how many processes are affected when it occurs. The overhead of detection is not negligible and is proportional to how often it runs. The cost of recovery, that is, the work that is lost, is also taken into account. PostgreSQL takes a middle path: if a transaction has waited for a lock for `deadlock_timeout` (1 second by default), it scans the wait-for graph. If it finds a cycle, it aborts one of the transactions and returns the message `ERROR: deadlock detected` to the client. In this way no scan is made for waits that end by themselves within a short time.

7.5.10 Recovering from deadlock

Once a deadlock is found, the cycle has to be broken somehow. There are three ways of breaking it, and none comes free.

Definition 7.5

Deadlock recovery is breaking the cycle by resource preemption, rollback, or process termination. The process chosen to break the cycle is called the **victim**.

- **Resource preemption.** A resource is temporarily taken from its owner and given to another process. For example, the job that a printer is printing is set aside, the printed pages are separated, and the printer is given to another job. This method depends on the nature of the resource; it is often done by hand or is not possible at all.
- **Rollback.** Processes record a **checkpoint** at intervals: they write their memory and resource state to a file. The victim is returned to the last checkpoint before it acquired the contested resource, and the resource is thereby released. The work done after that point is lost. In databases, aborting the transaction is the counterpart.
- **Termination.** The crudest method. Either all the processes on the cycle are killed, or they are killed one at a time until the cycle is broken. In the second option the detection algorithm is rerun after each kill. A process from outside the cycle may also be chosen, as long as the resources it releases break the cycle.

The goal in choosing the victim is the lowest cost. The first factors in the cost are the process's priority, how long it has been running, and how much it has left to do. The number of resources it holds and how many more it needs in order to finish also count. One also considers whether the process is interactive or a batch job and whether it can be safely restarted. A compilation can be restarted; a half-finished database update needs care. If the same process is chosen as victim every time, it can never finish, and **starvation** results. To prevent this, the number of times the process has been rolled back is added to the cost.

Prevention and detection are not recovery methods. And no recovery is free of interruption: every recovery method brings lost work or the cost of rerunning.

7.5.11 Recovery step by step

Suppose that, in an example with four processes, the detection algorithm has found the cycle P1 → R2 → P2 → R3 → P3 → R1 → P1. P4 is stuck as well, because it waits for R2. Figure 7.22 shows recovery by terminating one process at a time.

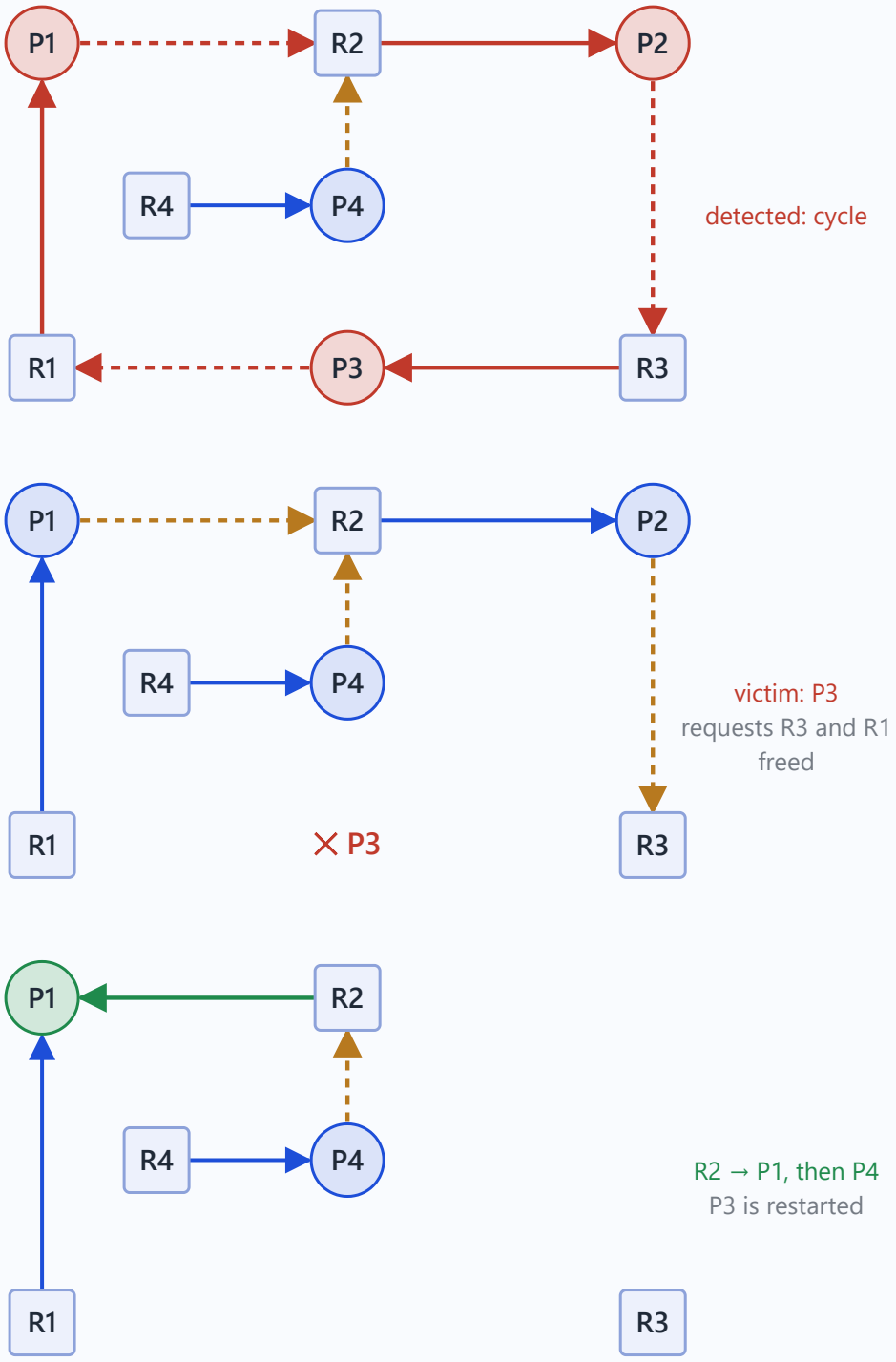


Figure 7.22: Recovery. (a) The detection algorithm finds the cycle. (b) P3, the process whose loss costs the least work, is chosen as the victim; the R3 it holds is freed. (c) R3 is given to P2; when P2 finishes, R2 goes first to P1 and, when P1 finishes, to P4. P3 is rerun from the beginning or from a checkpoint.

1. **Choosing the victim.** The process whose loss costs the least work is P3. P3 is terminated or rolled back to its checkpoint; the R3 it holds is freed.

2. **Detecting again.** After P3's edges are removed, no cycle is left in the graph.
3. **A chain of progress.** R3 is given to P2, which is waiting for it. P2 finishes and releases R2 and R3. R2 is given to the next waiter, P1; when P1 finishes, P4 gets R2 as well.
4. **Rerunning the victim.** P3 is restarted from the beginning or from its last checkpoint. The work lost is the work P3 did after that point.

When processes are terminated one at a time, the detection algorithm is rerun after each victim, and the process stops once the cycle is broken. In this example a single victim was enough.

7.6 Deadlock avoidance

The detection strategy lets deadlock happen and then pays the price. The avoidance strategy never lets deadlock happen at all. At every resource request the system asks, “if I grant this request, will deadlock become unavoidable later?” and, if the answer is yes, postpones the request. To answer this question, the maximum amount of resources each process may request must be known in advance.

7.6.1 Resource trajectories

The idea of avoidance is seen most clearly in a drawing with two processes. This representation goes back to Dijkstra. Let processes A and B use a printer and a plotter. A uses the printer between instructions I1 and I3 and the plotter between I2 and I4. B uses the plotter between I5 and I7 and the printer between I6 and I8.

The horizontal axis shows the instructions A has executed, the vertical axis those of B. Each moment of the system is a point in the plane. Since there is a single CPU, the trajectory moves only to the right (A is running) or up (B is running); it cannot go diagonally. The two shaded regions are the points where the two processes would use the same resource at the same time. Because of mutual exclusion these regions cannot be entered.

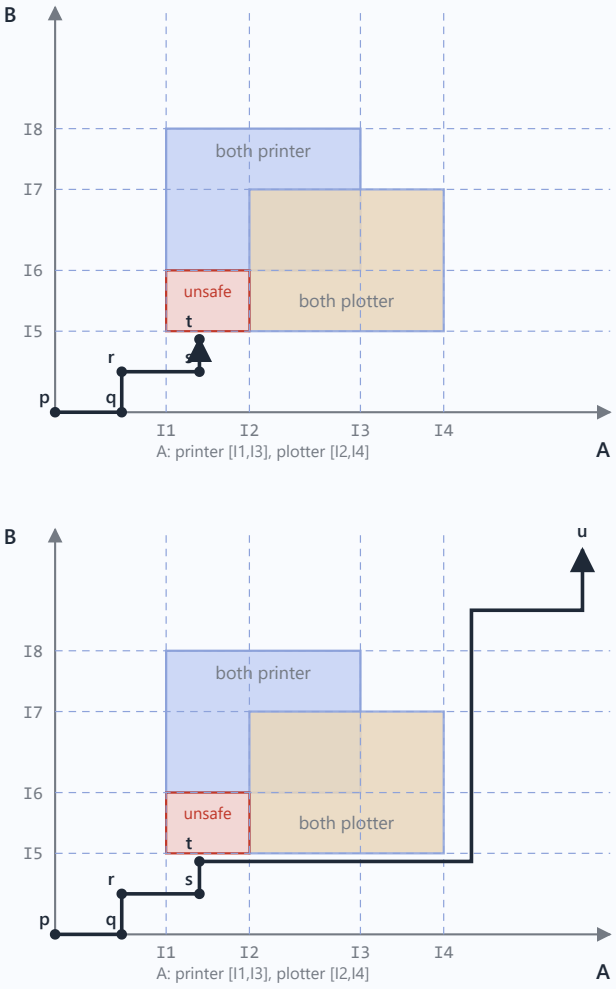


Figure 7.23: Resource trajectories of two processes (Tanenbaum Figure 6-8). (a) The trajectory p → q → r → s → t; the red box is the unsafe region. (b) At point *t* B's request for the plotter is postponed; A runs up to I4 and releases both resources, and then B continues.

Let us follow the trajectory. Between *p* and *q* A runs; between *q* and *r* B runs. Between *r* and *s* A runs again and acquires the printer at I1. Between *s* and *t* B advances and is about to request the plotter at I5. The red box is the region where A holds the printer and B holds the plotter. If the trajectory enters this box, both processes are forced to enter one of the shaded regions in order to move right or up. Since that is impossible, deadlock is inevitable. Nobody is blocked yet inside the box, but the outcome is certain. That is why this region is called **unsafe**.

Avoidance means postponing the request that would send the trajectory into the unsafe region. At point *t* the operating system makes B's plotter request wait. A runs up to I4 and releases both resources; then B continues, and there is no deadlock. What is critical here is that the operating system knows A will later ask for the plotter. Without this knowledge the request at *t* would look harmless.

7.6.2 Safe and unsafe states

The trajectory picture is limited to two processes and two resources. In the general case avoidance looks at the state of the system. The state is determined by the existing resources E, the available resources A, the allocation matrix C, and the maximum need (Max) of each process.

Definition 7.6

A state is **safe** if there is a **safe sequence** $\langle P_1, \ldots, P_n \rangle$ that lets all the processes run to completion in some order, even if every process were to make its maximum request immediately. In the sequence, the remaining need of each P_i does not exceed the sum of the available resources and the resources held by the P_j that precede it in the sequence ($j < i$). If there is no such sequence, the state is **unsafe**.

The logic of the definition is the same as that of the detection algorithm; the only difference is the row that is examined. In detection one checks whether the processes' *current requests* can be satisfied. In avoidance the worst case is assumed: every process is assumed to ask for up to its maximum, and one checks whether its *remaining need* can be satisfied. The first process in the sequence has its remaining need met from the available resources, finishes, and releases everything. The second process is then satisfied from the enlarged pool, and the chain continues in this way.

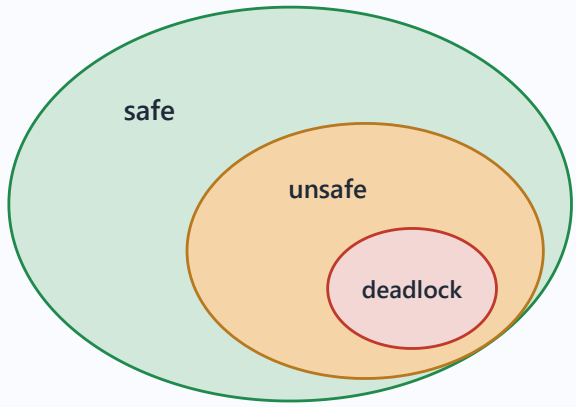


Figure 7.24: The state space. Deadlocked states are a subset of the unsafe states; safe states are entirely disjoint from deadlock.

An unsafe state is not a deadlocked state. In an unsafe state nobody is permanently blocked yet. If the processes do not make their maximum requests, for example if one releases a resource before making its additional request, everything may go well. The difference is this: in a safe state the system can guarantee that everyone finishes; in an unsafe state it has no such guarantee. The avoidance algorithm always keeps the system in a safe state and postpones every request that would lead to an unsafe one.

Common mistake

Using “unsafe” and “deadlocked” as synonyms. An unsafe state says only that the guarantee of completion is lost. The banker's algorithm rejects an unsafe state not because it is a deadlock but because a deadlock *may* occur. Even if the system remains in an unsafe state, the processes may finish without making their maximum requests.

7.6.3 A safe-state example

Let there be 10 instances of a single resource type. In the tables, “Has” is the number of instances the process holds right now, and “Max” is the greatest number of instances it will need until it finishes. The remaining need is Max − Has. In avoidance the operating system is in a position to choose the order of requests itself: if a safe sequence exists, it can finish the processes in the order of that sequence.

Example 7.4

A holds 3 instances (Max 9), B holds 2 (Max 4), C holds 2 (Max 7). Of the 10 instances in total, 7 have been handed out and 3 are free. Is the state safe? (Tanenbaum Figure 6-9)

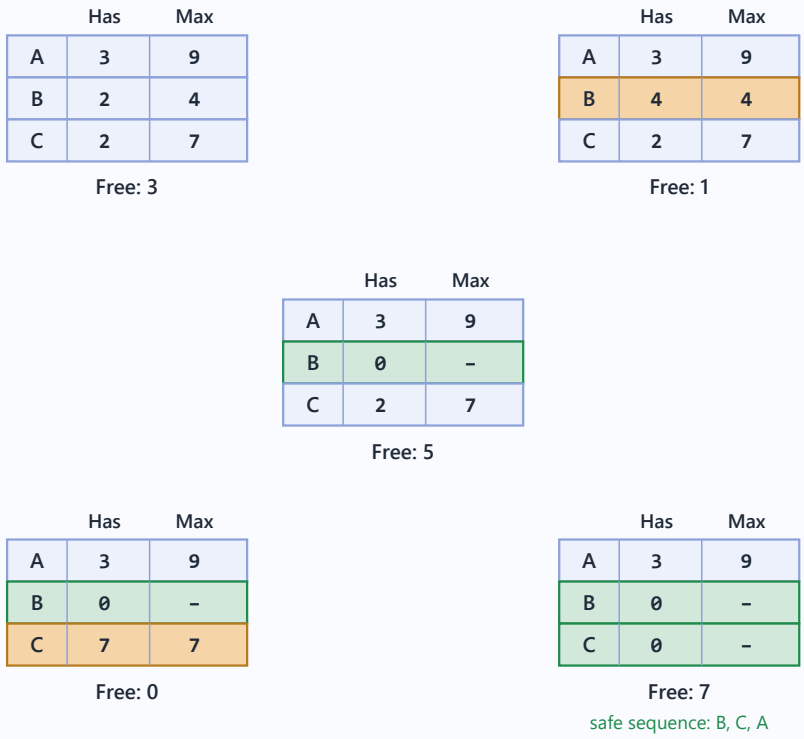


Figure 7.25: A safe state. (a) Start, free 3. (b) B is given 2, free 1. (c) B finishes, free 5. (d) C is given 5, free 0. (e) C finishes, free 7; A's remaining 6 can be met.

1. **Remaining needs.** A: 9 − 3 = 6; B: 4 − 2 = 2; C: 7 − 2 = 5.
2. **B.** Its remaining need is 2 ≤ 3. B is given 2 instances; 1 is left free. B reaches its maximum, finishes, and releases 4 instances: free 5.
3. **C.** Its remaining need is 5 ≤ 5. C is given 5; 0 is left free. C finishes and releases 7 instances: free 7.
4. **A.** Its remaining need is 6 ≤ 7. A finishes too.
5. **Result.** ⟨B, C, A⟩ is a safe sequence; the state is safe. Trying all orders shows that there is no other safe sequence: in the first step only B can be satisfied, and after B only C.

7.6.4 An unsafe-state example

Start from the same initial state; the difference is a single decision. A asks for one more instance, and the operating system grants it.

Example 7.5

A holds 4 instances (Max 9), B holds 2 (Max 4), C holds 2 (Max 7); free 2. Is the state safe? (Tanenbaum Figure 6-10)

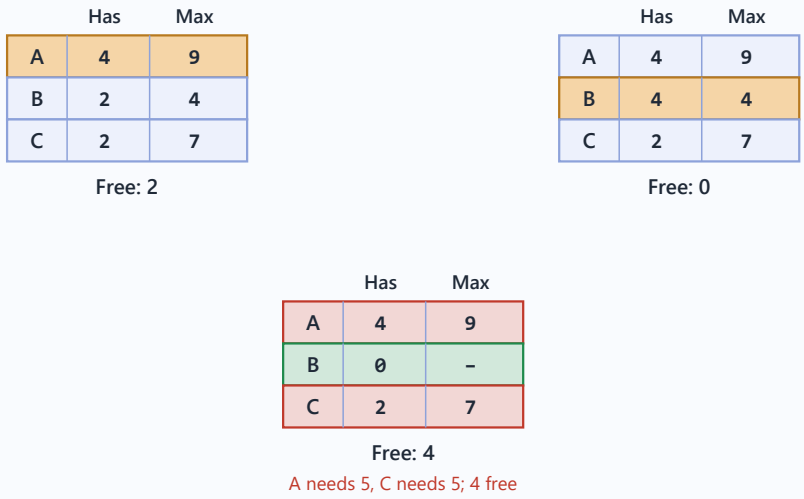


Figure 7.26: An unsafe state. (a) A has 4/9, free 2. (b) Only B can be satisfied; B is given 2. (c) B finishes, free 4; A needs 5 and C needs 5, and neither can be guaranteed.

1. **Remaining needs.** A: 5; B: 2; C: 5.
2. **B.** 2 ≤ 2; B is given 2, finishes, and releases 4 instances: free 4.
3. **A and C.** Both have a remaining need of 5, and 5 > 4. Neither can be guaranteed.

4. **Result.** There is no safe sequence; the moment A was given the extra instance, the state became unsafe. A's request should have been postponed.

There is no deadlock here yet. If A releases an instance before making its additional request, there may be no trouble. But the operating system assumes the worst case: if everyone makes their maximum request, completion is not guaranteed. Avoidance does not grant the request that loses this guarantee.

7.6.5 The banker's algorithm: one resource type

In 1965 Dijkstra described the avoidance algorithm with a banking analogy. A small-town banker extends credit limits to customers. The sum of the limits is more than the cash in the vault, because not all customers ask for their entire limit at the same time. At each request the banker asks, “if I grant this loan, will I in the end be able to meet the limits of all my customers?”

Definition 7.7

The **banker's algorithm** is an avoidance algorithm that, at every resource request, computes the new state as if the request had been granted, and grants the request if that state is safe and makes the process wait if it is unsafe. The request is re-evaluated later, when resources are released.

The safety check goes like this: find a customer whose remaining need can be met from the money in the vault; this customer is treated as finished and the credit is added back to the vault. The step is repeated until all customers have finished or no one can be satisfied.

Example 7.6

The vault holds 10 units. Four customers have credit limits A 6, B 5, C 4, D 7; the total is 22. After a while A uses 1, B 1, C 2, and D 4 units. Is the state safe? In this state, should a request by B for one more unit be granted? (Tanenbaum Figure 6-11)

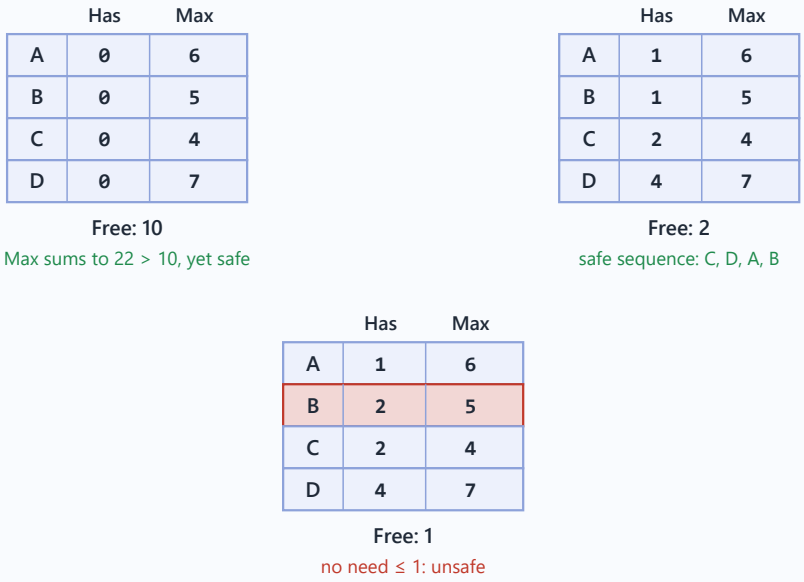


Figure 7.27: The banker's algorithm, one resource type. (a) Start: nobody is using credit, 10 units in the vault. (b) 8 in use, 2 in the vault: safe. (c) If B takes one more unit, 1 is left in the vault: unsafe.

1. **Start.** Although the sum of the limits is 22 > 10, as long as nobody is using credit the state is safe; the customers can be finished one after another.
2. **State (b).** In use 1 + 1 + 2 + 4 = 8; there are 2 units in the vault. The remaining needs are A 5, B 4, C 2, D 3.
3. **Safe sequence.** C's need 2 ≤ 2: C finishes and 4 units come back, 4 in the vault. D's need 3 ≤ 4: D finishes, 8 in the vault. When A (5 ≤ 8) finishes, 9; when B (4 ≤ 9) finishes, 10. ⟨C, D, A, B⟩ is a safe sequence.
4. **B's request.** If B takes one more unit, 1 is left in the vault. The remaining needs are A 5, B 3, C 2, D 3; even the smallest is greater than 1. The state is unsafe; the request is refused and B is made to wait.

A state is safe when all the processes, not just one, can finish in some order. The banker's algorithm does not resolve deadlock after the fact; as long as all resource requests go through the algorithm and every process has correctly declared its maximum need in advance, it never lets the system get into deadlock. The price is also clear: in state (c) B is made to wait even though there is actually money in the vault.

Example 7.7

Processes request identical resources of the same type (tape drives) one at a time and release them all when finished. (a) There are 4 processes, and each uses at most 3 drives at once. What is the smallest number of drives that guarantees deadlock never occurs? (b) With 6

processes, at most 4 drives per process, and 18 drives in the system, can deadlock occur? (c) With 18 drives and at most 4 drives per process, what is the largest number of processes that can run safely?

1. **The worst case.** Let there be n processes, each requesting at most k drives. In the worst case each process holds $k - 1$ drives and waits for one more. If there is one more drive, one of the processes completes its need, finishes, and releases its drives. The minimum number needed is $n \cdot (k - 1) + 1$.
2. **(a)** $4 \cdot 2 + 1 = 9$ drives. With 8 drives, each of the four processes can hold two drives and wait for a third.
3. **(b)** $6 \cdot 3 + 1 = 19 > 18$: deadlock can occur. If all 18 drives are distributed three apiece among the six processes, everyone waits for a fourth.
4. **(c)** We need $3n + 1 \le 18$; $n \le 5.67$, so at most 5 processes.

This bound is the number of resources at which hold and wait cannot turn into circular wait, and it corresponds to the single-resource worst case of the banker's algorithm. The values were verified by brute force in Python, trying all distributions.

7.6.6 The banker's algorithm: several resource types

The algorithm generalizes directly to several resource types. For n processes and m resource types, using Silberschatz's naming, four data structures are kept:

- **Available[*m*]**: the number of free instances of each type.
- **Max[*n*][*m*]**: the maximum request each process has declared in advance.
- **Allocation[*n*][*m*]**: the instances each process currently holds.
- **Need[*n*][*m*]** = Max − Allocation: the remaining need.

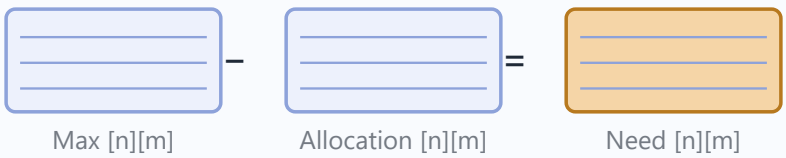


Figure 7.28: The remaining-need matrix is found by subtracting the allocation matrix from the maximum-request matrix: Need = Max − Allocation.

The algorithm has two parts. The **safety algorithm** tells whether a state is safe:

```
/* safety algorithm */
Work ← Available
Finish[i] ← false  (all i)
while some i has Finish[i] = false
      and Need[i] ≤ Work:
  Work ← Work + Allocation[i]
  Finish[i] ← true
SAFE if Finish[i] for all i
```

`Work` is a temporary vector that represents the free pool during the test; `Available` itself is not modified. `Finish[i]` records whether process i is considered finished. As long as an unfinished process whose remaining need can be met from `Work` can be found, that process is considered finished and what it holds is added to `Work`. If all `Finish` values are true at the end, the state is safe.

The **request algorithm** runs when a process asks for `Request[i]`:

```
/* Ti requests Request[i] */
if Request[i] > Need[i]:
  error  /* exceeds Max */
if Request[i] > Available:
  Ti waits  /* not available */
/* pretend to allocate: */
Available  -= Request[i]
Allocation[i] += Request[i]
Need[i]    -= Request[i]
if safe: grant the request
else: restore, Ti waits
```

The first check is an error check: a process cannot ask for more than the maximum it declared. The second check looks at whether the resources are actually free right now; if not, the process has to wait anyway. If both checks pass, the algorithm makes the request *as if granted*: it updates the three structures and runs the safety algorithm on the new state. If the state is safe, the changes become permanent. If it is unsafe, the three changes are undone and the process is made to wait.

The cost of the safety algorithm is the same as that of the detection algorithm: $O(m \cdot n^2)$. The difference is that this test runs at every request.

Common mistake

Adding the *Need* row of the finished process to `Work` in the safety algorithm. A finished process releases what it holds, that is, its *Allocation* row. *Need* is only the amount required for the process to be able to finish, and it is used in the comparison; what is added to the pool is *Allocation*.

7.6.7 The safety algorithm: an example

Silberschatz's classic example has five threads (T0–T4) and three resource types: 10 instances of type A, 5 of type B, and 7 of type C. The allocations and maximum requests are as follows:

Thread	Allocation (A B C)	Max (A B C)	Need (A B C)
T0	0 1 0	7 5 3	7 4 3
T1	2 0 0	3 2 2	1 2 2
T2	3 0 2	9 0 2	6 0 0
T3	2 1 1	2 2 2	0 1 1
T4	0 0 2	4 3 3	4 3 1

Table 7.8: The initial state of the Silberschatz example.

The column sums of the allocations are (7 2 5); Available = (10 5 7) − (7 2 5) = (3 3 2). The Need column is computed as Max − Allocation. The safety algorithm starts with `Work` = (3 3 2) and scans the rows cyclically, beginning at T0 and continuing from where it left off.

```
Work ← Available
Finish[i] ← false
for i = T0 … T4 (repeatedly):
  if Finish[i] = false and
     Need[i] ≤ Work:
    Work ← Work + Allocation[i]
    Finish[i] ← true
if all Finish[i]: SAFE
```

Step	Row	Need ≤ Work?	Result	Work afterwards
1	T0	(7 4 3) ≤ (3 3 2)	✗ skip	(3 3 2)
2	T1	(1 2 2) ≤ (3 3 2)	✓ + (2 0 0)	(5 3 2)
3	T2	(6 0 0) ≤ (5 3 2)	✗ not enough A	(5 3 2)
4	T3	(0 1 1) ≤ (5 3 2)	✓ + (2 1 1)	(7 4 3)
5	T4	(4 3 1) ≤ (7 4 3)	✓ + (0 0 2)	(7 4 5)
6	T0 (second pass)	(7 4 3) ≤ (7 4 5)	✓ + (0 1 0)	(7 5 5)
7	T2	(6 0 0) ≤ (7 5 5)	✓ + (3 0 2)	(10 5 7)

Table 7.9: Trace of the safety algorithm.

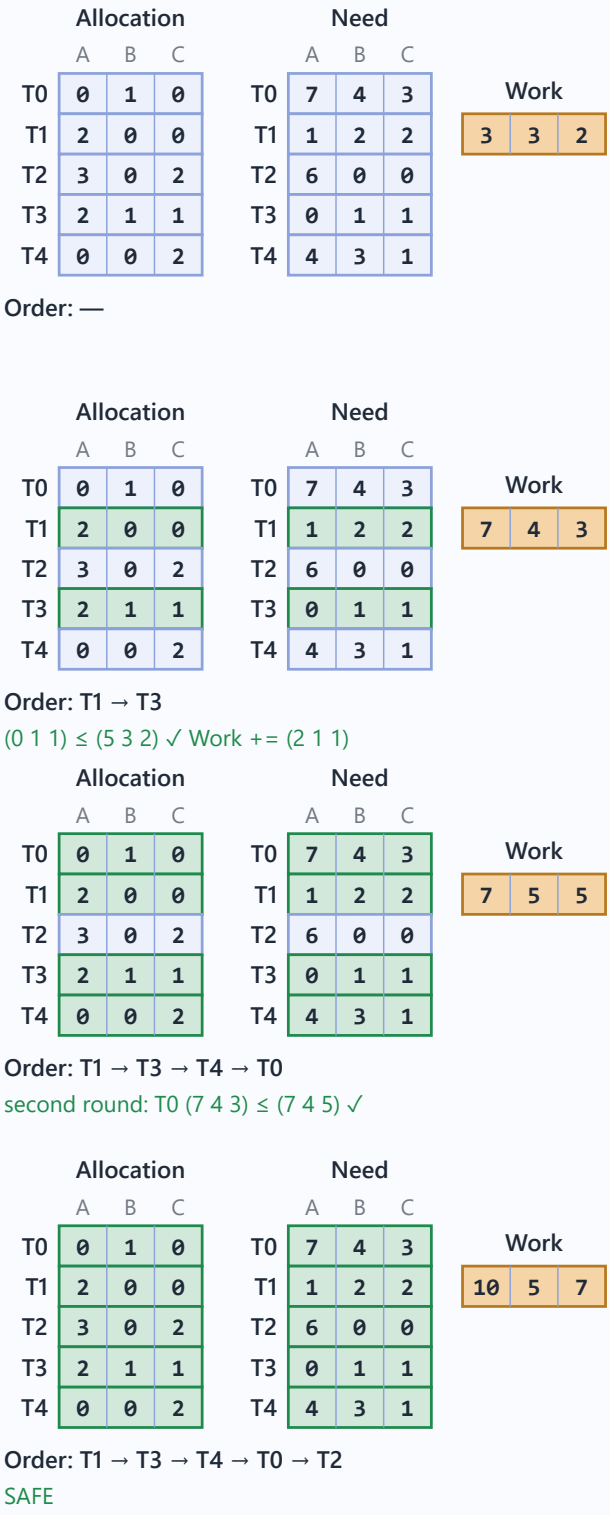


Figure 7.29: The safety algorithm. (a) Start: Work = (3 3 2). (b) After T1 and T3 finish, Work = (7 4 3). (c) In the second pass T0 finishes, Work = (7 5 5). (d) All Finish values are true: the state is safe.

All `Finish` values are true, and at the end `Work` = (10 5 7), that is, all the resources. The state is safe; the safe sequence found is ⟨T1, T3, T4, T0, T2⟩. The safe sequence is not unique. Trying all orders finds 16 safe sequences for this state; ⟨T1, T3, T4, T2, T0⟩ is one of them. What they all have in

common is that the first process is T1 or T3, because with the initial (3 3 2) only these two can be satisfied.

7.6.8 Granting a request

In the same state, let T1 ask for Request_1 = (1 0 2). The request algorithm decides in three steps.

Example 7.8

T1 requests (1 0 2). Should the request be granted?

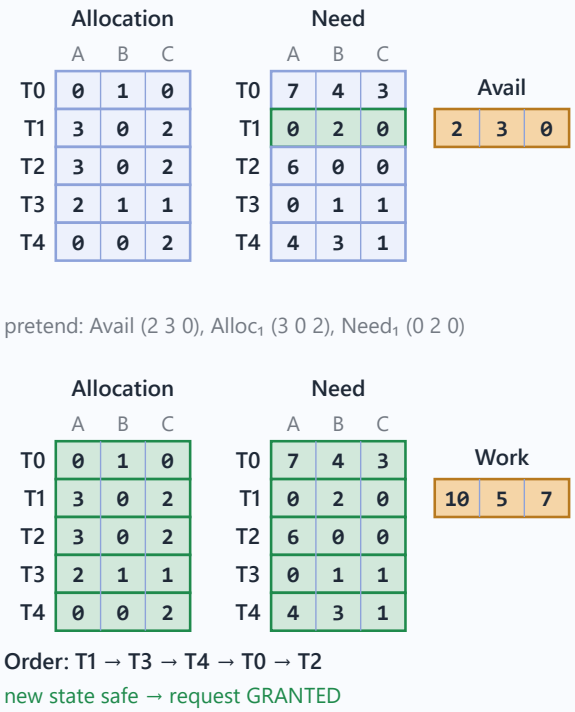


Figure 7.30: T1's request. (a) The pretend state: Available (2 3 0), Allocation_1 (3 0 2), Need_1 (0 2 0). (b) The safety algorithm finds the sequence ⟨T1, T3, T4, T0, T2⟩: the request is granted.

1. **Need check.** (1 0 2) ≤ Need_1 = (1 2 2) ✓. T1 does not exceed its declared maximum.
2. **Available check.** (1 0 2) ≤ Available = (3 3 2) ✓. The resources are free right now. But this alone is not enough.
3. **The pretend state.** Available = (3 3 2) − (1 0 2) = (2 3 0); Allocation_1 = (2 0 0) + (1 0 2) = (3 0 2); Need_1 = (1 2 2) − (1 0 2) = (0 2 0).
4. **Safety test.** Work = (2 3 0). T0 (7 4 3) ✗. T1 (0 2 0) ✓ → Work = (5 3 2). T2 (6 0 0) ✗. T3 (0 1 1) ✓ → (7 4 3). T4 (4 3 1) ✓ → (7 4 5). T0 (7 4 3) ✓ → (7 5 5). T2 (6 0 0) ✓ → (10 5 7).
5. **Decision.** The safe sequence ⟨T1, T3, T4, T0, T2⟩ exists; the request is granted and the new state becomes permanent.

The safety test is made on the "pretend" state. Had the state turned out unsafe, all three changes would have been undone and T1 would have been made to wait.

7.6.9 Refusing a request

After T1's request has been granted, suppose two new requests arrive. They show two different reasons a process may be made to wait.

Example 7.9

The state is the one after T1's request was granted: Available = (2 3 0). (a) What happens if T4 requests (3 3 0)? (b) What happens if T0 requests (0 2 0)?

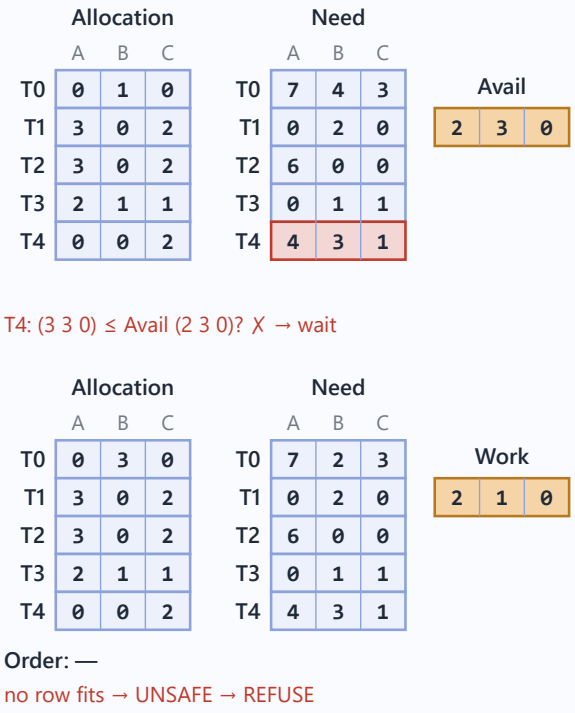


Figure 7.31: Two refusals. (a) For T4's request only 2 instances of type A are free: T4 waits. (b) When T0's request is pretend-granted, Available becomes (2 1 0) and no row can be satisfied: the state is unsafe and the request is refused.

1. **(a) T4.** (3 3 0) ≤ Need_4 = (4 3 1) ✓. But (3 3 0) ≤ (2 3 0) fails: only 2 instances of type A are free. The resource is not available, so T4 waits. The safety test is never reached.
2. **(b) T0, two checks.** (0 2 0) ≤ Need_0 = (7 4 3) ✓ and (0 2 0) ≤ (2 3 0) ✓. The resource is actually free.
3. **The pretend state.** Available = (2 1 0); Allocation_0 = (0 3 0); Need_0 = (7 2 3).
4. **Safety test.** Work = (2 1 0). T0 (7 2 3): not enough A. T1 (0 2 0): not enough B. T2 (6 0 0): not enough A. T3 (0 1 1): not enough C. T4 (4 3 1): not enough A. No row can be satisfied.

5. **Decision.** The state is unsafe. The changes are undone and T0 is made to wait, even though the resource is free.

In the first case the resource is not available; in the second it is, but granting would make the state unsafe. The price of avoidance is the second case: requests are sometimes postponed while resources sit idle, and resource utilization drops.

7.6.10 Code trace: the banker's request algorithm

To trace the request and safety algorithms together, return to the initial state: Available = (3 3 2). This time T3 requests Request = (0 1 1). The code gives the two algorithms together:

```
/* Ti requests Request */
if Request > Need[i]: error /* exceeds Max */
if Request > Available: Ti waits
Available -= Request  /* pretend */
Allocation[i] += Request
Need[i] -= Request
if safe(): GRANT the request
else: restore, Ti waits
safe():
  Work ← Available; Finish ← false
  for i = T0 … T4 (repeatedly):
    if !Finish[i] and Need[i] ≤ Work:
      Work ← Work + Allocation[i]
      Finish[i] ← true
  return all Finish[i]
```

Step	Line	Evaluation	Values changed
1	`Request > Need[3]` ?	(0 1 1) ≤ (0 1 1): maximum not exceeded	—
2	`Request > Available` ?	(0 1 1) ≤ (3 3 2): resources available	—
3	pretend	the three structures are updated	Available (3 2 1), Allocation[3] (2 2 2), Need[3] (0 0 0)
4	`safe()`	Work ← (3 2 1); T0 (7 4 3), T1 (1 2 2), T2 (6 0 0) cannot finish	Work (3 2 1)
5	T3	(0 0 0) ≤ (3 2 1) ✓	Work (5 4 3), Finish[3]
6	T4	(4 3 1) ≤ (5 4 3) ✓	Work (5 4 5), Finish[4]
7	T0, T1 (second pass)	not enough A for T0; T1: (1 2 2) ≤ (5 4 5) ✓	Work (7 4 5), Finish[1]
8	T2	(6 0 0) ≤ (7 4 5) ✓	Work (10 4 7), Finish[2]
9	T0 (third pass)	(7 4 3) ≤ (10 4 7) ✓	Work (10 5 7), Finish[0]

Step	Line	Evaluation	Values changed
10	result	all Finish true: safe	sequence ⟨T3, T4, T1, T2, T0⟩; the request is granted

Table 7.10: Trace of T3's request for (0 1 1).

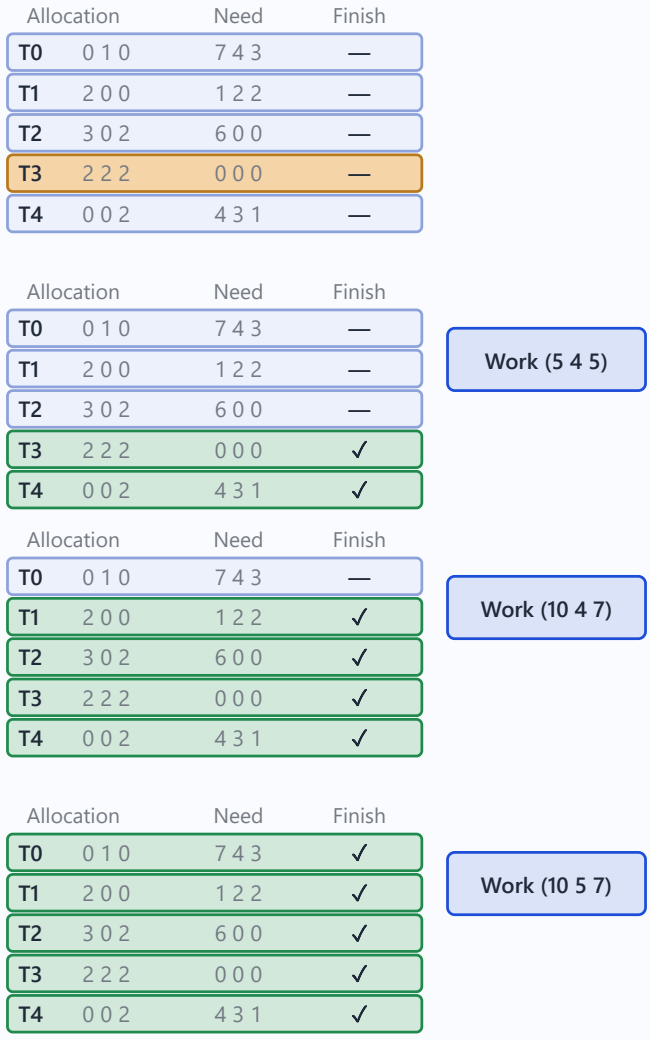


Figure 7.32: Trace of T3's request. (a) The pretend state: Allocation[3] = (2 2 2), Need[3] = (0 0 0). (b) T3 and T4 have finished, Work = (5 4 5). (c) T1 and T2 have finished too, Work = (10 4 7). (d) T0 finishes as well: all Finish values are true.

The notable point in the trace is that T1 cannot finish in the first pass. In the initial state T1 could finish immediately. But because T3 was given one B and one C, `Work` = (3 2 1), and T1's need for C is 2 > 1. Since T3's remaining need drops to (0 0 0), T3 finishes at once and returns the (2 2 2) it holds; after that the chain builds up. All the values were recomputed in Python with the request and safety algorithms themselves.

7.6.11 A second example

Tanenbaum's multi-resource example has five processes (A–E) and four resource types: tape drive (Tp), plotter (Pl), printer (Pr), and CD. The existing resources are E = (6 3 4 2), and the total of assigned resources is P = (5 3 2 2). The available resources are $A = E - P$ = (1 0 2 0).

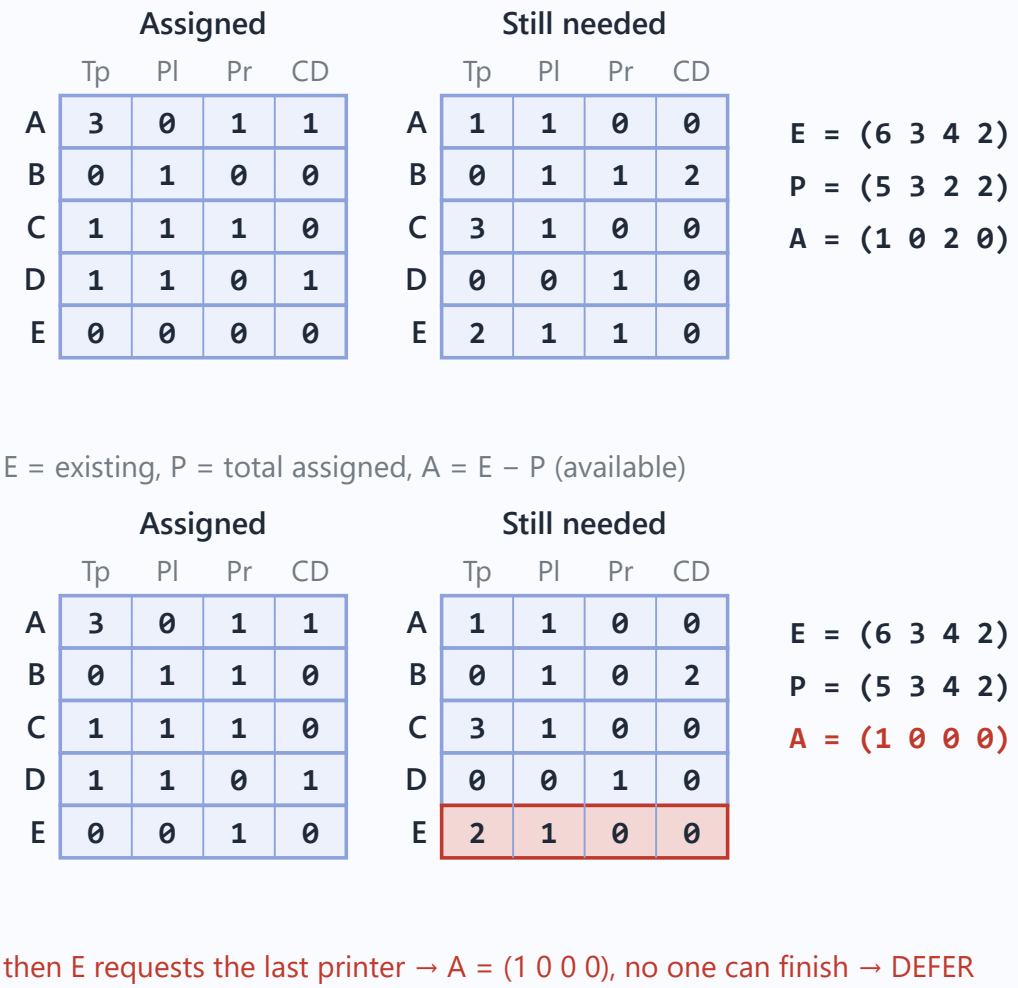


Figure 7.33: The banker's algorithm with several resources (Tanenbaum Figure 6-12). Top: the initial state, with the assigned resources, the resources still needed, and the vectors E, P, A. Bottom: the state after B and E are each given a printer: with A = (1 0 0 0) no row can be satisfied.

Example 7.10

The resources still needed are A (1 1 0 0), B (0 1 1 2), C (3 1 0 0), D (0 0 1 0), E (2 1 1 0). Is the state safe? If B requests a printer, should it be granted? What happens if E then requests the last printer?

1. **Safety.** D's need (0 0 1 0) ≤ (1 0 2 0). D finishes and (1 1 0 1) comes back: A = (2 1 2 1). Then E (2 1 1 0) ≤ (2 1 2 1) finishes; since E holds nothing, A does not change. A finishes, A = (5 1 3 2); B finishes, (5 2 3 2); C finishes, (6 3 4 2). ⟨D, E, A, B, C⟩ is a safe sequence.

2. **B's printer request.** Pretend-granted, A = (1 0 1 0) and P = (5 3 3 2); B's need drops to (0 1 0 2). D can still finish ((0 0 1 0) ≤ (1 0 1 0)) and the same sequence is kept. The request is granted.
3. **E's printer request.** Pretend-granted, A = (1 0 0 0) and P = (5 3 4 2). The plotter needs of A and B, the tape and plotter needs of C, the printer need of D, and the plotter need of E cannot be met. No row is ≤ A: the state is unsafe and the request is postponed.

The steps "look for a row that is ≤ A" are the safety test here. The same logic is used with the request matrix in the detection algorithm and with the remaining-need matrix in avoidance. The code of the two algorithms is almost identical; only the meaning of the row being compared changes.

7.6.12 The banker's algorithm in practice

The banker's algorithm is an elegant idea, but it rests on four assumptions:

- Every process declares its maximum need in advance.
- The number of processes is fixed.
- The number of resources does not change; for example, a device does not fail.
- Processes release resources in a finite time.

These assumptions do not hold in general-purpose operating systems. How much memory or how many file locks a word processor will request is not known in advance; processes are born and die all the time; a disk can fail. That is why the algorithm is almost never used in real operating systems. A similar "admission control" idea, however, is common. In real-time scheduling, one checks whether a new task would break the timing guarantees of the other tasks. In cloud systems quotas are set, and connection pools have upper limits.

Avoidance is usually more expensive than detection and recovery because of the test at every request, which costs $O(m \cdot n^2)$, and because of needless postponements. Its advantage is that it needs no recovery. The algorithm does not run continuously in the background: the test is made only when a request arrives and, for waiting requests, when a resource is released.

7.7 Deadlock prevention

Avoidance requires knowing the future requests of the processes, and in practice that information is not available. Prevention takes a more radical route: since the four conditions are jointly necessary, removing one of them from the system altogether makes deadlock impossible. The decision is made at design time, not at run time.

7.7.1 Attacking the conditions

Table 7.11 summarizes the method used against each condition and its price.

Condition	Attack	Price
Mutual exclusion	Make the resource shareable: spooling, read-only sharing, lock-free structures	Most resources are inherently unshareable (the mutex itself!)
Hold and wait	Request all resources at once up front, or release what you hold before a new request	Low utilization, starvation; needs must be known in advance
No preemption	Take back the resources of a process that is about to wait; back off with tryLock; virtualize	Only for resources whose state can be saved; risk of livelock
Circular wait	Give the resources a total order and request only in increasing order (lock ordering)	Following the order takes programmer discipline

Table 7.11: Ways of attacking the four conditions.

Two misunderstandings are common. Removing mutual exclusion does not mean "only one process may request the resource at a time"; that is mutual exclusion itself. Removing the "no preemption" condition does not mean "let the process keep its resources until it finishes"; on the contrary, it means that resources can be taken back from their owner.

7.7.2 Attacking mutual exclusion

If a resource can be given to several processes at the same time, nobody waits for it and it cannot take part in a deadlock. The printer is the classic example.

Definition 7.8

Spooling is queuing the jobs destined for a slow device on disk and sending them to the device in turn. Processes write their output to a spool directory; only a single background process (a *daemon*) uses the device.

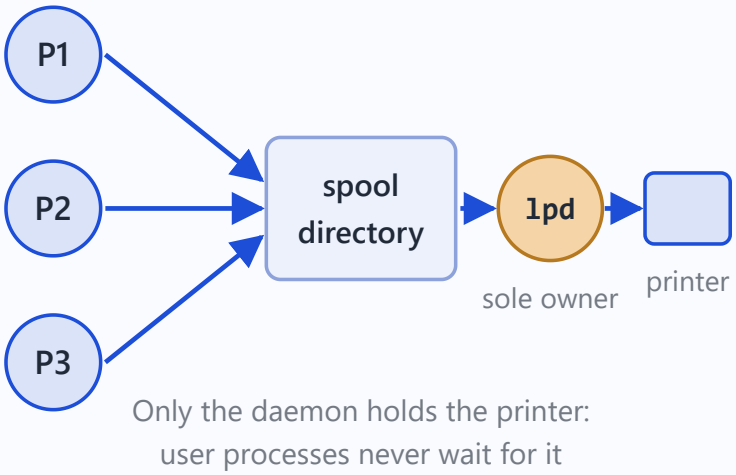


Figure 7.34: Printer spooling. P1, P2, and P3 write their output to the spool directory; only the `lpd` daemon holds the printer. User processes never wait for the printer.

From the processes' point of view the printer is now shareable: everyone can leave a job at the same time, and a single process uses the real device. Contention for the printer disappears. There are other examples. Read-only files and the code pages of programs are shared at the same time. Reader locks do not exclude each other; only a writer lock excludes. *Lock-free* data structures advance with the compare-and-swap (CAS) instruction; since there is no lock, there is no deadlock, but starvation is still possible.

The method has a limit: the spool disk is also a resource. In Tanenbaum's example many processes have written half of their output to the disk and the disk has filled up. Since no output is complete, the daemon does not start printing, and the processes wait for space: deadlock again. The general principle is this: a resource is assigned only when it is truly needed, and the number of processes that may request it is kept to a minimum.

7.7.3 Attacking hold and wait

If no process waits for another resource while holding one, no ring can be built. There are two protocols for achieving this:

1. **Protocol 1.** The process obtains all the resources it will need in a single request before it starts running: all or nothing.
2. **Protocol 2.** A process may make a request only when it holds no resources. Before asking for a new resource it releases everything it holds.

In Silberschatz's example a process copies data from a DVD to disk, sorts the file on disk, and prints the result on the printer. In Protocol 1 the process acquires the DVD drive, the disk file, and the printer at the start. Although the printer is needed only at the very end, it is held from the beginning

and nobody else can use it. In Protocol 2 the process first takes the DVD and the disk, copies, and releases both; then it takes the disk and the printer. Utilization improves, but other problems arise: by the time the process asks for the disk the second time, someone else may have modified the file.

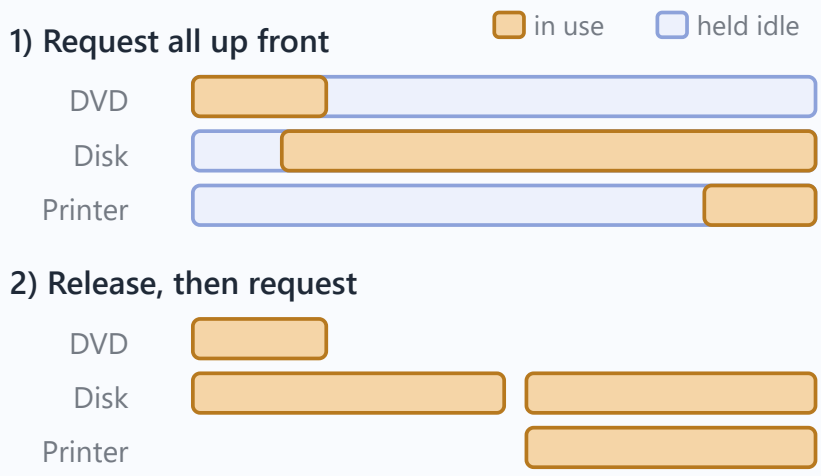


Figure 7.35: Timelines of the two protocols. In Protocol 1 the printer is held from the very start and most of that time is wasted. In Protocol 2 the resources are held only while they are used (Silberschatz §8.5.2).

The two protocols share some problems. Resource utilization drops, because resources are held even when they are not needed. A process that wants many popular resources may never get started, because they are never all free at the same time; this too is starvation. In addition, the needs are often not known in advance. In practice this idea lives on in two places: “all or nothing” acquisition with `pthread_mutex_trylock`, and conservative two-phase locking in databases that takes all locks up front.

7.7.4 Attacking no preemption

If resources can be taken back by force, the ring of waiting can be broken. Silberschatz gives the following preemption protocol: if a process holding resources makes a request that cannot be satisfied at once, all the resources it holds are implicitly released. The process is restarted when it can obtain its old resources and the new one it asked for together.

This protocol suits only resources whose state can be saved and restored: CPU registers, memory, database transactions. A print job interrupted halfway or a data structure protected by a mutex cannot be taken back this way. Some resources are made preemptable by virtualizing them: the printer is spooled, memory is paged.

For a mutex, a workable approach is the **timed attempt**. A thread that cannot get the second lock within a certain time releases the lock it holds, of its own accord, and tries again after a random time. The resource is not taken by force, but its owner gives it up voluntarily; the result is the same, and the ring is broken.

7.7.5 Code trace: backing off with tryLock

In Java, `ReentrantLock.tryLock(time, unit)` waits for the lock for at most the given time; it returns `true` if it gets the lock and `false` if it does not. The method below transfers money from one account to another. Each account has its own lock.

```
boolean tryXfer(Account a, Account b, int m)
        throws InterruptedException {
  if (!a.lock.tryLock(50, MILLISECONDS))
    return false;
  try {
    if (!b.lock.tryLock(50, MILLISECONDS))
      return false;  /* a released too */
    try { a.balance -= m; b.balance += m;
          return true; }
    finally { b.lock.unlock(); }
  } finally { a.lock.unlock(); }
}
/* caller: while (!tryXfer(x, y, 1))
              Thread.sleep(random); */
```

The method first waits at most 50 ms for the lock of the source account (`a`). If it cannot get it, it returns `false` at once. If it gets it, it also waits 50 ms for the lock of the destination account (`b`). If the second lock cannot be obtained, `return false` executes; but the outer `finally` block releases `a` 's lock before the return. So it is the `finally` that makes the comment “a released too” come true. If both locks are obtained, the transfer is made and the locks are released in reverse order. The caller sleeps for random times and retries until the method succeeds.

Let two threads transfer in opposite directions: T1 moves 1 unit from `x` to `y` , and T2 moves 1 unit from `y` to `x` . Both accounts start with 1000 units. Table 7.12 traces one unlucky timing.

t (ms)	T1	T2	x lock	y lock	x / y balance
0	ready	ready	free	free	1000 / 1000
0	gets x at once	ready	T1	free	1000 / 1000
1	holds x	gets y	T1	T2	1000 / 1000
2	tries y (≤ 50 ms)	tries x (≤ 50 ms)	T1	T2	1000 / 1000
52	timed out: `false`	tries x	T1	T2	1000 / 1000
52	`finally` : releases x	tries x	free	T2	1000 / 1000
52	sleeps randomly	gets x	T2	T2	1000 / 1000
52	sleeping	transfers	T2	T2	1001 / 999
53	sleeping	releases x, then y; `true`	free	free	1001 / 999
70	retries, gets both locks, and transfers	done	T1	T1	1000 / 1000

Table 7.12: Trace of backing off with tryLock.

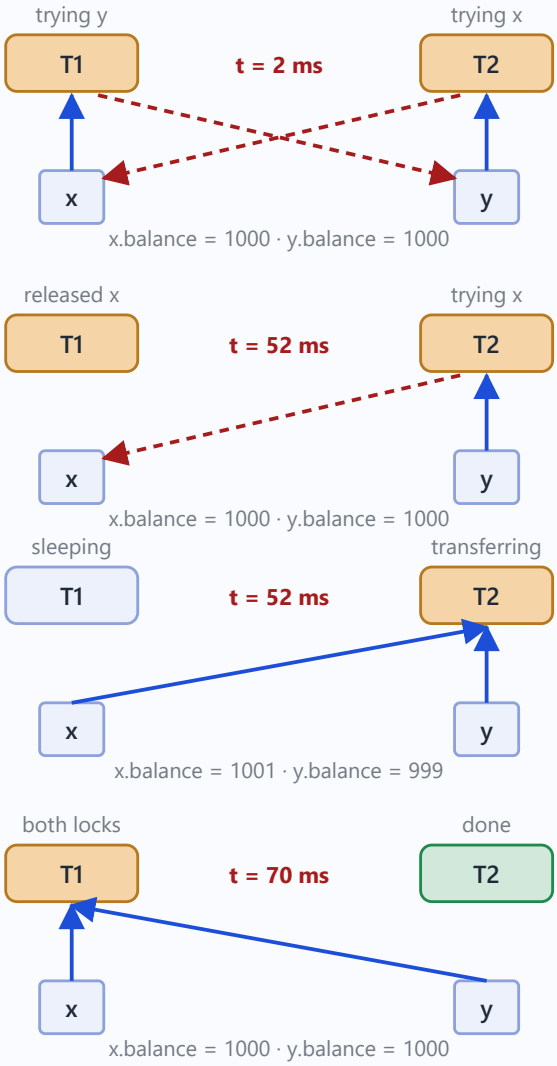


Figure 7.36: Backing off with tryLock. (a) Each waits at most 50 ms for the second lock: circular wait exists, but it is timed. (b) T1's time runs out and `finally` releases x. (c) T2 gets x and transfers. (d) After sleeping a random time, T1 retries; the total is again 2000.

At step 2 a circular wait really does form, but it is not unbounded. At the 52nd millisecond T1 gives up and releases `x`; the no-preemption condition has been broken voluntarily. With a plain `lock()`, the deadlock would have become certain at t = 2 ms. The program was run with JDK 21 using 2 threads and 1000 opposite-direction transfers each: the result is `x=1000 y=1000 total=2000`, and the program finished every time.

The random wait is important. If two threads always wait the same time and retry at the same moment, the same collision can happen again and again. The processes are not blocked, but none makes progress; this situation is called livelock.

7.7.6 Attacking circular wait

The prevention method used most in practice is to break circular wait. Each resource type is given a number, and processes are allowed to request resources only in increasing numerical order.

#	Resource
1	Imagesetter
2	Printer
3	Plotter
4	Tape drive
5	Blu-ray drive

Table 7.13: An example of resource numbering (Tanenbaum).

A process that needs the printer and the tape drive asks first for 2 and then for 4. After it holds the tape drive it cannot ask for the printer.

Theorem 7.3 (Resource ordering)

If every process requests resources only with numbers greater than those it already holds, circular wait cannot occur.

Proof. First consider the case of two processes. If A holds resource number i and requests j, the rule says $i < j$. If B holds j and requests i, we would need $j < i$. Both cannot be true; this edge never arises. In the general case, suppose there is a cycle, and call the resource with the greatest number on the cycle R_k. The process holding R_k waits for the next resource on the cycle. That resource's number is less than k, because k is the greatest. But by the rule a process may not request a resource numbered lower than what it holds. A contradiction; there is no cycle.

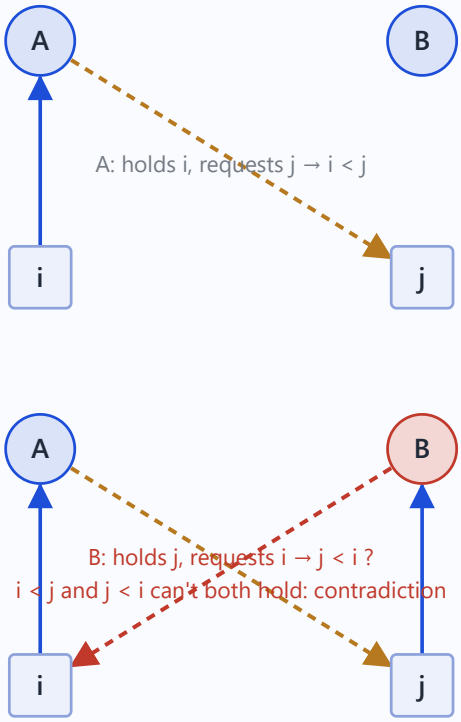


Figure 7.37: Numbering breaks the cycle (Tanenbaum Figure 6-13). (a) If A holds i and requests j, then $i < j$. (b) B holding j and requesting i would require $j < i$: a contradiction, so this edge cannot arise.

There is also a variant of the rule: if a process is going to request R_j, it first releases all the R_i ($i \geq j$) that it holds. The hardest part of this method is finding a suitable ordering that all programs will follow. If a process in the natural flow of its work uses the tape drive before the printer, then to obey the rule it has to acquire the printer earlier than it needs it.

Example 7.11

The resources are numbered as in Table 7.13. Processes make the following sequences of requests without releasing anything. Which follows the rule? (a) Tape → Printer → Blu-ray, (b) Plotter → Imagesetter → Tape, (c) Blu-ray → Plotter → Tape, (d) Printer → Tape → Blu-ray.

1. **(a)** 4 → 2 → 5: the printer (2) is requested while the tape drive (4) is held; this violates the rule.
2. **(b)** 3 → 1 → 4: the imagesetter (1) is requested while the plotter (3) is held; it violates the rule.
3. **(c)** 5 → 3 → 4: the plotter (3) is requested while the Blu-ray drive (5) is held; it violates the rule.
4. **(d)** 2 → 4 → 5: strictly increasing; the only sequence that follows the rule.

7.7.7 Lock ordering in practice

The software counterpart of resource numbering is **lock ordering**. Return to the example of transfers in opposite directions. T1 transfers from `x` to `y`, and T2 from `y` to `x`. The "natural" order, locking the source account first and then the destination account, is reversed between the two threads and is open to deadlock. The solution is to take the locks according to the accounts' ID numbers; this order is common to all threads.

```
void transfer(Account a, Account b, int m) {
  Account first = a.id < b.id ? a : b;
  Account second = a.id < b.id ? b : a;
  first.lock.lock();
  try {
    second.lock.lock();
    try { a.balance -= m; b.balance += m; }
    finally { second.lock.unlock(); }
  } finally { first.lock.unlock(); }
}
/* T1: transfer(x, y, 1)   x.id = 1
   T2: transfer(y, x, 1)   y.id = 2 */
```

`first` is always the account with the smaller number and `second` the one with the larger number. Whichever direction the transfer goes, the locks are taken in the same order. The transfer line is written in terms of `a` and `b`, so the direction is preserved. Figure 7.38 shows the trace of the two threads.

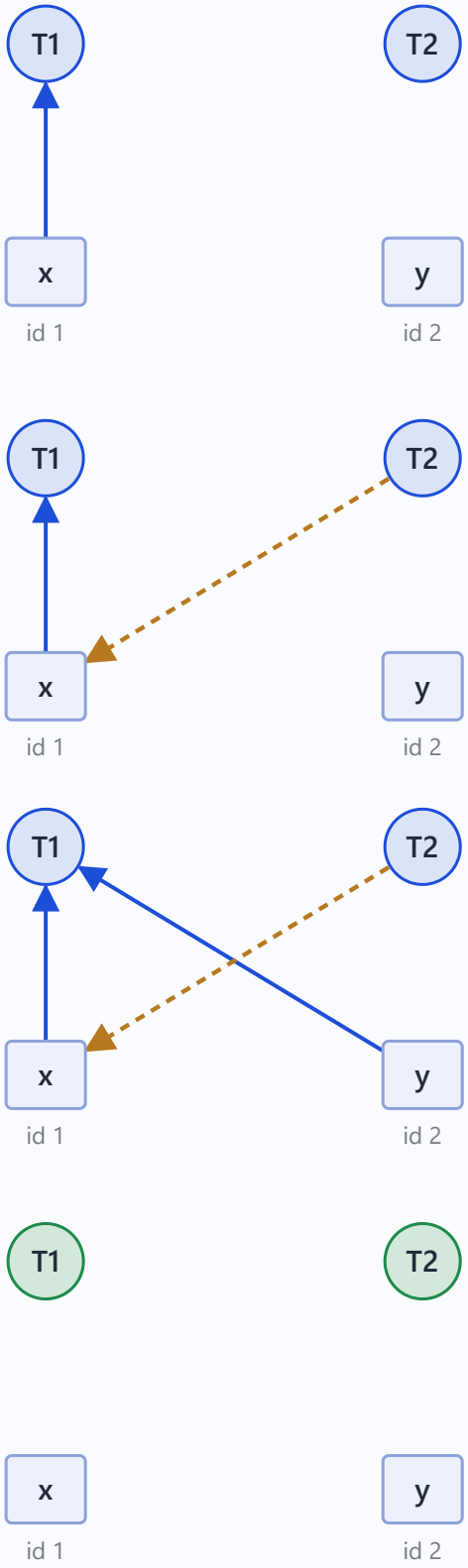


Figure 7.38: Lock ordering. (a) For T1, `first` = x (1 < 2); it takes x. (b) For T2, `a` = y but `first` is still x; T2 waits for x and holds no lock. (c) T1 takes y too and transfers. (d) T1 releases the locks; T2 takes x and then y. No cycle can form.

1. **T1** calls `transfer(x, y, 1)`. Since x.id = 1 < y.id = 2, `first` = x. T1 takes x.
2. **T2** calls `transfer(y, x, 1)`. Here `a` = y, but `first` is still x (id 1). T2 waits for x. What matters is that T2 holds no lock while it waits.
3. **T1** takes y as well; nobody holds y. It makes the transfer.
4. **T1** releases the locks. T2 takes x, then y, and makes its own transfer.

The program was run with JDK 21, each thread making 100,000 opposite-direction transfers; the result is `x=1000 y=1000` and there was no deadlock. The Linux kernel uses the same method: it takes two locks of the same type in order of their memory addresses. The `double_rq_lock` function, which locks two of the scheduler's run queues together, is an example. The lock order in the kernel is documented and checked with `lockdep`.

Common mistake

Applying the lock order in only one function. Ordering works only if it is applied in *every* place where the locks are acquired. If another function takes the same two locks in the opposite order, a cycle can still form. That is why in large projects the lock order is documented and checked with tools.

7.8 Other issues

Deadlock is not the only problem of concurrent programs. The locking protocols of databases, processes that communicate by messages, processes that keep trying but make no progress, and processes whose turn never comes give rise to similar but different problems. This section tells them apart from deadlock.

7.8.1 Two-phase locking

Definition 7.9

Two-phase locking (2PL) is a protocol for database transactions. In the **growing** phase the transaction only acquires locks; in the **shrinking** phase it only releases them. After the first release no new lock may be acquired.

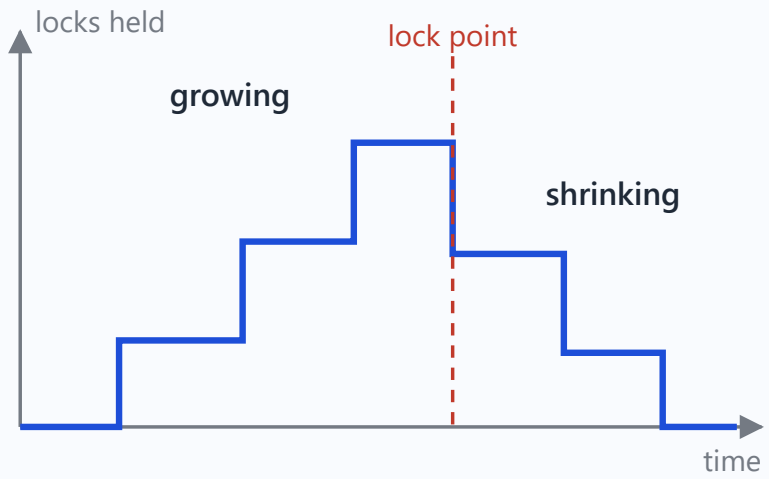


Figure 7.39: Two-phase locking. The number of locks held rises in the growing phase and falls in the shrinking phase after the lock point.

The purpose of 2PL is not to prevent deadlock but **serializability**: the result of concurrently running transactions is as if the transactions had run one after another. Plain 2PL can deadlock. T1 locks record *a* and T2 locks record *b*; then T1 requests *b* and T2 requests *a*. Both transactions are in the growing phase and wait for each other.

Tanenbaum describes a variant: in the first phase the transaction tries to lock all the records. If another transaction has locked one of them, it releases everything it holds and starts over. This variant removes hold and wait, because the transaction never waits while holding a lock. In **strict** 2PL, which is common in today's databases, the locks are released at the end of the transaction, at commit or abort. This variant can deadlock too; the database management system detects the deadlock with a wait-for graph and rolls back one of the transactions.

7.8.2 Communication deadlocks

All the deadlocks so far have been resource deadlocks. In processes that communicate by messages, deadlock can occur without any resource. A sends a request to B and waits for the reply. If the request is lost in the network, A waits forever for the reply and B waits forever for the request. Since each process waits only for an event the other could trigger, this is a deadlock; but no resource is being held.

In this kind of **communication deadlock**, resource ordering and avoidance do not help, because there is no resource to order or to compute with. The solution is the timeout: if the sender gets no reply within a certain time, it sends the message again. This creates a new problem: if the original message was not lost but only delayed, the receiver gets the same request twice. To tell duplicate messages apart, the protocol needs sequence numbers.

In network hardware, buffers are real resources. Figure 7.40 shows a network of four routers connected in a ring.

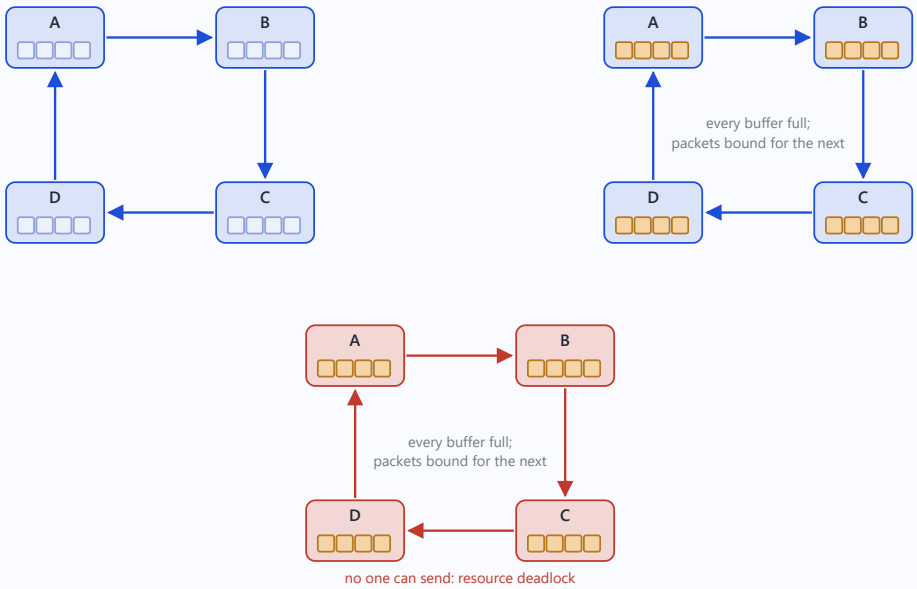


Figure 7.40: Resource deadlock in routers (Tanenbaum §6.7.2). (a) Four routers are connected in a ring; each has limited buffer memory. (b) Under heavy traffic all the buffers fill up with packets bound for the next router clockwise. (c) Nobody can send: A waits for B's free buffer, B for C's, C for D's, and D for A's.

This time the deadlock is a resource deadlock, and the circular wait is plainly visible. The solutions also rest on prevention ideas: dropping packets on timeout (preemption), using virtual channels, and preventing the buffers from filling in the first place with credit-based flow control.

7.8.3 Livelock

Consider two people who meet in a narrow corridor. Both politely step to the right at the same moment, then both step to the left at the same moment. Nobody bumps into the wall, both keep moving, but neither gets past.

> **Definition 7.10**
>
> **Livelock** is a situation in which the processes are not blocked and keep changing their state, but none of them makes progress. In a deadlock the processes sleep; in a livelock they run but do no useful work.

In Tanenbaum's example the processes acquire resources by busy waiting. If `enter_region` cannot get the lock, it keeps spinning in a loop:

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: resource_2 first, then 1 */
```

If B takes the resources in reverse order, A gets `resource_1` and B gets `resource_2`. Then, since neither can get the other's, both spin forever. The processes are not blocked and they burn CPU time; yet they make no progress. A more typical livelock shows up in two threads that back off with tryLock:

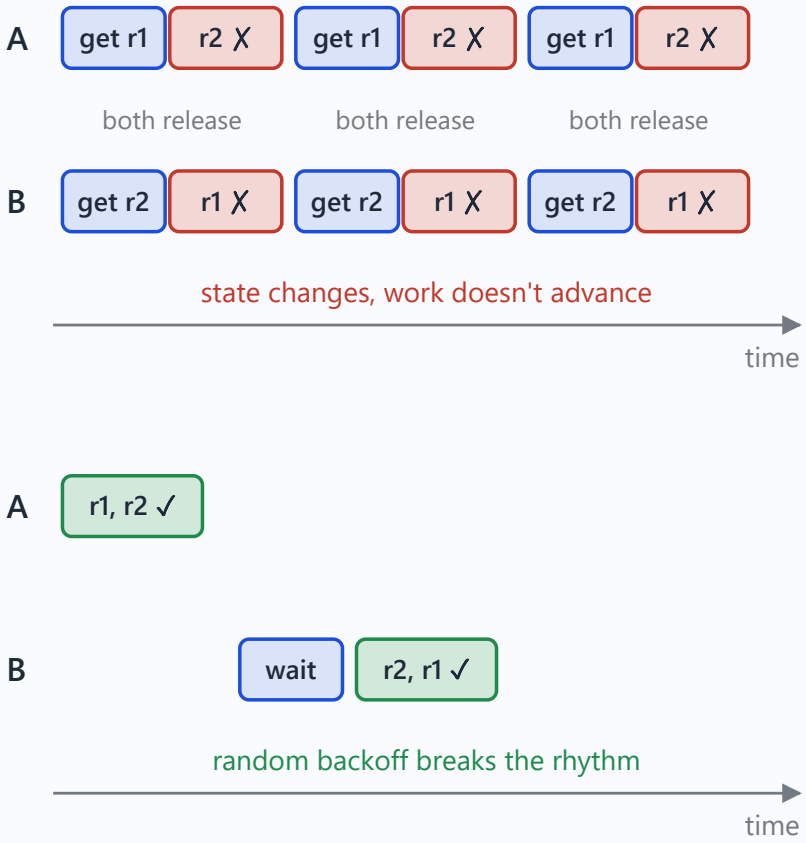


Figure 7.41: Livelock. Top: A gets r1 and B gets r2; each tries the other's, fails, and releases at the same moment. The same rhythm repeats: the state changes, the work does not advance. Bottom: random backoff breaks the rhythm and both processes get their resources.

1. A gets r1 and B gets r2. A tries r2 and B tries r1; both attempts fail.
2. Both behave politely: each releases what it holds and retries after the same delay.

3. The same thing repeats. The processes are not blocked, they use CPU time, and their state keeps changing; but neither makes progress.

The remedy is to break the rhythm. A random wait time that grows with each failure (*exponential backoff*) makes it improbable that the two processes retry at the same moment. Giving priorities or sequence numbers and using lock ordering also prevent livelock. The exponential backoff in Ethernet's CSMA/CD protocol rests on the same idea: two colliding stations wait random times.

7.8.4 Starvation

Definition 7.11

Starvation is the indefinite waiting of a process whose turn never comes even though the resource is available. The other processes progress; the system is not deadlocked.

Starvation is a consequence of the resource-allocation policy. For example, giving the printer to the smallest file every time reduces the average waiting time. But if new small jobs keep arriving, a 100 MB file never gets its turn.

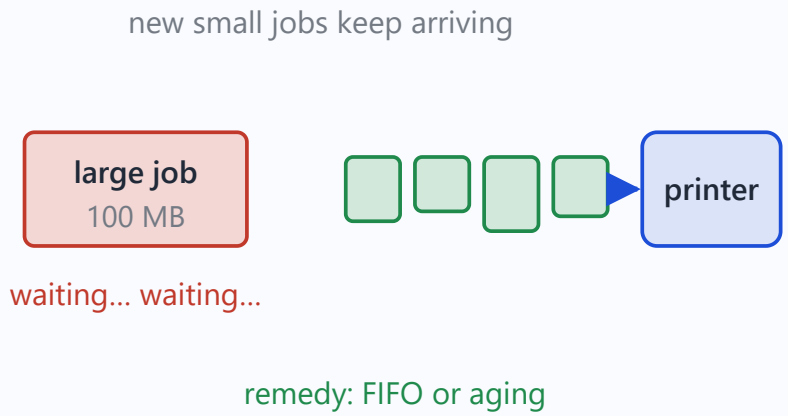


Figure 7.42: Starvation. Under the shortest-job-first policy, a steady stream of small jobs makes the large job wait indefinitely. The remedy is FIFO or aging.

There are other examples of starvation: a low-priority job under strict priority scheduling, a writer in a readers-preference readers–writers lock, a process chosen as the victim every time during recovery. The remedy is either a first-in, first-out (FIFO) policy or **aging**, which raises a process's priority as it waits. Table 7.14 compares the three problems.

	Deadlock	Livelock	Starvation
Process state	blocked	running	ready or blocked

	Deadlock	Livelock	Starvation
Who is affected?	everyone on the cycle	the interacting group	one or a few processes
Does the system progress?	those off the cycle do	the group does not	yes, only the starved one does not
Typical remedy	ordering, detection, avoidance	random backoff	FIFO, aging

Table 7.14: Comparison of deadlock, livelock, and starvation.

7.8.5 Bugs that are not deadlocks: atomicity and ordering

When people think of concurrency bugs, deadlock comes to mind first, but the most common bug is not deadlock. In 2008 Lu and colleagues studied 105 concurrency bugs in MySQL, Apache, Mozilla, and OpenOffice. Of these, 74 are non-deadlock bugs, and almost all of them fit two patterns.

Atomicity violation. Another thread gets in between two accesses that were supposed to be atomic together.

```
/* T1 */
if (sess->name != NULL)
    puts(sess->name);  /* NULL now? */
/* T2 */
sess->name = NULL;
```

T1 first checks that `sess->name` is not `NULL`, then prints it. If T2 sets the field to `NULL` between the two lines, `puts` is called with a null pointer and the program crashes. The remedy is to put the check, the use, and T2's write under the same mutex.

Order violation. It is assumed that "A happens before B," but nothing enforces that order.

```
/* T1: init() */
worker = thread_create(body);
/* T2: body(), the new thread */
s = worker->state;  /* worker set yet? */
```

T1 creates the new thread and assigns its handle to the variable `worker`. The new thread (T2) accesses `worker->state` on its first line; the programmer has assumed that the assignment will be made first. But the new thread can start running before `thread_create` returns and the assignment is made. Then `worker` has not been assigned yet. The remedy is a `ready` flag and a condition variable: T2 waits with `while (!ready) wait`, and T1 sets `ready = 1` after the assignment and signals. A semaphore that starts at zero does the same job.

Common mistake

Thinking “if every access is locked, the code is correct.” Even if the check and the use are each locked separately, an atomicity violation persists if the lock is released in between. Data-race tools such as ThreadSanitizer may not see this case, because each access is protected on its own. The unit that must be atomic is not the individual accesses but the check and the use as a whole.

7.9 Deadlock in practice

General-purpose operating systems do not concern themselves with the deadlocks of user programs; the responsibility lies with the programmer. This section explains, with Java and Linux examples, how deadlock shows up in everyday code, how encapsulation hides it, and with what tools it is found. The Java programs in this section were compiled and run with JDK 21.

7.9.1 Deadlock in Java

In Java every object has a **monitor**. A `synchronized (a) { … }` block acquires the monitor of `a` and releases it when the block ends. A monitor can belong to only one thread at a time; this is mutual exclusion. The following two threads acquire two monitors in opposite orders:

```
Object a = new Object(), b = new Object();

Thread t1 = new Thread(() -> {
  synchronized (a) {
    nap(10);   /* widen the window */
    synchronized (b) { work(); }
  }
}, "T1");
Thread t2 = new Thread(() -> {
  synchronized (b) {
    nap(10);
    synchronized (a) { work(); }
  }
}, "T2");
t1.start(); t2.start();
```

`nap(10)` is a helper method that calls `Thread.sleep(10)`; it widens the deadlock window after the first lock has been acquired. `work()` stands for the work done under both locks. The states of Java threads also help in tracking down a deadlock: `RUNNABLE` means ready to run or running, `TIMED_WAITING` means in a timed sleep, and `BLOCKED` means waiting for a monitor.

Step	Event	Owner of a	Owner of b	T1	T2
1	`t1.start(); t2.start();`	—	—	RUNNABLE	RUNNABLE
2	T1 enters a's monitor and sleeps	T1	—	TIMED_WAITING	RUNNABLE
3	T2 enters b's monitor and sleeps	T1	T2	TIMED_WAITING	TIMED_WAITING
4	T1 wakes and requests b: b is held by T2	T1	T2	BLOCKED	TIMED_WAITING
5	T2 wakes and requests a: a is held by T1	T1	T2	BLOCKED	BLOCKED

Table 7.15: Trace of `synchronized` blocks in reverse order.

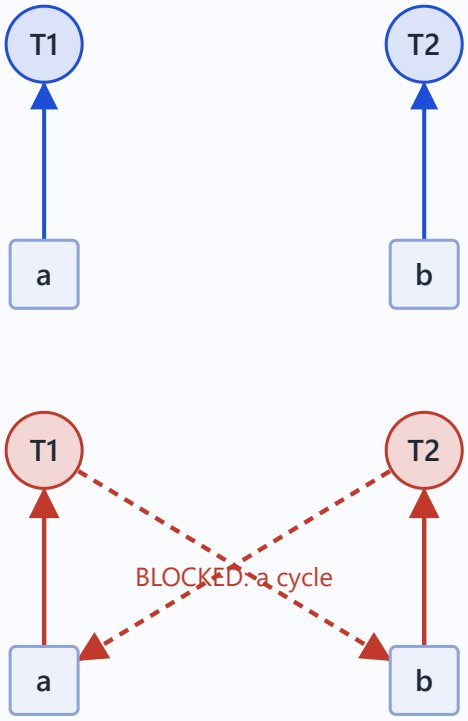


Figure 7.43: Deadlock in Java. (a) T1 holds a and T2 holds b. (b) Each requests the other's monitor and becomes `BLOCKED`: a cycle.

JVM monitors are nonpreemptable; there is no way to interrupt a waiting `synchronized` block, to put a timeout on it, or to take the lock back. The program never finishes. `nap(10)` makes the deadlock very likely but does not guarantee it: if one thread acquires the second lock before the other has even started, the program finishes normally. When these notes were checked, the program was run with JDK 21, and in five runs out of five the two threads stayed `BLOCKED`. A version that sleeps 50 ms instead of 10 ms likewise printed nothing until it was stopped by an eight-second timeout. A version that makes the threads wait with a `CountDownLatch` until both first locks have been acquired deadlocks on every run, because it leaves no room for timing.

7.9.2 Encapsulation hides lock order

Reverse-ordered locks are not always plainly visible in the code. `Collections.synchronizedList` wraps a list and protects each of its methods with the list's own lock. The call `a.addAll(b)` first acquires the lock of `a` internally. Then, to copy the argument's elements, it calls `b.toArray()`, which acquires the lock of `b`.

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* inside a.addAll(b):               */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Fix: take the copy outside the lock */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

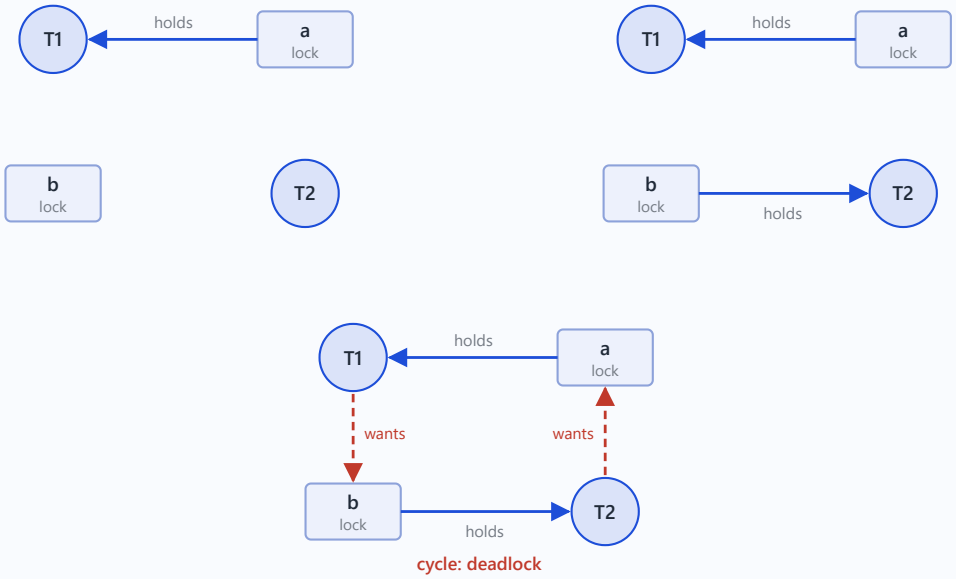


Figure 7.44: A hidden lock order. (a) T1: `a.addAll(b)` acquires a's lock first. (b) At the same time T2: `b.addAll(a)` acquires b's lock first. (c) Internally `b.toArray()` and `a.toArray()` are called; each asks for the other's lock.

If two threads call `a.addAll(b)` and `b.addAll(a)` at the same time, an acquisition in reverse order arises on its own: one locks in the order a → b, the other in the order b → a. The person who wrote the code sees no lock at all. The fix is to take the copy outside the lock: `b.toArray()` is done first in a

separate call and releases its own lock; then `a.addAll` acquires only the lock of `a`. At no moment are two locks held together.

Cross call	JDK 8	JDK 21
`Vector`: `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList`: same	10/10	10/10
Fixed version	0/10	0/10

Table 7.16: Cross `addAll` trials (2 threads × 200,000 rounds; number of deadlocked runs out of 10; the result can vary from run to run, e.g. 9/10).

The interesting row in the table is `Vector`. In JDK 8, `Vector.addAll` calls the argument's `toArray()` while holding its own lock, and deadlocks. In JDK 21 it calls the argument's `toArray()` before taking its own lock, so it does not deadlock. The same source code behaves differently because an internal detail of the library changed. The rule is: do not call a foreign method, that is, one whose internal locking is unknown, while holding a lock. When these notes were checked, the same experiment was repeated: in JDK 8, `Vector` and `synchronizedList` deadlocked in 10 of 10 trials. In JDK 21, `synchronizedList` deadlocked in 9 of 10 trials and `Vector` never deadlocked. The fixed code never deadlocked in either version. Because deadlock depends on chance, the difference between 9 and 10 is expected variation.

7.9.3 Finding a deadlock: jstack

A deadlocked Java program gives no error message; it simply stops. The `jstack` tool that comes with the JDK prints the stacks of all threads in a running JVM and, if there is a deadlock, reports it separately at the end of the output:

```
$ jstack <pid>
Found one Java-level deadlock:
=============================
"T1":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T2"
"T2":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T1"

Java stack information for the threads listed above:
"T1":
        at Hang.run(Hang.java:7)
        - waiting to lock <0x…d0> (a java.lang.Object)
        - locked <0x…c0> (a java.lang.Object)
…
Found 1 deadlock.
```

The report says who is waiting for which monitor and who holds that monitor: T1 waits for the monitor held by T2, and T2 for the one held by T1. The stack information also shows on which line of the code the deadlock arises. When these notes were checked, `jstack` was run while the deadlocked program was running on JDK 21, and the report came out in the same form: "Found one Java-level deadlock," the "waiting to lock" and "which is held by" lines for the two threads, and at the end "Found 1 deadlock." The same report can be obtained with `jcmd <pid> Thread.print`; in graphical tools such as JConsole and VisualVM, the "Detect Deadlock" button does the same job.

A program can also find its own deadlock. The `ThreadMXBean` interface exposes the JVM's thread information:

```
ThreadMXBean mx =
    ManagementFactory.getThreadMXBean();
long[] ids = mx.findDeadlockedThreads();
if (ids != null)
  for (ThreadInfo ti : mx.getThreadInfo(ids))
    System.out.println(ti.getThreadName()
      + " waits, lock owner "
      + ti.getLockOwnerName());
```

`findDeadlockedThreads()` returns the IDs of the deadlocked threads; if there is no deadlock it returns `null`. On the deadlocked program this code prints the following:

```
T1 waits, lock owner T2
T2 waits, lock owner T1
```

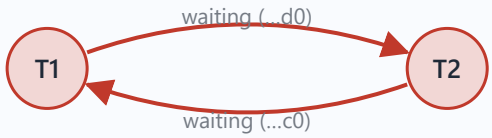


Figure 7.45: The wait-for graph the JVM builds: T1 waits for the object held by T2 (...d0), and T2 for the object held by T1 (...c0).

The JVM builds a wait-for graph from monitor and `ReentrantLock` ownership and searches it for cycles; that is, it runs the detection algorithm for single-instance resources. But it only detects; it does not recover. The JVM cannot kill a thread and hand its monitor to someone else, because the protected data might be left inconsistent. The deadlocked process has to be restarted.

7.9.4 lockdep and other tools

Every environment has its own tools for deadlock. Table 7.17 lists the main ones.

Environment	Tool or behavior
Linux kernel	`lockdep` (`CONFIG_PROVE_LOCKING`): lock-order validator; hung-task detector
C/C++	ThreadSanitizer: `lock-order-inversion` warning; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	runtime: `fatal error: all goroutines are asleep - deadlock!`
Database	PostgreSQL, InnoDB: wait-for graph and rollback of a victim transaction

Table 7.17: Deadlock-finding tools.

`lockdep` in the Linux kernel rests on a different idea. It ties every lock to a **class**; all lock instances of the same kind belong to the same class. When a task holding a lock of class A acquires a lock of class B, an edge "A → B" is added to the graph. If on another code path A is acquired while B is held, a cycle forms in the graph and `lockdep` issues a "possible circular locking dependency" warning.

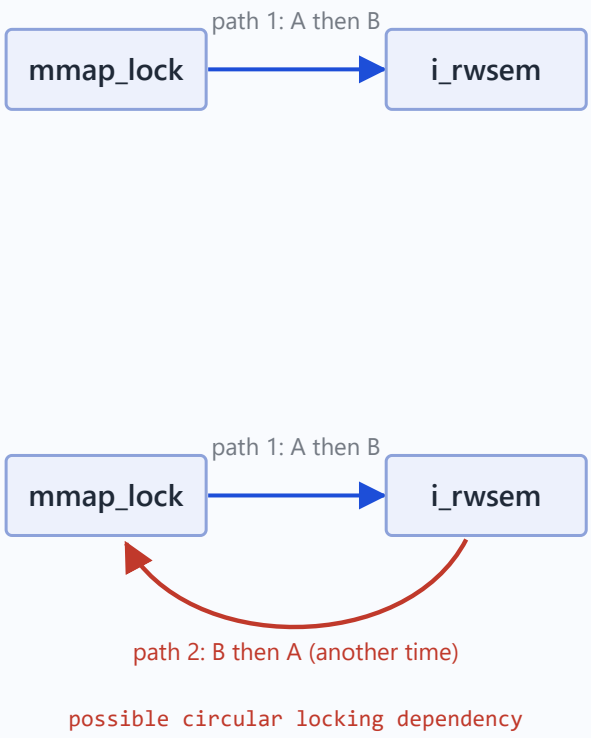


Figure 7.46: `lockdep` . (a) On one path, `i_rwsem` is acquired while `mmap_lock` is held: the edge A → B is recorded. (b) At another time, on another path, the reverse order is seen: the cycle is reported without a deadlock ever having occurred.

The two code paths may never have run at the same time; `lockdep` warns all the same. Tools fall into two groups in this respect. `lockdep` and ThreadSanitizer find a *possible* deadlock, that is, a lock-order violation, even if no deadlock has ever occurred. `jstack` and the Go runtime report a deadlock

that *has occurred*. The first group are tools of the prevention strategy and the second of the detection strategy.

7.10 Summary

- **Deadlock** is the situation in which every process in a set waits for an event that only another process in the set can trigger. Deadlocks concern **nonpreemptable** resources; **preemptable** resources such as the CPU and memory can be reallocated.
- A process **requests, uses, and releases** a resource. Acquiring two resources in reverse order turns a program that never deadlocks into one that deadlocks now and then.
- The **four conditions** are jointly necessary: mutual exclusion, hold and wait, no preemption, and circular wait. The first three are policy; the fourth is the current state.
- If the **resource-allocation graph** has no cycle, there is no deadlock. With single-instance resources, cycle ⇔ deadlock; with multiple-instance resources a cycle is necessary but not sufficient. In the single-instance case the **wait-for graph** is used.
- There are four strategies: ignoring (the ostrich), detection and recovery, avoidance, and prevention. General-purpose operating systems choose the ostrich approach for user processes.
- With single-instance resources, detection is cycle search; colored DFS runs in $O(V + E)$. With multiple-instance resources, the loop "a process whose request can be met from the free resources finishes and releases what it holds" is run on the structures E, A, C, R; its cost is $O(m \cdot n^2)$.
- **Recovery** is done by resource preemption, rollback, or termination, and always costs some lost work. If the same process is always chosen as the victim, starvation results.
- In a **safe state** there is a sequence that finishes all processes even if every process makes its maximum request. An unsafe state is not a deadlock, but it carries no guarantee of completion.
- The **banker's algorithm** pretends to grant each request and grants it if the new state is safe. A process is made to wait for two reasons: the resource is not available, or granting would make the state unsafe. Because it requires maximum needs to be known in advance, it is rarely used in practice.
- **Prevention** breaks one of the conditions. The most effective method is **lock ordering**: if locks are acquired in an order common to everyone, no cycle can form. Backing off with tryLock breaks the no-preemption condition, but without a random wait it can lead to **livelock**.
- **2PL** serves serializability and can deadlock. **Starvation** is a consequence of policy and is cured by aging. The most common concurrency bugs are not deadlocks but atomicity and order violations.
- In Java a deadlock is found, but not resolved, with `jstack` or `ThreadMXBean`. `lockdep` and ThreadSanitizer report a possible deadlock before any deadlock happens.

Approach	When?	Information needed	Price	Used in
Ignoring	—	none	occasional restarts	general-purpose operating systems

Approach	When?	Information needed	Price	Used in
Detection and recovery	afterwards	current allocations and requests	periodic $O(m \cdot n^2)$, lost work	databases, JVM tools
Avoidance	at every request	maximum needs (Max)	postponements, $O(m \cdot n^2)$ per request	rarely, special systems
Prevention	at design time	resource order	loss of flexibility and utilization	kernels, application code

Table 7.18: Comparison of the four strategies.

7.11 Key terms

deadlock preemptable / nonpreemptable resource Coffman conditions mutual exclusion hold and wait no preemption circular wait resource-allocation graph wait-for graph deadlock detection deadlock recovery checkpoint deadlock avoidance safe / unsafe state banker's algorithm deadlock prevention spooling mutex semaphore monitor busy waiting two-phase locking (2PL) livelock starvation aging atomic instruction

7.12 Test Yourself

1. In a system, processes acquire locks with `trylock`. A process that cannot get the second lock releases the lock it holds and always waits exactly 5 ms before retrying. Which deadlock condition has been broken? What new problem can arise in this design, and how is it remedied?

   Answer

   Instead of waiting, the process releases the resource it holds of its own accord; the resource has in effect been taken back from its owner. This is a voluntary breaking of the no-preemption condition. If two processes acquire and release in the same rhythm, neither makes progress: livelock. The processes are not blocked; they use CPU time. The remedy is to make the wait time random and to increase it after each failure (exponential backoff), or to acquire the locks in a common order.

2. In a wait-for graph the edges are P1 → P2, P2 → P3, P2 → P4, P4 → P5, and P5 → P3. Colored DFS starts at P1 and tries neighbors in numerical order. Is a cycle found? In what order are the nodes

colored black?

Answer

P1, P2, P3 are colored gray in turn. P3 has no neighbors; the first black node is P3. P2's next neighbor, P4, is white; P4 and then P5 become gray. P5's only neighbor, P3, is black; an edge to a black node is not a cycle. P5 turns black, followed by P4, P2, and P1. The order of blackening is P3, P5, P4, P2, P1, and `cycle` returns `false`: there is no cycle. The edge P5 → P3 only shows that two paths merge at P3.

3. There are three resource types and E = (3 2 2). The allocations are P1 (1 0 1), P2 (1 1 0), P3 (0 1 1); the requests are P1 (1 1 0), P2 (0 0 1), P3 (1 0 0). If the detection algorithm is run, rescanning from the beginning after each marking, does the system come out deadlocked? What would happen if P3's request were (1 0 1)?

Answer

The allocations sum to (2 2 2); A = (1 0 0). P1 (1 1 0) asks for the second type, and none is free. P2 (0 0 1) asks for the third type, and none is free. P3 (1 0 0) ≤ (1 0 0) ✓; P3 finishes and A = (1 1 1). From the start again: P1 (1 1 0) ≤ (1 1 1) ✓, A = (2 1 2). P2 (0 0 1) ✓, A = (3 2 2) = E. The order is P3, P1, P2; there is no deadlock. If P3's request were (1 0 1), then with A = (1 0 0) none of the three rows could be satisfied, and all three processes would be deadlocked.

4. There are 12 units of a single resource type. A holds 5 units (at most 10), B holds 2 (at most 4), C holds 2 (at most 9). Is the state safe? Which are the safe sequences? In this state, if C asks for one more unit, should the request be granted?

Answer

There are 12 − 9 = 3 units free. The remaining needs are A 5, B 2, C 7. Only B can be satisfied (2 ≤ 3); B finishes and free becomes 5. Then only A can be satisfied (5 ≤ 5); A finishes and free becomes 10. Finally C (7 ≤ 10) finishes. The only safe sequence is ⟨B, A, C⟩; the state is safe. If C is given 1 unit, free is 2 and the needs are A 5, B 2, C 6. B finishes and free rises to 4; but neither A's 5 nor C's 6 can be met. The state is unsafe; the request is refused and C is made to wait.

5. Five processes request resources of the same type one at a time; each uses at most 3 instances at once and releases them all when finished. What is the smallest number of instances that

guarantees deadlock never occurs? How does one fewer lead to deadlock?

Answer

In the worst case each process holds 2 instances and waits for a third: $5 \cdot 2 = 10$ instances. If there is one more instance, one of the processes gets its third, finishes, and releases 3 instances; the others finish in turn. At least $5 \cdot 2 + 1 = 11$ instances are needed. With 10 instances each of the five processes can hold two; no instance is left free and everyone waits for a third: deadlock.

6. In the initial state of the Silberschatz example (Available = (3 3 2)), T4 requests (3 3 0). What does the banker's algorithm decide? In the same initial state, what would the decision be if T0 requested (0 2 0)?

Answer

T4: (3 3 0) ≤ Need_4 = (4 3 1) and ≤ (3 3 2); both checks pass. In the pretend state Available = (0 0 2) and Need_4 = (1 0 1). None of the rows T0 (7 4 3), T1 (1 2 2), T2 (6 0 0), T3 (0 1 1), and T4 (1 0 1) can be satisfied with (0 0 2): the state is unsafe and the request is refused. T0: in the pretend state Available = (3 1 2) and Need_0 = (7 2 3). T3 (0 1 1) finishes → (5 2 3); T1 (1 2 2) → (7 2 3); T2 (6 0 0) → (10 2 5); T0 (7 2 3) → (10 5 5); T4 (4 3 1) → (10 5 7). The safe sequence ⟨T3, T1, T2, T0, T4⟩ exists; the request is granted. The same request was refused after T1's request (1 0 2): the decision depends not on the request itself but on the whole state.

7. In a program, the function `transfer(a, b)` acquires locks in order of account number. The function `report(acct1, acct2)` in the same program always acquires the locks of two accounts in the order `acct1` first, then `acct2`. Can the program deadlock? Why?

Answer

It can. Lock ordering prevents cycles only if it is applied everywhere the locks are acquired. A call `report(y, x)` can hold y and request x. If at the same time `transfer` holds x in numerical order and requests y, a cycle forms. The remedy is to make `report` follow the same order. Tools such as `lockdep` or ThreadSanitizer report this reverse order without any deadlock ever occurring.

8. In one Java server two threads have fallen into a livelock; in another server two threads are deadlocked. Which one does `jstack` report as “Found one Java-level deadlock”? How does CPU

utilization look in the two cases?

Answer

`jstack` reports only the deadlock: the deadlocked threads are `BLOCKED`, and the wait-for graph built from monitor ownership contains a cycle. Threads in a livelock appear as `RUNNABLE` or, during short sleeps, `TIMED_WAITING`. Because they keep acquiring and releasing locks, there is no permanent cycle in the graph and no report is produced. In a deadlock CPU utilization drops, because the threads sleep. In a livelock the CPU looks busy but the work does not advance.

7.13 Further reading

- A. S. Tanenbaum and H. Bos, *Modern Operating Systems* (4th ed.), Chapter 6: treats the resource-allocation graph, the cycle-search and matrix-based detection algorithms, resource trajectories, and the banker's algorithm in detail, with the examples used in this chapter.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th ed., 2018), Chapter 8: gives the four conditions, the prevention protocols, and the safety and request parts of the banker's algorithm in pseudocode; it also introduces the Linux `lockdep` tool.
- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapter “Common Concurrency Problems”: describes atomicity and order violations, the four conditions for deadlock, and lock ordering with bugs taken from real systems. The book is online and free.
- S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics,” ASPLOS 2008: a study that classifies 105 concurrency bugs in four large open-source projects and shows the weight of non-deadlock bugs.

7.14 References

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, ostep.org — “Common Concurrency Problems”.
2. E. G. Coffman, M. J. Elphick, A. Shoshani, “System Deadlocks”, *ACM Computing Surveys* 3(2), 1971.
3. E. W. Dijkstra, “Cooperating Sequential Processes”, EWD-123, 1965.
4. R. C. Holt, “Some Deadlock Properties of Computer Systems”, *ACM Computing Surveys* 4(3), 1972.
5. Linux kernel documentation: Documentation/locking/lockdep-design.rst.
6. S. Lu, S. Park, E. Seo, Y. Zhou, “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”, ASPLOS 2008.
7. Java SE 21 API documentation: `java.lang.management.ThreadMXBean`, `java.util.concurrent.locks.ReentrantLock`; the `jstack` manual.
8. PostgreSQL documentation: “Explicit Locking — Deadlocks”, `deadlock_timeout`.

9. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 8: Deadlocks.
10. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014 (5th ed. 2023) — Chapter 6: Deadlocks.

PART III

Memory and Storage

Chapters 8–11

Memory Management

A program can run only after it has been loaded into main memory. The CPU reads instructions and data only from registers and main memory; the disk is a slow store reached only through I/O operations. In a system that runs many processes at once, main memory has to be shared among them. Sharing raises three questions. How can one process be kept from reaching another's memory? How do addresses stay correct when a program is loaded anywhere in memory? How is free space tracked, and how is it kept from breaking into pieces? These notes begin with the memory hierarchy and the cache. They then describe address translation, starting from base and limit registers. Binding and loading, swapping, contiguous allocation, and free-space management follow. The emphasis is on paging, the foundation of today's systems: the page table, the TLB, effective access time, and multilevel, hashed, and inverted tables. The last sections cover segmentation and compare the real designs of x86-64, ARMv8, and RISC-V.

LEARNING OUTCOMES

Abstraction
Explain the memory hierarchy, cache lines, and average access time; explain the address space and the problems of protection and relocation.

Addresses
Explain binding times, the distinction between logical and physical addresses, and the MMU's base–limit check.

Linking
Discuss dynamic loading, static and dynamic linking, shared libraries with the PLT/GOT, and swapping.

Contiguous allocation
Apply the fit algorithms to a request sequence; analyze internal and external fragmentation, bitmaps, and linked lists.

Paging
Split an address into (p, d), translate with a page table and a TLB, and compute the effective access time.

Real systems
Compare hierarchical, hashed, and inverted tables; four- and five-level paging on x86-64; huge pages; segmentation; and ARMv8.

8.1 Memory hardware

To understand memory management one must first look at memory itself. The operating system sees memory as an abstract array of bytes, but behind that array lie several layers of very different speeds and sizes. Knowing which layer the hardware manages and which the operating system manages explains all the later design decisions.

8.1.1 The memory hierarchy and the memory manager

The CPU can reach storage directly in only two places: registers and main memory. An instruction takes its operand from a register or from memory; it cannot reach data on disk directly. Data on disk must first be brought into memory by an I/O operation. That is why the code and data of a running program must be in main memory.

The memory unit also offers a very simple interface. It receives only a stream of addresses: either "read this address" or "write this data to this address." The memory unit does not know where an address came from. It may have been produced from a program counter, an array index, or a pointer; to memory they are all the same. This simplicity also shows why a layer of hardware that modifies addresses before they reach memory is possible.

The speed gap is very large. A register is read in one clock cycle. A DRAM access in main memory takes hundreds of cycles, and the CPU stalls in the meantime. The gap is closed by small, fast memories inside the processor, the **caches**. Figure 8.1 shows typical values today: registers under a nanosecond, L1 and L2 cache 1–5 ns, L3 10–20 ns, DRAM 80–120 ns, NVMe SSD 10–100 µs, hard disk 5–10 ms. Going from top to bottom, latency and capacity grow and the cost per byte falls.

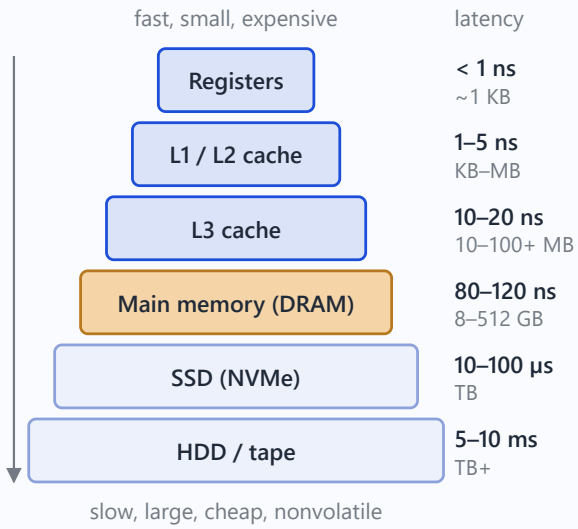


Figure 8.1: The memory hierarchy. From registers toward disk, latency and capacity grow. The values are typical orders of magnitude as of 2026; they vary with core and model.

What a programmer expects of memory is in fact an impossible list: that it be unlimited, very fast, persistent, and protected. No single technology gives all these properties at once. The real solution is the hierarchy. Small, fast layers hold the frequently used data, and large, slow layers hold the rest. A well-built hierarchy gives the programmer the impression of a memory "almost as fast as the top layer and almost as large as the bottom layer."

Different managers control the layers of the hierarchy. Hardware manages the caches entirely; the operating system does not interfere with their contents. The operating system manages main memory. The part of the operating system that handles this job is called the **memory manager**.

Definition 8.1

Memory management is the sharing of main memory among processes. The memory manager keeps track of which regions of memory are free and which are in use; it allocates memory to processes, takes it back when they finish, and protects processes from one another. When memory runs short, moving some of the processes or pages to disk is also its job.

The memory manager's job is often confused with the CPU scheduler's job. Choosing the next process from the ready queue is the scheduler's task. The memory manager deals with questions such as "which process is where, how much room does it occupy, and how much more can be given to whom."

8.1.2 Types of main memory

In everyday language the word “memory” is used for several different kinds of hardware. Table 8.1 compares the four basic types. The type used as main memory is **RAM** (random access memory). RAM can be read and written, is fast, but is volatile: its contents are lost when power is cut. DRAM cells are tiny capacitors, and because charge leaks away they must be refreshed at regular intervals. SRAM needs no refresh and is faster, but it is more expensive and takes more space. That is why main memory is DRAM (today DDR5, LPDDR5X) and the caches are SRAM.

Type	Properties	Use
RAM (random access)	Readable and writable, fast, volatile. DRAM needs refresh; SRAM is faster and more expensive.	Main memory (DRAM: DDR5, LPDDR5X), caches (SRAM)
ROM (read-only)	Written at manufacture, cannot be changed, persistent, cheap.	Simple devices, fixed code in microcontrollers
EEPROM / Flash	Can be erased and rewritten electrically, persistent; writes are slow and write endurance is limited.	UEFI/BIOS firmware (SPI flash), SSDs, USB sticks, phone storage (NAND flash)
CMOS / NVRAM	A small area kept alive by a battery or flash.	Clock, boot settings

Table 8.1: Types of main memory.

ROM (read-only memory) is written during manufacture and cannot be changed afterward. It is persistent and cheap; in simple devices and microcontrollers it holds code that will never change. **EEPROM** and its variant **flash memory** can be erased and rewritten electrically. They are persistent, but writing is much slower than reading, and a cell can be written only a limited number of times. In today's computers the BIOS/UEFI code is not in ROM but on an SPI flash chip on the motherboard, which is why firmware can be updated. SSDs, USB sticks, and phone storage also use NAND flash. Finally, **CMOS** or **NVRAM** is a small area kept alive by a battery or flash; the clock and the boot settings are kept there.

Memory management is concerned with only one of these types: the main memory in which processes run, that is, DRAM. In the rest of the chapter the word “memory” is used in this sense.

8.1.3 Cache lines and locality

Data moves between the cache and main memory not byte by byte but in fixed-size blocks. This block is called a **cache line**. In most processors today a line is 64 bytes. Memory is thought of as divided into consecutive lines of this size: line 0 covers addresses 0–63, line 1 covers 64–127, and so on.

Which line an address a lies in is found by a simple division. The line number is $\lfloor a / 64 \rfloor$ and the position within the line is $a \bmod 64$. Since $64 = 2^6$, the division is really a bit-splitting operation: the low 6 bits of the address give the position within the line, and the remaining upper bits give the line number. Figure 8.2 takes address 200 as an example. In binary 200 is 11001000. The low six bits are 001000, that is, 8; the upper two bits are 11, that is, 3. Indeed $3 \times 64 + 8 = 200$.

Figure 8.2: Splitting address 200 with 64-byte lines: the upper bits give line number 3, the low 6 bits give offset 8 within the line.

When the CPU reads a word, the hardware looks in the cache first. If the line is in the cache, this is a **hit**, and the data arrives from L1 in 4–5 cycles. If not, there is a **miss**; the request goes down a level, in the worst case to DRAM, and takes hundreds of cycles. Because the cache is expensive it is small. Instead of one big cache, levels such as L1, L2, and L3 are built, each larger and slower than the one before.

A small cache is useful in front of a large memory because of how programs behave. Programs do not access memory at random; their accesses follow two patterns.

Temporal locality
: A piece of data used a moment ago is used again soon. A loop counter, the instructions of a loop body, and a frequently called function are examples.

Spatial locality
: A program that accesses one address soon accesses neighboring addresses too. Walking through an array from start to end and executing consecutive instructions are examples.

The cache line exploits spatial locality: because all 64 bytes are fetched on a miss for one byte, later accesses to neighboring bytes are hits. Temporal locality is exploited by the line staying in the cache.

8.1.4 The path of a read through the hierarchy

Suppose the CPU wants to read variable x with the instruction `LOAD R1, [x]`. The hardware first looks at the nearest and fastest level, L1. If the line is not in L1, about 1 ns has been lost and the search continues in L2. If L2 and L3 miss too, the latencies of the levels add up: 1 + 4 + 15 = 20 ns. Finally the data comes from DRAM; this adds about 80 ns, for a total of 100 ns (Figure 8.3).

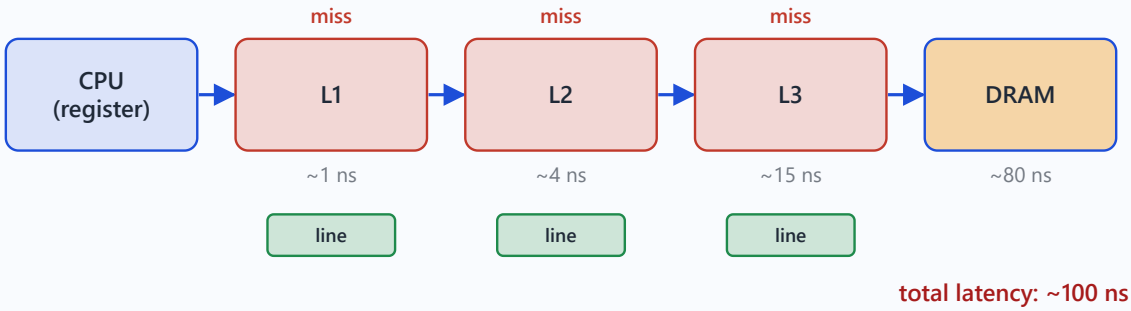


Figure 8.3: A read that misses at all three levels. The data comes from DRAM; not a single word but the whole 64-byte line is placed in L3, L2, and L1. The total latency is about 100 ns.

What comes from DRAM is not a single word but the whole 64-byte line containing the word. On the way the line is placed in L3, L2, and L1 as well. When another piece of data in the same line is accessed a second time, it is now an L1 hit and about 1 ns suffices (Figure 8.4). The 100 ns cost of the first access is thus shared with the accesses to the rest of the line. This is the locality that makes the cache worthwhile.

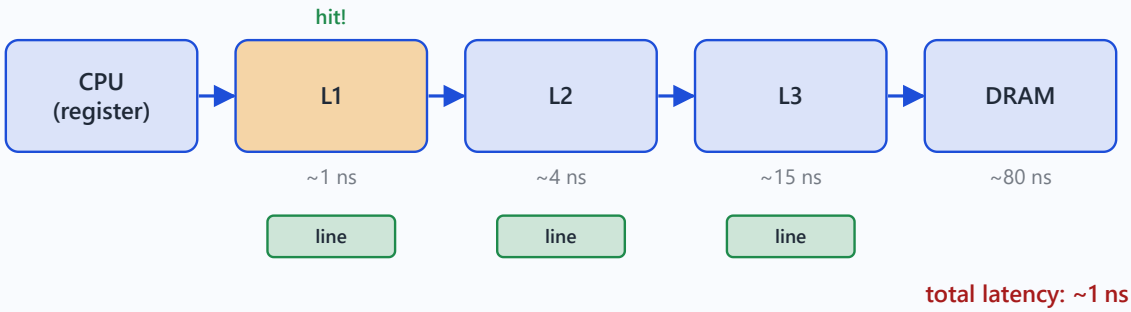


Figure 8.4: A second access to the same line is an L1 hit; the latency drops to about 1 ns.

8.1.5 Code trace: a direct-mapped cache

Where in the cache can a line be placed? The simplest answer is **direct mapping**: the cache is divided into a number of *sets*, and each block can go to only one set. If the block number is b and the number of sets is K, block b goes to set b mod K. If another block is currently in the set, the newcomer evicts it.

The C fragment below simulates a direct-mapped cache with four sets and 64-byte lines. The array `tag` records which block is currently in each set; −1 means empty.

```
int addr[6] = {0, 8, 64, 256, 0, 72};
int tag[4] = {-1, -1, -1, -1};
int hits = 0, misses = 0;
for (int i = 0; i < 6; i++) {
  int block = addr[i] / 64;
  int set = block % 4;
  if (tag[set] == block) hits++;
  else { tag[set] = block; misses++; }
}
```

For each address the loop first computes the block number (`addr / 64`) and then the set (`block % 4`). If the tag in the set equals this block, the hit counter is incremented. Otherwise the block is written into the set and the miss counter is incremented. In this simplified model the whole block number is stored as the tag. Real hardware already knows the set number from the address, so it stores only the upper bits, the real tag.

i	addr	block	set	Result	tag[0..3] afterwards	Comment
0	0	0	0	miss	0, −1, −1, −1	set 0 empty
1	8	0	0	hit	0, −1, −1, −1	8 is in the same line as 0: spatial locality
2	64	1	1	miss	0, 1, −1, −1	set 1 empty
3	256	4	0	miss	4, 1, −1, −1	block 4 evicts block 0 (conflict miss)
4	0	0	0	miss	0, 1, −1, −1	block 0 evicts block 4 (conflict miss)
5	72	1	1	hit	0, 1, −1, −1	72 is in the same line as 64

Table 8.2: Trace of the direct-mapped cache code (4 sets, 64 B lines). The values were verified by recomputing them in Python.

When the loop ends, `hits = 2` and `misses = 4` ; the hit rate is 33%. Two different kinds of miss show up in the trace. The first three misses are unavoidable: the blocks come into the cache for the first time. These are called *compulsory misses*. The misses at steps 3 and 4 have a different cause. Addresses 0 and 256 are in different blocks (0 and 4), but both fall into set 0. Although sets 2 and 3 of the cache sit empty, these two blocks evict each other. This kind of miss is called a **conflict miss**.

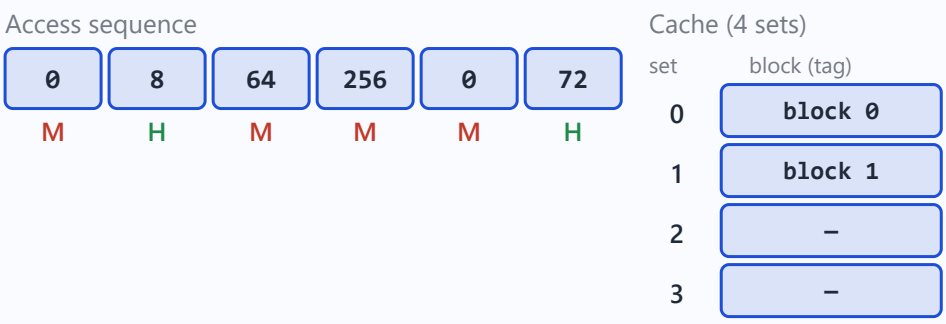


Figure 8.5: The final state of the direct-mapped cache trace. M marks a miss and H a hit. Because 0 and 256

fall into the same set, they evicted each other.

The way to reduce conflict misses is to give a block more than one place. In a **k-way set-associative** cache each set holds k lines. A block still goes to a single set, but it can be placed in any of the k slots of that set.

Example 8.1

Let the same four-line cache be organized as 2 sets with 2 ways, and let the least recently used line be evicted when a set needs room. How many hits are there on the same sequence 0, 8, 64, 256, 0, 72?

1. **Sets.** The blocks are 0, 0, 1, 4, 0, 1. The set number is now block mod 2: 0, 0, 1, 0, 0, 1.
2. **Trace.** 0: set 0 empty, miss. 8: block 0 is in set 0, hit. 64: set 1 empty, miss. 256: block 4 goes to set 0; since the set has one more free slot, block 0 stays where it is, miss. 0: block 0 is still in set 0, hit. 72: block 1 is in set 1, hit.
3. **Result.** 3 hits, 3 misses. Direct mapping had 2 hits. The gain comes entirely from the disappearance of the conflict miss.

Common mistake

Computing the set number from the address itself (for example 256 mod 4 = 0, 72 mod 4 = 0). What determines the set is not the address but the block number. First the address is divided by the line size, then the remainder of the block number modulo the number of sets is taken: 72 / 64 = 1, 1 mod 4 = 1.

8.1.6 Code trace: average access time

How well a hierarchy works can be summarized in a single number: the **average memory access time**. Every access first goes to the top level and spends that level's time. If it misses, the request goes down a level and that level's average time is added. If a level's time is s, its miss rate is m, and the average time of the level below is T_{below}:

$$T = s + m \cdot T_{below}$$

The formula repeats itself: the time of the level below is computed in the same way. DRAM, at the very bottom, satisfies every request, so there T is just DRAM's time. The code below does the computation from the bottom up.

```c
/* L1, L2, L3, DRAM access time (ns) */
double lat[4] = {1, 4, 20, 80};
double miss[3] = {0.10, 0.40, 0.50};
double t = lat[3];          /* DRAM */
for (int k = 2; k >= 0; k--)
  t = lat[k] + miss[k] * t;
printf("average = %.1f ns\n", t);
```

`t` starts with DRAM's 80 ns. The loop goes down from `k = 2` (L3) to `k = 0` (L1) and computes the time of the next level up in each round. Table 8.3 shows each round.

Step	k	Level	Computation	t (ns)
start	—	DRAM	`t = lat[3]`	80
1	2	L3	20 + 0.50 × 80	60
2	1	L2	4 + 0.40 × 60	28
3	0	L1	1 + 0.10 × 28	3.8
end	−1	—	the loop ends, `average = 3.8 ns` is printed	3.8

Table 8.3: Trace of the average-access-time code. The values were verified by recomputing them in Python.

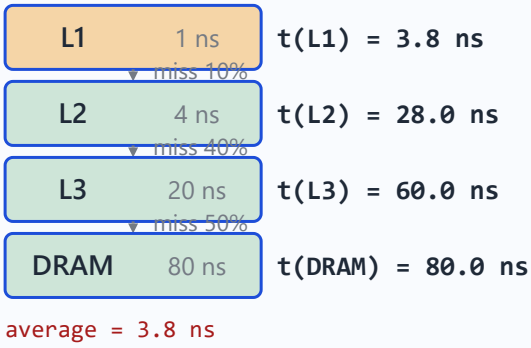


Figure 8.6: Access times and miss rates of the four levels, and the average times computed from the bottom up.

The result is striking. Although 10% of all accesses miss L1, the average is 3.8 ns. That is about 21 times faster than DRAM's 80 ns (80 / 3.8 ≈ 21.05). What makes the hierarchy worthwhile is the high hit rate at the upper levels.

The form of the formula also shows how much the miss rates matter. If L1's miss rate falls from 10% to 5%, the 28 ns below stays the same and the average becomes 1 + 0.05 × 28 = 2.4 ns. Reducing L1's own time from 1 ns to 0.5 ns saves only 0.5 ns. Misses dominate the average.

Example 8.2

L1 responds in 2 ns and misses on 5% of requests. L2 takes 10 ns and misses on 20% of the requests that reach it. DRAM takes 100 ns. What is the average access time? What would it be without L2?

1. **The L2 level.** $T_{L2} = 10 + 0.20 \times 100 = 30$ ns.
2. **The L1 level.** $T = 2 + 0.05 \times 30 = 2 + 1.5 = 3.5$ ns.
3. **Without L2.** The L1 misses go straight to DRAM: $2 + 0.05 \times 100 = 7$ ns. The second level has halved the average.

Common mistake

Expressing the second level's miss rate relative to all accesses. L2's 20% miss rate applies to the requests that *reach* L2; only 5% of all accesses reach L2. So the fraction of accesses that go down to DRAM is not 20% but $0.05 \times 0.20 = 1\%$. The formula performs this multiplication automatically.

8.1.7 The four questions of cache design

Every cache design has to answer four questions.

1. **When is data put into the cache?** Usually on a miss. The hardware also fetches the next lines ahead of time (*prefetch*) by predicting the access pattern.
2. **Where is it put?** This is the mapping question: direct-mapped, k-way set-associative, or fully associative.
3. **Which line is evicted when room is needed?** This is the replacement policy: approximations of LRU, random choice, or FIFO.
4. **Where does the evicted line go?** If the line has been modified (is "dirty"), it is written to the level below when evicted (*write-back*). The other option is to pass every write to the level below at once (*write-through*).

Besides these four questions there are three trade-offs. A big cache gives more hits but is slower and more expensive. A smarter replacement policy increases hits but complicates the hardware. In multicore processors, copies of the same line in different cores have to be kept up to date; this is called coherence and is provided by protocols such as MESI.

The same four questions are not specific to hardware caches. They also appear in the "caches" managed by the operating system: the page cache that holds files, virtual memory, and the TLB. The TLB described in these notes is a cache directly; virtual memory uses main memory as a cache for the disk.

8.2 The address space

For several programs to run safely in the same memory, two problems must be solved. The first is protection: a program must not be able to reach another program's memory. The second is relocation: a program must work correctly wherever in memory it is loaded. This section describes where the two problems come from and the simplest hardware solution.

8.2.1 Memory without abstraction

The earliest computers had no abstraction for memory. On mainframes before the 1960s and personal computers before the 1980s every program saw physical memory as it was. The instruction `MOV REGISTER1, 1000` read the word at physical address 1000. The address the program saw was the same as the address the memory chip saw.

In this arrangement two programs cannot be in memory at the same time. Both use the same physical addresses, and one overwrites the other's data. Memory can still be shared between the operating system and a single user program. Figure 8.7 shows three common layouts. In (a) the operating system is at the bottom of RAM. In (b) the operating system is at the top, in ROM. In (c) the device drivers are at the top in ROM and the rest of the operating system is at the bottom in RAM. In layouts (a) and (c) a bug in the user program can write into the region holding the operating system and crash it. Layout (b) was used in some embedded systems, and (c) in the first PCs running MS-DOS, where the BIOS drivers were in ROM.

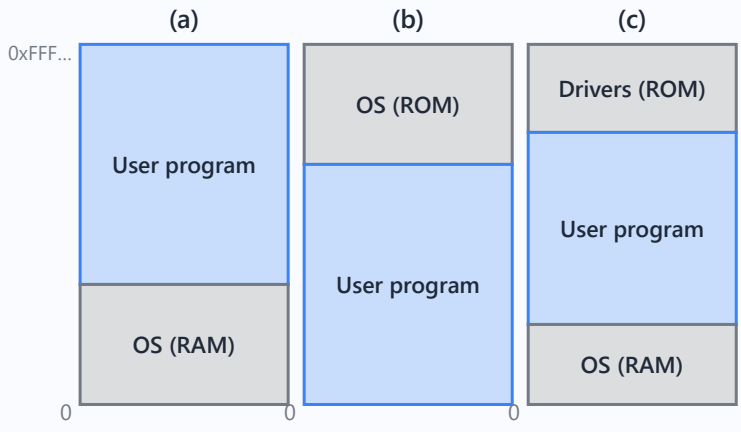


Figure 8.7: Three ways of sharing memory between the operating system and a single user program without abstraction.

This approach has not become history. Microcontrollers without a memory management unit still work this way today. Most of them nevertheless have a **memory protection unit** (MPU). An MPU does no address translation; it only checks access permissions for certain regions.

8.2.2 The relocation problem

The problem that arises when two programs are put in memory at once is shown clearly by Tanenbaum's example. A and B are two 16 KB programs, both compiled on the assumption that they start at address 0. A's first instruction is `JMP 24` and B's first instruction is `JMP 28`. The operating system loads A at address 0 and B right above it, at address 16384.

A runs without trouble. B crashes at its first instruction. `JMP 28` wants to jump to the `CMP` instruction at byte 28 of B. But 28 is an absolute address, and at physical address 28 there is A's `ADD` instruction. Instead of its own code, B jumps into the middle of A (Figure 8.8).

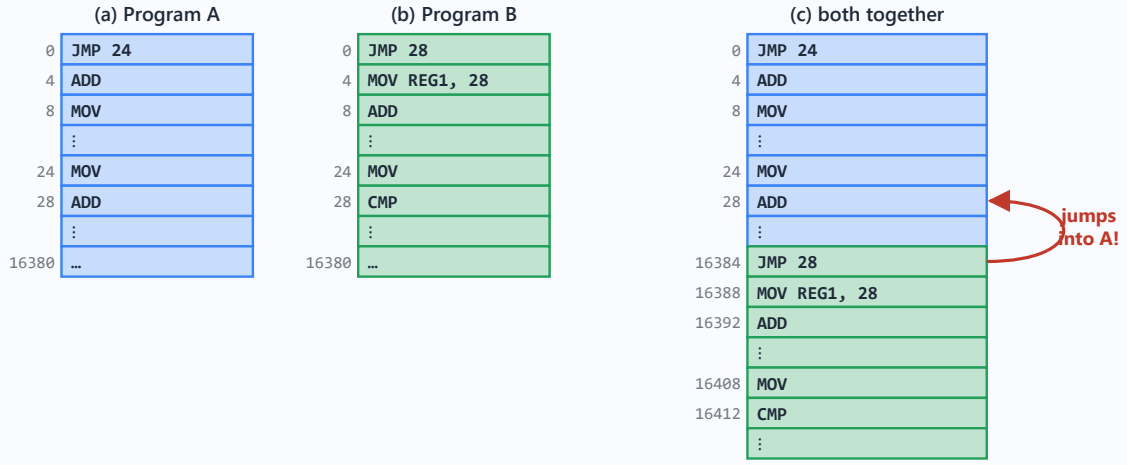


Figure 8.8: The relocation problem. (a) and (b) two programs compiled to start at 0, (c) both in memory together. B's `JMP 28` jumps into A.

This is called the **relocation** problem. The addresses in a program must be corrected according to where the program is loaded in memory. The first solution, found on the IBM 360, is *static relocation*: when the loader puts the program into memory, it adds the program's starting address to every address word. Since B is loaded at 16384, `JMP 28` becomes `JMP 16412` (Figure 8.9).

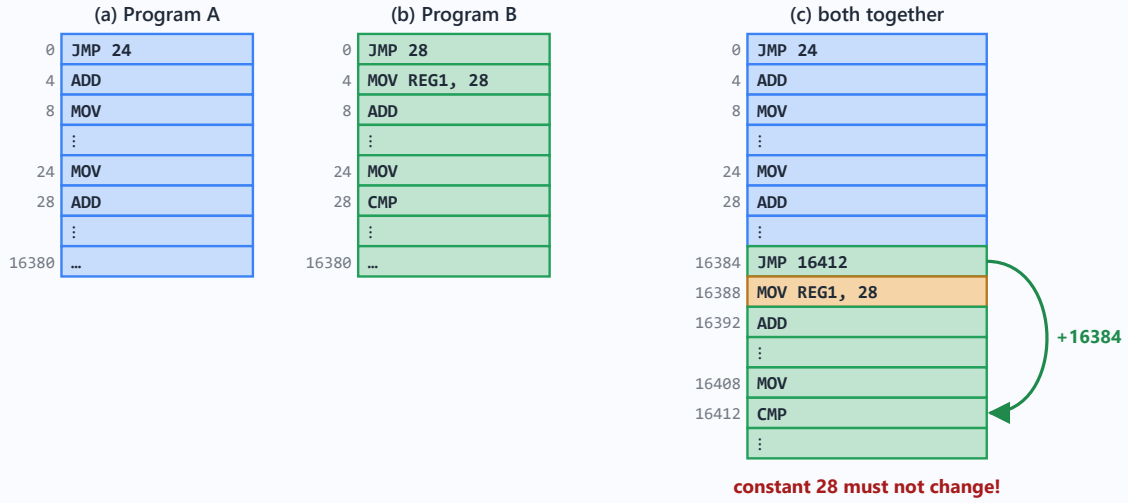


Figure 8.9: Static relocation. The loader adds 16384 to the jump address; but the 28 in `MOV REG1, 28` is a constant and must not change.

Static relocation has two weaknesses. First, loading becomes slower, because every word of the program has to be examined. The second is subtler: the loader cannot tell which words are addresses and which are ordinary numbers. B's second instruction `MOV REG1, 28` loads the number 28 into a register. This 28 is not an address; if 16384 is added to it, the program goes wrong. For the loader to decide correctly, the compiler has to produce extra relocation information indicating which words of the program are addresses.

8.2.3 Base and limit registers

The abstraction that solves protection and relocation together is the **address space**.

> **Definition 8.2**
>
> An **address space** is the set of addresses a process can use in memory. Each process has an address space of its own, independent of those of other processes.

The simplest hardware for implementing an address space is two registers. The **base register** holds the process's starting address in memory, and the **limit register** holds the length of the region. The program is loaded as it is into a contiguous area of memory. In the example of Figure 8.10 the base is 300040 and the limit is 120900. The process can access addresses from 300040 to 300040 + 120900 − 1 = 420939.

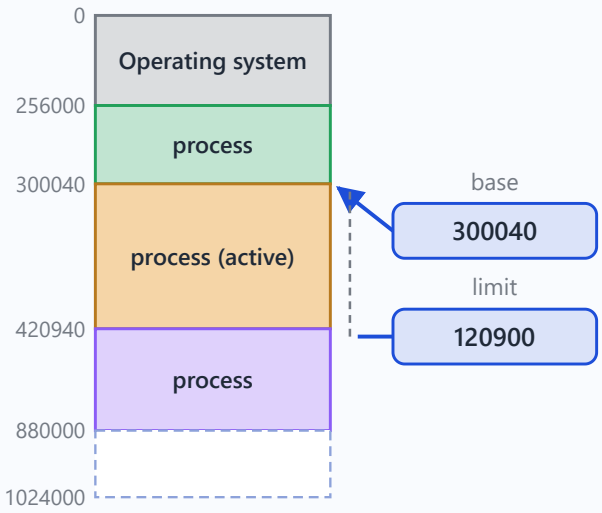


Figure 8.10: Base and limit registers. The running process can access addresses from 300040 to 420939.

These two registers can be loaded only by privileged instructions, in kernel mode. A user program cannot change its own base or limit; if it could, protection would be meaningless. When the operating system switches context, it puts the new process's base and limit values into these registers.

History offers two extreme examples of this arrangement. The CDC 6600 supercomputer worked exactly this way. The segment registers of the Intel 8088 served only as a base; since there was no limit register, there was relocation but no protection.

8.2.4 Hardware address protection

Base and limit registers provide protection with two comparisons on every memory access. The address produced by the CPU is first compared with the base: if the address is smaller than the base, a trap occurs. Then it is compared with base + limit: if the address is not smaller than this value, a trap occurs again. If both conditions hold, the access goes to memory (Figure 8.11). The comparisons are made by special hardware working in parallel and do not slow the access down.

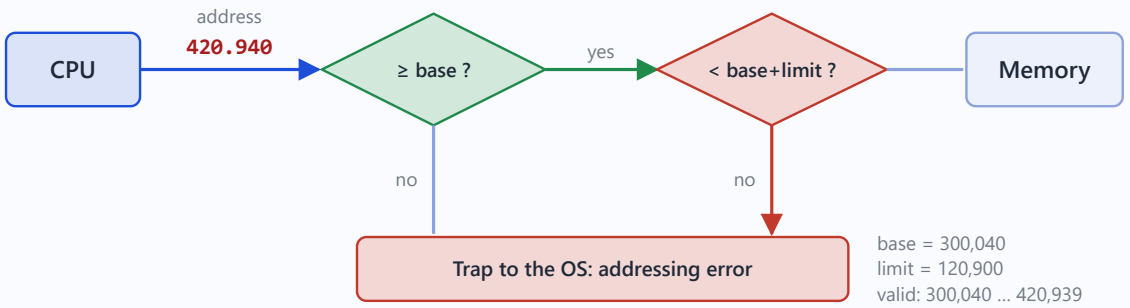


Figure 8.11: Hardware address protection. Since 420940 = base + limit, the second comparison fails and a trap is raised to the operating system.

Address	≥ base?	< base + limit?	Result
300040	yes	yes	access (first valid address)
356000	yes	yes	access
420939	yes	yes	access (last valid address)
420940	yes	no	trap
299999	no	—	trap; another process's area is protected

Table 8.4: Checking five addresses with base = 300040 and limit = 120900 (valid range 300040 … 420939).

When the trap occurs, control passes to the kernel. The kernel treats it as an addressing error and usually terminates the process; on Unix systems the process receives the signal `SIGSEGV`. The check is made only for accesses in user mode. In kernel mode there is no check, because the operating system must be able to reach all of memory: loading a process into memory and reading a system-call argument from a user buffer both require it.

Common mistake

Counting the address base + limit as valid. The limit is the length of the region; the valid addresses are base, base + 1, …, base + limit − 1. The last byte of a 120900-byte region is at 300040 + 120899 = 420939. 420940 is already outside the region.

8.2.5 Address binding times

On its way from source code to a process in memory, a program passes through several stages (Figure 8.12). The compiler turns `main.c` into an object file (`main.o`). The linker combines this file with other object files and static libraries to produce the executable (`a.out`). The loader brings this file into memory, adds the dynamic libraries, and the process starts to run.

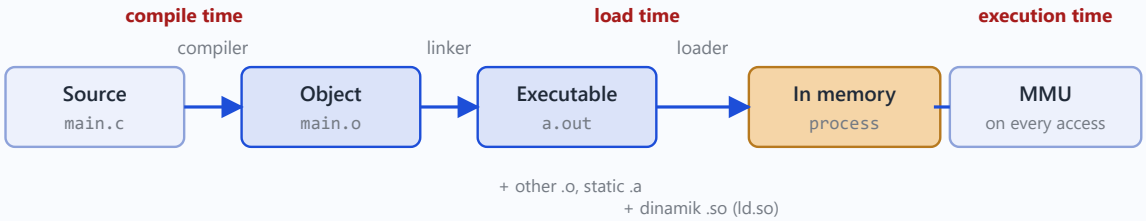


Figure 8.12: A program passing through the compiler, linker, and loader and being placed in memory. Address binding can be done at any one of these three stages.

Addresses change form along this journey. In source code addresses are symbolic: the programmer writes a variable called `count`. The compiler binds this symbol to a relative address: "14 bytes from the start of this module." The linker or loader converts the relative address into an absolute one, for example 74014. When a symbol is bound to an absolute address determines whether the program can be moved in memory. Table 8.5 compares the three options.

Binding time	Code produced	If the location changes	Hardware
Compile time	Absolute code (e.g. MS-DOS .COM)	recompile	not needed
Load time	Relocatable code; the loader fixes the addresses	reload	not needed
Execution time	Logical addresses; can be moved while the process runs	nothing	MMU (base–limit, paging)

Table 8.5: Address binding times.

In **compile-time** binding the compiler knows in advance where in memory the program will be loaded and produces absolute code. MS-DOS `.COM` files were like this. If the starting address changes, the program has to be recompiled. In **load-time** binding the compiler produces relocatable code; the absolute addresses are fixed at load time by static relocation. If the starting address changes, the program is reloaded. In **execution-time** binding, address translation is done on every access while the program runs. A process can be moved from one memory region to another while it is running. Special hardware is needed for this. All of today's general-purpose operating systems use execution-time binding.

8.2.6 Logical and physical addresses, the MMU

Execution-time binding separates two kinds of addresses.

Definition 8.3

A **logical address**, also called a **virtual address**, is the address the CPU produces. A **physical address** is the address the memory unit sees, the one loaded into the memory address register (MAR). The set of all logical addresses a program produces is the **logical address space**, and the set of the corresponding physical addresses is the **physical address space**.

In compile-time and load-time binding the two addresses are the same. In execution-time binding they differ, and the conversion between them is done at run time by a piece of hardware: the **memory management unit** (MMU). The program sees and works with logical addresses only; it never sees physical addresses. A C program that prints the value of a pointer therefore always prints a logical address.

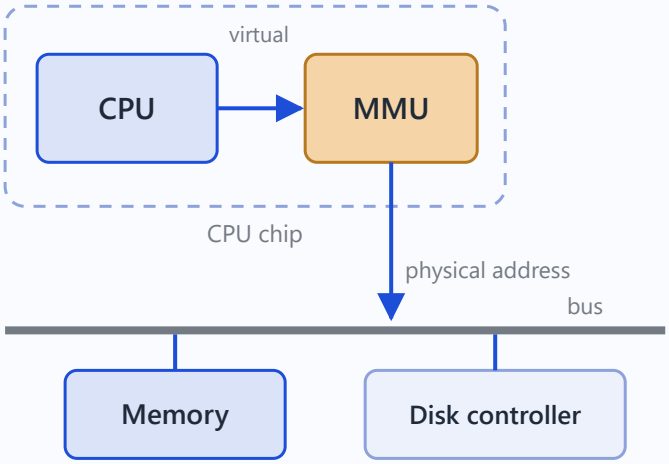


Figure 8.13: The MMU is inside the processor chip. The CPU gives the virtual address to the MMU; the MMU puts the physical address on the bus.

Figure 8.13 shows where the MMU sits. In older systems the MMU was a separate chip. In today's processors the MMU, with its TLBs and page-table walker, is inside every core. Every address that goes out on the memory bus is already a physical address.

8.2.7 Dynamic relocation with the MMU

In its execution-time form, the base–limit arrangement has the MMU do two jobs. First it compares the logical address with the limit; if the address is not smaller than the limit, it raises a trap. Then it adds the base value to the address and obtains the physical address. Silberschatz calls this register the **relocation register**; it is the same thing as Tanenbaum's base register.

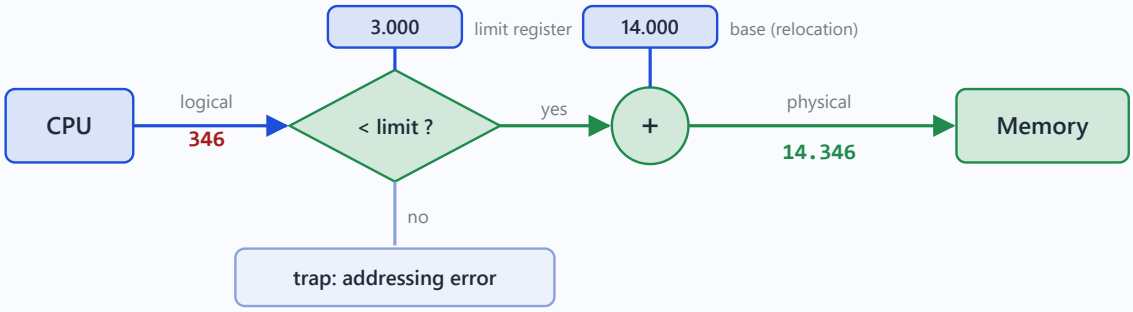


Figure 8.14: Dynamic relocation with the MMU. Logical 346 is less than the limit 3000; it is added to the base and physical 14346 results.

Example 8.3

Let the limit be 3000 and the base 14000. Where do logical addresses 346, 2999, and 3500 go?

1. **346.** 346 < 3000, valid. The physical address is 346 + 14000 = 14346.
2. **2999.** 2999 < 3000, the last valid logical address. Physical 2999 + 14000 = 16999.
3. **3500.** 3500 ≥ 3000: a trap occurs before any addition. Because the comparison is made on the logical address, the result is the same whatever the base value is.

The program sees itself as occupying 0 through 2999. If the operating system moves the process somewhere else, it changes only the base register; not a single address in the program changes.

This arrangement also rescues program B in the relocation example. For B let base = 16384 and limit = 16384. The instruction `JMP 28` stays in memory as it is. When the instruction executes, the MMU compares 28 with the limit, finds it valid, and produces the physical address 28 + 16384 = 16412 (Figure 8.15). Neither static relocation nor extra relocation information is needed; the constant in `MOV REG1, 28` never changes, because only addresses that go to memory are translated.

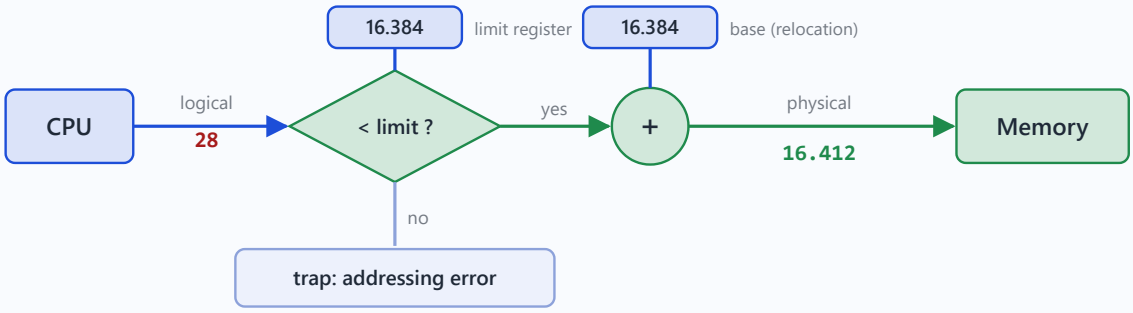


Figure 8.15: Program B runs with base = limit = 16384. `JMP 28` stays unchanged; the MMU produces 16412.

This has a price: every memory access needs an addition and a comparison. The comparison is fast. The addition, because of carry propagation, remains slow relative to an access unless special circuits are used. This price is the first sign of a problem that will grow with paging: address translation lies on the path of every access, and it has to be very fast.

8.3 Linking and loading

Loading a program into memory is not as simple as "copy the whole file into memory." The program refers to libraries; will these libraries be loaded together with the program or separately? Should a large piece of code that may never be used be loaded up front? This section describes today's answers to these questions.

8.3.1 Dynamic loading

Dynamic loading means that a routine is not brought into memory until it is called. The main part of the program is loaded and starts to run. When a routine is needed, the program has it loaded itself. Code that is never called is never loaded. This is useful for large but rarely executed parts such as error-handling code.

Dynamic loading needs no special hardware; the program is designed accordingly. The operating system only provides the necessary library functions. On POSIX systems these are `dlopen`, `dlsym`, and `dlclose`; their Windows counterparts are `LoadLibrary` and `GetProcAddress`. The program below loads the math library at run time and calls the `sqrt` function in it.

```
#include <dlfcn.h>
#include <stdio.h>

int main(void) {
  void *h = dlopen("libm.so.6", RTLD_LAZY);
  if (!h) { puts(dlerror()); return 1; }
  double (*root)(double) =
      (double (*)(double)) dlsym(h, "sqrt");
  printf("%.3f\n", root(2.0));
  dlclose(h);
  return 0;
}
```

The program runs line by line as follows.

1. When the program starts, the address space contains only the code of `main`, the C library (libc), the heap, and the stack. The math library (libm) is not loaded.
2. The call `dlopen("libm.so.6", RTLD_LAZY)` runs the dynamic linker (`ld.so`). The dynamic linker finds the file and maps it into the process's address space with `mmap` (Figure 8.16). Loading happens at the program's request, now. The returned `h` is a handle that represents the library; if

the file is not found, `NULL` is returned and `dlerror()` reports the reason.

3. `dlsym(h, "sqrt")` looks up the name `sqrt` in the library's symbol table and returns its address in this process's address space. The returned `void *` value is cast to a function pointer that takes a `double` and returns a `double`.
4. The call `root(2.0)` is made through the function pointer, and `1.414` is printed ($\sqrt{2}$ in the %.3f format).
5. `dlclose(h)` decrements the library's reference count. When the count reaches 0, the library is removed from the address space.

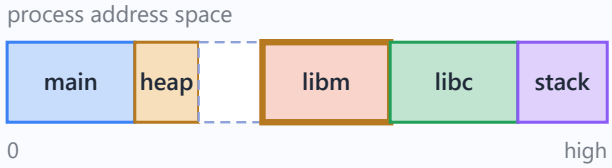


Figure 8.16: The address space after `dlopen`: libm is mapped between main and libc.

With older glibc versions the option `-ldl` was needed when compiling this program. Since glibc 2.34 these functions are inside libc and no extra option is needed.

8.3.2 Static and dynamic linking

When a program calls a library function, the code of that function has to be somewhere. There are two ways. In **static linking** the linker (`ld`) copies the library routines into the executable after compilation; for example `ld *.o -lc -lm`. In **dynamic linking** only a small *stub* is left in the file. The real address of the routine is resolved at run time. Table 8.6 compares the two approaches.

	Static linking	Dynamic linking
Who, and when?	The linker (ld), after compilation	The dynamic linker (ld.so), at load time or at run time
File	.a, .lib	.so, .dll, .dylib
Memory	A separate copy in every program	Code pages once in memory, shared
Updating	Must relink	Replacing the library is enough (versioning: soname)
Risk	Large file; security patches are delayed	Version mismatch (“DLL hell”)

Table 8.6: Static and dynamic linking.

The biggest gain of dynamic linking is in memory. Hundreds of processes use the same C library. With static linking every process would contain a separate copy. With dynamic linking the library's code pages exist once in physical memory and are mapped into every process's address space (Figure 8.17). A library of this kind is called a **shared library**.

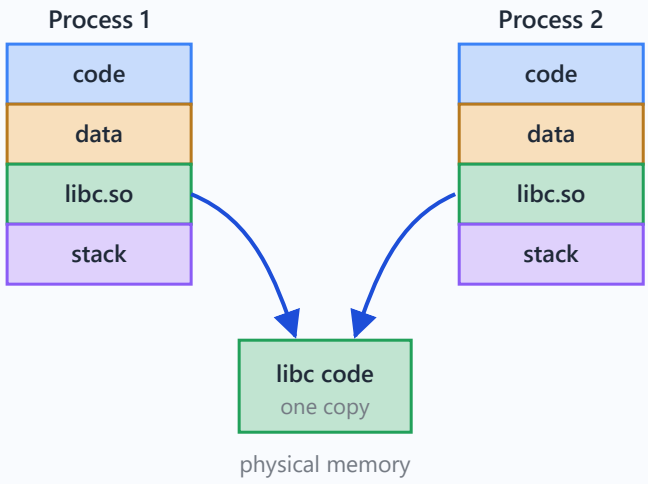


Figure 8.17: Two processes map the same libc code pages into their own address spaces; there is a single copy in physical memory.

Updating also becomes easier. When a security hole is closed in the C library, it is enough to install the new library; all programs that use it get the patched version the next time they run. Statically linked programs carry the hole until they are relinked. To keep incompatible versions from being mixed, libraries have a version name (soname); the 6 in `libm.so.6`, for example, is such a version. The confusion that results when this precaution is not taken is known in the Windows world as “DLL hell.”

8.3.3 Lazy binding: the PLT and GOT

Textbooks usually describe dynamic linking like this: when the stub is called for the first time, it finds the routine's address and replaces itself with that address. This account is conceptually right, but on systems that use ELF files, like Linux, what changes is not the code. The real mechanism rests on two tables.

PLT (procedure linkage table)
: A table of small jump stubs in the executable. The program does not call `printf` directly but calls the stub `printf@plt`. This is the “stub” of the textbooks.

GOT (global offset table)
: A per-process, writable table of addresses. The PLT stub jumps indirectly to the address the corresponding GOT entry points to.

On the first call the GOT entry points not to the real `printf` but to the dynamic linker's resolver. The resolver looks the symbol up in the loaded libraries, writes the address it finds into the GOT entry, and jumps to `printf`. Later calls follow the path PLT → GOT → `printf`; only one indirect jump is added. This arrangement is called **lazy binding**, because a symbol is resolved only when it is first used. Table 8.7 lists the steps.

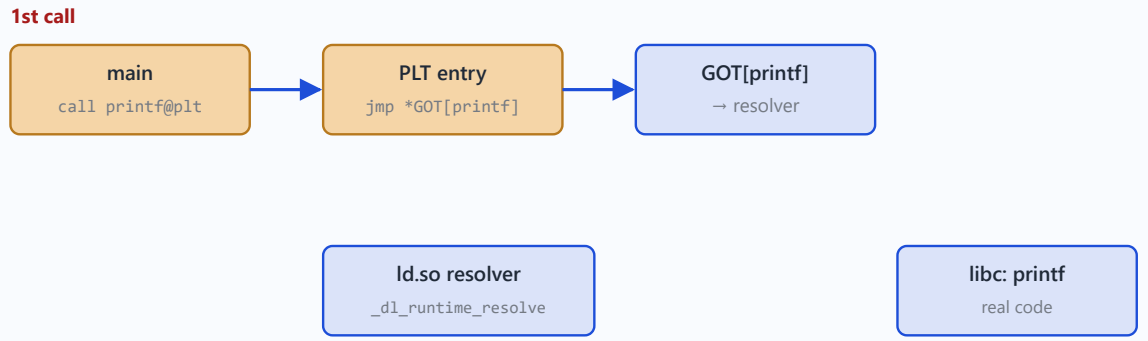


Figure 8.18: The first call under lazy binding: the PLT goes to the resolver through the address in the GOT.

Step	What happens?	GOT[printf]
1	`main` goes to the PLT stub with `call printf@plt`.	resolver
2	The PLT jumps with `jmp *GOT[printf]` to the address in the GOT; this address is the resolver.	resolver
3	The `ld.so` resolver looks up the symbol and writes the address it finds into the GOT entry.	printf
4	The resolver jumps to `printf`. The code pages did not change; only the process-specific GOT page did.	printf
5	Later calls: PLT → GOT → `printf`, a single indirect jump.	printf

Table 8.7: The first and later calls of `printf`.

That what changes is the GOT and not the code is critical for sharing. Since the code pages are never written, they remain shareable among all processes. Only each process's own GOT page is different. This work is done not by the operating system but by the dynamic linker running in user space.

Lazy binding has a weakness from the security point of view: because the GOT is writable, an attacker can use a buffer overflow to change an address there and take over control. That is why many distributions compile programs with the `-z now` option. In this “full RELRO” arrangement all symbols are resolved at load time and the GOT is then made read-only.

Common mistake

Thinking that after the first call the target of the `call printf@plt` instruction changes, or that the body of libc's `printf` is rewritten. Neither happens: the code pages are read-only and shared among processes. The only place that changes is the process-specific GOT entry.

8.3.4 Building static and shared libraries

Two kinds of library can be built from a C source file (`func.c`). A static library is built and used with these commands:

```
$ gcc -c func.c -o func.o
$ ar rcs libfunc.a func.o
$ gcc main.c -o main -static -L. -lfunc
$ ./main
```

The `ar` tool gathers object files into a single archive (`libfunc.a`). The linker copies from this archive only the objects the program actually uses. The `-static` option asks for the static version of the C library (`libc.a`) to be used as well; the result is a file that depends on no shared library.

For a shared library the steps are as follows:

```
$ gcc -fPIC -c func.c -o func.o
$ gcc -shared -o libfunc.so func.o
$ gcc main.c -o main -L. -lfunc
$ export LD_LIBRARY_PATH=$(pwd)
$ ./main
$ ldd ./main   (libfunc.so => ./libfunc.so)
```

The key option here is `-fPIC`: **position-independent code**. Code compiled this way uses no absolute addresses. Jumps are made relative to the program counter, and global data is reached through the GOT. This allows the same physical page to be mapped at a different virtual address in each process. Address space layout randomization (ASLR) also rests on this. The `ldd` command shows which shared libraries the program will take from which files.

Figure 8.19: With static linking the library code is copied into every program; with a shared library a single copy is mapped into the address spaces of two processes.

The `LD_LIBRARY_PATH` variable is handy for experiments but is not a permanent solution. The permanent solution is either to embed the library's search path in the executable (`-Wl,-rpath,'$ORIGIN'`) or to install the library in a system directory and run `ldconfig`. Today's distributions also compile the programs themselves as position-independent executables (PIE). In this way the program's own code is also loaded at a different address on each run, thanks to ASLR.

8.4 Swapping

What happens if the total needs of the processes in memory exceed physical memory? One solution is to move the processes that are not running at the moment to disk temporarily. This idea is called **swapping**.

8.4.1 Standard swapping

In standard swapping an entire process is written temporarily to a **backing store**, usually a fast disk. When the process is about to run again, it is brought back into memory (Figure 8.20). Thanks to swapping, the total memory needs of the processes can exceed physical memory and more processes can be run at the same time; that is, the degree of multiprogramming rises.

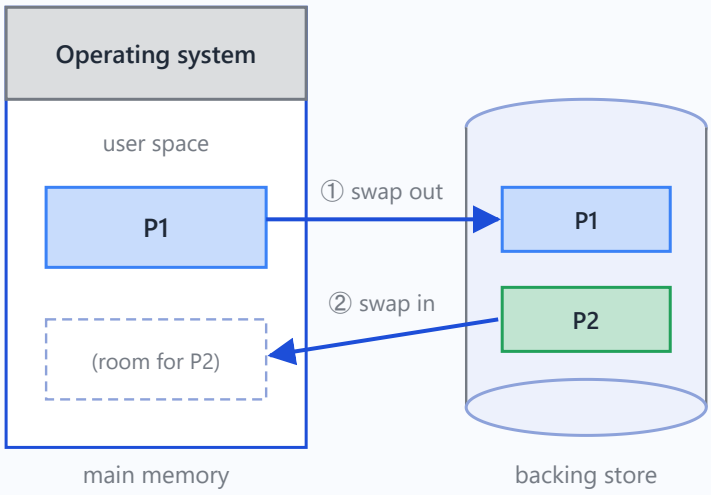


Figure 8.20: Standard swapping: the whole of process P1 is written to the backing store and P2 is read into memory.

A process returning from a swap may be placed not at its earlier location in memory but in some other place that happens to be free. That is why swapping requires execution-time binding. A process bound at compile or load time can return only to exactly its old place.

The cost of standard swapping is the transfer time. In Silberschatz's example a 100 MB process is written to a disk with a transfer rate of 50 MB/s. Writing it out takes 100 / 50 = 2 s and reading it back takes another 2 s; the round trip is 4 seconds. During this time the swapped process cannot run.

There is one more constraint in choosing the process to swap: a process with a pending I/O operation should not be swapped out. If the process is waiting for data from a device, the device will write the data into the process's buffer by direct memory access (DMA). If the process is swapped out and

another is loaded in its place, the DMA transfer writes to the old physical address, that is, now into another process's memory. The remedy is either not to swap such a process or always to perform I/O through kernel buffers.

Example 8.4

A 200 MB process is swapped to a disk with a transfer rate of 50 MB/s; latencies are ignored. How long does the round trip take?

1. **Out.** 200 / 50 = 4 s.
2. **In.** Again 4 s.
3. **Total.** 8 s. Considering that a time slice is tens of milliseconds, it is clear why moving a whole process is not acceptable today.

8.4.2 Page swapping and today's systems

Today the sizes of processes are measured in gigabytes. A 1 GiB process takes about 2 × 1.07 / 3 ≈ 0.72 s for a round trip even on an NVMe SSD with a 3 GB/s read-write rate. That is why Linux, Windows, and macOS swap processes not as wholes but page by page (Figure 8.21). Only pages that have not been used for a long time, the “cold” pages, go to disk. A 4 KiB page comes back in tens of microseconds.

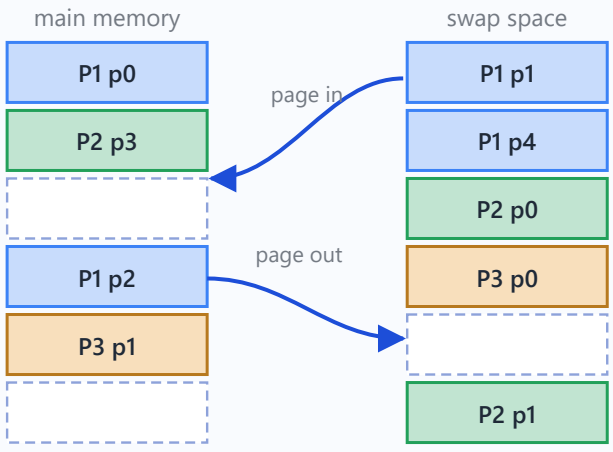


Figure 8.21: Swapping with paging: not whole processes but individual pages are moved between memory and the swap area.

Today's systems use one more intermediate step before going to disk: compression. Cold pages are first compressed in memory to make them smaller. In Linux zswap and zram, in Windows memory compression, and in macOS the compressor do this. Decompressing a compressed page is much faster than reading it from disk.

On mobile systems the situation is different. The write endurance of flash memory is limited, and every write costs energy. That is why phones have either no traditional swap or a very limited one. Android compresses with zram; if memory is still short, `lmkd` kills background applications. iOS also uses compression and kills applications when necessary (jetsam). From the user's point of view this shows up as an application restarting from scratch when one returns to it in the background.

8.5 Contiguous allocation

In a system that works with base and limit registers, each process occupies a single unbroken region of memory. This section describes how free regions are chosen in such a system and why, over time, they break up into small, useless pieces.

8.5.1 Contiguous memory allocation

Memory is first divided into two regions: one for the operating system and the other for user processes. The operating system can be placed at the low or the high end of memory. The Linux and Windows kernels put it at high addresses. In the user region each process occupies a single contiguous region, and protection is provided by the base–limit registers.

> **Definition 8.4**
>
> **Contiguous allocation** is giving each process a memory region made up of a single run of consecutive addresses.

There are two ways of dividing the user region among processes. In **fixed partitioning** memory is divided in advance into a certain number of partitions of certain sizes; the MFT version of IBM OS/360 worked this way. One process goes into each partition, so the number of partitions determines the number of processes that can be in memory at once, that is, the degree of multiprogramming. If a small process goes into a large partition, the rest of the partition is wasted. In **variable partitioning** (OS/360 MVT), each process is given exactly the space it needs. The free blocks left between processes are called **holes**.

There is one more problem: processes grow. A program asks for space from the heap with `malloc`, and function calls deepen the stack. If another process sits right above a contiguous region, growth is impossible. That is why processes are left some room to grow from the start (Figure 8.22). In (a) only the data segment grows. In (b) the stack grows from top to bottom and the data from bottom to top, and the spare room lies between the two. When the process is swapped out and brought back, only the part it uses is moved, not the free room.

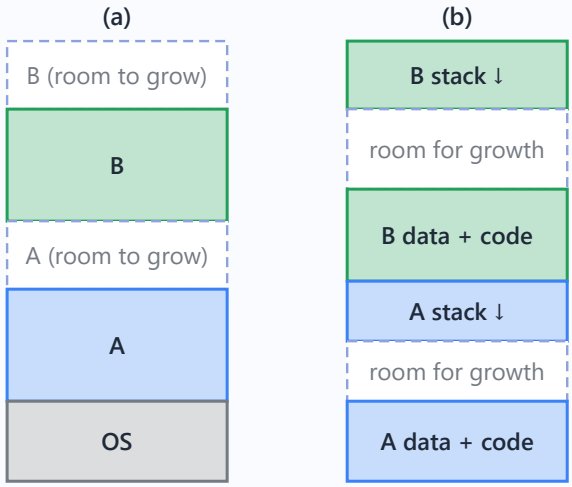


Figure 8.22: Leaving room for growing processes: (a) the data segment grows, (b) the stack grows down and the data grows up.

8.5.2 Variable partitioning

In variable partitioning the operating system keeps a table or list: which regions belong to which processes and which regions are holes. When a process arrives, a hole large enough to take it is found. The process is placed at one end of the hole, and the remainder stays as a smaller hole. When the process ends, its space returns to the hole list and is merged at once with neighboring holes.

Let the first 128 K of a 1024 K memory belong to the operating system, and let processes be placed in the first hole that fits. Table 8.8 follows the events in order.

Event	Placement	Holes afterwards
start	—	896 K @ 128
A (320 K) arrives	128–448	576 K @ 448
B (224 K) arrives	448–672	352 K @ 672
C (288 K) arrives	672–960	64 K @ 960
A terminates	—	320 K @ 128, 64 K @ 960
D (256 K) arrives	128–384	64 K @ 384, 64 K @ 960
B terminates	—	288 K @ 384, 64 K @ 960
E (192 K) arrives	384–576	96 K @ 576, 64 K @ 960

Table 8.8: A sequence of events under variable partitioning (1024 K memory, OS at 0–128 K). Holes are written as “size @ start”; retraced with Python.

D goes into the 320 K hole that A left, and a small 64 K hole remains. When B terminates, its 224 K area merges with the 64 K hole to its left and forms a single 288 K hole. If adjacent holes were not merged, no room could be found for a 288 K request. After E goes into this hole, the total free space is

96 + 64 = 160 K, but it is split into two pieces. If a 100 K process arrived, it would not fit, even though the total space is enough.

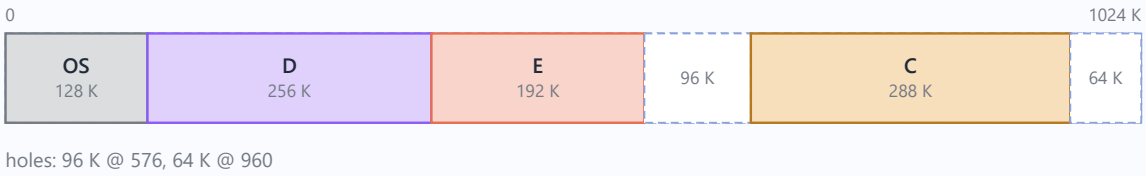


Figure 8.23: The end of the event sequence: D, E, and C are in memory; the two holes of 96 K and 64 K are separate.

8.5.3 Placement algorithms

If several holes could take an incoming request, which should be chosen? The classic answers to this question are as follows.

First fit
: The list is scanned from the start and the search stops at the first hole that is large enough. It is fast. Over time, small holes accumulate at the head of the list.

Next fit
: A variant of first fit; the search continues from where the previous search stopped, not from the start. In Bays's 1977 simulations it gives slightly worse results than first fit.

Best fit
: The whole list is scanned and the hole closest in size to the request is chosen. It is slow unless the list is kept sorted by size. It leaves behind holes too small to be of any use.

Worst fit
: The largest hole is chosen; the aim is for the remainder to stay large enough to serve another request. It does not give good results in simulations.

Quick fit
: Separate lists are kept for frequently requested sizes (4 K, 8 K, 12 K ...). Finding a suitable hole is very fast; but finding the neighbors of a freed block and merging them is expensive.

All the algorithms can be sped up by keeping separate lists for processes and holes: the search is then made in the hole list only. Today's `malloc` implementations (jemalloc, tcmalloc) use a generalized form of quick fit, size classes. The outcome of the simulations reported by Silberschatz and Knuth is clear. In terms of both time and memory utilization, first fit and best fit are better than worst fit. First fit is usually faster.

8.5.4 Code trace: best fit

The function below performs a best-fit search in an array holding hole sizes. It shrinks the hole it finds by the size of the request and returns the hole's index; if there is no room it returns −1.

```c
int best_fit(int hole[], int n, int req) {
  int best = -1;
  for (int i = 0; i < n; i++)
    if (hole[i] >= req &&
        (best == -1 || hole[i] < hole[best]))
      best = i;
  if (best != -1) hole[best] -= req;
  return best;
}
/* hole = 120, 450, 230, 300, 180 (K) */
/* best_fit(hole, 5, 210); best_fit(hole, 5, 160); */
```

The variable `best` is the index of the best candidate found so far; −1 means “no candidate yet.” For each hole the loop checks two conditions. The first is that the hole is large enough for the request (`hole[i] >= req`). The second is that either there is no candidate yet or this hole is smaller than the current candidate. If both hold, the candidate changes. When the loop ends, the chosen hole is shrunk by the size of the request. The function walks the whole list on every call, so it spends O(n) time for n holes.

Call	i	hole[i]	Decision	best
req = 210	0	120	120 < 210, does not fit	−1
	1	450	fits, first candidate	1
	2	230	fits, smaller than 450	2
	3	300	fits but larger than 230	2
	4	180	does not fit	2
Returns 2; hole[2] = 230 − 210 = 20. Holes: 120, 450, 20, 300, 180.				
req = 160	0	120	does not fit	−1
	1	450	fits, first candidate	1
	2	20	does not fit	1
	3	300	fits, smaller than 450	3
	4	180	fits, smaller than 300	4
Returns 4; hole[4] = 180 − 160 = 20. Holes: 120, 450, 20, 300, 20.				

Table 8.9: Two calls of the `best_fit` function (holes 120, 450, 230, 300, 180 K). The values were verified with Python.

Figure 8.24: The hole list after the two calls. Best fit twice left a useless 20 K hole.

The trace shows the typical behavior of best fit: because it picks the hole closest to the request each time, very small remainders are left behind. Had the same two requests been served by first fit, both would have gone into the 450 hole: first 240, then 80 K would have remained. With worst fit, 210 K would again go into 450 (240 remains), and 160 K would go into 300, the largest hole at that moment (140 remains). Which remainders will be useful in the future depends on later requests.

8.5.5 Four algorithms on the same request sequence

Silberschatz's classic exercise shows the difference between the algorithms in a single example.

Example 8.5

The holes, in address order, are 100, 500, 200, 300, and 600 K. The requests, in order, are 212, 417, 112, and 426 K. Where do first fit, next fit, best fit, and worst fit put each request?

1. **212 K.** First fit and next fit put it into the first sufficient hole, 500 (288 remains). Best fit puts it into the smallest of those that fit (500, 300, 600), namely 300 (88 remains). Worst fit puts it into 600 (388 remains).
2. **417 K.** In first fit 288 is not enough, so it goes into 600 (183 remains). Next fit continues the search after 500; 200 and 300 are not enough, and it goes into 600 (183 remains). In best fit the holes that fit are 500 and 600; it goes into 500 (83 remains). In worst fit the largest hole at that moment is 500 (600 is now 388); it goes into 500 (83 remains).
3. **112 K.** First fit searches from the start: 100 is not enough, 288 is (176 remains). Next fit stops at 183, the remainder of the 600 hole where it last stopped (71 remains). Best fit puts it into 200, the smallest of those that fit (88 remains). Worst fit puts it into 388, the largest hole (276 remains).
4. **426 K.** In first fit the largest hole is 300, in next fit 300, and in worst fit 300: there is no room and the request waits. Only in best fit is the 600 hole still whole; the request goes there (174 remains).

In this sequence only best fit can place all four requests. In the others the total free space is more than 426 K, but it is not a single piece.

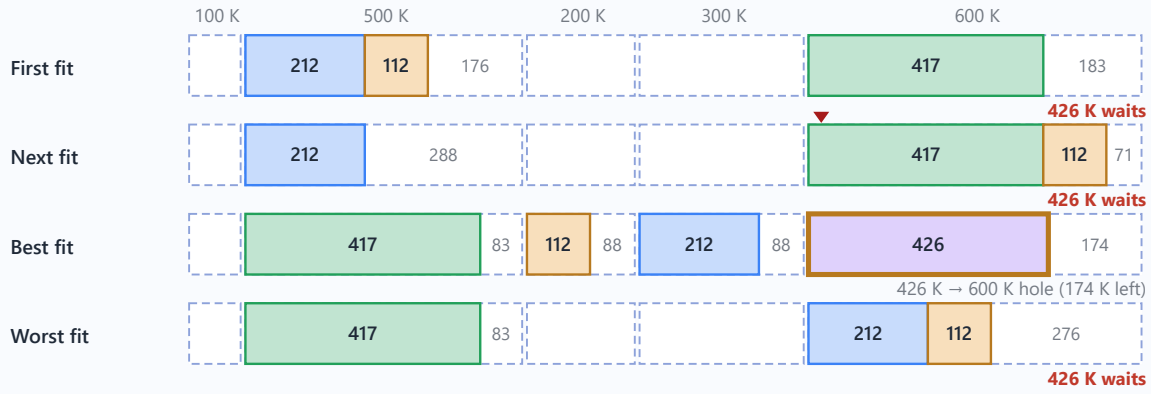


Figure 8.25: The state of the four algorithms after the same request sequence. Only best fit could place 426 K.

This result does not mean that best fit is always superior. The example is a sequence in which keeping the large hole whole pays off. With other sequences the outcome changes. Take, for instance, holes of 150, 400, 80, 250, and 600 K and requests of 230, 120, 380, and 90 K. This time worst fit cannot place the 380 K request, while first fit and best fit satisfy all four requests.

Common mistake

In worst fit, still counting the hole chosen by the first request as the largest hole for later requests. Holes shrink after every placement. After 212 K the 600 hole drops to 388; for 417 K the largest hole at that moment is now 500.

8.5.6 Internal and external fragmentation

The problem that arises in variable partitioning and that we saw in the earlier examples is called **external fragmentation**.

Definition 8.5

External fragmentation is the situation in which there is enough total free memory to satisfy a request but that memory is not contiguous. **Internal fragmentation**, on the other hand, is the situation in which the block given to a process is larger than requested and the difference is wasted inside the block.

The extent of external fragmentation is summarized by Knuth's **50-percent rule**. In a system running first fit, with N allocated blocks, on average about 0.5 N blocks are lost to fragmentation. The share of lost blocks is 0.5 N / (N + 0.5 N) = 1/3; that is, one third of memory can become unusable.

Internal fragmentation appears in every arrangement that works with fixed-size blocks: in fixed partitioning, in paging, and in the buddy system described below. Tanenbaum's example shows an interesting cause as well (Figure 8.26). If there is a hole of 18,464 bytes and a process of 18,462 bytes arrives, a hole of 2 bytes would be left. The cost of tracking these 2 bytes as a separate hole in the list is greater than the hole itself. So the process takes all 18,464 bytes, and 2 bytes are wasted inside the block.

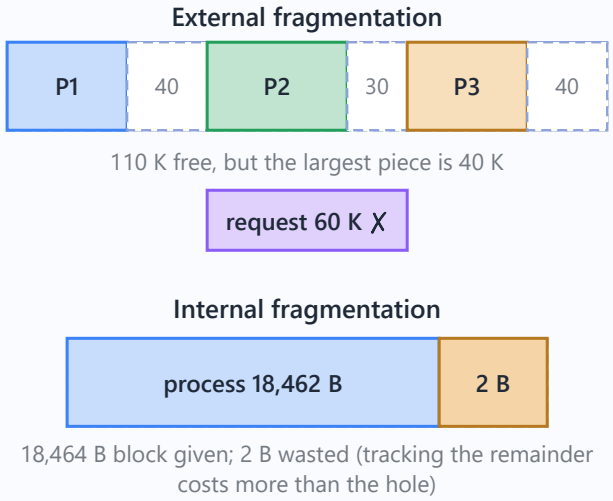


Figure 8.26: Top: external fragmentation: 110 K is free, but because the largest piece is 40 K, a 60 K request does not fit. Bottom: internal fragmentation: 2 bytes of the given block are unused.

External fragmentation has two remedies. The first is *compaction*: processes are shifted and the holes are gathered into a single block. The second is more radical: remove the requirement that a process's address space be contiguous in physical memory. Paging and segmentation follow this second route.

A well-known source of internal fragmentation is the **buddy system**, which is also used in Linux for allocating physical frames. In this scheme blocks are always a power of two in size. A request is rounded up to the smallest power of two that satisfies it. If there is no free block of the right size, a larger block is split in two; the two halves are called each other's "buddies." When a block is freed and its buddy is also free, the two merge again. The buddy's address is found with a single XOR: address $\oplus$ size.

Example 8.6

In a buddy system that starts with one free block of 1024 K, let A = 100 K, B = 240 K, C = 64 K, and D = 256 K be requested in turn. Where do the blocks go and how much internal fragmentation results?

1. **A = 100 K.** Rounded up to 128 K. 1024 is split (512 + 512), the left 512 is split (256 + 256), the left 256 is split (128 + 128). A gets the 128 K block at address 0. Internal fragmentation 28 K. Free: 128 K @ 128, 256 K @ 256, 512 K @ 512.

2. **B = 240 K.** Rounded up to 256 K and goes into the free block at 256. Internal fragmentation 16 K.
3. **C = 64 K.** Exactly 64 K. The free block at 128 is split (64 + 64); C takes the half at 128. Free: 64 K @ 192, 512 K @ 512.
4. **D = 256 K.** There is no free 256 K block; the block at 512 is split and D takes the half at 512. Free: 64 K @ 192, 256 K @ 768.
5. **Freeing.** When A is freed, its buddy ($0 \oplus 128 = 128$) is inside C's region, so there is no merging. When C is freed, its buddy at 192 is free: 128 K @ 128 results; its buddy, the block at 0 that was A's, is free too: 256 K @ 0 results. When B and D are also freed, the blocks merge up to 512 and then to 1024.

Total internal fragmentation is 28 + 16 = 44 K. (Traced step by step with Python.)

8.5.7 Compaction

Definition 8.6

Compaction is shifting the processes in memory toward one end and gathering all the holes into a single large block.

Figure 8.27 shows the initial state. The operating system is at 0–128 K. P1 starts at 128 (160 K), P2 at 384 (192 K), P3 at 640 (128 K), and P4 at 928 (64 K). Between them are four holes of 96, 64, 160, and 32 K; although there is a total of 352 K of free space, the largest hole is 160 K.

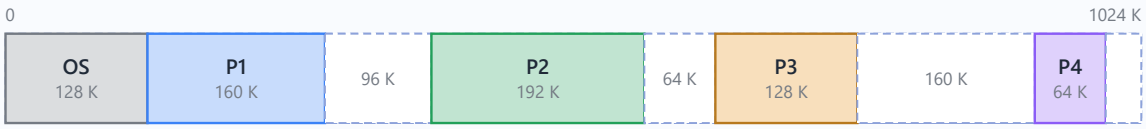


Figure 8.27: Before compaction: four holes between four processes, 352 K in total.

Compaction moves the processes down in turn. P1 is already right above the operating system. P2 moves from 384 to 288 (128 + 160). P3 moves from 640 to 480 (288 + 192), and P4 from 928 to 608 (480 + 128). In the end a single 352 K hole remains between 672 and 1024 (Figure 8.28). For this, 192 + 128 + 64 = 384 K has been copied. The base register of each moved process is updated with the new address.

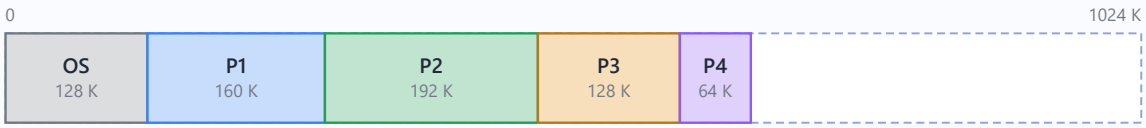


Figure 8.28: After compaction: the processes are contiguous and there is a single 352 K hole.

Compaction is possible only in systems that use execution-time binding. If the addresses in a process were fixed at load time, after the process is moved they would point to the wrong places. In a system that uses a base register, only the value of the register changes.

Compaction is expensive. By Tanenbaum's calculation, a machine that copies 8 bytes in 8 ns, that is, 1 GB per second, compacts 16 GB of memory in about 16 s. Even at today's copy rate of about 20 GB/s the job takes about a second (16 / 20 = 0.8 s; ≈ 0.86 s for 16 GiB). The processes stop in the meantime. Compaction must not be confused with data compression. The aim is not to make data smaller but to make blocks adjacent.

Compaction has not entirely become history, either. To produce contiguous physical frames for huge pages, Linux performs page migration in the background; the `kcompactd` kernel thread does this work.

8.6 Free-space management

The placement algorithms all rest on the same information: which regions are free? How this information is kept determines how fast searching and freeing can be. There are two basic structures: the bitmap and the linked list.

8.6.1 Bitmaps and linked lists

Memory is first divided into fixed-size *allocation units*; a unit can be a few words or a few kilobytes.

> **Definition 8.7**
>
> A **bitmap** is an array that keeps one bit for each allocation unit: 0 shows that the unit is free and 1 that it is in use. In the **linked-list** approach, a node is kept for every process and every hole; the node holds the kind (P: process, H: hole), the starting unit, and the length.

Figure 8.29 gives two representations of the same memory. In a 32-unit memory there are processes A, B, C, D, and E and three holes. The first byte of the bitmap is 11111000: units 0–4 belong to A, and 5–7 are free. In the linked list the same information is kept in the nodes “P 0 5, H 5 3, P 8 6, ….”

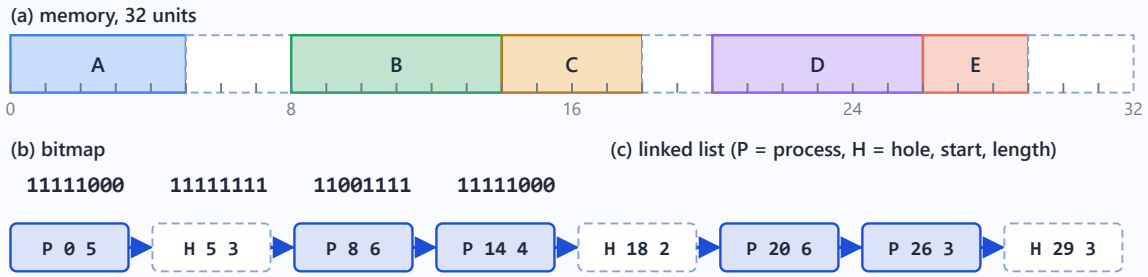


Figure 8.29: (a) A 32-unit memory with five processes and three holes, (b) the bitmap of the same state, (c) its linked list.

	Bitmap	Linked list
Structure	1 bit per allocation unit	A node for each process or hole
Size	Fixed: 1/33 of memory if the unit is 4 B; 512 KiB for 16 GiB of memory and 4 KiB units	Proportional to the number of processes and holes
Search	Finding k consecutive 0s is slow	The hole list is walked; merging is easy if it is sorted by address

Table 8.10: Comparison of the bitmap and the linked list.

The size of a bitmap depends on the unit size and is fixed. If the unit is 4 bytes, that is, 32 bits, 1 bit is spent for every 32 bits of memory; the bitmap takes 1/33 of memory. If 16 GiB of memory is divided into 4 KiB units, there are $2^{34} / 2^{12} = 2^{22}$ units. The bitmap then holds 2^{22} bits, that is, 512 KiB. As the unit gets smaller, the bitmap gets larger; as the unit gets larger, the internal fragmentation in the last unit increases.

Example 8.7

A 512 MiB memory is tracked with a bitmap using 4 KiB units. How many bytes is the bitmap?

1. **Number of units.** $2^{29} / 2^{12} = 2^{17}$ units.
2. **Bytes.** 1 bit per unit: 2^{17} bits / 8 = 2^{14} bytes = 16 KiB.

A common mistake is to take the number of bits (128 Ki) for bytes and get 128 KiB.

8.6.2 Code trace: searching for free space in a bitmap

The weakness of the bitmap is searching. Finding room for a process of k units means finding k consecutive 0s in the array. The function below does this with a linear scan.

```c
/* find k consecutive free (0) units */
int find(const char bit[], int n, int k) {
  int start = 0, len = 0;
  for (int i = 0; i < n; i++) {
    if (bit[i] == 0) {
      if (len == 0) start = i;  /* new run */
      if (++len == k) return start;
    } else
      len = 0;                  /* run broken */
  }
  return -1;
}
```

`len` holds the length of the run of zeros currently in progress, and `start` the index at which the run began. When a 0 is seen and the run has just begun (`len == 0`), the start is recorded and the length goes up by one. When the length reaches k, the start is returned. When a 1 is seen, the run is broken and `len` is reset to zero. Table 8.11 traces the search for k = 3 in the 10-unit bitmap 1 1 0 0 1 0 0 0 1 1.

i	bit[i]	start	len	Comment
0	1	0	0	unit in use
1	1	0	0	unit in use
2	0	2	1	new run
3	0	2	2	the run grows, has not reached k yet
4	1	2	0	the 2-unit run is broken; this hole is too small for the process
5	0	5	1	new run
6	0	5	2	the run grows
7	0	5	3	len = k: 5 is returned

Table 8.11: Trace of `find(bit, 10, 3)`. Verified with Python.

orange: current free run · green: area found

Figure 8.30: The result of the search: units 5, 6, and 7 are allocated to the process.

The function returns 5. The allocator then sets bits 5, 6, and 7 to 1. The search starts from the beginning of the array every time and in the worst case walks the whole bitmap: O(n) for n units. In a large bitmap this is slow. Real implementations speed it up by scanning the bitmap word by word and using the hardware's “find first zero bit” instructions, but the linear structure does not change.

Example 8.8

The bitmap of an 18-unit memory is 1 1 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 (units start at 0). Requests of 3, 4, and 5 units arrive in turn under first fit. Where does each go?

1. **Runs of zeros.** [2] (1 unit), [4–5] (2 units), [8–10] (3 units), [13–16] (4 units).
2. **3 units.** The first two runs are not enough; the run starting at 8 is exactly enough. 8–10 become full.
3. **4 units.** The remaining runs are [2], [4–5], [13–16]; only 13 is enough. 13–16 become full.
4. **5 units.** Only 3 free units are left and the longest run is 2: the request cannot be satisfied. There was no run longer than 4 at the start either.

8.6.3 Freeing in a linked list

The strength of the linked list is freeing. If the list is kept sorted by address, the two neighbors of a terminating process X are right before and after it in the list. There are four cases, depending on the kind of the neighbors (Table 8.12).

Left neighbor	Right neighbor	What happens?	Number of nodes
process	process	X's node is turned into H	unchanged
process	hole	X merges with the hole on the right	decreases by 1
hole	process	the hole on the left grows by the size of X	decreases by 1
hole	hole	three nodes shrink to one hole	decreases by 2

Table 8.12: The four cases when process X is freed.

Figure 8.31: When X terminates with holes on both sides, three nodes shrink to a single hole.

Merging is done in constant time, by looking only at the neighbors. In the last case the previous node has to be modified, so a doubly linked list makes the job easier. If the holes are kept in a separate list sorted by size, best fit gets faster. When the list is scanned from the start, the first hole that is big enough is already the smallest one that is big enough; first fit turns into best fit. But this time freeing gets slower, because neighboring holes are no longer next to each other in the list.

One more detail is important: no extra memory is needed to keep the hole list. The list can be stored inside the holes themselves. The first words of each hole hold that hole's size and the address of the next hole. Since the hole is free anyway, this space is free of charge.

8.6.4 Inside `malloc`: the header and the embedded free list

The same ideas apply in user space. The `malloc` and `free` functions of the C library manage the region called the heap with a free list. A special problem has to be solved for this: a separate data structure cannot be built for the nodes of the free list, because allocating space for that structure would itself require calling `malloc`. The solution is to embed the node inside the free chunk itself.

In the example allocator each chunk begins with a 16-byte header. If the chunk is in use, the header holds its size and a fixed magic number; if it is free, the header holds its size and the address of the next free chunk. Table 8.13 traces the operations on a 4096-byte heap.

Operation	Computation	Free list afterwards
start	4096 − 16 = 4080	[4080]
`p1 = malloc(200)`	200 + 16 = 216 B used; 4080 − 216 = 3864	[3864]
`p2 = malloc(100)`, `p3 = malloc(300)`	3864 − 116 − 316 = 3432	[3432]
`free(p2)`	size 100 is read from the header at `(char *)p2 - 16`	[100] → [3432]
`free(p1)`	merges with the neighboring free chunk: 216 + 116 = 332, size 332 − 16 = 316	[316] → [3432]

Table 8.13: An allocator with 16-byte headers on a 4096-byte heap. Verified with Python.

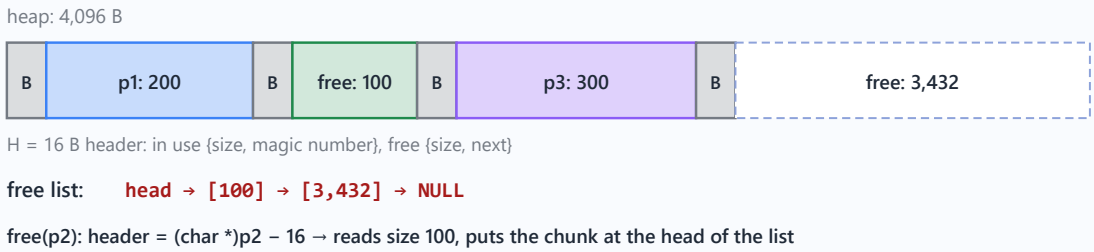


Figure 8.32: The heap after `free(p2)`: the free chunk of 100 bytes has been added to the head of the list. A request for 150 bytes no longer fits there.

The `free` function takes no size; it reads the size from the header 16 bytes before the address it is given. After `free(p2)` there is a free chunk of 100 bytes, but a request for 150 bytes does not fit in it: external fragmentation arises in user space too. With `free(p1)`, p1's chunk merges with the free chunk that is its neighbor in address order, and the list gets shorter.

Magic number
: The fixed pattern in the header can be corrupted by a buffer overflow, by freeing the same pointer twice, or by a wrong pointer. The library may notice this and stop the program.

The real request
: `malloc(n)` consumes at least n + header bytes. In addition, because addresses must be aligned, the size is rounded up; in glibc the alignment is 16 bytes.

Is it a system call?

No, `malloc` is a library function. When the heap runs out of space it grows the heap with the `brk` / `sbrk` system calls. Large requests (128 KiB and above in glibc) it takes directly with `mmap`.

Example 8.9

In the same allocator with 16-byte headers, `malloc(1000)` is called three times on a 4096-byte heap. What is the size of the free chunk? Can a fourth `malloc(1040)` be satisfied?

1. **Each call.** 1000 + 16 = 1016 bytes are consumed.
2. **What remains.** 4080 − 3 × 1016 = 4080 − 3048 = 1032 bytes.
3. **The fourth request.** Since 1040 > 1032, it cannot be satisfied from this heap. The allocator grows the heap with `sbrk` or `mmap`.

Common mistake

Giving `free` an address that `malloc` did not return, for example `free(p + 4)`. The allocator looks for the header at address `p + 4 − 16`; there is no valid header there. The result is undefined behavior. An allocator with a magic-number check stops the program; one without it corrupts the free list, and the bug shows up much later, in an unrelated place.

8.7 Paging

All the problems of contiguous allocation come from a single assumption: that the process sits in one piece in physical memory. Paging removes this assumption. The process is divided into small, equal pieces, and each piece can be put in any free place in memory. All of today's general-purpose operating systems use paging.

8.7.1 The basic idea

Definition 8.8

In **paging**, physical memory is divided into fixed-size blocks, **frames**. Logical memory is also divided into blocks of the same size, **pages**. Each process has a **page table**; the table records in which frame each page of the process is located.

The consequences of this arrangement are seen at once. The process's location in physical memory need not be contiguous; consecutive pages can sit in frames far apart (Figure 8.33). Hence there is no external fragmentation: every free frame is suitable for every page of every process. For a process of N pages, any N free frames are enough. Internal fragmentation remains only in the last page of the process, because the process size need not be an exact multiple of the page size.

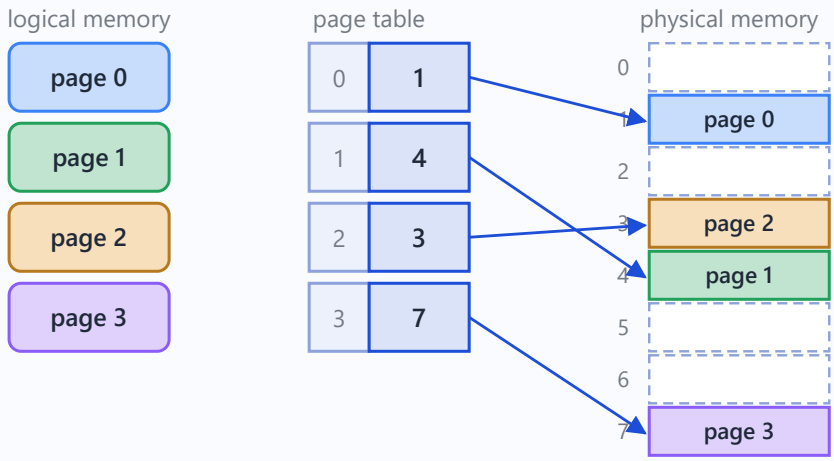


Figure 8.33: Paging. Logical pages are mapped through the page table to non-contiguous physical frames: 0 → 1, 1 → 4, 2 → 3, 3 → 7.

The operating system keeps track of which frames are free and builds each process's page table. The backing store used for swapping is also divided into blocks of the same size; thus a page can be written to any block on disk and read back into any frame.

8.7.2 Splitting the address into (p, d)

In paging, every logical address the CPU produces is split into two parts: the **page number** p and the **offset** d within the page. If the logical address space is 2^m bytes and the page size is 2^n bytes, the low n bits of the address give the offset and the upper m − n bits give the page number (Figure 8.34).

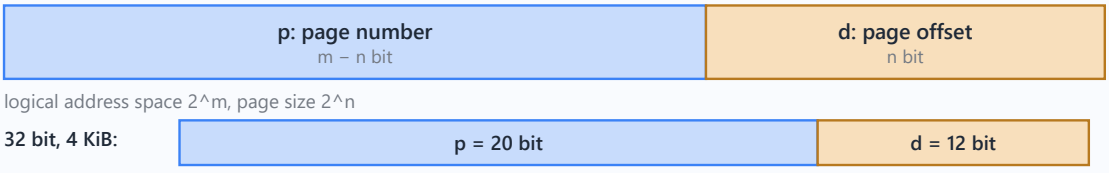


Figure 8.34: The logical address is split into the page number p and the offset d. With a 32-bit address and 4 KiB pages, p is 20 bits and d is 12 bits.

The page table translates p into a frame number f. The offset never changes. The physical address is found by adding the offset to the start of the frame:

> $p = \lfloor a / \text{page size} \rfloor$, $d = a \bmod \text{page size}$, $\text{physical} = f \times \text{page size} + d$

Because the page size is a power of two, this division and remainder are a bit shift and a mask. With 4 KiB pages, p = a >> 12 and d = a & 0xFFF. The hardware does this for free; a page size that is not a power of two would need a real divider circuit.

Example	Computation	Result
32-bit address, 4 KiB page	$4096 = 2^{12}$ → d 12 bits, p = 32 − 12 = 20 bits	2^{20} = 1,048,576 pages
0x12345678	p = 0x12345678 >> 12, d = 0x12345678 & 0xFFF	p = 0x12345 (74,565), d = 0x678 (1,656)
16-bit address, 4 KiB (Tanenbaum)	$8196 = 2 \cdot 4096 + 4$	p = 2, d = 4
48-bit (x86-64), 4 KiB	p = 48 − 12	36-bit page number

Table 8.14: Examples of splitting an address (verified with Python).

In hexadecimal the split is especially easy: each hex digit is 4 bits, so a 12-bit offset is the last three digits. The last three digits of 0x12345678 give the offset 0x678, and the rest gives the page number 0x12345.

Common mistake

Forgetting the offset when finding the physical address, or adding the whole logical address to the frame start. For a 2 KiB page, logical 9000, and the mapping 4 → 7, the correct computation is: p = ⌊9000 / 2048⌋ = 4, d = 9000 − 8192 = 808, physical = 7 × 2048 + 808 = 15144. Writing just 7 × 2048 = 14336 forgets the offset; 14336 + 9000 = 23336 adds the whole address.

8.7.3 Translation with the page table

Silberschatz's small example is suited to tracing a translation by hand. The page size is 4 bytes and physical memory is 32 bytes; that is, there are 8 frames. The process's 16-byte logical memory is filled with the letters a through p and occupies four pages. The page table is 0 → 5, 1 → 6, 2 → 1, 3 → 2.

Logical	Letter	p	d	Frame	Physical
0	a	0	0	5	5 × 4 + 0 = 20
3	d	0	3	5	5 × 4 + 3 = 23
4	e	1	0	6	6 × 4 + 0 = 24
13	n	3	1	2	2 × 4 + 1 = 9

Table 8.15: Translation with 4-byte pages (Silberschatz Figure 9.10).

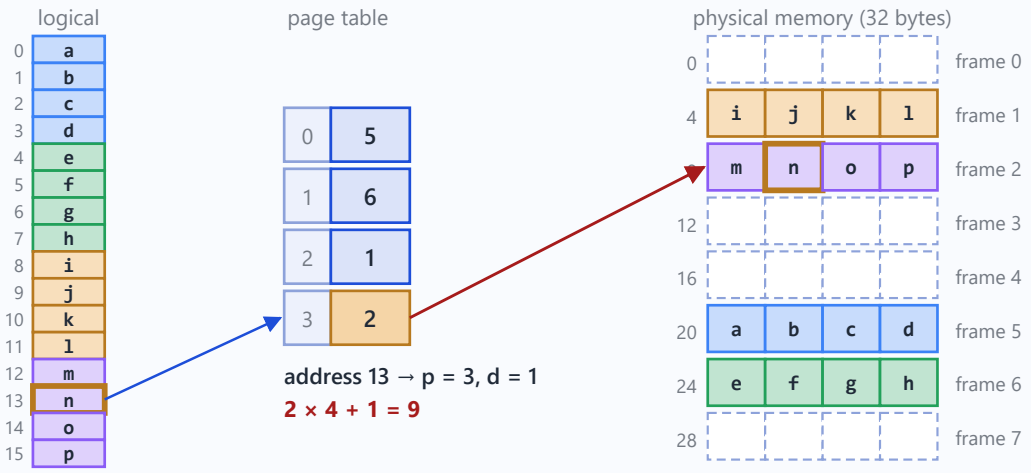


Figure 8.35: Translation with 4-byte pages: logical 13 (n) is page 3, offset 1; in frame 2 it corresponds to physical 9.

The example shows two important points. First, the letters of the process do not keep their order in physical memory: i…l are in frame 1, m…p in frame 2, a…d in frame 5. The program nevertheless sees its memory as consecutive from a to p. Second, the index of a page-table entry is the page number; the page number is not stored in the table separately. Looking at entry 3 of the table is the answer to the question “where is page 3?”

8.7.4 Translation inside the MMU

In Tanenbaum's example the virtual address is 16 bits (64 KiB, 16 pages) and physical memory is 32 KiB (8 frames). Since the page size is 4 KiB, the upper 4 bits of the virtual address are the page number and the lower 12 bits are the offset. The frame number is 3 bits. The instructions below access five different addresses; the page table is given in the comments.

```
/* 16-bit virtual address, 4 KiB pages */
MOV REG, 0
MOV REG, 8192
MOV REG, 20500
MOV REG, 8196
MOV REG, 32780

/* page table (page → frame) */
/* 0→2 1→1 2→6 3→0 4→4 5→3 */
/* 9→5 11→7, others: absent */
```

The MMU does the same job for every address. It uses the upper 4 bits as the table index and takes the 3-bit frame number from the entry. It appends the lower 12 bits to this number unchanged. Table 8.16 shows the translation of the five addresses.

Virtual	Binary (p | d)	p	d	Frame	Physical
0	`0000 000000000000`	0	0	2	8192
8192	`0010 000000000000`	2	0	6	24576
20500	`0101 000000010100`	5	20	3	12288 + 20 = 12308
8196	`0010 000000000100`	2	4	6	24576 + 4 = 24580
32780	`1000 000000001100`	8	12	absent	page fault

Table 8.16: MMU translation (16-bit virtual address, 4 KiB pages). Verified with Python.

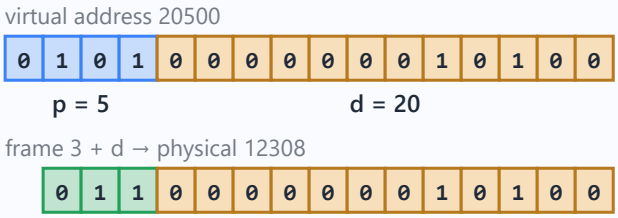


Figure 8.36: Translation of address 20500: the upper 4 bits (0101 = 5) are replaced by the frame number (011 = 3), and the lower 12 bits are copied as they are.

The last address is interesting. 32780 = 8 × 4096 + 12, and page 8 is not mapped; the present/absent bit of its entry in the table is 0. The MMU cannot produce a physical address and raises a **page fault** trap. Control passes to the operating system. If the page is on disk, the operating system brings it into a frame and updates the table; if the page is not valid at all, it terminates the process.

8.7.5 Page size and internal fragmentation

In paging, internal fragmentation arises only in the last page, but it is important enough to influence the choice of page size.

Example 8.10

The page size is 2048 bytes and the process is 72,766 bytes. How many frames does the process occupy, and how much internal fragmentation results?

1. **Division.** 72,766 = 35 × 2048 + 1086. The process occupies 35 full pages and a remainder of 1086 bytes.
2. **Frames.** The remainder also needs a frame: 36 frames.
3. **Internal fragmentation.** In the last frame 2048 − 1086 = 962 bytes are wasted.

In the worst case the process spills over a page by only one byte, and the internal fragmentation is one page minus one byte. If process sizes are randomly distributed, the average internal fragmentation is half a page. This is an argument in favor of small pages. On the other hand, small pages mean more pages and hence a larger page table and more TLB misses. Large pages give more efficient transfers between disk and memory and fewer table entries.

Tanenbaum measures this trade-off with a simple model. Let the average process size be s bytes, the size of a table entry e bytes, and the page size p. A process needs about s / p pages and s·e / p bytes of table per process; the internal fragmentation is p / 2 on average. The total overhead is:

> overhead = s·e / p + p / 2

As p gets smaller the first term grows, and as it gets larger the second term grows. Differentiating and setting the derivative to zero gives $-s{\cdot}e / p^2 + 1/2 = 0$, that is, $p = \sqrt{(2{\cdot}s{\cdot}e)}$. For s = 1 MiB and e = 8 bytes, $p = \sqrt{(2 \times 2^{20} \times 8)} = \sqrt{(2^{24})} = 2^{12} = 4096$ bytes (Figure 8.37). At this value the overhead is 2048 + 2048 = 4096 bytes; with pages of 2048 and 8192 bytes it rises to 5120 bytes.

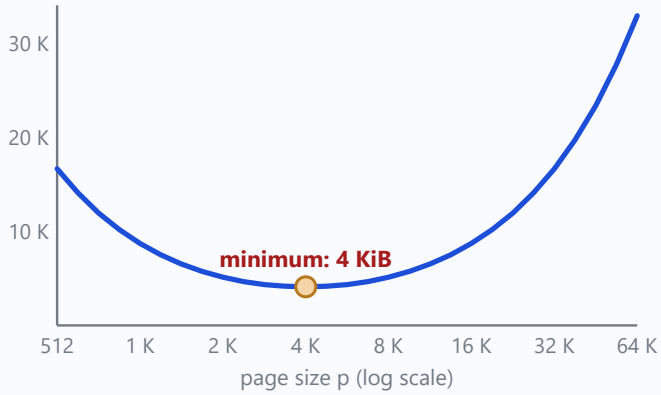


Figure 8.37: The overhead curve s·e / p + p / 2; for s = 1 MiB and e = 8 bytes the minimum is at 4 KiB.

Real systems stay in this neighborhood too. On x86-64 the base page is 4 KiB; there are also huge pages of 2 MiB and 1 GiB. ARM64 can use pages of 4, 16, or 64 KiB; Apple Silicon uses 16 KiB.

8.7.6 Free frames

When a new process arrives, the operating system takes as many frames from the free-frame list as the process has pages and fills in the page table. In the example of Figure 8.38 the free list is 14, 13, 18, 20, 15. A process of four pages takes the first four frames of the list: page 0 → 14, 1 → 13, 2 → 18, 3 → 20. Only 15 stays on the list.

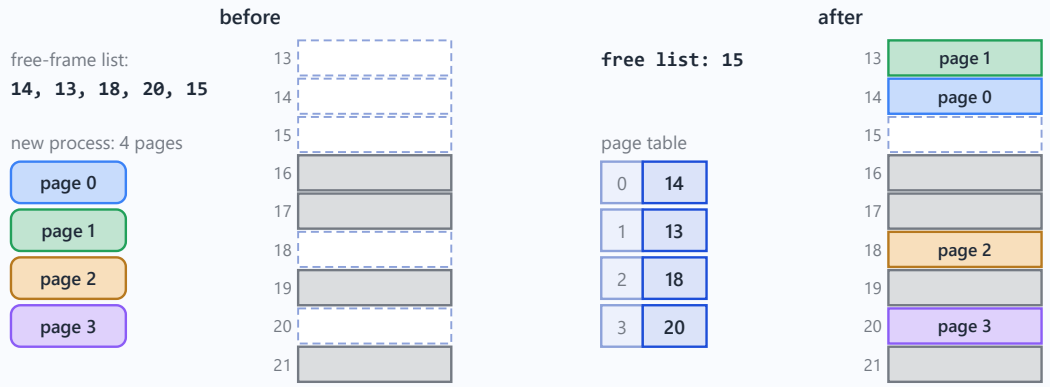


Figure 8.38: The four pages of the new process are placed in frames 14, 13, 18, and 20 of the free-frame list.

For this the operating system keeps a **frame table**: for every physical frame, whether the frame is free and, if it is in use, which page of which process it holds. This table is needed not only for allocation but for other jobs as well. For example, when a process gives a buffer address to the `read` system call, this address is logical. To be able to write the data there, the kernel translates the address into a physical address with the process's page table.

Because each process has a separate page table, on a context switch the hardware must be pointed at the new process's table. The dispatcher does this by updating a register; on x86 this register is CR3. This extra work lengthens the context-switch time.

8.7.7 The page-table entry

A page-table entry (PTE) holds not only the frame number. Alongside it are a number of bits used by the hardware and the operating system. Figure 8.39 shows the 64-bit entry of x86-64.

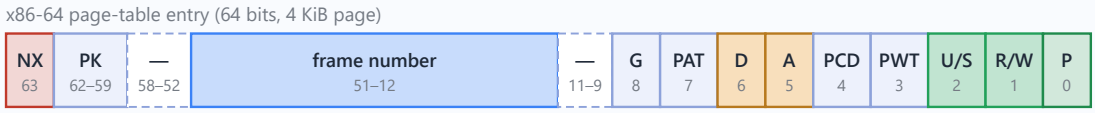


Figure 8.39: The x86-64 page-table entry (64 bits, 4 KiB page): NX, protection key, frame number, G, PAT, D, A, cache bits, U/S, R/W, and P.

Bit	Meaning (Tanenbaum's name)	Who writes it?
P	Present/absent: if 0, an access produces a page fault	OS
R/W, U/S, NX	Protection: writable? accessible in user mode? executable?	OS
A	Accessed (referenced): becomes 1 on a read or write; used in page replacement	hardware (the OS clears it)
D	Dirty (modified): becomes 1 on a write; a dirty page is written to disk when evicted	hardware

Bit	Meaning (Tanenbaum's name)	Who writes it?
PCD, PWT, PAT	Cache disabling and write policy: for memory-mapped I/O	OS
G	Global: not flushed from the TLB on a context switch (kernel pages)	OS

Table 8.17: The bits of a page-table entry.

The accessed and dirty bits give the operating system very valuable information. The hardware sets A to 1 on every access to a page and D to 1 on every write. The operating system periodically clears the A bits and learns which pages have been used recently. The D bit tells whether a page has to be written to disk when it is evicted from memory: the copy of an unmodified page on disk is already up to date.

The cache bits are needed for memory-mapped I/O. If a device's register is mapped to a memory address, the value read from that address must come from the device, not from the cache. Otherwise the program reads the device's old state.

If a page is not in memory, where is its place on disk kept? The disk address is not kept in the PTE; the hardware does not use it. For an entry whose P bit is 0, the remaining bits belong entirely to the operating system. Linux encodes the location in the swap area in these bits.

8.7.8 The valid / invalid bit

The page table contains entries for the whole address space of the process, but the process does not use all of this space. The entries for the unused pages are marked **invalid**. In Silberschatz's example the address is 14 bits (0 … 16383) and the page is 2 KiB, so there are 8 pages. The process uses only 0 through 10468: pages 0–5 are valid and are in frames 2, 3, 4, 7, 8, 9; pages 6 and 7 are invalid.

Address	p, d	Entry	Result
3,000	1, 952	valid, frame 3	3 × 2048 + 952 = 7,096
10,468	5, 228	valid, frame 9	9 × 2048 + 228 = 18,660 (the process's last address)
12,000	5, 1760	valid, frame 9	20,192 (outside the program, but not caught)
12,500	6, 212	invalid	trap

Table 8.18: Translation with the valid/invalid bit (2 KiB pages). Verified with Python.

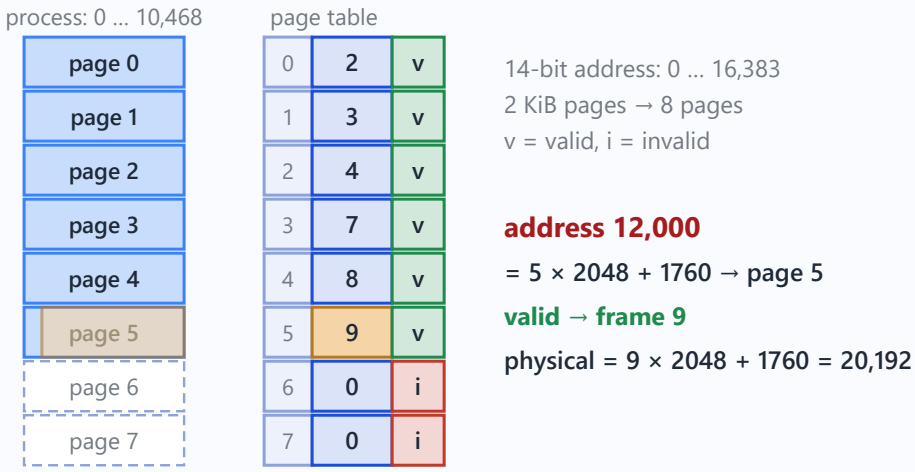


Figure 8.40: Address 12,000 is outside the program, but because it lies inside page 5, the hardware does not catch it.

The third row shows a limit of paging. The last address of the process is 10,468; 12,000 does not belong to the program. But page 5 covers 10,240 through 12,287 and is marked valid. Hardware protection is at page granularity: every address inside a page is either valid or none is. The unused part of the process's last page is another face of internal fragmentation. 12,500, on the other hand, is on page 6, and the invalid entry produces a trap. The operating system decides whether this trap is a programming error (`SIGSEGV`) or an access to a page not yet brought into memory.

Instead of keeping the whole table, a **page-table length register** (PTLR) can also be used. This register holds how many entries the table contains; a larger page number produces a trap directly. In this way no space is spent on the entries for unused pages.

8.7.9 Protection bits

In paging, protection is also done at the page level. The permission bits are kept separately in every PTE and are checked on every access, together with the TLB entry.

Read / write (R/W)
: Code pages are mapped read-only; an attempt to write to code produces a protection fault.

User / supervisor (U/S)
: Kernel pages cannot be accessed in user mode. In this way the same table can hold both the kernel's and the process's mappings.

No-execute (NX / XD)
: Code cannot be executed from data, stack, and heap pages. Operating systems enforce the W^X rule: a page is either writable or executable, never both. In this way attack code written onto the stack cannot be executed.

SMEP / SMAP
: While in kernel mode, they prevent the execution of code in user pages (SMEP) and unauthorized access to user data (SMAP).

Protection keys (PKU)
: With a 4-bit key in the PTE and a user register (PKRU), the permission of a group of pages can be changed without a system call.

Valid / invalid
: The crudest form of access permission: if the page does not exist in the address space at all, every access is a trap.

8.7.10 Shared pages

An important gain of paging is that common code can be shared easily. *Reentrant* code, that is, code that does not modify itself while running, can be shared among many processes. If three users run the same text editor, the editor's code pages are in memory only once. Each process's page table points to these frames; only the data pages are separate for each process (Figure 8.41).

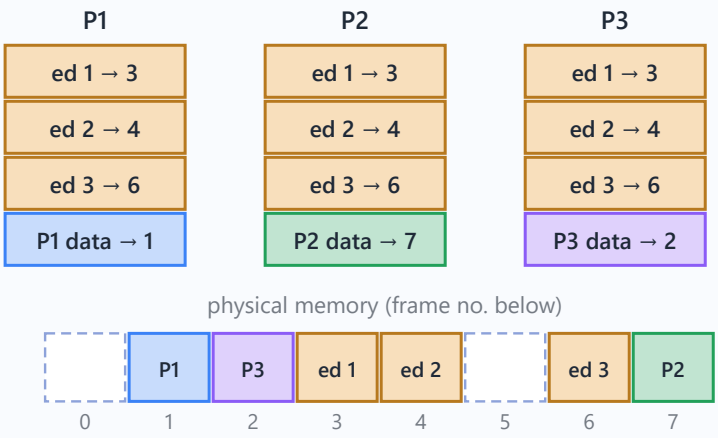


Figure 8.41: Three processes map the three pages of the editor's code to the same physical frames (3, 4, 6); each process's data is in a separate frame.

Shared libraries are shared in this way too. Sharing has a consequence: when the process that "owns" a page terminates, the frame cannot be freed at once, because other processes may be using it. That is why a reference count is kept for every shared frame.

Today's systems have other forms of sharing as well. On a `fork()` call the parent and child share all pages by **copy-on-write** (COW). The pages are marked read-only in both processes; when one of the processes tries to write to a page, a protection fault occurs and the kernel makes a copy of only that page. Memory-mapped files (`mmap`) and shared memory (`shm_open`), on the other hand, let two processes see the same frames; the processes communicate without copying data.

8.8 The TLB

Paging eliminates external fragmentation, but it has a price: every memory access first needs a table access. This section describes how this price is brought down to almost nothing.

8.8.1 Where does the page table live?

If the page table were small, it could be kept inside the processor in special registers. But with a 32-bit address and 4 KiB pages a table has 2^{20}, that is, more than a million entries. That many entries do not fit in registers. The table therefore lives in main memory. Its starting address is held in the **page-table base register** (PTBR); on x86 this register is CR3, and on ARM it is TTBR0/TTBR1. On a context switch only this register changes.

The problem with this solution is obvious: every data access is now two memory accesses. First the entry at address PTBR + p is read, then the data is accessed. The memory access time doubles. The solution is a small hardware cache that holds recently used translations.

Definition 8.9

The **TLB** (translation look-aside buffer) is a small, fast, associative cache inside the MMU. It holds recently used (page number, frame number) pairs. Being associative means that a page number is compared with all the entries at the same time, in parallel.

Translation proceeds as follows (Figure 8.42). When the CPU produces an address, the page number is first looked up in the TLB. If it is found (a *TLB hit*), the frame number is obtained at once and no table access is made. If it is not found (a *TLB miss*), the table in memory is reached through the PTBR, the entry is read, and it is loaded into the TLB for the next access. If the TLB is full, an entry is evicted.

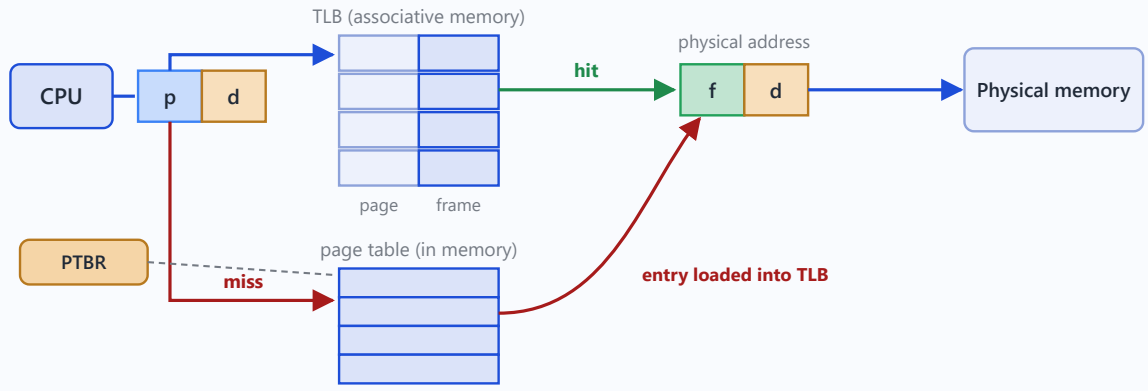


Figure 8.42: Paging hardware with a TLB. The page number is first looked up in the TLB; on a miss the table in memory is reached through the PTBR and the entry is loaded into the TLB.

In today's processors each core typically has an L1 TLB of 64 entries and an L2 TLB of 1–3 thousand entries. TLB entries can be tagged with an address-space identifier (ASID; PCID on Intel). In this way each entry knows which process it belongs to, and the TLB need not be flushed on a context switch.

8.8.2 Tracing TLB hits and misses

Let the page table of Tanenbaum's MMU example (0 → 2, 2 → 6, 5 → 3, 9 → 5, 11 → 7) now be used with a TLB. The TLB has 4 entries and evicts the least recently used (LRU) entry when room is needed. The TLB starts out empty. The pseudocode below summarizes what is done on each access.

```
p = addr / 4096;  d = addr % 4096;
if (p in TLB) {           /* hit */
  f = TLB[p];  make p the newest;
} else {                  /* miss */
  f = page_table[p];
  if (TLB full) evict the oldest entry;
  add (p, f) to the TLB;
}
physical = f * 4096 + d;
```

Access	p	Result	Physical	Evicted	TLB afterwards
0x0010	0	miss	0x2010	—	0
0x2004	2	miss	0x6004	—	0, 2
0x0020	0	hit	0x2020	—	2, 0
0x5010	5	miss	0x3010	—	2, 0, 5
0x9000	9	miss	0x5000	—	2, 0, 5, 9
0x2008	2	hit	0x6008	—	0, 5, 9, 2
0xB004	11	miss	0x7004	0	5, 9, 2, 11
0x0030	0	miss	0x2030	5	9, 2, 11, 0

Table 8.19: Trace of a 4-entry LRU TLB. In the TLB column the entries are ordered from oldest to newest. Verified with Python.

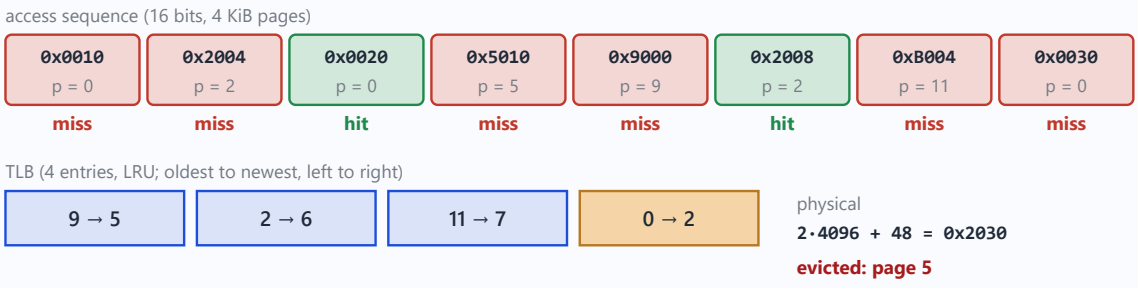


Figure 8.43: The last step of the trace: for 0x0030, page 0 is not in the TLB, and page 5, the least recently used, is evicted.

In eight accesses there are only 2 hits. On hits the page table is not consulted at all. After the hit on 0x0020, page 0 moves to the "newest" position. But when 0xB004 arrives, the TLB is full and the least recently used entry is again page 0, because 5, 9, and 2 have been used since 0x0020. The 0x0030 that follows immediately therefore misses. In short examples the ratio comes out low; in real programs, thanks to locality, the TLB hit rate exceeds 99%.

8.8.3 Effective access time

The gain from the TLB is measured with a single number.

Definition 8.10

The **effective access time** (EAT) is the average memory access time, found by taking into account the probabilities of TLB hits and misses. If the TLB lookup time is ε, the memory access time t, and the TLB hit ratio α, then for a single-level table:

$$\text{EAT} = (t + \varepsilon) \cdot \alpha + (2t + \varepsilon) \cdot (1 - \alpha) = 2t + \varepsilon - t \cdot \alpha$$

The logic of the formula is simple. On a hit the TLB is consulted and a single memory access is made: t + ε. On a miss the TLB is consulted, and then one memory access is made for the table entry and one for the data: 2t + ε. The average is the sum of these two times weighted by their probabilities.

Example 8.11

Let ε = 20 ns and t = 100 ns. What is the effective access time for α = 80% and α = 98%?

1. **Two cases.** Hit: 20 + 100 = 120 ns. Miss: 20 + 100 + 100 = 220 ns.
2. **α = 80%.** 120 × 0.80 + 220 × 0.20 = 96 + 44 = 140 ns. With the closed form, 2 × 100 + 20 − 100 × 0.80 = 140 ns.
3. **Interpretation.** Without a TLB every access would take 200 ns; 140 ns is better than that, but it is still 40% slower than the plain memory time.
4. **α = 98%.** 120 × 0.98 + 220 × 0.02 = 117.6 + 4.4 = 122 ns. The slowdown falls to 22%. In real workloads the ratio is usually above 99%; at α = 99% the EAT is 121 ns.

With multilevel tables a miss is much more expensive. In x86-64's four-level table a miss needs four table accesses and one data access: 20 + 4 × 100 + 100 = 520 ns. In that case for α = 98% the EAT = 120 × 0.98 + 520 × 0.02 = 117.6 + 10.4 = 128 ns, and for α = 99% it is 118.8 + 5.2 = 124 ns. The more expensive a miss is, the more the hit ratio matters. That is why the hardware also keeps the intermediate levels of the table walk in a separate cache (a *page walk cache*). A real miss thus often comes down not to four but to one or two memory accesses.

Example 8.12

The memory access takes 80 ns, the TLB lookup 10 ns, the table is single-level, and the hit ratio is 95%. (a) What is the EAT? (b) What must the hit ratio be at least for the EAT to fall to 92 ns?

1. **Two cases.** A hit takes 10 + 80 = 90 ns, a miss 10 + 80 + 80 = 170 ns.
2. **(a)** 0.95 × 90 + 0.05 × 170 = 85.5 + 8.5 = 94 ns.
3. **(b)** $90\alpha + 170(1 - \alpha) \le 92 \rightarrow 170 - 80\alpha \le 92 \rightarrow \alpha \ge 78 / 80 = 0.975$. The hit ratio must be at least 97.5%.

It is also instructive to count the effect of the TLB over a loop.

Example 8.13

The loop `for (i = 0; i < 1000; i++) a[i] = 0;` compiles to the four instructions below. The array a starts at virtual address 40,000, an `int` is 4 bytes, a page is 4 KiB, and the code lies in a single page. There is a linear page table. How many memory accesses are made in total without and with a TLB?

```
loop:  movl $0,(%edi,%eax,4)  /* a[i] = 0 */
       incl %eax              /* i++      */
       cmpl $1000,%eax
       jne  loop
```

1. **One iteration.** Four instructions are fetched and `movl` writes to memory once: 5 accesses.
2. **No TLB.** A PTE is read before every access: 10 per iteration, 10 × 1000 = 10,000 accesses in total.
3. **With a TLB.** To the 5 × 1000 = 5000 accesses are added only the PTE reads on TLB misses. The array lies between 40,000 and 43,999 and spans pages ⌊40,000 / 4096⌋ = 9 and ⌊43,999 / 4096⌋ = 10. Together with the code page there are three misses: 5000 + 3 = 5003 accesses.

8.8.4 TLB management

How the TLB is filled and flushed varies from architecture to architecture. Table 8.20 summarizes the main topics.

Topic	Description
Hardware-managed	x86-64, ARMv8, RISC-V: on a miss the MMU walks the page table itself (*page walk*); the operating system only builds the table.
Software-managed	In older RISC processors such as MIPS, SPARC, and Alpha, a miss traps to the operating system; the kernel finds the entry and writes it into the TLB. This is not true of all RISCs: ARMv8 and RISC-V are hardware-managed.
Soft / hard miss	Soft miss: the page is in memory and missing only from the TLB (nanoseconds). Hard miss: the page is on disk and I/O is needed (microseconds or milliseconds).
ASID / PCID	The entry is tagged with a process ID; the TLB is not flushed on a context switch. PCID reduces the cost of KPTI, which was introduced after Meltdown (2018).
Global entries	Kernel mappings are marked with the G bit and remain valid in all processes.
TLB shootdown	On a multicore system, when a mapping changes, the TLBs of the other cores are invalidated with inter-processor interrupts (IPIs); newer processors have broadcast instructions (AMD INVLPGB).

Table 8.20: TLB management.

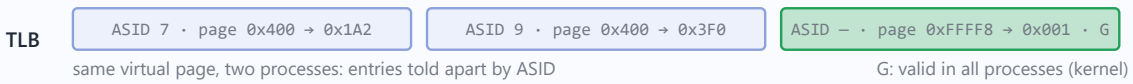


Figure 8.44: A TLB tagged with ASIDs: the same virtual page (0x400) of two processes maps to different frames; the kernel entry marked G is valid in all processes.

The value of the ASID shows at context switch. Without tags, the whole TLB has to be flushed so that the new process does not use the old process's translations, and the new process then suffers a run of misses on its first accesses. With ASIDs, the same virtual page of two processes can sit side by side in the TLB (Figure 8.44). The *TLB shootdown*, on the other hand, is a problem of multicore systems: when one core changes a mapping, it has to tell the other cores that hold the same mapping in their TLBs. This communication is expensive and limits scaling in large systems.

8.9 Page-table structures

A single-level, linear page table works for small address spaces. As the address space grows, the table itself becomes a memory problem. This section describes three ways of shrinking the table: hierarchical, hashed, and inverted tables.

8.9.1 The hierarchical page table

With a 32-bit address and 4 KiB pages, a single-level table has 2^{20} entries; with 4-byte entries the table takes 4 MiB. This table must be separate for each process and contiguous in memory. In a system with 100 processes, the page tables alone amount to 400 MiB. On 64 bits the situation is far

worse. With 4 KiB pages and 8-byte entries, a single-level table takes $2^{52} \times 8$ B = 32 PiB, about 33.6 million GiB. Even if only 48 bits of the address are used, $2^{36} \times 8$ B = 512 GiB is needed.

The solution is to page the table itself. The virtual address is split into two indexes and an offset. On 32-bit x86, 10 bits index the top-level table (the page directory), 10 bits index the second-level table, and 12 bits are the offset (Figure 8.45). What is split here is not the table entry but the virtual address.

Definition 8.11

A **hierarchical page table** is a page table kept as a multilevel tree. Each entry of the top-level table points to a lower-level table; only the regions in use have lower-level tables.

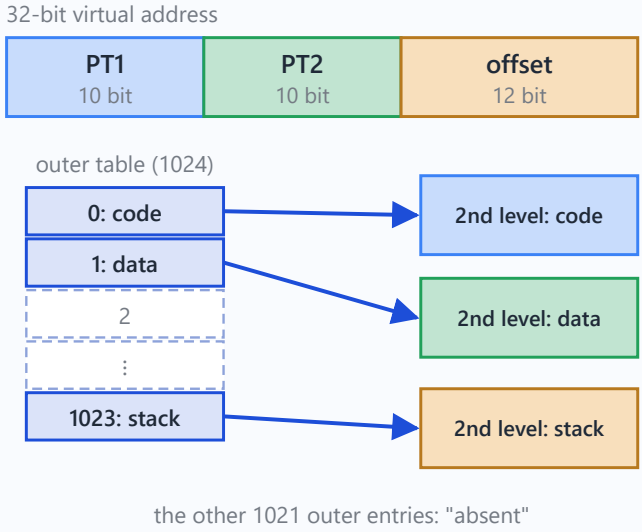


Figure 8.45: A two-level page table. Only the code, data, and stack entries of the top-level table point to second-level tables; the remaining 1021 top-level entries are “absent.”

The gain comes from sparse address spaces. Consider a process that uses 4 MiB of code, 4 MiB of data, and 4 MiB of stack. Each second-level table covers a region of 1024 pages × 4 KiB = 4 MiB. For this process one top-level table and three lower tables are enough: 4 × 4 KiB = 16 KiB. A single-level table would take 4 MiB. The lower tables of unused regions are never created; their entries in the top-level table are simply marked “absent.”

8.9.2 A two-level translation example

Example 8.14

In Tanenbaum's example the virtual address is 0x00403004 (4,206,596). This address lies between 4 MiB and 8 MiB, in the data region. How is it translated in a two-level table?

1. **Splitting.** 0x00403004 = 0000000001 0000000011 000000000100. The upper 10 bits give PT1 = 1, the middle 10 bits PT2 = 3, and the offset = 4.
2. **Top-level table.** Entry 1 of the top-level table points to the second-level table of the 4–8 MiB region. Each top-level entry covers a region of 4 MiB.
3. **Second level.** Entry 3 of that table is consulted. This entry is the page at byte range 12,288–16,383 within the region; in absolute terms 4,206,592–4,210,687. If the entry's present bit were 0, a page fault would occur.
4. **Physical address.** The entry gives the frame number (0x1F3 in the example). The offset is added: 0x1F3004.

On a TLB miss this translation needs two table accesses and one data access. The frame numbers in the example are for illustration.

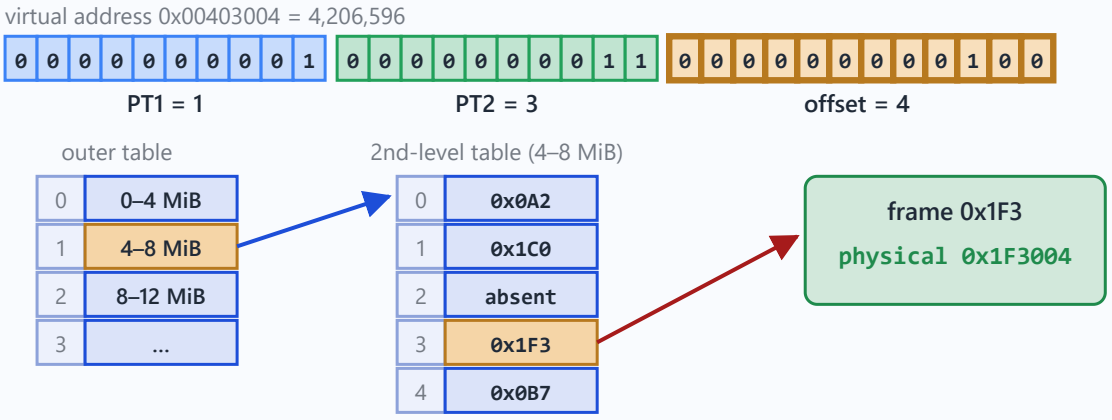


Figure 8.46: Translating address 0x00403004: PT1 = 1, PT2 = 3, offset = 4; frame 0x1F3, physical 0x1F3004.

On 32-bit x86 the address of the top-level table is in CR3. Each of the tables at the two levels fills exactly one 4 KiB page: 1024 entries × 4 bytes. This choice is no coincidence. That a table fits exactly in a page means that the lower tables can be placed in any frame like ordinary pages and, if necessary, swapped to disk.

8.9.3 Code trace: two-level translation

What the MMU does can be written in C as follows:

```c
/* 32 bits: 10 + 10 + 12 bits */
uint32_t translate(uint32_t va) {
  uint32_t p1 = va >> 22;
  uint32_t p2 = (va >> 12) & 0x3FF;
  uint32_t d  = va & 0xFFF;
  pte_t e1 = outer[p1];
  if (!e1.p) page_fault(va);
  pte_t e2 = e1.table[p2];
  if (!e2.p) page_fault(va);
  return (e2.frame << 12) | d;
}
```

`va >> 22` takes the upper 10 bits. `(va >> 12) & 0x3FF` first drops the offset and then keeps the lower 10 bits, that is, the middle index. `va & 0xFFF` is the offset. The function goes to the entry of the top-level table and from there to the entry of the second-level table; at both steps it raises a page fault if the present bit is 0. At the end it shifts the frame number left by 12 bits and combines it with the offset.

Variable	va = 0x00801A3C	va = 0x01403004
`p1 = va >> 22`	2	5
`p2 = (va >> 12) & 0x3FF`	0x801 & 0x3FF = 1	0x1403 & 0x3FF = 3
`d = va & 0xFFF`	0xA3C	0x004
`e1 = outer[p1]`	P = 1 (1st memory access)	P = 0 → page fault
`e2 = e1.table[p2]`	frame 0x2F, P = 1 (2nd access)	—
return	(0x2F << 12) | 0xA3C = 0x0002FA3C	—

Table 8.21: Trace of the `translate` function with two addresses. In the top-level table only entry 2 is valid; in its table, entry 1 → frame 0x2F. Verified with Python.

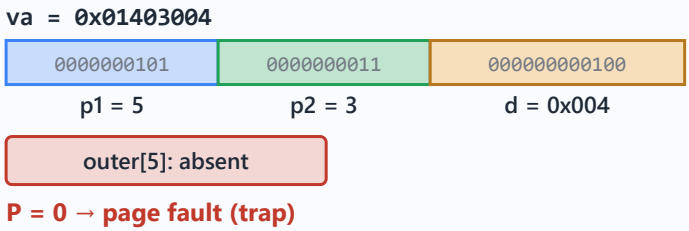


Figure 8.47: For 0x01403004, entry 5 of the top-level table is invalid: there is no second-level table at all for the 20–24 MiB region.

For the first address the third memory access goes to the data. The second address lies in the 20–24 MiB region. No second-level table has been created for this region; the MMU raises a trap at the top-level table. That no memory has been spent on the table is the whole gain of the hierarchical table.

Example 8.15

With the same 10 + 10 + 12 split, a process uses the lowest 12 MiB and the topmost 4 MiB of its address space. How many page tables are needed?

1. **Lower tables.** Each lower table covers 4 MiB: 3 tables for the lowest 12 MiB and 1 for the topmost 4 MiB.
2. **Total.** 4 lower tables + 1 top-level table = 5 tables = 20 KiB. A linear table would be 4 MiB.

8.9.4 x86-64: four- and five-level paging

On x86-64 each page table is 512 entries × 8 bytes = 4 KiB; that is, again exactly one page, and each level uses a 9-bit index. With four levels and a 12-bit offset, 4 × 9 + 12 = 48 bits of virtual address are obtained; this is an address space of 256 TiB (Figure 8.48). The levels are named, from top to bottom, PML4, PDPT, PD, and PT.

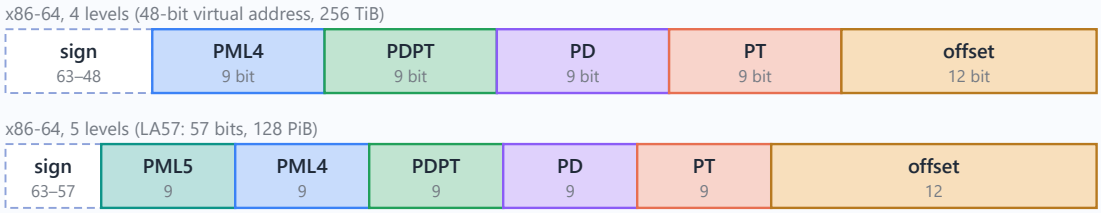


Figure 8.48: The split of an x86-64 virtual address in four- and five-level paging: each level 9 bits, the offset 12 bits.

The upper 16 bits of a 64-bit address are unused, but they cannot be arbitrary either: all of them must be copies of bit 47. Such addresses are called *canonical addresses*. As a result the address space is divided in two: the lower half goes to the user and the upper half to the kernel. Five-level paging (LA57) adds a PML5 level and gives a 57-bit, 128 PiB virtual address. Intel Ice Lake server processors and AMD Zen 4 and later support it; Linux has been able to use it since 4.14.

This is why the hierarchical table works on 64 bits too: not all 64 bits are used, and the address space in use is sparse. Today's x86-64 and ARMv8 use exactly a hierarchical table. If the PS bit in an intermediate-level entry is 1, the walk ends there: a huge page has been found, 2 MiB at the PD level or 1 GiB at the PDPT level.

8.9.5 The four-level walk

Example 8.16

How is VA = 0x00007FFE12345678, an address near the user stack, translated in four levels?

1. **Splitting (9 + 9 + 9 + 9 + 12).** PML4 = (VA >> 39) & 0x1FF = 255, PDPT = (VA >> 30) & 0x1FF = 504, PD = (VA >> 21) & 0x1FF = 145, PT = (VA >> 12) & 0x1FF = 325, offset = 0x678.
2. **PML4.** CR3 gives the frame of the PML4 table (0x1A2B). Entry 255 is read; the frame of the PDPT table is 0x3C4.
3. **PDPT.** PDPT[504] gives the frame of the page directory (PD): 0x7D10. Had PS = 1 here, a 1 GiB page would have been found.
4. **PD.** PD[145] gives the frame of the last-level table: 0x2E95. Had PS = 1, a 2 MiB page would have been found.
5. **PT.** PT[325] gives frame 0x5A3F7; the offset is added: physical 0x5A3F7678.

On a TLB miss, 4 table accesses and 1 data access are made. The frame values are illustrative; the indexes were computed with Python.

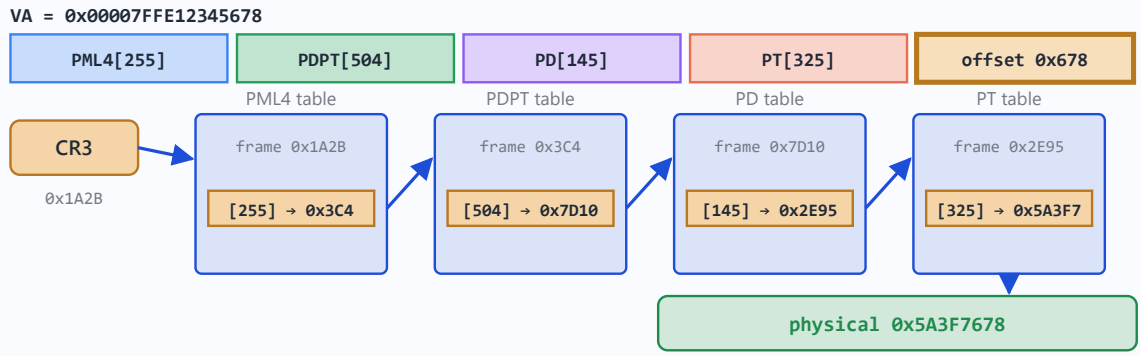


Figure 8.49: The end of the x86-64 four-level page-table walk: PT[325] gives frame 0x5A3F7.

Common mistake

Using a 10-bit index on x86-64 as on 32-bit x86. Because the entry grows to 8 bytes, 512 entries fit in a 4 KiB table; the index is 9 bits. Indexes are masked with 0x1FF, not 0x3FF.

8.9.6 Huge pages

Programs with a large working set, such as databases, virtual machines, and AI workloads, suffer constant TLB misses with 4 KiB pages. The measure of the problem is the **TLB reach**: the number of entries × the page size. A TLB of 1536 entries covers only 6 MiB with 4 KiB pages; with 2 MiB pages the same TLB covers 3 GiB (Figure 8.50).

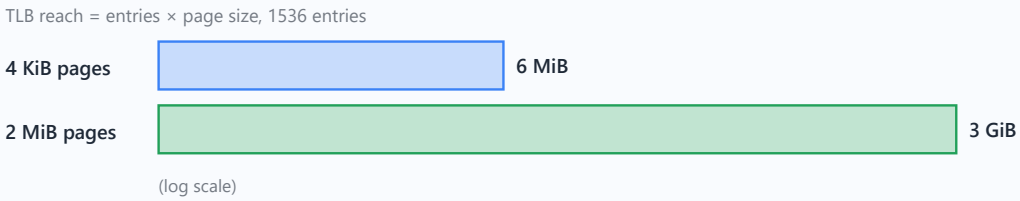


Figure 8.50: TLB reach: with 1536 entries, 6 MiB with 4 KiB pages and 3 GiB with 2 MiB pages.

The table size shrinks as well. Mapping 1 GiB with 4 KiB pages takes $2^{30} / 2^{12} = 262{,}144$ PTEs, that is, a 2 MiB table. With 2 MiB pages only 512 PD entries, that is, 4 KiB, are enough.

Linux offers huge pages in three forms. `hugetlbfs` is for programs that explicitly ask for pre-reserved 2 MiB or 1 GiB pages. With transparent huge pages (THP), the kernel maps suitable regions with huge pages on its own. mTHP in recent versions uses intermediate sizes of 16–64 KiB. On Windows the same job is done by "large pages."

Huge pages have two costs: internal fragmentation increases, and a contiguous 2 MiB physical region has to be found. The problem of finding contiguous regions explains why the buddy allocator and compaction are still important today. ARMv8 offers an intermediate solution to this problem: 16 adjacent 4 KiB entries marked with the *contiguous* bit can be held as a single TLB entry, as 64 KiB.

8.9.7 The hashed page table

For address spaces larger than 32 bits, another approach is the hash table.

Definition 8.12

In a **hashed page table** the virtual page number is passed through a hash function. Each bucket of the table is a linked list of the entries (virtual page, frame, next) that fall on the same hash value.

Example 8.17

In a table with 8 buckets, h(p) = p mod 8. The chain of bucket 2 is (p = 10 → f = 7), (p = 26 → f = 3). Where is virtual address 0x1A0AB translated to? (4 KiB pages)

1. **Splitting.** p = 0x1A = 26, d = 0x0AB.
2. **Hash.** h(26) = 26 mod 8 = 2.
3. **Chain.** The first entry is p = 10: no match, move on to the next. The second entry is p = 26: it matches, frame 3.
4. **Physical address.** 3 × 4096 + 0xAB = 0x30AB.

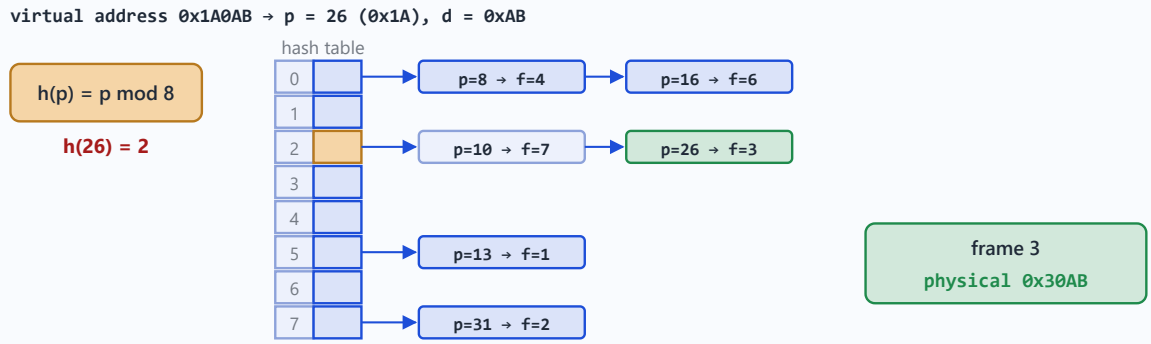


Figure 8.51: Looking up page 26 in a hashed page table: in the chain of bucket 2 the second entry matches.

The size of a hash table is proportional to the number of pages in use, not to the size of the address space. This is an attractive property for sparse 64-bit address spaces. In the *clustered* variant each entry holds the frames of several (for example 16) consecutive pages instead of a single page. This variant suits address spaces that are sparse but locally dense.

8.9.8 The inverted page table

In all the tables so far there was one table per process, and the size of the table depended on the virtual address space. The inverted table turns the point of view around.

Definition 8.13

An **inverted page table** is a single table in the system that keeps one entry per physical frame. Each entry records which page of which process is in that frame, that is, a (pid, page) pair. The index of the entry is the frame number.

Translation is a search. For the address (pid = 2, p = 5, d = 0x123) the table is scanned from the start. Entry 0 (1, 0), entry 1 (2, 3), entry 2 (1, 5), and entry 3 (3, 1) do not match. Entry 4 (2, 5) matches. Since the index itself is the frame number, the physical address is 4 × 4096 + 0x123 = 0x4123 (Figure 8.52).

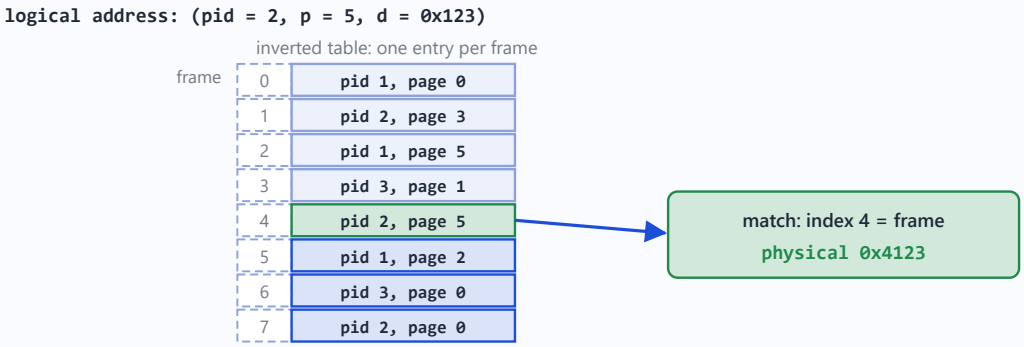


Figure 8.52: Looking up (pid 2, page 5) in the inverted page table: a match is found at index 4, and the physical address is 0x4123.

The size of the table is proportional to physical memory and independent of the number of processes. With 16 GiB of RAM and 4 KiB frames there are 4,194,304 entries; with 8 bytes per entry the table takes 32 MiB. This is a single table for the whole system.

There are three costs. The first is searching: a linear scan is unacceptably slow on every access. Real systems therefore hash the (pid, p) pair, go straight to the relevant entry, and put a TLB in front. The second is sharing: since a frame can hold only one (pid, page) pair, shared memory becomes hard. The third is page faults: the place on disk of a page that is not in memory cannot be found in the inverted table; for this a separate external table per process is needed. The PowerPC and IBM POWER architectures use a hashed inverted table; IA-64 also adopted this approach.

8.10 Segmentation

Paging is a mechanism completely hidden from the programmer: the address space is a one-dimensional array of bytes, and page boundaries have nothing to do with the logical structure of the program. Segmentation, on the other hand, divides the address space according to the program's own parts.

8.10.1 Why segmentation?

Tanenbaum's example is a compiler. While running, a compiler builds many tables: the source text, the symbol table, the table of constants, the parse tree, and the call stack. All of these tables grow and shrink during the run. In a one-dimensional address space each table is given a consecutive region. When a table fills up, it collides with its neighbor even if the neighbor has free space (Figure 8.53). The symbol table overflows into the source text's region; the compiler either stops or has to shift the tables by hand.

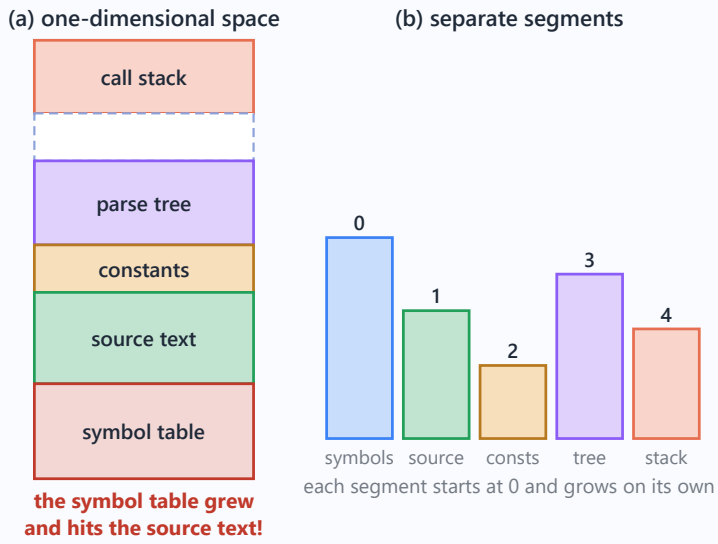


Figure 8.53: (a) In a one-dimensional address space the compiler's tables collide with each other. (b) In separate segments each table starts at 0 and grows independently.

Definition 8.14

Segmentation is giving a process several independent, variable-length address spaces. Each address space is called a **segment**. An address has two parts: (segment number, offset within the segment).

Segmentation matches the programmer's view. The programmer thinks of the program not as a single array of bytes but as separate units such as the main program, functions, the stack, arrays, and the symbol table. Segmentation gives each of these units its own address space. Each segment starts at 0 and can grow and shrink without affecting the other segments.

8.10.2 Translation with the segment table

Translation is done with a **segment table**. Each entry of the table holds the base (physical starting address) and the limit (length) of one segment. The location of the table in memory is held by the segment-table base register (STBR), and the number of entries by the segment-table length register (STLR). For an address (s, d) the hardware first checks the condition d < limit[s]. If it holds, the physical address is base[s] + d; if not, a trap occurs. Segmentation is really a separate base–limit pair for each segment.

(s, d)	d < limit?	Result
(2, 53)	53 < 400	4,300 + 53 = 4,353
(3, 852)	852 < 1,100	3,200 + 852 = 4,052
(1, 399)	399 < 400	6,300 + 399 = 6,699

(s, d)	d < limit?	Result
(0, 1,222)	1,222 ≥ 1,000	trap

Table 8.22: Silberschatz's five-segment example. Segment table (limit, base): 0 → (1000, 1400), 1 → (400, 6300), 2 → (400, 4300), 3 → (1100, 3200), 4 → (1000, 4700). Verified with Python.

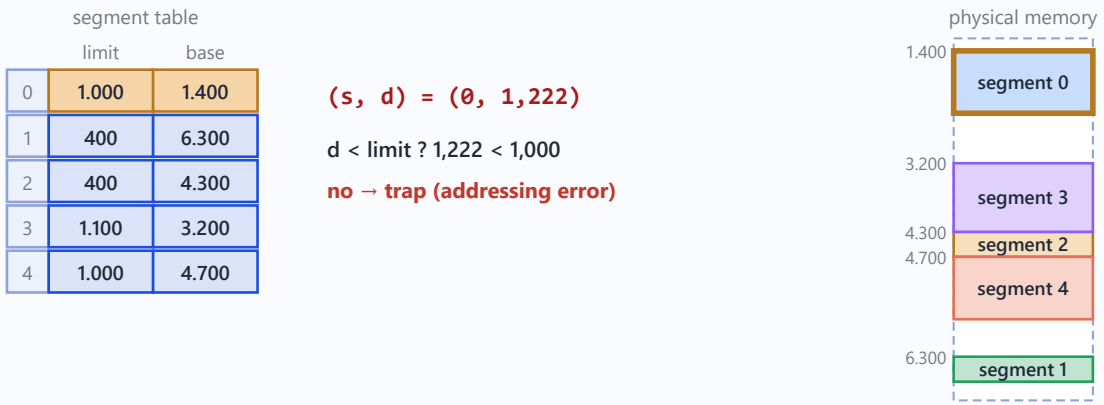


Figure 8.54: Address (0, 1,222): segment 0 is only 1,000 bytes long, so the hardware raises a trap.

Without the check in the last row, the address would be 1,400 + 1,222 = 2,622. This address is outside segment 0; it might belong to another segment of the process or to the memory of another process. The limit check prevents exactly this.

Common mistake

Counting an access as valid when the offset equals the limit. Valid offsets are 0 … limit − 1. In a segment with limit 400, (1, 399) refers to the last valid byte; (1, 400) is already a trap.

8.10.3 The bits that select the segment, and the downward-growing stack

The segment number need not be a separate part of the address; the upper bits of the virtual address can also select the segment. In an example similar to the one in OSTEP, the virtual address is 14 bits (16 KiB). The upper two bits select the segment: 00 code, 01 heap, 11 stack. The remaining 12 bits are the offset, so a segment can be at most 4 KiB. The segment registers are as follows:

Segment	Base	Size	Direction	Protection
00 code	24,576	3,072	+	`r-x`
01 heap	40,960	2,048	+	`rw-`
11 stack	32,768	2,048	−	`rw-`

Table 8.23: Segment registers (14-bit virtual address).

The “direction” bit of the stack segment is negative: the stack grows downward. In such a segment the offset is interpreted as negative. The largest segment size (4,096) is subtracted from the offset; the absolute value of the negative number obtained must not exceed the size. The physical address is the base plus the negative offset.

Example 8.18

Where do virtual addresses 1,000, 4,500, 16,000, and 7,000 go?

1. **1,000.** Binary 00 001111101000: segment 00 (code), offset 1,000. 1,000 < 3,072, valid. Physical 24,576 + 1,000 = 25,576.
2. **4,500.** 01 000110010100: segment 01 (heap), offset 4,500 − 4,096 = 404. 404 < 2,048, valid. Physical 40,960 + 404 = 41,364. Adding the whole virtual address to the base (40,960 + 4,500 = 45,460) would be wrong.
3. **16,000.** 11 111010000000: segment 11 (stack), offset 3,712. Because the stack grows downward, 3,712 − 4,096 = −384. |−384| ≤ 2,048, valid. Physical 32,768 − 384 = 32,384.
4. **7,000.** 01 101101011000: segment 01 (heap), offset 2,904. 2,904 ≥ 2,048: the hardware raises a trap, and the operating system terminates the process with `SIGSEGV`.

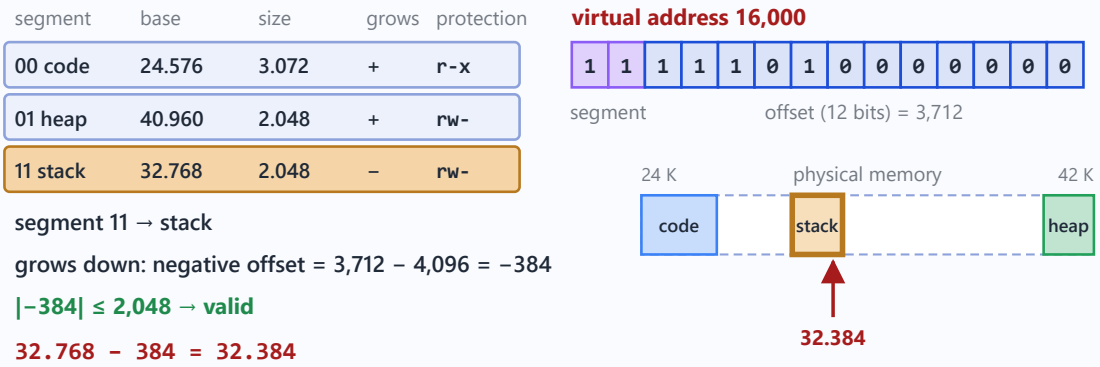


Figure 8.55: Address 16,000: the upper bits 11 select the stack; because the stack grows downward, the offset is interpreted as −384 and the physical address is 32,384.

Example 8.19

With the same registers, where do virtual addresses 13,000 and 14,500 go?

1. **13,000.** 13,000 − 12,288 = 712; segment 11 (stack), offset 712. 712 − 4,096 = −3,384; since |−3,384| > 2,048, a trap.
2. **14,500.** Segment 11, offset 2,212. 2,212 − 4,096 = −1,884; |−1,884| ≤ 2,048, valid. Physical 32,768 − 1,884 = 30,884.

Protection bits also make sharing possible. A code segment with `r-x` permission can be mapped to the same physical region in many processes. Because the segment is read-only, if one of the processes tries to write to it, a protection fault occurs and the others are not affected.

8.10.4 Pros and cons of segmentation

The advantages of segmentation come from the logical structure of the program:

- Tables that grow or shrink do not affect one another.
- Segments can be compiled separately. If a function is in its own segment, the addresses of the other segments do not change when the function changes.
- A library can be put in a separate segment and shared among processes.
- Protection is set per segment: code is only executable, constants only readable, data readable and writable.

The disadvantages are familiar from contiguous allocation. Segments are of variable length; as they move around in memory, holes form between them. This external fragmentation is also called *checkerboarding* because of its pattern (Figure 8.56). The remedy is again compaction. In addition, a segment has to be in memory in its entirety; a very large segment may not fit in memory. The solution to this last problem is to page the segments as well; MULTICS and the Intel 386 followed this route.

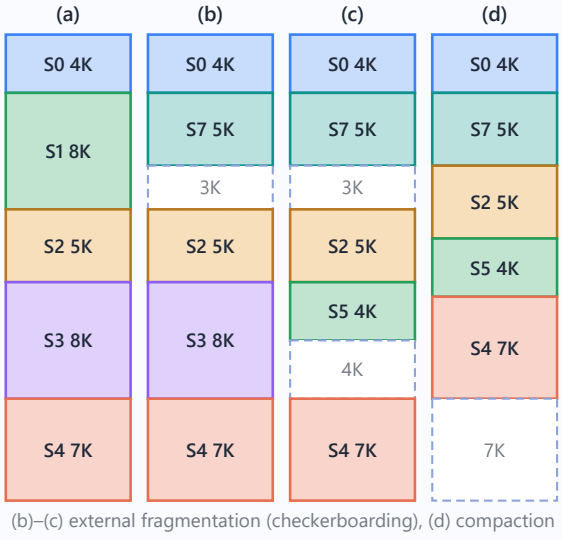


Figure 8.56: The checkerboard that results from segments being moved around: (b)–(c) external fragmentation, (d) compaction.

8.10.5 Comparison with paging

Table 8.24 gives Tanenbaum's comparison table, adapted to today's hardware.

Issue	Paging	Segmentation
Must the programmer be aware of the technique?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed physical memory?	Yes	Yes
Can code and data be separated and protected separately?	Yes, with page bits*	Yes
Can tables whose size varies be accommodated easily?	No	Yes
Is sharing of functions among users easy?	With shared pages*	Yes
Fragmentation	Internal (last page)	External
Why was it invented?	A linear address space larger than physical memory	To split program and data into independent logical spaces; sharing, protection

Table 8.24: Paging and segmentation (adapted from Tanenbaum Figure 3-32). * Tanenbaum says “no” for paging here; today's R/W/NX bits and shared pages do the same job at page level.

Figure 8.57: In paging, the single linear address space is divided into equal pages; in segmentation, code, data, and stack are separate segments of different sizes.

The two techniques were born to solve different problems. Paging was invented to provide a single linear address space larger than physical memory; segmentation to split the program into logical pieces and to protect and share these pieces separately. History has come down on the side of paging: the protection and sharing advantages of segmentation can be had with page bits, while paging's advantage of escaping external fragmentation cannot be had in any other way.

8.11 Real architectures and practice

8.11.1 x86: segmentation, paging, and the flat model

The x86 family carries the whole history of segmentation. In the 8086's *real mode* the physical address is computed as segment × 16 + offset. The result is 20 bits, giving a 1 MiB address space; there is no protection. In the 80386's *protected mode*, a logical address consists of a selector and an offset. Of the 16-bit selector, 13 bits are the index, one bit selects the table (GDT or LDT), and two bits are the requested privilege level (RPL). The selector picks an 8-byte descriptor from the table; the

descriptor holds the base and limit of the segment. Base + offset gives the *linear address*. The linear address then goes through two-level paging and is translated into a physical address (Figure 8.58). This is the hardware counterpart of MULTICS's idea of "paged segments."

Figure 8.58: In the Intel 32-bit architecture, segmentation translates a logical address into a linear address and paging translates the linear address into a physical address.

Despite this rich hardware, Linux and Windows effectively turned segmentation off even on 32 bits. All segments were given base 0 and limit 4 GiB. The linear address is then the same as the logical address; this is called the **flat model**. Protection is provided entirely by page bits. One reason for this is portability: segmentation existed only on x86, whereas paging exists on every architecture.

The long mode of x86-64 reflected this in hardware. The bases of the CS, DS, ES, and SS segments are taken as 0 by the hardware and the limit is not checked. Only the bases of the FS and GS segments are still used: in user space to point to thread-local storage (TLS), in the kernel to point to per-CPU data. CS additionally carries the current privilege level.

8.11.2 ARMv8-A and RISC-V

The two other big architectures of today implement the same ideas under different names (Table 8.25).

	ARMv8-A (AArch64)	RISC-V
Page (granule)	4, 16, or 64 KiB; Apple Silicon and some Android 15+ devices use 16 KiB	4 KiB (+ 2 MiB, 1 GiB, 512 GiB huge pages)
Virtual address	48 bits (4 levels with 4 KiB, L0–L3); 52 bits with LVA/LPA2	Sv39 (3 levels), Sv48 (4), Sv57 (5)
Table base	TTBR0_EL1 lower half (user), TTBR1_EL1 upper half (kernel)	satp (mode + ASID + root frame)
Process tag	ASID 8/16 bits	ASID (up to 16 bits)
Protection	AP bits, UXN/PXN (no-execute), PAN (≈ SMAP)	R/W/X/U bits, SUM
TLB	Hardware walk; A/D bits optionally in hardware (v8.1); contiguous bit	Hardware walk (in most implementations); `sfence.vma`

Table 8.25: Memory management in ARMv8-A and RISC-V.

Sv39: 3 levels × 512 entries; satp holds the root table's frame and the ASID

Figure 8.59: A RISC-V Sv39 virtual address: three 9-bit indexes (VPN[2], VPN[1], VPN[0]) and a 12-bit offset; the upper 25 bits are copies of bit 38.

Neither architecture has segmentation. The virtual address is flat; translation is done with a multilevel page table, the TLB, and ASIDs. ARMv8's two table base registers correspond to x86-64's canonical address split: user and kernel addresses are translated through separate tables. RISC-V's Sv39 gives a 39-bit, 512 GiB address space (3 × 9 + 12 = 39); Sv48 and Sv57 exist for larger spaces. In short, the same ideas as on x86-64 are implemented under different names.

8.11.3 Separating policy from mechanism

The operating system takes part in paging at four moments. When a process is created, the page table is built. On a context switch the new process's table is activated and the TLB is set up. On a page fault the missing page is brought in. When the process terminates, the frames and the swap space are released.

Some of these jobs depend on the machine (the MMU's table format) and some do not (which page is placed where and when). The Mach kernel made this distinction explicit. The *mechanism* stayed in the kernel: a machine-dependent MMU handler and a machine-independent page-fault handler. The *policy* was given to an external pager running in user space (Figure 8.60). On a page fault, the handler in the kernel asks the external pager for the needed page. The pager reads the page from disk and hands it back. The MMU handler then maps the page into the process's address space.

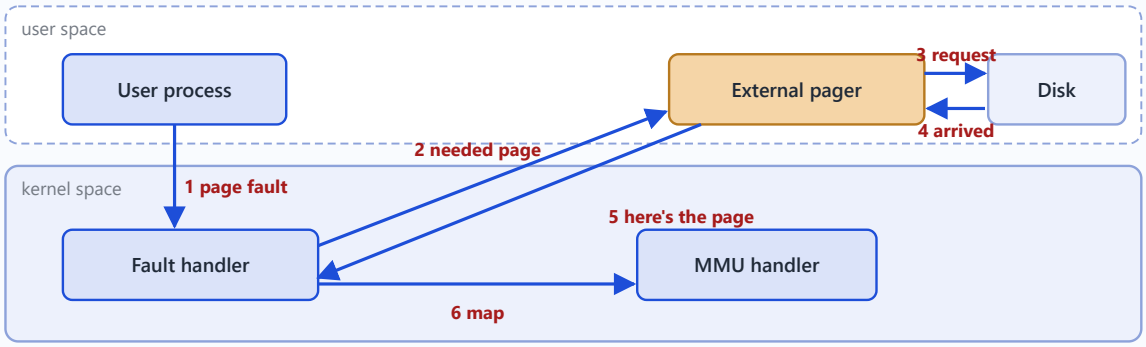


Figure 8.60: A page fault with an external pager: the fault handler in the kernel asks the pager in user space for the page; the MMU handler maps the page.

This design gives flexibility and modularity: different applications can use different paging policies. The price is crossing the user–kernel boundary several times on every page fault. The idea has present-day counterparts too. Linux's `userfaultfd` mechanism forwards page faults to a program in

user space; live migration of virtual machines and freezing a process and resuming it on another machine (CRIU) rest on it. In microkernels such as seL4 and Fuchsia the pager is already in user space.

8.12 Summary

- The CPU reaches only registers and main memory directly. In the **memory hierarchy** hardware manages the caches and the operating system manages main memory. The cache works with 64-byte **lines** and pays off thanks to temporal and spatial **locality**.
- In a **direct-mapped** cache, block b goes to set b mod K; blocks that fall in the same set evict one another (conflict misses). The **average access time** is computed from the bottom up with $T = s + m \cdot T_{\text{below}}$; in the example it is 3.8 ns, about 21 times faster than DRAM.
- The **address space** solves the problems of protection and relocation. With **base and limit** registers the valid addresses are base … base + limit − 1; anything outside raises a trap.
- Address binding can be done at compile, load, or execution time. Today's systems bind at **execution time**: the CPU produces a **logical** address and the **MMU** translates it into a **physical** address.
- **Dynamic loading** (`dlopen`) loads code when it is called. With **dynamic linking** the library code is in memory once and shared; in ELF, what changes on the first call is not the code but the process-specific **GOT** entry. `-fPIC` produces position-independent code.
- Standard **swapping** moves a whole process and is slow; today's systems swap cold pages and compress them first.
- In **contiguous allocation** the hole is chosen by first fit, next fit, best fit, or worst fit. In **external fragmentation** the total free space is enough but not contiguous; by the 50-percent rule one third of memory can be lost. **Internal fragmentation** is the unused part inside a block. **Compaction** merges the holes but is expensive.
- Free space is tracked with a **bitmap** (one bit per unit; looking for k consecutive zeros is linear) or a **linked list** (merging takes constant time if it is sorted by address). `malloc` embeds the free list inside the free chunks.
- In **paging** logical memory is divided into pages and physical memory into **frames**; the address becomes (p, d) and physical = f × page size + d. There is no external fragmentation; internal fragmentation averages half a page. The optimal page size is about $\sqrt{(2se)}$.
- The **page-table entry** holds the P, R/W, U/S, NX, A, D, and G bits along with the frame number. The **valid/invalid** bit catches addresses outside the address space, but protection is at page granularity.
- The **TLB** holds recent translations. The **effective access time** is $\text{EAT} = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$; for $\varepsilon = 20$ and $t = 100$ ns it is 140 ns at $\alpha = 80\%$ and 122 ns at 98%. With multilevel tables a miss is much more expensive.

- The **hierarchical table** keeps lower tables only for the regions in use; x86-64 uses four (48-bit) or five (57-bit) levels with 9 bits per level. **Huge pages** enlarge the TLB reach. The size of a **hashed** table is proportional to the number of pages in use, that of an **inverted** table to physical memory.
- **Segmentation** divides the address space into logical pieces of the form (segment, offset); each segment is a base–limit pair, and it leads to external fragmentation. On x86-64 only the FS/GS bases remain of segmentation; ARMv8 and RISC-V have no segmentation.

8.13 Key terms

memory management cache, cache line hit / miss base and limit registers relocation
logical / physical address memory management unit (MMU) shared library swapping
contiguous allocation internal / external fragmentation compaction bitmap buddy system
paging, page, frame page table, page-table entry valid–invalid bit translation lookaside buffer (TLB)
effective access time hierarchical page table hashed page table inverted page table segmentation

8.14 Test Yourself

1. A direct-mapped cache with 4 sets and 64-byte lines starts out empty, and addresses 16, 80, 40, 272, and 20 are accessed in turn. Which accesses are hits?

   Answer

   The blocks are 0, 1, 0, 4, 0; the sets are 0, 1, 0, 0, 0. 16: miss. 80: miss. 40: block 0 is in set 0, hit. 272: block 4 evicts block 0, miss. 20: block 0 is gone, miss. In total 1 hit and 4 misses.

2. A process's base register is 5000 and its limit register is 2000. The MMU first checks the limit and then adds the base. What happens to logical addresses 1999 and 2000?

   Answer

   1999 < 2000, so it is valid and goes to physical 6999. 2000 equals the limit; since the valid offsets are 0 … 1999, a trap occurs.

3. The holes, in address order, are 300, 120, 500, and 200 K. The requests, in order, are 180, 100, and

450 K. In which of first fit, best fit, and worst fit can a request not be satisfied?

Answer

First fit: 180 → 300 (120 remains), 100 → the 120 remainder of the same hole (20 remains), 450 → 500 (50 remains); all are placed. Best fit: 180 → 200 (20), 100 → 120 (20), 450 → 500 (50); all are placed. Worst fit: 180 → 500 (320), 100 → 320 (220); for 450 the largest hole is 300, so the request cannot be satisfied.

4. The page size is 1 KiB and page 4 is mapped to frame 9. What is the physical address corresponding to logical address 5000?

Answer

$p = \lfloor 5000 / 1024 \rfloor = 4$, $d = 5000 - 4096 = 904$. Physical $= 9 \times 1024 + 904 = 9216 + 904 = 10120$.

5. The TLB lookup takes 5 ns, a memory access 100 ns, the table is single-level, and the hit ratio is 90%. What is the effective access time? What must the hit ratio be for the EAT to be 125 ns?

Answer

A hit takes 105 ns and a miss 205 ns. EAT $= 0.9 \times 105 + 0.1 \times 205 = 94.5 + 20.5 = 115$ ns. From $205 - 100\alpha = 125$, $\alpha = 0.8$, that is, 80%.

6. In 32-bit x86 two-level paging (10 + 10 + 12), what are the top index, the bottom index, and the offset for virtual address 0x00C05123?

Answer

The top index is 0x00C05123 >> 22 = 3, the bottom index is (0x00C05123 >> 12) & 0x3FF = 0xC05 & 0x3FF = 5, and the offset is 0x123. The address lies in the 12–16 MiB region.

7. How much space does an inverted page table occupy with 8 GiB of RAM, 4 KiB frames, and 8-byte entries? Would this value change if the system had 200 processes?

Answer

$2^{33} / 2^{12} = 2^{21} = 2{,}097{,}152$ entries; × 8 bytes = 16 MiB. It would not change: the inverted table is proportional to physical memory and is a single table in the system, independent of the number of processes.

8. In Silberschatz's segment table (segment 2: limit 400, base 4300; segment 4: limit 1000, base

4700), where do addresses (4, 999) and (2, 400) go?

Answer

(4, 999): 999 < 1000, physical 4700 + 999 = 5699. (2, 400): the offset equals the limit; since the valid offsets are 0 … 399, a trap.

9. In a 1024 K buddy system, what size of block does a 70 K request get, and how much internal fragmentation results?

Answer

The smallest power of two that satisfies 70 K is 128 K. The process gets a 128 K block; the internal fragmentation is 128 − 70 = 58 K.

10. In a dynamically linked program, which page changes after `puts` is called for the first time, and why does it matter?

Answer

Only the `puts` entry in the process-specific GOT page changes; the resolver writes the real address there. Because the code pages of the program and of libc do not change, they stay read-only and continue to be shared among all processes.

8.15 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters “Address Translation,” “Segmentation,” “Free-Space Management,” “Paging: Faster Translations (TLBs),” and “Paging: Smaller Tables”: treat segment registers and the TLB with short, runnable simulations. The book is free at ostep.org.
- A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, Chapter 3: treats the relocation problem, bitmaps and linked lists, and the MMU and multilevel-table examples, with the originals of the figures in these notes.
- U. Drepper, “What Every Programmer Should Know About Memory,” 2007: a long paper that shows with measurements the effect of the cache, the TLB, and NUMA on program performance.
- Linux kernel documentation, “Transparent Hugepage Support”: describes how transparent huge pages work, when they pay off, and how they are tuned.

8.16 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 9: Main Memory.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 3: Memory Management (§3.1–3.3, 3.5–3.7).
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Address Spaces", "Address Translation", "Segmentation", "Free-Space Management", "Paging", "TLBs", "Smaller Tables".
4. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, §2.5 Dynamic Storage Allocation (the 50-percent rule).
5. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Vol. 3A, Chapters 4–5 (Paging, Protection); *Arm Architecture Reference Manual for A-profile* (VMSAv8-64); *RISC-V Privileged Specification* (Sv39/48/57).
6. U. Drepper, "How To Write Shared Libraries," 2011; U. Drepper, "What Every Programmer Should Know About Memory," 2007; Linux kernel documentation: mm/, "Transparent Hugepage Support," userfaultfd.

Virtual Memory

A running program does not need all of its code and data at the same time. The error-handling code may never execute, a large array may be touched only in its first rows, and most menu options are never selected. Virtual memory exploits this observation: only the pages a process actually uses stay in physical memory, while the rest wait on disk or are never loaded at all. As a result a process can be larger than physical memory, more processes can run in the same memory, and programs start faster. These notes begin with the hardware and kernel mechanisms that make this work: the valid bit, the page fault, and instruction restart. The cost of a page fault is then measured with the effective access time, and copy-on-write and memory-mapped files are presented as two uses of the same mechanism. The longest part of the notes covers the page-replacement algorithms that decide which page to evict when memory is full. FIFO, OPT, LRU, second chance, clock, NRU, aging, the working set, and WSClock are all run step by step on the same reference string. The final sections cover frame allocation, thrashing, and design decisions such as page size. The buddy and slab allocators that the kernel uses for its own memory, together with today's Linux and Windows solutions, complete the chapter. All traces and numbers were checked by actually running the algorithms.

LEARNING OUTCOMES

Demand paging
Explain the purpose of virtual memory; describe the valid–invalid bit, the steps of page-fault handling, and instruction restart.

Performance
Compute the effective access time from the page-fault probability; interpret the differences among HDD, SSD, and compressed swap.

Copy-on-write and mmap
Trace step by step how copy-on-write works after `fork()` and how memory-mapped files relate to the page cache.

Page replacement
Run FIFO, OPT, LRU, second chance, clock, NRU, aging, working-set, and WSClock by hand on a reference string; explain Belady's anomaly.

Allocation and thrashing
Compare equal and proportional allocation and local and global allocation; explain thrashing in terms of the working set and the page-fault frequency.

Design and real systems
Recognize the design issues of page size, TLB reach, program structure, page pinning, and the buddy and slab allocators, and the approaches of Linux (MGLRU, zswap) and Windows.

9.1 Introduction to virtual memory

In a paged system a process sees its address space as a sequence of fixed-size pages, and each page occupies a frame in physical memory. Classical paging expects all of a process's pages to be in memory. Virtual memory lifts this requirement: some pages may stay in memory and others on disk, and the process still runs. This section first recalls the parts of paging we need, then explains why lifting the requirement is valuable.

9.1.1 Background: paging and the TLB

Every address a process generates is a **virtual address**. The hardware splits it into two parts: the high-order bits form the page number p, and the low-order bits give the position inside the page, the *offset* d. With 4 KB pages the offset is 12 bits. The page table translates p into a frame number f; the offset is carried over to the physical address unchanged. The physical address is thus simply f and d written side by side.

Because the page table itself lives in memory, going to the table on every access would double the memory traffic. The **TLB** (translation lookaside buffer) prevents this. The TLB is a small, very fast cache that holds recently used page-to-frame mappings. Since programs touch only a few pages over short periods, the hit ratio is above 99%. On a miss the hardware walks the page table; on x86-64 this walk has 4 or 5 levels (Figure 9.1).

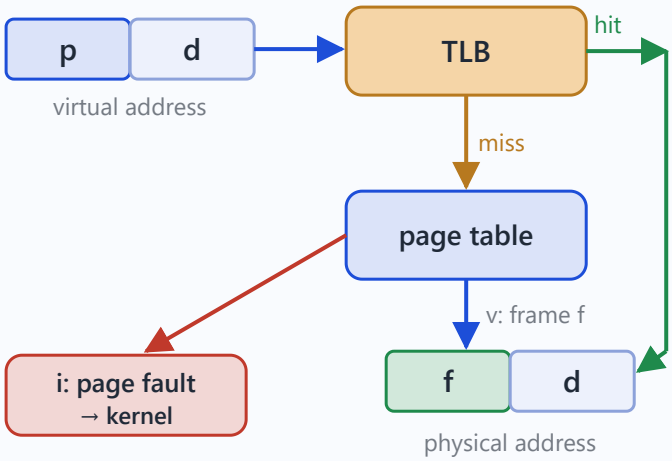


Figure 9.1: Address translation. The virtual address is split into (p, d). The TLB is consulted first; on a miss the page table is walked. If the entry is valid, the frame f and the offset d are combined; if it is invalid, a page-fault trap passes control to the kernel.

Besides the frame number, each page-table entry carries a few bits. The **present** (valid) bit says whether the page is in memory. The **R** (reference; *accessed* on x86) bit is set to 1 by the hardware whenever the page is accessed. The **M** (modified; *dirty* on x86) bit becomes 1 when the page is written. Protection bits specify write permission (R/W), user access (U/S), and the ban on executing code (NX). The R and M bits are the raw material of the replacement algorithms. An access to an entry whose present bit is 0 causes a **page fault**. If the page is not in memory, it is fetched from the backing store; if necessary it is loaded in place of a chosen victim page.

9.1.2 Why virtual memory?

Looking at real programs, we see that a program is rarely used in its entirety. Code that handles error conditions does not run in most executions. Rarely used options and features occupy memory for nothing. Programmers allocate arrays for the worst case: a 100×100 table is often filled only up to 10×10. Even when the whole program is needed, it is not all needed at the same time.

The old solution was **swapping**: a process is either brought into memory as a whole or taken out to disk as a whole. For a large process this is very expensive, because its entire image is moved even if the process is about to use only a few of its pages.

Definition 9.1

Virtual memory is the technique that separates a process's logical memory from physical memory. The process sees a large, contiguous address space, of which only part is in physical memory. The process runs even when it is only partly in memory (Figure 9.2).

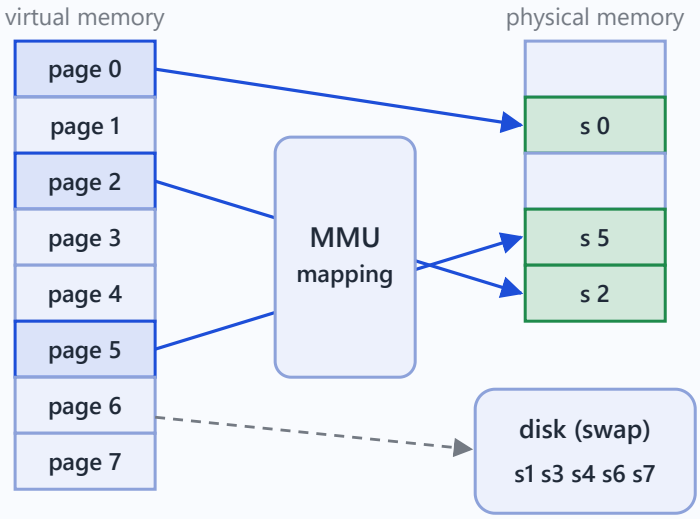


Figure 9.2: Of the eight pages of virtual memory, only pages 0, 2, and 5 are in physical memory. The others stay on disk, in swap space. The MMU (memory management unit) performs the mapping.

The benefits of this separation reach both the programmer and the system:

- A program is not limited by physical memory. On a 16 GB machine, processes whose virtual size exceeds 40 GB can run.
- Each process uses fewer frames, so more processes fit in the same memory. The degree of multiprogramming and CPU utilization rise.
- Less I/O is needed for loading and swapping: only the pages in use are read. Programs start faster.
- Page sharing becomes easier; the same frame can appear in the tables of several processes.

Before virtual memory, programs larger than physical memory were split into pieces by hand. In this older **overlay** method the programmer had to divide the program into parts that replace one another in memory and manage the loading personally. With virtual memory the operating system does the same job automatically, at page granularity.

9.1.3 The virtual address space

The logical view of a process in memory is called its **virtual address space**. The process sees itself at contiguous addresses starting from 0. Physical memory, by contrast, consists of scattered frames, and the MMU maps between the two views.

The layout of the space follows the same logic on all Unix-like systems (Figure 9.3). At the bottom is the code (*text*), above it the initialized data and the zero-initialized bss section. The heap grows upward from there. The stack grows downward from the highest user address. In between lies the area where shared libraries and `mmap` regions are mapped. The remaining unused ranges are **holes**.

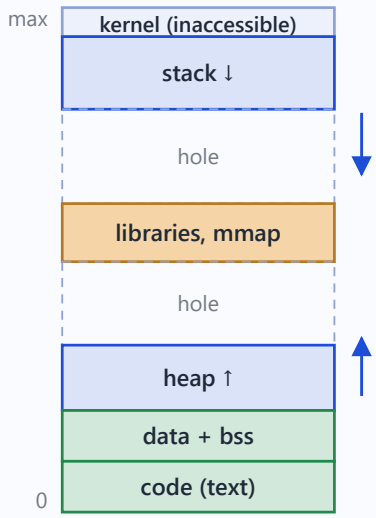


Figure 9.3: The virtual address space. The stack grows from the top down, the heap from the bottom up. Libraries and mmap regions find room in the gap between them; holes consume no pages or frames.

It matters that a hole consumes no frames. No physical memory is allocated for a page until the stack or heap grows into it. Such an address space, mostly unused, is called a **sparse address space**. The page table does not need entries for holes either; in a multilevel table, unused lower-level tables are never allocated.

Today's x86-64 processors use 48-bit virtual addresses; half of that, 128 TiB, is reserved for the user. With 5-level paging the virtual address grows to 57 bits. Address space layout randomization (ASLR) shifts the location of the regions randomly on every run. An attacker therefore cannot know the address of a library or of the stack in advance.

Virtual memory can be built in two ways. The universal one today is **demand paging**. Demand segmentation, which works with variable-size segments, is a historical approach; Burroughs computers and OS/2 used it. On Linux the regions of a process can be seen with `cat /proc/self/maps`. Each line of the output is a **virtual memory area** (VMA): a start and end address, the permissions, and the mapped file, if any.

9.1.4 Shared pages and libraries

A side benefit of virtual memory is sharing. The same physical frame can appear in the page tables of several processes, and each process sees it wherever it likes in its own address space (Figure 9.4).

- **Shared libraries** (libc, libstdc++): the read-only code pages are mapped at their own address in each process, but there is a single physical copy. Even if a hundred processes use libc, the libc code sits in memory once.
- **Interprocess communication through shared memory**: two processes map the same region; what one writes, the other sees immediately.

- `fork()`: when a child process is created, pages are shared, not copied. A copy is made only when one of them writes.

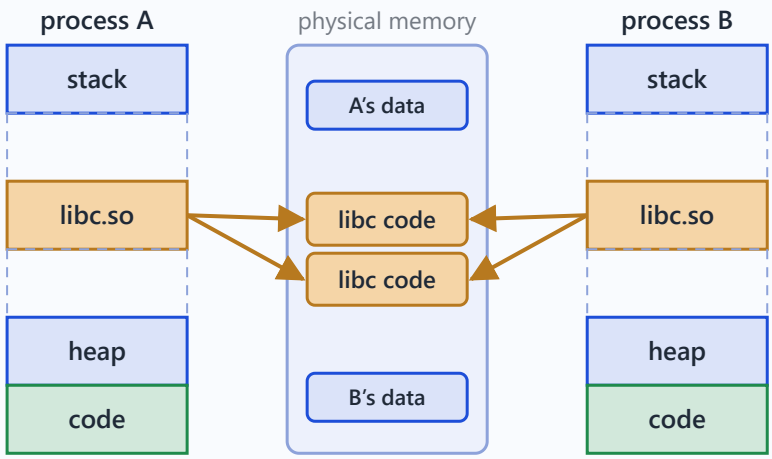


Figure 9.4: Two processes map the libc.so region at different places in their own address spaces, but both refer to the same physical frames. Their data, however, lives in separate frames in each process.

Writable data pages are private to each process. Sharing is safe because code pages are read-only: no process can modify the common copy.

9.2 Demand paging

When a program is loaded from disk there are two options. The first is to read the whole program into memory up front; but if the user will use only a small part of it, this wastes time and memory. The second is to bring pages in only when they are needed. Virtual memory systems follow the second way.

9.2.1 Lazy loading

> **Definition 9.2**
>
> In **demand paging**, no pages (or very few) are loaded when a process starts. A page is brought from disk into memory only at the moment it is accessed. A page that is never accessed is never loaded.

The payoff of this lazy approach is clear: less I/O, less memory, faster start-up, and more users in the same memory. The price is that every first access incurs a delay. An access falls into one of three cases:

1. The address lies in none of the process's regions, so the access is **invalid**; the process is terminated.
2. The address lies in a valid region and the page is in memory; the access proceeds normally.
3. The address is valid but the page is not in memory; the page is brought in and the access is repeated.

The unit that does this job is called the **pager**. Older books also use the name *lazy swapper*, but it is misleading: a swapper moves the entire process, whereas a pager deals with individual pages (Figure 9.5).

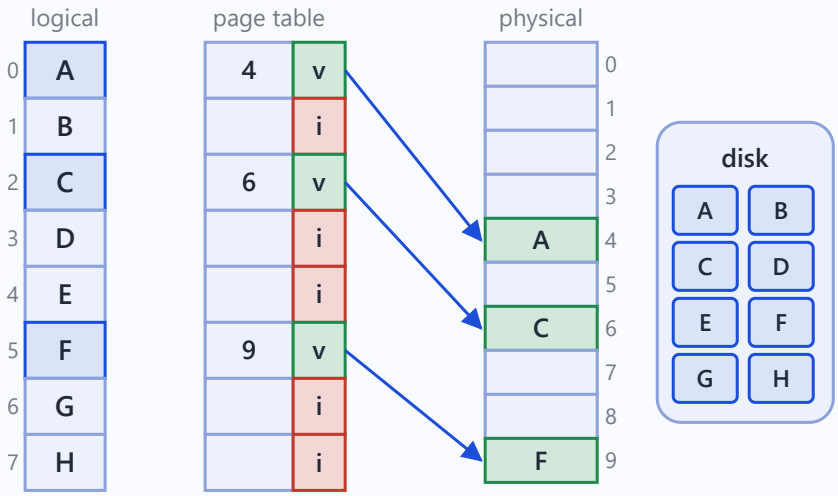


Figure 9.5: Logical memory of eight pages, A to H. In the page table, A (frame 4), C (frame 6), and F (frame 9) are valid. The other entries are invalid; those pages exist only on disk.

The extreme form is called **pure demand paging**. The process is started with no pages in memory, so even the first instruction arrives with a page fault. During the first few hundred microseconds the faults come one after another. But as the process collects the pages it needs, the fault rate drops quickly. What makes this work is the **locality** of programs: over a short time a program works in a small region of its address space. Loops use the same code, arrays use neighboring addresses, and the stack uses its top few pages. Without locality every instruction would cause several page faults and demand paging would be useless.

9.2.2 The valid–invalid bit

Demand paging works with a single bit in the page-table entry.

Definition 9.3

The **valid–invalid bit** is present in every page-table entry. *v* (valid) indicates that the page both belongs to the address space and is in memory. *i* (invalid) indicates that the page is either not in memory or does not belong to the address space at all.

The *i* bit covers two different situations. The hardware does not distinguish between them; it raises a trap in both. The operating system makes the distinction by consulting its own records, namely the VMA list. The example below traces this distinction step by step for a four-page region.

Example 9.1

The process's VMA covers pages 0–3. Physical memory has six frames, numbered 0–5. The process first accesses page 2, then page 2 again, and finally page 6. What happens at each access?

1. **Initially.** All four table entries are set to *i*; no frame has been allocated (Figure 9.6a).
2. **First access to page 2.** The MMU sees *i* in entry 2 and raises a page-fault trap. The kernel verifies that the address is in a valid region (VMA 0–3). It reads the page into free frame 5, sets the entry to (5, v), and re-executes the instruction (Figure 9.6b).
3. **Second access to page 2.** The entry is now valid. The translation is done directly and the mapping is also loaded into the TLB. No trap occurs.
4. **Access to page 6.** The MMU raises a trap again. This time the kernel finds no region in the VMA list that covers page 6. This is an **invalid reference**; the kernel terminates the process with the `SIGSEGV` signal.

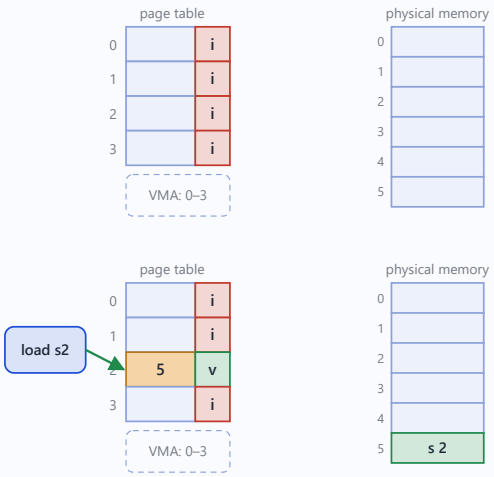


Figure 9.6: (a) Initially all four entries are invalid. (b) After the first access to page 2: the page has been read into frame 5 and the entry is (5, v). An access to page 6, outside VMA 0–3, ends with `SIGSEGV`.

Common mistake

Thinking “invalid bit = bad access.” In demand paging the *i* bit usually means only “this page has not been loaded yet,” and the trap is a perfectly normal event. The process is terminated only if the kernel finds that the address belongs to no region or that the permissions were violated. The line `int *p = NULL; *p = 5;` is an example of the second case: page 0 is deliberately considered part of no region.

9.2.3 Handling a page fault

Definition 9.4

A **page fault** is the trap the MMU raises when a page marked invalid in the page table is accessed. The trap hands control to the page-fault handler in the kernel.

Silberschatz's classic diagram describes the handling of a page fault in six steps (Figure 9.7). The process wants to load a value from address M, and the entry for M's page has the bit *i*.

1. **Reference.** The MMU reads the entry and sees that the page is not in memory.
2. **Trap.** The hardware switches to the kernel. The kernel checks whether the reference is valid; if it is not, it terminates the process.
3. **Page on the backing store.** The location of the page on disk is found; it may be in swap space or in a file. A free frame is chosen.
4. **Fetch.** The disk read is scheduled and the page is read into frame 7. Since the read can take milliseconds, the CPU is given to another process in the meantime.
5. **Update the table.** Frame 7 is written into the entry and the bit is set to *v*.
6. **Restart.** The instruction that caused the fault runs again from the beginning. The process continues as if the page had never been missing.

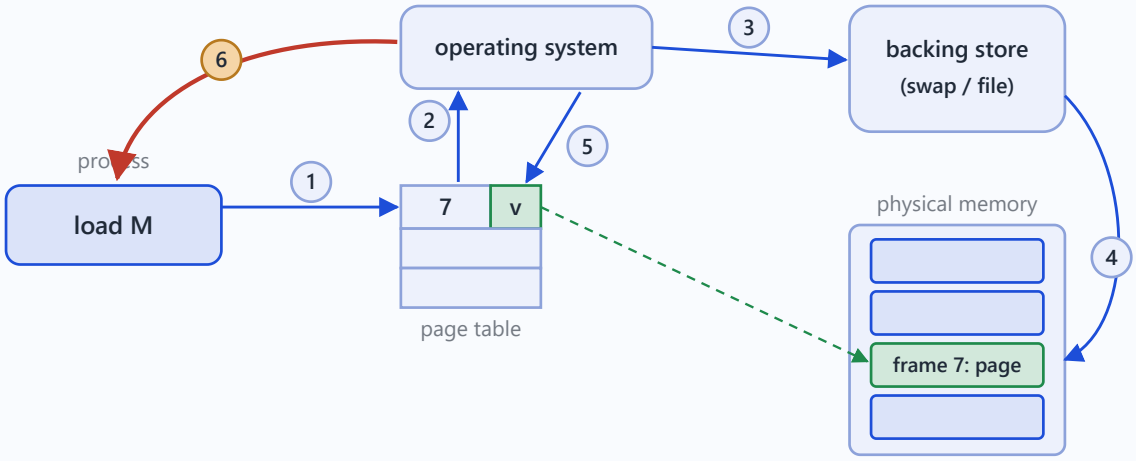


Figure 9.7: Handling a page fault (a redrawing of Silberschatz Figure 10.5). 1 reference, 2 trap, 3 page located on the backing store, 4 page read into frame 7, 5 table updated, 6 instruction restarted.

A page fault is a trap of the *fault* kind. For this kind, the saved program counter points to the instruction that caused the fault, not the next one. This is what makes it possible to re-execute the instruction. x86 processors also write the faulting virtual address into the CR2 register; the handler reads from it which page was requested.

One more detail concerns the free frame. If there is no free frame in the third step, a victim must be chosen. If the victim has been modified (is dirty), it is written to disk first. While the write is in progress the frame is marked “busy” so that another fault does not choose it as a victim too. Choosing the victim is the job of the page-replacement algorithms.

9.2.4 Code trace: address translation and page fault

Address translation is done by hardware, but its logic can be written in a few lines of C. The function `translate()` below translates a virtual address into a physical address with 4 KB pages; on a TLB miss it goes to the page table and, if necessary, calls the page-fault handler.

```
int translate(int va) {
  int p = va >> 12;      /* page no */
  int d = va & 0xFFF;    /* offset */
  int f = tlb_find(p);   /* -1: miss */
  if (f < 0) {
    if (!pt[p].valid)    /* not in memory */
      page_fault(p);     /* from disk */
    f = pt[p].frame;
    tlb_add(p, f);
  }
  pt[p].R = 1;
  return (f << 12) | d;
}
```

The first two lines split the address. `va >> 12` shifts the address right by 12 bits to get the page number; `va & 0xFFF` isolates the low 12 bits, the offset. `tlb_find(p)` looks for the page in the TLB and returns −1 if it is not found. On a miss the `if` block runs: if the entry's valid bit is 0, `page_fault(p)` brings the page from disk and updates the entry. The frame is then read from the table and the mapping is added to the TLB. In every case the page's R bit is set to 1; the replacement algorithms use this bit. The last line shifts the frame number left by 12 bits and combines it with the offset.

Initially the TLB holds page 0 (frame 1) and page 1 (frame 4). Page 2 is in memory (frame 5) but not in the TLB. Page 3 is on disk. The function is called with three addresses in turn (Table 9.1).

Call	p	d	TLB	Table	f	Result
`translate(0x2A10)`	2	0xA10	miss	v, no page fault	5	(5 << 12) | 0xA10 = 0x5A10
`translate(0x2B04)`	2	0xB04	hit	not consulted	5	0x5B04
`translate(0x3008)`	3	0x008	miss	i → page fault, (v, 7)	7	0x7008

Table 9.1: Trace of `translate()`. The TLB starts as {0→1, 1→4}; page 2 is in frame 5, page 3 is on disk; free frame 7.

- **0x2A10.** The high bits are 0x2 and the low 12 bits are 0xA10: p = 2, d = 0xA10. The TLB has no entry for p = 2, so `f = -1`. Since the bit of entry 2 is *v*, the page-fault handler is not called. f = 5 is read from the table and (2, 5) is added to the TLB. The R bit becomes 1; the result is (5 << 12) | 0xA10 = 0x5A10.
- **0x2B04.** Another byte of the same page: p = 2, d = 0xB04. This time the TLB hits, the `if` block is skipped, and the page table is never visited. The result is 0x5B04. Thanks to locality, the vast majority of accesses take this short path.
- **0x3008.** p = 3, d = 0x008. It is not in the TLB. The bit of entry 3 is *i*, so a page fault occurs. The kernel reads the page from disk into free frame 7; the process waits meanwhile. The handler sets the entry to (v, 7). When the instruction re-executes, f = 7 is read and added to the TLB; the result is 0x7008 (Figure 9.8).

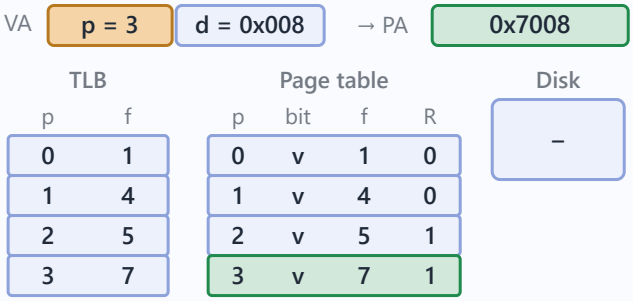


Figure 9.8: After the three calls: the TLB holds four mappings, page 3 is now in frame 7 and valid. The R bits of pages 2 and 3 are 1.

Across the three translations there is 1 TLB hit and 2 misses; one of the misses turned into a page fault, that is, disk I/O. The function itself is O(1). The real cost lies in the table walk on a miss and, above all, in the disk read of the page fault. The two differ by several orders of magnitude.

9.2.5 Timeline of a page fault

Textbooks describe a page fault in fifteen detailed steps. These steps group into three major costs: handling the trap (microseconds), reading the page (milliseconds on an HDD, 10–100 µs on an NVMe SSD), and restarting the process (microseconds). The timeline shows them in the rows of process A, the kernel, the disk, and process B (Figure 9.9).

1. **Steps 1–5.** Trap to the kernel; the user registers and the process state are saved. The kernel recognizes that the interrupt is a page fault and that the reference is valid. The location of the page on disk is found.
2. **Steps 6–9.** The read request is placed in the disk or SSD queue. The device is awaited; on an HDD, seek and rotational latency are added. Meanwhile the CPU is given to another process, B.
3. **Steps 10–13.** The "I/O complete" interrupt arrives. B's state is saved and the interrupt is recognized as coming from the disk. The page table is updated and A is put on the ready queue.
4. **Steps 14–15.** When A gets the CPU again, its registers and page table are loaded; the interrupted instruction runs again from the start.

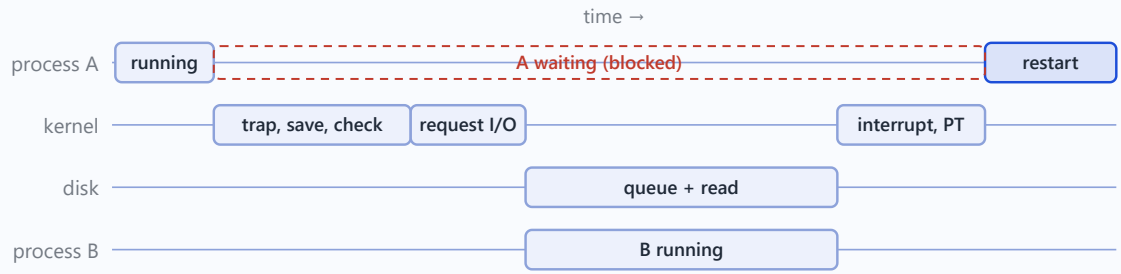


Figure 9.9: Timeline of a page fault. A traps, the kernel requests I/O, and B runs while the disk reads. When the interrupt arrives, the table is updated and A restarts.

The timeline shows one thing clearly: a page fault is a long wait for A but only a context switch for the system. The CPU is not wasted; B makes progress while the read is under way. But from A's point of view, a single memory access has become thousands of times longer than normal. The numerical measure of this difference is the effective access time.

9.2.6 Instruction restart and kinds of faults

The ability to re-execute an instruction after a page fault is the only real feature demand paging asks of the hardware. A fault can occur while the instruction is being fetched or while an operand is being read. In most cases re-executing the instruction from the beginning is enough: the instruction has not

yet changed anything.

The hard case is a fault that strikes after the instruction has already modified something. The IBM 360's MVC instruction moves a block of up to 256 bytes. If the source or destination block crosses a page boundary, the move can fault halfway through. If the blocks overlap, the source has already been corrupted and the instruction cannot simply be re-run. Autoincrement addressing creates a similar problem: the instruction has already incremented the register, so when it re-executes it starts from the wrong address.

On the Motorola 680x0 family, the instruction `MOVE.L 6(A1),2(A0)` is three words long. If a fault occurs while the second operand is being read, the program counter points somewhere inside the instruction. The operating system cannot tell where the instruction began. For this reason some processors copy the program counter into a hidden register before every instruction.

There are two kinds of solutions. The first is to probe both ends of the regions the instruction will touch before it starts; if a page is missing, the fault is taken before the instruction begins. The second is to keep modified values in temporary registers and restore the old state if a fault occurs. In today's processors **precise exceptions** guarantee this in hardware: at the moment of the fault, everything before the faulting instruction appears complete and everything after it appears never to have started.

Linux divides page faults into three kinds according to their consequences (Table 9.2). The distinction matters for performance, because only major faults require disk I/O.

Kind	What happens?
Minor	The page is already in memory: in the page cache, in a shared library, as the zero page, or as a copy-on-write page. Only the table is updated; there is no I/O.
Major	The page must be read from disk or from swap. There is I/O and waiting.
Invalid	There is no region, or a permission was violated. The process receives `SIGSEGV`.

Table 9.2: Kinds of page faults in Linux.

On Linux it is easy to measure how many minor and major faults a program incurs:

```
$ /usr/bin/time -v ./prog | grep -i fault
$ ps -o min_flt,maj_flt -p PID
$ perf stat -e page-faults,major-faults ./prog
```

`/usr/bin/time -v` prints the “Minor” and “Major page faults” lines at the end of the program. `ps` shows the counters of a running process. `perf stat` reads the kernel's event counters. When the same program is run a second time, the major-fault count is usually close to zero: the file pages read in the first run remain in the page cache and are matched by a minor fault.

9.2.7 Where do pages come from?

To fetch the missing page, the page-fault handler must first know where the page comes from. Pages fall into three groups according to their source (Figure 9.10).

- **File-backed pages**: code, read-only data, and files mapped with `mmap`. If the in-memory copy is clean, it is simply discarded; when needed, it is read again from the file.
- **Anonymous pages**: heap, stack, and modified data. There is no file behind them. Before being dropped from memory they must be written to swap space, otherwise their contents would be lost.
- **Zero-fill-on-demand pages**: the first access to the bss section and to new heap pages is satisfied with a zeroed frame. Zeroing is a security requirement: the frame may previously have held another process's data.

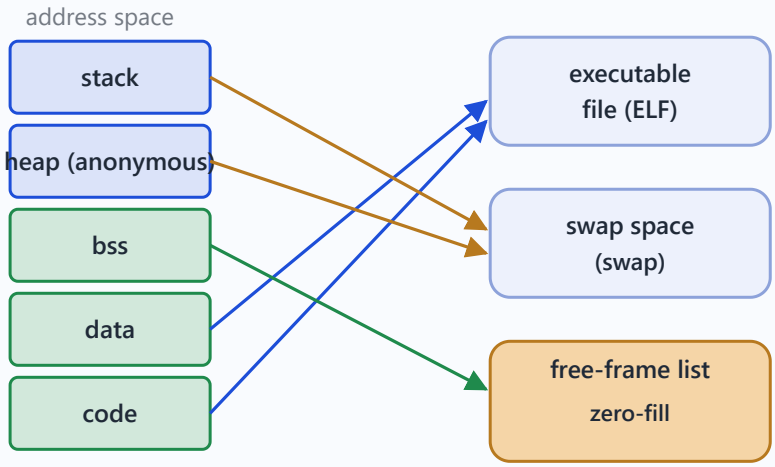


Figure 9.10: The source of pages. Code and data come from the executable file (ELF), the heap and stack from swap space. The bss and new heap pages are supplied zeroed from the free-frame list.

Searching for a free frame at fault time wastes time. The operating system therefore keeps a **free-frame pool**, and the fault handler takes a frame from it. When the pool falls below a threshold, reclamation starts and pages are evicted to refill the pool. Swap space can be a separate disk partition or a file; on Linux, active swap areas are listed with `swapon --show`.

9.2.8 The backing store

Definition 9.5

The **backing store**, or **swap space**, is the area on disk where the pages that are not in memory are kept.

There are two approaches to how space in swap is allocated (Figure 9.11).

- **Static allocation.** When the process starts, contiguous swap space as large as the whole process is set aside and its start address is recorded in the process table. The disk address of page k is computed as "base + k × page size." Separate areas can be reserved for code, data, and stack; the stack, which can grow, then needs special handling.
- **Dynamic allocation.** No space is reserved in advance. When a page is written out to disk, a free slot is found and its address is stored in a per-page **disk map**. Unmodified code pages are never written to swap; when needed they are read again from the executable file.

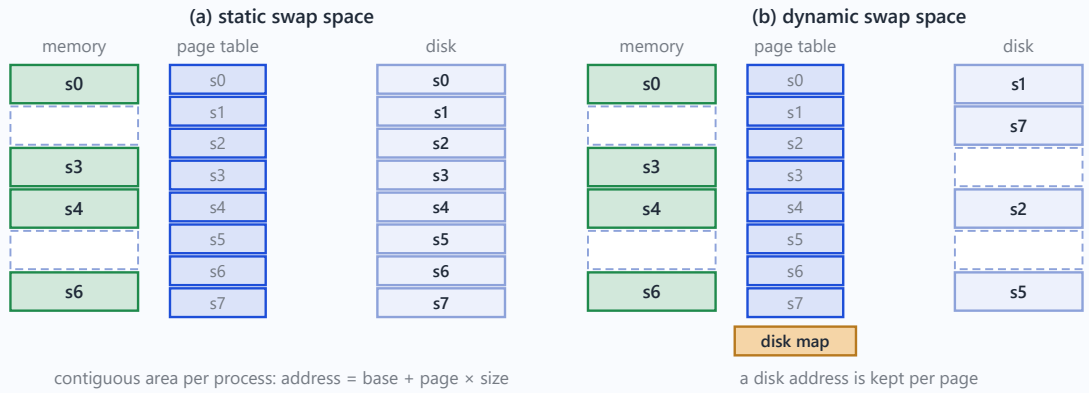


Figure 9.11: (a) With static swap space, every page has a fixed copy on disk. (b) In the dynamic approach, space is allocated when the page goes out to disk and is recorded in the disk map.

The location of swap space can also vary. Linux traditionally uses a separate swap partition, but a swap file in the file system (`swapfile`) works too. Windows writes pages to the file `pagefile.sys` . A separate partition is slightly faster because it skips the file-system layer; a file is flexible because its size is easy to change.

9.3 Performance: effective access time

Demand paging slows memory access on average. Most accesses run at normal speed, but the occasional page fault requires a trip to disk. The weighted average of the two tells how fast the system really is.

Definition 9.6

The **effective access time** (EAT) is the average memory access time that takes the page-fault probability into account. If the probability that an access causes a page fault is p ($0 \le p \le 1$) and the memory access time is ma, then:

$$\text{EAT} = (1 - p) \times ma + p \times \text{page-fault time}$$

The page-fault time is the sum of three parts: handling the trap, reading the page, and restarting the process. The first and last parts take microseconds. The read varies with the type of storage and is what determines the total. If $p = 0$ there are no faults and EAT = ma; if $p = 1$, every access is a fault. This calculation is for faults that fetch the page from disk or swap (*major faults*). If the page is already in memory (for example in the page cache or shared with another process) and only the page table needs updating, the fault is a *minor fault*; it completes in under a microsecond or a few microseconds and is much cheaper.

Example 9.2

Memory access takes 100 ns, and the page-fault service time from a hard disk (HDD) is 25 ms. If one access in a thousand faults ($p = 0.001$), what is the EAT? By what factor does memory slow down?

1. **Units.** 25 ms = 25,000,000 ns.
2. **Simplify.** EAT = $100(1 - p) + 25{,}000{,}000\,p = 100 + 24{,}999{,}900\,p$.
3. **Substitute.** EAT = $100 + 24{,}999{,}900 \times 0.001 = 100 + 24{,}999.9 = 25{,}099.9$ ns ≈ 25 µs.
4. **Ratio.** $25{,}099.9 / 100 \approx 251$. A single fault in a thousand accesses slows memory by a factor of about 251.

The result is striking. 99.9% of the accesses finish in 100 ns, but each of the remaining 0.1% takes 25 ms, so they dominate the average. The real question is therefore asked the other way around: if we accept a given slowdown, how high can the fault rate be?

Example 9.3

In the same system we want the slowdown to be at most 10%. What is the largest possible p?

1. **Goal.** A 10% slowdown means EAT < 110 ns.
2. **Inequality.** $110 > 100 + 24{,}999{,}900\,p \rightarrow 10 > 24{,}999{,}900\,p$.
3. **Solution.** $p < 10 / 24{,}999{,}900 \approx 4.0 \times 10^{-7}$.

4. **Interpretation.** 1 / (4×10^{-7}) = 2,500,000: there must be fewer than one fault per roughly 2.5 million accesses.

That the fault rate can be kept this low is, once again, thanks to locality. But the result also shows that the system has no tolerance for mistakes in its replacement decisions.

As storage technology changes, so does the calculation. As of 2026, reading a page from an NVMe SSD takes 10–100 µs. In Linux's zram mechanism the page stays compressed in RAM; the “read” is a decompression and takes about 10 µs (Table 9.3, Figure 9.12).

Storage	Fault time	EAT (p = 0.001)	Slowdown	p limit for 10%
HDD	25 ms	25,099.9 ns	≈ 251×	$4{,}0 \times 10^{-7}$
NVMe SSD	100 µs	199.9 ns	≈ 2×	$1{,}0 \times 10^{-4}$
zram	10 µs	109.9 ns	≈ %10	$1{,}0 \times 10^{-3}$

Table 9.3: EAT for three storage types with a 100 ns memory access.

The NVMe row is computed as follows: 0.999 × 100 + 0.001 × 100,000 = 99.9 + 100 = 199.9 ns. The 10% limit comes out as 10 / 99,900 ≈ 1.0×10^{-4}. For zram, 99.9 + 0.001 × 10,000 = 109.9 ns.

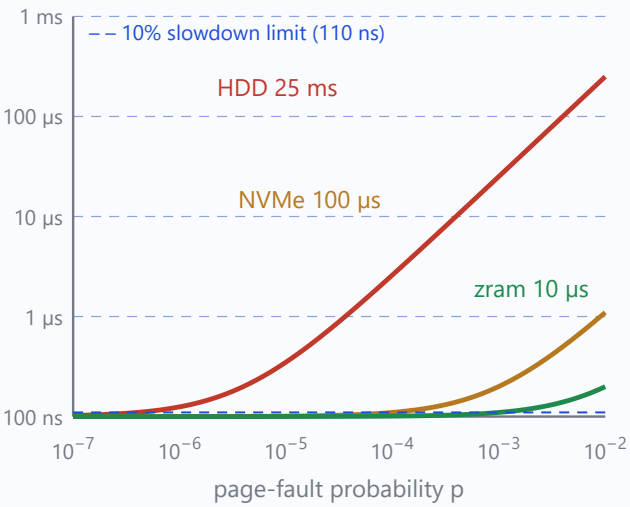


Figure 9.12: Effective access time as a function of the page-fault probability (both axes logarithmic). Curves for HDD, NVMe, and zram; the dashed line is the 10% slowdown limit (110 ns).

Fast storage shrinks the fault penalty by two or three orders of magnitude: the HDD's 25 ms is 250 times NVMe's 100 µs and 2,500 times zram's 10 µs. Even so, memory is still hundreds of times faster, and the fault rate continues to determine the EAT. This is why replacement algorithms remain important today.

The 10th edition of Silberschatz does the same calculation with 200 ns memory and an 8 ms fault time. For $p = 0.001$, EAT = 200 × 0.999 + 8,000,000 × 0.001 = 199.8 + 8,000 = 8,199.8 ns ≈ 8.2 µs. This is a slowdown of 8,199.8 / 200 ≈ 41. The 10% limit follows from the inequality 220 > 200 + 7,999,800 p, giving $p < 2.5 \times 10^{-6}$.

Common mistake

Calculating without matching units. If 100 ns and 25 ms are added in the same formula without converting ms to ns, the result is meaningless. The conclusion "0.1% faults, hence 0.1% slowdown" for $p = 0.001$ is also wrong. The slowdown is determined not by the fault rate alone but by the product of the fault rate and the fault penalty.

9.4 Copy-on-write

`fork()` creates a copy of the parent. In its naive form all of the parent's pages would have to be copied. Yet most children call `exec()` right after being created and throw all of those copies away. Virtual memory provides a way to avoid this waste.

9.4.1 fork() and shared pages

Definition 9.7

In **copy-on-write** (COW), `fork()` does not copy the parent's pages. The parent and the child share the same physical pages; writable pages are marked read-only and COW in both. When either side writes to such a page, only that page is copied.

The mechanism rests on the protection bits. A write attempt to a read-only page causes a **protection fault**. When the kernel sees the COW mark in the entry, it understands that this is not a real violation. It takes a free frame, copies the page's contents into it, and maps the copy as writable for the writer. The write instruction then runs again (Figure 9.13).

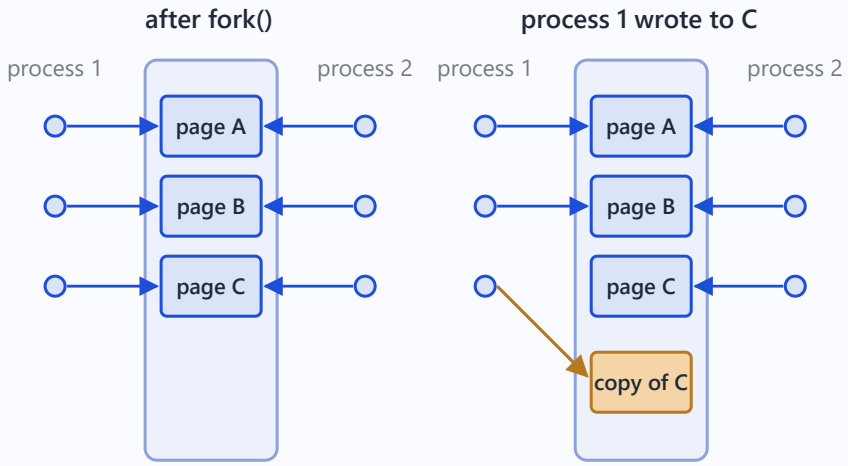


Figure 9.13: Copy-on-write. On the left, after `fork()`: the two processes share pages A, B, and C. On the right, after process 1 writes to C: only a copy of C has been made, and A and B remain shared.

Pages that are never modified, such as code and read-only data, stay shared as long as both processes live. This is why calling `exec()` after `fork()` is almost free: only the page tables are copied, and `exec()` discards them too. The unit of sharing is neither a file nor a region but individual pages; only the page that is written gets copied.

The frame needed for the copy is taken from the pool of zero-filled pages. There is also `vfork()`. This call does not even copy the page table: the child uses the parent's address space directly, and the parent waits until the child calls `exec()` or `_exit()`. If the child modifies a variable in the meantime, the parent sees the change too; this is why `vfork()` is dangerous. Today `posix_spawn()` is preferred for starting a new program.

To manage sharing correctly, the kernel keeps a **reference count** for each frame: how many page tables map the frame. On a protection fault, if the reference count is greater than 1, a copy is needed. If the count is 1, the page now belongs to this process alone; no copy is needed and the page is simply made writable.

9.4.2 Code trace: fork() and COW

The program below uses a global variable (data page D) and a heap buffer (heap page H). The child reads and modifies the variable, and the parent writes to the buffer.

```
int x = 1;                  /* data: D */
int main(void) {
  char *b = malloc(4096);   /* heap: H */
  b[0] = 'a';
  pid_t pid = fork();
  if (pid == 0) {           /* child */
    printf("%d\n", x);     /* read */
    x = 2;                  /* write */
    _exit(0);
  }
  wait(NULL);
  b[0] = 'b';               /* parent */
  return 0;
}
```

The trace follows the page tables of the two processes and the reference counts of the frames step by step (Table 9.4).

Step	Parent D, H	Child D, H	ref(f1), ref(f2)	Event
`b[0] = 'a'`	f1 RW, f2 RW	—	1, 1	only the parent exists
`fork()`	f1 RO, f2 RO	f1 RO, f2 RO	2, 2	only the tables are copied
child `printf`	same	same	2, 2	read allowed, no fault
child `x = 2`	f1 RO, f2 RO	f3 RW, f2 RO	1, 2	protection fault; D is copied to f3
child `_exit(0)`	f1 RO, f2 RO	—	1, 1	f3 is freed
parent `b[0] = 'b'`	f1 RO, f2 RW	—	1, 1	fault; ref = 1, no copy

Table 9.4: Trace of `fork()` and COW. f1 holds D and f2 holds H; f3 is free.

- **Initially.** The parent uses pages D and H as writable (RW); `b[0] = 'a'` has already loaded H.
- `fork()`**.** The kernel copies only the page tables. D and H are marked read-only (RO) on both sides, and the reference counts of f1 and f2 become 2.
- **The child reads** `x`**.** Reading is allowed, so there is neither a fault nor a copy. 1 is printed.
- **The child writes** `x = 2`**.** Since D is read-only, a protection fault occurs. The kernel recognizes COW: it allocates f3, copies the contents of D into it, and maps it RW for the child. The reference count of f1 drops to 1. The parent's `x` is still 1.
- **The child exits.** Its mappings are removed, f3 returns to the free list, and the reference count of f2 drops to 1.
- **The parent writes to H.** A fault occurs because H is still marked read-only. But the reference count of f2 is 1, so no copy is needed; the page is simply made writable.

Over the whole program a single page was copied. The parent's page D is still marked RO. If the parent writes to D, one more fault occurs, but that too is resolved without a copy because the reference count is 1. The price of the method is one trap and one page copy for every shared page that

is written. Its gain is that all the pages that are never written are never copied.

Example 9.4

After `fork()`, the parent and the child share 100 pages. The child writes to 3 of them and then calls `exec()`. How many pages are copied?

1. **Pages written.** Each write by the child hits a page whose reference count is 2: three protection faults, three copies.
2. **Pages read or never touched.** They produce no copies.
3. `exec()`**.** It discards all of the child's mappings, copies included. The reference counts of the parent's pages drop to 1.
4. **Result.** 3 pages are copied. A naive `fork()` would have copied 100 pages.

Common mistake

Assuming that every write produces a copy. What decides the copy is the current reference count of the frame being written. If, after the child has copied page A, the parent writes to A, the parent's frame now belongs to it alone and no copy is needed.

9.5 Memory-mapped files

The usual way to read a file is through `read()` and `write()` calls. Each call is a system call and copies data between a kernel buffer and a user buffer. Virtual memory opens a second way: map the file into the address space and use it like memory.

9.5.1 mmap() and mapping types

Definition 9.8

A **memory-mapped file** is a file whose pages are mapped into a region of a process's address space. In POSIX the mapping is established by `mmap()`. From then on the file is read and written not with read and write calls but with ordinary memory accesses.

The mapping itself reads nothing; it only creates a VMA. Pages are brought in on demand: the first access to each page produces a page fault. If the page is already in the page cache the fault is minor, otherwise it is major. Written pages become dirty and are written back to the file in the background or by a call to `msync()` or `munmap()` (Figure 9.14).

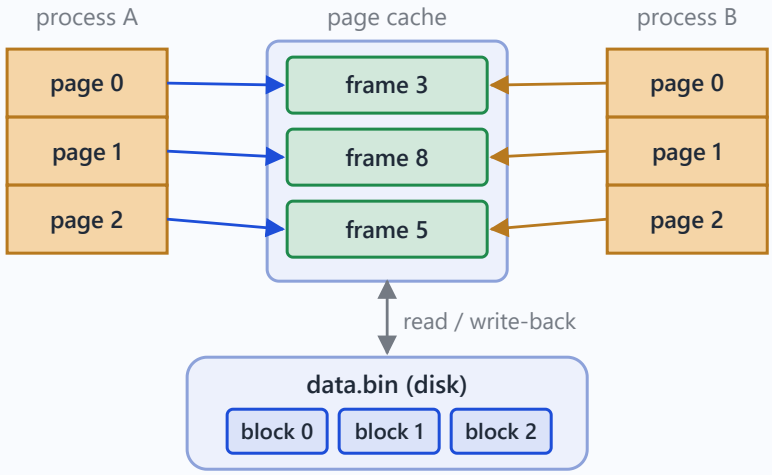


Figure 9.14: Two processes map the same file. The virtual pages refer to the same frames in the page cache; the frames are read from the file blocks on disk and written back when needed.

The flags of `mmap()` determine how the mapping behaves:

- `MAP_SHARED`: changes are visible in the file and to everyone who maps the same file. Two processes can set up shared memory and communicate this way.
- `MAP_PRIVATE`: writes go to a private copy through copy-on-write, and the file does not change. The data section of executables is loaded this way: the program takes its initial values from the file and keeps its changes to itself.
- `MAP_ANONYMOUS`: a zero-filled region with no file behind it. `malloc()` allocates large blocks this way.

On Windows the same job is done by a mapping object created with `CreateFileMapping` and the `MapViewOfFile` call.

The advantage of memory mapping is that the copy disappears. The frame in the page cache is mapped directly into the process's address space; there is no second copy from the kernel buffer to the user buffer. The system-call cost is also eliminated for small, frequently read pieces. The price is that every new page produces a page fault and that I/O errors arrive as a signal (`SIGBUS`).

9.5.2 Code trace: mmap

The program below maps a 12 KiB file as shared and writable, reads one byte, writes one byte, flushes the change to disk, and unmaps the file.

```
int fd = open("data.bin", O_RDWR);
struct stat st;
fstat(fd, &st);          /* 12 KiB = 3 pages */
char *p = mmap(NULL, st.st_size,
     PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
close(fd);               /* mapping stays */
char c = p[0];           /* page 0 */
p[5000] = 'x';           /* page 1 */
msync(p, st.st_size, MS_SYNC);
munmap(p, st.st_size);
```

The trace follows the page-table entries and the page cache together (Table 9.5).

Line	p0	p1	p2	Page cache	Event
`open` , `fstat`	—	—	—	empty	size 12 KiB = 3 pages
`mmap`	empty	empty	empty	empty	only the VMA is set up
`close(fd)`	empty	empty	empty	empty	mapping remains
`c = p[0]`	v	empty	empty	block 0	fault, block 0 is read
`p[5000] = 'x'`	v	v, M = 1	empty	block 0, block 1	fault, read, write
`msync`	v	v, M = 0	empty	block 0, block 1	block 1 is written to the file
`munmap`	—	—	—	block 0, block 1	VMA is removed

Table 9.5: mmap trace; the file has 3 pages (4 KiB each, p0, p1, p2).

- **Open.** The file is opened, and `fstat` reports that the size is 12 KiB, that is, 3 pages.
- `mmap` **.** Only a VMA is created; the table entries are empty and no page has been read.
- `close(fd)` **.** It does not break the mapping, because the VMA holds its own reference to the file.
- `p[0]` **.** The first access to page 0 causes a page fault. Block 0 is read into the page cache (a major fault) and the entry becomes valid.
- `p[5000]` **.** 5000 / 4096 = 1 (integer division), so page 1; the offset is 5000 − 4096 = 904. Again there is a fault and a read, and then the write makes the page dirty (M = 1).
- `msync` **.** With the `MS_SYNC` flag the dirty page is written to the file, and the call does not return until the write finishes; M is cleared. Page 2 was never touched, so it was never read.
- `munmap` **.** The VMA is removed. The pages may stay in the page cache; if another process opens the file, it uses them with a minor fault.

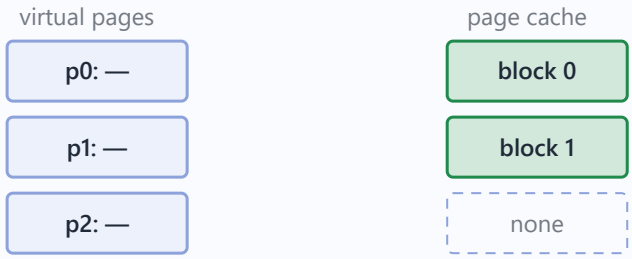


Figure 9.15: After `munmap`: the mapping of the virtual pages is gone, but block 0 and block 1 remain in the page cache. Block 2 was never read.

Common mistake

Believing that data written through `MAP_PRIVATE` goes to the file at `munmap` time. In a private mapping the first write produces a protection fault and the kernel makes a private copy of the page. The change exists only in that process's address space and is lost at `munmap`. Another process that has mapped the same file with `MAP_SHARED` never sees this write either.

9.6 Page replacement

In the description of demand paging we assumed that a free frame could be found at every page fault. In real systems memory is usually full. The reason is not waste but the opposite: idle RAM is wasted RAM. The operating system uses free memory as a page cache and takes it back only when needed. A page must therefore usually be evicted at fault time.

9.6.1 When there is no free frame

With no free frame, the operating system has three options. Terminating the process is unacceptable: virtual memory must be invisible to the user. Swapping out an entire process is possible, but it is a heavy measure and is used under severe overload. The usual way is to replace a single page.

Definition 9.9

Page replacement means that, when there is no free frame, a page in memory (the **victim**) is chosen and evicted, and the requested page is loaded into its frame. The rule that chooses the victim is called the **page-replacement algorithm**.

Replacement adds five steps to the page-fault handler (Figure 9.16):

1. The replacement algorithm chooses a victim frame.

2. If the victim is dirty (M = 1), it is written to swap space or to its file. If it is clean, no write is needed; the copy on disk is already up to date. This halves the I/O.
3. The victim's entry is made invalid and its mapping is removed from the TLB. On a multicore system the TLBs of the other cores must be flushed too; this is called a **TLB shootdown**.
4. The requested page is read into the freed frame.
5. The entry of the new page is made valid and the instruction that caused the trap is restarted.

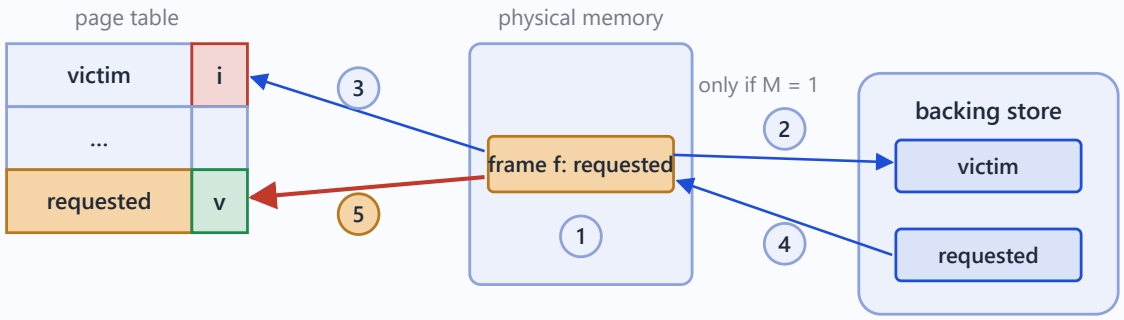


Figure 9.16: The steps of page replacement: the victim is chosen, written to disk if dirty, its entry is made invalid, the requested page is read into the freed frame, and the new entry is made valid.

The M bit gives an important saving here. A dirty victim costs two disk operations: one write and one read. A clean victim costs only the read. For this reason many algorithms prefer the clean one of two equally old pages.

A virtual memory system must answer two separate questions. **Frame allocation**: how many frames will each process get? **Page replacement**: which page will be evicted? This section deals with the second question. The goal of both is the same: to minimize the number of page faults and to keep the same page from moving back and forth between memory and disk again and again.

9.6.2 Evaluating the algorithms

To evaluate an algorithm we run it on a sequence of memory accesses and count the page faults. Real address sequences contain millions of items, but the algorithm's decision depends only on the page number. The sequence can therefore be shortened.

Definition 9.10

A **reference string** is the sequence of page numbers a process accesses. The string is derived from an address trace: each address is converted to a page number, and consecutive references to the same page are counted once. This reduction does not change the result, because an access to a page immediately after the previous one never causes a fault.

Example 9.5

The page size is 100 bytes. The address trace goes 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, ... What is the reference string?

1. **Page numbers.** It is enough to divide the address by 100: 1, 4, 1, 6, 1, 1, 1, 1, 6, 1, ...
2. **Dropping consecutive repeats.** 0102, 0103, 0104, 0101 are all on page 1 and count as a single reference.
3. **Result.** Silberschatz's entire trace reduces to the string 1 4 1 6 1 6 1 6 1 6 1.

In these notes the algorithms are run on Silberschatz's common string with 3 frames:

> 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

The string has 20 references and 6 distinct pages (7, 0, 1, 2, 3, 4). The first load of each distinct page is an unavoidable fault; no algorithm can therefore go below 6. As the number of frames grows, the number of faults is expected to fall (Figure 9.17). When the number of frames reaches 6, all pages fit and only the initial loads remain.

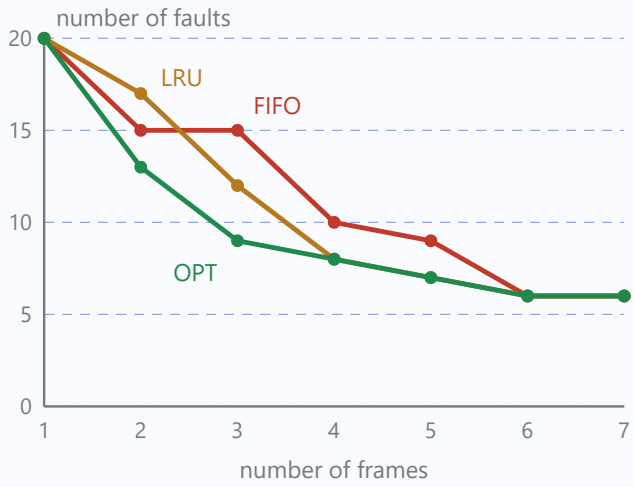


Figure 9.17: Page faults versus number of frames on the common string. For 1 to 7 frames, FIFO incurs 20, 15, 15, 10, 9, 6, 6 faults; LRU 20, 17, 12, 8, 7, 6, 6; OPT 20, 13, 9, 8, 7, 6, 6.

The values on the curves were checked by running the algorithms. It is striking that FIFO stays the same (15) at 2 and 3 frames: adding a frame gains nothing on this string. The algorithms to be run in this chapter are OPT, FIFO, LRU, second chance, clock, NRU, NFU and aging, the working set, and WSClock.

9.6.3 FIFO

Definition 9.11

In the **FIFO** (first-in, first-out) algorithm, pages are kept in a queue in the order they were loaded. A newly loaded page is appended to the end of the queue. When room is needed, the page at the head of the queue, the one loaded earliest, is evicted.

FIFO is very easy to implement and requires no hardware bits. Its problem is that it never considers when a page was *used*. The oldest page may be the most heavily used one: if a variable loaded at the beginning is used throughout the run, FIFO still evicts it at the first opportunity.

The trace on the common string is as follows (Figure 9.18). The first three references (7, 0, 1) are loaded into free frames: 3 faults. The queue order is 7, 0, 1.

- **Reference 4 (page 2).** Fault. 7 at the head of the queue is evicted and 2 enters at the end. Queue: 0, 1, 2. Fault count 4.
- **Reference 5 (0).** In memory, a hit. In FIFO a hit does not change the queue.
- **Reference 6 (3).** Fault; 0 is evicted. Queue: 1, 2, 3. Faults 5.
- **Reference 7 (0).** Fault; 1 is evicted. 0 had been evicted one step earlier and is requested right back. Queue: 2, 3, 0. Faults 6.
- **References 8–11 (4, 2, 3, 0).** All four are faults. 2, 3, 0, and 4 are evicted in turn. Each will be needed again in the next steps. Faults 10, queue: 2, 3, 0.
- **References 12, 13 (3, 2).** Hits.
- **References 14, 15 (1, 2).** Fault; 2 and 3 are evicted. Faults 12, queue: 0, 1, 2.
- **References 16, 17 (0, 1).** Hits.
- **References 18–20 (7, 0, 1).** All three are faults; 0, 1, and 2 are evicted. 15 faults in total.

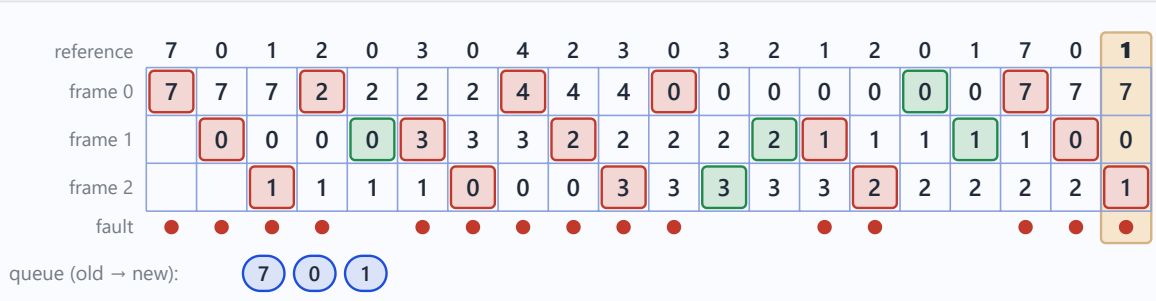


Figure 9.18: FIFO, common string, 3 frames. A red cell marks a page loaded by a fault, a green cell a hit, and the dot in the bottom row a page fault. 15 faults in total: 3 initial loads and 12 replacements.

The order of the victims is 7, 0, 1, 2, 3, 0, 4, 2, 3, 0, 1, 2. This is exactly the load order: FIFO tracks loading, not access. Each step of the algorithm is O(1): whether a page is in memory is determined with a hash table, and the victim is found at the head of the queue.

9.6.4 Belady's anomaly

It seems natural that page faults should decrease as the number of frames grows. In 1969 Belady, Nelson, and Shedler showed that this is not always true for FIFO.

Definition 9.12

Belady's anomaly is the phenomenon that, for some page-replacement algorithms, the number of page faults can increase when the number of frames increases.

Example 9.6

The reference string is 1 2 3 4 1 2 5 1 2 3 4 5. How many faults does FIFO incur with 3 and with 4 frames?

1. **3 frames, references 1–7.** 1, 2, 3 are loaded into the free frames. When 4 arrives, 1 is evicted; when 1 arrives, 2 is evicted; when 2 arrives, 3 is evicted; when 5 arrives, 4 is evicted. All of the first seven references are faults.
2. **3 frames, references 8–12.** Memory holds {5, 1, 2}. 1 and 2 are hits. When 3 arrives, 1 is evicted; when 4 arrives, 2 is evicted. The final 5 is a hit. 9 faults in total.
3. **4 frames, references 1–6.** 1, 2, 3, 4 are loaded into the free frames; 1 and 2 are hits. 4 faults.
4. **4 frames, references 7–12.** When 5 arrives, the oldest page, 1, is evicted. Then the following 1 evicts 2; 2 evicts 3; 3 evicts 4; 4 evicts 5; and 5 evicts 1. All six are faults. 10 faults in total.
5. **Result.** 4 frames incur 10 faults and 3 frames incur 9: more memory, more faults.

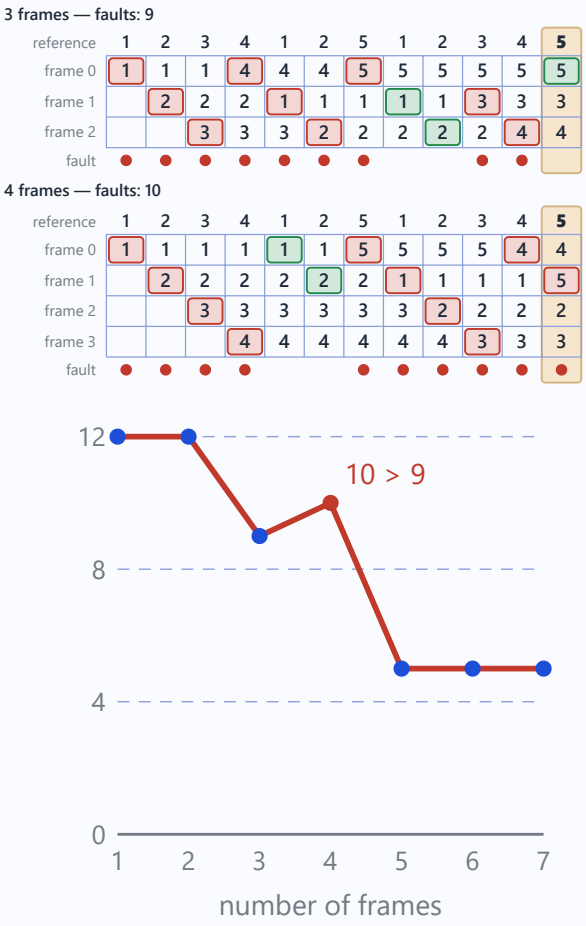


Figure 9.19: Belady's anomaly. On the left, the FIFO tables of the same string with 3 and 4 frames; on the right, the number of faults against the number of frames. 9 faults with 3 frames, 10 with 4.

The cause of the anomaly is that, in FIFO, the memory contents for the two frame counts diverge from each other. After the 7th reference, the 3-frame memory holds {1, 2, 5} and the 4-frame memory holds {2, 3, 4, 5}. Page 1, which is in the small memory, is not in the large one. Because the next references are exactly 1 and 2, the small memory wins.

In some algorithms this divergence never happens.

Theorem 9.1 (Stack algorithms)

Suppose that, at every moment, the set of pages in n frames is a subset of the set in $n + 1$ frames. Such an algorithm is called a **stack algorithm**. Stack algorithms do not exhibit Belady's anomaly: every fault with $n + 1$ frames is also a fault with n frames.

LRU is a stack algorithm. An LRU memory with n frames holds the n most recently used distinct pages. With $n + 1$ frames it holds the $n + 1$ most recently used distinct pages, and the first set is always contained in the second. OPT has the same property. On the same string LRU incurs 10 faults

with 3 frames and 8 with 4 frames. FIFO, on the other hand, is not a stack algorithm; in this example the steps where the sets break the inclusion are the 7th, 8th, and 11th references.

Common mistake

Understanding Belady's anomaly as "FIFO always gets worse as frames are added." The anomaly is seen only on certain strings and between certain frame counts. On the same string FIFO incurs 12, 12, 9, 10, 5 faults with 1, 2, 3, 4, 5 frames; the general trend is still downward. The anomaly says that FIFO *cannot guarantee* this decrease.

9.6.5 OPT

Definition 9.13

The **optimal algorithm** (OPT or MIN) evicts the page that will be used latest in the future, or never. In 1966 Belady showed that this algorithm gives the fewest faults for a given string and number of frames.

OPT is easy to define but impossible to implement: the operating system cannot know which page a process will access in the future. This is why OPT is also called the "crystal ball" algorithm. It is still very valuable because it gives a **lower bound**. How close a new algorithm comes to OPT tells how good it is. In a real system OPT can be computed only by recording a trace and running it a second time; the first run is for learning the future.

The trace on the common string concentrates on the victim choices. At each fault, the next-use times (reference numbers) of the pages in memory are compared (Table 9.6).

Reference no.	Requested	Next uses	Victim
4	2	7 → 18, 0 → 5, 1 → 14	7
6	3	2 → 9, 0 → 7, 1 → 14	1
8	4	2 → 9, 0 → 11, 3 → 10	0
11	0	2 → 13, 4 → none, 3 → 12	4
14	1	2 → 15, 0 → 16, 3 → none	3
18	7	2 → none, 0 → 19, 1 → 20	2

Table 9.6: The replacement moments of OPT. The right-hand column gives the reference number at which each page in memory is used next.

At reference 4, memory holds 7, 0, and 1. 0 is needed right away at the 5th reference, 1 at the 14th; 7 will be needed only at the 18th. The farthest one, 7, is evicted. At reference 11, page 4 will never be used again; if there is such a page, it becomes the victim immediately. Six replacements plus 3 initial

loads give a total of 9 faults (Figure 9.20). This is the best value attainable for this string and 3 frames.

Figure 9.20: OPT, common string, 3 frames. 9 faults in total (3 initial loads, 6 replacements): the lower bound for this string.

9.6.6 LRU

OPT looks into the future. If the future cannot be known, the best guess is the recent past: by the principle of locality, a page that has not been used in the recent past will most likely not be used in the near future either. LRU turns this idea into a rule; in a sense it is OPT looking backward.

Definition 9.14

The **LRU** (least recently used) algorithm evicts the page that has gone unaccessed for the longest time. A last-use time is kept for each page, and the victim is the page whose last use is the oldest.

The trace on the common string starts the same as FIFO, but hits now change the order (Figure 9.21).

- **References 1–3.** 7, 0, 1 are loaded into the free frames. 3 faults.
- **Reference 4 (2).** 7, the least recently used, is evicted. Faults 4.
- **Reference 5 (0).** Hit; 0 is now the most recent page.
- **Reference 6 (3).** Memory holds 1, 2, 0; their last uses are at 3, 4, and 5. 1 is evicted. Faults 5. FIFO had evicted 0 here; LRU knows that 0 was just used.
- **Reference 7 (0).** Hit. LRU wins where FIFO faulted.
- **References 8–11 (4, 2, 3, 0).** All four are faults; 2, 3, 0, and 4 are evicted in turn. Faults 9. In these four steps the string asks every time for the page evicted one step earlier, and LRU does poorly too.
- **References 12, 13 (3, 2).** Hits.
- **Reference 14 (1).** 0 is evicted. Faults 10. **Reference 15 (2)** is a hit.
- **Reference 16 (0).** 3 is evicted. Faults 11. **Reference 17 (1)** is a hit.
- **Reference 18 (7).** 2 is evicted. Faults 12. **References 19, 20 (0, 1)** are hits.

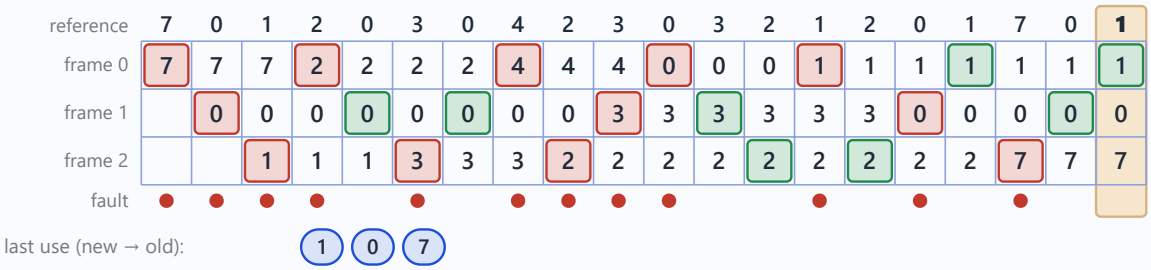


Figure 9.21: LRU, common string, 3 frames. 12 faults in total (9 replacements): better than FIFO (15), worse than OPT (9).

LRU gives very good results in practice and is the target that many algorithms try to approximate. But it requires a record to be updated on every memory access. On a processor making billions of accesses per second, this cannot be done without hardware support. A warning about names: the algorithm that evicts the *most* recently used page is called MRU (most recently used). LRU evicts the *oldest* used one.

9.6.7 Implementing LRU: counter and stack

There are two classical implementations of exact LRU.

Counter (timestamp). A logical clock that advances on every memory access is added to the CPU. On every access to a page, the value of the clock is written into that page's table entry. The victim is the page with the smallest timestamp. In this method the update is simple but frequent: every memory access means a write to the page table. Selection is expensive: the whole table must be scanned at every fault. Clock overflow also needs a solution.

Stack. Page numbers are kept in a stack. When a page is accessed, it is removed from the middle of the stack and put on top. The most recently used page is on top and the page unused for the longest time is at the bottom; the victim is the bottom one. The stack is implemented as a doubly linked list with head and tail pointers. Removing a node from the middle and adding it at the head takes at most six pointer changes. The victim is taken from the tail in O(1) and no search is needed.

Example 9.7

The reference string is 4 7 0 7 1 0 1 2 1 2 7 1 2. How does the LRU stack change at each step? If a replacement were needed at the end of the string, which page would be evicted?

1. **4, 7, 0.** None of the three is on the stack; they are pushed on top in turn. Stack (top to bottom): 0, 7, 4.
2. **7.** It is on the stack; it is removed from the middle and put on top: 7, 0, 4.
3. **1.** New; on top: 1, 7, 0, 4.
4. **0, 1.** Both are moved to the top: first 0, 1, 7, 4; then 1, 0, 7, 4.

5. **2.** New; on top: 2, 1, 0, 7, 4.
6. **1, 2, 7, 1, 2.** All are on the stack and are moved to the top. Final state: 2, 1, 7, 0, 4.
7. **Result.** At the bottom is 4: it has not been used since the first reference. In a replacement it would be evicted.

Figure 9.22: The LRU stack at the end of the string: 2 on top, 4 at the bottom. In a doubly linked list, an access means unlinking the node and adding it at the head; the victim is taken from the tail in O(1).

In the stack method the update is expensive and the selection cheap; in the counter method it is the reverse. Both demand work on every memory access, and neither is found in the hardware of any general-purpose processor today. In practice, algorithms that approximate LRU and rely on the R bit are used.

9.6.8 Implementing LRU: the matrix method

Tanenbaum describes an elegant hardware solution for small numbers of frames. On a machine with n frames, the hardware maintains an $n \times n$ bit matrix. Initially all bits are 0.

Definition 9.15

The LRU matrix method: when frame k is accessed, first all bits of row k are set to 1, then all bits of column k are set to 0. At any moment, the smaller the binary value of a row, the longer that frame has gone unused. The victim is the row with the smallest binary value.

It is easy to see why the method works. An access to k makes row k "newer than everyone": in row k every bit except its own column is 1. At the same time, bit k of every other row becomes 0, which records "I am not newer than k." If bit j of row i is 1, then i was used more recently than j.

Example 9.8

4 frames; access order 0 1 2 3 2 1 0 3 2 3. What are the binary values of the rows after each access and (once the frames are full) the LRU frame?

1. **0.** Row 0 = 1111, then column 0 is cleared: row 0 = 0111 = 7. The other rows are 0. Values 7, 0, 0, 0; the smallest rows are 1, 2, 3 (tied). While free frames exist no victim is chosen; these values only show the order.
2. **1.** Row 1 = 1011 = 11; bit 1 of row 0 is cleared: 0011 = 3. Values 3, 11, 0, 0 → smallest row 2 (free frames exist, no victim is chosen).
3. **2.** Row 2 = 1101 = 13; row 0 = 0001 = 1, row 1 = 1001 = 9. Values 1, 9, 13, 0 → smallest row 3 (free frame exists, no victim is chosen).
4. **3.** Row 3 = 1110 = 14; the last bit of the others is cleared: 0, 8, 12. Values 0, 8, 12, 14. The frames are now full → LRU (the victim) is 0.
5. **Remaining accesses.** 2 → 0, 8, 13, 12 (LRU 0); 1 → 0, 11, 9, 8 (LRU 0); 0 → 7, 3, 1, 0 (LRU 3); 3 → 6, 2, 0, 14 (LRU 2); 2 → 4, 0, 13, 12 (LRU 1); 3 → 4, 0, 12, 14 (LRU 1).

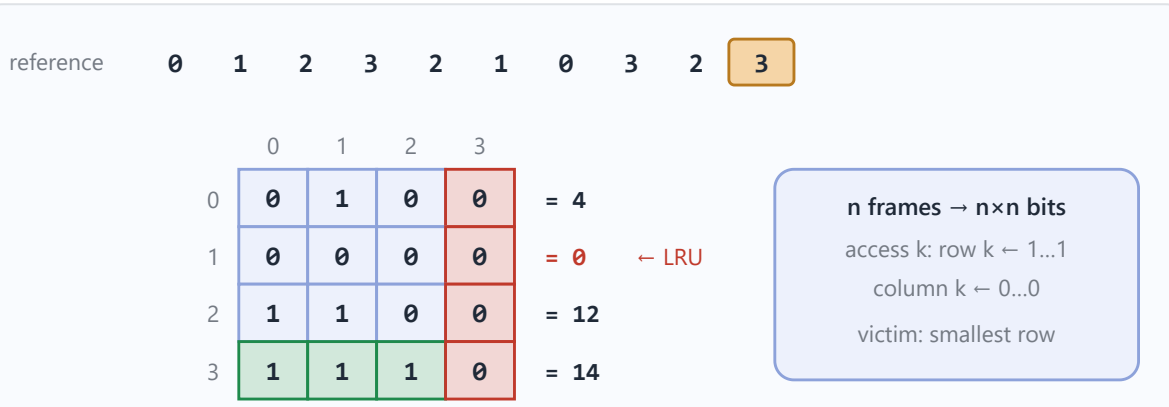


Figure 9.23: The matrix after ten accesses. The row values are 4, 0, 12, 14; since the smallest row is 1, the LRU frame is 1. Indeed, 1 was last used at the sixth access.

To verify the final state it is enough to look at the order of use. The last accesses are, in order, 3 (tenth), 2 (ninth), 0 (seventh), and 1 (sixth). The row values from largest to smallest are 14, 12, 4, 0, that is, 3, 2, 0, 1. The matrix encodes the exact LRU order. Its cost is n^2 bits: for 4 frames 16 bits is no problem, but on a machine with millions of frames the method is unusable.

9.6.9 Second chance

There is no hardware for exact LRU, but nearly all processors keep the R bit. The R bit tells whether the page has been used since it was last cleared. It does not give the order of use, but it answers the question “was it used recently?” FIFO's biggest flaw is precisely that it does not ask this question.

Definition 9.16

The **second-chance** algorithm looks at the R bit of the page at the head of the FIFO queue. If R = 0, the page is both old and unused and is evicted at once. If R = 1, the bit is cleared, the page is moved to the end of the queue as if it had just been loaded, and the search continues.

Example 9.9

In Tanenbaum's example, eight pages A to H are in FIFO order; their load times are 0, 3, 7, 8, 12, 14, 15, 18. The R bits are A = 1, B = 0, C = 1, D = 0, E = 1, F = 0, G = 1, H = 1. A page fault occurs at time 20. Which page is evicted?

1. **A at the head.** R = 1: it was used recently. It is not evicted; its R is set to 0 and it is moved to the end of the queue as if its load time were 20.
2. **New head B.** R = 0: both old and unused. This is the victim.
3. **Result.** B is evicted, and the new page I is added to the end of the queue with R = 1. The queue is now C, D, E, F, G, H, A, I (Figure 9.24).

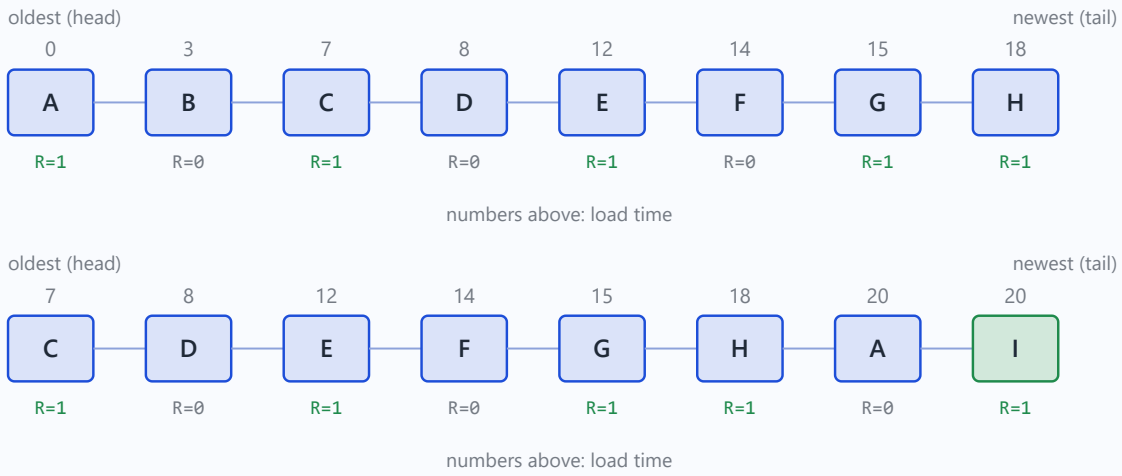


Figure 9.24: Second chance (a redrawing of Tanenbaum Figure 3-15). Above, the FIFO list before the fault at time 20; below, after the fault: A has been moved to the end with a second chance, B has been evicted, and I has been added.

What happens if the R bits of all pages are 1? The algorithm goes around once, clearing every page's R and moving it to the end. When the round ends, the page at the head is again the first page and its R is now 0; it is evicted. In this case second chance behaves like pure FIFO. The algorithm always terminates, because in the worst case after one round every R bit is 0.

The weakness of second chance lies in the implementation: pages are constantly moved in the list. These moves are unnecessary work, and the same decision can be made more cheaply.

9.6.10 The clock algorithm

Definition 9.17

The **clock algorithm** makes the same decision as second chance but does not move pages. The frames are kept in a circular list and a **hand** (pointer) points to the oldest page. At fault time, if the R of the page the hand points to is 0, that page is evicted and the hand advances by one. If R = 1, R is cleared, the hand advances, and the search continues (Figure 9.25).

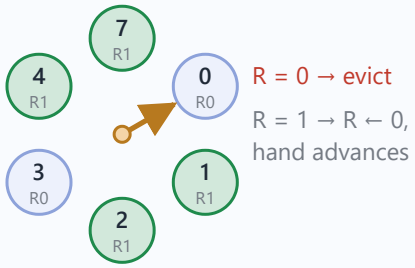


Figure 9.25: The clock algorithm: the frames in a circular list, the hand pointing to the oldest page. R = 0 → evict; R = 1 → R ← 0, the hand advances.

The R bit of a page that is loaded or accessed is set to 1 by the hardware. The hand moves only at fault time; on hits it stays put. The hand circulates not through the page table but through the *frame list*: it walks over the physical frames in order.

On the common string the clock algorithm works as follows. The first three references load 7, 0, 1 into frames 0, 1, 2; all three have R = 1 and the hand points to frame 0 (Table 9.7).

Reference no.	Requested	R cleared	Victim	Fault
4	2	7, 0, 1	7	4
6	3	0	1	5
8	4	2, 0, 3	2	6
9	2	—	0	7
11	0	3, 4, 2	3	8
12	3	—	4	9
14	1	2, 0, 3	2	10
15	2	—	0	11
16	0	—	3	12
18	7	1, 2, 0	1	13
20	1	—	2	14

Table 9.7: The replacement moments of the clock algorithm, common string, 3 frames. “R cleared” lists the pages whose R the hand reset as it passed.

- **Reference 4 (2).** 7, 0, and 1 all have R = 1. The hand clears the R of all three, makes a full circle, and comes back to frame 0, to 7. The R of 7 is now 0, so 7 is evicted. Since all the R bits were 1, the clock behaved like FIFO here.
- **Reference 5 (0).** Hit; the hardware sets the R of 0 to 1, and the hand does not move.
- **Reference 6 (3).** The hand is at frame 1, on page 0. Since R = 1, 0 gets a second chance. The hand moves to page 1 in the next frame; its R is 0, and 1 is evicted. FIFO had evicted 0 here.
- **Reference 9 (2).** Page 0, whose R was cleared in the previous scan and which has not been used since, immediately becomes the victim. The clock's weakness shows here: 0 will be requested again soon, at the 11th reference.
- **Result.** 14 faults in total: one fewer than FIFO (15), two more than LRU (12).

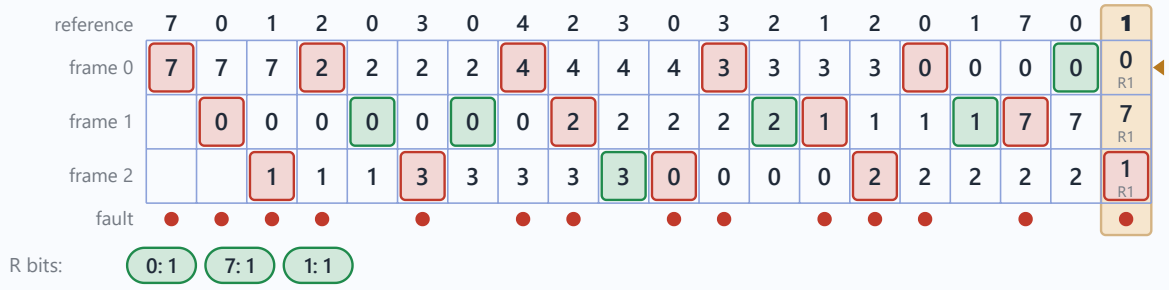


Figure 9.26: The clock algorithm, common string, 3 frames. The cells show the R bit; the orange triangle is the clock hand. 14 faults in total.

The clock algorithm is a rough approximation of LRU: it asks only “has it been used since the last scan?” In return, it does no work at all on a hit, and on a fault it usually finds a victim in a few steps. This cheapness has made it the foundation of real systems.

9.6.11 Code trace: the clock algorithm

The code below implements the clock algorithm together with the M bit. `frm` is the frame array, `hand` is the clock hand, and `N` is the number of frames.

```
int pick_victim(void) {
  while (frm[hand].R == 1) { /* 2nd chance */
    frm[hand].R = 0;
    hand = (hand + 1) % N;
  }
  int k = hand;            /* victim */
  hand = (hand + 1) % N;
  return k;
}
void fault(int p) {
  int k = pick_victim();
  if (frm[k].M) write_back(k); /* dirty */
  frm[k].page = p; frm[k].R = 1;
  frm[k].M = 0;
}
```

The `while` loop in `pick_victim()` runs as long as the R of the frame under the hand is 1: it sets R to 0 and advances the hand by one (`% N` makes the advance circular). The loop ends at the first frame with R = 0. That frame is the victim, and the hand moves on to the next one. If the victim is dirty, `fault()` first writes it to disk; then it places the new page with R = 1, M = 0.

The initial state of the four frames is as follows: frame 0 holds s3 (R = 1, M = 0), frame 1 holds s8 (R = 0, M = 1), frame 2 holds s5 (R = 1, M = 0), frame 3 holds s6 (R = 1, M = 1). The hand is at frame 2. A fault occurs first for page 9, then for page 4 (Table 9.8).

Step	`hand`	Page in frame	R	Decision
fault(9)	2	s5	1	R ← 0, hand to 3
	3	s6	1	R ← 0, hand to 0
	0	s3	1	R ← 0, hand to 1
	1	s8	0	victim k = 1, hand to 2; M = 1 → write to disk; s9 is loaded
fault(4)	2	s5	0	victim k = 2, hand to 3; M = 0 → no write; s4 is loaded

Table 9.8: Trace of `fault(9)` and `fault(4)`. Each row is one step of the loop.

On the first fault the hand makes a full circle. s5, s6, and s3 were used recently and get a second chance. The R of s8 is 0, but its M bit is 1: the copy in memory is not on disk. Before being evicted it is written to disk, which is an extra I/O. s9 is loaded into frame 1 with R = 1, M = 0.

On the second fault the hand is at frame 2. The R of s5 was reset in the previous round and it has not been used since. The victim is found immediately. s5 is clean and the copy on disk is up to date; no write is needed, and a clean victim is cheaper. In the final state the frames are (s3, 0, 0), (s9, 1, 0), (s4, 1, 0), (s6, 0, 1); the hand waits at frame 3 (Figure 9.27). Over the two faults, 3 R bits were reset and 1 disk write was performed.

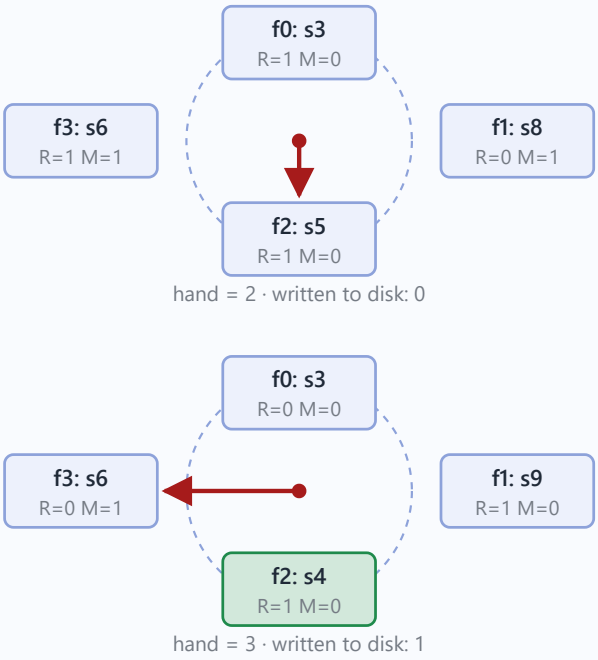


Figure 9.27: The initial and final states of the clock code. At the start the hand is at frame 2; after the two faults s8 and s5 are out, s9 and s4 are in, and the hand is at frame 3. The number of pages written to disk is 1.

In the worst case `pick_victim()` takes $N + 1$ steps: if all R bits are 1, the hand makes a full circle and ends where it started. On average it is much shorter. When memory is very large, even a single hand completes its circle very late; the solution to this problem is the two-handed clock.

9.6.12 NRU

The clock algorithm looks only at the R bit. Yet evicting the clean one of two equally old pages saves a disk write. NRU uses the two bits together.

Definition 9.18

In the **NRU** (not recently used) algorithm, each page has R and M bits. A clock interrupt clears the R bits periodically. Pages fall into four classes: class 0 = (R, M) = (0, 0), class 1 = (0, 1), class 2 = (1, 0), class 3 = (1, 1). At fault time, a page is evicted from the lowest nonempty class.

The class number is 2R + M; the R bit weighs more. This choice is deliberate: evicting a clean page that was used recently is worse than writing out an unused dirty page. At first sight class 1 may look impossible: how can a page be modified without being used? It was modified, but then a clock tick cleared its R. M, on the other hand, is not cleared by the tick, because it tells whether the page has to be written to disk.

The difference between NRU and LRU is one of detail. LRU tracks the exact order of use. NRU asks only two questions: was it used in the last clock interval, and is it dirty? The result is crude but cheap. Which page to choose within a class is undetermined; in these notes the one loaded earliest is chosen on a tie.

Example 9.10

Common string, 3 frames. The 2nd, 6th, 9th, and 15th references are writes (0*, 3*, 2*, 2*). A clock tick arrives every 4 references and clears all the R bits. How many faults and how many dirty writes does NRU incur?

1. **References 1–3.** 7, 0*, 1 are loaded. Because 0 was loaded by a write, M = 1.
2. **Reference 4 (2).** Classes: 7 → 2, 0 → 3, 1 → 2. The lowest class, 2, contains 7 and 1; the one loaded earliest, 7, is evicted. Then the tick: all R bits are 0.
3. **Reference 6 (3*).** Because 0 was just read, R = 1. Classes: 2 → 0, 0 → 3, 1 → 0. The earliest one in class 0, 1, is evicted.
4. **Reference 8 (4).** Classes: 2 → 0, 0 → 3, 3 → 3. 2 is evicted; then the tick.
5. **Reference 9 (2*).** The tick has cleared the R bits: 4 → 0, 0 → 1, 3 → 1. 4 is evicted. 0 and 3 are dirty but unused.
6. **Reference 14 (1).** Classes 2 → 3, 0 → 1, 3 → 1. The earliest one in class 1, 0, is evicted; since it is dirty, it is first written to disk (write 1).
7. **Reference 16 (0).** 2 → 3, 1 → 2, 3 → 1. 3 is evicted and written (write 2); then the tick.
8. **Reference 18 (7).** 2 → 1, 1 → 2, 0 → 0. 0 is evicted; this time it is clean because it came back by a read.
9. **Reference 19 (0).** 2 → 1, 1 → 2, 7 → 2. The dirty 2 is evicted and written (write 3).
10. **Result.** 11 faults, 3 dirty-page writes.

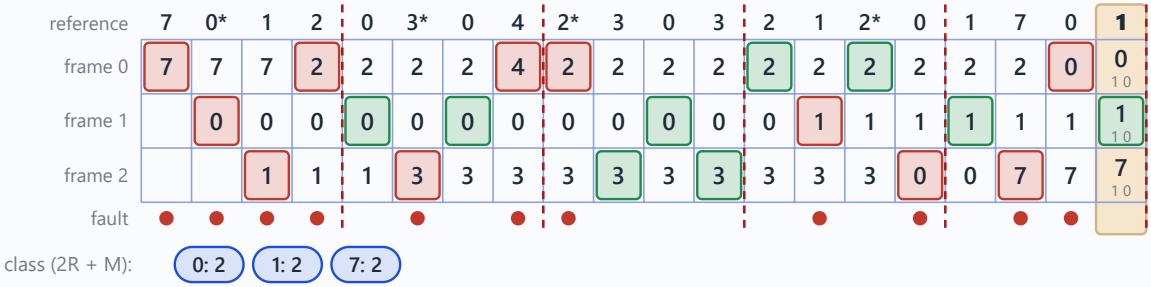


Figure 9.28: NRU, common string, 3 frames. An asterisk marks a write access, the small numbers under the cells are the R and M bits, and the dashed lines are clock ticks. 11 faults in total, 3 dirty writes.

NRU's 11 faults are fewer than LRU's 12. This does not mean that NRU is better than LRU. The result of algorithms based on the R and M bits depends on where the ticks fall and on the tie-breaking rule. On a single short string the share of luck is large.

9.6.13 Enhanced clock

NRU's class idea can be combined with the circular list of the clock algorithm. Silberschatz calls this algorithm *enhanced second chance*; older versions of macOS used it too.

Definition 9.19

The **enhanced clock** algorithm makes two kinds of passes at fault time. *Pass 1*: starting from the hand, look for a page of class (0, 0); the bits are not touched. *Pass 2*: look for a page of class (0, 1); the hand sets the R of every page it passes to 0. If pass 2 also fails, go back to pass 1. Because all R bits are now 0, a victim is certain to be found by the end of the search.

The algorithm has no clock tick; the R bits are cleared only as the hand passes. The trace with the common string and the same writes is as follows:

- **Reference 4 (2).** Memory holds 7 (1, 0), 0 (1, 1), 1 (1, 0) and the hand is at frame 0. Pass 1 finds no (0, 0). Pass 2 finds no (0, 1) either, but as the hand passes it clears all the R bits. In pass 1 again the hand is on 7, and 7 is now (0, 0): 7 is evicted.
- **Reference 6 (3*).** 2 (1, 0), 0 (1, 1), 1 (0, 0); the hand is at frame 1. In pass 1 it passes 0 and finds 1 at (0, 0): 1 is evicted.
- **Reference 8 (4).** Pass 1 and pass 2 both fail; pass 2 clears all the R bits. In pass 1 again, 2 is found at (0, 0): 2 is evicted.
- **Reference 9 (2*).** Memory holds 4 (1, 0), 0 (0, 1), 3 (0, 1). Pass 1 finds no (0, 0). Pass 2 finds 0, under the hand, at (0, 1): 0 is evicted and first written to disk (write 1).
- **Later faults.** At the 11th reference 4 is evicted, at the 14th 0, at the 16th the dirty 3 (write 2), at the 18th the dirty 2 (write 3).
- **Result.** 11 faults, 3 dirty writes (Figure 9.29).

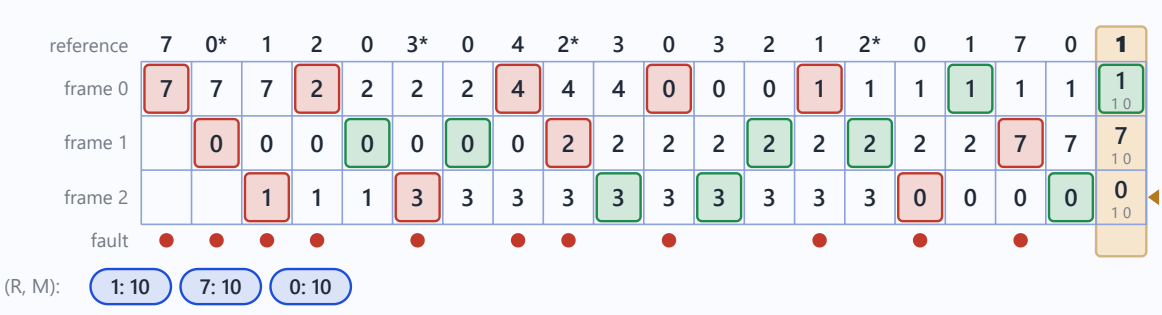


Figure 9.29: Enhanced clock, common string, 3 frames. The cells show the R and M bits, and the triangle is the

clock hand. 11 faults in total, 3 dirty writes.

The result is the same as NRU's, but the victims differ: NRU evicts 7, 1, 2, 4, 0, 3, 0, 2, while the enhanced clock evicts 7, 1, 2, 0, 4, 0, 3, 2. The difference arises at the 9th reference. In NRU a tick has cleared all the R bits and 4 is in class 0. In the enhanced clock, the R of 4 is still 1, and the dirty but unused 0 is chosen. The price is scanning memory several times on one fault.

9.6.14 NFU and aging

The R bit remembers only the last interval. A longer history can be kept in software.

Definition 9.20

In the **NFU** (not frequently used) algorithm, each page has a software counter. At every clock tick, the current R bit is added to the counter (counter += R) and R is cleared. At fault time, the page with the smallest counter is evicted.

NFU's flaw is that it never forgets. A page heavily used in the first phase of the program accumulates a high counter. Even if the program moves to another phase and never touches that page again, its counter stays high and the page continues to stay in memory. The counter records the *number* of uses, not the *time* of use. A small change fixes this flaw.

Definition 9.21

In the **aging** algorithm, at every tick the counter is first shifted right by one bit, and then the R bit is added at the left, that is, at the most significant bit. Thus recent use weighs more heavily in the counter. At fault time, the page with the smallest counter is evicted.

Example 9.11

In Tanenbaum's example there are six pages (0–5) and 8-bit counters; the counters start at 0. The R bits seen at five ticks (for pages 0 to 5) are: tick 0: 1 0 1 0 1 1, tick 1: 1 1 0 0 1 0, tick 2: 1 1 0 1 0 1, tick 3: 1 0 0 0 1 0, tick 4: 0 1 1 0 0 0. What are the final counters, and which page is evicted?

1. **Tick 0.** The counters of the pages with R = 1 become 10000000: pages 0, 2, 4, 5. The others are 00000000.
2. **Tick 1.** First the shift, then R: page 0 → 11000000, 1 → 10000000, 2 → 01000000, 3 → 00000000, 4 → 11000000, 5 → 01000000.
3. **Ticks 2–3.** The same operation: at the end of tick 3, 11110000, 01100000, 00010000, 01000000, 10110000, 01010000.

4. **Tick 4.** 01111000 (120), 10110000 (176), 10001000 (136), 00100000 (32), 01011000 (88), 00101000 (40).
5. **Decision.** The smallest value is 32: page 3 is evicted.

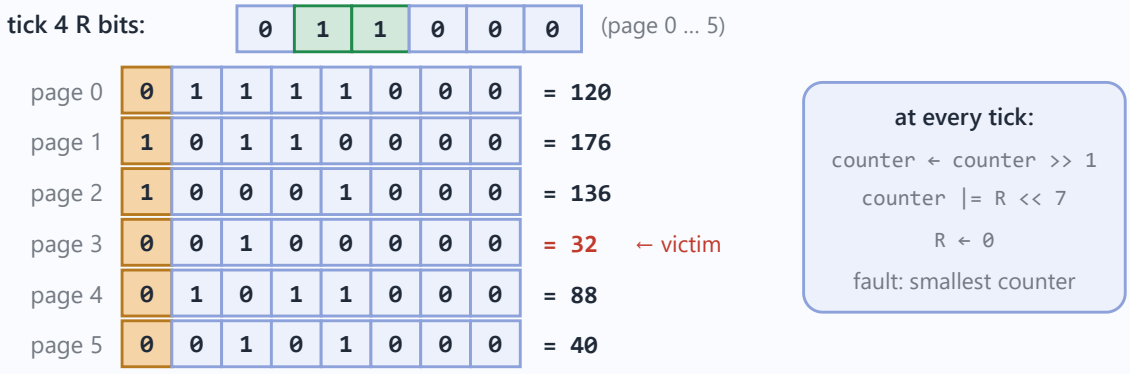


Figure 9.30: Aging (Tanenbaum Figure 3-17; the counter values were recomputed from the R bits). After the fifth tick, the page with the smallest counter is page 3.

Comparing pages 3 and 5 shows the logic of aging. Neither was used in the last two ticks. Page 3 was last used at tick 2 (00100000); page 5 was also used at tick 2, but additionally at tick 0 (00101000). When the recent past is equal, the longer history decides.

Aging approximates LRU well, but it differs in two ways. First, of two pages used in the same tick interval, it cannot tell which was used later; both have 1 in the most significant bit. Second, an 8-bit counter remembers only the last 8 ticks. A page used 9 ticks ago and a page used 1,000 ticks ago have the same counter: 0. If ticks arrive about 20 ms apart, 8 bits mean a history of 160 ms, which is often enough.

Common mistake

Reading the bits of the counter in reverse order. The most recent tick is the most significant bit. For a page whose R bits from tick 0 to tick 4 arrive as 1, 1, 1, 0, 0, the counter is not 11100000 but 00111000 (56): it was not used in the two newest ticks. NFU would have given the same page 3 points and would have evicted a page that was used only twice but within the last two ticks.

9.6.15 Aging on the common string

Running aging on the common string reveals a detail. The ticks arrive every 4 references. A newly loaded page has counter 0 and R = 1. In the comparison, the R bit that has not yet been added to the counter counts as the most significant bit, that is, a 9-bit value (R, counter) is used. Had we not done

this, the newly loaded page would have counter 0 and would be evicted at the very first fault. On a tie, the page loaded earliest is chosen.

Reference no.	Requested	Keys	Victim
4	2	7: 100000000, 0: 100000000, 1: 100000000	7
6	3	2: 010000000, 0: 110000000, 1: 010000000	1
8	4	2: 010000000, 0: 110000000, 3: 100000000	2
9	2	4: 010000000, 0: 011000000, 3: 010000000	3
10	3	4: 010000000, 0: 011000000, 2: 100000000	4
14	1	3: 010000000, 0: 011100000, 2: 110000000	3
18	7	1: 110000000, 0: 011110000, 2: 011000000	2

Table 9.9: Aging, common string, 3 frames, a tick every 4 references. The values are 9-bit (R, counter) keys.

At reference 4 the keys of the three pages are equal and 7, the one loaded earliest, is evicted. Then the first tick arrives and the R bits are shifted into the counters. At reference 6 the keys of 2 and 1 are equal (only their counters contain the use from one tick earlier); 1 is evicted because it was loaded earlier. 0, on the other hand, was used again at the 5th reference, so its R is 1 and it is kept. At references 9 and 10 two faults occur within a short interval: 3 is evicted for 2, and 4 for 3. 10 faults are incurred in total (Figure 9.31).

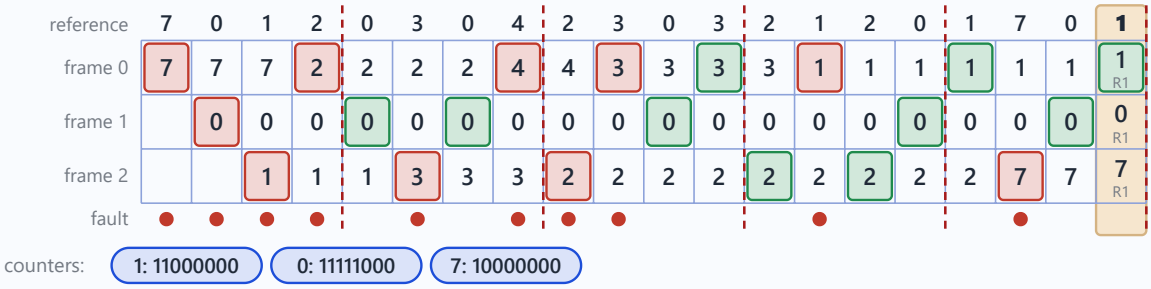


Figure 9.31: Aging, common string, 3 frames, a tick every 4 references (dashed lines). 10 faults in total; final counters 1: 11000000, 0: 11111000, 7: 10000000.

With 10 faults, aging does better than LRU (12) on this string and comes very close to OPT (9). This result, too, should not be generalized. A single short string is not enough to rank algorithms; the result changes when the tick frequency or the tie-breaking rule changes.

9.7 The working set

The algorithms so far answered a single question: which page should be evicted? In 1968 Denning changed the question: for a process to run efficiently, *which pages* does it need to keep in memory? The answer again comes from locality, and it guides both replacement and frame allocation.

9.7.1 The working-set model

A process uses a small group of pages for a while and then moves to another group. For example, when a function is called, its code, its local variables, and the global data it uses form a **locality**. When the function returns, the process moves to another locality. Localities may overlap.

Definition 9.22

The **working set** WS(t, Δ) is the set of distinct pages used in the last Δ memory references ending at time t. Δ is called the **working-set window**.

The choice of Δ is important. If Δ is too small, the window does not cover the whole locality and needed pages fall outside it. If it is too large, several localities overlap and the set becomes larger than necessary. With Δ = ∞ the working set is all the pages the program touches in its lifetime. The essence of the model is this: if the process's working set is in memory, few faults occur; if not, the process thrashes.

Example 9.12

Silberschatz's string (37 references): 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4. The window is Δ = 10 references. What is the working set at different moments?

1. ***t* = 10.** The window is references 1–10: 2 6 1 5 7 7 7 7 5 1 → {1, 2, 5, 6, 7}, size 5. This is moment t_1 in Silberschatz.
2. ***t* = 14.** The window is references 5–14: 7 7 7 7 5 1 6 2 3 4 → {1, 2, 3, 4, 5, 6, 7}, size 7. The old and the new locality overlap.
3. ***t* = 18.** The window is references 9–18 → {1, 2, 3, 4, 5, 6}, size 6.
4. ***t* = 22.** The window is references 13–22 → {1, 2, 3, 4}, size 4.
5. ***t* = 26.** The window is references 17–26: 3 4 4 4 3 4 3 4 4 4 → {3, 4}, size 2. This is Silberschatz's moment t_2; the process has settled into a new locality.
6. ***t* = 30 and 37.** {1, 2, 3, 4} (size 4) and {2, 3, 4} (size 3).

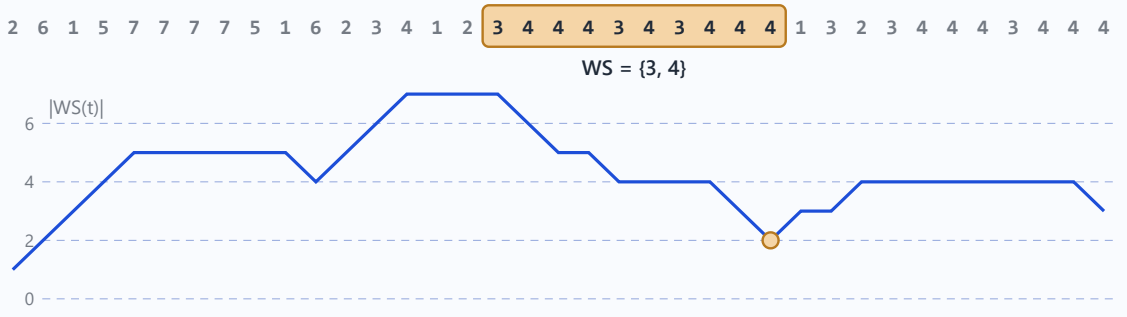


Figure 9.32: The working-set window (Δ = 10) at t = 26: the window contains only 3 and 4. The line below shows how |WS(t)| changes over time.

The size of the working set fluctuates over time: from t = 10 to 37 the sizes go 5, 4, 5, 6, 7, 7, 7, 7, 6, 5, 5, 4, 4, 4, 4, 3, 2, 3, 3, 4, … In the transition from one locality to the next, the set temporarily grows, because the window sees the pages of both localities together. When the new locality settles, the set shrinks.

9.7.2 The working-set algorithm

The exact definition, the set of the last Δ references, would need a register shifted on every memory access. That is far too expensive. The practical approximation uses time instead of reference counts: the working set is the pages used during the last τ units of virtual time. **Virtual time** is the process's own CPU time; it does not advance while the process waits, and it is measured in clock ticks.

Definition 9.23

In the **working-set algorithm**, each page-table entry holds the time of last use and the R bit. At fault time the table is scanned. For a page with R = 1, the time is set to the current virtual time and R is cleared. For a page with R = 0, if its **age** (now − time of last use) is greater than τ, the page is not in the working set and can become the victim.

Example 9.13

Tanenbaum's example: the current virtual time is 2204 and τ = 500. The (time of last use, R) values of the eight pages are: (2084, 1), (2003, 1), (1980, 1), (1213, 0), (2014, 1), (2020, 1), (2032, 1), (1620, 0). Which page is evicted?

1. **Pages with R = 1.** Six pages were used in this tick; their times are set to 2204 and their R bits are cleared.
2. **Pages with R = 0.** The ages are 2204 − 1213 = 991 and 2204 − 1620 = 584. Both are greater than τ = 500, so neither is in the working set.

3. **Choice.** The first suitable page found, the one with time 1213, is evicted. The scan continues, updating the other entries.

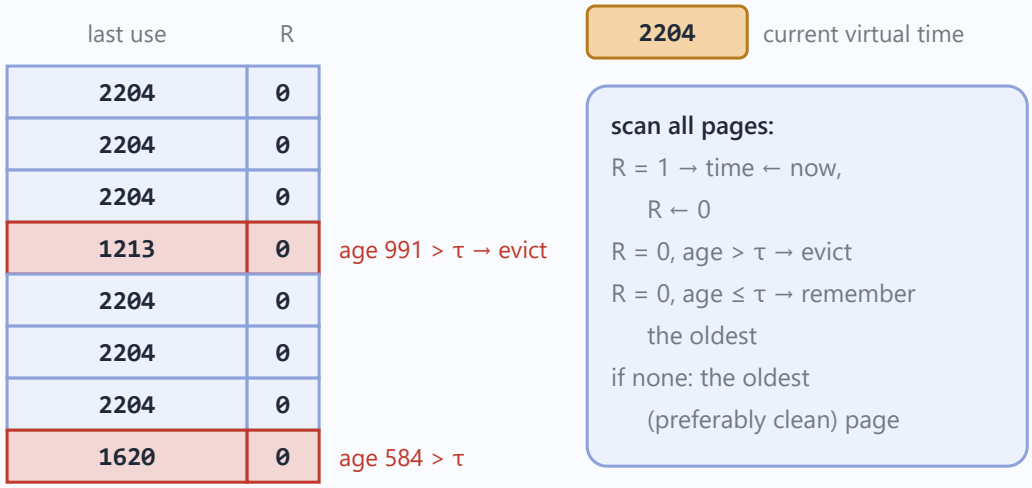


Figure 9.33: The working-set algorithm (a redrawing of Tanenbaum Figure 3-20). At the end of the scan the pages with R = 1 have time 2204; the page with age 991 is evicted.

If all pages are in the working set, that is, all pages with R = 0 have an age smaller than τ, the oldest among the R = 0 pages is chosen. If there is no R = 0 page at all, a random page is chosen, preferably a clean one. The algorithm's drawback is scanning the entire table on every fault.

9.7.3 WSClock

The algorithm that combines the working-set idea with the clock's circular list is Carr and Hennessy's WSClock of 1981. Instead of scanning the whole table, the hand walks over the frames in order and stops at the first suitable victim.

Definition 9.24

In the **WSClock** algorithm each frame has a time of last use and R and M bits. When the hand arrives at a frame: if R = 1, then R ← 0, time ← now, and the hand advances. If R = 0, age > τ, and the page is clean, the page is the victim. If R = 0, age > τ, and the page is dirty, a write of the page to disk is scheduled and the hand advances. If R = 0 and age ≤ τ, the page is in the working set; the hand advances.

It is important that a dirty old page is not evicted immediately. The write is handed to the disk while the hand goes on looking for a clean page. When the write finishes, the page has become clean and can be evicted cheaply later. If the hand makes a full circle and returns to where it started, there are two cases:

- If at least one write was scheduled, the hand goes around until the first page whose write has finished and evicts it.
- If no write was scheduled, all pages are in the working set. Any clean page is evicted; if there is none, the page under the hand is written out and evicted.

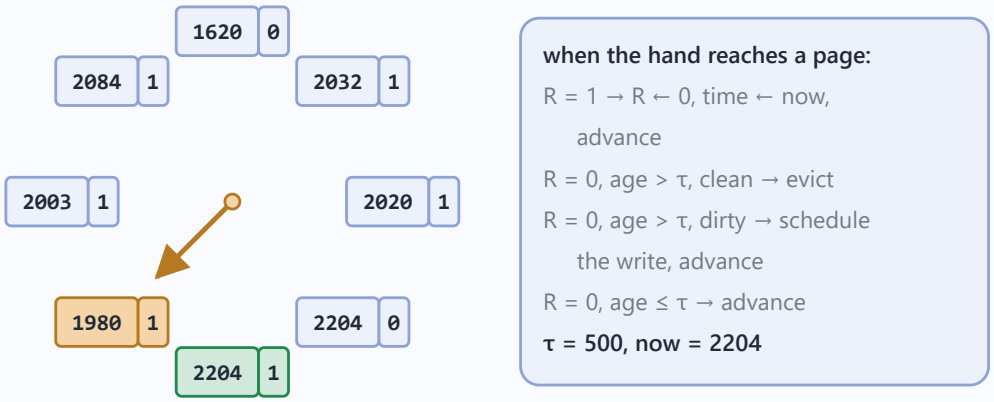


Figure 9.34: WSClock (a redrawing of Tanenbaum Figure 3-21), $\tau = 500$, now = 2204. The hand finds R = 1 at the page with time 2014, updates it, and moves on; because the page with time 1213 has age 991 > τ, the new page is loaded into that frame.

In Tanenbaum's example the hand is first at the page with time 2014. R = 1, so the page was used in this tick and is not a good victim. R is cleared, the time is set to 2204, and the hand advances. The next page has time 1213 and R = 0. Its age is 2204 − 1213 = 991 > τ, so it is not in the working set. If it is clean, it becomes the victim; if it were dirty, its write would be scheduled and the hand would advance. The new page is loaded into this frame with time 2204 and R = 1. WSClock is a simple, efficient, and widely used design in practice.

9.7.4 WSClock on the common string

To run WSClock on the common string, the reference number is taken as virtual time and $\tau = 4$ is chosen. The writes are the same as in NRU (the 2nd, 6th, 9th, and 15th references). A page whose write has been scheduled is considered cleaned by the end of the round.

- **Reference 4 (2).** The hand walks over 7, 0, and 1 in turn; all three have R = 1. The R bits are cleared and their times are set to 4. The round ends and there is no write. The first clean page is 7 (0 is dirty), and 7 is evicted.
- **Reference 6 (3*).** 0: R = 1 → R = 0, time 6. 1: R = 0, age 6 − 4 = 2 ≤ τ, in the working set. 2: R = 1 → R = 0, time 6. The round ends and there is no write. Starting again from the hand, the first clean page is 1; 1 is evicted.
- **References 8 and 9.** In the same way, first 2 and then 4 are evicted as the first clean page.
- **Reference 14 (1).** Memory holds 0*, 3*, 2*; all three are dirty and their R bits are 1. The round ends and there is no clean page: 0, under the hand, is written out and evicted (write 1).

- **References 16–18.** Each time, no clean page outside the working set is found; at the end of the round the first clean page is evicted: 1, 0, 1 respectively.
- **Reference 19 (0).** 3: R = 0, age 19 − 14 = 5 > τ and dirty: its write is scheduled. 2: age 3 ≤ τ. 7: R = 1 → R = 0. The round ends; the written 3 is now clean and becomes the victim (write 2).
- **Reference 20 (1).** 7 is evicted as the first clean page.
- **Result.** 13 faults, 2 dirty writes (Figure 9.35).

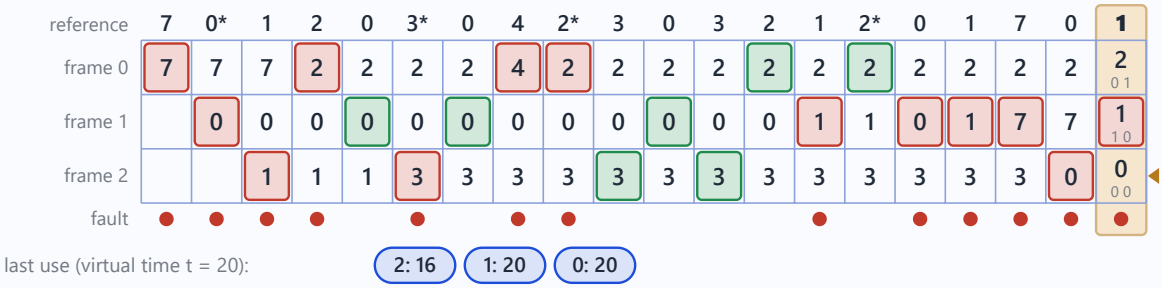


Figure 9.35: WSClock, common string, 3 frames, τ = 4. The cells show the R and M bits, and the status row shows the times of last use. 13 faults in total, 2 dirty writes.

13 faults is a result even worse than LRU's. The reason is the smallness of the example: with 3 frames every page is in the working set, and the τ criterion cannot distinguish any page. At most faults the algorithm falls back on the fallback rule at the end of the round, “the first clean page.” The payoff of the working-set idea appears in a memory with thousands of frames and in a long run.

9.8 Comparing the algorithms

9.8.1 Results on the common string

On the common string (3 frames) the results of all the algorithms can be put side by side (Table 9.10, Figure 9.36).

Algorithm	Fault	Comment
OPT	9	Not implementable; the lower bound for comparison.
Aging*	10	An efficient algorithm that approximates LRU well.
NRU*	11	A very crude approximation of LRU; cheap.
Enhanced clock*	11	Combines NRU's classes with the clock.
LRU	12	Very good, but the exact version needs hardware.
WSClock*	13	Good and efficient; its advantage is not visible in the small example.
Clock	14	A realistic implementation of second chance.
FIFO	15	May evict important pages; Belady's anomaly.

Algorithm	Fault	Comment
Second chance	—	A big improvement over FIFO (its decisions are the same as the clock's).
NFU	—	A rather crude approximation of LRU; never forgets.
Working set	—	Good, but scanning the table on every fault is expensive.

Table 9.10: Fault counts on the common string and Tanenbaum's general assessment. Starred algorithms also use write and tick information.

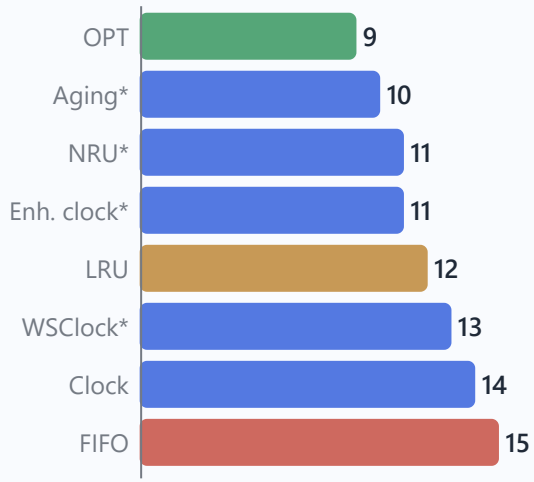


Figure 9.36: Fault counts on the common string: OPT 9, aging 10, NRU and enhanced clock 11, LRU 12, WSClock 13, clock 14, FIFO 15.

The ranking in the table can be misleading. Algorithms with R and M bits exploit the write and tick information that LRU does not use at all; a single short string does not give a general ranking. Tanenbaum's assessment rests on long measurements: in practice the two best algorithms are aging and WSClock. NFU does not approximate LRU well because it never forgets; the good approximation is aging.

9.8.2 The workload determines the policy

How good an algorithm is depends on the access pattern. To see this, the experiment in OSTEP can be repeated. There are three synthetic workloads making 10,000 accesses to 100 distinct pages (50 pages in the loop). Each is run with five policies while the number of frames varies from 0 to 100 (Figure 9.37). For the random policy the average of 5 trials is taken.

- **The no-locality workload.** Each access is chosen at random from the 100 pages. The past says nothing about the future; LRU, FIFO, clock, and random all follow the same line, and the hit rate is approximately frames / 100. Only OPT, which knows the future, is ahead. With 100 frames all pages fit and all reach a 99% hit rate (the remaining 1% are the initial loads).

- **The 80-20 workload.** 80% of the accesses go to 20% of the pages. When there are hot pages, the policies that use the past win. With 20 frames OPT gives an 81% hit rate, LRU 58%, clock 56%, and FIFO and random about 53%. Clock is very close to LRU; FIFO is no better than random.
- **The looping workload.** 0, 1, ..., 49, 0, 1, ... are repeated in order. A loop slightly larger than memory is the worst case for LRU. With 49 frames, the page evicted is always the one that will be needed soonest, and LRU, FIFO, and clock miss on every access: 0%. Random gives a 95.5% and OPT a 97.5% hit rate. With 50 frames the loop fits and all rise to 99.5%.

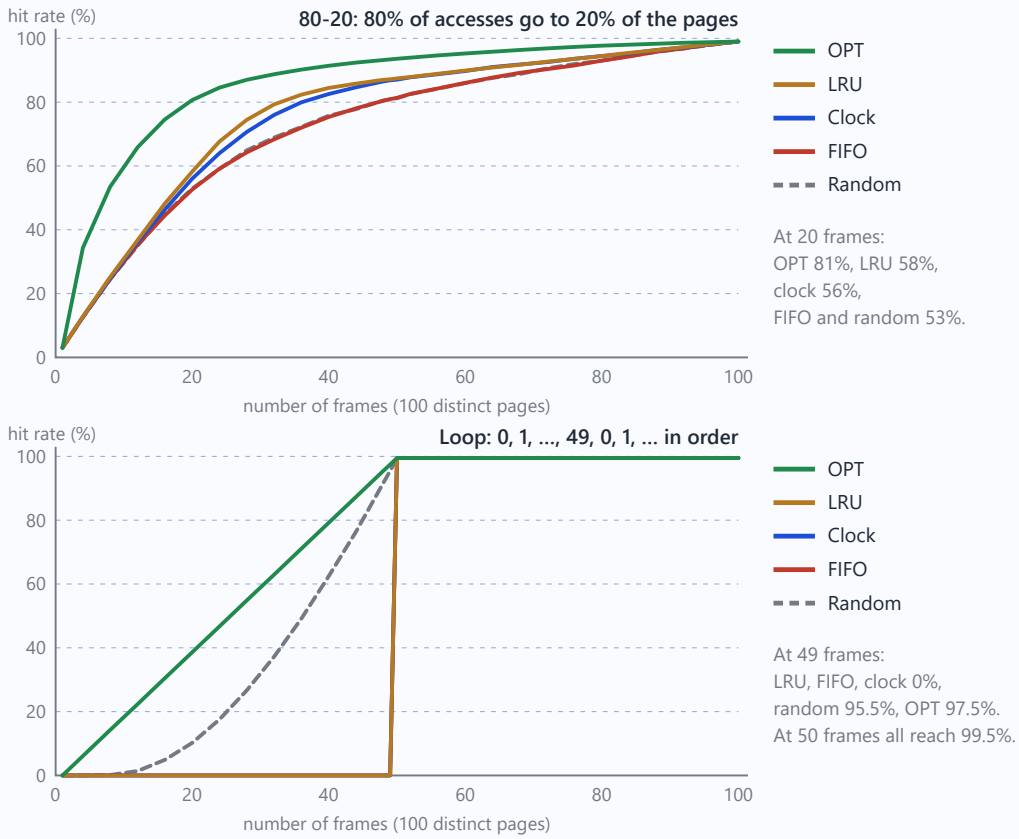


Figure 9.37: Hit rate versus number of frames. Top: the 80-20 workload, where the policies that use the past are ahead. Bottom: the loop; with 49 frames LRU, FIFO, and clock achieve 0% and random 95.5%.

This experiment was rerun in Python for these notes. The same results come out, up to small differences due to the random number generator: in 80-20 with 20 frames, OPT 80.7%, LRU 58.1%, clock 55.2%, FIFO 52.7%, random 52.4%; in the loop with 49 frames, OPT 97.5%, random 95.5%, the others 0%.

The looping pattern is not merely an artificial experiment. Reading a large file once from start to end is also a scan, and it hits LRU in real systems: file pages that will never be used again drive the hot pages out of memory. This is why Linux puts a new page first on the inactive list and moves it to the active list only on the second access.

Common mistake

Believing that LRU is better than FIFO and random in every case. LRU relies on locality. When the accesses follow a loop slightly larger than memory, LRU always evicts the wrong page and blind random choice beats it by a wide margin.

9.9 Frame allocation

When several processes share memory, the second question arises: how many frames should each process get?

9.9.1 Equal and proportional allocation

Allocation has two limits. The **lower limit** is set by the architecture: all the pages an instruction may touch must be in memory at the same time, or the instruction can never complete. The instruction itself, its operands, and indirect addresses may cross page boundaries. For example, the IBM 370's memory-to-memory move instruction MVC may need six frames: the instruction itself, the source block, and the destination block can each straddle two pages. The **upper limit** is the free physical memory. Between these two limits there are three approaches.

Definition 9.25

Equal allocation: m frames are divided among n processes as $\lfloor m / n \rfloor$ frames each. **Proportional allocation**: if the virtual size of process i is s_i and the total is $S = \Sigma\, s_i$, then the process gets $a_i = s_i / S \times m$ frames (rounded down). **Priority allocation**: the ratio is based on priority instead of size, or on priority together with size.

Example 9.14

62 frames, P1 of 10 pages and P2 of 127 pages. What do equal and proportional allocation give?

1. **Equal.** $\lfloor 62 / 2 \rfloor = 31$: 31 frames each. P1, with 10 pages, can never use 21 of its frames.
2. **Proportional.** $S = 137$. P1: $10 / 137 \times 62 \approx 4.53 \to 4$. P2: $127 / 137 \times 62 \approx 57.47 \to 57$.
3. **Remainder.** $4 + 57 = 61$; because of rounding down, 1 frame is left unused (Figure 9.38).

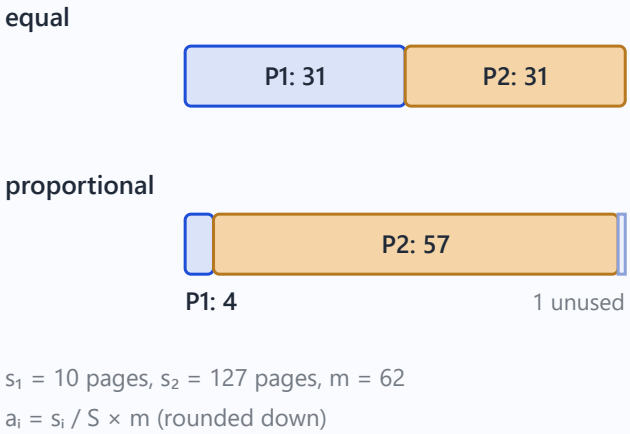


Figure 9.38: Distributing 62 frames between two processes: 31 and 31 under equal allocation, 4 and 57 under proportional (1 frame unused).

In priority allocation, when a high-priority process takes a page fault, it may take a frame from a lower-priority process. As the degree of multiprogramming changes, so does the allocation: when a new process arrives, everyone's share shrinks a little. On NUMA (non-uniform memory access) systems it also matters *where* a frame is. A process should be fed from memory close to the processor node it runs on; the memory of a remote node is slower.

9.9.2 Local and global replacement

Which frames the victim is chosen from is also a decision.

Definition 9.26

In **global replacement** the victim is chosen from among all frames; a process may take another process's frame. In **local replacement** a process chooses its victim only from among its own frames.

Example 9.15

Three processes (A, B, C) share 16 frames. The numbers are aging counters: A0 10, A1 7, A2 5, A3 4, A4 6, A5 3; B0 9, B1 4, B2 6, B3 2, B4 5, B5 6, B6 12; C1 3, C2 5, C3 6. A faults on page A6. Which page is evicted?

1. **Local.** The candidates are only A's frames. The smallest value is A5's 3; A5 is evicted. A's frame count stays at 6.
2. **Global.** All frames are candidates. The smallest value is B3's 2; B3 is evicted. A grows to 7 frames and B shrinks to 6 (Figure 9.39).

Figure 9.39: Local and global replacement (a redrawing of Tanenbaum Figure 3-22). Above, the initial state; below, the result of the global policy: A6 has taken B3's frame. Under the local policy A5 would have been evicted.

The global policy uses memory more efficiently: idle frames pass to whoever needs them, and total throughput rises. The price is unpredictability. A process's running time can vary a lot depending on the memory behavior of other processes; the same program may take 10 seconds one day and 30 the next. The local policy provides this consistency, but frames lying idle in one process cannot pass to another that is cramped. It also needs a separate mechanism to adjust the number of frames as the working set grows and shrinks; page-fault frequency is that mechanism. Linux and Windows use global reclaim; limits are set with cgroups and job objects.

9.10 Thrashing

9.10.1 Futile paging

Definition 9.27

Thrashing is the state in which the system spends most of its time moving pages between memory and disk. It arises when the processes' active pages, that is, their working sets, do not fit in memory.

The mechanism is a vicious circle. If a process's working set does not fit in memory, the process faults continuously. The page it evicts is needed again right away; to bring that page in, it evicts another needed page. The processes spend most of their time waiting for pages, the disk queue fills, the ready queue empties, and CPU utilization falls.

There is also a feedback that makes things worse. The long-term schedulers of older systems monitored CPU utilization and interpreted low utilization as “the CPU is idle, we can take on more work.” When a new process enters the system, everyone's frame share shrinks further, faults increase, CPU utilization falls even more, and the scheduler adds yet another process. As the degree of multiprogramming rises, CPU utilization first increases, then after some point collapses (Figure 9.40).

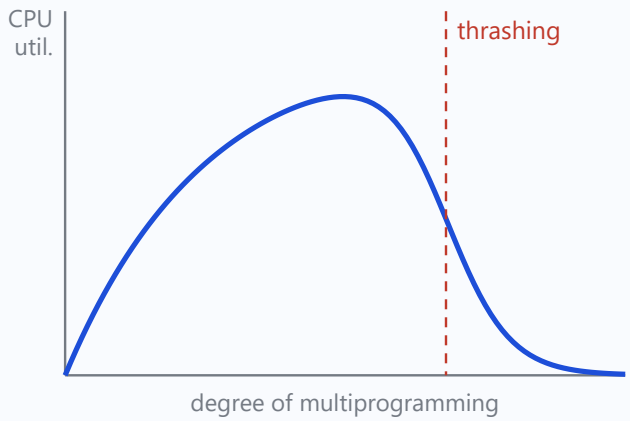


Figure 9.40: CPU utilization first rises as the degree of multiprogramming increases; once thrashing begins it falls rapidly.

The formula for the cause is short: $\Sigma\ |WS_i| > m$, that is, the sum of the working sets is larger than memory. Local replacement confines thrashing to a single process: the cramped process cannot steal other processes' frames. But it does not solve the problem; that process is still thrashing, and by filling the disk queue it slows other processes' page faults too.

The remedies rest on restoring the balance between memory and demand. The degree of multiprogramming is reduced: some processes are suspended or swapped out. RAM is added. The last resort is the system's out-of-memory (OOM) killer. On Linux the symptom shows in the output of `vmstat 1`: high swap traffic in the `si` and `so` columns, high I/O wait in the `wa` column. The file `/proc/pressure/memory` (PSI, pressure stall information) reports memory pressure as the percentage of time that processes spend waiting because of memory.

Common mistake

Always taking low CPU utilization for idle capacity. If a monitor shows CPU at 7%, the swap disk at 100%, and a very high page-fault rate, the system is not idle but thrashing. In this case taking on new work is wrong; on the contrary, some processes must be suspended.

9.10.2 Preventing it with the working set and PFF

The working-set strategy. The operating system tracks each process's working set and computes the total demand: $D = \Sigma\ |WS_i|$. If $D > m$, thrashing is approaching and a process is suspended; its pages are written out and its frames are distributed to the others. If D is small enough, a new process can be started. For a process returning from suspension, loading its working set first, that is, prepaging, prevents a storm of faults on return.

Page-fault frequency. Tracking the working set is expensive. A more direct feedback is to look at the process's fault rate.

Definition 9.28

In the **page-fault frequency** (PFF) method, an upper and a lower bound are set on each process's fault rate. If the rate exceeds the upper bound, the process is given a frame; if it falls below the lower bound, a frame is taken back from the process (Figure 9.41).

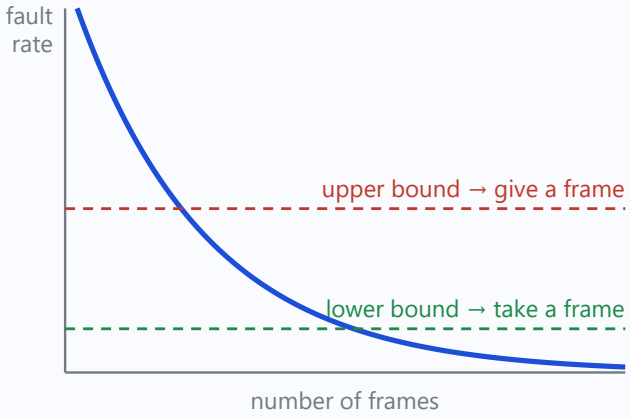


Figure 9.41: The page-fault rate decreases as the number of frames grows. Above the upper bound the process is given a frame; below the lower bound a frame is taken back.

If the fault rate is high but there is no free frame to give, a process is suspended and its frames are distributed. There is also a link between page faults and localities. When a process moves to a new locality, the fault rate peaks. Once the working set is loaded, the rate drops and stays low until the next transition. PFF reads these peaks as “more frames are needed” and the calm periods as “there are too many frames.”

9.10.3 Code trace: PFF

The function below is a simple form of PFF. It is called at every page fault. `t` is the current reference number, `s->last` is the time of the last fault, and `C` is the threshold. If the time since the last fault is greater than `C`, faults are rare and the pages not used since the last fault are released. If it is smaller, faults are frequent and the set grows without any page being evicted.

```
void pff(Proc *s, int p, int t) {
  if (t - s->last > C) {  /* rare faults */
    for (int i = s->n - 1; i >= 0; i--)
      if (s->set[i].R == 0)
        evict(s, i);    /* unused */
  }
  for (int i = 0; i < s->n; i++)
    s->set[i].R = 0;      /* clear */
  add(s, p);              /* with R = 1 */
  s->last = t;
}
```

In the rare-fault branch the loop walks the set from the end toward the beginning. This direction is deliberate: when `evict()` removes an element, the elements behind it shift left by one. Walking from the beginning would skip an element. Then all the R bits are reset, so that at the next fault the question “was it used in this interval?” can be asked again. The new page is added with R = 1 and the fault time is recorded.

The process starts with no memory; `C` = 2 and the reference string (from time 1 to 14) is 1 2 1 3 1 3 1 3 4 4 5 4 5 2 (Table 9.11).

`t`	Page	`t - last`	Branch	Evicted	Set after
1	s1	1 − 0 = 1	frequent	—	{s1}
2	s2	2 − 1 = 1	frequent	—	{s1, s2}
4	s3	4 − 2 = 2	frequent	—	{s1, s2, s3}
9	s4	9 − 4 = 5	rare	s2	{s1, s3, s4}
11	s5	11 − 9 = 2	frequent	—	{s1, s3, s4, s5}
14	s2	14 − 11 = 3	rare	s3, s1	{s4, s5, s2}

Table 9.11: Trace of `pff()`, C = 2. Only the fault moments are shown.

- ***t* = 1, 2, 4.** The intervals are 1, 1, and 2; none exceeds `C`. Faults are frequent and no page is evicted; the set grows to 3 pages. Reference 3 in between (s1) is a hit.
- ***t* = 9.** References 5–8 hit s1 and s3 and set their R bits to 1. At the fault on s4 the interval is 5 > 2: faults are rare. The only page not used since the last fault (R = 0) is s2, and it is evicted.

- ***t* = 11.** The interval is 2 ≤ C. No page is evicted and the set grows to 4 pages. While the process moves to a new locality (s4, s5), the old pages (s1, s3) are still in memory.
- ***t* = 14.** References 12 and 13 hit s4 and s5. At the fault on s2 the interval is 3 > 2. s3 and s1, which have R = 0, are evicted. The final set is s4 (R = 0), s5 (R = 0), s2 (R = 1).

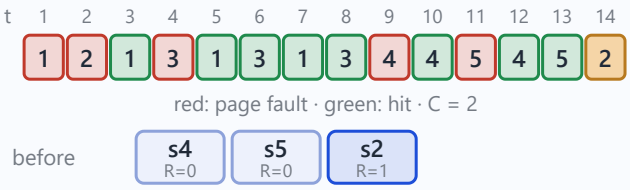


Figure 9.42: The end of the PFF trace: 6 faults in 14 references. Red cells are page faults, green ones hits; the final set is s4, s5, s2.

There were 6 faults in 14 references. When the locality shifted from pages 1–3 to pages 4–5, faults first became frequent and the set grew. Then a rare fault was the opportunity to release the pages of the old locality. The cost of the function is linear in the set size, O(n); but it runs only at fault time.

9.11 Design issues

The replacement algorithm and the allocation policy are only part of a virtual memory system. How well the system actually works is determined by many small decisions: when to look for a victim, whether pages are brought in ahead of time, how large a page should be, what the programmer can do, and which pages must never be evicted.

9.11.1 Cleaning policy and the paging daemon

Looking for a victim only at page-fault time makes the faulting process wait needlessly. If the victim is dirty, the process waits for both the write and the read. A better way is to keep free frames ready in advance.

Definition 9.29

The **paging daemon** is a kernel thread that wakes up periodically. If the number of free frames is low, it frees pages ahead of time using the replacement algorithm and writes dirty pages to disk ahead of time.

The payoff of writing ahead is this: when the moment of eviction comes, the page is usually clean and the fault waits only for a read. A freed frame is not wiped immediately either. Its contents stay in place, and if the page is needed again it can be reclaimed from the pool; in this case a minor fault suffices instead of a disk read.

One way of doing this job with the clock algorithm is the **two-handed clock**. The front hand has dirty pages written out and clears the R bits. The back hand follows it at a certain distance and frees the pages that still have R = 0, that is, that were not used in the time between the two hands. The distance between the hands determines the time a page is given "to be used."

On Linux this daemon is called `kswapd` and works with three watermarks per memory zone: min, low, and high (Figure 9.43). When the number of free pages falls below low, kswapd wakes up and reclaims pages in the background. When the number rises to high, it goes back to sleep. If the number falls below min as well, the allocating process gets to work without waiting and does the reclaim itself; this is called **direct reclaim** and it increases latency. Dirty file pages are also written to disk by writeback threads.

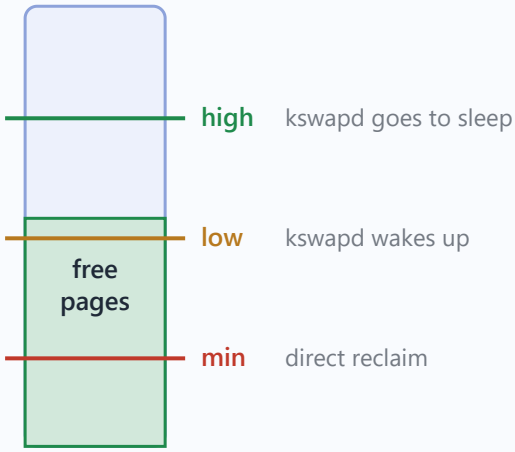


Figure 9.43: Free-page watermarks: above high, kswapd sleeps; it wakes when the count falls below low; below min, the allocating process performs direct reclaim.

9.11.2 Prepaging

Demand paging suffers a storm of faults when a process starts and when it returns from suspension: each page arrives with a separate fault and a separate I/O.

Definition 9.30

Prepaging means bringing in, ahead of time and with a single bulk read, the pages that are predicted to be needed, without waiting for faults.

Whether prepaging pays off can be seen with a simple calculation. Suppose s pages are brought in ahead of time and a fraction α of them is actually used ($0 \le \alpha \le 1$). The gain is that $\alpha \cdot s$ page faults are avoided. The cost is reading $(1 - \alpha) \cdot s$ unnecessary pages and having them occupy memory. If α is close to 1, prepaging gains; if it is close to 0, it loses (Figure 9.44).

Figure 9.44: In demand paging each page arrives with a separate fault (8 faults, 8 separate I/Os). In prepaging the working set arrives with one bulk read; $\alpha \cdot s$ pages are used and $(1 - \alpha) \cdot s$ are wasted.

Example 9.16

$s = 40$ pages of the working set of a process returning from suspension are brought in ahead of time, and a fraction $\alpha = 0.6$ of them is used. Bringing in a page by an individual fault takes 5 ms, and reading all 40 pages at once takes 30 ms. Is prepaging profitable? Below what value of α does it lose?

1. **Pages used.** $\alpha \times s = 0.6 \times 40 = 24$.
2. **Without prepaging.** 24 faults × 5 ms = 120 ms.
3. **With prepaging.** A single bulk read, 30 ms. The gain is 90 ms.
4. **Threshold.** The gain is $40\alpha \times 5 = 200\alpha$ ms and the cost is 30 ms: $200\alpha > 30 \rightarrow \alpha > 0.15$.
5. **Hidden cost.** $(1 - \alpha) \times s = 16$ unnecessary pages hold frames and may cause other pages to be evicted.

In this example the threshold is low because the bulk read is very cheap. As the cost of the bulk read approaches that of individual faults, the threshold rises. The classical practice is to remember the working set of a suspended process and load it on return. Linux uses readahead for files and the `page-cluster` setting for swap. An application can also give hints: `madvise(MADV_WILLNEED)` or `mmap(MAP_POPULATE)`. On Windows, the Prefetcher and SysMain (formerly SuperFetch) load frequently used pages while the system is idle.

9.11.3 Page size

The page size is usually determined by the hardware, but when a new architecture is designed it is a choice, and there are competing forces (Table 9.12).

Small page	Large page
less internal fragmentation	smaller page table
tracks locality finely, brings in little unneeded data	fewer page faults, efficient I/O (seek and setup cost is paid once)
many TLB misses	large TLB reach

Table 9.12: The advantages of small and large pages.

The two forces can be weighed with a simple model. Let the process size be s bytes, the page-table entry size e bytes, and the page size p bytes. The page table has s/p entries, that is, se/p bytes. The last page of the process is on average half empty and creates $p/2$ bytes of internal fragmentation.

Theorem 9.2 (Optimal page size)

If the overhead is $f(p) = se/p + p/2$, its minimum is at $p = \sqrt{(2se)}$.

The proof is by differentiation: from the equation $f'(p) = -se/p^2 + 1/2 = 0$ we get $p^2 = 2se$. Since the second derivative is positive, this point is a minimum.

Example 9.17

What is the optimal page size for s = 1 MB and e = 8 B? What changes if s = 4 MB?

1. **Powers of two.** 1 MB = 2^{20}, 8 = 2^3. $2se = 2 \times 2^{20} \times 2^3 = 2^{24}$.
2. **Square root.** $p = \sqrt{(2^{24})} = 2^{12}$ = 4,096 B = 4 KB.
3. **Overhead.** $f(4{,}096) = 2^{23} / 2^{12} + 2{,}048 = 2{,}048 + 2{,}048 = 4{,}096$ B. With 2 KB it is 5,120, and with 8 KB again 5,120 B.
4. ***s* = 4 MB.** $2 \times 2^{22} \times 2^3 = 2^{26}$, $\sqrt{(2^{26})} = 2^{13}$ = 8 KB.

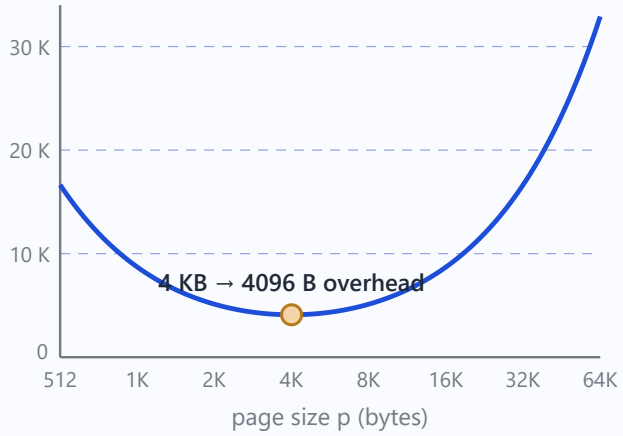


Figure 9.45: The overhead se/p + p/2 for s = 1 MB and e = 8 B. The minimum is at 4 KB, 4,096 B.

The model is crude; it does not account for the TLB or for I/O at all. Even so, it explains why 4 KB has remained the standard for decades. As of 2026, x86-64 uses 4 KB pages and supports huge pages of 2 MB and 1 GB. Apple Silicon uses 16 KB pages. Android 15 and later support kernels with 16 KB pages. Some ARM64 servers run with 64 KB pages. 1 KB pages are seen only in old systems. The trend is that pages grow as memory grows.

9.11.4 TLB reach and huge pages

Definition 9.31

The **TLB reach** is the amount of memory that can be accessed through the TLB without a miss: the number of TLB entries × the page size.

If the process's working set does not fit in the reach, a significant fraction of accesses end in TLB misses and time goes to page-table walks. There are two ways to increase the reach. Increasing the number of entries is expensive, because the TLB is an associative memory and each entry needs a parallel comparison circuit. Enlarging the page is far more effective (Figure 9.46).

Example 9.18

(a) What is the reach of a 64-entry TLB with 4 KB and with 2 MB pages? (b) A database's hot data is 48 MB; the second-level TLB has 1,024 entries. Does the data fit in the reach with 4 KB and with 2 MB pages?

1. **(a)** 64 × 4 KB = 256 KB; 64 × 2 MB = 128 MB.
2. **(b) 4 KB.** 1,024 × 4 KB = 4 MB. The 48 MB of data does not fit; most accesses become TLB misses.

3. **(b) 2 MB.** 1,024 × 2 MB = 2 GB. The data fits comfortably.

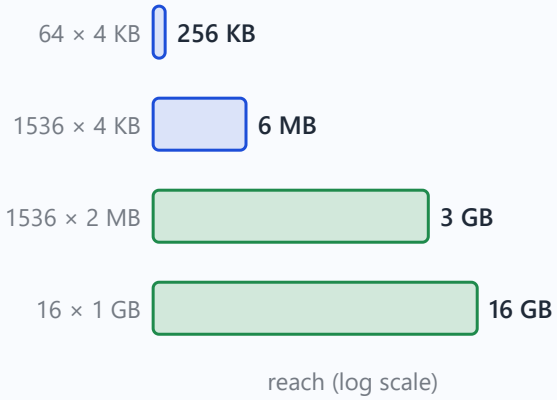


Figure 9.46: TLB reach on a logarithmic scale: 64 × 4 KB = 256 KB, 1536 × 4 KB = 6 MB, 1536 × 2 MB = 3 GB, 16 × 1 GB = 16 GB.

On x86-64 the huge pages are 2 MB and 1 GB. Linux offers two ways: `hugetlbfs`, which the application requests explicitly, and **transparent huge pages** (THP), which the kernel uses on its own. In THP the `khugepaged` thread merges contiguous small pages into huge pages in the background. 6.x kernels can also use sizes between 4 KB and 2 MB, with multi-size folios and mTHP support. On Windows, a special privilege is needed to use large pages.

Huge pages have costs too: internal fragmentation, the need to find a large contiguous physical region (which may require memory compaction), and a longer copy and zeroing time per page. The software that benefits most from huge pages is databases and virtual machines; both use large, frequently accessed memory regions.

9.11.5 Program structure and locality

Demand paging should be invisible to the programmer, but the programmer's choices can change the number of faults greatly. The best-known example is the traversal order of a two-dimensional array. C stores arrays in **row-major** order: `data[i][0]`, `data[i][1]`, … are consecutive in memory.

```
int data[128][128];   /* row = 1 page */
/* A: column by column */
for (int j = 0; j < 128; j++)
  for (int i = 0; i < 128; i++)
    data[i][j] = 0;
/* B: row by row */
for (int i = 0; i < 128; i++)
  for (int j = 0; j < 128; j++)
    data[i][j] = 0;
```

The array is 128 × 128 `int`s; a page is 128 words, so each row is exactly one page. The process is allocated 64 frames, none is filled at the start, and replacement is LRU.

Example 9.19

How many page faults do loops A and B produce?

1. **A, column 0.** The inner loop is over `i`: `data[0][0]`, `data[1][0]`, … Each `i` is a different row, that is, a different page. Over column 0 there are 128 faults. 128 pages do not fit in 64 frames; when the column ends, pages 0–63 have long since been evicted.
2. **A, column 1.** `data[0][1]` is again on page 0, but page 0 has been evicted: another fault. LRU evicts the page that will be needed soonest every time.
3. **A, total.** Every access is a fault: 128 × 128 = 16,384 faults.
4. **B, row 0.** `data[0][0]` faults on page 0; then `data[0][1…127]` are on the same page and are hits.
5. **B, total.** One fault per row: 128 faults. The same work, 128 times fewer faults.

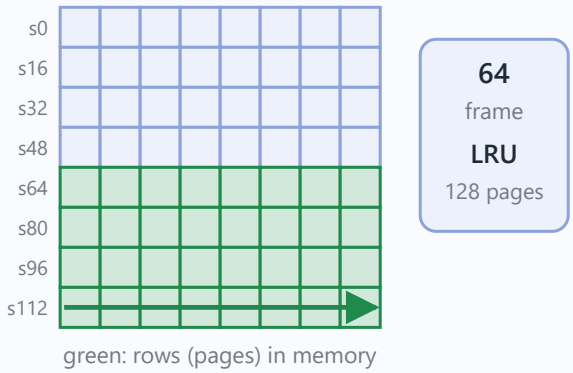


Figure 9.47: A to-scale picture of the 128 × 128 array: each row is a page. Loop B, which goes row by row, makes one fault per row, 128 in total.

The same principle holds at other sizes. In a 64 × 1024 `int` array (each row 4 KB = one page), filling row by row with 32 frames causes 64 faults, while filling column by column causes 64 × 1,024 = 65,536 faults: a factor of 1,024. Fortran, on the other hand, stores arrays in column-major order, and there the correct order is the reverse.

The choice of data structure also affects locality. A stack shows good locality: accesses are always near the top. A hash table deliberately scatters accesses and shows poor locality. Linked structures that chase pointers are also poor, because they scatter their nodes across memory. Compilers sometimes swap the inner and outer loops on their own with a **loop interchange** transformation. The compiler and the loader can also place code with page boundaries in mind: for example, putting functions that call each other on the same page.

9.11.6 I/O locking and page pinning

Some pages must absolutely not be evicted, at least for a while. The most important example is I/O buffers.

Example 9.20

Process A supplies a buffer for a disk read. The global replacement policy is in use. What can go wrong?

1. **Request.** The kernel programs direct memory access (DMA) with the *physical* address of frame 42, where the buffer is.
2. **Waiting.** A blocks; the device waits in the queue or is running. Meanwhile frame 42 looks like an ordinary page.
3. **Replacement.** B takes a page fault. The global policy chooses frame 42 as the victim and loads B's page into it.
4. **Corruption.** When the DMA completes, the device writes the data into frame 42, that is, over B's page. The data has gone into the wrong process's memory.

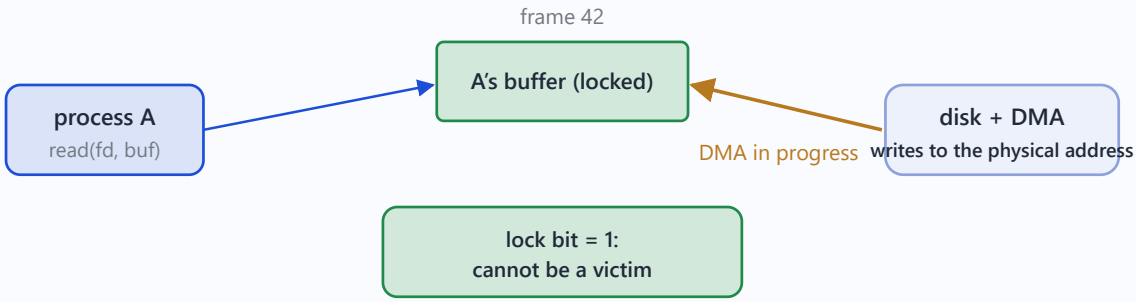


Figure 9.48: I/O locking. Frame 42, A's buffer, is locked while the DMA is in progress; since the lock bit is 1, it cannot be chosen as a victim.

Definition 9.32

Pinning, or **page locking**, means setting a lock bit on a frame so that it is kept out of the replacement candidates.

There are two solutions to the problem. The first is to lock the buffer frame for the duration of the I/O. The second is always to do the I/O into kernel buffers and copy the data to the user buffer afterwards; this is safe but requires an extra copy.

Other pages are pinned too. Part of the kernel's code and data is never evicted; if the page-fault handler itself were evicted, who would bring it back? Real-time processes, which cannot tolerate latency, lock their memory with `mlock()` or `mlockall()`. Linux drivers pin user pages with `pin_user_pages()`; RDMA network cards, GPUs, and io_uring's fixed buffers use this. The IOMMU makes devices use translated addresses too and prevents DMA to a wrong address in hardware.

There is also a subtlety about priority. A low-priority process, after a long wait, may have just brought in its page without yet having run. The R bit of this page is 0, and a fault by a high-priority process may immediately choose it as the victim. The effort of fetching is wasted. A newly loaded page can therefore be kept locked until it has been used once.

The risk of locking is also clear: if a locked page is forgotten, the frame can never be used again. Many locks squeeze memory and make thrashing easier. For this reason the amount of memory a user can lock is limited; on Linux the limit can be seen with `ulimit -l`.

9.12 Kernel memory

The memory of user processes is paged and comes from the free-frame pool. The kernel's own memory follows different rules. Most kernel memory is not paged, because code such as the page-fault handler must not fault. The kernel allocates structures much smaller than a page (process descriptors, file objects, semaphores), and page-level allocation is wasteful for these. Some devices need physically contiguous memory for DMA, because they do not know virtual addresses. Two allocators serve these needs.

9.12.1 The buddy system

Definition 9.33

The **buddy system** is an allocator that divides a physically contiguous region into blocks of power-of-two sizes only. A request is rounded up to the smallest power of two that satisfies it. If there is no free block of the right size, a larger block is split into two equal halves, two **buddies**. When a block is released, if its buddy is also free, the two merge.

Example 9.21

The kernel has 256 KB of physically contiguous memory. A request for 21 KB arrives. How do the splits go, and how much is the internal fragmentation?

1. **Rounding.** The smallest power of two that satisfies 21 KB is 32 KB.
2. **First split.** The 256 KB is split into the buddies A_L and A_R of 128 KB each.
3. **Second split.** A_L is split into B_L and B_R of 64 KB each.

4. **Third split.** B_L is split into C_L and C_R of 32 KB each. C_L is given to the request.
5. **Cost.** 21 requested, 32 KB given: 11 KB of internal fragmentation.

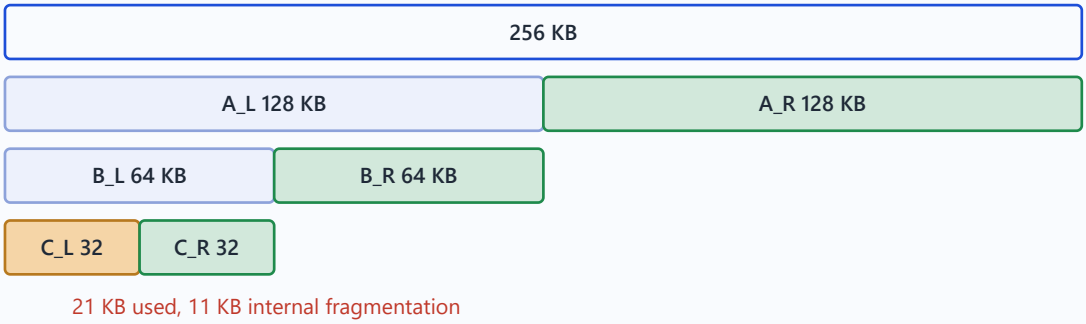


Figure 9.49: The buddy system: the 256 KB block is split into buddies of 128, 64, and 32 KB for a 21 KB request; C_L is given to the request.

The real strength of the buddy system is coalescing. When C_L is released, if its buddy C_R is free, the two re-form B_L; if B_R is also free, A_L forms, and so on. Finding the buddy's address takes a single operation: the buddy of a block of size k is found by flipping the bit of the block address that corresponds to k (address XOR size). Linux's page allocator is a buddy allocator. It keeps a separate free list for each order from 0 to 10: from 4 KB to 4 MB. The state of the lists can be seen with `cat /proc/buddyinfo`.

9.12.2 Code trace: the buddy allocator

The code below shows the core of the buddy allocator. Sizes are kept as powers of two in KB: `alloc(k)` asks for a block of 2^k KB. `fl[j]` is the list of free blocks of 2^j KB. `pop(j)` takes a block from the list, `push(j, a)` adds one to the list, and `remove(k, b)` removes `b` from the list if it is there and returns true.

```
int alloc(int k) {          /* 2^k KB */
  int j = k;
  while (fl[j] == NULL) j++;
  int a = pop(j);
  while (j > k) {   /* split, right half free */
    j--; push(j, a + (1 << j));
  }
  return a;
}
void release(int a, int k) {
  int b = a ^ (1 << k);     /* buddy */
  while (k < MAX && remove(k, b)) {
    a = a < b ? a : b;      /* merge */
    k++; b = a ^ (1 << k);
  }
  push(k, a);
}
```

`alloc()` first looks for the smallest list `j ≥ k` that has a free block and pops a block from it. Then the block is halved until it reaches the requested size: at each split the right half (`a + (1 << j)`) is added to a free list, and the left half continues to be split. `release()` finds the buddy's address with `a ^ (1 << k)`. If the buddy is on the free list of the same size, it removes it from the list, makes the lower-addressed of the two blocks the new block, and tries again at the next size up. If the buddy is not free, or the largest size (`MAX`) has been reached, the block is added to the list.

We start with a 256 KB region (`MAX` = 8) and call, in order, `A = alloc(5)`, `B = alloc(6)`, `release(0, 5)`, `release(64, 6)` (Table 9.13).

Call	Event	fl[5]	fl[6]	fl[7]	fl[8]
start	single block	—	—	—	@0
`alloc(5)`	j = 8, @0 popped; splits @128, @64, @32; A = 0	@32	@64	@128	—
`alloc(6)`	@64 ready in fl[6]; B = 64	@32	—	@128	—
`release(0, 5)`	buddy 0 ⊕ 32 = 32 free → @0 (64); buddy 0 ⊕ 64 = 64 in use	—	@0	@128	—
`release(64, 6)`	buddy 64 ⊕ 64 = 0 free → @0 (128); buddy @128 free → @0 (256)	—	—	—	@0

Table 9.13: Trace of the buddy allocator. Addresses are in KB; "@" marks the block start.

- `A = alloc(5)`. The 21 KB request is rounded up to 32 KB (2^5). fl[5], fl[6], and fl[7] are empty; the first nonempty list is fl[8]: j = 8. The 256 KB block at @0 is popped. At j = 7 the right half @128, at j = 6 @64, and at j = 5 @32 go onto the free lists. When j = k, the loop ends and the left half, the 32 KB at @0, is given to A.
- `B = alloc(6)`. @64 is ready in fl[6]; no split is needed. A's internal fragmentation is 32 − 21 = 11 KB.

- `release(0, 5)`. The buddy's address is found by flipping a single bit: 0 ⊕ 32 = 32. @32 is in fl[5]; they merge into 64 KB at @0. The new buddy is 0 ⊕ 64 = 64, but B is using @64. The merging stops and @0 is added to fl[6].
- `release(64, 6)`. The buddy 64 ⊕ 64 = 0 is free; the 128 KB block @0 forms. The new buddy @128 is also free and they merge into 256 KB. Since k = MAX, the loop stops; the region is a single piece again, as at the start.

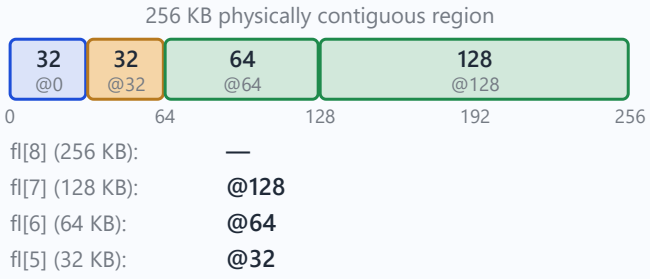


Figure 9.50: The region at the end of `alloc(5)`: A is the 32 KB at @0; @32 (32 KB), @64 (64 KB), and @128 (128 KB) are on the free lists.

Allocation and release take as many steps as there are splits and merges, which is at most the number of orders, that is, O(log *n*). Since a separate free list is kept for each size, lookup is fast too. Finding the buddy's address with XOR relies on blocks always starting at addresses that are multiples of their own size.

9.12.3 Buddy: a request-sequence example

Example 9.22

We start with a single free block of 1 MiB (1,024 K). In order, A = 70 K, B = 35 K, C = 80 K are requested; A is released; D = 60 K is requested; B, D, and C are released. What happens at each step?

1. **A = 70 K.** Rounded up to 128 K. 1,024 is split → 512 + 512 → 256 + 256 → 128 + 128; A takes 0–128. Internal fragmentation 58 K.
2. **B = 35 K.** Rounded up to 64 K. The free block 128–256 is split into 64 + 64; B takes 128–192. Internal fragmentation 29 K.
3. **C = 80 K.** Rounded up to 128 K. The free block 256–512 is split into 128 + 128; C takes 256–384. Internal fragmentation 48 K.
4. **A is released.** Its buddy 128–256 is in a split state and contains B: no merge.
5. **D = 60 K.** Rounded up to 64 K. B's free buddy 192–256 is given; no split is needed. Internal fragmentation 4 K.

6. **B is released.** Its buddy D is in use: no merge.
7. **D is released.** 128–192 and 192–256 merge into 128–256; that merges with A's free 0–128 into 0–256. Its buddy 256–512 holds C: stop.
8. **C is released.** 256–384 + 384–512 → 256–512; with 0–256 → 0–512; with 512–1024 → 0–1024. Memory is a single block again.

The total internal fragmentation for A, B, and C is 58 + 29 + 48 = 135 K. The buddy system is fast at allocating and merging, but in the worst case about half of a block is wasted: a request of $2^k + 1$ bytes gets a block of 2^{k+1} bytes. This is why a second allocator is built on top of the buddy system for small objects.

Common mistake

Believing that two adjacent free blocks can always be merged. Only *buddies* merge: the two halves of the same size that came from splitting the same block. For example, 64–128 and 128–192 are adjacent and of the same size, but they are not buddies; the buddy of 64 is 0 and the buddy of 128 is 192.

9.12.4 The slab allocator

Definition 9.34

The **slab allocator** keeps a separate **cache** for each frequently used type of kernel object. A cache consists of one or more **slabs**, and each slab consists of one or several physically contiguous pages. A slab is packed with objects of that type at their exact size.

A slab is in one of three states: full (all its objects in use), partial (some in use), and empty. Objects wait in the cache already initialized; allocation is simply taking an object from the list (Figure 9.51).

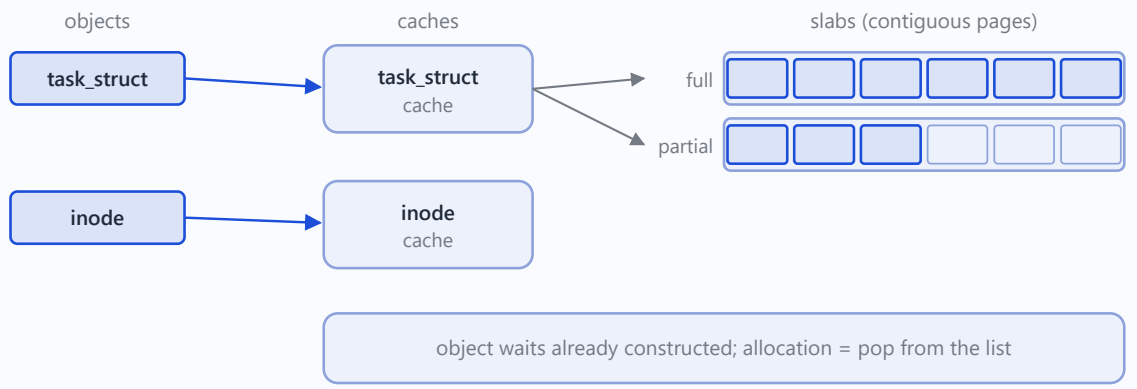


Figure 9.51: The slab allocator: separate caches for task_struct and inode objects; each cache has full and partial slabs, and the slabs consist of contiguous pages.

The path of a request is as follows. `fork()` asks for a new `task_struct`. The allocator first looks for a free object in the partial slabs; if it finds one, it hands it out. If the partial slabs fill up, it moves on to the empty slabs. If no partial or empty slab remains, new pages are obtained from the buddy allocator and a new slab is built. Under memory pressure, emptied slabs are returned to the buddy allocator.

Bonwick designed the slab allocator for Solaris in 1994. Its benefits are as follows:

- There is no fragmentation: objects are of exact size and are packed side by side within a page.
- Allocation and release are O(1); a released object is merely marked "free."
- Objects in the cache are hot and ready; they do not have to be initialized again at every allocation.
- Per-CPU lists provide a lock-free fast path.

In Linux three implementations of the slab allocator coexisted for a long time: SLAB, SLUB, and SLOB. SLOB was removed in version 6.4 and the classic SLAB in version 6.8 (2024); today the only allocator is SLUB. The general-purpose `kmalloc()` is also fed from slab caches divided into size classes. The state of the caches is monitored with `slabtop` and `/proc/slabinfo`.

9.13 Real systems

9.13.1 Linux: page reclaim

On Linux, reclaim is triggered by watermarks. When the number of free pages in a zone falls below low, kswapd wakes up; if it falls below min, the allocating process performs direct reclaim and latency rises.

Classical Linux reclaim uses two pairs of lists that resemble the clock: the **active** and **inactive** lists. There is a separate pair for anonymous pages and for file pages (Figure 9.52). A new page enters the inactive list; if accessed again, it is promoted to the active list. Pages at the tail of the active list move

down to the inactive list as they age. Victims are chosen from the tail of the inactive list. An anonymous victim goes to swap or to zswap; a clean file page is simply dropped. For locked pages there is also a separate *unevictable* list.

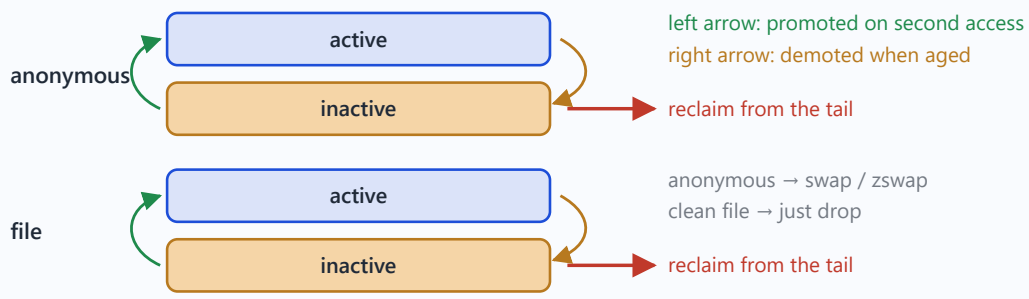


Figure 9.52: The Linux reclaim lists: a new page enters the inactive list and is promoted to the active list on its second access; an aging page moves back down and the victim is chosen from the tail of the inactive list.

This two-list arrangement is a safeguard against the looping pattern hitting LRU: file pages read once stay in the inactive list and cannot drive out the hot pages. The `vm.swappiness` setting (0–200) determines the preference between swapping out anonymous pages and dropping file pages. If memory really runs out, the OOM killer selects a process and terminates it.

9.13.2 Linux: MGLRU, zswap, and zram

MGLRU (multi-generational LRU) arrived in 2022 with Linux 6.1. Pages are divided into four generations instead of two lists. Access bits are collected in bulk by walking the page tables; an accessed page moves to the youngest generation, and the victim is chosen from the oldest generation (Figure 9.53). The result is less CPU spent and better victim choices. MGLRU is the default on ChromeOS and Android; many distribution kernels enable it with a build option. Its status can be read from the file `/sys/kernel/mm/lru_gen/enabled`.

zswap is a compressed cache that sits in front of swap. Before an anonymous page goes to disk, it is compressed in RAM and held in a pool. When the pool fills, the oldest entries are written to the real swap device. **zram**, on the other hand, is a compressed block device in RAM and is used directly as swap space; pages never go to disk. Fedora (since version 33), Android, and ChromeOS use zram by default.

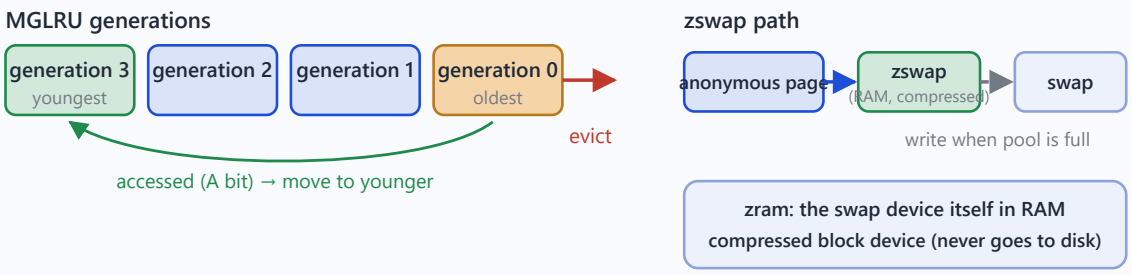


Figure 9.53: MGLRU: pages are divided into generations, evicted from the oldest generation, and an accessed page moves to the youngest. zswap compresses an anonymous page and keeps it in RAM, writing to swap when the pool is full; zram is the swap device itself.

Compression spends CPU, but it is hundreds of times cheaper than going to disk. The 10 µs fault time of zram in the EAT calculation shows this in numbers: at $p = 0.001$ the HDD gives a slowdown of 251, while zram brings only a 10% slowdown.

9.13.3 Windows: working sets and compression

Windows uses Denning's model under its own name. Each process has a **working set**, with minimum and maximum limits on its size. A page fault makes room in the set or grows it. When memory runs low, the **balance set manager** trims the sets.

A trimmed page is not discarded immediately (Figure 9.54). If clean, it goes to the **standby** list; if dirty, to the **modified** list. Modified pages move to the standby list once written. A page on the standby list that is requested again is reclaimed with a minor fault. The free and zeroed lists feed new allocations.

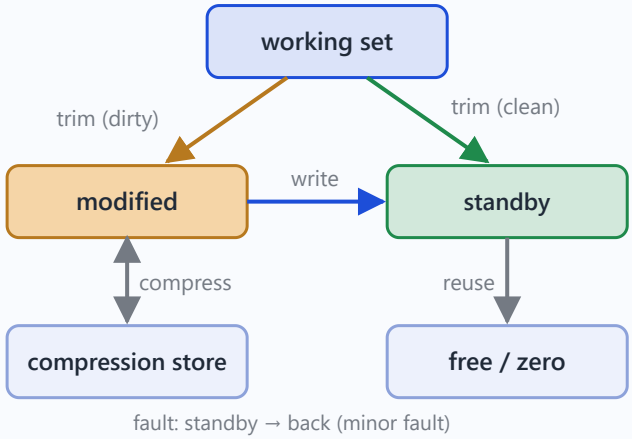


Figure 9.54: Windows page lists: a clean page trimmed from the working set goes to the standby list, a dirty page to the modified list. The compression store and the free/zero lists are also part of the flow.

Windows 10 (2015) brought **memory compression**: trimmed pages are held in a compression store in the System process; writing to `pagefile.sys` is the last resort. Prefetching is done by the Prefetcher and SysMain. Large pages are requested with `VirtualAlloc(MEM_LARGE_PAGES)`.

macOS and iOS follow a similar path: compressed memory is used. iOS has no swap at all; under memory pressure the system terminates background applications through the Jetsam mechanism. What the three systems have in common is that they try every possibility of staying in memory before going to disk.

9.14 Summary

- Virtual memory separates logical memory from physical memory. A process runs while only partly in memory; programs are not limited by physical memory, more processes are kept in the same memory, and sharing becomes easier.
- In demand paging a page is brought in only when it is accessed. An access to an invalid entry produces a page-fault trap. The kernel checks the reference, reads the page from the backing store into a free frame, updates the table, and restarts the instruction.
- Linux divides page faults into minor (no I/O), major (a read from disk), and invalid (`SIGSEGV`). Pages are file-backed, anonymous, or zero-filled.
- The effective access time is EAT = $(1 - p) \times ma + p \times$ fault time. On an HDD (25 ms), $p = 0.001$ slows memory by a factor of 251; a 10% slowdown requires $p < 4 \times 10^{-7}$. NVMe and zram reduce the penalty but do not remove it.
- Copy-on-write shares pages after `fork()` and copies only the page that is written; the frame's reference count decides whether a copy is needed. `mmap()` maps a file into the address space; `MAP_SHARED` writes changes to the file, `MAP_PRIVATE` to a private copy.
- If there is no free frame, a victim is chosen; a dirty victim is written to disk first. Algorithms are compared by their number of faults on a reference string.
- On the common string (3 frames) OPT incurs 9 faults, aging 10, NRU and enhanced clock 11, LRU 12, WSClock 13, clock 14, and FIFO 15. OPT cannot be implemented but is the lower bound.
- FIFO can show Belady's anomaly: on the string 1 2 3 4 1 2 5 1 2 3 4 5, 3 frames give 9 faults and 4 frames give 10. LRU and OPT are stack algorithms and do not show the anomaly.
- Exact LRU needs hardware (counter, stack, matrix). In practice, approximations based on the R and M bits are used: second chance, clock, NRU, enhanced clock, NFU, and aging. Aging approximates LRU well; NFU cannot because it never forgets.
- The working set WS(t, Δ) is the set of pages used in the last Δ references. The working-set algorithm and WSClock implement this idea with virtual time and a threshold τ.
- The workload determines the policy: with no locality the policies are equal, with hot pages LRU wins, and on a loop slightly larger than memory LRU gets a zero hit rate.

- Frames are distributed equally, proportionally, or by priority; the victim is chosen locally or globally. Thrashing begins when $\Sigma\ |WS_i| > m$; the working-set strategy and PFF prevent it by adjusting the degree of multiprogramming.
- Design decisions: the paging daemon and watermarks, prepaging (the α threshold), the optimal page size $\sqrt{(2se)}$, TLB reach and huge pages, row-major traversal, pinning of I/O buffers.
- Kernel memory is allocated by the buddy allocator in power-of-two blocks, and buddies are found and merged with XOR; for small objects the slab allocator (SLUB on Linux) is used.
- Linux uses active/inactive lists, MGLRU, zswap, and zram; Windows uses working sets, standby lists, and memory compression.

9.15 Key terms

virtual memory virtual address demand paging page fault backing store (swap space)
effective access time copy-on-write memory-mapped file page cache page replacement
reference string reference bit / dirty bit Belady's anomaly optimal page replacement (OPT)
least recently used (LRU) second chance clock algorithm aging locality working set thrashing
translation lookaside buffer (TLB) internal fragmentation buddy system slab allocator
compression

9.16 Test Yourself

1. The reference string is 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6 and the number of frames is 4. How many page faults do FIFO, LRU, and OPT incur?

   Answer

   FIFO 14, LRU 10, OPT 8. LRU's victims are, in order, 3, 4, 5, 6, 1, 7; OPT's are 4, 5, 1, 7. Since there are 7 distinct pages, at least 7 faults are unavoidable; OPT comes very close to this. With 3 frames the results are FIFO 16, LRU 15, OPT 11.

2. Memory access takes 50 ns and the page-fault service time is 10 ms. What is the largest p that keeps the slowdown at or below 20%?

   Answer

   A 20% slowdown means EAT < 60 ns. 10 ms = 10,000,000 ns. $60 > 50 + (10{,}000{,}000 - 50)\,p \rightarrow p < 10 / 9{,}999{,}950 \approx 1.0 \times 10^{-6}$: fewer than one fault in about a million accesses.

3. With four free frames and the string 1 2 3 4 1 5 2 6 1 3, how many faults does the clock algorithm

incur (the hand starts at frame 0 and moves only on a fault)? Which pages are evicted?

Answer

8 faults. The first four references are loaded into the free frames; 1 is a hit. When 5 arrives, all four pages have R = 1; the hand clears them all, makes a full circle, and evicts 1. 2 is a hit (R ← 1). When 6 arrives, the hand clears the R of 2 and evicts 3, which has R = 0. When 1 arrives, 4 (R = 0), under the hand, is evicted. When 3 arrives, the hand clears the R of 5 and evicts 2. The victims are 1, 3, 4, 2.

4. The reference string (from time 1) is 1 2 1 3 2 4 4 5 4 5 6 1, with Δ = 4. What are W(4), W(8), and W(12)?

Answer

W(4) = {1, 2, 3} (1 2 1 3), W(8) = {2, 4, 5} (2 4 4 5), W(12) = {1, 4, 5, 6} (4 5 6 1). The sets drift as the process moves to new pages; W(12) has grown during a locality transition.

5. What is the reach of a TLB with 2,048 entries and 16 KB pages? Is it sufficient for a process whose working set is 48 MB?

Answer

2,048 × 16 KB = 32,768 KB = 32 MB. A 48 MB working set does not fit; some of the accesses go to a page-table walk through a TLB miss. 2 MB huge pages would raise the reach to 4 GB.

6. The process size is 2 MB and the page-table entry is 4 bytes. According to the overhead model $se/p + p/2$, what is the optimal page size?

Answer

$p = \sqrt{(2se)} = \sqrt{(2 \times 2^{21} \times 2^{2})} = \sqrt{(2^{24})} = 2^{12} = 4$ KB.

7. In a 128 K buddy region, A = 10 K, B = 30 K, and C = 20 K are requested in order; then A and C are released. Where does each request go, what is the total internal fragmentation, and what are the free blocks at the end?

Answer

A is rounded up to 16 K: 128 → 64 + 64 → 32 + 32 → 16 + 16; A takes 0–16 (6 K). B is 32 K: the free block 32–64 is ready (2 K). C is 32 K: the free block 64–128 is split and C takes 64–96 (12 K). Total 20 K. When A is released, its buddy 16–32 is free and 0–32 forms; its buddy holds B, so stop. When C is released, its buddy 96–128 is free and 64–128 forms; its buddy 0–64 is split, so stop. The free blocks are 0–32 and 64–128; B is at 32–64.

8. After `fork()`, 50 pages are shared by copy-on-write. The child writes to pages 0–4; then the parent writes to pages 0, 1, and 10. How many pages are copied?

   Answer

   6. The child's five writes produce five copies. When the parent writes to 0 and 1, the reference count of those frames is now 1; no copy is needed and the pages are simply made writable. Page 10 is still shared (reference count 2) and produces the sixth copy.

9. The 8-bit aging counters of three pages (s0, s1, s2) start at 0. The R bits seen at four ticks are: tick 0: 1 1 0, tick 1: 1 0 1, tick 2: 0 1 1, tick 3: 0 1 0. Which page do aging and NFU evict?

   Answer

   The counters are s0 = 00110000 (48), s1 = 11010000 (208), s2 = 01100000 (96). Aging evicts s0: it was not used in the last two ticks. The NFU counters are 2, 3, 2; s0 and s2 are tied and NFU cannot tell them apart. Yet s2 was used one tick earlier and is more recent.

10. On a server CPU utilization has fallen to 10% and the swap disk is constantly busy. An administrator says “the CPU is idle” and wants to add new batch jobs to the queue. What happens? What is the right intervention?

    Answer

    The system is thrashing: $\Sigma\ |WS_i|$ exceeds memory and the processes are waiting for pages. New work shrinks every process's frame share further, faults increase, and CPU utilization falls even more. The right move is to lower the degree of multiprogramming: suspend or swap out some processes and, as a lasting fix, add RAM. PFF makes this decision automatically from the per-process fault rate.

9.17 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters “Swapping: Mechanisms,” “Swapping: Policies,” and “Complete VM Systems”: they compare replacement policies on no-locality, 80-20, and looping workloads and describe the virtual memory design of VAX/VMS and Linux; free at ostep.org.
- P. J. Denning, “The Working Set Model for Program Behavior,” *Communications of the ACM*, 11(5), 1968: the foundational paper that introduced the working-set concept and its use against thrashing.
- J. Bonwick, “The Slab Allocator: An Object-Caching Kernel Memory Allocator,” USENIX Summer 1994: presents the object-cache idea and the Solaris measurements in a short and readable form.

- Linux kernel documentation, `admin-guide/mm/multigen_lru.rst` and `admin-guide/mm/zswap.rst`: describes the generations of MGLRU and the settings of zswap in the kernel's own documentation.

9.18 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th edition, Wiley, 2018 — Chapter 10: Virtual Memory.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th edition (2014) / 5th edition (2023), Pearson — Chapter 3: Memory Management.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Swapping: Mechanisms," "Swapping: Policies," "Complete VM Systems."
4. L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal* 5(2), 1966; P. J. Denning, "The Working Set Model for Program Behavior," *CACM* 11(5), 1968.
5. R. W. Carr, J. L. Hennessy, "WSCLOCK — A Simple and Effective Algorithm for Virtual Memory Management," SOSP 1981; J. Bonwick, "The Slab Allocator," USENIX Summer 1994.
6. Linux kernel documentation: `admin-guide/mm/multigen_lru.rst`, `admin-guide/mm/zswap.rst`, `admin-guide/mm/transhuge.rst`, `admin-guide/blockdev/zram.rst`.
7. P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7th edition, Part 1, Chapter 5: Memory management.

File Systems

When a process ends, its memory is reclaimed and the data in it is lost. Everything that must persist is written to disk. A disk, however, knows only numbered blocks: “read this block,” “write this block.” The file system builds on top of these raw blocks files with names, sizes, and owners, nested directories, and access rights. These notes first describe the face the user sees: the file, attributes, system calls, open-file tables, directories, links, and permissions. They then turn to implementation: partitioning the disk, allocating blocks to files by contiguous, linked-list, FAT, and i-node methods, tracking free space, and caching. The last sections address consistency after a crash: fsck, journaling, copy-on-write, and log-structured file systems. The chapter closes with VFS and NFS.

LEARNING OUTCOMES

File abstraction
Define a file by its name, type, attributes, and operations; trace open-file tables and file descriptors.

Directories
Compare directory structures, resolve a path step by step, and explain hard and symbolic links and mounting.

Protection
Read the rwx permission bits and the special bits; distinguish what r, w, and x mean on a file and on a directory.

Allocation
Apply contiguous, linked-list, FAT, and i-node allocation to the same file; compute the maximum file size and the disk block of a given file offset.

Free space and performance
Compute the sizes of the bitmap, list, grouping, and counting methods; explain the buffer cache, the page cache, and read-ahead.

Reliability and real systems
Compare fsck, journaling, copy-on-write, and LFS; recognize NTFS, ext4, FAT32/exFAT, VFS, NFS, and TRIM on SSDs.

10.1 Why a file system?

While a program runs, its data lives in its address space. This place is inadequate for three requirements. The first is size: a database or a video archive may be too large for the address space, or even for main memory. The second is **persistence**: when the process ends, or even when the machine is switched off, the data must stay where it is. The third is **concurrent access**: many processes must be able to read and modify the same data. Files written to disk meet these three requirements.

The disk offers the operating system a very plain interface. Both a hard disk and an SSD present themselves as a long array of numbered, fixed-size blocks. The number of a block is called its **logical block address** (LBA). The disk knows two operations: read block k and write block k. Which block belongs to which file, what the file is called, and who may read it are questions the disk does not care about. The file system answers them.

> **Definition 10.1**
>
> A **file** is a named, persistent sequence of bytes together with its attributes. A **file system** is the operating system component that turns the blocks on disk into files and directories; it names, places, and protects them.

A file is a logical unit, similar to an address space in memory. Processes create, read, write, and delete files. The operating system manages naming, internal structure, access, protection, and the placement on disk. Figure 10.1 shows these layers. The application talks to files through the calls `open`, `read`, `write`, and `close`. The file system translates a request such as “byte 5000 of the file” into an address such as “disk block 1729.” The block device then simply reads that block.

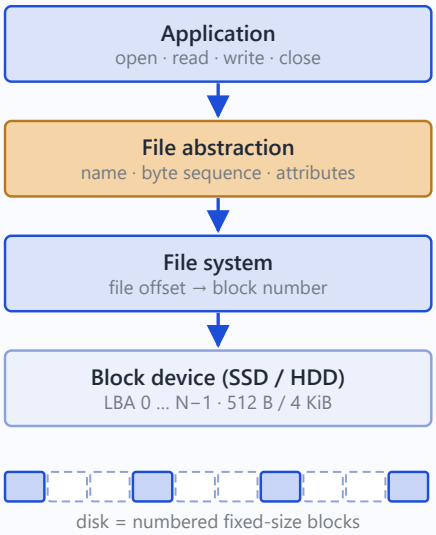


Figure 10.1: The layers from the application to the block device. The file abstraction hides the blocks; the disk knows only numbered, fixed-size blocks.

The subject can be viewed from two sides. The **user** is concerned with what is visible: the file's name, its type, the directory it is in, its permissions, and which operations can be performed. The **implementer** thinks about what goes on inside: how are blocks distributed to files, how is free space tracked, and how do the structures stay consistent if the power fails? These notes follow the user's view first and then the implementer's.

10.2 The file concept

10.2.1 File names and extensions

The process that creates a file gives it a name; after that process ends, other processes find the file by that name. Naming rules vary from file system to file system. Table 10.1 compares four common systems. The FAT system of old MS-DOS used the “8.3” format: a name of at most eight characters, a dot, and an extension of three characters. Long names were stored in additional directory entries in VFAT, which came with Windows 95. The short name continued to exist in truncated form such as `REPORT~1.TXT` for backward compatibility.

File system	Name length	Case	Note
FAT (MS-DOS)	8.3 (e.g. `REPORT~1.TXT`)	insensitive	VFAT long name in extra entries: 255 UTF-16
NTFS	255 UTF-16 units	preserving, insensitive	`\ / : * ? " < > |` forbidden
ext4, XFS	255 bytes	sensitive	only `/` and NUL forbidden

File system	Name length	Case	Note
APFS	255 UTF-8 characters	insensitive by default on macOS	Unicode normalization

Table 10.1: File-naming rules.

On a “case-sensitive” system, `Report.txt` and `report.txt` are two different files. ext4 is like this. NTFS stores the name as written but ignores case when searching: the two names find the same file. All of today's systems store names as Unicode, so names with non-ASCII characters cause no trouble. One should also note that ext4's limit is in bytes, not characters. In UTF-8 a letter such as “ş” takes two bytes, so a name dominated by such characters reaches the limit before 255 characters.

The part of the name after the last dot is called the **extension**: the extension of the file `thesis.pdf` is `pdf`. In UNIX the extension is only a convention; the kernel attaches no meaning to it. Programs may. For example, `gcc` compiles a source file with the extension `.c` and merely links an object file with the extension `.o`. On Windows an extension is associated with a program in the registry. Double-clicking the file opens that program. This association also creates a security trap. Because Explorer hides known extensions by default, a program named `invoice.pdf.exe` appears on screen as `invoice.pdf`.

File systems have changed over time. Older Windows versions used FAT16 and FAT32; current Windows uses NTFS for the system disk. FAT32 and exFAT live on today in removable media such as USB sticks and SD cards, because almost every device can read them.

10.2.2 Magic numbers and file structure

An extension is a name the user can change; it does not prove the real type of the file. A more reliable clue is the first bytes of the file. Many formats begin with a particular byte sequence. This sequence is called a **magic number**. Figure 10.2 gives a few examples: a PNG image begins with `89 50 4E 47`, a PDF with `%PDF`, Linux's executable format ELF with `7F 45 4C 46`, and a ZIP archive with `PK`. The UNIX command `file report` tells the type by looking at these bytes, not at the extension. An interesting consequence: `docx`, `xlsx`, and `pptx` files also begin with `PK`, because they are ZIP archives containing XML files.

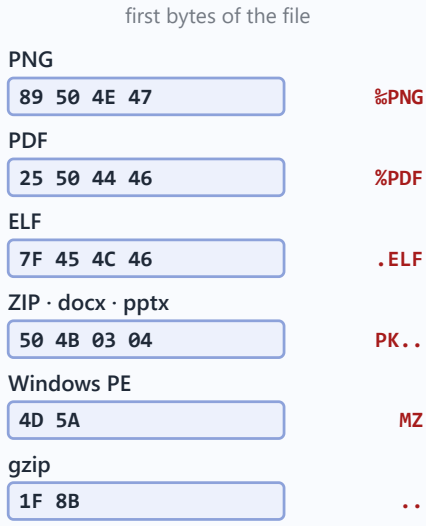


Figure 10.2: Magic numbers that identify the file type: the header bytes of PNG, PDF, ELF, ZIP, Windows PE, and gzip.

How the inside of a file is organized is called the **file structure**. Figure 10.3 shows three structures seen in history. In the **byte-sequence** structure in (a), the operating system sees the file only as an ordered sequence of bytes. Anything can be put in it; the application determines the structure. This is the most flexible choice, and UNIX, Windows, and macOS all use it. In the **fixed-length record** structure in (b), the unit of reading and writing is the record; 80-byte records are a legacy of the 80 columns of punched cards. This structure lives on today in mainframe data sets. In the **record tree** in (c), each record has a key field. A request such as "fetch the record whose key is Pony" is supported directly.

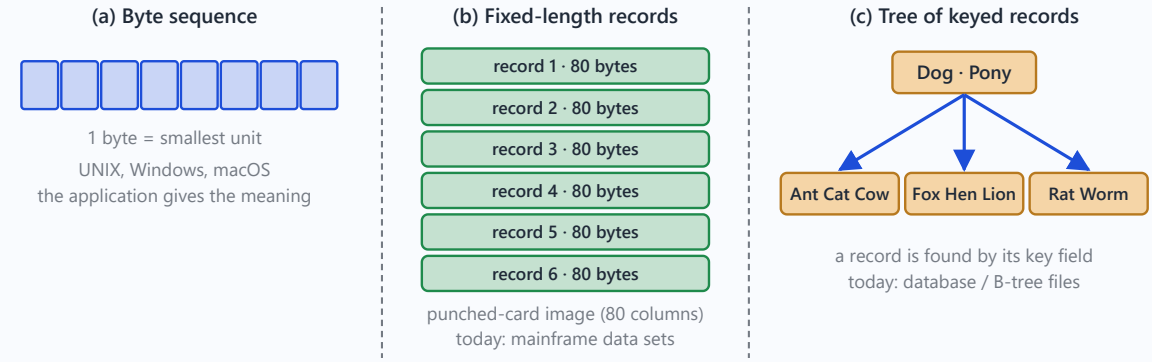


Figure 10.3: Three file structures: (a) a sequence of bytes, (b) a sequence of fixed-length records, (c) a tree of records searched by key.

Today access by key is done not in the operating system but in libraries and databases. SQLite places a B+ tree inside an ordinary byte-sequence file. The kernel thus stays simple, and complex structures are built in user space.

Binary files also have their own internal layout. Figure 10.4 gives three examples. (a) In the `a.out` format of old UNIX, the file opens with a header. The header holds the magic number, the sizes of the code, data, and BSS segments, the size of the symbol table, and the entry point. The header is followed by the code, data, relocation bits, and the symbol table. (b) Linux's current format, ELF, builds on the same idea more flexibly: the program headers are aimed at the loader and the section headers at the linker. Its counterpart on Windows is PE, and on macOS the Mach-O format. (c) An archive file (`.a` or `.lib`) stores compiled but unlinked object modules (`.o`) one after another, each preceded by a member header carrying its name and size. The kernel runs a file as a program only if it begins with the right magic number.

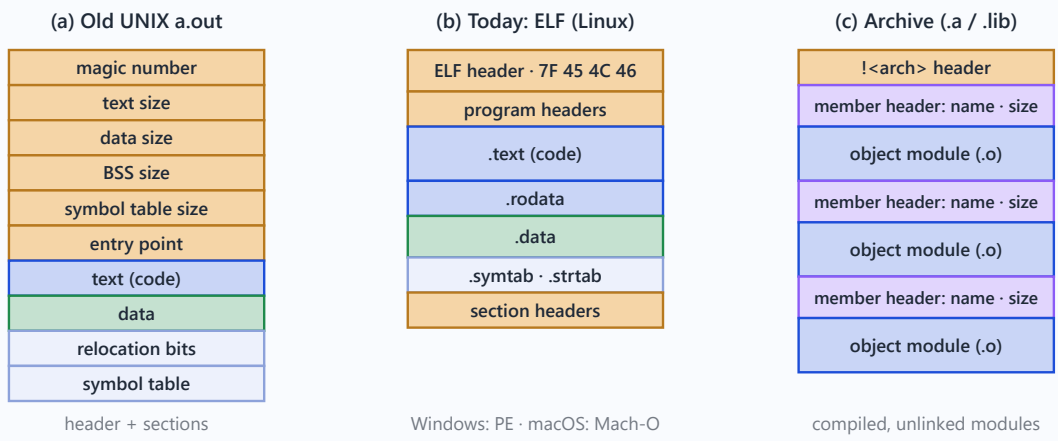


Figure 10.4: Executable and archive formats: (a) old a.out, (b) ELF, (c) an object-module archive.

10.2.3 File types

UNIX divides files into several types. **Regular files** hold user data. They are either text (lines of printable characters; the line ending is LF in UNIX and CRLF in Windows) or binary (only the program that uses it knows its structure). **Directories** are system files that hold the structure of the file system. **Special files** model input/output devices. Character devices (terminal, serial port) deliver bytes in sequence. Block devices (disks) are open to random access in blocks.

UNIX's famous principle “everything is a file” says that devices, pipes, and sockets are also used through the same `open` / `read` / `write` interface. For example, anything written to `/dev/null` disappears, and reading from it returns end-of-file at once. Table 10.2 lists the types together with the first character of the `ls -l` output.

Letter	Type	Example
`-`	regular file (text or binary)	`notes.txt` , `/bin/ls`
`d`	directory	`/home`
`l`	symbolic link	`/bin → usr/bin`
`c`	character device (serial, byte by byte)	`/dev/tty` , `/dev/null`
`b`	block device (block by block, random)	`/dev/nvme0n1` , `/dev/sda`

Letter	Type	Example
`p`	named pipe (FIFO)	`mkfifo channel`
`s`	UNIX socket	`/run/systemd/notify`

Table 10.2: UNIX file types and their letters in the `ls -l` output.

10.2.4 Attributes, stat, and ls -l

Besides the name and the data, the operating system keeps a set of information for every file. These are called **attributes** or **metadata**. Table 10.3 lists the common attributes.

Attribute	Meaning	Where / how
Name, type, location	the name is in the directory entry; the type and block addresses are in the i-node or the MFT record	all systems
Size	logical size (`st_size`) and allocated space (`st_blocks`)	the two differ for a sparse file
Owner, group	UID / GID; SID on Windows	`chown` , `icacls`
Protection	rwx bits, ACL	`chmod` , `setfacl`
Timestamps	access (atime), content change (mtime), i-node change (ctime), creation (btime)	`stat` ; relatime on Linux
Link count	number of names that point to the file	blocks are freed at 0
Read-only, Hidden, System	read-only, hidden, system file	Windows `attrib` ; in UNIX hidden = name starts with a dot
Archive	changed since the last backup	cleared by the backup tool
Compressed, Encrypted	compressed / encrypted	NTFS, Btrfs, APFS; fscrypt, BitLocker
Sparse, Temporary	zero blocks not on disk; temporary (cache hint)	NTFS flags
Reparse point	special handling: symbolic link, junction, cloud placeholder	NTFS
Extended attributes	name = value pairs	xattr (SELinux label), NTFS ADS

Table 10.3: File attributes.

Some attributes need explanation. In a **sparse file**, regions that were never written take no space on disk; when read, they return zeros. A sparse file of size 1 GiB can therefore occupy only a few kilobytes on disk: `st_size` is 1 GiB but `st_blocks` is small. The three timestamps are often confused. **mtime** is the moment the content was last changed, and **ctime** the moment the i-node (permissions, owner, link count) was last changed; ctime is not the “creation time.” The creation time (btime) is a separate field. Updating atime on every read would turn every read into a write. Linux therefore uses *relatime* by default: atime is updated only if it is older than mtime or older than a day.

Some Windows attributes are also misunderstood. The Temporary attribute does not delete the file automatically; it merely tells the cache manager “do not hurry to write this to disk.” Symlink, Device, and Junction are not attributes but file types. Fields such as Creator and Max size belong to old

systems.

A program reads attributes with the `stat` system call. The call fills in the following structure:

```
struct stat {
  dev_t     st_dev;      /* device (file system)   */
  ino_t     st_ino;      /* i-node number          */
  mode_t    st_mode;     /* type + permission bits */
  nlink_t   st_nlink;    /* hard link count        */
  uid_t     st_uid;      /* owner user             */
  gid_t     st_gid;      /* owner group            */
  dev_t     st_rdev;     /* no. if a device file   */
  off_t     st_size;     /* size (bytes)           */
  blksize_t st_blksize;  /* preferred I/O size     */
  blkcnt_t  st_blocks;   /* 512 B units allocated  */
  struct timespec st_atim; /* last access  */
  struct timespec st_mtim; /* last content */
  struct timespec st_ctim; /* last i-node  */
};
```

Most of the fields correspond to the attributes in the table. `st_ino` is the file's i-node number; within one file system it identifies a file uniquely. `st_mode` carries the type and permission bits together. `st_nlink` is the hard-link count. `st_blocks` gives the space allocated to the file in units of 512 bytes; even if the block size is 4 KiB, the unit is 512 B. Since POSIX.1-2008 the times are `struct timespec` fields with nanosecond resolution. The old name `st_atime` remains as a macro for backward compatibility. On Linux the creation time is read with a separate call, `statx()`.

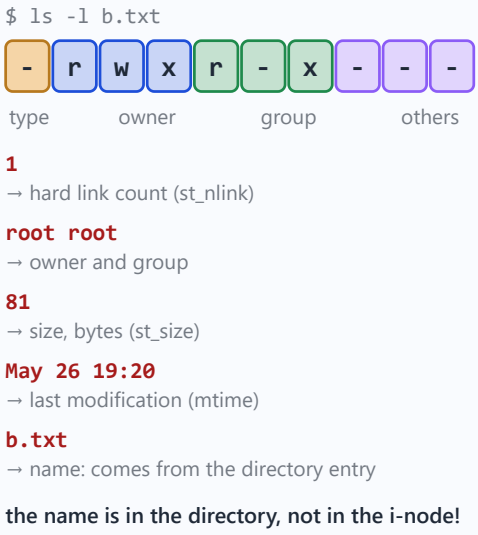


Figure 10.5: The fields of the `ls -l` output: type, permissions, link count, owner, group, size, modification time, and name.

`ls -l` converts this structure into a form humans can read (Figure 10.5). The first character is the type, and the next nine characters are the permissions of the owner, the group, and others. Then come the link count, the owner and group names, the size in bytes, the mtime, and the name. The name is not in the `stat` structure: the name lives in the directory entry, not in the i-node. This is also why one i-node can be given several names.

10.2.5 File operations and system calls

A file is an abstract data type: it is defined by the operations that can be performed on it. Table 10.4 gives the basic operations and their POSIX counterparts.

Operation	POSIX call	What it does
Create	`open(O_CREAT)`, `creat`	creates an empty file and a directory entry and sets the attributes
Delete	`unlink`	removes the name; when the link count reaches 0 and the last descriptor is closed, the space is freed
Open	`open`	resolves the path, brings the i-node into memory, returns a descriptor (fd)
Close	`close`	releases the fd and, if necessary, the table entries
Read / write	`read`, `write`	reads / writes from the file position and advances the position
Append	`open(O_APPEND)`	every write goes atomically to the end of the file
Seek	`lseek`	changes the position (random access); performs no I/O
Read / write attributes	`stat`, `chmod`, `chown`, `utimensat`	reads / changes the attributes
Rename	`rename`	changes the name; atomic within the same file system
Truncate, map	`ftruncate`, `mmap`	shortens or extends the size; maps the file into the address space

Table 10.4: File operations and POSIX system calls.

A few details in the table matter. `unlink` deletes not the file but a name. The file's blocks are freed only when no name remains and the last descriptor holding it open has been closed. This is why a process can keep reading a file that is open even after the file is deleted. `lseek` only changes the position counter in the kernel; it does not touch the disk. On a file opened with `O_APPEND`, the operation “go to the end and write” is indivisible. Even if two processes write to the same log file at the same time, their records do not overlap.

That `rename` is atomic is the basis of a common pattern in applications (Figure 10.6). To save a document safely, the new content is first written to a temporary file. Then `fsync` is used to wait until the data reaches the disk. Finally `rename(tmp, target)` is called. Whenever the crash occurs, the disk shows either the whole old file or the whole new file; a half-written file never appears.

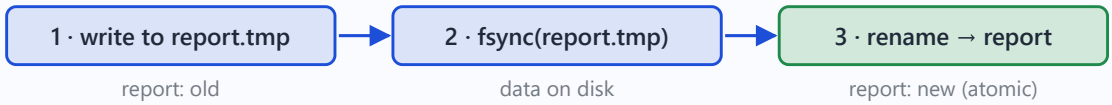


Figure 10.6: Safe saving: write to a temporary file, fsync, and rename. At every moment either the old or the new file is visible.

Common mistake

Opening the document in place, truncating it with `O_TRUNC`, and writing the new content. If a crash occurs between the truncation and the write, neither the old nor the new content remains. Skipping the `fsync` step is also dangerous: `rename` can reach the disk before the data itself, and the result is a "new" file of zero bytes.

10.2.6 Open-file tables

Resolving the file's path from scratch at every `read` call would be very expensive. The file is therefore opened first. `open` resolves the path once, brings the file's i-node into memory, and returns a small integer to the process. Later calls refer to the file by this number.

Definition 10.2

A **file descriptor** (fd) is an index into the descriptor table the kernel keeps for each process. 0, 1, and 2 are standard input, standard output, and standard error, respectively. `open` always returns the smallest free number.

The kernel tracks open files at three levels (Figure 10.7). At the top is each process's own **descriptor table**; its entries point to the middle table. In the middle is the system-wide **open-file table**. Each of its entries represents one opening and carries two important pieces of information: the current position in the file (the offset) and the access mode (read-only, read-write). At the bottom is the **in-memory i-node table**. Here each file appears exactly once, however many times it has been opened.

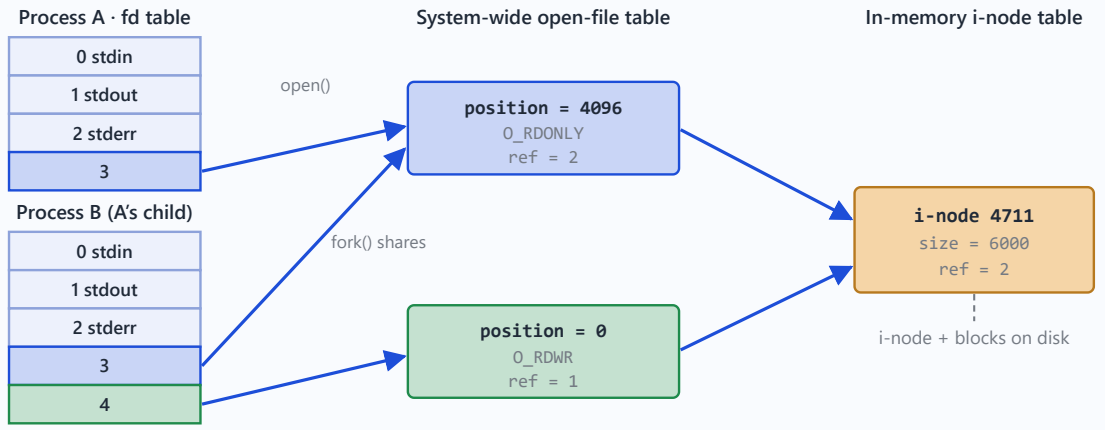


Figure 10.7: The per-process fd tables, the system-wide open-file table, and the in-memory i-node table. A's fd 3 and its child B's fd 3 share the same entry; B's fd 4 is a second opening of the same file.

The consequence of this separation can be summarized as follows. `fork` and `dup` copy the entry in the descriptor table; the copies point to the same open-file entry. They therefore share the position: if one reads 10 bytes, the other's position also advances by 10 bytes. Opening the same file a second time with `open`, however, creates a new entry. The position of this entry is separate and starts at 0. In both cases the i-node is single. The reference count in the open-file entry records how many descriptors point to it, and the reference count in the i-node records how many entries point to it. When a count drops to 0 the corresponding structure is released. On Windows the same role is played by each process's **handle** table.

Example 10.1

The contents of the file `letters.txt` are `ABCDEFGHIJ` (10 bytes). A process opens the file and reads 4 bytes. Then it calls `fork`; the child reads 3 bytes from the same descriptor and exits. The parent waits for the child, opens the file a second time, and reads 2 bytes from the new descriptor. Finally it reads 2 bytes from the first descriptor. Which bytes are read?

1. **First read.** `open` returns fd 3; entry E1 is created, position 0. `ABCD` is read, and E1's position is 4.
2. **fork.** The child's fd 3 also points to E1; E1's reference count is 2.
3. **Child.** It reads 3 bytes from the shared position 4: `EFG`. E1's position is 7. When the child exits, the reference count drops to 1 and the position stays 7.
4. **Second open.** A new entry E2 is created, fd 4, position 0. `AB` is read; E1 is unaffected.
5. **The parent's fd 3.** E1's position is 7: `HI` is read, and the position becomes 9.

The parent's last read is `HI`, not `EF`, because the child advanced the shared position.

10.2.7 Code trace: open, read, close

The program below reads the 6000-byte file `notes.txt` three times with a 4096-byte buffer and closes it.

```
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  char buf[4096];
  int fd = open("notes.txt", O_RDONLY);
  ssize_t n = read(fd, buf, sizeof buf);
  n = read(fd, buf, sizeof buf);
  n = read(fd, buf, sizeof buf);
  close(fd);
  return 0;
}
```

`open` resolves the path and brings the file's i-node (let its number be 4711) into the in-memory table. It creates an open-file entry with position 0 and returns the smallest free number, 3; 0, 1, and 2 are the terminal descriptors inherited from the shell. The first `read` brings the file's first block into the cache (reading from disk if necessary) and copies 4096 bytes into `buf`. The position becomes 4096. The second `read` also asks for 4096 bytes, but only 6000 − 4096 = 1904 bytes remain in the file. The call returns 1904; the position becomes 6000. This is not an error: a read that returns fewer bytes than requested, a **short read**, is a normal situation. The third `read` returns 0. A return of zero means end of file (EOF); if it were an error, −1 would be returned and `errno` set. `close` frees fd 3. The entry's reference count drops to 0 and the entry is deleted. The i-node's count also becomes 0; it can now be evicted from memory. Table 10.5 summarizes these steps.

Step	Returns	Position	Entry ref.	i-node ref.
`open`	3	0	1	1
1st `read`	4096	4096	1	1
2nd `read`	1904	6000	1	1
3rd `read`	0 (EOF)	6000	1	1
`close`	0	—	0 (deleted)	0

Table 10.5: Trace of `open`, `read`, `close` (6000-byte file, 4096-byte buffer).

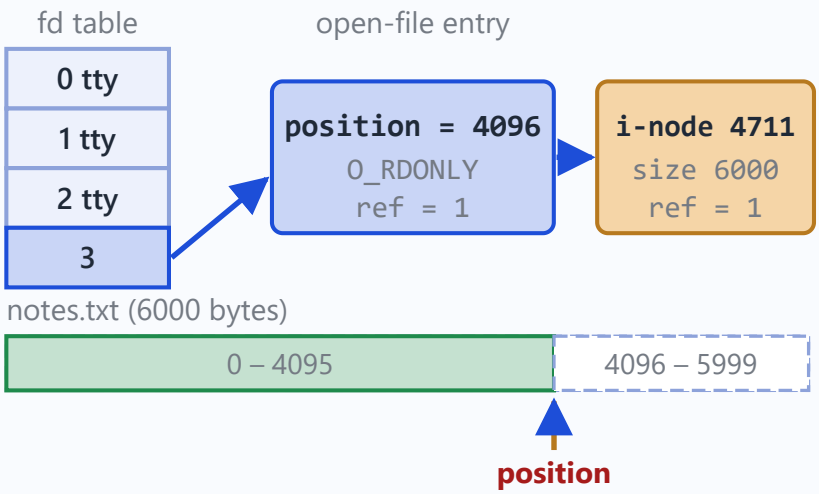


Figure 10.8: The tables after the first `read`: fd 3 points to the open-file entry with position 4096; the entry points to i-node 4711.

10.2.8 Code trace: copying a file

Copying is the simplest program that uses reading and writing together:

```
#define BUF 4096

int copy(const char *src, const char *dst) {
  char buf[BUF];
  int in  = open(src, O_RDONLY);
  int out = open(dst, O_WRONLY | O_CREAT
                      | O_TRUNC, 0644);
  ssize_t n;
  while ((n = read(in, buf, BUF)) > 0)
    write(out, buf, n);
  close(in);
  close(out);
  return 0;
}
```

The source is opened for reading and gets number 3. The destination is opened with three flags and gets number 4. `O_WRONLY` asks for write-only access. `O_CREAT` creates the file if it does not exist. `O_TRUNC` erases the contents if it does. When `O_CREAT` is used, the third argument is the permissions of the new file: `0644`, that is, read-write for the owner and read-only for the others. On each turn the loop reads at most `BUF` bytes and writes as many as it read. The assignment in the condition needs care: the return value of `read` is first stored in `n`, and then it is tested whether it is greater than 0.

Let `abc` be 10,000 bytes. The trace is given in Table 10.6. The first two turns copy 4096 bytes each; 8192 in total. The third `read` returns the remaining 10,000 − 8192 = 1808 bytes and `write` writes exactly 1808 bytes. The fourth `read` returns 0 and the loop ends. In all, 4 `read` and 3 `write` calls

are made.

Turn	`read` return (n)	`write`	Total copied
1	4096	4096 bytes	4096
2	4096	4096 bytes	8192
3	1808	1808 bytes	10.000
4	0 (EOF)	—	10,000; the loop ends

Table 10.6: Trace of `copy("abc", "xyz")` (abc is 10,000 bytes, BUF = 4096).

Common mistake

Writing `write(out, buf, BUF)` in the loop. On the last turn the buffer holds only 1808 valid bytes; the remaining 2288 bytes are garbage left over from the previous turn. The destination file becomes 12,288 bytes and its end is corrupt. The amount to write is always the `n` returned by `read`.

The code above does not check for errors, to keep it short. In a real program the −1 return of every call must be checked. `write` can also write fewer bytes than requested (for example when the disk is filling up or when writing to a pipe); the remainder must therefore be completed with an inner loop. The buffer size matters too. If a 6 MiB (6,291,456-byte) file is copied byte by byte, 6,291,457 `read` and 6,291,456 `write` calls are needed. If each entry into the kernel takes 0.3 µs, this overhead alone is about 3.77 seconds. With a 4096-byte buffer the number of calls falls to 1537 + 1536 = 3073 and the overhead to about 0.92 ms. 4096 bytes also matches the page and block size.

10.2.9 Access methods

The order in which the inside of a file is accessed is called the **access method** (Figure 10.9). In **sequential access** the file is read or written from beginning to end. The current position advances with every operation; to go back, one has to rewind. Tapes and cassettes worked this way. Even today it is the most common pattern for log files, video streams, and a compiler reading a source file.

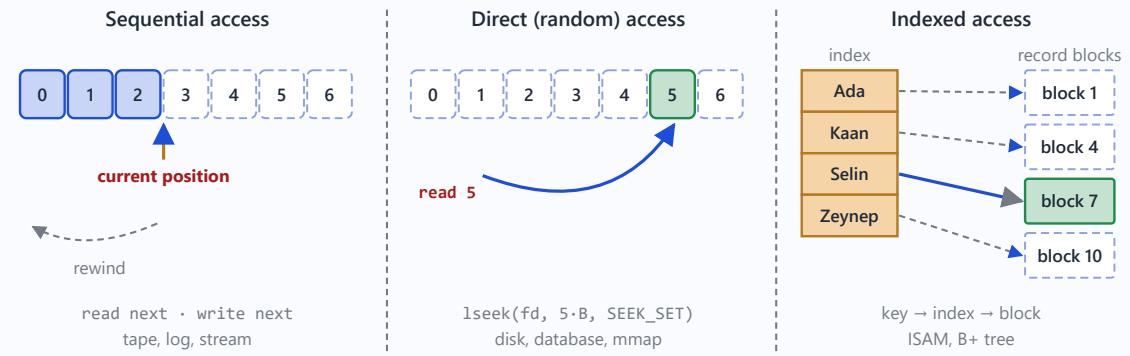


Figure 10.9: Sequential, direct, and indexed access.

In **direct** (random) access the file is a sequence of numbered blocks or fixed-length records. One can start at any wanted record: with records of B bytes, `lseek(fd, 5·B, SEEK_SET)` is enough for record 5. `pread` does the positioning and the reading in a single call and does not change the shared position. Disks and SSDs support direct access; databases use it. **Indexed access** is built on top of direct access. A small index maps a key to the block in which the record lies. If the index grows, an index of the index is kept. IBM's ISAM method and today's B+ trees are examples of this idea. Finally, in **memory-mapped access** (`mmap`) the file's pages are mapped into the address space; reading and writing turn into ordinary memory accesses.

10.3 Directories

10.3.1 Directory operations

> **Definition 10.3**
>
> A **directory** is a special file that maps file names to files. Each mapping is called a **directory entry**; in UNIX an entry maps a name to an i-node number.

A directory is also a file, but the user cannot modify it with `write` . Because a corrupt directory can make a whole subtree unreachable, the kernel changes the contents of directories only through certain calls. Table 10.7 lists these calls.

Call	Function
`mkdir`	creates an empty directory (containing only `.` and `..`)
`rmdir`	removes a directory; the directory must be empty
`opendir` / `closedir`	opens / closes a directory stream
`readdir`	returns the next entry (name, i-node no., type) in an open directory
`rename`	renames or moves a file or directory (atomic within the same file system)

Call	Function
`link`	adds a new name (hard link) to an existing file; the i-node's link count rises
`symlink`	creates a symbolic link containing the target path
`unlink`	removes a name; if the link count reaches 0 the file is deleted

Table 10.7: Directory operations.

A new directory has two entries from the start: `.` points to the directory itself and `..` to the parent. These two entries also affect the link counts (Figure 10.10). Let the i-node of `/home` be 20 and let it contain no other subdirectory. Its link count is 2: the `home` entry in the root and its own `.` entry. `mkdir /home/ali` gives the new directory i-node 31. The entry `ali → 31` is added to `/home`; the new directory contains `. → 31` and `.. → 20`. As a result the link count of i-node 31 becomes 2 (`ali` and `.`). That of i-node 20 rises to 3, because the new directory's `..` entry also points to it. The general rule is: the link count of a directory is 2 plus the number of its subdirectories.

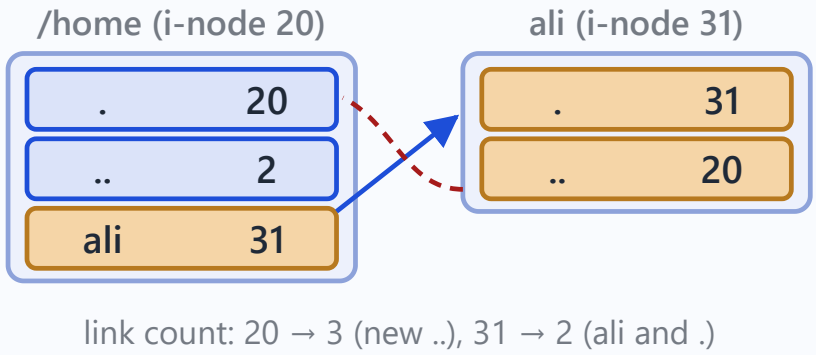


Figure 10.10: `mkdir /home/ali`: the entry `ali` is added to the parent directory; in the new directory `.` points to itself and `..` to the parent.

The name `link` can mislead. This call does not link one directory to another; it gives an existing file a second name. Giving hard links to directories is forbidden to ordinary users, because it can create cycles in the directory tree.

10.3.2 Directory structures

How directories are organized relative to one another has changed over time. The simplest structure is the **single-level directory** (Figure 10.11): all files sit in a single root directory. Early supercomputers such as the CDC 6600 and some embedded devices were like this. The structure is simple but causes two problems. Two users cannot use the same name, for example `test`. Files also cannot be grouped by subject.

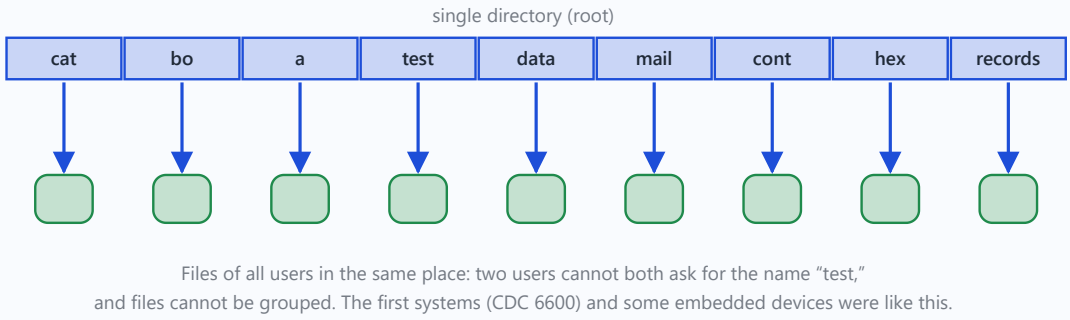


Figure 10.11: A single-level directory: all users' files in the same place.

In a **two-level directory**, a master file directory (MFD) gives each user a separate user file directory (UFD) (Figure 10.12). Name conflicts disappear: `ali/test` and `can/test` are different files. The full name of a file is now a path of the form “user/file.” System programs need a separate “user” directory. The idea of a **search path**, which says which directories to look in when a command is typed, comes from here. Sharing and grouping are still hard.

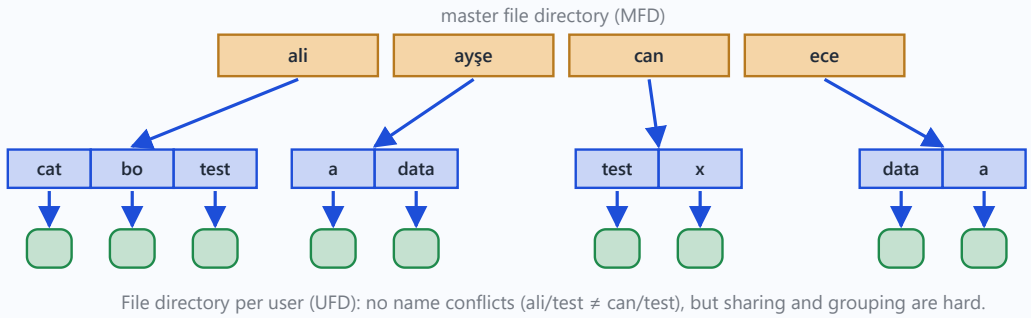


Figure 10.12: A two-level directory: one directory per user under the master directory.

All of today's general-purpose systems use a **tree** (hierarchical) structure. The user can create as many subdirectories as wanted, and every file has a single path starting from the root. When sharing is added to the tree, the structure becomes a **directed acyclic graph** (DAG) (Figure 10.13). In the figure `project.c` appears under both `ali/homework` and `ayşe/shared`: the same file has two names. Sharing brings two new questions. A backup or counting program that walks the tree may count the same file twice. When a shared file is deleted, it becomes unclear which name is deleted and when the data is freed. The link count answers the second question: the data is freed when the last name is deleted.

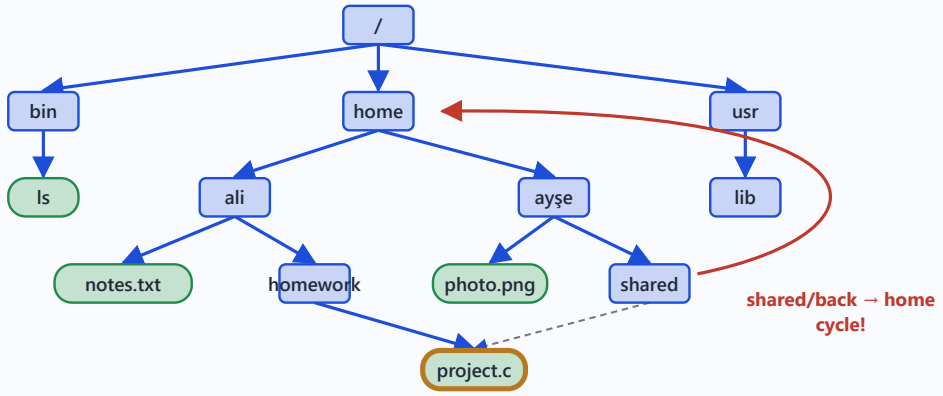


Figure 10.13: The shared `project.c` turns the structure into a directed acyclic graph. If a link named `shared/back` returns to `home`, a cycle arises in the general graph.

If giving links to a directory is allowed, a **general graph** arises. The link `shared/back → home` in the figure builds a cycle. A search walking the tree can go on forever. The link counts of a cycle that can no longer be reached from anywhere also do not fall to zero; finding these directories requires an expensive garbage collection. UNIX therefore forbids hard links to directories.

10.3.3 Path names

Definition 10.4

An **absolute path** starts from the root directory: `/home/sercan/os/slides`. A **relative path** starts from the process's working directory. The working directory is seen with `pwd` and changed with `cd`.

Two special names present in every directory make relative paths powerful: `.` is the current directory and `..` the parent. In the root, `..` again points to the root. In Figure 10.14 the working directory is `/home/sercan/course`. `../os/slides` first goes up to the parent, that is, `/home/sercan`, and from there down to `os` and `slides`. The separator is `/` in UNIX and `\` in Windows. The Windows API accepts `/` too. On Windows an absolute path begins with a drive letter (`C:\Users\sercan\os`), and a network share with two backslashes (`\\server\share`).

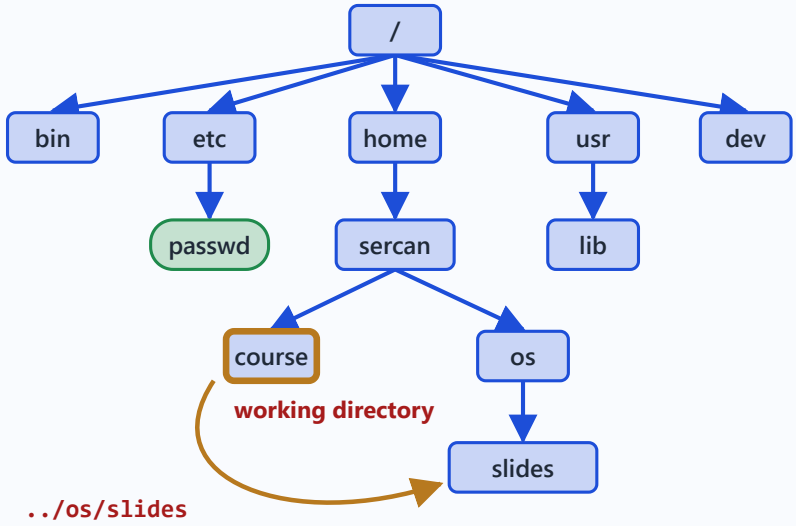


Figure 10.14: A UNIX directory tree; the working directory is `/home/sercan/course`, the relative path `../os/slides`.

Directory names have a history too. In the first UNIX systems user directories were under `/usr`; the `/usr/ast` in Tanenbaum's examples comes from this. Today user directories are under `/home` (`/Users` on macOS). `/usr` is reserved for read-only system software.

Example 10.2

The working directory is `/home/elif/homework/os`. Which file does `../../notes/./chapter10.txt` refer to?

1. The first `..`: `/home/elif/homework`.
2. The second `..`: `/home/elif`.
3. `notes`: `/home/elif/notes`.
4. `.` is the current directory; the path does not change.
5. `chapter10.txt`: the result is `/home/elif/notes/chapter10.txt`.

10.3.4 Path resolution

Converting a path to an i-node number is called **path resolution**. The kernel walks the path component by component. For each directory it first reads the directory's i-node; the i-node says in which block the directory's contents are. Then it reads that block and searches for the name of the next component. The entry it finds gives the next i-node number.

Tanenbaum's classic example resolves the path `/usr/ast/mbox` (Figure 10.15). Because the path is absolute, the search starts at the root. The i-node number of the root directory is known: in this example 1; in V7 UNIX and ext4 it is 2. The root i-node was brought into memory at mount time.

1. **The root directory block is read.** `usr` is searched for and i-node 6 is found.
2. **i-node 6 is read.** `/usr` is a directory; its contents are in block 132. The permission check is also done at this step: x permission is needed to pass through the directory.
3. **Block 132 is read.** `ast` is searched for and i-node 26 is found.
4. **i-node 26 is read.** The contents of `/usr/ast` are in block 406.
5. **Block 406 is read.** `mbox` is searched for and i-node 60 is found.

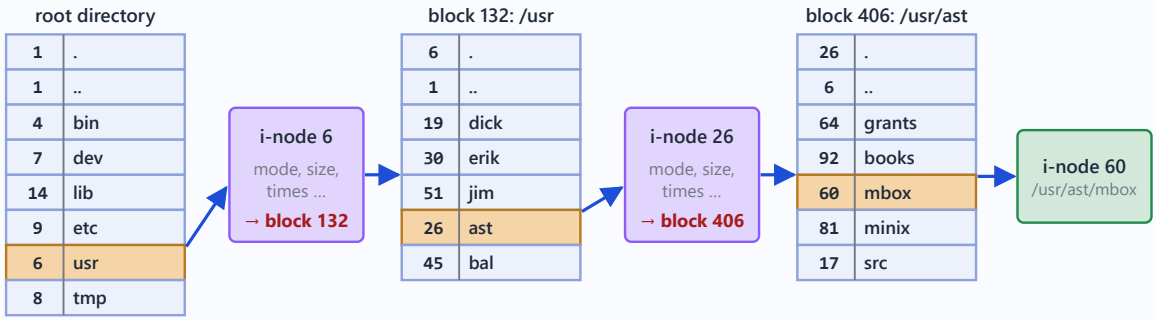


Figure 10.15: Resolving the path `/usr/ast/mbox`: root directory → i-node 6 → block 132 → i-node 26 → block 406 → i-node 60.

Without a cache this resolution costs 5 disk reads. Two more reads are needed to read the file's own i-node (60) and its first data block. The general rule is easy to derive. Let the root i-node be in memory and each directory fit in one block. Reaching the last i-node of a path with k components takes 2k reads, and reading the first data block as well takes 2k + 1 reads. In a relative path the search starts at the i-node of the working directory instead of the root; since the working directory is already in memory, it is cheaper. Linux keeps freshly resolved name → i-node mappings in the **dentry cache**. When the same path is opened a second time, most of the disk reads disappear.

Example 10.3

The first data block of the file `/var/log/system/events.txt` is to be read. Only the root i-node is in the cache; each directory fits in one block. What is the minimum number of disk reads?

1. There are four components in the path: `var`, `log`, `system`, `events.txt`.
2. Two reads are made for each component: the block of the directory in which it is searched and the i-node found. The root directory's block is the first read; four components cost 4 × 2 = 8 reads.
3. Finally the file's first data block is read: $2 \cdot 4 + 1 = 9$ reads in total.

10.3.5 Hard and symbolic links

Definition 10.5

A **hard link** is a second directory entry that points to an existing i-node. A **symbolic link** is a small special file with its own i-node whose content is the path of another file.

The difference between the two kinds of link is best seen with a trace (Table 10.8). `echo hello > a.txt` creates a new file. The directory has the entry `a.txt → 812` and the link count of i-node 812 is 1. `ln a.txt b.txt` makes a hard link. The new entry `b.txt → 812` points to the same i-node and the link count becomes 2. The two names are completely equal; the question of which is the "original" has no answer. `ln -s a.txt c.txt` makes a symbolic link: a new file with i-node 905 is created and its content is the string `"a.txt"`. The link count of i-node 812 does not change.

Command	Directory entries	link count of i-node 812	Result
`echo hello > a.txt`	a.txt → 812	1	new file
`ln a.txt b.txt`	a.txt → 812, b.txt → 812	2	two equal names
`ln -s a.txt c.txt`	+ c.txt → 905 (content "a.txt")	2	the symbolic link has its own i-node
`rm a.txt`	b.txt → 812, c.txt → 905	1	`cat b.txt`: hello; `cat c.txt`: ENOENT

Table 10.8: Trace of hard and symbolic links (simulated with Python).

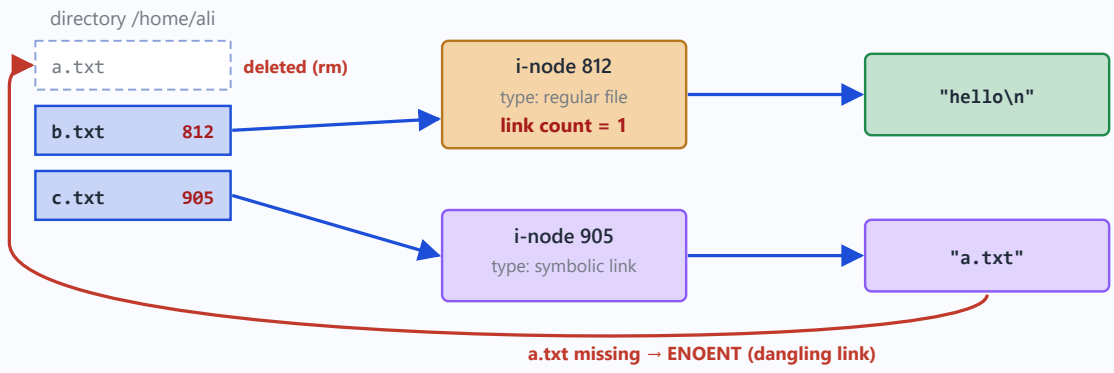


Figure 10.16: After `rm a.txt`: `b.txt` still points to the same i-node; `c.txt` now points to a name that no longer exists and is left dangling.

The last row shows the essence of the two kinds (Figure 10.16). `rm a.txt` deletes only a name and lowers the link count to 1. The data stays in place; `b.txt` works. A symbolic link, however, stores not the i-node but the name. When `c.txt` is opened, the kernel resolves the path `"a.txt"` again, cannot

find the name, and returns the error `ENOENT` (no such file). This situation is called a **dangling link**. If a new file is created with the same name, the symbolic link silently starts opening that new file; a hard link keeps pointing to the old data.

Tanenbaum points to a subtle side effect of hard links. B gives a hard link to user C's file; then C deletes the file. B's name and the data remain, but the owner of the i-node is still C. The space the file occupies also keeps being charged to C's quota.

Let us compare the two kinds. A hard link has no extra cost; but it can be made only within the same file system (an i-node number is meaningless in another file system) and cannot be given to directories. A symbolic link can point to directories, to files in other file systems, even to network paths. Its price is an extra path resolution at every access. Symbolic links can also form cycles. The kernel therefore limits the number of links followed while resolving a path; if the limit is exceeded, the error `ELOOP` is returned. On Windows a hard link is made with `mklink /H` and a symbolic link with `mklink`. For directories there is also a *junction* kind.

Example 10.4

In an empty directory (ext4), `mkdir d`, `echo x > d/a`, `ln d/a d/b`, `ln -s a d/c`, and `mkdir d/e` are run. What are the link counts of `d`, `d/a`, `d/b`, `d/c`, and `d/e`?

1. **d: 3.** The entry `d` in the parent directory, its own `.` entry, and the `..` entry of the subdirectory `e`.
2. **d/a and d/b: 2.** They are two names of the same i-node.
3. **d/c: 1.** It is the symbolic link's own i-node; since `stat` does not follow the link, the count of the link, not of the target, is returned.
4. **d/e: 2.** The entry in `d` and its own `.` entry.

If `rm d/a` is then run, the link count of `d/b` drops to 1 and the data stays. `d/c`, however, is left dangling because it looks for the name `a`.

10.3.6 Mounting

A machine has several file systems: the partitions of the system disk, an attached USB stick, a share from the network. UNIX combines them in a single tree.

Definition 10.6

Mounting means attaching the root of a file system to a directory in the existing directory tree, the **mount point**.

Figure 10.17 shows the process in three steps. Initially there are two separate file systems: the root system in ext4 format and a USB stick in exFAT format. `/mnt/usb` is an ordinary directory and contains `old.txt`. `mount /dev/sdb1 /mnt/usb` adds a record to the kernel's mount table. From then on, when path resolution reaches `/mnt/usb`, it jumps to the root i-node of the USB stick. `/mnt/usb/photos` is now the `photos` directory on the USB stick. `old.txt` is not deleted, only hidden. `umount /mnt/usb` removes the mount if there are no open files in that file system; if there are open files it returns the error `EBUSY`. When the mount is removed, `old.txt` becomes visible again.

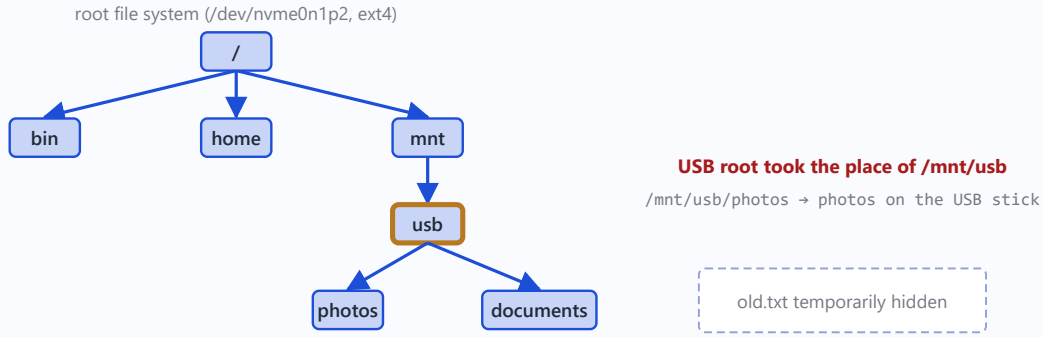


Figure 10.17: After `mount /dev/sdb1 /mnt/usb`: the root of the USB stick takes the place of `/mnt/usb`; the old content is temporarily hidden.

At boot time the root file system is mounted first. The others are mounted according to the definitions in the file `/etc/fstab` or in systemd units. The command `findmnt` shows the mount tree. Options can be given when mounting: `ro` mounts read-only, `noexec` disables the execution of programs on that system, and `nosuid` disables the effect of the setuid bit. With these options a safe boundary is drawn around an attached USB stick. Windows gives each volume a drive letter (`C:`, `D:`), but NTFS can also mount a volume on an empty folder.

10.4 Protection

10.4.1 The rwx permission bits

On a shared system it must be determined what each user may do with each file. UNIX uses a crude but fast method for this. Users are divided into three classes: the file's **owner** (u), the file's **group** (g), and **others** (o). Each class has three rights: read (r), write (w), and execute (x). The nine bits are thus written as three octal digits: r = 4, w = 2, x = 1. `chmod 754 script.sh` gives the owner 7 = rwx, the group 5 = r-x, and others 4 = r--, and `rwxr-xr--` appears in `ls -l`.

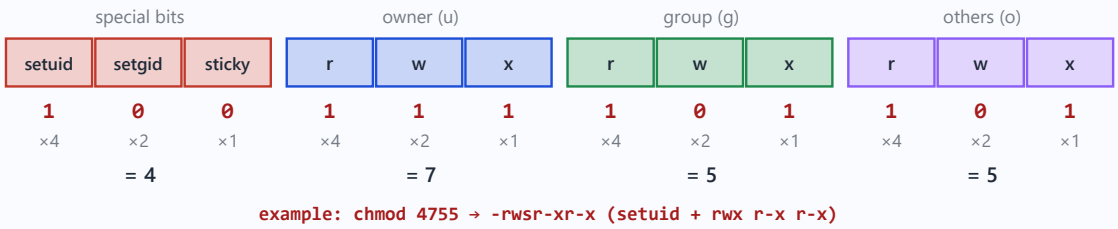


Figure 10.18: The 12 permission bits: special bits, owner, group, and others. Example 4755 = setuid + rwx r-x r-x.

The meaning of the rights differs on a file and on a directory (Table 10.9). On a file, r means reading the content, w modifying it, and x running it as a program. The content of a directory is name → i-node mappings. On a directory r means listing the entries (`ls`), w adding, deleting, or renaming entries, and x passing through the directory: path resolution and `cd` require it.

Right	On a file	On a directory
r	read the content	list the entries (`ls`)
w	modify the content	add, delete, or rename entries (the file's own permission is irrelevant)
x	run as a program	pass through (path resolution, `cd`)

Table 10.9: The meaning of r, w, and x on a file and on a directory.

This table has two surprising consequences. The first: deleting a file requires the w right not on the file but on the directory that contains it. Deleting removes a directory entry; it does not touch the file's content. A read-only file can be deleted in a directory with w permission. The second: having x without r on a directory is a meaningful combination. If the directory `/home/ali` is `drwx--x--x` , another user cannot list it. But if they know the name of a file inside, the path resolves, and they can read the file if the file's own r bit allows it.

The order of checking also matters. The kernel first looks at whether the requester is the owner of the file; if so, it applies only the owner's rights. If not, and the requester is in the file's group, it applies the group rights; otherwise it applies the rights of others. The first matching class is used and the later ones are not examined.

Example 10.5

`-r--rw-rw- 1 ali students 120 report.txt` . ali is also a member of the group students. Can ali write to the file?

1. The requester is ali and the owner of the file is ali: the owner class matches.
2. The owner's rights are `r--` ; there is no w.
3. The group and others classes are not examined at all. The write is rejected with `EACCES` .

The result looks odd at first sight: the owner cannot write to a file that the group, and even everyone, can write. If the owner wishes, they can give themselves w with `chmod u+w`, because the right to change permissions belongs to the owner.

10.4.2 Special bits, umask, and ACLs

In front of the nine bits there are three more **special bits** (Figure 10.18): setuid (4), setgid (2), and sticky (1). A program with the **setuid** bit runs with the identity of the file's owner, not of the user who runs it. The `passwd` command belongs to root and has the setuid bit; when an ordinary user changes their password, this is how it updates the password file that only root can write. `ls -l` shows setuid as an `s` in the place of the owner's x: 4755 → `-rwsr-xr-x`. The **sticky** bit is used on directories. In a directory that everyone can write, only the owner of the file, the owner of the directory, or root can delete or rename an entry. `/tmp` is like this: `drwxrwxrwt` (1777). If the sticky bit is set but the x right of others is off, the last character is a capital `T`: 1770 → `drwxrwx--T`.

The permissions of a new file are determined by the **umask**. Programs usually ask to create files with permission 666 and directories with 777. The kernel turns off the bits in the umask from this request: permission = request & ~umask. With umask 027, a new file gets 666 & ~027 = 640 (`rw-r-----`) and a new directory 777 & ~027 = 750 (`rwxr-x---`). With this setting the group can only read, and others can do nothing.

Common mistake

Subtracting the umask from the request. Octal subtraction gives 666 − 027 = 637: the group could write and execute, and others could do everything. The umask is a bit-by-bit mask: 666 & ~027 = 640. The two operations give the same result only if all the bits of the umask are also set in the request.

The distinction between owner, group, and others is coarse-grained. It does not meet a request such as “let ali read this file but not can, if both are in the same group.” An **access control list** (ACL) is used for this. An ACL keeps separate entries for named users and groups; on Linux it is set with `setfacl` and a `+` appears at the end of the permissions in the `ls -l` output. All of NTFS's protection is ACL-based.

10.5 Disk layout

10.5.1 Partitioning: MBR and GPT

A disk is divided into one or more **partitions**. Each partition can hold a separate file system; a partition can also be swap space, an LVM physical volume, or an encrypted container. The list of partitions is kept in a table at the beginning of the disk. This table has two formats (Figure 10.19).

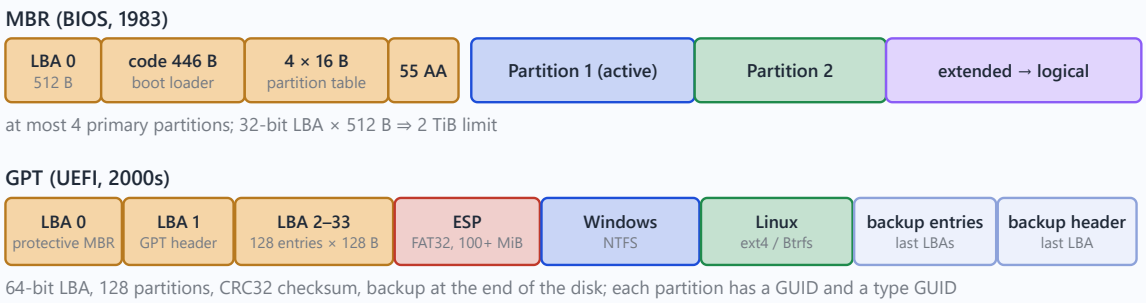


Figure 10.19: The MBR and GPT disk layouts.

MBR (master boot record) is the old format in use since 1983. The first sector of the disk, LBA 0, is 512 bytes. The first 446 bytes hold the boot loader code, followed by four 16-byte partition entries, and finally the signature `55 AA`. Each entry keeps the start and the length of the partition as 32-bit sector numbers. Two limits follow from this. At most four primary partitions can be created; for more, logical partitions are chained inside an "extended" partition. The second and more serious limit is the largest addressable area: $2^{32} \times 512$ B = 2 TiB. If a 3 TiB disk is partitioned with MBR, no partition can be defined on the last 1 TiB.

GPT (GUID partition table) is part of the UEFI standard and uses 64-bit LBAs. At LBA 0 sits a **protective MBR**; this dummy table prevents old tools that do not recognize GPT from thinking the disk is "empty" and writing over it. At LBA 1 is the GPT header, and at LBA 2–33 there are 128 partition entries of 128 bytes. Each partition has its own GUID and a type GUID indicating its type. The header and the entries are protected with a CRC32 checksum. Backups are kept at the end of the disk. Disks larger than 2 TiB and Windows 11 require GPT.

10.5.2 Booting

The format of the partition table also determines how the machine boots (Figure 10.20). When power is applied, the hardware tests itself (POST). The following steps differ in the two schemes.

The old **BIOS** runs in 16-bit real mode and does not recognize file systems. It loads the first sector of the disk, the MBR, into memory and runs it. The MBR code finds the partition marked "active" in the partition table and runs its first block, the **boot block**. This block loads the real boot loader (GRUB; NTLDR in old Windows). **UEFI**, on the other hand, runs in 32 or 64 bits and has its own drivers and FAT file-system support. It looks for files, not sectors. It reads the boot entries in non-volatile memory (NVRAM) and loads the `.efi` program in the EFI system partition (ESP, formatted as FAT32). The active flag is not used. In both schemes the boot loader then loads the kernel; the difference lies only in how the boot loader is reached.

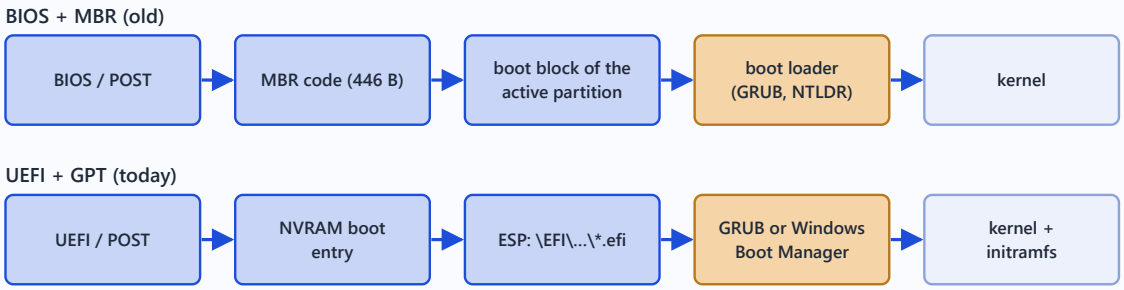


Figure 10.20: The BIOS/MBR and UEFI/GPT boot chains.

PCs have shipped with UEFI since around 2012. The BIOS compatibility mode (CSM) has been removed from most new systems since 2020.

10.5.3 The layout of a file system

The inside of a partition, although it varies with the file system, consists of similar parts (Figure 10.21). The first block of the partition is the **boot block**. This place is set aside even if the partition is not used for booting. The first block of the disk is the MBR; the two should not be confused.

> **Definition 10.7**
>
> The **superblock** is the block that holds the identity and basic parameters of the file system: the magic number, the block size, the numbers of blocks and i-nodes, the free counters, and the mount state. It is the first structure read when the file system is mounted.

The superblock is followed by the structure that tracks free space (a bitmap or a list), the i-node table with one i-node per file, the root directory, and the rest of the data blocks. In this layout every file access goes first to the i-node at the start of the disk and then to the distant data blocks. On a hard disk this means long arm movements. ext4 therefore divides the partition into **block groups**. With 4 KiB blocks each group has 32,768 blocks, that is, 128 MiB. Each group has its own bitmaps and i-node table. By placing a file's i-node and data in the same group, head movement is reduced. The idea comes from the cylinder groups of BSD's fast file system (FFS). Backups of the superblock and the group descriptors are kept in some groups; if the primary superblock is damaged, it is recovered from a backup.

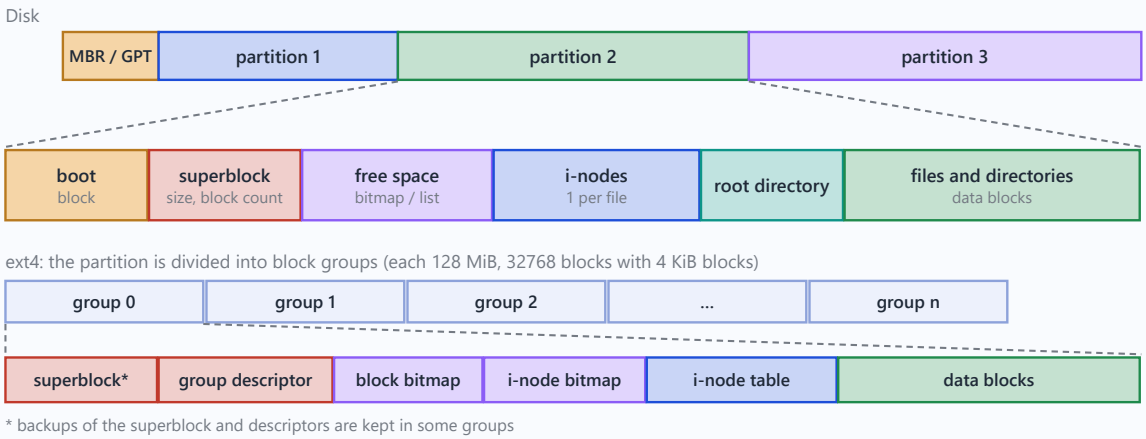


Figure 10.21: The layout of a file system in a partition and the ext4 block groups.

10.5.4 From i-node number to disk address

Because the i-node table lies in consecutive blocks, the location of an i-node is computed from its number. Three pieces of information from the superblock are needed for this: the block where the table starts, the block size, and the i-node size. The computation has three steps:

> offset = n × i-node size, (table block, offset in block) = (offset div block size, offset mod block size), disk block = table start + table block

Example 10.6

The block is 4 KiB and the i-node 128 B. Block 0 is the superblock, blocks 1–2 are the bitmaps, and blocks 3–10 are the i-node table. i-node numbers start at 0; the disk is addressed with 512-byte sectors. Where on disk is i-node 77?

1. **i-nodes per block.** 4096 / 128 = 32. The table holds 256 i-nodes in 8 blocks.
2. **Offset from the start of the table.** 77 × 128 = 9856 = 2 × 4096 + 1664. The table's block 2 (counting from 0), byte 1664 inside the block; 1664 / 128 = 13, that is, the 13th i-node of that block.
3. **Disk block.** 3 + 2 = block 5.
4. **Sector.** The byte address is 5 × 4096 + 1664 = 22,144; 22,144 / 512 = 43.25 → sector 43, and inside it byte 0.25 × 512 = 128. There are 512 / 128 = 4 i-nodes in a sector.

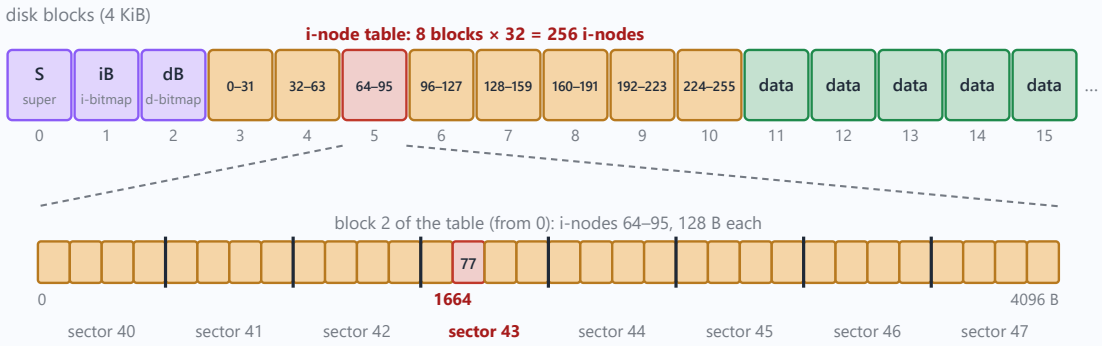


Figure 10.22: The location of i-node 77: block 2 of the table (disk block 5), byte 1664 inside the block, sector 43.

The "where is the i-node table" information in the superblock is indispensable for this computation. Two details are added in ext4. Numbers start at 1 and the table is divided into block groups. If there are g i-nodes per group, group = (n − 1) div g and the index within the group = (n − 1) mod g. With g = 8192 and an i-node size of 256 B, for i-node 20,000: 19,999 div 8192 = 2, 19,999 mod 8192 = 3615. The offset in the group's table is 3615 × 256 = 925,440 = 225 × 4096 + 3840: block 225 of the group's table, offset 3840.

Common mistake

Mistaking the table block for the disk block. 9856 div 4096 = 2 is the table's block 2; the disk block is found by adding the start of the table (3 + 2 = 5). Whether numbers start at 0 or 1 also shifts the computation by one i-node.

10.6 Allocation methods

The central question in implementing a file system is this: how is it recorded where the blocks of a file are on disk? The recording method determines three things: how efficiently free space is used, whether the file can grow, and how quickly a block in the middle of the file can be reached. There are four basic methods: contiguous, linked-list, FAT, and indexed allocation.

10.6.1 The same file, four methods

The file `jeep` in Silberschatz's example is five blocks long. Below we place it on a 32-block disk with the four methods; Figure 10.23 shows the indexed layout.

- **Contiguous:** five blocks side by side, at 9–13. The directory entry has only the start (9) and the length (5). Block i = start + i.

- **Linked list:** the blocks are scattered over the disk: 9, 16, 1, 10, 25. Each block carries the number of the next one; the directory entry knows the first and the last block.
- **FAT:** the same chain, but the pointers are not in the blocks but in a table in memory: FAT[9] = 16, FAT[16] = 1, FAT[1] = 10, FAT[10] = 25, FAT[25] = EOF. The blocks are left entirely to data.
- **Indexed:** all five addresses sit in order in a single index block (19): 9, 16, 1, 10, 25. Block i is accessed directly.

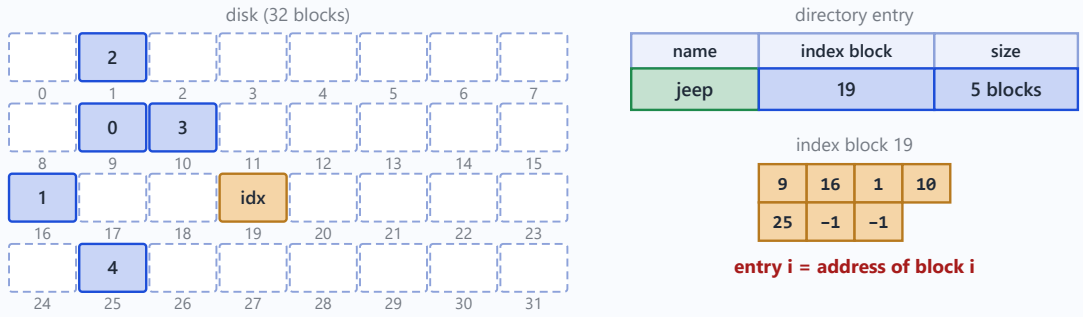


Figure 10.23: Indexed allocation of the file `jeep` : the directory entry points to the index block (19), and the index block points to the five data blocks in order.

10.6.2 Contiguous allocation

> **Definition 10.8**
>
> **Contiguous allocation** means placing each file in consecutive blocks on disk. The directory entry holds the (start, length) pair.

The advantages of the method are clear. It is simple to implement: only two numbers are stored. Random access is easy: the file's block i is start + i. The whole file is read with a single seek and sequential reads; on a hard disk there is no faster layout.

The drawbacks appear as time goes by. Table 10.10 follows Tanenbaum's example. A 40-block disk is full with seven files: A 4, B 3, C 6, D 5, E 12, F 6, G 4 blocks. D and F are deleted. Two holes are left behind: 5 blocks at 13–17 and 6 blocks at 30–35. There are 11 free blocks in total. Now file H of 8 blocks arrives. The total free space suffices, but no hole is 8 blocks: H does not fit.

State	Holes (start, size)	Total free	Largest hole
7 files, disk full	—	0	—
D and F deleted	(13, 5), (30, 6)	11	6
H (8 blocks) arrives	(13, 5), (30, 6)	11	6 < 8 → does not fit

State	Holes (start, size)	Total free	Largest hole
after compaction	H goes to 29–36, (37, 3)	3	3

Table 10.10: External fragmentation in contiguous allocation (40 blocks; recomputed with Python).

This is the same as **external fragmentation** in memory management. One solution is compaction (defragmentation): E and G are shifted left, a single hole of 11 blocks forms, and H fits (Figure 10.24). The cost is high: 12 + 4 = 16 blocks on disk must be read and rewritten. The second problem of contiguous allocation is that the final size of the file must be known in advance. If a file wants to grow and the block behind it is occupied, it either cannot grow or must be moved entirely to another place.

Figure 10.24: Compaction: G and E were shifted, a single hole of 11 blocks formed, and H was placed. The cost is rewriting 16 blocks.

Despite these drawbacks, contiguous allocation is still used today in media whose content is written once and does not change: CD-ROM and DVD file systems (ISO 9660, UDF) are like this. The method also has a modern descendant. The file is divided into several contiguous pieces instead of a single piece. These pieces are called **extents**; they are the basis of ext4 and NTFS.

10.6.3 Linked-list allocation

Definition 10.9

In **linked-list allocation** each block carries the number of the file's next block. The directory entry knows the first block (and, in Silberschatz's layout, the last block).

There is no external fragmentation in this method: any free block can be appended to the end of the file. The file grows easily and its size need not be known in advance. Internal fragmentation exists here as in every method: the unfilled part of the last block is wasted.

The problem is random access. The blocks of file A in Figure 10.25 are in the chain 4 → 7 → 2 → 10 → 12. Suppose we want to read logical block 3 (counting from 0). The directory entry knows only physical 4. Physical 4 is read: the next is 7. 7 is read: the next is 2. 2 is read: the next is 10. Only now is

it clear that physical 10 is logical block 3, and it is read. For a single block, 4 disk reads have been made. Logical block k takes k + 1 reads; random access is O(n). On a hard disk each of these reads may also be a separate seek.

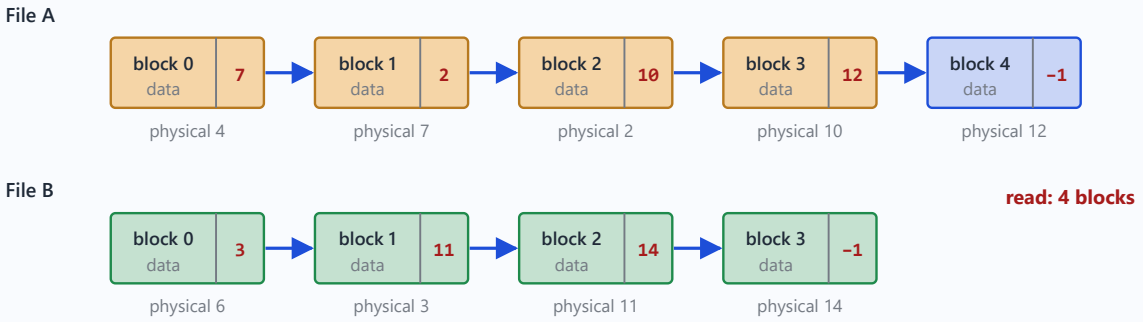


Figure 10.25: Linked-list allocation: to reach logical block 3 of A (physical 10), 4 blocks are read.

There are two more small problems. Because a few bytes of each block go to the pointer, the amount of data in a block is 4096 − 4 = 4092 bytes. This number is not a power of two; for programs working with 4096-byte pages each read spreads over two blocks. The second is reliability: if a single pointer is damaged, the rest of the chain, that is, the end of the file, is lost.

10.6.4 FAT: allocation with a table

The remedy for both problems of the linked list is to take the pointers out of the blocks. The pointers are gathered in a single table at the beginning of the disk: the **file allocation table** (FAT). Entry i of the table is the number of the block that comes after block i. Free blocks are marked 0, and the end of a chain is marked with a special EOF value. The blocks are now entirely data.

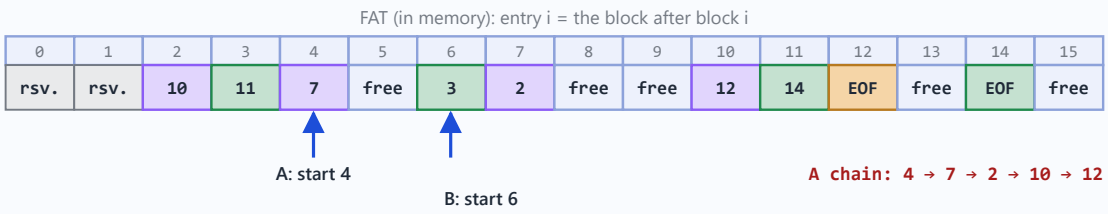


Figure 10.26: In the FAT table, the chain 4 → 7 → 2 → 10 → 12 of file A; file B is 6 → 3 → 11 → 14.

There are two files in the table in Figure 10.26. For A: entry 4 = 7, entry 7 = 2, entry 2 = 10, entry 10 = 12, entry 12 = EOF. For B: 6 → 3 → 11 → 14 → EOF. Random access still requires following the chain, but since the table is in memory, the chain is followed in memory: there is no disk read. Because the table is so important, it is kept on disk in two copies.

The real problem of FAT is its size. The table has an entry for every block on the disk; the table grows in proportion to the disk size and must reside entirely in memory to be fast.

Example 10.7

A 200 GiB disk is divided into 1 KiB blocks; each FAT entry is 4 bytes. How much memory does the table need?

1. Number of blocks: $200 \times 2^{30} / 2^{10} = 200 \times 2^{20} = 209{,}715{,}200$.
2. Table: 209,715,200 × 4 B = 838,860,800 B = 800 MiB. Even if the entry is squeezed to 3 bytes, 600 MiB.

Real FAT32 uses **clusters** made of several blocks instead of blocks. A 2 TiB volume with 32 KiB clusters has $2^{41} / 2^{15} = 67{,}108{,}864$ clusters; with 4-byte entries the table is 256 MiB.

10.6.5 Code trace: a FAT chain

The function below finds the disk block that contains byte number `off` in a file whose first block is `first`. The block size is `BLOCK` = 1024 bytes; `fat` is the table in memory and `END` marks the end of the chain.

```
int find_block(int first, long off) {
  int b = first;
  long n = off / BLOCK;  /* block index */
  while (n > 0 && b != END) {
    b = fat[b];          /* follow chain */
    n--;
  }
  return b;          /* END: past EOF */
}
```

`n = off / BLOCK` is an integer division and gives which block of the file (counting from 0) the wanted byte is in. The loop advances n steps along the chain; at each step `b` moves to the next block. The loop's second condition, `b != END`, is a guard against the chain ending early. If the requested block is not in the file, the function returns `END`. The running time is O(n), but each step is only a memory access.

Let us trace two calls (Table 10.11). In `find_block(4, 3500)`, file A starts at block 4. 3500 / 1024 = 3: the wanted byte is in the file's block 3, and one must advance 3 steps along the chain. fat[4] = 7 (n = 2), fat[7] = 2 (n = 1), fat[2] = 10 (n = 0). The loop ends; the disk block is 10. Byte 3500 is byte 3500 − 3 · 1024 = 428 of this block.

Call	n at start	Chain followed (b, n)	Returns
`find_block(4, 3500)`	3	(4, 3) → (7, 2) → (2, 1) → (10, 0)	10; byte 428 in the block
`find_block(6, 5000)`	4	(6, 4) → (3, 3) → (11, 2) → (14, 1) → (END, 0)	END: past the end of the file

Table 10.11: Trace of `find_block` (run by translating it one to one into Python).

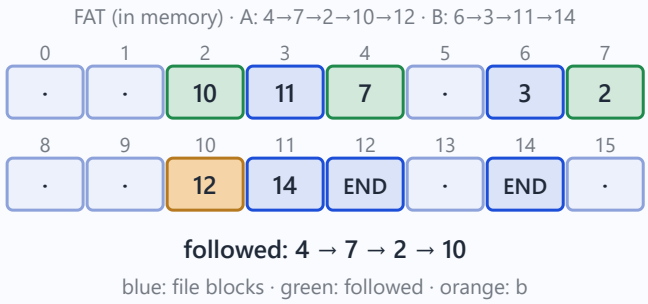


Figure 10.27: The last step of `find_block(4, 3500)` : the chain followed is 4 → 7 → 2 → 10.

`find_block(6, 5000)` is more instructive. File B is 4 blocks, so it holds at most 4096 bytes. 5000 / 1024 = 4: the file's block 4 (the fifth block) is requested. The chain goes 6 → 3 → 11 → 14; at the fourth step fat[14] = END arrives and n = 0. The file has no block 4; the offset lies outside the file. A `read` from this offset returns 0 bytes (end of file). If a `write` is made at the same offset, the file system allocates a new block and links it to the entry of 14.

Example 10.8

The cluster size is 4 KiB; in the FAT, FAT[5] = 9, FAT[9] = 2, FAT[2] = 11, FAT[11] = EOF. The first cluster of `slides.pptx` is 5. In which clusters does the file lie, and in which cluster is byte 9000?

1. 9000 / 4096 = 2 (integer division): the byte is in the file's cluster 2 (counting from 0).
2. The chain: 5 (cluster 0) → 9 (1) → 2 (2) → 11 (3) → EOF. The file is in 4 clusters; it holds at most 16 KiB and is scattered on disk.
3. Cluster 2 of the file is physical cluster 2. The position inside the cluster is 9000 − 2 × 4096 = byte 808.

10.6.6 The FAT family

The name FAT describes a family running from MS-DOS to today. The family members are named by the number of bits in the FAT entry (Table 10.12). The wider the entry, the more clusters can be addressed.

	FAT12	FAT16	FAT32	exFAT
FAT entry	12 bits	16 bits	28 bits used	32 bits
Largest volume	≈ 16 MiB	2 GiB (32 KiB cluster)	2 TiB (512 B sector)	128 PiB
Largest file	as large as the volume	2 GiB	4 GiB − 1 byte	16 EiB (theoretical)
Free space	0 in the FAT	0 in the FAT	0 in the FAT + FSInfo hint	separate bitmap

	FAT12	FAT16	FAT32	exFAT
Use	floppy disk	old disks	USB, UEFI ESP	SDXC/SDUC, large USB

Table 10.12: FAT12, FAT16, FAT32, and exFAT.

In FAT the directory entry carries everything: name, extension, attributes, time, date, first cluster, and size (Figure 10.28). Since the size field is 32 bits, a file in FAT32 can be at most 4 GiB − 1 bytes. A 4 GiB movie file cannot be copied to a FAT32-formatted USB stick. exFAT removes this limit. It also keeps free space in a separate bitmap and, for files known to be contiguous, may not use the FAT chain at all. Windows's graphical formatting tool has historically limited FAT32 to 32 GB; this limit belongs to the tool, not to the file system.

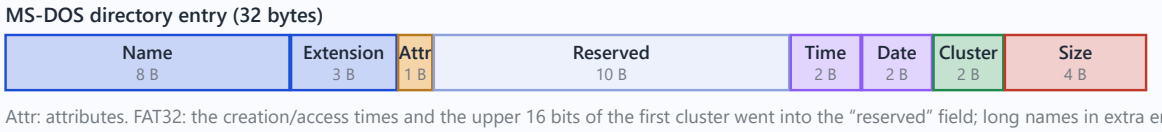


Figure 10.28: The 32-byte directory entry of MS-DOS. In FAT32 the creation and access times and the upper 16 bits of the first cluster were placed in the “reserved” field.

10.6.7 Indexed allocation and the i-node

Definition 10.10

In **indexed allocation** all the block addresses of a file are gathered in an index block; entry i is the address of block i. In UNIX this structure is called an **i-node** (index node). The i-node holds the file's block addresses and attributes together.

Indexed allocation keeps the good sides of the linked list: there is no external fragmentation, and the file grows wherever there are free blocks. On top of that it speeds up random access: the address of block i is in entry i of the index block, and there is no chain to follow.

A big advantage of the i-node over FAT lies in memory use. An i-node is in memory only while the file is open. If K files are open, the memory needed is K × the i-node size. Unlike FAT, it is proportional not to the disk size but to the number of open files. With 1000 open files and 256-byte i-nodes, 256,000 bytes (about 250 KiB) suffice, whether the disk is 1 TB or 100 TB.

The price is twofold. Even for a small file, space is spent on the index. Because the i-node has a fixed size, the addresses of a large file do not all fit in it. The solution is to use the last few entries of the i-node to point not to data blocks but to other index blocks: **indirect blocks**.

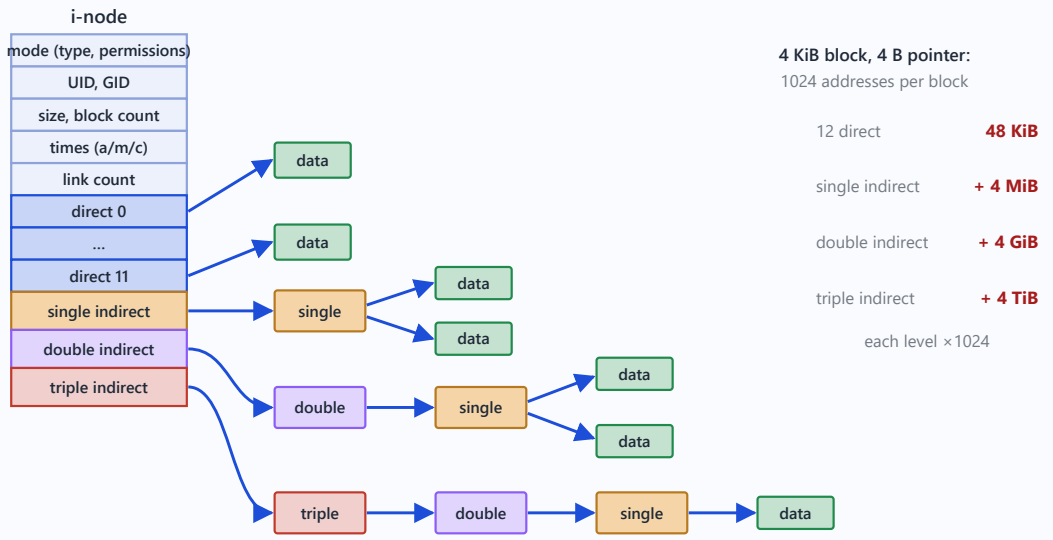


Figure 10.29: The UNIX i-node: attributes, 12 direct block addresses, and single, double, and triple indirect blocks. With 4 KiB blocks each level multiplies the capacity by 1024.

Figure 10.29 shows the structure of ext2/ext3 and classical UNIX. The i-node has 12 **direct** pointers; the addresses of the first 12 blocks of the file are directly here. The 13th pointer is **single indirect**: it points not to data but to an index block full of addresses. The 14th pointer is **double indirect**: it points to an index block that points to index blocks. The 15th pointer is **triple indirect**: one more level. Small files use only the direct pointers and no indirect block is ever allocated. As the file grows, the tree gets deeper. In V7 UNIX the number of direct pointers was 10.

10.6.8 Maximum file size

With this structure the largest size a file can have can be computed. Assumptions: the block is 4 KiB = 4096 B and a block address is 4 B. An index block holds 4096 / 4 = 1024 addresses; let P = 1024.

Example 10.9

The i-node has 12 direct, 1 single, 1 double, and 1 triple indirect pointer. How many bytes is the largest file?

1. **Direct:** 12 blocks × 4 KiB = 48 KiB (49,152 B).
2. **Single indirect:** one index block points to 1024 data blocks: 1024 × 4 KiB = 4 MiB.
3. **Double indirect:** each of 1024 index blocks points to 1024 data blocks: 1024^2 = 1,048,576 blocks = 4 GiB.
4. **Triple indirect:** one more level: 1024^3 = 1,073,741,824 blocks = 4 TiB.
5. **Total:** 12 + 1024 + 1024^2 + 1024^3 = 1,074,791,436 blocks; × 4096 = 4,402,345,721,856 B ≈ 4.004 TiB.

Almost all of the size comes from the triple indirect block; the contribution of the direct pointers is negligibly small.

The general formula is: with D direct pointers, block size B, and address size a, P = B / a and the maximum size is $(D + P + P^2 + P^3) \times B$. In Tanenbaum's example the block is 1 KiB, the address 4 B (P = 256), and there are 10 direct pointers: 10 + 256 + 65,536 + 16,777,216 = 16,843,018 blocks, about 16 GiB.

This calculation is only the limit of the pointer structure. In real systems other limits come into play earlier. In ext2/ext3 the block counter in the i-node is 32 bits and counts 512-byte units. With 4 KiB blocks the real limit is $2^{32} \times 512$ B = 2 TiB. ext4 raises this limit to 16 TiB with extents.

Common mistake

Taking the double indirect capacity to be 2 × 1024 blocks. The double indirect block points to 1024 index blocks and each of them to 1024 data blocks: the product is 1024^2. Each level of indirection multiplies the capacity by P; it does not add P.

10.6.9 From file offset to disk block

In a `read` call the file system converts a byte offset to a disk block. First the logical block number is found: lb = offset div 4096. Then one looks at which region this number falls into. For the same structure (12 direct, P = 1024) the regions are:

- lb < 12: direct pointer lb. If the i-node is in memory, 1 read (the data).
- 12 ≤ lb < 12 + 1024: entry (lb − 12) of the single indirect block. 2 reads.
- the next 1024^2 blocks: double indirect. r = lb − 1036; outer index r div 1024, inner index r mod 1024. 3 reads.
- the next 1024^3 blocks: triple indirect. r = lb − 1,049,612; indices r div 1024^2, (r div 1024) mod 1024, and r mod 1024. 4 reads.

Offset (B)	Logical block	Level and indices	Offset in block	Reads
40.000	9	direct 9	3136	1
1.000.000	244	single indirect [232]	576	2
5.000.000	1220	double indirect [0][184]	2880	3
6.000.000.000	1.464.843	triple indirect [0][405][511]	3072	4

Table 10.13: Translating four offsets (4 KiB block, 12 direct; verified with Python).

The rows of the table can be followed one by one. 40,000 / 4096 = 9 and 9 < 12: direct pointer 9 gives the address of the block. 1,000,000 / 4096 = 244; since 244 − 12 = 232 < 1024, entry 232 of the single indirect block is consulted. 5,000,000 / 4096 = 1220; 1220 − 12 − 1024 = 184 falls in the double

indirect region: outer index 184 div 1024 = 0, inner index 184. For 6 GB, 6,000,000,000 / 4096 = 1,464,843. Subtracting 12 + 1024 + 1024^2 = 1,049,612 leaves 415,231. The three indices are 415,231 div 1024^2 = 0, (415,231 div 1024) mod 1024 = 405, and 415,231 mod 1024 = 511.

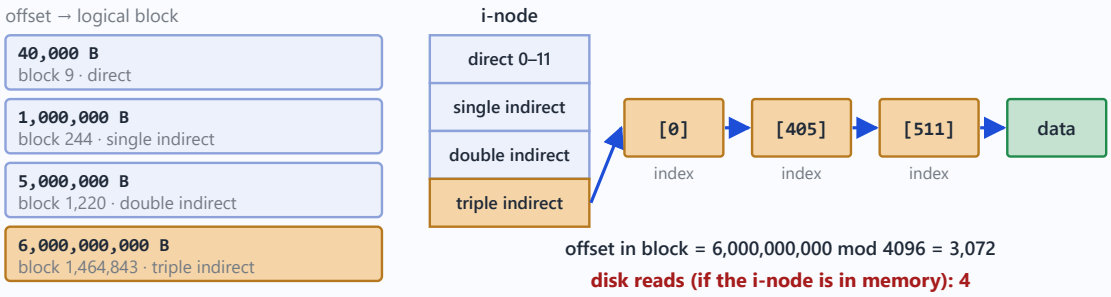


Figure 10.30: The path of byte 6,000,000,000: i-node → triple indirect block [0] → double indirect block [405] → single indirect block [511] → data. 4 reads if the i-node is in memory.

The read counts are worst cases. A program reading sequentially refers to the same indirect blocks again and again; the cache keeps the first levels in memory and most of the extra reads disappear.

Example 10.10

Logical block 699 of a file of 1000 blocks is to be read. The directory entry and the i-node are in memory; nothing else is in the cache. The block is 4 KiB, the address 4 B, and the i-node has 12 direct pointers. How many disk reads are needed with the four methods?

1. **Contiguous: 1.** Address = start + 699.
2. **Linked list: 700.** The address of block 699 cannot be learned without reading blocks 0 … 698; 699 is read too.
3. **FAT: 1.** 699 steps are followed in memory along the chain, then a single read is made.
4. **i-node: 2.** 699 − 12 = 687 < 1024: the single indirect block and the data block.

10.6.10 Extents and NTFS

The classical i-node keeps a separate address for each block. For a large and mostly contiguous file this is very wasteful: to describe 32,768 consecutive blocks, 32,768 addresses are written. ext4 uses **extents** instead.

Definition 10.11

An **extent** is a triple (logical start, length, physical start) that describes a contiguous sequence of blocks with a single record.

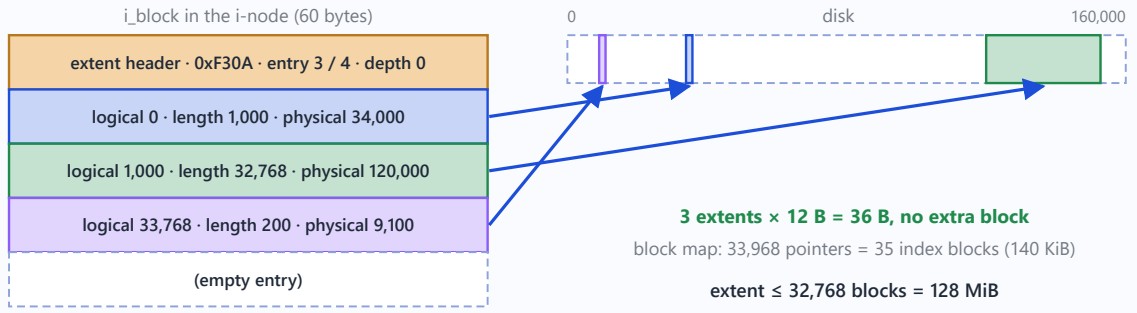


Figure 10.31: The extent header and three extents in an ext4 i-node; each extent describes a contiguous region on disk.

The file in Figure 10.31 consists of three extents. 1000 blocks starting at logical 0 are at physical 34,000, 32,768 blocks starting at logical 1000 are at physical 120,000, and 200 blocks starting at logical 33,768 are at physical 9100. The file is 33,968 blocks in total. Since each extent takes 12 bytes, three extents fit in 36 bytes; there is room in the i-node's 60-byte `i_block` field and no extra block is needed. The same file with the classical block map needs 33,968 pointers. These spread over 12 direct pointers, 1 single indirect block, 1 double indirect block, and the 33 index blocks it points to: 35 index blocks, that is, 140 KiB.

In ext4 the physical address is 48 bits: with 4 KiB blocks a volume can grow up to 1 EiB. A file can be at most 16 TiB. One extent describes at most 32,768 blocks (128 MiB). Four extents fit in the i-node; if more are needed, the entries point to the index nodes of an **extent tree**. Extents pay off only if the file is actually written contiguously. ext4 uses **delayed allocation** for this: data waits in memory first, and blocks are chosen only when it is written to disk. Since by then it is known how large the file has grown, the multiblock allocator can find large contiguous areas. XFS, Btrfs, APFS, and NTFS also use extents.

At the center of NTFS lies the **master file table** (MFT) (Figure 10.32). In NTFS everything is a file; the MFT itself is record 0 of the MFT. The first records are reserved for the system's own metadata files: `$MFTMirr` (backup), `$LogFile` (journal), `$Volume`, the root directory (record 5), `$Bitmap` (the free-cluster bitmap), and so on. User files start from record 24.

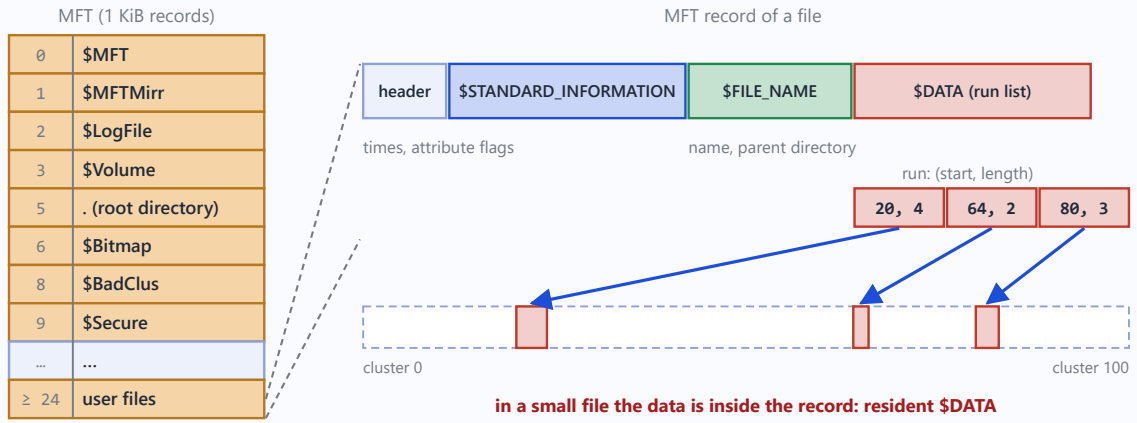


Figure 10.32: The NTFS master file table: the metadata files, and the attributes and data runs of a file record.

Each file is at least one 1 KiB MFT record, and the content of the record is a list of attributes. `$STANDARD_INFORMATION` holds the times and flags, and `$FILE_NAME` the name and the parent directory. The data itself is also an attribute: `$DATA`. The data of a small file fits inside the record; this is called **resident** data, and no other block is read to read the file. Large data, by contrast, is kept as a run list. In the example, (20, 4), (64, 2), (80, 3) are three extents. Directories are B+ trees (`$INDEX_ROOT`, `$INDEX_ALLOCATION`). NTFS also supports hard links, reparse points, alternate data streams (ADS), compression, EFS encryption, and quotas. On Windows servers there is, in addition, ReFS, which uses copy-on-write and keeps checksums.

10.6.11 Comparing file systems

Table 10.14 compares today's common file systems by their limits and crash-consistency methods.

File system	Largest file	Largest volume	Crash consistency	Typical use
FAT32	4 GiB − 1 B	2 TiB	none (chkdsk/fsck)	USB, UEFI ESP
exFAT	16 EiB*	128 PiB	none	SD card, external disk
NTFS	16 EiB* (Windows: 256 TiB–8 PiB)	256 TiB–8 PiB	metadata journal	Windows
ext4	16 TiB (4 KiB block)	1 EiB	journal (jbd2)	Linux, Android
XFS	8 EiB	8 EiB	metadata journal	RHEL, large servers
Btrfs	16 EiB	16 EiB	copy-on-write + checksums	Fedora, openSUSE, NAS
ZFS	16 EiB	2^{128} B (theoretical)	copy-on-write + checksums	FreeBSD, TrueNAS, servers
APFS	8 EiB	8 EiB	copy-on-write, snapshots	macOS, iOS

Table 10.14: Comparison of file systems (* theoretical limit; the operating system and the cluster size impose a lower limit).

Since version 1709 of Windows 10, NTFS reaches volumes of 8 PiB with 2 MiB clusters. Apple's old HFS+ gave way to APFS in 2017, and on Linux ext3 gave way to ext4.

10.7 Implementing directories

10.7.1 Directory entries and long names

When a file is opened, the directory entry must supply enough information to find the file's data. This information varies with the allocation method. With contiguous allocation the address of the first block suffices, with linked lists and FAT the number of the first block, and in UNIX the i-node number. Where the attributes are kept is solved in two ways (Figure 10.33). (a) In MS-DOS the attributes and the disk address are directly inside the directory entry. (b) In UNIX the directory entry is only a name and an i-node number; the attributes are in the i-node. The second layout makes hard links possible: two entries pointing to the same i-node share the same attributes.

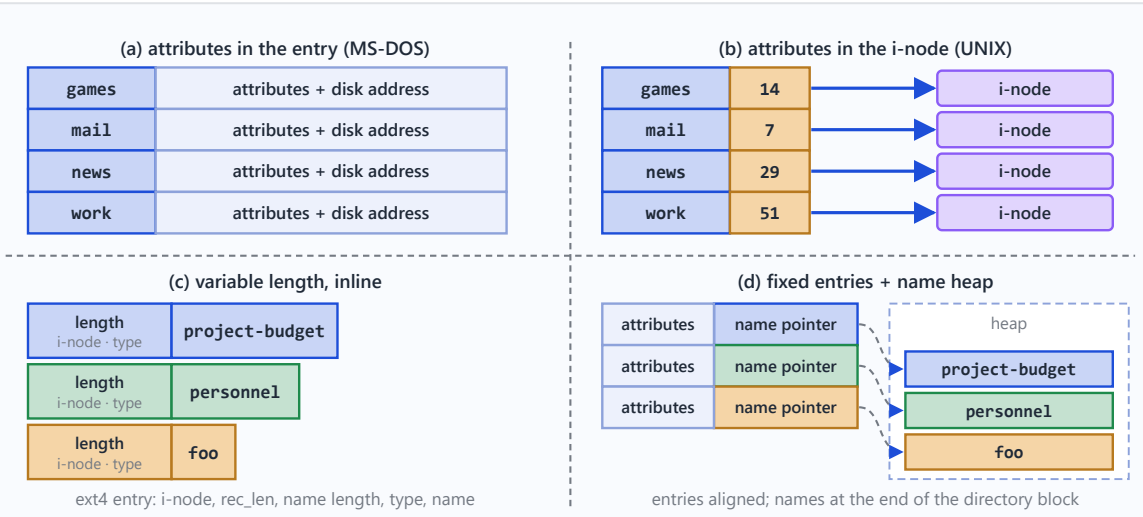


Figure 10.33: Directory entry formats: (a) attributes in the entry, (b) attributes in the i-node, (c) variable-length entries, (d) fixed entries and a name heap.

Long names are a separate problem. Reserving a fixed 255-character name field for every entry wastes a lot of space, because most names are short. There are two solutions. (c) In **variable-length entries** each entry carries its own length, and the name sits inside the entry. ext4's entry is like this: the i-node, the entry length (`rec_len`), the name length, the type, and the name. When a file is deleted, a hole is left in the place of the entry. A new entry may not fit into this hole. If an entry falls on a page boundary, another page fault may occur while reading. (d) In the second solution the entries are fixed-length and aligned. Each entry carries a name pointer instead of the name itself; the names sit in a **heap** at the end of the directory block. Scanning the entries becomes easier, but the heap must be managed separately.

10.7.2 Searching a directory

At each step of path resolution a name is searched for in a directory. The speed of the search depends on how the directory is kept (Figure 10.34).

The **linear list** is the simplest structure: entries sit one after another and the search scans from beginning to end. In the list in Figure 10.34, finding `report.pdf` takes 7 comparisons, against `.`, `..`, `notes.txt`, `photo.png`, `homework`, `makefile`, and `report.pdf`. The search is O(n). Before a new name is added, one must check that the same name does not exist, which means a full scan. On deletion the entry is marked empty, or the last entry is moved into the place of the deleted one.

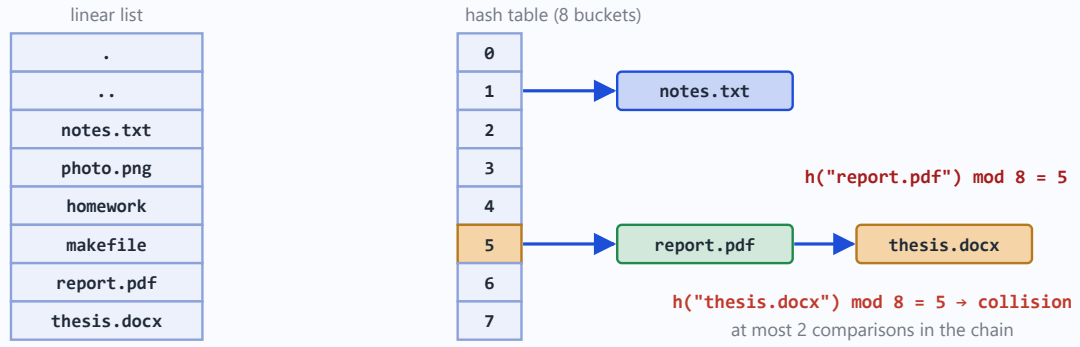


Figure 10.34: A linear list and a hash table with 8 buckets. `report.pdf` and `thesis.docx` fall into the same bucket (5); the collision is resolved with a chain.

A **hash table** chooses a bucket from the digest of the name: h("report.pdf") mod 8 = 5. The search is O(1) on average. Two names can fall into the same bucket; in the figure `thesis.docx` also falls into 5. This **collision** is resolved with a chain hanging off the bucket, and at most 2 comparisons are made in the chain. A fixed number of buckets becomes a problem in a growing directory: in a directory of 100,000 entries each chain of 8 buckets has an average length of 12,500 entries and the search turns linear again. Hashing also does not preserve order; `ls` has to sort the entries separately for alphabetical output.

Modern systems therefore use trees. For large directories ext4 builds an **htree** ordered by the name digest. NTFS, XFS, and Btrfs use a **B+ tree**. In a B+ tree the search is O(log n) and the entries can be listed in order; two advantages are obtained at once. Linux also keeps the most recently used name → i-node mappings in the in-memory dentry cache; frequently opened paths are resolved without ever going down to the disk.

10.8 Free-space management

10.8.1 Choosing the block size

The disk is divided into fixed-size blocks; choosing the block size is a trade-off between two goals. A large block brings a lot of data in one seek and raises the data rate. With small files, however, most of the block stays empty and space is wasted. A small block does the opposite.

Tanenbaum's model shows this trade-off in numbers (Figure 10.35). The disk's average seek time is 5 ms and its rotation time 8.33 ms; a track holds 1 MB. The time to read a block is the sum of the seek, the average wait of half a rotation, and the transfer time:

> time = 5 + 4.165 + (b / 1 MB) × 8.33 ms

Let all files be 4 KiB. With 4 KiB blocks a read takes 9.198 ms: the data rate is 4096 B / 9.198 ms ≈ 0.45 MB/s. With 128 KiB blocks a read takes 10.206 ms but brings 131,072 bytes: ≈ 12.8 MB/s. The data rate has risen by a factor of about 29. On the other hand, a 4 KiB file fills only 4/128 = 3.1% of a 128 KiB block. For blocks ≤ 4 KiB the space efficiency is 100%; it falls to 50% at 8 KiB and to 3.1% at 128 KiB.

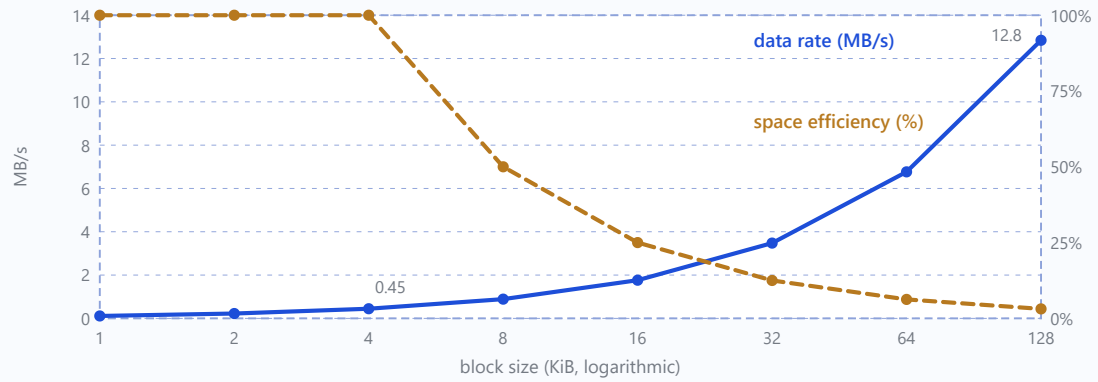


Figure 10.35: As the block size grows, the data rate rises and the space efficiency for 4 KiB files falls (Tanenbaum's model; computed with Python).

Block	Read time (ms)	Data rate (MB/s)	Space efficiency
1 KiB	9,173	0,11	%100
4 KiB	9,198	0,45	%100
16 KiB	9,295	1,76	%25
64 KiB	9,686	6,77	%6,25

Block	Read time (ms)	Data rate (MB/s)	Space efficiency
128 KiB	10,206	12,84	%3,1

Table 10.15: The block-size model (seek 5 ms, rotation 8.33 ms, track 1 MB = 2^{20} B, files of 4 KiB).

Measurements of real file sizes paint an interesting picture. Most files are small, but most of the disk space is taken up by a small number of large files. Today the default block size is 4 KiB, the same as the page size. Large files are supported with extents and cluster options: the block stays small, but the blocks of a large file are allocated contiguously.

10.8.2 Four methods

The file system must know which blocks are free. In Silberschatz's example, 15 blocks of a 32-block disk are free: 2–5, 8–13, 17–18, and 25–27. The same set of free blocks can be kept in four ways.

Definition 10.12

A **bitmap** is an array that keeps one bit per block. In these notes and in the example in the slides 1 = free and 0 = in use; some systems use the opposite.

In the **bitmap**, 32 blocks take 32 bits, that is, 4 bytes (Figure 10.36). For example, the bitmap is `00111100 11111100 01100000 01110000`. The size of the bitmap is fixed and small. Finding contiguous free blocks is easy: look for consecutive 1s. Because processors can find the first 1 bit in a word with a single instruction, finding the first free block is fast: the first nonzero word is found, and the first 1 bit in it is taken. For the bitmap to be fast, it must be kept in memory.

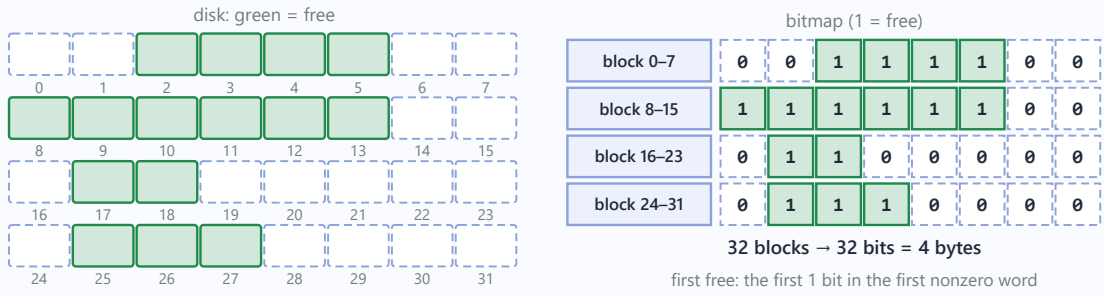


Figure 10.36: The bitmap of a 32-block disk (1 = free): 15 free blocks, 32 bits = 4 bytes.

In the **linked list** the free blocks are chained together. A head pointer points to the first free block (2); each free block carries the number of the next: 2 → 3 → 4 → 5 → 8 → … → 13 → 17 → 18 → 25 → 26 → 27 → −1. No extra space is needed, because the information lies inside the free blocks. Taking a single free block is cheap: the head of the list is given. But finding many or contiguous free blocks is expensive; walking the list means an I/O for each block.

Grouping improves the linked list. The first free block holds the addresses of the next n − 1 free blocks and the block of the next group. With n = 4 the groups in the example are: block 2 → {3, 4, 5}, next group 8; block 8 → {9, 10, 11}, next 12; block 12 → {13, 17, 18}, next 25; block 25 → {26, 27}, end. With a single read, n − 1 free blocks are found. The free-list blocks of classical UNIX worked this way.

The **counting** method exploits the fact that free blocks are often found in contiguous runs. Each run is kept as a (first block, number of consecutive blocks) pair (Figure 10.37). The 15 free blocks in the example are just 4 pairs: (2, 4), (8, 6), (17, 2), (25, 3). This is the extent idea applied to free space. XFS keeps the counting pairs in two B+ trees, one ordered by start and one by length; thus both the question “free space near this block” and the question “a free run of at least this size” are answered quickly.

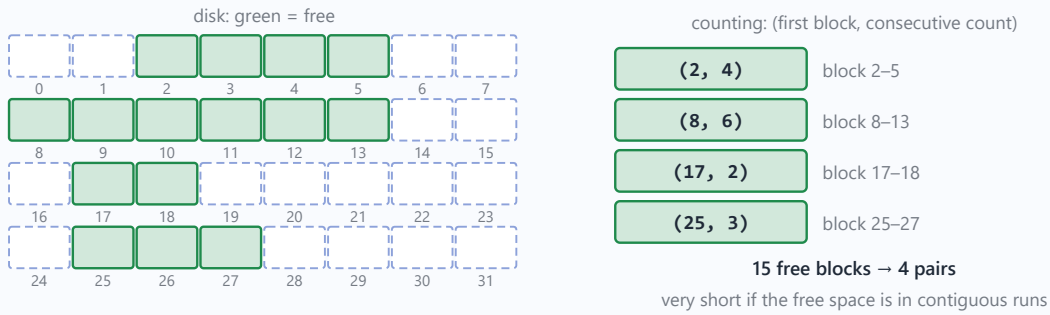


Figure 10.37: Counting: 15 free blocks are described by 4 (start, length) pairs.

Example 10.11

On a 32-block disk the free blocks are 3–6, 10–11, 20–24, and 30. How are they represented with the counting method and with a bitmap (1 = free, block 0 at the far left)?

1. **Counting:** the start and the length of each run: (3, 4), (10, 2), (20, 5), (30, 1).
2. **Bitmap:** 00011110 for blocks 0–7, 00110000 for 8–15, 00001111 for 16–23, 10000010 for 24–31.

The bitmap is 32 bits however full the disk is. The length of the counting list, however, depends on the number of runs; it is very short when the disk is full or when the free space is in large runs.

10.8.3 Code trace: allocating a block with a bitmap

The function below looks for a free block in the bitmap of a disk of N = 16 blocks. In this code the meaning of the bitmap is reversed: `bitmap[i] == 1` means the block is in use and 0 means free. The search starts from block number `hint` and wraps around to the beginning when it reaches the end of the array.

```
int alloc_block(int hint) {
  for (int k = 0; k < N; k++) {
    int i = (hint + k) % N;
    if (bitmap[i] == 0) {  /* free? */
      bitmap[i] = 1;
      return i;
    }
  }
  return -1;               /* disk full */
}
```

`i = (hint + k) % N` is the position k steps ahead of `hint`; the mod operation makes the search return to 0 after 15. When the first free block is found, its bit is set to 1 and its number is returned. If all the blocks are in use, the loop ends after N turns and −1 is returned. In the worst case the time is O(N). The purpose of the `hint` parameter is locality: if the file's new block is searched for right after its last block, the blocks fall close to one another and head movement is reduced in sequential reading.

Initially the bitmap is 1 1 0 1 1 1 0 0 1 1 1 0 1 0 1 1 (from block 0 to 15). The free blocks are 2, 6, 7, 11, and 13. A file ends at block 3 and wants 3 more blocks. The hint of each call is one more than the file's current last block. Table 10.16 gives the trace.

Call	Positions examined (i)	Returns	Blocks of the file
`alloc_block(4)`	4 (in use), 5 (in use), 6 (free)	6	..., 3, 6
`alloc_block(7)`	7 (free)	7	..., 3, 6, 7
`alloc_block(8)`	8, 9, 10 (in use), 11 (free)	11	..., 3, 6, 7, 11

Table 10.16: Trace of `alloc_block` (run by translating it one to one into Python).

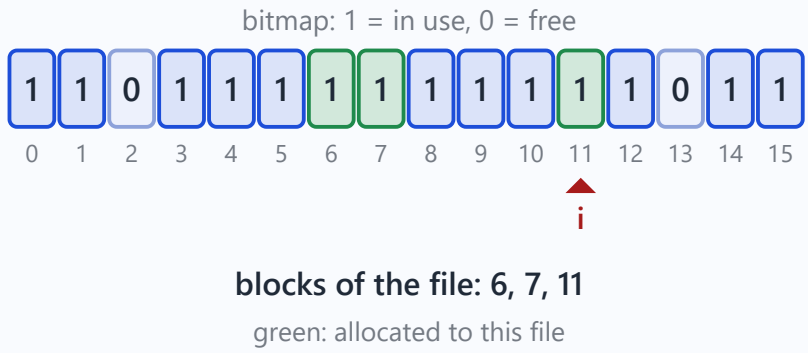


Figure 10.38: The bitmap after three calls: blocks 6, 7, and 11 were allocated to the file.

The file took blocks 6, 7, and 11. The blocks are not contiguous but are close to one another. If the search had no hint and always started from 0, the first call would have given block 2; the file would have gone back to the beginning of the disk. If the hint is near the end, wrap-around comes into play:

with the same initial bitmap, `alloc_block(14)` finds 14 and 15 in use, also passes 0 and 1, and returns block 2.

10.8.4 Numerical comparison

How much space the bitmap and the free-block list take can be computed on a large disk (Table 10.17). In the list each block number is 32 bits. A 1 KiB list block holds 256 numbers: 255 free-block addresses and 1 link to the next list block.

	Tanenbaum's example	Modern example
Disk / block	500 GB, 1 KiB block	1 TiB, 4 KiB block
Number of blocks	$500 \times 10^9 / 1024 \approx 488.3$ million	$2^{40} / 2^{12} = 268.435.456$
List block	1 KiB / 4 B = 256 → 255 addresses + 1 link	4096 / 4 = 1024 → 1023 addresses + 1 link
List, with the disk empty	488.3 M / 255 ≈ 1.91 million blocks	268.4 M / 1023 ≈ 262,401 blocks (≈ 1 GiB)
Bitmap	488.3 M bits / 8192 ≈ 59,605 blocks	2^{28} bits = 32 MiB = 8192 blocks

Table 10.17: The size of the free-space record (with the disk completely empty; computed with Python).

In Tanenbaum's example the number of blocks is $500 \times 10^9 / 1024 = 488{,}281{,}250$. With the disk completely empty the list takes 488,281,250 / 255 ≈ 1,914,828 blocks. The bitmap means one bit per block: a 1 KiB block carries 8192 bits, and 488,281,250 / 8192 ≈ 59,605 blocks suffice. The bitmap is about 32 times smaller than the list. The list is small only when the disk is nearly full, because it lists only the free blocks. The size of the bitmap is independent of how full the disk is.

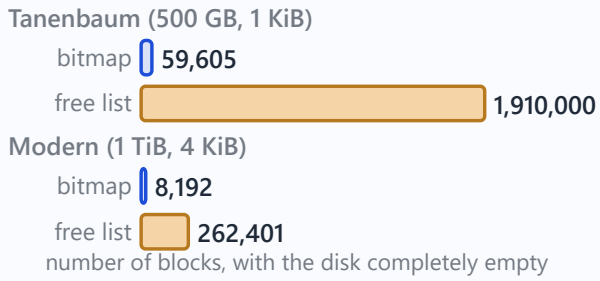


Figure 10.39: The number of blocks of the bitmap and of the free list when the disk is empty.

The list has an advantage too: it is enough to keep a single list block in memory. When the block fills, it is written to disk; when it empties, a new one is read. This simple policy has a trap. If, while the block in memory is exactly full, a temporary file of three blocks is deleted, the block overflows and is written to disk. If the same file is created again right afterwards, the block empties and a new one is read from disk. Temporary files thus cause constant disk I/O. The solution Tanenbaum proposes is to keep the pointer block in memory half full; then both allocation and release proceed for a while without disk I/O.

Real systems use combinations of these methods. ext4 keeps a separate bitmap for each block group. XFS implements the counting method with two B+ trees. ZFS uses a **space map** that accumulates allocation and release records like a journal. Btrfs tracks free and used space in the extent tree.

10.8.5 Disk quotas

On a multiuser system a single user must not fill the whole disk. A **quota** limits the number of blocks and files each user may use (Figure 10.40). When a file is opened, its entry in the open-file table points to its owner's quota record. At every new block or i-node allocation the user's counters are incremented and compared with the limits.

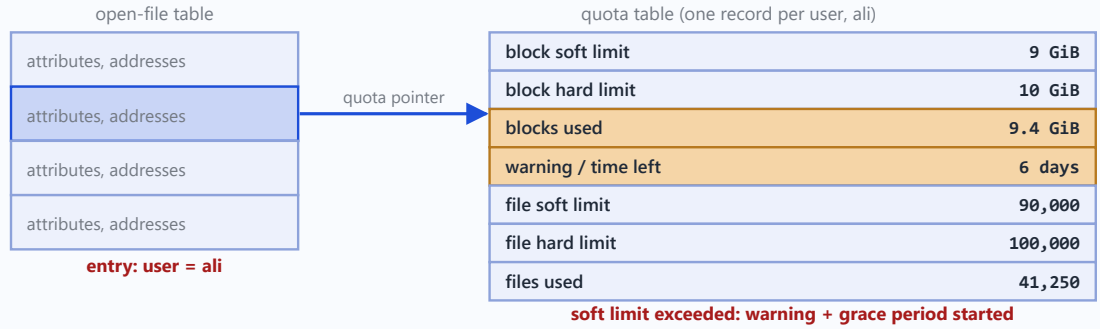


Figure 10.40: The open-file entry points to the user's quota record. ali's usage of 9.4 GiB has exceeded the soft limit of 9 GiB; a warning was issued and the grace period has begun.

Each limit has two levels. The **soft limit** can be exceeded; the user is warned and a grace period (for example 7 days) begins. Within this period the user must drop below the limit by deleting files. The **hard limit** can never be exceeded. If the period ends or the hard limit is reached, writing is rejected with the `EDQUOT` error. In the figure, ali has used 9.4 GiB and exceeded the soft limit of 9 GiB; 6 days remain, and ali can write up to the hard limit of 10 GiB. In the number of files, with 41,250 files, ali is far below the soft limit of 90,000. Today Linux has quota tools (project quotas on ext4 and XFS), NTFS has volume quotas, ZFS has dataset quotas, and Btrfs has subvolume (qgroup) quotas.

10.9 Performance

10.9.1 How many I/Os does it take to create a file?

To see how much work file-system calls put on the disk, let us consider a simple file system without a cache. Let it use i-nodes and a bitmap, and let each directory fit in a single block. Let the location of the root directory's i-node be known from the superblock. Counting of this kind in the OSTEP book by Arpaci-Dusseau and Arpaci-Dusseau reveals the hidden cost behind the calls. Table 10.18 lists the I/Os of four operations.

Call	I/O sequence	Total
`open("/course/note.txt")`, file exists	read root i-node, root data, course i-node, course data, note i-node	5 = 5 reads
`read(fd, buf, 4096)`	read note i-node, note data[0]; write note i-node (access time)	3 = 2 reads + 1 write
`creat("/course/note.txt")`, file does not exist	read root i-node, root data, course i-node, course data, i-bitmap; write i-bitmap, course data; read note i-node; write note i-node, course i-node	10 = 6 reads + 4 writes
`write(fd, buf, 4096)`, new block	read note i-node, d-bitmap; write d-bitmap, note data, note i-node	5 = 2 reads + 3 writes

Table 10.18: Disk I/Os of calls in a file system without a cache.

Opening is proportional to the path length: an i-node and a data block for each directory, and at the end the file's i-node. The read finds the block address in the i-node, reads the block, and updates the access time in the i-node. The last write is often eliminated by the `noatime` or `relatime` mount options.

Creating is much more expensive. First the path is resolved (4 reads). Then a free i-node is chosen from the i-node bitmap and marked (1 read, 1 write). A new entry is added to the directory (1 write). The block that contains the new i-node is read, the i-node is initialized, and it is written (1 read, 1 write). Finally the size and time are updated in the directory's i-node (1 write). The total is 10 I/Os. Each `write` that needs a new block also costs 5 I/Os: the i-node is read, a block is chosen and marked in the data bitmap, the data is written, and the new address and size are written to the i-node. Only one of these 5 I/Os is the actual data.

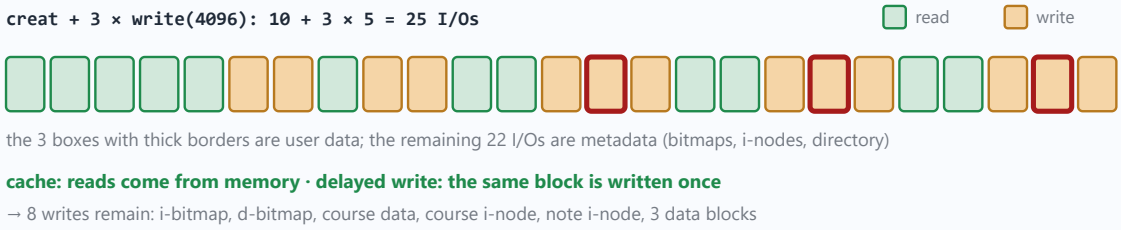


Figure 10.41: `creat` and three 4 KiB `write` s: 10 + 3 × 5 = 25 I/Os. The three boxes with thick borders are user data; the remaining 22 I/Os are metadata.

Creating a file without a cache and writing 12 KiB thus costs 10 + 3 × 5 = 25 I/Os (Figure 10.41). Only three of them are the user's data. The cache reduces this number dramatically. Reads are served from memory. Delayed writing also merges writes made to the same block: the i-node bitmap, the data bitmap, the directory data, the directory's i-node, the file's i-node, and the three data blocks are each written once. Of the 25 I/Os, 8 writes remain. This count shows why the buffer cache and delayed writing are indispensable. The order in which the writes are made determines crash consistency.

10.9.2 The buffer cache

Definition 10.13

The **buffer cache** is the kernel cache that keeps copies of disk blocks in memory. If the requested block is in the cache, it is a **hit**; if not, a **miss**.

The value of the cache comes from the large speed gap between memory and disk (Table 10.19). Reading a random 4 KiB from an NVMe SSD takes about a thousand times as long as a DRAM access; on a hard disk, seek and rotation are that much slower again.

Access	Latency
64 bytes from DRAM	≈ 80–120 ns
NVMe SSD 4 KiB random read	≈ 50–100 µs
HDD seek + rotation	≈ 5–10 ms (100–280 MB/s sequential)

Table 10.19: Typical access latencies (2026).

The classical buffer cache consists of two structures (Figure 10.42). A **hash table** finds a buffer from the (device, block number) pair in O(1) time; buffers that fall into the same bucket are chained. An **LRU list** arranges the buffers in order of last use. When space is needed, the buffer at the head of the list, the one unused for the longest time, is chosen. Important blocks such as i-nodes and directory blocks can be written without delay when they are modified; if they are damaged, the whole file system suffers.

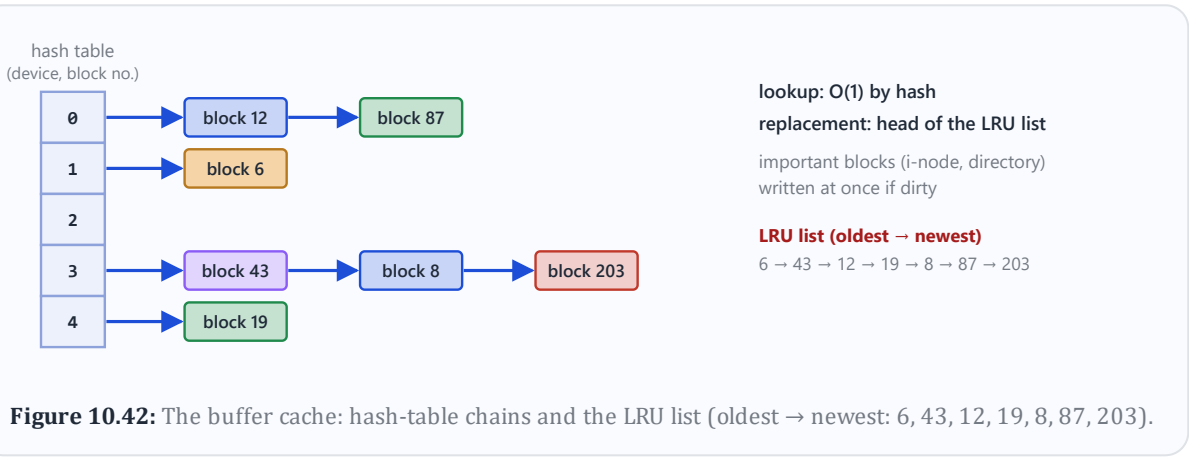


Figure 10.42: The buffer cache: hash-table chains and the LRU list (oldest → newest: 6, 43, 12, 19, 8, 87, 203).

There are two policies for writes. **Write-through** sends every change to disk immediately. It is safe but slow; old MS-DOS and removable media work this way, because the user may pull out the USB stick at any moment. **Write-back** accumulates the changed, that is, **dirty**, blocks in memory and writes them later. Linux writes dirty pages older than 30 seconds to disk in the background. If an application wants to wait until the data reaches the disk, it calls `fsync`.

Common mistake

Assuming that the data is on disk because `write` returned successfully. In a write-back cache, `write` only copies the data into the page cache. If the power fails within a few seconds, the data is lost. `close` does not guarantee writing to disk either. `fsync(fd)` or `O_SYNC` is required for durability.

10.9.3 Code trace: bread

The function `bread` below reads a block through the cache:

```
Buffer *bread(int blk) {
  Buffer *t = hash_find(blk);
  if (t != NULL) {          /* hit */
    lru_to_tail(t);
    return t;
  }
  t = lru_head();           /* oldest */
  if (t->dirty)
    disk_write(t);        /* write back */
  t->blk = blk; t->dirty = 0;
  disk_read(t);             /* miss: I/O */
  lru_to_tail(t);
  return t;
}
```

The function first searches the hash table for the block. If it finds it (a hit), it moves the buffer to the end of the LRU list, the newest position, and returns it; there is no disk I/O. If it does not find it (a miss), it takes the oldest buffer at the head of the list. If the buffer is dirty, its contents are first written to disk so that they are not lost. Then the buffer is assigned to the new block, its dirty bit is cleared, the content is read from disk, and the buffer is added to the end of the list.

Let the cache have 3 buffers: in LRU order, blocks 5, 9, and 2. Block 9 is dirty, that is, modified but not yet written to disk. The requests are 9, 7, 2, and 4 in turn. Table 10.20 gives the trace.

Request	Result	Evicted	LRU after (old → new)	Read / write
9	hit	—	5, 2, 9*	0 / 0
7	miss	5 (clean)	2, 9*, 7	1 / 0
2	hit	—	9*, 7, 2	1 / 0
4	miss	9 (dirty → written)	7, 2, 4	2 / 1

Table 10.20: Trace of `bread` (initial LRU: 5, 9*, 2; * dirty; simulated with Python).

The first request is a hit; block 9 is moved to the newest position and the oldest block is now 5. `bread(7)` is a miss. Block 5 is clean: since its copy on disk is up to date, there is no need to write it back. The buffer is reused for block 7 and 1 read is made. `bread(2)` is a hit again. After this step the dirty block 9 has come to the head of the list. `bread(4)` is a miss and the oldest buffer holds 9. So that the change is not lost, block 9 is first written to disk, and then block 4 is read. Evicting a dirty buffer doubles the cost of a miss. In the end, 2 of the 4 requests were hits, with 2 reads and 1 write in total. Without a cache, 4 reads and a separate write for block 9's change would have been needed.

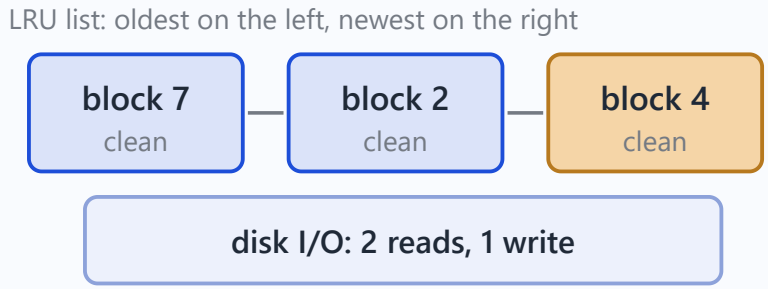


Figure 10.43: The cache after four requests: 7, 2, 4; 2 reads and 1 write in total.

10.9.4 The page cache and the unified cache

Files are accessed in two ways: with `read` / `write` calls and with memory-mapped I/O through `mmap`. In old systems these two paths used separate caches (Figure 10.44). The data of `mmap` lived in the virtual memory system's **page cache** and the data of `read` / `write` in the buffer cache. When the same file was used through both paths, the same data was in memory twice. This **double caching** both wasted memory and made it hard to keep the two copies consistent.

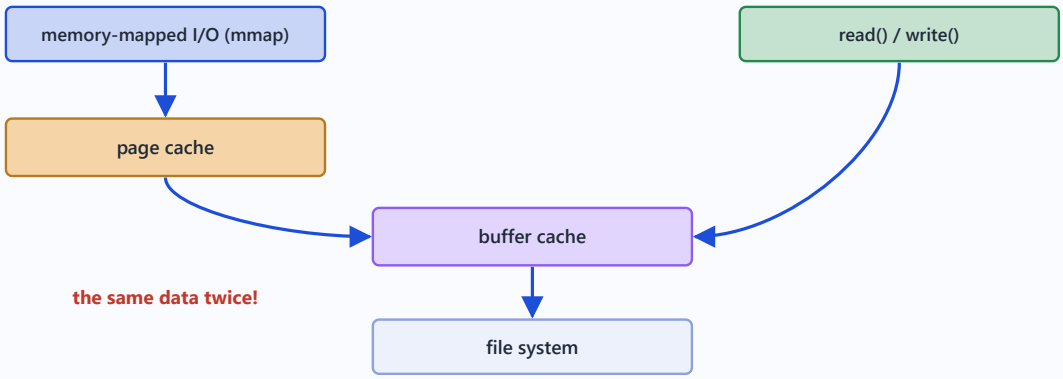


Figure 10.44: A non-unified cache: the data of `mmap` and of `read` / `write` in two separate caches, the same data twice.

In a **unified buffer cache** the file pages live in a single place and both paths see the same pages. Linux has worked this way since 2.4; Windows's cache manager and Solaris follow the same path. The page cache shares the same page-replacement algorithms as virtual memory: a file page and a process's anonymous page compete in the same pool. There are exceptions too. ZFS keeps its own ARC cache. Databases that manage their own cache skip the operating system's cache completely with the `O_DIRECT` flag.

10.9.5 Read-ahead, placement, and defragmentation

Read-ahead means sensing sequential access in advance. When block k is requested, block k + 1 is also read (if it is not in the cache); when the program asks for it, it is ready. Read-ahead helps only with sequential access. With random access, unnecessary reads keep the disk busy. A per-file trace of "is this file being read sequentially?" is therefore kept. Linux enlarges or shrinks the window according to the access pattern; the default window is 128 KiB. An application can also give hints with `posix_fadvise`.

On a hard disk the most expensive thing is arm movement. Putting blocks that will be read consecutively close together on the disk reduces this movement. With a bitmap, finding contiguous free blocks is easy. An old suggestion was to allocate blocks two at a time. Today extents, delayed allocation, and the multiblock allocator do this job. The placement of i-nodes also matters (Figure 10.45). (a) If i-nodes are gathered at the beginning of the disk, every file access requires a long arm movement between the i-node and the data. (b) If the disk is divided into cylinder groups or block groups and each group has its own i-nodes, the i-node and its data are in the same group. BSD FFS and ext2/3/4 work this way.

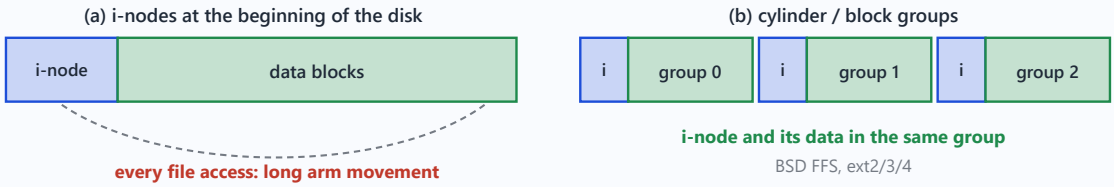


Figure 10.45: i-node placement: (a) at the beginning of the disk, (b) in each cylinder or block group.

Over time files become fragmented: their blocks are scattered over the disk. **Defragmentation** makes files contiguous again. On Windows the tool is called "Defragment and Optimize Drives." Linux has similar tools: `e4defrag`, `xfs_fsr`, `btrfs filesystem defragment`. ext4's allocation policy greatly reduces this need. On an SSD, since there is no seek time, defragmentation brings no gain; on the contrary, it means unnecessary writes, that is, wear. On an SSD Windows sends TRIM instead of defragmenting.

10.9.6 SSDs: FTL and TRIM

A page of flash memory cannot be updated in place; before it is written, the large erase block it belongs to must be erased as a whole. The **flash translation layer** (FTL) in the SSD's controller therefore directs every write to a new, empty page and maps the logical block address to that page. The old page becomes invalid. The inside of the SSD thus resembles a log-structured system. When free pages run low, the controller performs **garbage collection**: it copies the valid pages in an erase block elsewhere and erases the block.

A gap in information arises here. When the file system deletes a file, it merely marks the blocks free in its own bitmap. The SSD is unaware of this; it still considers those pages valid and copies them needlessly during garbage collection. Needless copies increase **write amplification**, which both lowers speed and accelerates wear. The solution is for the file system to tell the SSD about freed blocks with the **TRIM** command (Deallocate in NVMe). On Linux the notification is made either in bulk with the weekly `fstrim.timer` or at every delete with the `discard` mount option.

10.10 Consistency and recovery

10.10.1 Backup

Backup has two purposes. The first is **disaster recovery**: disk failure, fire, ransomware. The second is **recovery from carelessness**: restoring a file that was deleted or overwritten by mistake. In practice the second is needed far more often.

There are two basic forms of backup (Table 10.21). A **physical dump** copies the disk from block 0 to the end. It is very fast and simple, but it cannot skip directories, cannot take incremental backups, and makes restoring a single file hard. A **logical dump** copies chosen directories and files together with their paths. It is slower and more complex; in return it can take only the changed files and easily restores a single file, even onto another machine.

	Physical dump	Logical dump
What is copied?	all (used) blocks from block 0 to the end	chosen directories and files, with their paths
Speed, simplicity	very fast, simple	slower, complex
Skipping directories, incremental backup	cannot	can (the changed ones)
Restoring a single file	hard	easy; on another machine too
Bad blocks	the controller must remap them or the OS must know the list	not a problem

Table 10.21: Physical and logical dumps.

A common strategy combines full and incremental dumps (Figure 10.46). For example, a full dump once a month and, every day, only what has changed since the last dump. In a restore, the last full dump is written first onto an empty file system: first the directories, then the files. Then the incremental dumps are applied in order. Temporary files and special files (`/dev` , `/proc` , `/tmp`) are not backed up. A backup taken while the system is in use can be inconsistent: a directory copied at the start of the dump may have changed by its end. A **snapshot** (LVM, ZFS/Btrfs, Windows VSS) is therefore taken first, and the backup is made from that frozen image.

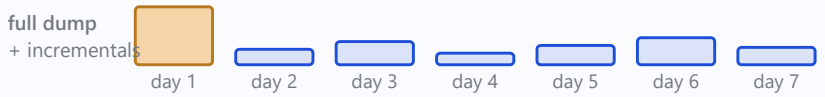


Figure 10.46: Full and incremental dumps: a full dump on the first day, only the changes on the following days.

Where backups are kept also matters. The **3-2-1 rule** says: 3 copies, 2 different media, 1 copy somewhere else (offline or immutable). Because ransomware can encrypt every disk on the network, the offline copy is the last safeguard. Tape still lives on today in archives; an LTO-10 cartridge holds up to 40 TB uncompressed.

10.10.2 The logical dump algorithm

UNIX's incremental logical dump algorithm works in four phases and uses an i-node bitmap. In the tree in Figure 10.47, squares are directories and circles are files; the numbers are i-node numbers. Under root 1 there are 2, 3, and 4. Under 2 are 5 and 6, under 4 are 7 and 8, under 7 is 9, and under 8 is 10. The files changed since the last dump are 3, 6, and 10.

1. **Phase 1.** The tree is walked starting from the root. Every changed file and all directories, changed or not, are marked: 1, 2, 3, 4, 6, 7, 8, 10.
2. **Phase 2.** The tree is walked again. The mark is removed from directories that have nothing changed below them: 7, because it contains only the unchanged 9.
3. **Phase 3.** The marked directories are dumped in i-node order: 1, 2, 4, 8. These directories are dumped even if they are unchanged, because they are needed to rebuild the paths of the changed files.
4. **Phase 4.** The marked files are dumped: 3, 6, 10.

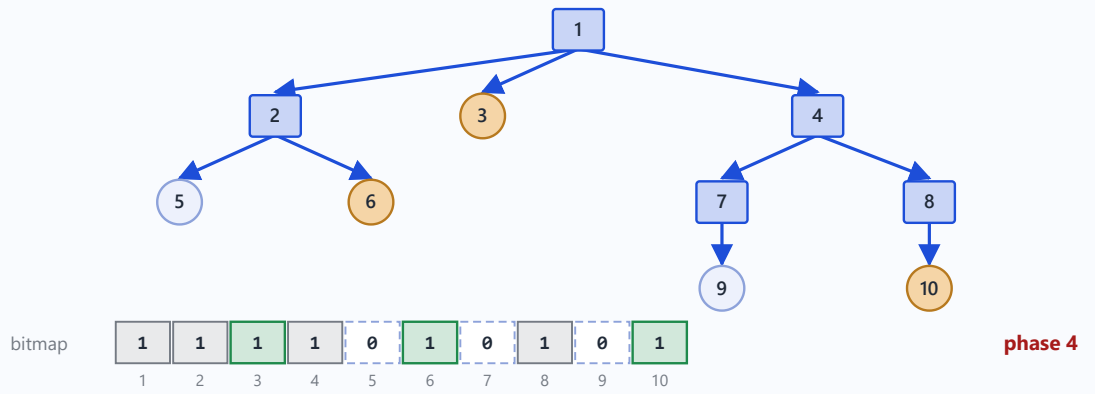


Figure 10.47: The last phase of the logical dump: the marked files 3, 6, 10 are dumped. In the bitmap 5, 7, and 9 are unmarked.

Dumping directories before files makes the restore easier: first the directory tree is built, then the files are put in their places. There are a few more details. The free list is not dumped; it is rebuilt after the restore. A file with several hard links should be dumped only once, and all of its names must be linked again on restore. The holes of sparse files should not be filled with zeros; otherwise the restored file swells to its real size on disk. FIFOs and device files are dumped not by content but by type.

10.10.3 Consistency checking: fsck

After a crash, a file system may be left inconsistent: some of the changes have reached the disk and some have not. UNIX's `fsck` program (`chkdsk` on Windows) finds and repairs these inconsistencies. It makes two kinds of checks: block consistency and file consistency.

For **block consistency**, fsck keeps two counters for each block. It reads all the i-nodes and increments the "in use" counter of the blocks each i-node uses. Then it reads the free list or the bitmap and increments the "free" counter. In a consistent file system the sum of the two counters is 1 for every block: the block is either in a file or free (Figure 10.48). Three problems can arise:

- **Missing block** (0, 0): the block appears nowhere. It is harmless but it is a loss of space; the block is added to the free list.
- **Duplicate block in the free list** (0, 2): the same block could later be given to two files at once. The free list is rebuilt. With a bitmap this situation cannot occur, because a bit is either 0 or 1.
- **Duplicate data block** (2, 0): two files use the same block. This is the most dangerous: when one of the files changes, the other is corrupted too. fsck allocates a new block, copies the content, gives the copy to one of the files, and informs the user.

Figure 10.48: The fsck counters: block 5 appears in two files (a duplicate data block). A new block is allocated, the content is copied, and the user is notified.

If a block appears both in use and on the free list (1, 1), it is removed from the free list.

For **file consistency**, fsck walks all directories starting from the root and keeps a seen counter for each i-node. Because of hard links, an i-node can appear in several directories. At the end the counter is compared with the link count in the i-node. If the link count is greater than the seen count, there is only a waste of space: even if all the names are deleted, the link count does not fall to 0 and the file is never freed. If it is smaller, the situation is dangerous: when one of the names is deleted, the link count falls to 0 and the file is freed while another name still exists. In both cases the link count is set equal to the seen count. I-nodes that appear in no directory but are in use are put in the `lost+found` directory.

Example 10.12

The fsck counters came out as follows (blocks 0–9): in use 1 0 1 1 0 2 1 0 0 1; on the free list 0 1 0 0 1 0 0 0 2 0. Which are the problem blocks?

1. The total should be 1 for each block. Block 5: (2, 0), a duplicate data block; the most dangerous.
2. Block 7: (0, 0), a missing block; it is added to the free list.
3. Block 8: (0, 2), twice on the free list; the list is rebuilt.

fsck's big problem is its running time. All the i-nodes and all the directories have to be read. On a terabyte disk a full scan takes hours, and the system cannot be used during this time. This is why journaling and copy-on-write were born.

10.10.4 Crashes and atomicity

The root of the problem is this: a file operation changes several blocks, but the disk guarantees atomically only the writing of a single sector. Deleting a file takes three steps (Figure 10.49): removing the directory entry, returning the i-node to the free pool, and returning the blocks to the free pool. If a crash occurs after the first step, the i-node and the blocks are left ownerless; this is a leak, and fsck finds it. If a crash occurs after the second step, the blocks leak. If the order were

reversed, things would be much worse. If the i-node is freed first and given to another file, and a crash occurs before the directory entry is deleted, the directory points to the wrong file. The data of the two files gets mixed up.

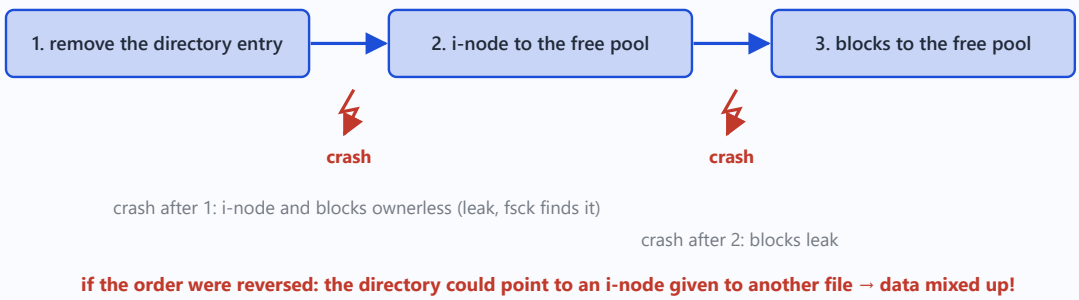


Figure 10.49: The three steps of deleting a file and the crash points.

There are four families of solutions to this problem. (1) Repair afterward: fsck. (2) Write the intent first: journaling. (3) Never overwrite: copy-on-write and log-structured file systems. (4) Order the writes carefully: BSD's *soft updates* method makes the writes in an order that leaves only harmless leaks, whenever the crash occurs.

Definition 10.14

An operation that gives the same result whether applied once or many times is called **idempotent**.

Journaling relies on idempotence. During recovery an operation may be replayed many times: the crash can even occur in the middle of recovery. “Mark block n free” is idempotent; however many times it is done, the block is free. “Append n to the end of the free list” is not idempotent; if done twice, n is on the list twice. This is why it is not operations but the new contents of blocks that are written to the journal.

10.10.5 Journaling

Definition 10.15

Journaling means writing changes first to a separate area on disk, the **journal**, and only then applying them to their final places. In databases this idea is called write-ahead logging.

Suppose a new block is appended to a file. Three blocks change: the file's i-node (I), the data bitmap (B), and the new data block (Db). In ext4's default **ordered** mode the operation proceeds in five steps (Figure 10.50).

1. **Data.** The new data block Db is first written to its final place.
2. **Journal write.** The transaction-begin record TxB and the metadata blocks to be changed, that is, the new versions of I and B, are written sequentially to the journal.
3. **Commit.** When it is certain that the earlier writes have reached the disk (a barrier or FLUSH command), the transaction-end record TxE is written. If TxE is on disk, the transaction is committed.
4. **Checkpoint.** I and B are written to their final places. It does not matter if these writes are interrupted midway; they can be repeated from the journal.
5. **Free.** The transaction is marked complete in the journal's superblock. The journal is reused like a circular buffer.

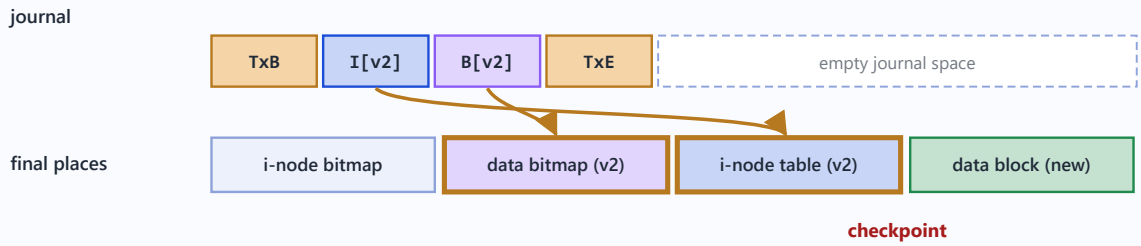


Figure 10.50: Checkpoint: in the journal TxB, I[v2], B[v2], TxE; in the final places the bitmap and the i-node table have been updated to v2.

Recovery is simple. At boot the journal is scanned. Transactions that have a TxE are replayed from the beginning: their blocks are rewritten to their final places (*redo*). Because rewriting is idempotent, it does not matter that some of the blocks were already written before the crash. Transactions without a TxE are discarded; since their final places were never touched, the file system is in its old, consistent state. fsck is not needed; recovery takes seconds.

Whether the crash occurred before or after TxE is therefore decisive. If a crash occurs before TxE is written, the transaction counts as never having happened and the change is lost. If a crash occurs after TxE is written, for example in the middle of the checkpoint, the transaction is replayed from the journal and the change is preserved. In both cases the file system stays consistent.

ext4 has three modes. In **journal** mode the data is also written to the journal; it is the safest but the slowest, because every piece of data is written twice. In **ordered** mode only the metadata goes into the journal, but the data is always written before the metadata; thus an i-node never points to a block containing garbage. In **writeback** mode there is no ordering guarantee for the data. Among the systems that use journals are ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), and JFS. At Apple, the journaled HFS+ gave way to APFS, which uses copy-on-write.

10.10.6 Code trace: recovery from the journal

The function `recover` below scans a journal of n records from the beginning. The type of a record is `TXB`, `BLOCK`, or `TXE`; `BLOCK` records carry the number of the target block and its new content.

```
void recover(Rec g[], int n) {
  int i = 0;
  while (i < n && g[i].type == TXB) {
    int j = i + 1;
    while (j < n && g[j].type == BLOCK)
      j++;
    if (j == n || g[j].type != TXE)
      break;              /* uncommitted: drop */
    for (int k = i + 1; k < j; k++)
      write(g[k].dest, g[k].data);
    i = j + 1;          /* next */
  }
}
```

`i` points to the start of a transaction. The outer loop continues as long as `g[i]` is a TxB. The inner loop moves `j` over the block records to the end of the transaction. If `j` has reached the end of the journal, or there is no TxE there, the transaction is uncommitted: the loop breaks and the transaction is discarded. The blocks of a committed transaction are copied to their final places with `write`, and `i` moves to just after the TxE. The function scans the journal once and writes each block record at most once: O(n).

The system crashed while the third transaction was being written. The journal has ten records: TxB 1, I7 v1, B v1, TxE 1, TxB 2, I9 v2, D40 v2, TxE 2, TxB 3, I7 v3. All blocks in the final places are still v0. Table 10.22 gives the trace.

i	j at the end of the inner loop	g[j]	Decision	Final places after (I7, B, I9, D40)
0 (TxB 1)	3	TxE 1	replay: I7 ← v1, B ← v1	v1, v1, v0, v0
4 (TxB 2)	7	TxE 2	replay: I9 ← v2, D40 ← v2	v1, v1, v2, v2
8 (TxB 3)	10 = n	—	uncommitted: drop, exit the loop	v1, v1, v2, v2

Table 10.22: Trace of `recover` (run by translating it one to one into Python).

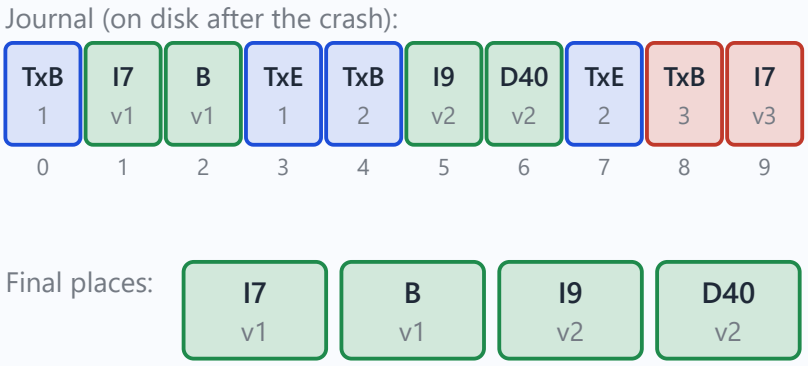


Figure 10.51: The end of recovery: transactions 1 and 2 were applied; the I7 v3 version of transaction 3 was never written.

Transactions 1 and 2 are applied completely, and transaction 3 is as if it never happened. The v1 version of I7 may already have been written to its final place before the crash; rewriting it changes nothing. The file system is consistent and no fsck scanning the disk from end to end is needed.

Common mistake

Replaying every block found in the journal. The I7 v3 record of transaction 3 sits in the journal, but it has no TxE. Writing it would mean applying only part of a transaction; that is inconsistency itself. Only committed transactions are replayed.

10.10.7 Copy-on-write

Journaling writes every change twice: first to the journal, then to its final place. File systems that use **copy-on-write** (CoW) follow another way: they never overwrite any block. The file system is a tree of blocks (Figure 10.52). At the top is the superblock (the *uberblock* in ZFS), below it the root, the internal nodes, and at the bottom the data blocks.

1. **Leaf.** Let block L3 be the one to change. L3 is not updated in place; its new version L3' is written to a free spot. The old L3 is still intact.
2. **Path.** The parent nodes that point to L3 must also change, because their pointers will change. I2 and the root are also copied: I2', R'. The unchanged subtrees (I1, L4) are shared with the new version.
3. **Atomic switch.** Finally the superblock is updated with a single atomic write to point to R'. If the crash occurs before this write, the old tree is seen; if after, the new tree. There is no intermediate state; no journal is needed.

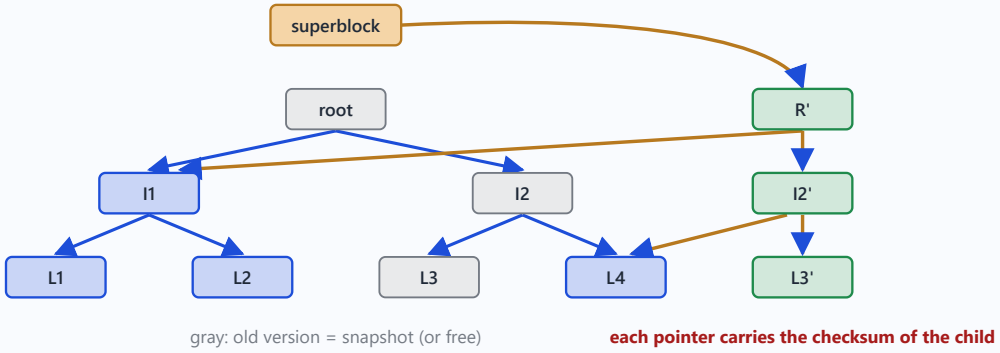


Figure 10.52: Copy-on-write: the changed leaf and its path are written to a new place, and the superblock is updated atomically. Gray: the old version, which can be kept as a snapshot.

This design has two nice by-products. If the old root is kept, the whole old tree is still valid: a **snapshot** comes for free. ZFS and Btrfs also keep, next to each pointer, a **checksum** of the block it points to. If a block on the disk is silently corrupted, it is caught on reading and repaired from the mirror copy. The operation that reads the whole disk and does this check is called a *scrub*.

The systems that use copy-on-write are ZFS (Sun, 2005; today OpenZFS), Btrfs (Linux), APFS (Apple, 2017), and ReFS (Windows Server). The price shows up with random overwrites. Because every small change in the middle of a file is written to a new place, the file becomes fragmented over time. Because a change to one leaf requires copying the entire path, write amplification also rises. CoW can therefore be turned off for database and virtual-machine disk files.

10.10.8 The log-structured file system

In 1991 Rosenblum and Ousterhout proposed a radical design for the Sprite operating system: the **log-structured file system** (LFS). The design's assumption was this: as memories grow, most reads are served from the cache, and the traffic going to disk is mostly writes. Writes should then be done in the fastest way, that is, with a single long sequential write.

In LFS the whole disk is a single log. Writes are accumulated in memory: data blocks, modified i-nodes, and pieces of the i-node map (imap). When the buffer fills, everything is written sequentially to the end of the log as a single **segment** (for example 1–2 MiB) (Figure 10.53). I-nodes have no fixed place; they move at every write. The imap finds the latest location of an i-node from its number. The **checkpoint region** (CR) at a fixed place on the disk points to the latest pieces of the imap.

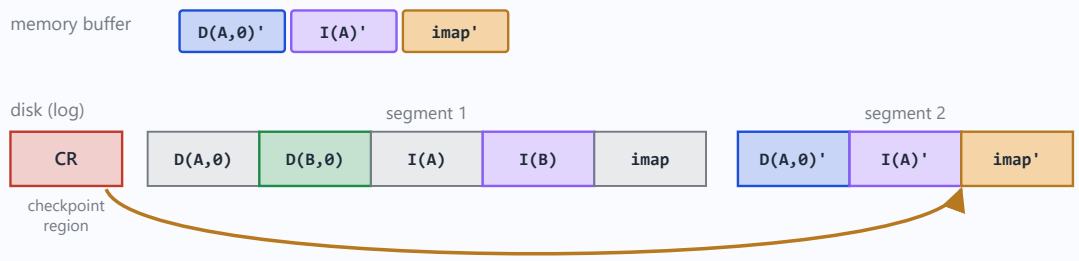


Figure 10.53: When a block of A changes, the new version is written to segment 2; the old D(A,0), I(A), and imap in segment 1 become dead.

When a block of file A changes, its new version, the new i-node, and the new piece of the imap are written to a new segment. The old copies in segment 1 are now dead but continue to occupy space. This space must be reclaimed before the disk fills up. The **cleaner** does this job: it reads segments with few live blocks, compacts the live blocks into a new segment, and frees the old segments. The cost of cleaning grows when the disk is full and under load; this is LFS's weak point.

LFS raises write bandwidth almost to the disk's sequential speed. The idea lives on in many places today: Android's F2FS, Linux's NILFS2, NetApp's WAFL, the copy-on-write of ZFS and Btrfs, and the FTLs of SSDs. The inside of an SSD is already a log-structured system.

10.11 VFS and the network file system

10.11.1 The virtual file system

On a Linux machine many file systems are mounted at the same time: ext4 in the root directory, XFS on a data disk, exFAT on an attached USB stick, an NFS share mounted from the network, and `/proc`, which lives in memory. The application reaches all of them with the same `open` and `read` calls. A layer of the kernel provides this unity.

> **Definition 10.16**
>
> The **virtual file system** (VFS) is the kernel layer that unifies different file systems behind a common object interface. Each concrete file system implements the operations of this interface with its own functions.

The Linux VFS works with four kinds of object: the **superblock** (a mounted file system), the **inode** (a file; the v-node in Sun's original design), the **dentry** (a name component in a path), and the **file** (an opening). Each object carries an operations table: a structure made of pointers to functions. The mechanism is set up in four steps (Figure 10.54).

1. **Registration.** A file system registers with the VFS at boot or when its module is loaded, using `register_filesystem()`, and supplies the table of addresses of the functions the VFS will call.
2. **Mounting.** The VFS asks that file system to build a superblock object. The object carries the superblock operations table (`s_op`).
3. **open.** The path is resolved with dentries. The concrete file system places the i-node information into a VFS i-node; the tables of i-node operations (`i_op`) and file operations (`f_op`) are attached. A file object is created and added to the process's descriptor table.
4. **read.** The VFS merely calls the function pointer `file->f_op->read_iter`. If the file is on ext4, ext4's function runs and brings the data through the page cache, the block layer, and the NVMe driver. If the file is on NFS, the NFS client's function runs and sends a remote procedure call (RPC) over the network.

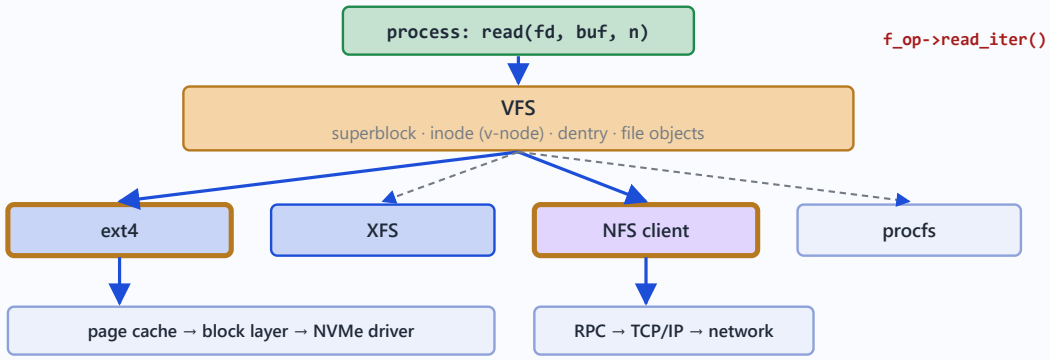


Figure 10.54: The VFS layer: the `read` call is routed to the concrete file system's function through the `f_op->read_iter` pointer in the file object.

This is polymorphism written in C: exactly the same as a virtual method call in an object-oriented language. The application sees no difference. The extra cost is a single indirect call; the loss in performance is small. The purpose of the VFS is to present a single interface to different file systems *at the same time*. Keeping several file systems on the same disk does not require the VFS; partitioning already provides that. On Windows the I/O manager and the file-system drivers play the same role.

Common mistake

Believing that the application uses a different system call depending on the file system, or that the kernel tries the code of all file systems in turn. The system call is the same. The right code is selected with a single indirect call from the operations table that was attached to the file object during `open`.

10.11.2 NFS

The **network file system** (NFS) attaches a directory of a remote machine to the local tree. The application uses remote files like local files. Sun designed NFS for SunOS in 1984 and published the protocol as an open standard. This is why it is found today in almost every UNIX-like system.

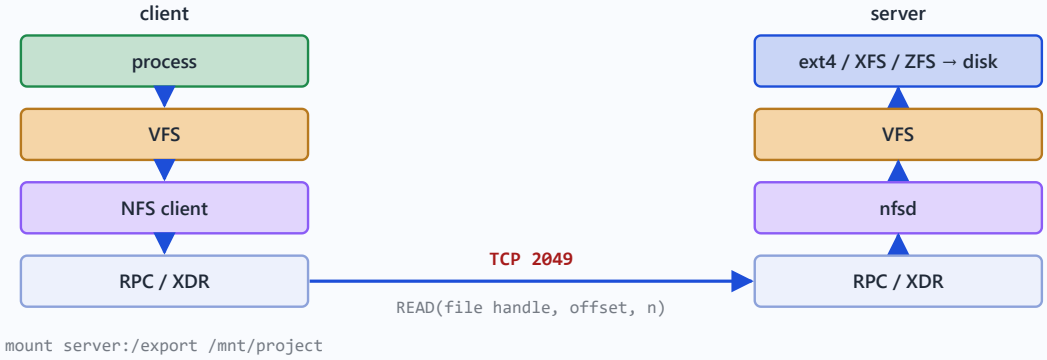


Figure 10.55: The NFS client and server layers: the NFS client under the VFS on the client, and nfsd and the local file system on the server.

Figure 10.55 shows the layers. On the client, `mount server:/export /mnt/project` mounts the remote directory. When a process reads a file under `/mnt/project`, the call goes from the VFS to the NFS client. The client prepares a `READ(file handle, offset, n)` request and sends it to the server on TCP port 2049. The `nfsd` on the server passes the request through its own VFS to the local file system (ext4, XFS, ZFS) and sends the reply back.

NFS achieves machine and architecture independence with two tools. Remote procedure call (RPC) makes a request-reply exchange look like a function call. External data representation (XDR) encodes numbers and structures in a form independent of byte order, so a client and a server of different architectures understand each other. The versions of NFS also show an important distinction in design. NFSv3 is **stateless**: the server does not remember which client has opened which file, and every request carries everything needed (file handle, offset). Even if the server crashes and restarts, the clients merely repeat their requests. NFSv4, however, is **stateful**. It uses only TCP and a single port (2049); it includes Kerberos security, ACLs, file locking, and delegation. pNFS in v4.1 allows data to be read in parallel from several servers. NFS originally used UDP/IP; today TCP is preferred. The only requirement is an IP network. Its counterpart in the Windows world is SMB 3, which supports encryption and multichannel connections.

10.12 Summary

- The disk offers only numbered blocks. On top of this, the **file system** builds **files**, which are named, persistent byte sequences, and the **directories** that map them. The type of a file is reliably recognized not by the extension but by the **magic number**.

- Attributes live in the i-node and the name in the directory entry. `stat` reads the attributes; mtime is the moment the content changed, and ctime the moment the i-node changed.
- An open file is tracked at three levels: the per-process **descriptor table** → the system-wide **open-file table** (position, mode) → the in-memory **i-node**. `fork` and `dup` share the position, and a second `open` creates a separate position. A short read is not an error; a return of 0 is end of file.
- Safe saving: write to a temporary file → `fsync` → `rename`.
- Directory structures evolved from single-level to trees and directed acyclic graphs; UNIX forbids hard links to directories. A path is resolved **component by component**: if the root i-node is in memory, k components take 2k reads, and 2k + 1 reads with the first data block.
- A **hard link** is a second name for the same i-node and raises the link count; a **symbolic link** stores the path of the target and is left dangling when the target is deleted. **Mounting** combines separate file systems in a single tree.
- The rwx bits are kept separately for owner, group, and others; the first matching class applies. On a directory w allows adding and deleting entries and x allows passing through. The setuid and sticky bits and the umask (permission = request & ~umask) complete this scheme.
- MBR is limited to 2 TiB by its 32-bit LBAs; GPT offers 64-bit LBAs, 128 partitions, and a backup header. The location of an i-node is computed from its number: offset = n × size, disk block = table start + offset div block size.
- **Contiguous allocation** is fast but produces external fragmentation. The **linked list** is flexible, but random access costs O(n) disk reads. **FAT** moves the chain into memory, but the table grows with the disk. The **i-node** needs memory only for open files; with 12 direct pointers and single, double, and triple indirect pointers it reaches ≈ 4 TiB with 4 KiB blocks. Extents describe contiguous blocks with a single record.
- Directories are kept as a linear list, a hash, or a B+ tree; in large directories the tree gives both O(log n) search and ordered listing.
- Free space is tracked with a **bitmap**, a linked list, **grouping**, or **counting**. The bitmap has a fixed size; the list is small only when the disk is full. The block size is a trade-off between data rate and space efficiency.
- Creating a file without a cache and writing 12 KiB costs 25 I/Os. The **buffer cache** works with a hash and an LRU list; evicting a dirty buffer doubles the cost of a miss. The unified page cache prevents double caching. Read-ahead and close placement speed up sequential access; blocks deleted on an SSD are reported with **TRIM**.
- The 3-2-1 rule applies to backups; the logical dump has four phases. **fsck** checks block and link counts but is slow. **Journaling** replays transactions that have a TxE and discards those that do not. **Copy-on-write** never overwrites any block and changes the superblock atomically. **LFS** writes everything to a sequential log and reclaims space with a cleaner.
- The **VFS** presents a single interface to different file systems through tables of function pointers. **NFS** attaches remote directories to the local tree with RPC and XDR.

10.13 Key terms

file system file descriptor open-file table directory, directory entry path name (absolute / relative) hard link / symbolic link mount permission bits (rwx) access control list (ACL) superblock i-node contiguous allocation linked allocation indexed allocation internal / external fragmentation bitmap buffer cache page cache journaling copy-on-write virtual file system (VFS) network file system (NFS)

10.14 Test Yourself

1. A 9000-byte file is read with a 2048-byte buffer until `read` returns 0. What do the `read` calls return, in order?

   Answer

   2048, 2048, 2048, 2048, 808, and 0. Four full reads bring 8192 bytes; the remaining 9000 − 8192 = 808 bytes arrive with a short read, and the last call reports end of file. 6 calls in total.

2. `mkdir p`, `mkdir p/x`, `mkdir p/y`, `echo z > p/f`, `ln p/f p/g`, and `ln -s f p/h` are run. What are the link counts of `p`, `p/x`, `p/f`, and `p/h`?

   Answer

   `p`: 4 (the entry in the parent directory, its own `.`, and the `..` entries of x and y). `p/x`: 2. `p/f`: 2 (f and g are the same i-node). `p/h`: 1 (the symbolic link's own i-node).

3. The i-node table starts at block 3; the block is 4 KiB, the i-node 128 B, and numbers start at 0. In which disk block and which sector (512 B) does i-node 50 start?

   Answer

   50 × 128 = 6400 = 1 × 4096 + 2304: block 1 of the table, that is, disk block 3 + 1 = 4, byte 2304 inside the block. The byte address is 4 × 4096 + 2304 = 18,688; 18,688 / 512 = 36.5 → sector 36, byte 256 inside it.

4. The cluster is 2 KiB; in the FAT, FAT[8] = 3, FAT[3] = 13, FAT[13] = 6, FAT[6] = EOF. In which

cluster, and at which byte of the cluster, is byte 5000 of the file whose first cluster is 8?

Answer

5000 div 2048 = 2: the file's cluster 2 (from 0). Since the chain is 8 → 3 → 13 → 6, cluster 2 is physical 13. The position within the cluster is 5000 mod 2048 = 904. Because the chain is followed in memory, the disk read is only for the data cluster itself.

5. The block is 8 KiB, the address 4 B; the i-node has 12 direct, 1 single, 1 double, and 1 triple indirect pointer. About how large is the largest file?

Answer

P = 8192 / 4 = 2048. The number of blocks is $12 + 2048 + 2048^2 + 2048^3 = 8{,}594{,}130{,}956$; × 8192 B ≈ 70.4×10^{12} B ≈ 64 TiB. Doubling the block size raises the capacity by a factor of about $2^4 = 16$: the block grows twofold and each of the three levels holds twice as many addresses.

6. 4 KiB block, 4 B address, 12 direct pointers. At which level is byte 10,000,000, and how many disk reads are needed if the i-node is in memory?

Answer

10,000,000 div 4096 = 2441. Since 2441 − 12 − 1024 = 1405 ≥ 0, it is in the double indirect region: outer index 1405 div 1024 = 1, inner index 1405 mod 1024 = 381. The offset in the block is 1664. Reads: the double indirect block, the single indirect block, the data = 3.

7. On a 24-block disk the free blocks are 1, 2, 3, 7, 8, 9, 10, 15, 20, and 21. How is this free space represented with the counting method and with a bitmap (1 = free)?

Answer

Counting: (1, 3), (7, 4), (15, 1), (20, 2). Bitmap: 01110001 11100001 00001100. Ten free blocks reduce to four pairs.

8. In a cache with 3 buffers the LRU order is 3, 8, 6 (3 is the oldest and dirty). The requests are 8, 1, 3, 6. How many reads and how many writes does `bread` make?

Answer

8 is a hit (order 3, 6, 8). 1 is a miss; since the oldest, 3, is dirty, it is written first, then 1 is read (6, 8, 1). 3 is a miss: the block just evicted is read again, and 6 is evicted (8, 1, 3). 6 is a miss and 8 is evicted (1, 3, 6). In total 3 reads, 1 write. In this small cache LRU evicts blocks that are requested right back.

9. After a crash the journal contains these records: TxB 1, I4 v1, TxE 1, TxB 2, D8 v2, B v2. In the final places everything is v0. At the end of `recover`, in which version are I4, D8, and B?

   Answer

   Transaction 1 is committed: I4 ← v1. Transaction 2 has no TxE; j reaches the end of the journal (6) and the transaction is discarded. The result is I4 v1, D8 v0, B v0. Although D8 v2 is in the journal, it is not written.

10. A file's permission is `-rw-r--r--` and the permission of the directory it is in is `drwxrwxrwx`. Can an ordinary user who is not the file's owner delete the file? What would change if the directory's permission were `drwxrwxrwt`?

    Answer

    Yes. Deleting removes a directory entry and requires the directory's w right; the file's own permission is irrelevant. If the sticky bit (the t at the end) were set, only the file's owner, the directory's owner, or root could delete the entry.

10.15 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters “File System Implementation,” “Crash Consistency: FSCK and Journaling,” and “Log-structured File Systems”: they describe the I/O counts and journaling steps of these notes in much more detail. The book is free at ostep.org.
- M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry, “A Fast File System for UNIX,” ACM TOCS, 1984: the classic paper that introduced cylinder groups and the idea of close placement and explains the layout of today's ext4.
- M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” ACM TOCS, 1992: describes the design of LFS, the cleaner policies, and the measurements.
- Linux kernel documentation, “Overview of the Linux Virtual File System” and “ext4 Data Structures and Algorithms”: first-hand descriptions of the VFS objects and of ext4's on-disk layout, extent tree, and journal.

10.16 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th edition, Wiley, 2018 — Chapters 13 (File-System Interface), 14 (File-System Implementation), 15 (File-System Internals).
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th edition (2014) / 5th edition (2023), Pearson — Chapter 4: File Systems.

3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — “Files and Directories,” “File System Implementation,” “Fast File System,” “Crash Consistency: FSCK and Journaling,” “Log-structured File Systems,” “Flash-based SSDs,” “Data Integrity,” “Sun's NFS.”
4. M. Rosenblum, J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” ACM TOCS, 1992; M. K. McKusick et al., “A Fast File System for UNIX,” ACM TOCS, 1984.
5. Linux kernel documentation: “Overview of the Linux Virtual File System,” “ext4 Data Structures and Algorithms”; Microsoft Learn: the NTFS and exFAT specifications; OpenZFS and Btrfs documentation; the UEFI specification (GPT).

Input/Output

A computer manages a wildly varied set of devices at the same time. A keyboard sends a few bytes per second, while an NVMe disk serves millions of requests per second. Their speeds differ by nine orders of magnitude, and their interfaces have little in common. The I/O part of the operating system tames this variety in two ways. On the hardware side, it issues commands to devices, reads their status, catches interrupts, and moves data without tying up the CPU. On the software side, it presents every device to applications through one uniform interface. The chapter starts with controllers, device registers, and the two addressing schemes. It then compares polling, interrupt-driven I/O, and DMA on one running printer example. The layers of I/O software, buffering, asynchronous interfaces, and clocks follow. The last sections cover the access time of a hard disk, disk-arm scheduling algorithms, SSDs, RAID arrays, and stable storage. Every numerical example (head movements, access times, buffer traces, XOR parity) was checked by actually running the algorithms.

LEARNING OUTCOMES

Hardware
Recognize device controllers, device registers, and buses such as PCIe, USB, and NVMe; explain port-mapped and memory-mapped I/O.

Transfer methods
Compare polling, interrupt-driven I/O, and DMA in terms of CPU cost; state the roles of MSI-X and the IOMMU.

Software layers
Follow a `read()` call through the interrupt handler, the device driver, the device-independent layer, and the user library; trace the circular buffer and the timer list in code.

Interfaces
Distinguish blocking, nonblocking, and asynchronous I/O; compare the models of epoll and io_uring.

Disks
Compute access time; find the head path and total head movement for FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK.

SSD and RAID
Explain the FTL, wear leveling, TRIM, and blk-mq; work out RAID 0/1/5/6/10 layouts, XOR parity, the small write, and recovery.

11.1 Overview of the I/O system

11.1.1 What the operating system does with devices

When a program reads a file, prints text on the screen, or receives a packet from the network, it ultimately has to talk to a device. The program does not do that talking itself. The operating system issues commands to devices, reads their status, catches the interrupts that announce completion, and handles errors. To the application it offers a layered, device-independent, uniform interface. The same `read()` call can read from an SSD, a USB stick, or a serial port.

One number conveys the scale of the job. Roughly 60–70% of the Linux source code lives in the `drivers/` directory. The largest part of the kernel is neither process management nor memory management but the drivers for thousands of different devices. Every new network card, graphics card, or sensor adds another driver to the kernel.

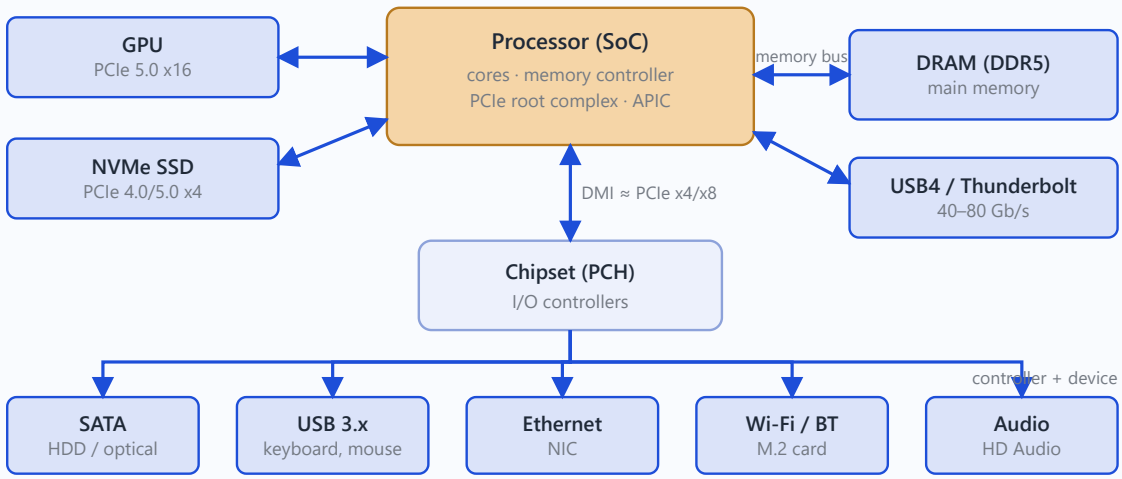


Figure 11.1: The I/O layout of a modern computer. The memory controller and the PCIe root complex are inside the processor. Fast devices (GPU, NVMe SSD) attach directly to the processor over PCIe. Slow devices come in through controllers in the chipset (PCH).

Figure 11.1 shows the I/O layout of a present-day desktop computer. Every device consists of two parts: a controller and the actual device it drives. The operating system talks only to the controller. Controllers, in turn, connect to the processor over various buses.

11.1.2 Buses: PCIe, USB4, SATA

Definition 11.1

A **bus** is a connection that carries data, address, and control signals between components. Older PCs had parallel buses (ISA, PCI) shared by many devices. Today point-to-point serial links are used: PCIe, USB, SATA.

On a shared parallel bus only one device can talk at a time. As the bus gets wider, timing skew between the wires becomes a problem too. Serial links solve both problems at once. Each link joins exactly two endpoints, and data flows at a very high frequency over a single pair of wires. In PCIe a **lane** is full duplex; a link combines x1, x4, x8, or x16 lanes. Bandwidth is available separately in each direction: an x4 link can use the same speed whether an NVMe disk is reading or writing.

Link	Raw rate	Effective data rate	Note
PCIe 4.0	16 GT/s per lane	≈ 1.97 GB/s per lane; x4 ≈ 7.9 GB/s	128b/130b encoding; NVMe SSDs
PCIe 5.0	32 GT/s per lane	≈ 3.94 GB/s per lane; x16 ≈ 63 GB/s	GPUs, SSDs in the 14 GB/s class
PCIe 6.0	64 GT/s (PAM4)	≈ 7.6 GB/s per lane; x16 ≈ 121 GB/s	FLIT + FEC; spreading in servers
USB 2.0 / 3.2	480 Mb/s / 5–20 Gb/s	≤ 60 MB/s / ≤ 2.5 GB/s	Host-polled protocol
USB4 v1 / v2	40 / 80 Gb/s	Tunnels PCIe and DisplayPort	Compatible with Thunderbolt 4/5
SATA III	6 Gb/s	≈ 600 MB/s (8b/10b)	AHCI: one queue, 32 commands
Ethernet	1 / 10 / 100 / 400 Gb/s	125 MB/s … 50 GB/s	Data center: 400–800 GbE

Table 11.1: The main links in use today and their speeds.

Example 11.1

One lane of PCIe 4.0 runs at 16 GT/s (16 billion transfers per second) and uses 128b/130b encoding. What are the effective data rates of one lane and of an x4 link?

1. **Encoding overhead.** Of every 130 bits, 128 are data; the remaining 2 bits are a synchronization header. The effective bit rate is 16 × 128 / 130 ≈ 15.75 Gb/s.
2. **Converting to bytes.** 15.75 / 8 ≈ 1.97 GB/s. This is the rate of one lane in one direction.
3. **An x4 link.** 4 × 1.97 ≈ 7.9 GB/s. This is why PCIe 4.0 NVMe disks on the market top out around 7 GB/s: once protocol overhead is added, the limit is close to this figure.

The same calculation for PCIe 5.0 gives 32 × 128 / 130 / 8 ≈ 3.94 GB/s, which is 63 GB/s on an x16 link. PCIe 6.0 changes the encoding entirely: PAM4 signaling carries two bits per symbol, and data travels in 256-byte FLIT packets that include forward error correction (FEC). 242 bytes of each FLIT are useful payload. This gives 64 × 242 / 256 / 8 ≈ 7.6 GB/s.

Some rows of the table are historical as well. FireWire, the 56K modem, and 10/100 Mb/s Ethernet are now just textbook examples. The Pentium-era “northbridge” has disappeared too. The memory controller moved into the processor (AMD in 2003, Intel in 2008). Today fast devices connect directly to the processor over PCIe, and slow devices to the chipset.

11.1.3 Device data rates

The first property that separates devices is their speed. Figure 11.2 shows typical upper limits on a logarithmic axis. Each line on the axis is 1000 times the one before it.

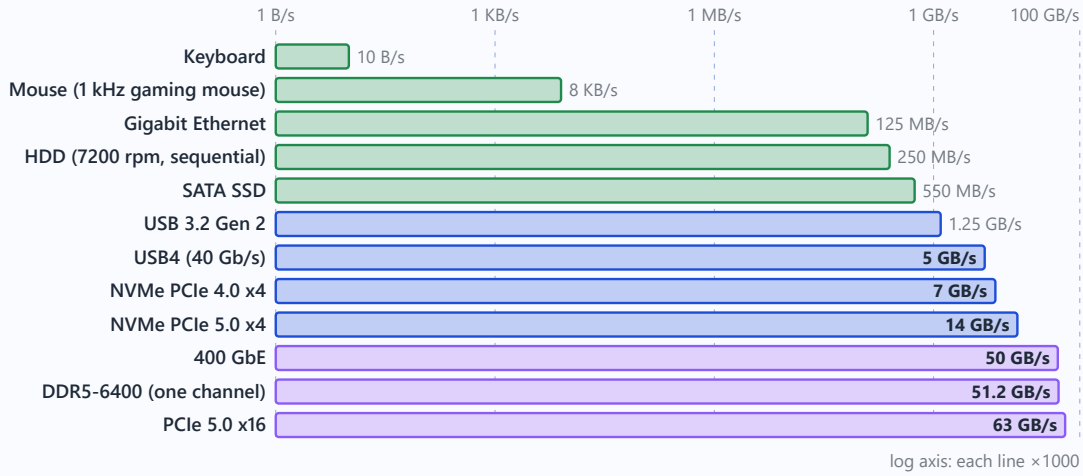


Figure 11.2: Device data rates on a logarithmic axis: the keyboard about 10 B/s, Gigabit Ethernet 125 MB/s, a PCIe 4.0 x4 NVMe disk 7 GB/s, a PCIe 5.0 x16 link 63 GB/s.

Between a keyboard and PCIe 5.0 x16 there is a factor of $63 \times 10^9 / 10 \approx 6 \times 10^9$. The same operating system has to handle both a few keystrokes per second and millions of NVMe requests per second. That is why no single transfer method suits every device. For a slow device, it does no harm for the CPU to attend to every byte individually. For a fast device, the CPU must stay out of the data path altogether. The values in the figure are typical upper limits. Real speed depends on signal quality, protocol overhead, and the device itself.

11.1.4 Device classes: block, character, network

After speed, the second distinction is how the data is addressed. The Unix tradition divides devices into three classes.

Definition 11.2

A **block device** stores data in fixed-size, numbered blocks; each block can be read and written independently of the others. A **character device** produces or consumes a stream of bytes; the stream is not addressable and there is no seek operation. A **network device** works with packets and is used through the socket interface.

Class	Example	Property	Typical interface
Block	HDD, SSD, USB stick, SD card	Fixed-size, addressable blocks (512 B / 4 KiB); each block is read and written independently	`read`, `write`, `lseek`; file system, `mmap`
Character	Keyboard, mouse, serial port, printer, audio	Byte stream; not addressable, no position	`read` / `write`, `ioctl`; `/dev/ttyS0`
Network	Ethernet, Wi-Fi	Packet-based; neither block nor character	Socket API (`send`, `recv`); no file in `/dev`
Other	Clocks, timers, GPU	Neither stream nor block: generates interrupts or is memory-mapped	`timerfd`, DRM/KMS, `mmap`

Table 11.2: Device classes.

On Linux, block and character devices appear as special files under `/dev`. The first letter of the `ls -l` output tells the class:

```
$ ls -l /dev/nvme0n1 /dev/ttyS0
brw-rw---- 1 root disk    259, 0  nvme0n1   b = block, major/minor 259,0
crw-rw---- 1 root dialout   4, 64 ttyS0     c = character
```

`b` marks a block device and `c` a character device. The two numbers where the file size would be are the major and minor numbers. The major number (259) tells the kernel which driver to call, and the minor number (0) says which unit of that driver is meant. File systems can be built only on block devices, because it must be possible to reach any block of a file by its number.

Common mistake

Counting a network interface as a character device. In Unix and Linux, networking is a separate class; there is no file in `/dev` for `eth0` or `wlan0`, and these interfaces are reached through sockets, not `open()`. The classification does not fit every device perfectly either. A tape drive writes blocks, but seeking is very slow. A clock produces neither a byte stream nor blocks; it only generates interrupts.

11.2 I/O hardware

11.2.1 Device controllers

Definition 11.3

An I/O unit has two parts: the mechanical or actual device part, and the electronic **device controller** that drives it. The controller is a chip or a card and attaches to the device through a connector. The operating system talks to the controller, not to the device.

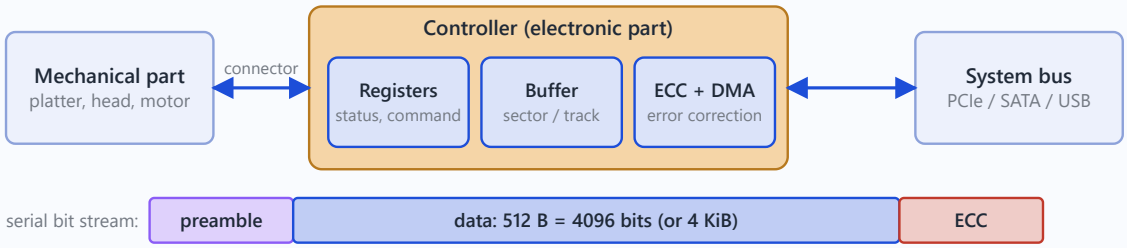


Figure 11.3: A disk controller sits between the mechanical part and the system bus. Inside are registers, a sector buffer, and an error-correction unit. The serial bit stream from the disk consists of a preamble, data, and an ECC field.

The controller's job is to turn the device's raw signals into units the operating system understands. Consider the disk controller in Figure 11.3. A stream of bits passes under the read head. The stream starts with a **preamble**. The preamble carries a pattern that synchronizes the controller's clock to the stream, together with the cylinder and sector numbers. Then come 512 bytes (4096 bits) or 4 KiB of data, and last the error-correcting code (ECC). The controller assembles the bit stream into a block, corrects errors with the ECC, and places the block in its own buffer. Only then is the block transferred to memory.

A standard controller interface is a great convenience for the operating system. Thanks to the SATA or NVMe interface, a single driver can run disks from many manufacturers. The operating system scans the buses at boot and whenever a device is hot-plugged. It reads the PCIe configuration space and the USB descriptors, learns the device's identity, and loads the appropriate driver. This arrangement is called **plug and play**.

Sector sizes have changed too. Modern disks use 4 KiB physical sectors (Advanced Format). Disks that present 512 B sectors to the outside for compatibility with old software are called 512e. When an unaligned 512-byte write arrives at such a disk, the controller first reads the entire 4 KiB sector, patches in the changed part, and writes the sector back.

11.2.2 Device registers: status, command, data

The controller talks to the operating system through its registers. A controller usually has four registers.

Figure 11.4: The four registers of an example controller. The status register holds the busy (BUSY), ready (RDY), and error (ERR) bits; the command register holds the write (WRITE) and command-ready (CMDRDY) bits. The value 0100 0001 in the data-out register is the code of the character 'A' (0x41).

- **Status register:** the CPU reads it. It reports whether the device is busy, whether the result is ready, and whether an error has occurred.
- **Command register:** the CPU writes it. It tells the controller what to do: read or write, and in which mode.
- **Data-out register:** data going from the CPU to the device is placed here.
- **Data-in register:** data coming from the device waits here for the CPU to read it.

The rule is simple: writing a register issues a command, and reading a register learns the status. Some controllers have large data buffers in addition to registers. The frame buffer of a graphics card and the transmit and receive rings of a network card (NIC) are examples. The layout of the registers is device-specific, and only the driver knows this detail. The bit names in Figure 11.4 are illustrative too; the register layout of a real SATA or NVMe controller is different.

11.2.3 Port-mapped and memory-mapped I/O

The question of how the CPU reaches a device register has two classic answers.

Definition 11.4

Port-mapped I/O places device registers in an address space separate from memory, the I/O ports. The CPU reaches these ports with the special `IN` / `OUT` instructions. **Memory-mapped I/O** (MMIO), on the other hand, puts the registers at ordinary memory addresses. Access uses normal load and store instructions; the address decoder routes the access to the device concerned.

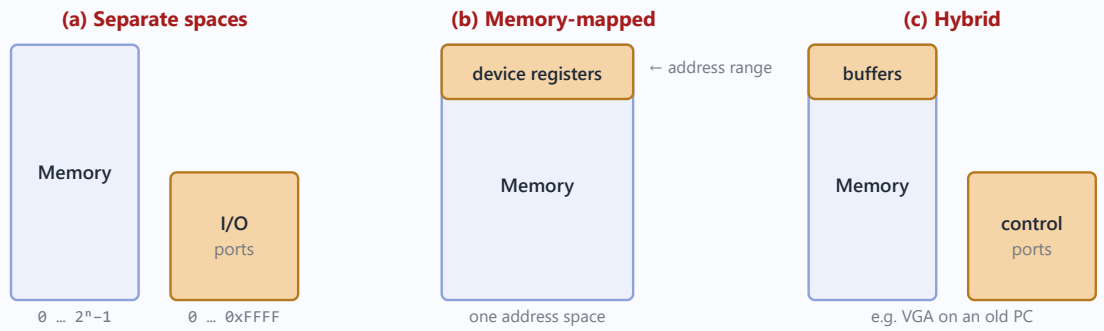


Figure 11.5: Three approaches: (a) separate address spaces for memory and I/O ports, (b) the memory-mapped layout, in which a range of a single address space is reserved for device registers, (c) the hybrid layout, in which control registers sit in ports and data buffers sit in memory.

The x86 family carries both methods. Many devices inherited from the IBM PC still sit at fixed port numbers (Table 11.3). The classic example of the hybrid layout in Figure 11.5 is the VGA card of the old PC: its control registers were in ports, while the screen memory was at memory addresses.

Port (x86)	Device
0x20–0x21	8259 interrupt controller
0x40–0x43	8254 timer (PIT)
0x60, 0x64	8042 keyboard controller
0x3F8–0x3FF	COM1 serial port
0xCF8–0xCFF	PCI configuration

Table 11.3: Some fixed I/O ports on x86.

The two access styles look like this in machine code:

```
in  al, 0x60          ; port: special instruction
mov eax, [rbx+0x14]   ; MMIO: ordinary load

Linux: inb()/outb()  ·  ioremap() + readl()/writel()
/proc/ioports  ·  /proc/iomem
```

The first line reads a byte from port 0x60, that is, from the data register of the keyboard controller. This needs a separate instruction (`in`). The second line is an ordinary memory load. If `rbx` points to the address where a device is mapped, this instruction reads the register at offset 0x14 of that device; from the CPU's point of view it is no different from reading memory. In the Linux kernel, ports are reached with `inb()` / `outb()` , while memory-mapped registers are first mapped with `ioremap()` and then accessed with `readl()` / `writel()` . Which ports and which physical address ranges belong to which device can be seen in the files `/proc/ioports` and `/proc/iomem` .

Today the weight is clearly on memory-mapped I/O. ARM and RISC-V processors have no port instructions at all; they use only MMIO. Even on x86, PCIe devices expose their registers memory-mapped through address windows called **BARs** (base address registers).

11.2.4 Memory-mapped I/O: pros and cons

Memory-mapped I/O became widespread for three reasons. First, no special instructions are needed. A driver can be written entirely in C; a register is read and written through a `volatile` pointer. Second, protection comes with the page table. If the page holding the registers is not mapped into user space, no user program can touch the device. If desired, this page can be given to a single process; user-space drivers such as VFIO and DPDK work this way. Third, every instruction that accesses memory also works on a register: instructions such as `TEST` or `AND` can operate directly on the register.

The method has costs as well.

- **Caching.** If the status register is cached, the CPU's later reads look at an old copy in the cache rather than at the register itself. Even when the device is ready, the program cannot see it. For this reason register pages are marked uncacheable (PAT/MTRR on x86; `ioremap` does this on Linux).
- **Multiple buses.** If memory sits on a separate, fast bus, I/O devices cannot see the addresses on that bus (Figure 11.6). A bridge or the root complex decides which address range goes to memory and which to devices.
- **Compiler and processor ordering.** The compiler may hoist a read that seems to have an unchanging value out of a loop, or swap two writes. The register pointer must be `volatile`, and memory barriers must be placed where order matters.

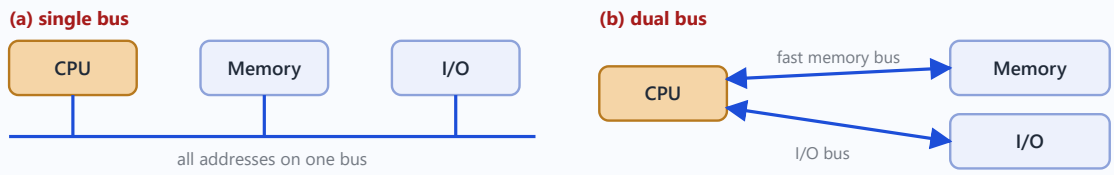


Figure 11.6: (a) On a single bus the CPU, memory, and I/O devices all see every address. (b) With dual buses, memory is on a separate, fast bus; I/O devices cannot see memory addresses, and routing is done by address ranges.

Common mistake

Not declaring the pointer `volatile` in a driver that waits for the status register with `while ((*status & READY) == 0) ;`. The compiler assumes the value of `*status` does not change inside the loop, performs the read once, and the loop never ends. The same symptom appears when the register page is mapped cacheable. Both safeguards are needed together: a `volatile` pointer and an uncacheable mapping.

11.3 Programmed I/O

11.3.1 Polling and the handshake

Let us examine in its plainest form how the CPU and a controller hand over a byte. The two sides agree through a few bits in the status and command registers. This agreement is called a **handshake**.

> **Definition 11.5**
>
> **Polling** means the CPU repeatedly reads the device's status register until the device is ready. Because the CPU does nothing else while it waits, it is also called **busy waiting**. The method in which the CPU itself carries every byte of data is called **programmed I/O** (PIO).

```
while (status & BUSY)
    ;          /* busy-wait */
command = WRITE;
data_out = 'A';
command |= CMDRDY;
/* --- controller side --- */
/* sees CMDRDY → BUSY = 1 */
/* read command, send byte */
/* CMDRDY = ERR = BUSY = 0 */
```

The driver executes the upper half of the code; the controller carries out the steps described by the comments in the lower half. Writing one byte, the character 'A' (0x41), takes eight steps:

1. **Waiting.** The driver reads the status register. The controller has not finished the previous job, so BUSY = 1 and the `while` loop keeps spinning. The CPU can do nothing else in the meantime.
2. **Exit.** When BUSY becomes 0, the loop exits. This wait can last anywhere from a few cycles to milliseconds, depending on the device.
3. **Command.** The WRITE bit is written to the command register: the job to do is a write.
4. **Data.** The byte to send (0x41) is placed in the data-out register.
5. **Command ready.** CMDRDY = 1 is set. This bit tells the controller that the job can begin. Order matters: CMDRDY must be set only after the command and data are in place.
6. **Controller busy.** The controller sees CMDRDY and sets BUSY = 1, thereby telling the host that it is busy.
7. **Work.** The controller reads the command register (WRITE), takes the byte from the data-out register, and sends it to the device.
8. **Completion.** CMDRDY, ERR, and BUSY are cleared. ERR = 0 shows that the job finished successfully. The loop starts over for the next byte.

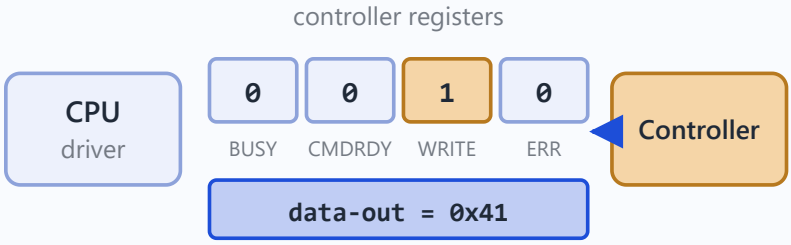


Figure 11.7: The end of the handshake: BUSY = 0, CMDRDY = 0, ERR = 0. The data-out register holds the last byte sent (0x41).

Programmed I/O is simple and needs no extra hardware. But the CPU spins uselessly throughout the wait. Worse, if the CPU switches to another process and does not pick up a byte in time, data can be lost. If the controller's buffer overflows, a newly arrived byte has nowhere to go.

11.3.2 The printer example

Tanenbaum's classic example is a user process sending the string “ABCD” to the printer. The kernel part of the system call looks like this:

```
copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY)
        ;   /* busy-wait */
    *printer_data_register = p[i];
}
return_to_user();
```

The first line copies the user's string into a kernel buffer (`copy_from_user`). The kernel does not use user memory directly, because the process could modify that memory from another thread, or the address might be invalid. Then two things happen for each character. The inner `while` loop spins until the status register is READY. When the printer is ready, the character is written to the data register; the printer prints it and becomes busy again. After all four characters have been printed, control returns to the process.

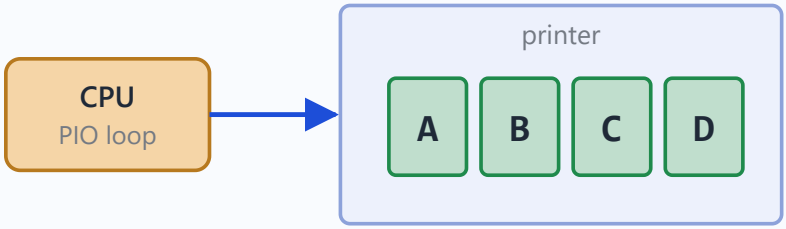


Figure 11.8: The end of the PIO loop: the printer has printed the characters A, B, C, D. The CPU stayed tied to this process the whole time.

Example 11.2

A printer prints 100 characters per second. With PIO, how long does the CPU wait for each character? How many loop iterations does a 2 GHz CPU spend in that time?

1. **Time per character.** 1 s / 100 = 10 ms. After each character the CPU polls the status register for about 10 ms.
2. **Wasted iterations.** 10 ms × 2 × 10^9 iterations/s = 2 × 10^7 iterations. In that time the CPU could have executed tens of millions of instructions.
3. **Result.** The four-character job takes about 40 ms and is limited entirely by the printer's speed. A faster CPU changes nothing.

Even so, PIO is not always bad. In an embedded system where the CPU has nothing else to do, or when the device responds very quickly, polling is the simplest and fastest solution. The next example shows in numbers when polling is harmless and when it is impossible.

Example 11.3

On a 2 GHz CPU, one poll (read the status, compare, fetch the data) takes 600 cycles. What percentage of CPU time is spent on a keyboard polled 100 times per second, and on a device that produces 50 MB/s and hands over 32 bytes per poll?

1. **Ratio.** CPU share = (polls/s × cycles/poll) / (cycles/s).
2. **Keyboard.** 100 × 600 = 60,000 cycles/s; 60,000 / (2 × 10^9) = 0.003%. Negligible.
3. **Fast device.** 50 × 10^6 / 32 ≈ 1.56 × 10^6 polls/s. That comes to 1.56 × 10^6 × 600 ≈ 9.4 × 10^8 cycles/s; about 47% of CPU time goes to polling alone.
4. **Interpretation.** For a slow device with rare events, polling is harmless. For high-speed block transfer, it is unacceptable for the CPU to deal with every piece individually; the solution is DMA.

11.4 Interrupts

11.4.1 Interrupt hardware: from a wire to MSI

The flaw of polling is that the CPU queries the device constantly. A better arrangement is for the device to notify the CPU when it finishes.

Definition 11.6

An **interrupt** is a signal by which a device asks for the CPU's attention. The CPU finishes the current instruction, suspends the running program, and runs the **interrupt handler** belonging to that device. When the handler finishes, control returns to the suspended program.

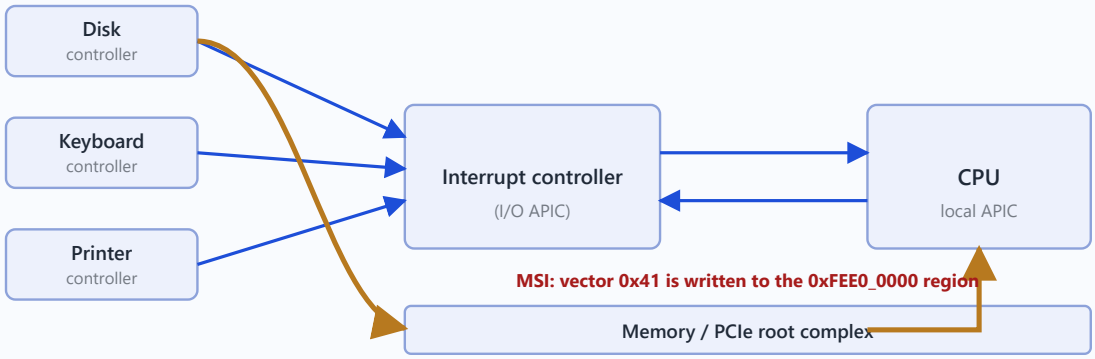


Figure 11.9: Interrupt hardware. Device controllers connect to the interrupt controller (I/O APIC), which connects to the core's local APIC. A modern PCIe device uses no wire at all: it sends a memory write containing the vector number (here 0x41) to the 0xFEE0_0000 region.

In the classic arrangement, each device has a line (IRQ) to the interrupt controller. The controller shares the CPU's single interrupt input among the devices and sets priorities. A disk read is signaled in this order:

1. **Signal.** When the disk controller finishes its job, it asserts its own interrupt line.
2. **Forwarding.** If no other interrupt is being handled at that moment, or the new interrupt has higher priority, the interrupt controller forwards the interrupt to the CPU. Otherwise the request is held back; the line stays active until the CPU acknowledges it.
3. **Vector.** The CPU accepts the interrupt and the controller supplies a vector number. The CPU uses this number as an index into the interrupt vector table and finds the address of the handler.

This description is based on the historical PC arrangement: two 8259 PICs and 15 IRQ lines. The controller “putting the vector on the address bus” is likewise true only for that arrangement. Today an x86 system has an I/O APIC and one local APIC per core (x2APIC); on ARM the GIC takes over this job.

Definition 11.7

MSI (message-signaled interrupts) and its extended form **MSI-X** deliver an interrupt through a memory write instead of a wire. The device sends a write containing the vector number to a special address region (0xFEE0_0000 on x86); the local APIC of the target core turns it into an interrupt.

MSI has two advantages. Line sharing disappears, because each device can obtain as many separate vectors as it wants. The interrupt can also be steered to a chosen core. MSI-X allows up to 2048 vectors for one PCIe function. NVMe disks and network cards get a separate vector for each queue and send that queue's interrupt to the core that uses the queue.

11.4.2 Vectors, priority, NMI, and exceptions

Definition 11.8

An **interrupt vector** is an index into the interrupt vector table. Each entry of the table holds the address of a handler. On x86 this table is called the IDT; it has 256 entries, and its location is given by the IDTR register.

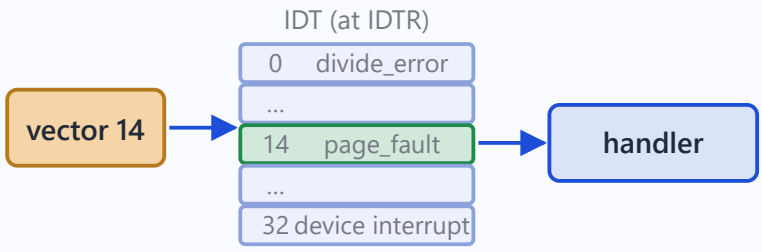


Figure 11.10: Vector 14 (page fault) selects entry 14 of the IDT; the entry gives the address of the handler.

Some of the vectors belong not to devices but to the processor itself. On x86, vectors 0–31 are reserved for processor exceptions (Table 11.4).

Vector	Event	Type
0	Divide error	fault
1 / 3	Debug / breakpoint	trap

Vector	Event	Type
2	NMI (non-maskable interrupt)	interrupt
6	Invalid opcode	fault
13	General protection fault	fault
14	Page fault	fault
18	Machine check	abort
32–255	Device interrupts, IPIs, APIC timer	maskable

Table 11.4: Some interrupt vectors on x86.

Definition 11.9

An **exception** is a synchronous event caused by the running instruction itself. Two kinds stand out. A **fault** (such as a page fault) prevents the instruction from completing; after the handler, the same instruction is executed again. A **trap** (a system call, `int3`) is triggered deliberately; after the handler, execution continues from the next instruction.

Interrupts differ among themselves as well. Maskable interrupts can be deferred in a critical region; on x86 the `cli` instruction disables interrupts and `sti` enables them again. An NMI (non-maskable interrupt), however, cannot be deferred. Hardware errors, memory ECC errors, and the watchdog use this path, because these events cannot tolerate waiting. Priority matters too. The APIC orders interrupts by vector class; a high-priority interrupt can interrupt a low-priority handler.

When there are more devices than vectors, one vector may be shared by several devices. In that case the table entry points not to a single handler but to a list of handlers. When the interrupt arrives, the handlers on the list take turns asking their own device “was this interrupt mine?”. This arrangement is called **interrupt chaining** or a shared IRQ. On Linux, which vector arrived how many times on which core can be seen in the file `/proc/interrupts`; the `irqbalance` service spreads interrupts across cores.

11.4.3 Interrupt handling step by step

Let us follow how an interrupt is handled from start to finish in a concrete case. Process A is running; process B is waiting for a disk read to finish. The disk read completes and the controller requests an interrupt.

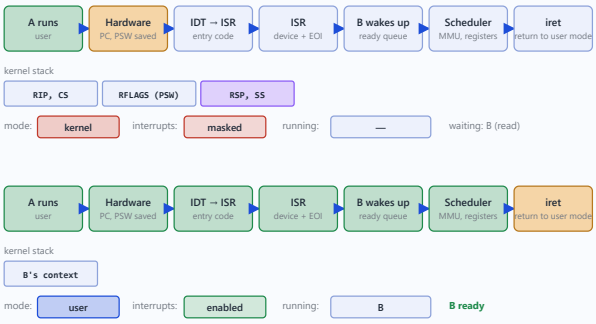


Figure 11.11: Two moments of the interrupt-handling flow. Above, the hardware saves the context; below, control returns to the chosen process with `iret`.

1. **Acceptance.** The CPU finishes the current instruction and then accepts the interrupt. An interrupt is taken only at an instruction boundary.
2. **Hardware save.** The hardware pushes the program counter (RIP on x86-64), the status word (PSW, RFLAGS), and the stack pointer onto the kernel stack. It switches to kernel mode and masks interrupts.
3. **Finding the handler.** The vector number becomes an index into the IDT, and the handler's address is found. The entry code (assembly) also saves the registers that the hardware did not save. Now the context and stack are ready for the interrupt service routine (ISR) written in C.
4. **ISR.** The handler reads the device's status, fetches the data, and sends an acknowledgment (EOI, end of interrupt) to the interrupt controller. After that, new interrupts may arrive. Work that will take long is deferred to later, to the bottom half.
5. **Wake-up.** The waiting process B is moved from the blocked state to the ready queue.
6. **Scheduling.** The scheduler picks the next process; let us assume it picks B here. B's page table is loaded into the MMU (the CR3 register on x86), and its registers are restored from the process table or from the stack.
7. **Return.** The `iret` instruction restores the saved PC and PSW, returns to user mode, and enables interrupts. B resumes at the instruction where it left off.

This sequence follows Tanenbaum's steps for an interrupt handler. The scheduler could also have picked A; that choice depends on the policy. Apart from making a waiting process ready, an interrupt does not change the scheduling decision by itself.

Common mistake

Thinking that the interrupt handler runs before the state is saved. Since the handler will use A's registers, first the hardware saves the return address and status word, and then the entry code saves the other registers. Without the save, the return address would be lost and A could not continue where it left off. The correct order: the device requests, the CPU stops at an

instruction boundary, the state is saved, the handler is found through the vector, the handler acknowledges the device and wakes the waiter, the scheduler chooses, and control returns with `iret`.

11.4.4 Precise and imprecise interrupts

For a program to resume where it left off after an interrupt, "where it left off" must be well defined. In a pipelined processor this does not come automatically.

Definition 11.10

An interrupt is **precise** if it satisfies four conditions: (1) the PC is saved in a known place; (2) all instructions before the PC have completed fully; (3) none of the instructions after the PC has started; (4) the state of the instruction at the PC is known. An interrupt that does not satisfy these conditions is called **imprecise**.

Figure 11.12: (a) Precise interrupt: PC = 316; the instructions at addresses 300–312 have completed, and 316 onward have not started. (b) Imprecise interrupt: instructions are left at different stages (304 at 60%, 312 at 30%, 316 at 80%, ...).

In a pipeline, many instructions are at different stages at the same time. On a superscalar processor, instructions can also complete out of order. In the situation of part (b) of Figure 11.12, a single PC value does not describe where the program stopped. To restart, either the hardware must undo half-finished work or the operating system must save and decode a large amount of internal state. The second option makes the handler both slower and more complicated.

Modern x86, ARM, and RISC-V processors provide precise interrupts. They do this with a **reorder buffer** (ROB). Even though instructions execute out of order, the ROB "retires" them in program order, that is, makes their results permanent. When an interrupt arrives, the effects of instructions that have not yet retired are discarded. The price is chip area and design complexity, but the operating system stays simple as a result.

11.4.5 Interrupt-driven I/O: the printer example

The printer example can be rewritten with interrupts. The goal is that the CPU does not spin while waiting for the printer.

Definition 11.11

In **interrupt-driven I/O**, the CPU gives the device a job and goes off to run another process. When the device finishes the job, it raises an interrupt; the interrupt handler either hands out the next job or wakes the waiting process.

```
/* system call */
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
i = 1; count = count - 1;
scheduler();      /* block process */
/* interrupt service routine */
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

The code has two parts. The upper part is the system call. It copies the string into a kernel buffer, enables the printer's interrupts, writes only the first character when the printer is ready, updates the counters, and calls the scheduler. The process blocks and the CPU runs another process. The lower part is the interrupt service routine. An interrupt arrives each time the printer finishes a character. The routine looks at `count`: if it is zero, all characters have been printed and the user process is woken. Otherwise the next character is written, `count` decreases by one, and `i` increases by one. In both cases the interrupt is acknowledged and control returns to the interrupted process.

Event	Written	`i`	`count`	Process
System call	p[0] = 'A'	1	2	blocked
Interrupt 1 ('A' printed)	p[1] = 'B'	2	1	blocked
Interrupt 2 ('B' printed)	p[2] = 'C'	3	0	blocked
Interrupt 3 ('C' printed)	—	3	0	ready (`unblock_user`)

Table 11.5: Interrupt-driven printing of the string “ABC” (initially `count` = 3).

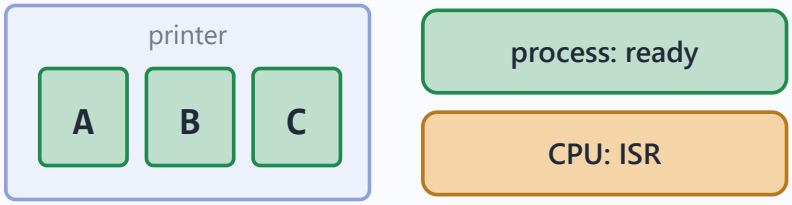


Figure 11.13: After the third interrupt: A, B, C have been printed; the ISR has moved the user process to the ready queue.

Table 11.5 reveals one detail: three characters produce three interrupts, but the third interrupt writes no new character. It only reports that the job is done. Because the system call writes the first character, the number of interrupts equals the number of characters. The price of the method is the context save and handler time per interrupt, typically 1–2 µs. While waiting for the printer, the CPU runs other processes; this gain is far larger than the few microseconds an interrupt costs.

11.4.6 Top half, bottom half, and hybrid polling

While an interrupt handler runs, at least interrupts on the same line are disabled. If the handler runs long, interrupts from other devices are delayed or even lost. For this reason operating systems split the handler in two.

Definition 11.12

The **top half** (*hardirq* on Linux) is the actual interrupt handler: it is kept short, acknowledges the device, copies the data into a buffer, and schedules the rest of the work. The **bottom half** is deferred work and runs with interrupts enabled. It takes the form of softirqs, tasklets, workqueues, or threaded IRQs (`request_threaded_irq`) on Linux, and of ISR + DPC (deferred procedure call) on Windows.

There are other ways to reduce the number of interrupts.

- **Interrupt coalescing.** Network cards and NVMe disks report several completions in a single interrupt. The number of interrupts drops, and latency rises slightly.
- **NAPI: interrupts combined with polling.** Under heavy network traffic, the Linux driver turns the interrupt off and polls the queue. When traffic falls, it returns to interrupts. Interrupts are efficient for rare events and polling for frequent ones; the hybrid method takes the best of both.
- **Pure polling.** When the event rate is very high and latency is critical, interrupts are not used at all. DPDK and SPDK, NVMe polled queues, and the IOPOLL mode of io_uring choose this path. One core is dedicated entirely to polling.

Interrupts also have a failure mode. If a faulty device raises interrupts continuously, the processor does nothing but run the handler and the system locks up. This is called an **interrupt storm**. If many interrupts arrive that no handler claims, Linux shuts the line off with a “nobody cared” message; there are also mechanisms that limit the interrupt rate.

11.5 Direct memory access (DMA)

11.5.1 The DMA controller

Interrupt-driven I/O removes the waiting, but an interrupt still arrives for every byte or word. Moving a 4 KiB disk block word by word means thousands of interrupts. The solution is to take the transfer job away from the CPU altogether.

Definition 11.13

Direct memory access (DMA) is a hardware mechanism that performs block transfers between a device and memory without the CPU's involvement. The CPU only sets up the transfer and deals with the single interrupt that arrives when the transfer is finished.

The registers of a DMA controller describe the transfer: a memory address, a byte count, and a control register. The control register determines the I/O port, the direction (read or write), the transfer unit (byte or word), and how many units are moved at a time (the burst size). A multichannel controller keeps a separate register set for each channel and manages many transfers at once.

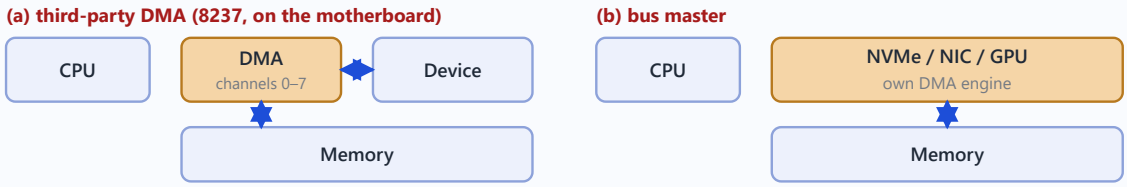


Figure 11.14: (a) Third-party DMA: a single DMA controller (8237) on the motherboard performs transfers on behalf of devices. (b) Bus master: an NVMe disk, a network card, or a GPU writes directly to memory with its own DMA engine.

The arrangement in which a single DMA controller on the motherboard serves all devices holds only for the historical 8237 chip (Figure 11.14 (a)). Today most devices have their own DMA engine, and these devices perform PCIe writes directly to memory. The concept stays the same: the CPU sets up the transfer, and the device carries it out.

11.5.2 A DMA transfer step by step

Let us follow a disk sector being read into memory with DMA. The sector is 4096 bytes and will be written to address 0x7A000 in memory.

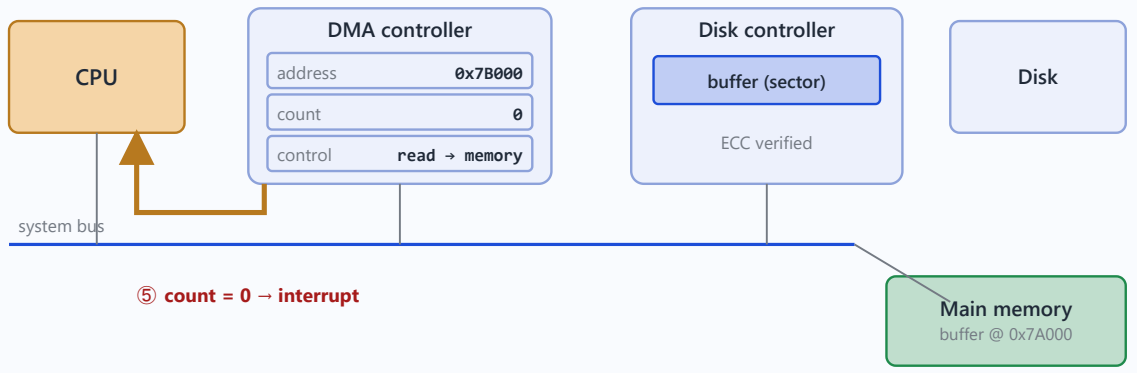


Figure 11.15: The last step of a DMA transfer: the count has dropped to 0, and the DMA controller sends an interrupt to the CPU. The data is already in memory.

1. **Setup.** The CPU programs the DMA controller (address = 0x7A000, count = 4096, direction = read) and the disk controller ("read the sector"), then returns to other work.
2. **Buffer.** The disk controller reads the sector into its own buffer and verifies it with the ECC. Thanks to the buffer, no data is lost if the bus is busy at that moment.
3. **Request.** The DMA controller requests the bus, places the memory address on it, and sends a read request to the disk controller.
4. **Write.** The disk controller writes the word directly to memory. The CPU is not involved in this transfer.
5. **Acknowledgment.** The disk controller sends an acknowledgment; the DMA controller increments the address and decrements the count. The request, write, and acknowledgment steps repeat until the count reaches 0.
6. **Interrupt.** When the count reaches 0, the DMA controller alerts the CPU with an interrupt. By the time the operating system handles the interrupt, the data is already in memory.

How DMA uses the bus matters too. In **cycle stealing** mode, the controller takes the bus from the CPU for a few cycles for each word. The CPU does not stop completely; it only waits a little when it wants to access memory. In **burst** mode the bus is held for an entire block; the transfer is faster, but the CPU may wait longer. There is one more distinction. In **fly-by** mode, data goes from the device to memory directly. In the other mode, the DMA controller first reads the word itself and then writes it to the destination; these two bus cycles are costly, but they also allow memory-to-memory copies.

11.5.3 Scatter-gather and the IOMMU

In a system with virtual memory, a process's 12 KiB buffer is contiguous in virtual addresses but may be scattered over three separate pages in physical memory. A simple DMA controller can write only to a single contiguous region. In that case the kernel must either bring the data into a contiguous buffer and copy it afterwards, or perform three separate transfers.

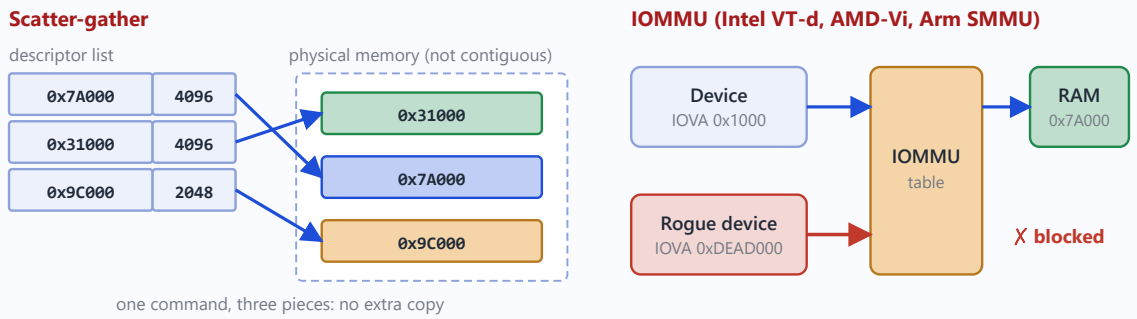


Figure 11.16: On the left, scatter-gather: the descriptor list gives three pieces (0x7A000/4096, 0x31000/4096, 0x9C000/2048) in a single command. On the right, the IOMMU: the address the device sees (IOVA) is translated to a physical address through a table; an access to an address that is not in the table is blocked.

Definition 11.14

In **scatter-gather** DMA, the device is given a list of (address, length) pairs; the device writes to or reads from the noncontiguous pages in a single operation. The **IOMMU** is a unit that translates the addresses a device uses (IOVAs) to physical addresses through a page table.

Scatter-gather underlies the `readv` / `writev` calls on Linux, fragmented network packets, and the PRP and SGL lists in NVMe. No extra copy is needed. The IOMMU has three jobs. The first is protection: a device can access only the pages mapped for it. This blocks DMA attacks in which an externally attached device such as one on Thunderbolt tries to read all of memory. The second is virtualization: a device can be assigned directly to a virtual machine (VFIO); the guest operating system supplies its own “physical” addresses, and the IOMMU translates them to real addresses. The third is that old devices that can address only 32 bits can use memory above 4 GiB. The unit is called VT-d on Intel, AMD-Vi on AMD, and SMMU on ARM.

11.5.4 The printer example with DMA

The third version of the printer example uses DMA:

```
/* system call */
copy_from_user(buffer, p, count);
set_up_DMA_controller();
scheduler();     /* block process */

/* interrupt service routine */
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

The system call copies the data into the kernel buffer, programs the DMA controller (buffer address, count = 3, the printer port as destination), and blocks the process. The DMA controller passes the characters to the printer without the CPU. During this time no interrupt reaches the CPU. When all characters have been transferred, DMA raises a single interrupt. The interrupt service routine does just two things: it acknowledges the interrupt and wakes the user process. There is one interrupt per buffer, not per character.

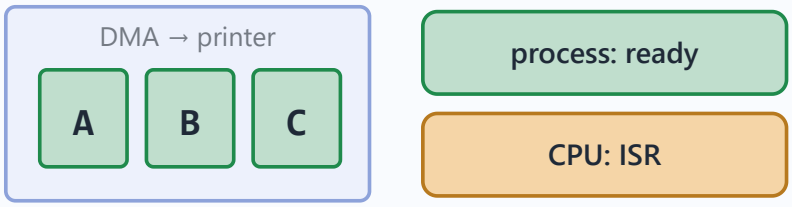


Figure 11.17: The end of printing with DMA: three characters have been transferred, and a single interrupt woke the process.

DMA does not always pay off. A DMA controller is usually slower than the CPU. If the device is very slow, or the CPU has nothing else to do, interrupt-driven I/O or PIO can be simpler and sufficient. Tanenbaum stresses this point in particular: the gain from DMA depends on the CPU being able to do useful work in the time it frees.

11.5.5 The cost of the three methods

Let us compare the three methods in numbers on the same job.

Example 11.4

A printer prints 1000 characters per second; 4096 characters are to be printed. Interrupt handling costs 2 µs and DMA setup 5 µs. How much time does the CPU spend on this job with each method?

1. **Polling.** The CPU waits for every character. At 1 ms per character, the CPU is busy for 4096 × 1 ms = 4.096 s. In programmed I/O, the time the CPU spends depends on the device's speed.
2. **Interrupt-driven.** One interrupt arrives per character: 4096 × 2 µs = 8192 µs ≈ 8.19 ms. This is 0.2% of the total duration of the job (4.096 s). The waiting disappears; the price is the number of interrupts.
3. **DMA.** 5 µs of setup and 1 interrupt × 2 µs: a total of ≈ 7 µs. Compared with the interrupt-driven method, 8192 / 7 ≈ 1170 times less CPU time is spent.
4. **Interpretation.** Between polling and DMA there is a gap of about six orders of magnitude (4.096 s / 7 µs ≈ 585,000 times), and between interrupt-driven I/O and DMA a gap of more than three (Figure 11.18). DMA still steals cycles from the bus, and its setup cost does not pay for itself in small transfers.

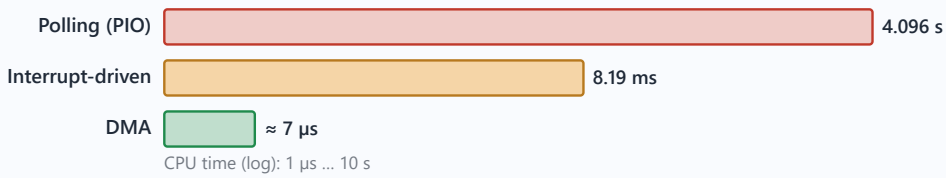


Figure 11.18: CPU time of the three methods for printing 4096 characters, on a logarithmic axis: polling 4.096 s, interrupt-driven 8.19 ms, DMA ≈ 7 µs.

Example 11.5

512 KiB of data is read from a disk controller. In the interrupt-driven method, one interrupt arrives for every 512-byte sector, and the handler spends 5 µs copying the sector into memory. With DMA, the setup takes 8 µs and the single interrupt at the end of the transfer 5 µs. What are the CPU loads?

1. **Number of sectors.** 512 KiB = 524,288 B; 524,288 / 512 = 1024 sectors.
2. **Interrupt-driven.** 1024 × 5 µs = 5120 µs ≈ 5.1 ms.
3. **DMA.** 8 + 5 = 13 µs. The CPU load falls by a factor of 5120 / 13 ≈ 394.
4. **Edge case.** If a single 512-byte sector is read on the same system, the interrupt-driven method spends 5 µs and DMA 13 µs. The fixed setup cost wipes out the gain on a small transfer.

These two examples make the rule clear. As the transfer grows, the fixed cost of DMA becomes negligible and the CPU load becomes independent of transfer length. As the transfer shrinks, the device slows down, or the CPU sits idle, the gain from DMA disappears.

11.6 I/O software

11.6.1 Goals of I/O software

The sheer variety of hardware makes it necessary for the software to be orderly. Six goals guide the design of I/O software.

- **Device independence.** A program should be able to work without knowing which device it is working with. The command `sort < input` works the same way whether the file `input` sits on an SSD or on a USB stick.
- **Uniform naming.** The name of a file or device should not depend on the type of device. `/dev/nvme0n1` and `/mnt/usb/report.txt` are parts of the same tree; different devices are attached to a single tree by mounting.
- **Error handling.** Errors should be solved as close to the hardware as possible. The controller corrects a bit error with the ECC, and the driver retries a failed operation. The upper layer learns about the error only when necessary.
- **Synchronous and asynchronous transfer.** At the hardware level most transfers are asynchronous: the CPU starts the job and waits for the interrupt. Offering the user a blocking interface makes programming easier. The operating system bridges the two.
- **Buffering.** A packet arriving from the network must be put somewhere before it is known where it is going. Real-time devices such as audio also depend on buffers.
- **Shared and dedicated devices.** A disk can serve many users at the same time. A printer or a scanner, on the other hand, is dedicated to a single user at a given moment. Dedicated devices create a deadlock risk: two processes can wait for devices held by each other.

11.6.2 I/O software layers

The way to reach these goals is a layered structure. Each layer has a well-defined task and an interface that it offers to the layer above.

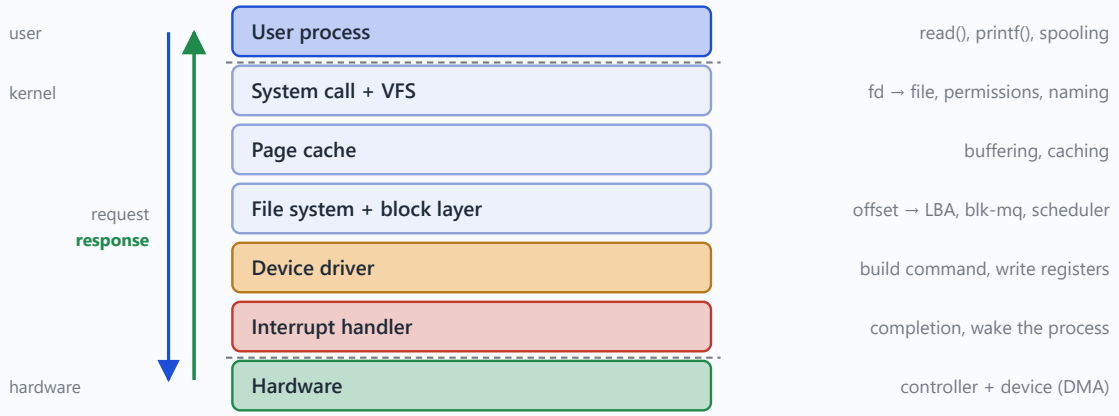


Figure 11.19: I/O software layers: user process, system call and VFS, page cache, file system and block layer, device driver, interrupt handler, and hardware. The request flows from top to bottom and the response from bottom to top.

Tanenbaum counts four layers: user-level I/O software, device-independent software, device drivers, and interrupt handlers. On Linux the "device-independent" part is itself divided into several pieces: the VFS (virtual file system), the page cache, the file system, and the block layer (Figure 11.19). The example below shows how a request passes through these layers.

11.6.3 The path of a `read()` call

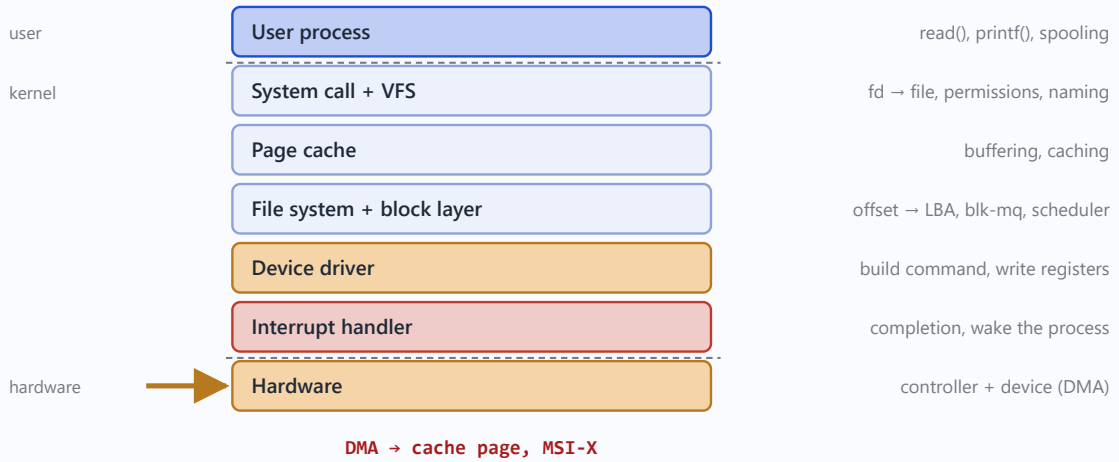


Figure 11.20: The path of a `read()` request through the layers. At this moment the SSD has written the data to the cache page by DMA and raised an MSI-X interrupt.

Example 11.6

A process reads 4 KiB from a file on an NVMe SSD with `read(fd, buf, 4096)`, and the requested page is not in memory. What steps does the request go through?

1. **System call.** The libc wrapper enters the kernel with the `syscall` instruction.
2. **VFS.** The file descriptor is resolved to an open file object, and the open file to an inode. The access permission was already checked when the file was opened.
3. **Page cache.** The requested page is not in the cache. A cache page is allocated and the read is started.
4. **File system and block layer.** The file system translates the file offset into a disk block (LBA, logical block address). The block layer builds a request (bio) and hands it to the hardware queue.
5. **Driver.** The NVMe driver writes the read command into the submission queue and notifies the device by writing to the doorbell register. The process blocks; on Linux its state appears as D (uninterruptible sleep).
6. **Device.** The SSD writes the data to the cache page by DMA, adds an entry to the completion queue, and raises an MSI-X interrupt.
7. **Interrupt handler.** The completion entry is processed, the page is marked “up to date”, and the waiting process becomes ready.
8. **Return.** When the process runs again, the VFS copies the data from the cache into the user buffer and the call returns 4096.

If the same data is read a second time, the page is found in the cache at the third step. The device is never visited; the operation finishes within microseconds. This difference also explains why the operating system devotes free memory to caching.

11.6.4 Device drivers

Definition 11.15

A **device driver** is a kernel component that encapsulates the code specific to a particular device or family of devices. Only the driver knows the device's registers, commands, and quirks.

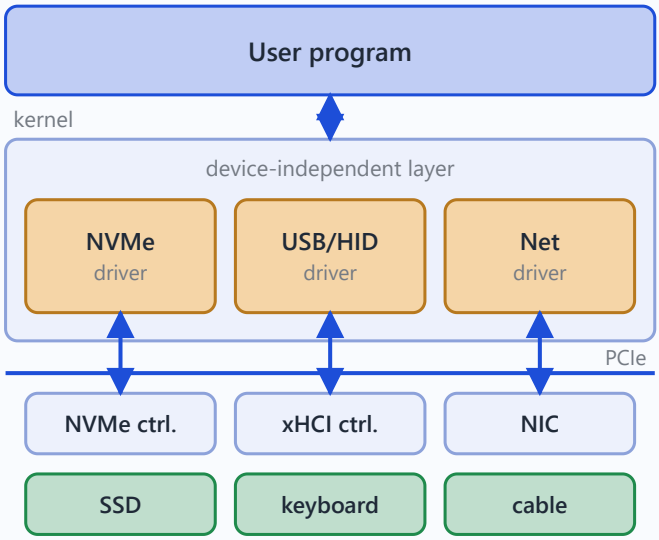


Figure 11.21: Device drivers run in the kernel and talk to controllers over the bus.

A driver follows a similar sequence of work for every request from the layer above. First it validates the parameters. Then it translates the request into a command the device understands; for a disk, this means an LBA and a queue entry. If necessary it starts the device, writes to registers, and waits for the interrupt, that is, it blocks. When the job finishes, it checks for errors and returns the result to the upper layer.

Writing a driver has two difficulties. First, a driver must be **reentrant**: while it is handling one request, a new request or an interrupt may arrive. Second, a hot-pluggable device can vanish while it is in use; the driver must handle this without crashing.

On Linux most drivers are inside the kernel source tree; some come separately from the vendor. Drivers are usually loadable modules: `modprobe` loads a module, and `lsmod` lists the loaded modules. But a faulty driver running in the kernel can crash the whole system. To reduce this risk, there are designs that move drivers into user space. In MINIX 3 all drivers are in user space. FUSE, VFIO/UIO, DPDK, and SPDK on Linux and UMDF on Windows are used for this purpose.

11.6.5 Device-independent I/O software

The layer above the driver takes on the jobs that are common to all drivers. Foremost among them is a uniform interface. Every device class offers the same functions, and the kernel calls the driver through a table of function pointers:

```
struct file_operations nvme_fops = {
  .open           = nvme_open,
  .release        = nvme_release,
  .read_iter      = nvme_read,
  .unlocked_ioctl = nvme_ioctl,
  .poll           = nvme_poll,
};
```

This table is the C counterpart of object-oriented programming. When a `read` is requested on a file, the VFS does not ask which device it is; it calls the `read_iter` pointer in the table attached to the file. For NVMe this is `nvme_read`; for a serial port it is a different function. Adding a new device means writing a new table without touching the rest of the kernel.

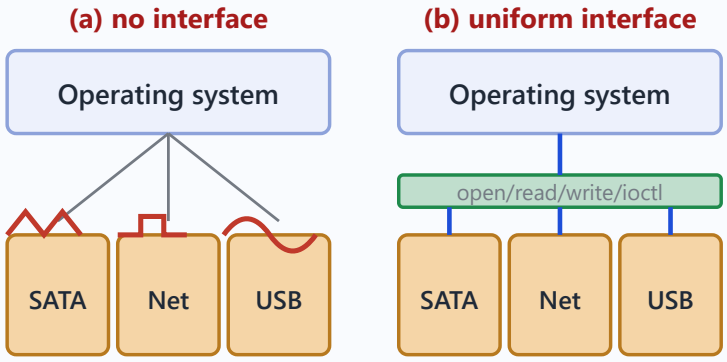


Figure 11.22: (a) Without a standard interface, the kernel calls each driver differently. (b) With a standard interface, all drivers offer the same functions.

The same layer does other jobs as well. **Naming**: a name such as `/dev/nvme0n1` is translated into a pair of a major number (driver) and a minor number (unit). **Protection**: the permissions of the device file decide who can access the device. It also presents a device-independent block size, manages buffering and error reporting, and handles the allocation and release of dedicated devices.

11.6.6 Buffering

Definition 11.16

A **buffer** is a memory area that temporarily holds data being transferred between two devices or between a device and an application. Using such an area is called **buffering**.

Silberschatz lists three reasons for buffering. The first is the **speed mismatch** between producer and consumer: a slow stream from a modem is not written to disk one piece at a time but accumulated in a buffer. The second is the **transfer-size mismatch**: small packets on the network are reassembled

on the receiving side. The third is **copy semantics**: after `write` returns, the application must be able to modify its own buffer; the data to be written to disk must already have been copied in the kernel as it was at that moment.

Consider a process that reads characters from a modem or a serial port. There are five different levels of buffering (Figure 11.23).

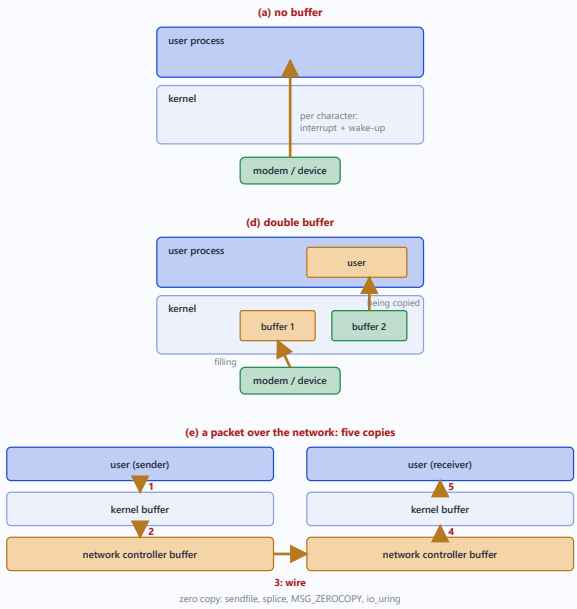


Figure 11.23: Buffering options. At the top the unbuffered arrangement, in the middle the double buffer, at the bottom the five copies of a packet over the network.

1. **Unbuffered.** An interrupt arrives for every character and the process is woken. There are many context switches; it is inefficient.
2. **User buffer.** The interrupt handler writes the characters directly into a buffer in the process; the process is woken only when the buffer fills. But if the buffer's page has been swapped out to disk in the meantime, where does the handler write? The page must be locked in memory. If many processes lock pages, the available memory shrinks.
3. **Kernel buffer.** Characters first accumulate in a buffer in the kernel; when the buffer fills, they are copied to the user in one go. But where does a character that arrives during the copy go?
4. **Double buffer.** The kernel has two buffers: one fills while the other is copied to the user, and then the roles swap. The general form of this arrangement is the circular buffer.
5. **Copies on the network.** A packet can be copied five times on its way across the network: from the sender's user space to the kernel, from the kernel to the network card, over the wire to the receiver's network card, from there to the kernel, and finally to the user. Every copy costs memory bandwidth and latency. That is why **zero-copy** techniques such as `sendfile`, `splice`, `MSG_ZEROCOPY`, and io_uring were developed.

Example 11.7

A block arrives from a device in T = 10 ms, the process takes C = 6 ms to process a block, and copying from the buffer to user space takes M = 2 ms. What is the time per block in steady state?

1. **Unbuffered.** Reading and processing happen one after the other: T + C = 16 ms.
2. **Single buffer.** While the next block arrives in the kernel buffer, the previous block is processed. But during the copy the buffer is busy and the new block cannot start: max(T, C) + M = 10 + 2 = 12 ms.
3. **Double buffer.** One fills while the other is copied; the copy overlaps too. Since C + M = 8 ≤ T, the bottleneck is the device: max(T, C) = 10 ms.
4. **Interpretation.** The limit is now the speed of the device (T > C). Adding more buffers does not raise the average rate; it only absorbs sudden bursts.

11.6.7 The circular buffer

Definition 11.17

A **circular buffer** (also *ring buffer*) uses an array of N slots as if its ends were joined. The producer writes to the slot at the *tail* index, and the consumer reads from the slot at the *head* index; both indices wrap around to 0 when they reach the end (mod N). If head = tail, the buffer is empty; if (tail + 1) mod N = head, it is full.

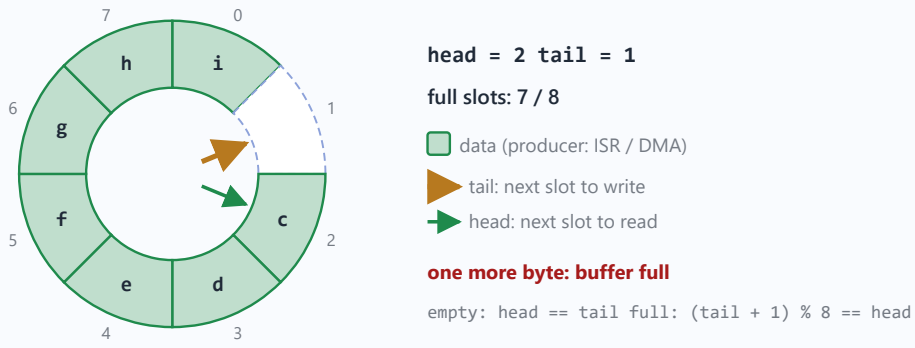


Figure 11.24: An eight-slot circular buffer when full: head = 2, tail = 1. Seven slots hold c, d, e, f, g, h, i; one slot always stays empty.

Example 11.8

A buffer with N = 8 slots is empty (head = tail = 0). The interrupt handler writes 3 bytes (a, b, c), the process reads 2 bytes, and then the producer writes 5 bytes (d … h) and one more byte (i). How do the indices change?

1. **Empty.** Head and tail point to the same slot.
2. **Three writes.** a, b, c go to slots 0, 1, 2; tail = 3. The consumer has not read yet.
3. **Two reads.** a and b are read; head = 2. Slots 0 and 1 can be reused.
4. **Five writes.** d … h go to slots 3–7. When the tail reaches the end, it wraps to the start: 8 mod 8 = 0.
5. **The last write.** i is written to slot 0, and tail = 1. Now (1 + 1) mod 8 = 2 = head: the buffer is full. The buffer holds 3 + 5 + 1 − 2 = 7 bytes, that is, N − 1.

When the buffer is full, the producer can do one of two things: wait for the consumer to make room, or drop the data. On a network card the second happens and the packet is lost. The reason one slot is always left empty is simple. If all slots could be filled, head = tail would hold for a full buffer too, and empty could not be told from full. The price is that a buffer of N slots holds at most N − 1 bytes.

Circular buffers are everywhere in I/O: keyboard and serial port drivers, the receive and transmit rings of network cards, the submission and completion queues of NVMe, io_uring. With a single producer and a single consumer, no lock is needed. The producer changes only the tail and the consumer only the head; to maintain the correct order, only memory barriers are required.

11.6.8 Code trace: the circular buffer

The two functions below implement the circular buffer in C. `put` runs in the interrupt handler and `get` in the process.

```
int put(char c) {          /* producer */
  int s = (tail + 1) % N;
  if (s == head) return -1; /* full */
  buf[tail] = c;
  tail = s;
  return 0;
}
int get(char *c) {         /* consumer */
  if (head == tail) return -1; /* empty */
  *c = buf[head];
  head = (head + 1) % N;
  return 0;
}
```

`put` first computes the next value of the tail in `s`. If `s` equals head, writing would make the buffer look “empty”; so the buffer is considered full and `-1` is returned. Otherwise the byte is written to `buf[tail]` and the tail advances. The order matters here: first the data is written, then the index is updated. Otherwise the consumer could read a slot that has not been written yet. `get` checks whether head equals tail; if so, the buffer is empty. Otherwise `buf[head]` is read and the head advances. Both functions run in O(1) time. The space of the buffer is N slots and never changes.

Call	Detail	`head`	`tail`	Returns
—	buffer empty	0	0	
`put('a')`	buf[0] = 'a'	0	1	0
`put('b')`	buf[1] = 'b'	0	2	0
`put('c')`	buf[2] = 'c'	0	3	0
`get`	head ≠ tail; 'a' is read	1	3	0
`put('d')`	buf[3] = 'd'	1	4	0
`put('e')`	buf[4] = 'e'; the tail wraps around	1	0	0
`put('f')`	s = (0 + 1) mod 5 = 1 = head: full	1	0	−1
`get`	'b' is read	2	0	0

Table 11.6: `put` and `get` calls for N = 5. Each row shows the state after the call.

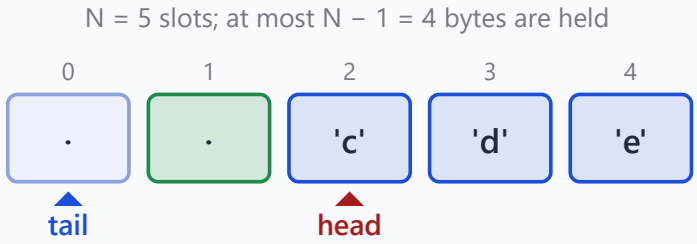


Figure 11.25: The end of the trace: slots 2, 3, 4 hold 'c', 'd', 'e'; head = 2, tail = 0.

When 'f' is rejected, the buffer holds four bytes ('b', 'c', 'd', 'e'): N − 1 = 4. After one read, room opens up and the next `put` succeeds.

Common mistake

Writing the full condition as `tail == head`. This condition is true for an empty buffer as well; when the buffer fills, the producer would have overwritten all the data on top of an “empty” buffer and the consumer could read nothing. If you do not want to leave a slot empty, you must keep a separate counter; but then both sides modify the same variable and lock-free

operation breaks.

11.6.9 Caching, spooling, and error handling

A cache and a buffer are different things. A buffer temporarily holds the only copy of the data. A cache, on the other hand, is a fast-access copy of data that lives somewhere else. The Linux page cache combines the two: data to be written to disk is buffered there, and data read from disk is cached there. Idle RAM goes to the cache; the `buff/cache` column in the output of `free -h` shows this.

Definition 11.18

Spooling means queuing the jobs destined for a dedicated device on disk, while a single background process sends them to the device one after another.

The printer is the classic example of spooling. The user puts the file to be printed in the spool directory; a single background process (CUPS on Linux, directory `/var/spool/cups`) prints the files in order. No user process locks the printer, so there is no deadlock over the printer either.

Error handling separates two kinds of errors. For programming errors, such as an invalid file descriptor, the call returns an error code (`EBADF`, `EINVAL`). For a real I/O error, the driver retries the operation; if the error is permanent, the call returns `EIO` and the event is written to the kernel log (`dmesg`).

Allocation and protection. A dedicated device is reserved with `open`. If the device is busy, the request is queued or the call returns `EBUSY`. All I/O instructions are privileged; a user process can perform I/O only through system calls.

At the very top sits the I/O software in user space. Library routines such as `printf` and `scanf` do formatting and keep their own buffers. stdio writes to a terminal line by line and to a file block by block. A `write` system call is made when the buffer is full or the line ends. This way each `printf` does not turn into a system call.

11.7 I/O interfaces

11.7.1 Blocking, nonblocking, and asynchronous I/O

When an application requests I/O, when the call returns determines the whole structure of the program. There are three models.

Definition 11.19

A **blocking I/O** call does not return until the requested data arrives; the process sleeps in the meantime. A **nonblocking I/O** call returns immediately: it gives whatever data is available at that moment, or returns the `EAGAIN` error if there is none. An **asynchronous I/O** call starts the job and returns immediately; the job completes in the background and the process is notified afterward.

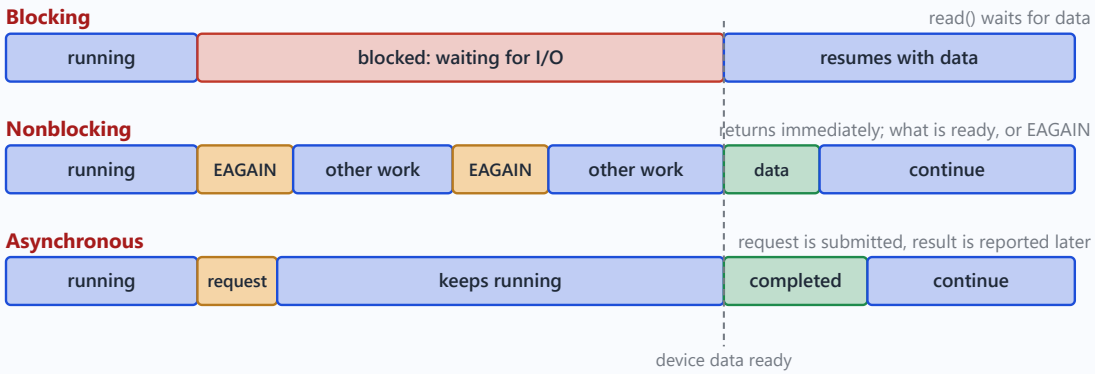


Figure 11.26: Timelines of the three models. In a blocking call the process sleeps until data arrives. In a nonblocking call the process does other work whenever it gets `EAGAIN` and tries again. In an asynchronous call the process keeps running and is notified when the job is done.

The blocking model is the easiest to program, and it is the model most programs use. But if a process is to serve thousands of network connections at once, starting a thread for each connection is expensive. So that a single thread can watch many descriptors, the kernel offers interfaces that answer the question "which descriptor is ready?" (Table 11.7).

Interface	Model	Note
`select` / `poll`	readiness notification	the whole descriptor list is scanned on every call: O(n)
`epoll` (Linux), `kqueue` (BSD)	readiness notification	interest list in the kernel; only the ready ones are returned
io_uring, Windows IOCP	completion notification	true asynchronous I/O; disk included

Table 11.7: Multiple-descriptor interfaces.

11.7.2 The event loop: epoll

```
int ep = epoll_create1(0);
for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
    epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
for (;;) {
  int n = epoll_wait(ep, evs, 64, -1);
  for (int i = 0; i < n; i++) {
    int fd = evs[i].data.fd;
    ssize_t r = read(fd, buf, sizeof buf);
    if (r < 0 && errno == EAGAIN) continue;
    handle(fd, buf, r);
  }
}
```

The code is the event loop of a typical server. `epoll_create1` creates an epoll object in the kernel. This object keeps two lists: the **interest list** of the descriptors being watched and the **ready list** of those that are ready at the moment. Then the sockets are added to the interest list in nonblocking mode (`O_NONBLOCK`). This is done once; unlike `select` , the list does not have to be passed again on every iteration.

In the endless loop, `epoll_wait` blocks the process while no socket is ready; the CPU does not spin. Suppose a packet arrives for socket 7. The network card raises an interrupt, the network stack processes the packet and adds fd 7 to the ready list; the process wakes up and `n = 1` is returned. The inner loop handles only the ready descriptor: 512 bytes are read and `handle` is called. The cost is proportional to the number of ready descriptors, not to the total number of descriptors. If the next `read` returns `EAGAIN` , no data is left to read and the loop goes back to `epoll_wait` .

Figure 11.27: After the data is handled: fd 5, 6, 7 in the interest list; the ready list is empty.

With this arrangement, a single thread can manage tens of thousands of connections. nginx, Node.js, and Redis work this way. epoll has a limit too. epoll only says “you may read”; the work is still done by `read` , and the copying comes out of the process's time. Disk files always look “ready”, because the kernel never refuses a disk read with `EAGAIN` ; it completes it by blocking. So true asynchrony for disks comes with io_uring.

11.7.3 io_uring: shared rings

Definition 11.20

io_uring (Linux 5.1, 2019) is an asynchronous I/O interface in which the application and the kernel share two circular buffers in shared memory. The application writes request entries (SQEs) to the **submission ring** (SQ); the kernel adds the results of completed jobs to the **completion ring** (CQ) as entries (CQEs).

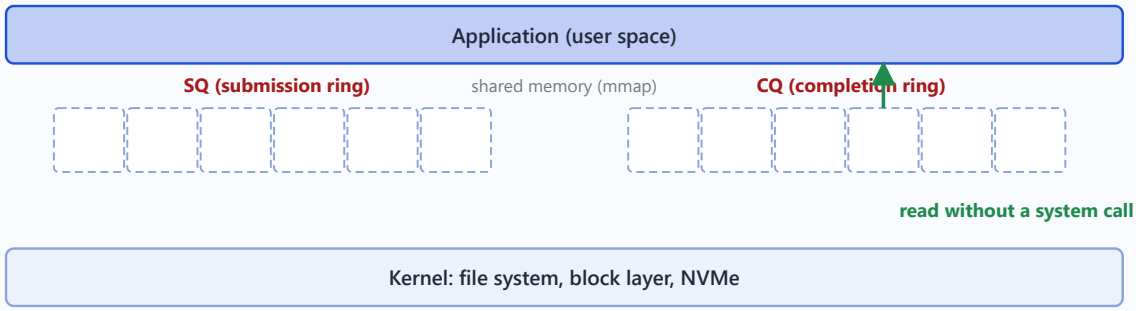


Figure 11.28: io_uring: the submission (SQ) and completion (CQ) rings shared between the application and the kernel. The application reads completions without making a system call.

1. **Rings.** The application and the kernel see the two rings in shared memory through `mmap`. Requests and results are not copied.
2. **Submission entries.** The application writes three SQEs: two reads and a write. Each entry carries the opcode, the file descriptor, the buffer address, and the offset.
3. **Batch submission.** A single `io_uring_enter` call submits all three requests at once. In SQPOLL mode a kernel thread polls the ring continuously and no system call is needed at all.
4. **Completion.** As devices finish the jobs, the kernel adds CQEs to the ring. The result field of each CQE carries the number of bytes transferred or a negative error code (−errno).
5. **Reaping.** The application reads the CQEs directly from memory and advances the head of the ring. Thanks to batch submission and batch reaping, context switches drop sharply.

io_uring is a **completion model**: the kernel says “your job is done”, like IOCP on Windows. epoll, on the other hand, is a **readiness model**: it says “you may read now”. io_uring supports hundreds of operation types, including network, file, `accept`, and timers. Because it is a complex interface, many vulnerabilities have been found in it; for this reason it is disabled in places such as Android and some container environments.

Common mistake

Treating nonblocking I/O and asynchronous I/O as the same thing. A nonblocking call does not do the job; it does only as much as can be done at that moment and leaves the rest to the process, which has to try again. An asynchronous call hands the whole job to the kernel and receives the result later. The price is more complicated programming: buffers must stay valid until the job is done, and the error path of every request has to be tracked separately.

11.8 Clocks and other devices

11.8.1 Clock hardware

Clocks are neither block nor character devices, but they are among the devices the operating system uses most. They drive time slices, the time of day, and all timers.

Definition 11.21

A **programmable clock** is a circuit that decrements a counter on each pulse from a crystal oscillator and raises an interrupt when the counter reaches zero. In **one-shot** mode the clock raises a single interrupt and stops. In **square-wave** (periodic) mode the counter is reloaded from a holding register and interrupts arrive at regular intervals. Each interrupt is called a **tick**.

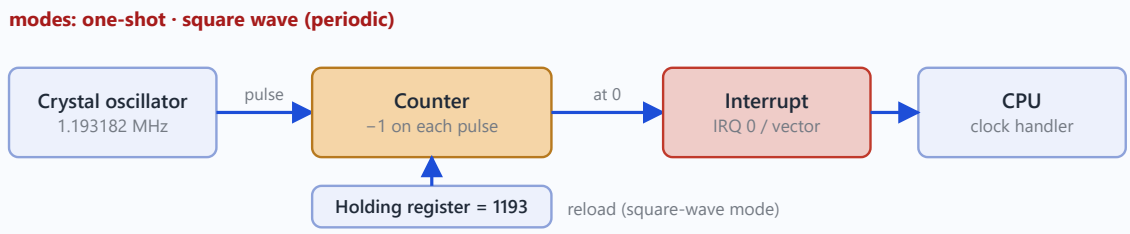


Figure 11.29: The 8254 PIT: a 1.193182 MHz oscillator decrements the counter on each pulse; when the counter reaches 0, IRQ 0 is raised and the counter is reloaded from the holding register (1193).

Example 11.9

The 8254 PIT counts 1,193,182 pulses per second. What is loaded into the holding register for a 1 kHz and a 100 Hz tick? What are the actual frequencies?

1. **1 kHz.** 1,193,182 / 1000 = 1193.18. The counter must be an integer: 1193. The actual frequency is 1,193,182 / 1193 ≈ 1000.15 Hz.

2. **100 Hz.** 1,193,182 / 100 = 11,931.82 → 11,932. The actual frequency is 1,193,182 / 11,932 ≈ 99.998 Hz.
3. **Result.** Because of rounding, the clock drifts slightly from the desired frequency. This drift, which accumulates over a long time, makes it necessary to correct the time of day now and then against a real time source.

The 8254 survives today only for compatibility. Modern x86 processors use the TSC (timestamp counter), the HPET, and the local APIC timer on each core. In the TSC-deadline mode of the APIC timer, the operating system tells it directly "raise an interrupt when the TSC reaches this value". On ARM processors the generic timer takes over this job. The battery-backed real-time clock (RTC) keeps the date while the computer is off; at boot the time of day is read from it.

The tasks of clock software are: keeping the time of day, calling the scheduler when a process's time slice expires, accounting for CPU usage, running the alarms and timers that processes request, feeding watchdog timers, and sampling for profiling.

Common mistake

Keeping the time of day in a 32-bit seconds counter. A signed 32-bit `time_t` can count at most $2^{31} - 1$ seconds from 1 January 1970. This limit is reached at 03:14:07 UTC on 19 January 2038, and the counter overflows to negative. The time of day must be kept in a 64-bit counter. Linux has moved to a 64-bit `time_t`, but the problem persists in old embedded systems.

11.8.2 Ticks and the tickless kernel

The classic kernel uses the clock in periodic mode. The timer raises an interrupt every 1/HZ seconds. On Linux, HZ is chosen at configuration time: 100, 250, or 1000; the tick interval is 10, 4, and 1 ms respectively. On every tick the handler advances time, decrements the remaining time slice of the running process, and runs timers that have expired.

Figure 11.30: (a) A periodic tick (HZ = 250, 4 ms): the CPU wakes on every tick even when idle; in the empty interval in the figure there are six needless wake-ups. (b) A tickless kernel: there is no tick while idle, and the clock is set to the next event (30 ms) with a one-shot.

The problem with a periodic tick is that the CPU wakes even when idle. Every wake-up pulls the CPU out of a deep sleep state and spends energy. The **tickless kernel** (NO_HZ_IDLE on Linux, 2007) stops the tick while the CPU is idle and sets the clock with a one-shot for the time of the next event. The `nohz_full` option goes one step further. On a core where only a single process runs, it removes the tick even while the process is running. This matters for high-performance computing and real-time workloads. Tickless operation saves battery on laptops and CPU time in virtual machines. For short waits at the microsecond level, soft timers and polling are used together. Linux's high-resolution timers (hrtimer) have nanosecond resolution and are kept ordered in a red-black tree.

11.8.3 The timer list

Many processes can set an alarm saying “notify me after this many ticks”. How can the clock software keep track of these alarms without looking at all of them on every tick? Tanenbaum's answer is the delta list.

Definition 11.22

A **delta list** is a linked list that keeps pending timers ordered by expiry time. Each node stores not the absolute time but how many ticks after the previous node it expires (the delta). The delta of the first node is relative to the current time.

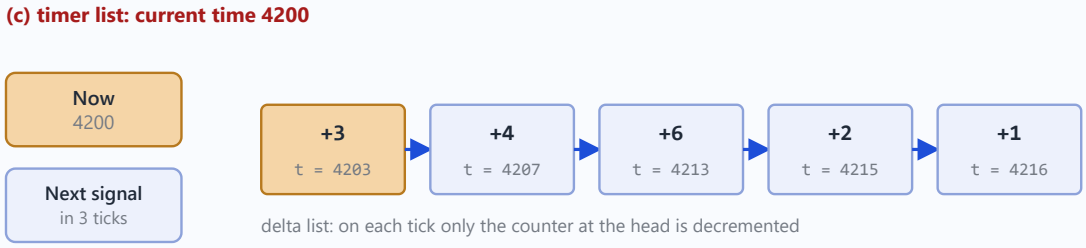


Figure 11.31: The current time is 4200; pending signals are at 4203, 4207, 4213, 4215, and 4216. The list stores only the differences: 3, 4, 6, 2, 1.

The advantage of this arrangement is in the tick handler: on each tick only the counter at the head is decremented. When the head delta reaches 0, the event fires and the node is removed from the list. No matter how many timers are in the list, the work per tick is O(1). The cost is in insertion: finding the place of a new timer means walking the list, which takes O(n).

11.8.4 Code trace: `add` and `tick`

```
void add(Timer *z, int t) {  /* t: ticks */
  Timer **p = &head;
  while (*p && (*p)->delta <= t) {
    t -= (*p)->delta;
    p = &(*p)->next;
  }
  z->delta = t; z->next = *p;
  if (*p) (*p)->delta -= t;
  *p = z;
}
void tick(void) {
  if (head) head->delta--;
  while (head && head->delta == 0)
    head = run(head);   /* remove */
}
```

`add` inserts a new timer into the list for t ticks from now. `p` is a pointer to a pointer: first it points to the list head (`&head`), then at each step to the `next` field of the previous node. This way insertion at the head and insertion in the middle need not be written separately. The loop subtracts the delta of each node it passes from t; t now means “how many ticks after that node”. The loop stops when it reaches a node whose delta is larger than t. The new node goes in there, and the delta of the node behind it is reduced by t; so the absolute time of the node behind does not change. `tick` , for its part, decrements only the delta at the head, and runs and removes, in order, every node whose delta is 0. Of two timers that expire at the same moment, the second has delta 0; that is why the `while` loop is there.

Example 11.10

The list is A 3, B 4, C 6, D 2, E 1 (absolute times 4203, 4207, 4213, 4215, 4216). At time 4200, timer X is inserted for 10 ticks later (4210): `add(X, 10)`. Then three ticks pass.

1. **A.** A's delta 3 ≤ 10: X comes after it. t = 10 − 3 = 7; `*p` moves to B.
2. **B.** B's delta 4 ≤ 7: X comes after that one too. t = 7 − 4 = 3; `*p` moves to C.
3. **C.** C's delta 6 > 3: X must go in before C; the loop ends.
4. **Insertion.** X goes in with delta = 3; C's delta becomes 6 − 3 = 3. The list is A 3, B 4, X 3, C 3, D 2, E 1. C's absolute time is 4203 + 4 + 3 + 3 = 4213, that is, unchanged. Only one node was updated.
5. **Ticks.** On the first two ticks only A's delta decreases: 2, then 1. On the third tick (4203) the delta becomes 0; A is run and removed from the list. The list is B 4, X 3, C 3, D 2, E 1.

Figure 11.32: On the left, after X is inserted: A 3, B 4, X 3, C 3, D 2, E 1. On the right, after A leaves at 4203: the list starts at B.

Example 11.11

The current tick is 8000; alarms are at 8002, 8006, 8006, 8011, and 8020. What is the delta list? Which alarms have fired after 8 ticks?

1. **Deltas.** 8002 − 8000 = 2, 8006 − 8002 = 4, 8006 − 8006 = 0, 8011 − 8006 = 5, 8020 − 8011 = 9. The list is 2, 4, 0, 5, 9.
2. **First alarm.** After 2 ticks the delta at the head becomes 0: the alarm at 8002 fires.
3. **Simultaneous alarms.** After 4 more ticks (8006) the delta at the head becomes 0. The loop removes this node; since the delta of the node behind it is also 0, it removes that one too. The two alarms at 8006 fire together.

4. **Remaining.** By 8008 two more ticks have passed: the delta at the head is 5 − 2 = 3. The list is 3, 9 (8011 and 8020).

11.8.5 Terminals, input devices, and power management

Terminals. In Unix there is a **line discipline** between the terminal driver (tty) and the application. In canonical mode the line discipline edits the line: backspace and line kill are handled there, and the line is delivered to the application only when Enter is pressed. In raw mode every key goes to the application immediately; text editors use this mode (`stty -icanon`). Terminal emulators and `ssh` sessions use a pseudo-terminal (pty). Screen control is done with ANSI escape sequences that move the cursor or change the color.

Keyboards and input devices. A keyboard sends scan codes, not characters: pressing and releasing each key are separate codes. The driver translates them into characters using a key map. This is why the Shift key is a separate event. USB keyboards and mice (HID devices) do not raise interrupts; the host polls them 125–1000 times per second. On Linux, input events travel from the `input` subsystem to the `/dev/input/event*` files (evdev), from there to libinput and the Wayland compositor, and finally to the window.

Power management. ACPI defines sleep states for the system: S0 is the working state (S0ix is modern standby), S3 is suspend to RAM, S4 is hibernation to disk, and S5 is off. The processor has two separate settings. C-states determine how deeply it sleeps when idle, and P-states determine which frequency and voltage are used while it runs; on Linux the `schedutil` governor chooses the P-states. Devices also have states between D0 (on) and D3 (off). Runtime power management puts an unused device to sleep while the system is running: APST on NVMe, ASPM on PCIe, Wi-Fi power saving, and screen dimming are examples. The operating system must put devices to sleep and wake them in the right order and decide which devices (keyboard, network, clock) may wake the system. The battery life of a laptop depends largely on how little power it draws during idle time.

11.9 Disks

11.9.1 Magnetic disk geometry

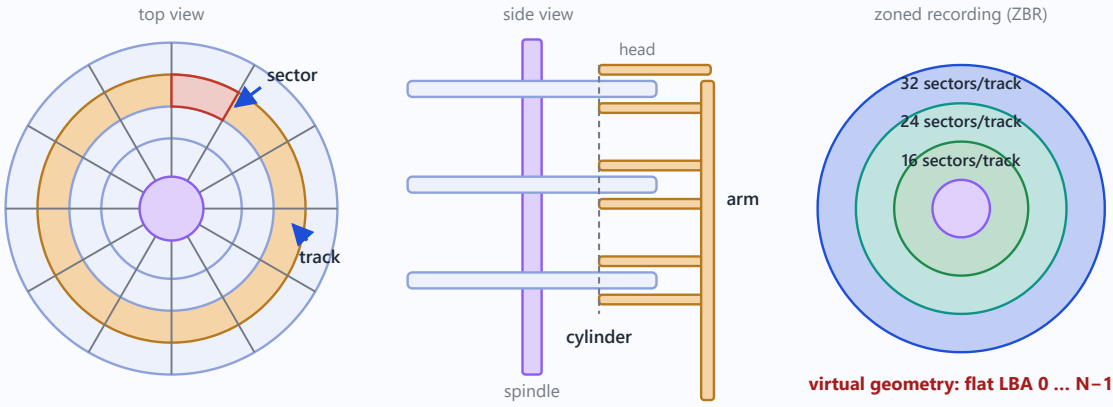


Figure 11.33: A hard disk: platters, tracks, sectors, a cylinder, read-write heads, and the arm. On the right, zoned bit recording (ZBR): the outer zones have more sectors per track; the operating system sees this as a flat array of LBAs.

Definition 11.23

A **hard disk** (HDD) consists of one or more magnetic **platters** spinning on the same spindle. Each surface has a read-write head; the heads move together on a common arm. The concentric rings on a surface are called **tracks**, the pieces of a track **sectors**, and the set formed by the tracks at the same radius on all surfaces a **cylinder**.

Platters spin at 5400–7200 rpm in desktop disks and 10,000–15,000 rpm in server disks. Because the outer tracks are longer, the outer zones have more sectors per track. This is called **zoned bit recording**; the disk transfers faster on the outer tracks.

The operating system no longer sees this geometry today. Disks used to be addressed with cylinder-head-sector (CHS) triples. Today the operating system sees only an **LBA** (logical block address), that is, blocks numbered from 0 to N − 1. The controller inside the disk translates the LBA to the real geometry, spare sectors, and its own cache by itself. The limit of the old CHS interface was 1024 cylinders × 255 heads × 63 sectors × 512 B ≈ 8.4 GB.

11.9.2 Access time

Definition 11.24

The **access time** of a disk request is the sum of three components. **Seek time** is the time for the arm to reach the right cylinder. **Rotational latency** is the wait for the desired sector to come under the head; its average is half a revolution. **Transfer time** is the time for the data to pass under the head.

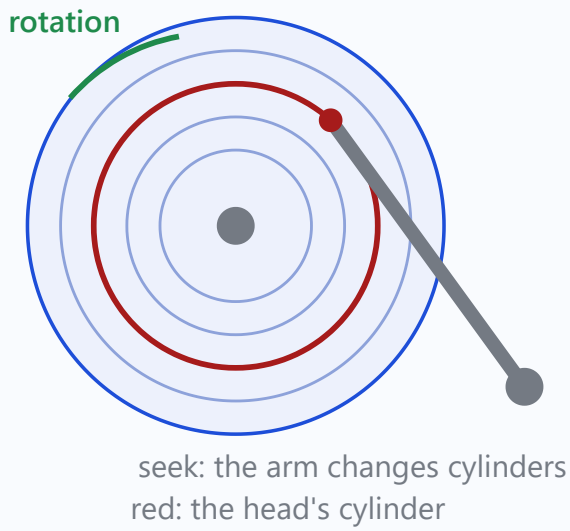


Figure 11.34: A seek is the arm changing cylinders; rotational latency is the platter bringing the desired sector under the head.

Example 11.12

A disk spins at 7200 rpm; the average seek time is 8 ms and the media transfer rate is 200 MB/s. How long does it take to read a random 4 KiB block?

1. **One revolution.** 60,000 ms / 7200 = 8.33 ms.
2. **Rotational latency.** On average the desired sector is half a revolution away: 8.33 / 2 = 4.17 ms.
3. **Transfer.** 4096 B / (200 × 10^6 B/s) = 0.020 ms.
4. **Total.** 8 + 4.17 + 0.02 = 12.19 ms. The transfer time, however small, is one of the three components and must be counted.

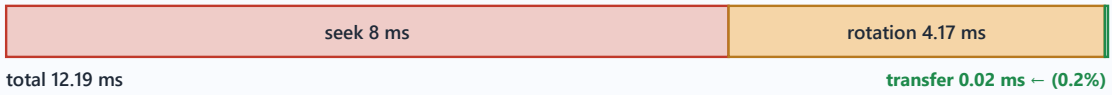


Figure 11.35: In a random 4 KiB read: seek (8 ms), rotational latency (4.17 ms), and transfer (0.02 ms; 0.2% of the total).

The conclusion is clear: the time is spent almost entirely on mechanical movement. In these 12 ms the CPU can execute tens of millions of instructions; the operating system does not wait but runs another process. The same disk gives very different speeds depending on the access pattern:

- **Random 4 KiB.** 1000 / 12.19 ≈ 82 requests per second (IOPS) can be served. That comes to 82 × 4096 B ≈ 0.34 MB/s.
- **Sequential 1 MiB.** The transfer takes 1,048,576 / (200 × 10^6) = 5.24 ms; the total is 8 + 4.17 + 5.24 = 17.41 ms. The rate is 1,048,576 B / 17.41 ms ≈ 60 MB/s.

These calculations make two simplifications: the media rate is taken as constant, and the seek and rotational latency are paid again for each piece. In a real long sequential read, the seek is made only once at the start; the following blocks pass under the head without interruption and the rate approaches the media rate (here 200 MB/s).

The same disk is about 180 times faster on a large sequential read than on small random reads. The effort of file systems to place blocks close together, and the importance of disk-arm scheduling, come from this gap. For comparison: a 4 KiB random read on an NVMe SSD takes 50–100 µs, and with parallel queues more than a million requests per second can be served.

Example 11.13

A server disk spinning at 15,000 rpm has an average seek time of 3.5 ms and a media rate of 250 MB/s (1 MB = 10^6 B). How long does a 64 KiB sequential read take, and how long does reading 16 scattered 4 KiB blocks take?

1. **Revolution and rotation.** 60,000 / 15,000 = 4 ms; the average rotational latency is 2 ms.
2. **64 KiB sequential.** The transfer is 65,536 / (250 × 10^6) s = 0.262 ms; the total is 3.5 + 2 + 0.262 ≈ 5.76 ms.
3. **16 × 4 KiB scattered.** Per block 3.5 + 2 + 0.016 = 5.516 ms; 16 blocks take 16 × 5.516 ≈ 88.3 ms. The same data is read about 15 times slower.
4. **Does a faster medium help?** If the media rate doubled, the transfer share in the scattered read would drop from 16 × 0.016 = 0.26 ms to 0.13 ms: a gain of only 0.15% of the total. In a small random read, seek and rotation dominate the time.

11.9.3 Formatting and cylinder skew

A disk must be formatted at two levels before it can be used. **Low-level formatting** is done at the factory. Tracks and sectors are written onto the surface; each sector gets a preamble, a data field, and an ECC, and spare sectors are set aside. This is why formatted capacity is less than raw capacity. **High-level formatting** writes an empty file system to a partition: a boot block, free-space management structures, and the root directory.

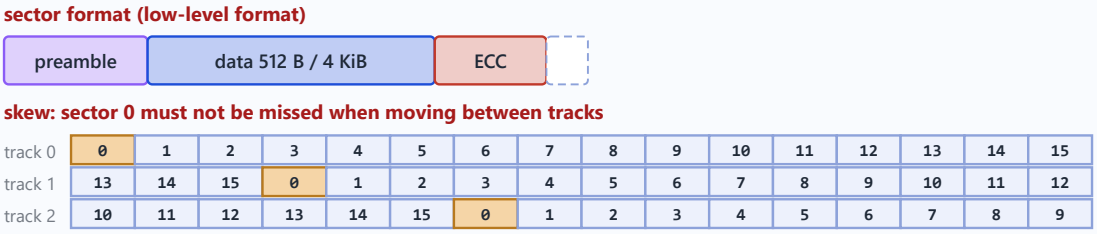


Figure 11.36: Sector format and cylinder skew. Sector 0 of each track is shifted relative to the previous one; sector 0 is not missed while the head changes tracks.

Definition 11.25

Cylinder skew is the shifting of the sector 0 positions of successive tracks relative to one another. While the head finishes one track and moves to the next, the disk keeps rotating; the shift ensures that when the head arrives at the new track, sector 0 is just about to arrive.

Example 11.14

A disk spins at 10,000 rpm and has 300 sectors per track. A track switch takes 800 µs. How many sectors should the skew be?

1. **Revolution.** 60,000 / 10,000 = 6 ms.
2. **Sector time.** 6 ms / 300 = 20 µs.
3. **Skew.** During the track switch 800 / 20 = 40 sectors pass under the head. Sector 0 of the next track must be shifted by 40 sectors.

In the past, **interleaving** was also used for slow controllers: so that the next sector would not pass under the head and be missed while the controller transferred one sector to memory, consecutive sectors were laid out on the track with gaps between them. Today the controller reads the whole track into its own buffer, so interleaving is unnecessary.

11.9.4 Optical discs

On optical discs the data lies along a single spiral track running from the inside out, instead of the concentric tracks of a magnetic disk. The track has pits and lands; each transition between a pit and a land stands for a 1 bit, and no transition for a 0. On pressed discs the pits are stamped into the polycarbonate with a mold, not burned with a laser. On recordable CD-R and DVD-R discs, the laser heats a layer of organic dye and changes its reflectivity.

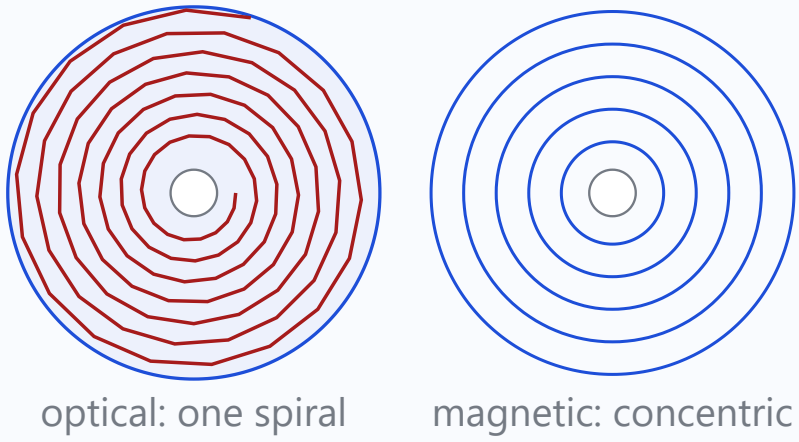


Figure 11.37: On an optical disc the data lies along a single spiral track; on a magnetic disk it lies in concentric tracks.

	CD (1982)	DVD (1996)	Blu-ray (2006)
Capacity	650–700 MB	4.7 / 8.5 / 9.4 / 17 GB	25 / 50 GB; BDXL 100–128 GB
Laser	780 nm (infrared)	650 nm (red)	405 nm (blue-violet)
Smallest pit	0.83 µm	0.40 µm	0.15 µm
Track pitch	1.6 µm	0.74 µm	0.32 µm

Table 11.8: Generations of optical discs.

As in Table 11.8, the capacity growth between generations comes from the shortening of the laser wavelength: a shorter wavelength focuses to a smaller spot, and the pits and track pitch shrink. The coding of the CD-ROM is interesting too. Each byte is converted to a 14-bit channel symbol by EFM (eight-to-fourteen modulation). This is a modulation code, not an error-correcting code; error correction is done with CIRC. Each of 98 frames carries 24 data bytes, forming a sector of 98 × 24 = 2352 B. In Mode 1, 2048 B of this is user data; the rest is synchronization, header, and additional ECC/EDC fields.

In the dual-layer versions of DVD, the upper layer is semitransparent; the laser reads the layer underneath by changing its focus. Optical media are mostly used today for archiving and distribution; they lag behind magnetic and flash media in density.

11.10 Disk-arm scheduling

11.10.1 Goal and criterion

The access-time example showed that most of the time of a disk request goes to seek and rotation. If many requests are waiting at the disk at the same time, the order in which they are served determines how far the arm travels.

Definition 11.26

Disk scheduling aims to reduce arm movement (and therefore seek time) and raise throughput by choosing the order in which pending disk requests are served. For each request the driver keeps the cylinder number and the arrival time. The criterion used to compare algorithms is the head's **total movement** in cylinders.

It must be said up front that this model is a simplification. Today the mapping of LBA to physical location is inside the disk; the operating system does not know the cylinders. The disk also reorders 32 requests by itself according to rotational position using NCQ (native command queuing). The operating system therefore schedules approximately by "LBA proximity": it assumes that blocks with nearby numbers are also near each other on the disk.

Figure 11.38: The common queue: the head is at cylinder 53; the pending requests are 98, 183, 37, 122, 14, 124, 65, 67. Cylinders 0–199.

To compare the algorithms, we will give all of them the same queue. In Silberschatz's classic example the cylinders range from 0 to 199, the head is at 53, and the requests in arrival order are 98, 183, 37, 122, 14, 124, 65, 67 (Figure 11.38). Table 11.9 summarizes the essence of the algorithms in advance.

Algorithm	Idea	Problem
FCFS	Arrival order	Fair, but the arm travels a lot

Algorithm	Idea	Problem
SSTF	Request closest to the head (greedy)	Requests at the edges may starve
SCAN (elevator)	Go to the end in one direction, then reverse	The edges and the middle do not wait equally
C-SCAN	Sweep in one direction, jump from the end to the start	The return movement is wasted
LOOK / C-LOOK	SCAN / C-SCAN, but only as far as the last request	The forms used in practice

Table 11.9: Disk-arm scheduling algorithms.

11.10.2 FCFS

Definition 11.27

FCFS (first-come, first-served) serves requests in the order they arrive.

Example 11.15

What is the total head movement of FCFS on the common queue?

1. 53 → 98: 45 cylinders; total 45.
2. 98 → 183: 85; total 130.
3. 183 → 37: 146; total 276.
4. 37 → 122: 85; total 361.
5. 122 → 14: 108; total 469.
6. 14 → 124: 110; total 579.
7. 124 → 65: 59; total 638.
8. 65 → 67: 2; total **640** cylinders.

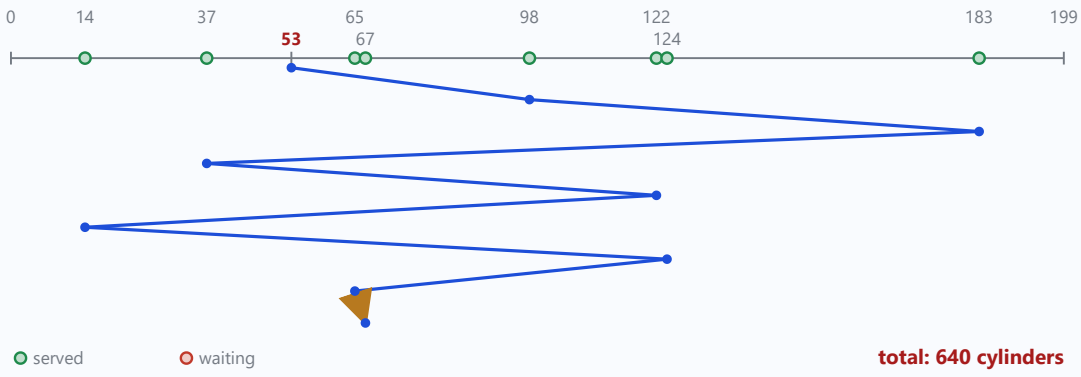


Figure 11.39: The head's path under FCFS: 53, 98, 183, 37, 122, 14, 124, 65, 67. Total 640 cylinders.

FCFS is fair and causes no starvation: every request is certainly served once those before it finish. But back-and-forth movements such as 122 → 14 → 124 are very expensive. The arm goes from one end of the disk to the other and back twice; the two requests to 14 and 124 alone consume 218 cylinders.

11.10.3 SSTF

Definition 11.28

SSTF (shortest seek time first) picks, at each step, the pending request closest to the head.

Example 11.16

What order does SSTF follow on the common queue, and what is the total movement?

1. Head at 53: the candidates are 65 (12 away) and 37 (16 away). 53 → 65: 12; total 12.
2. 65 → 67: 2; total 14.
3. Head at 67: 37 (30 away) is closer than 98 (31 away). 67 → 37: 30; total 44.
4. 37 → 14: 23; total 67.
5. The lower region is done. 14 → 98: 84; total 151.
6. 98 → 122: 24; total 175.
7. 122 → 124: 2; total 177.
8. 124 → 183: 59; total **236**.

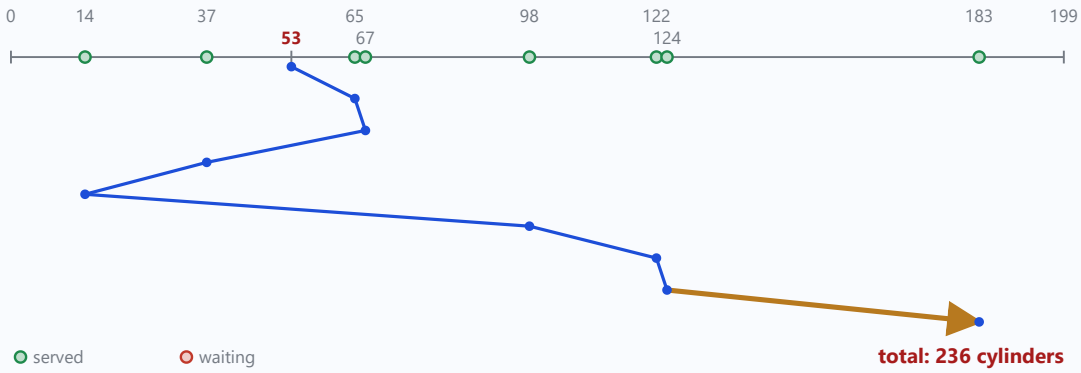


Figure 11.40: The head's path under SSTF: 53, 65, 67, 37, 14, 98, 122, 124, 183. Total 236 cylinders.

236 is about 37% of FCFS's 640. But SSTF is greedy and does not guarantee the best result. If the head had gone down first, the order would have been 53 → 37 → 14 → 65 → 67 → 98 → 122 → 124 → 183. The total comes to 16 + 23 + 51 + 2 + 31 + 24 + 2 + 59 = 208. In SSTF, the short 30-cylinder step at the third move led to an 84-cylinder turnaround afterward.

> **Common mistake**
>
> Thinking SSTF is the best algorithm. SSTF chooses the shortest move at each step, but it does not minimize the total movement; on the common queue it makes 236, while the best order is 208. The more important problem is fairness. Suppose that under heavy load requests keep arriving in the middle region of the disk: the head stays in the middle, and a request at cylinder 190 may wait for minutes. This is a case of **starvation**. FCFS and the sweep algorithms have no starvation.

11.10.4 Code trace: `sstf()`

A C version of SSTF is below. The array `done[]` records which requests have been served.

```
int sstf(int head, int q[], int n) {
  int total = 0;
  for (int k = 0; k < n; k++) {
    int best = -1;
    for (int i = 0; i < n; i++)
      if (!done[i] && (best < 0 ||
        abs(q[i]-head) < abs(q[best]-head)))
        best = i;          /* closest */
    total += abs(q[best] - head);
    head = q[best];
    done[best] = 1;
  }
  return total;
}
```

The outer loop serves one request per iteration and takes n iterations. The inner loop looks at all pending requests; `best` is the index of the closest request found so far. The `best < 0 ||` part of the condition makes the first pending request the candidate; later requests replace the candidate only if they are strictly closer. So if two requests are at equal distance, the one earlier in arrival order is chosen. At the end of the iteration the distance is added to `total`, the head moves to that request, and the request is marked done. Since n requests are scanned in each iteration, the algorithm uses $O(n^2)$ time and $O(n)$ extra space for `done[]`.

k	Head	Closest (distance)	Next candidate (distance)	total
0	50	43 (7)	24 (26)	7

k	Head	Closest (distance)	Next candidate (distance)	total
1	43	24 (19)	16 (27)	26
2	24	16 (8)	82 (58)	34
3	16	82 (66)	140 (124)	100
4	82	140 (58)	170 (88)	158
5	140	170 (30)	190 (50)	188
6	170	190 (20)	—	208

Table 11.10: Trace of `sstf()`. The head is at 50; the queue (arrival order) is 82, 170, 43, 140, 24, 16, 190.

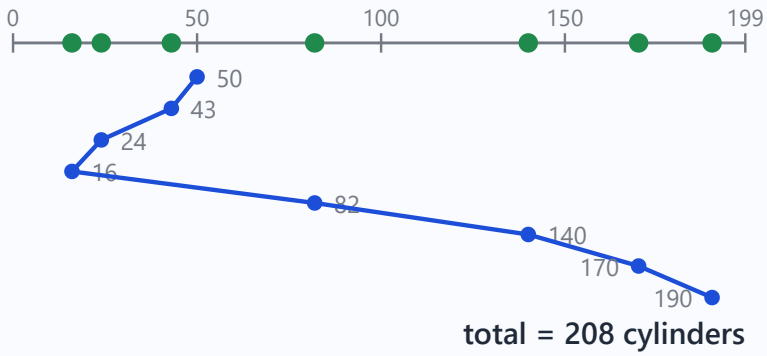


Figure 11.41: The end of the `sstf()` trace: all requests are served, `total` = 208.

On this queue FCFS would spend 642 cylinders; SSTF falls below a third of that with 208. The k = 3 row shows the weak point of SSTF: when the requests below are finished, the arm travels back a long 66-cylinder stretch. If requests in the middle kept arriving, those at the edges could wait.

11.10.5 SCAN (the elevator)

Definition 11.29

In the **SCAN** algorithm, the head moves in one direction to the end of the disk and serves the requests along its path; when it reaches the end it reverses and serves the requests in the other direction. Because it resembles an elevator going back and forth between the floors of a building, it is also called the **elevator algorithm**.

Example 11.17

On the common queue the head is moving toward 0 (downward). What is the total movement of SCAN?

1. 53 → 37: 16; total 16.

2. 37 → 14: 23; total 39.
3. 14 → 0: 14; total 53. Even though no requests remain below, the head goes all the way to the end of the disk.
4. The direction reverses. 0 → 65: 65; total 118.
5. 65 → 67: 2; total 120.
6. 67 → 98: 31; total 151.
7. 98 → 122: 24; total 175.
8. 122 → 124: 2; total 177.
9. 124 → 183: 59; total **236**.

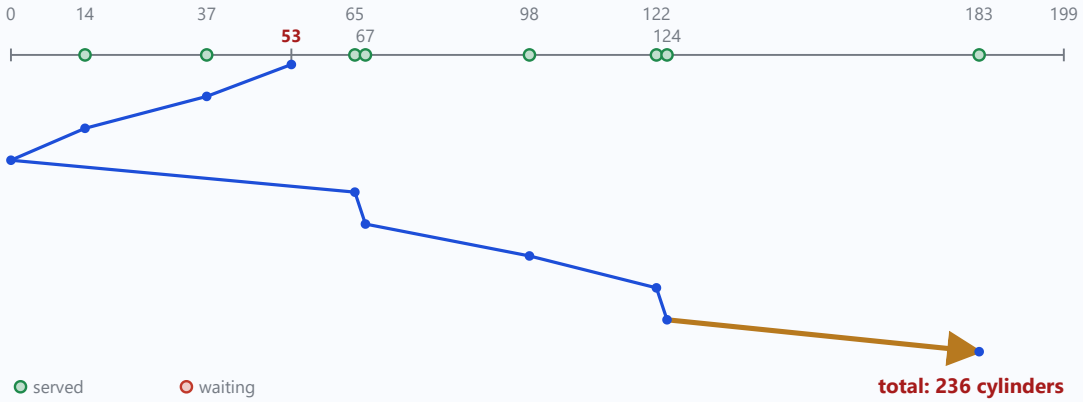


Figure 11.42: SCAN (downward): 53, 37, 14, 0, 65, 67, 98, 122, 124, 183. Total 236 cylinders.

On this queue SCAN gives the same total as SSTF but prevents starvation: the head serves every request along its path within one sweep. New requests that arrive ahead of the head are served in the same sweep, and those behind it on the return trip. The cost of going all the way to the end (14 → 0) is 14 wasted cylinders. The choice of direction also changes the result: on the same queue, if the head had started upward, the movement would have been 53 → 199 (146) and 199 → 14 (185), for a total of 331.

SCAN has a fairness flaw. When the head reaches one end and turns around, it has just served the region right behind it; few requests are waiting there. The place where requests are densest is the other end of the disk, and those requests have waited longest. The head passes the middle cylinders twice per round trip at equal intervals. The edge cylinders, however, it passes twice in quick succession, followed by a long gap. So requests that fall near the edges wait longer on average.

11.10.6 C-SCAN

Definition 11.30

C-SCAN (circular SCAN) serves in one direction only. The head goes up to the end; on reaching the end it returns to the start of the disk (to 0) without serving, and begins sweeping upward again. The disk is viewed as a ring.

Example 11.18

What path does C-SCAN (upward) follow on the common queue? What is the total if the return movement is counted too?

1. 53 → 65 → 67 → 98 → 122 → 124 → 183: 12 + 2 + 31 + 24 + 2 + 59 = 130.
2. 183 → 199: 16; total 146. There are no requests above, but the head goes to the end.
3. 199 → 0: the return, 199 cylinders, no service; total 345.
4. 0 → 14: 14; total 359.
5. 14 → 37: 23; total **382**.

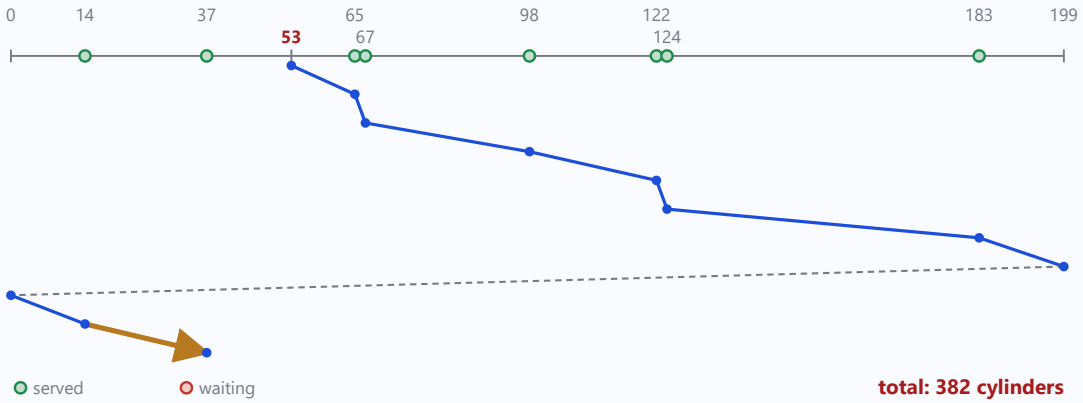


Figure 11.43: C-SCAN (upward): 53 ... 183, 199, return to 0, 14, 37. Total 382 if the return is counted, 183 cylinders if not.

If the return movement is not counted, the total is 382 − 199 = 183. Sources treat this differently; in an exam question it must be stated whether the return is counted. The return is in fact fast: the arm comes back in a single stroke without stopping at any sector. The gain of C-SCAN comes not from total movement but from fairness. Every request is served in the head's upward sweep, once; requests at the edge and in the middle wait about equally.

11.10.7 LOOK and C-LOOK

Definition 11.31

LOOK works like SCAN but does not go to the end of the disk: it turns around at the last request in that direction. **C-LOOK** serves in one direction like C-SCAN but does not go to the end: after the last request above, it jumps directly to the smallest pending request.

Example 11.19

What is the total movement of LOOK (downward) and C-LOOK (upward) on the common queue?

1. **LOOK, down.** 53 → 37 → 14: 16 + 23 = 39. No requests remain below; the head does not go to 0.
2. **LOOK, up.** The direction reverses: 14 → 65 → 67 → 98 → 122 → 124 → 183: 51 + 2 + 31 + 24 + 2 + 59 = 169. The total is 39 + 169 = **208**; 28 cylinders less than SCAN, because the 14 → 0 → 14 trip (2 × 14) is not made.
3. **C-LOOK, up.** 53 → 65 … 183: 130.
4. **The jump.** 183 → 14: 169 cylinders to the smallest pending request; the head does not go to 199 and 0. Total 299.
5. **The end.** 14 → 37: 23; total **322**. Without counting the jump, 322 − 169 = 153.

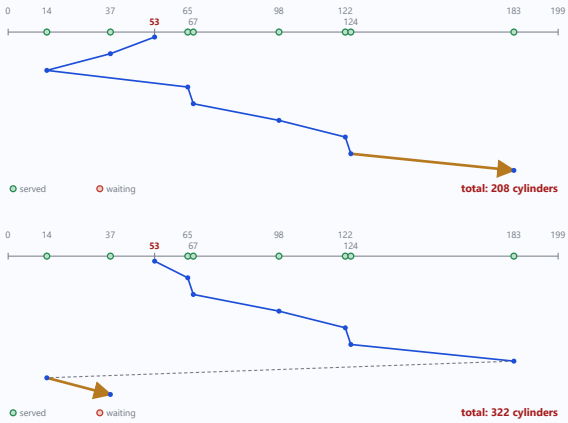


Figure 11.44: At the top, LOOK (downward): 53, 37, 14, 65 … 183; total 208. At the bottom, C-LOOK (upward): 53 … 183, jump to 14, 37; total 322 (153 no jump).

In practice, what is called “elevator” behavior is mostly LOOK; no real driver moves the arm to the end of the disk when there are no requests. Linux's old `deadline` and CFQ schedulers and today's `mq-deadline` dispatch requests in increasing LBA order and start over from the beginning after the

highest LBA. This is behavior similar to C-LOOK.

11.10.8 Code trace: `look()`

The version of LOOK that starts upward can be written with two loops over a sorted array:

```
/* s[] sorted; go up first, then down */
int look(int head, int s[], int n) {
  int i = 0, total = 0;
  while (i < n && s[i] < head) i++;
  for (int j = i; j < n; j++) {
    total += s[j] - head; head = s[j];
  }
  for (int j = i - 1; j >= 0; j--) {
    total += head - s[j]; head = s[j];
  }
  return total;
}
```

The first `while` loop finds the index (`i`) of the first request greater than or equal to the head. The first `for` loop advances upward from `i` to the end and adds up the distance of each step (`s[j] - head`). The second `for` loop descends from `i - 1` to 0; this time the distance is `head - s[j]` . Because the head turns around after the last request above without going to the end, the code is LOOK, not SCAN. The function runs in O(n) when the array is given sorted; sorting takes O(n log n).

Step	`j`	Target	Movement	`total`
`while`	`i` = 3: the first request greater than 50 is s[3] = 82. The 16, 24, and 43 below it will be taken on the way back.			
up	3	82	+32	32
up	4	140	+58	90
up	5	170	+30	120
up	6	190	+20	140
down	2	43	+147	287
down	1	24	+19	306
down	0	16	+8	314

Table 11.11: Trace of `look()` . s = 16, 24, 43, 82, 140, 170, 190; the head is at 50, moving up.

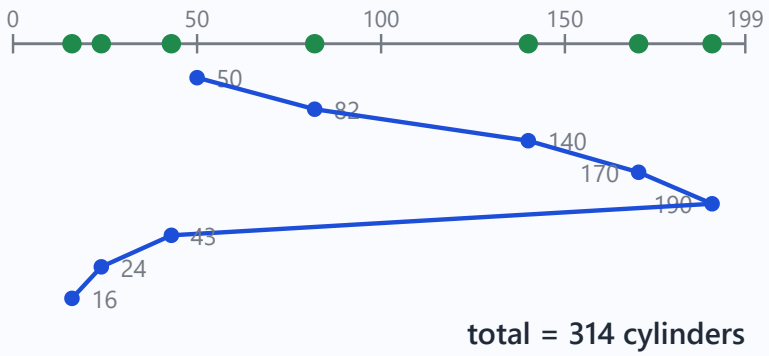


Figure 11.45: The end of the `look()` trace: total 314 cylinders.

On the same queue SSTF made 208; LOOK made 314. The difference comes from the choice of direction. If the head had started downward, LOOK would first have gone 50 → 16 (34 cylinders), then 16 → 190 (174 cylinders): 34 + 174 = 208. The choice of direction decides the result, but whichever direction is taken, no request starves.

11.10.9 Comparing the algorithms

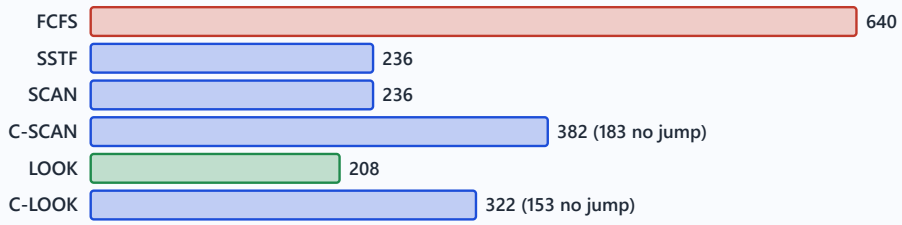


Figure 11.46: Total head movement of the six algorithms on the common queue: FCFS 640, SSTF 236, SCAN 236, C-SCAN 382 (183 no jump), LOOK 208, C-LOOK 322 (153 no jump).

Figure 11.46 shows the results on the common queue together. Total movement alone is not enough to choose an algorithm. LOOK gives the least movement (208); SSTF and SCAN follow with 236, but SSTF can cause starvation. C-SCAN and C-LOOK travel farther but spread the waiting times more evenly. When the load is light, all algorithms behave like FCFS, because there is usually only one request in the queue at a time.

Example 11.20

Tanenbaum's example: the head is at cylinder 11, and the requests in arrival order are 1, 36, 16, 34, 9, 12. How many cylinders do FCFS, SSTF, and the elevator (upward) move?

1. **FCFS.** 11 → 1 → 36 → 16 → 34 → 9 → 12: 10 + 35 + 20 + 18 + 25 + 3 = **111**.

2. **SSTF.** The order is 12, 9, 16, 1, 34, 36: 1 + 3 + 7 + 15 + 33 + 2 = **61**. With the head at 9, 16 (7 away) is closer than 1 (8 away).
3. **Elevator, up.** The order is 12, 16, 34, 36, 9, 1: 1 + 4 + 18 + 2 + 27 + 8 = **60**.
4. **Interpretation.** Tanenbaum's elevator turns around at the last request above (36) and does not go to the end; so it is actually LOOK.

Example 11.21

The cylinders are 0–199; the head is at 84 and moving upward because it last served 70. The requests in arrival order are 184, 142, 170, 8, 60, 155, 116, 37. What is the total movement of the six algorithms? Count the return trip in C-SCAN and C-LOOK.

1. **FCFS.** 100 + 42 + 28 + 162 + 52 + 95 + 39 + 79 = 597.
2. **SSTF.** 84 → 60 (24) → 37 (23) → 8 (29) → 116 (108) → 142 (26) → 155 (13) → 170 (15) → 184 (14): 252. At the start, 60 (24 away) is closer than 116 (32 away).
3. **SCAN.** 84 → 116 … 184 → 199, then 60, 37, 8: (199 − 84) + (199 − 8) = 115 + 191 = 306.
4. **C-SCAN.** 84 → 199 (115), 199 → 0 (199), 0 → 8 → 37 → 60 (60): 374. If the return is not counted, 175.
5. **LOOK.** 84 → 184 (100), 184 → 8 (176): 276.
6. **C-LOOK.** 84 → 184 (100), jump 184 → 8 (176), 8 → 60 (52): 328. If the jump is not counted, 152.
7. **Ranking.** SSTF 252 < LOOK 276 < SCAN 306 < C-LOOK 328 < C-SCAN 374 < FCFS 597. LOOK differs from SCAN only in not going needlessly to the end (199).

11.10.10 SPTF: counting rotational delay too

All the algorithms so far looked only at cylinder distance. Yet the time of a request is determined by seek and rotation together. If a nearby sector has just passed under the head, serving it means waiting almost a full revolution.

Definition 11.32

SPTF (shortest positioning time first) chooses, among the pending requests, the one for which the sum of seek time and rotational latency is smallest.

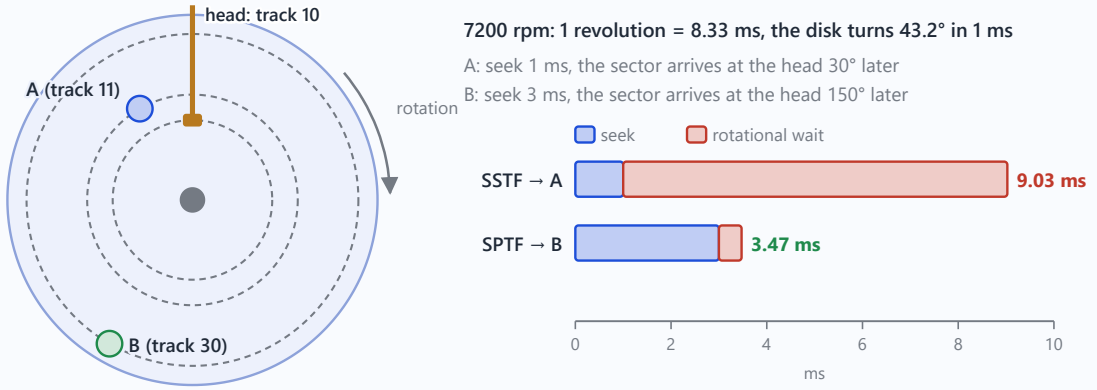


Figure 11.47: The head is at track 10. A (track 11) is near, but its sector is 30° ahead; B (track 30) is far, but its sector is 150° ahead. SSTF chooses A and spends 9.03 ms; SPTF chooses B and spends 3.47 ms.

Example 11.22

A disk spins at 7200 rpm. The head is at track 10. Request A is on track 11: seek 1 ms, and its sector is 30° ahead of the head in the direction of rotation. Request B is on track 30: seek 3 ms, and its sector is 150° ahead. Which do SSTF and SPTF choose, and how long does it take?

1. **Rotation speed.** One revolution is 60,000 / 7200 = 8.33 ms; the disk turns 360 / 8.33 = 43.2° in 1 ms.
2. **SSTF → A.** In the 1 ms seek the disk turns 43.2°. The sector was 30° ahead; when the seek ends it has passed the head by 13.2°. To bring the sector around again, 360 − 13.2 = 346.8° must be waited: 346.8 / 43.2 = 8.03 ms. Total 1 + 8.03 = 9.03 ms.
3. **SPTF → B.** In the 3 ms seek the disk turns 129.6°; 150 − 129.6 = 20.4° remain to B. The wait is 20.4 / 43.2 = 0.47 ms. Total 3 + 0.47 = 3.47 ms.
4. **Result.** The far track gave its result 9.03 / 3.47 ≈ 2.6 times sooner than the near track.

SPTF minimizes positioning time, but the operating system does not know the angular positions of sectors. So SPTF is implemented inside the disk, with NCQ. SPTF is prone to starvation just like SSTF. One method used against starvation is **F-SCAN**: when a sweep begins, the queue is frozen; new requests that arrive during the sweep wait for the next sweep. This way the old requests cannot be left behind by a constant stream of new requests being dumped in front of the head.

11.10.11 The controller cache and bad sectors

The controller cache. The disk keeps a track cache in its own DRAM (about 256 MB). It also reads blocks that pass under the head and have not yet been requested; the next sequential request is served directly from the cache. This cache is separate from the operating system's page cache.

Write cache and NCQ. The disk can take a write into its cache and immediately say “done”. The data has not yet reached the platter; if power is cut, it is lost. When durability is needed, the operating system uses the FLUSH command or a write with the FUA (force unit access) flag. NCQ orders the 32 queued requests by rotational position; this is an SPTF inside the hardware.

Bad sectors. Data written to a sector that goes bad during manufacture or use cannot be read back. The controller tracks the list of bad sectors found at the factory and those it finds in use. The “Reallocated Sectors” count in the SMART data shows the size of this list. A bad sector has two remedies (Figure 11.48).

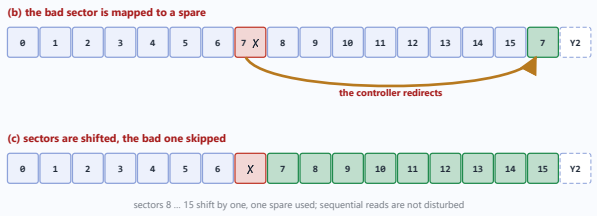


Figure 11.48: Two ways to handle bad sector 7. At the top, sector 7 is mapped to a spare. At the bottom, sectors 8–15 are shifted by one and one spare is used; sequential reads are not disturbed.

1. **Remapping to a spare.** The bad sector is mapped to a spare sector at the end of the track. It is simple, but a sequential request reading sectors 6, 7, 8 has to go to the end of the track and back for 7, and may pay an extra revolution.
2. **Shifting.** All sectors after the bad one are shifted by one. The whole track must be rewritten, but the order is preserved.

The operating system can keep a bad-block list too: on Linux the `badblocks` tool and the bad-block lists of file systems exist for this purpose. On today's disks, remapping is done entirely inside the disk, so this list often stays empty.

11.11 SSDs and NVMe

11.11.1 NAND flash

Definition 11.33

An **SSD** (solid-state drive) is a storage device with no moving parts that keeps data in NAND flash memory chips. The unit of reading and writing in flash is the **page** (about 16 KiB), and the unit of erasure is the **block**, made up of many pages (about 10–40 MiB).

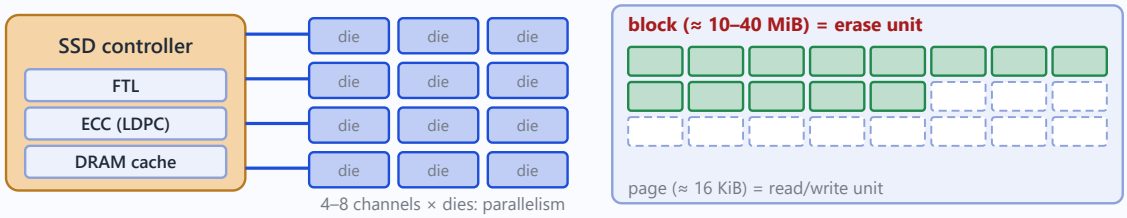


Figure 11.49: SSD internals: the controller (FTL, LDPC error correction, DRAM cache) connects to the NAND dies over 4–8 channels. Channels and dies work in parallel.

An SSD differs from a hard disk in two basic ways. First, with no moving parts there is no seek or rotational latency. Random access is about as fast as sequential access. Second, flash has **no in-place update**. A page can be written only when it is in the erased state, and erasing is done only at block level. In addition, the number of erasures each cell can withstand is limited. This number is called the P/E (program/erase) cycle count, and it depends on how many bits a cell stores (Table 11.12).

Cell	Bits/cell	Typical P/E cycles	Use
SLC	1	≈ 50–100 thousand	cache region, industrial
MLC	2	≈ 3–10 thousand	older enterprise
TLC	3	≈ 1–3 thousand	most consumer and server SSDs
QLC	4	≈ 0.1–1 thousand	high capacity, read-heavy

Table 11.12: NAND cell types.

As the bits per cell increase, the number of voltage levels the cell must distinguish grows as 2, 4, 8, 16. As the levels get denser, reads slow down and the cell wears out faster. The other way to raise capacity is vertical stacking: today's 3D NAND chips have more than 200–300 layers. The price gap per GB between SSDs and HDDs has narrowed; still, HDDs are cheaper for large capacity and archives.

11.11.2 The FTL, garbage collection, TRIM, and wear leveling

If in-place update is impossible, how is the operating system's request “overwrite LBA 1” satisfied? The answer is a translation layer in the SSD controller.

Definition 11.34

The **flash translation layer** (FTL) is controller software that maps logical block addresses (LBAs) to physical flash pages. Updated data is written to a new free page, the mapping table is changed, and the old page is considered invalid. This is called an **out-of-place update**.

Example 11.23

In a small SSD with two blocks (B0, B1) of four pages each, the following operations are performed: LBAs 0, 1, 2 are written; LBA 1 is rewritten; the file holding LBA 2 is deleted; LBA 3 is written. Then B0 is cleaned.

1. **First writes.** The FTL puts the data of the three LBAs into the free pages of B0 (s0, s1, s2) and records them in the table.
2. **Update.** LBA 1 is rewritten. The old page cannot be overwritten; the new data goes to B0.s3 and B0.s1 becomes invalid.
3. **TRIM.** When the file is deleted, the file system sends a TRIM command (Deallocate in NVMe). The SSD learns that the page of LBA 2 is no longer needed and marks it invalid.
4. **Block full.** B0 has no free pages left; LBA 3 is written to B1. Half of B0 is invalid.
5. **Garbage collection.** The valid pages in B0 (LBA 0 and LBA 1) are copied to B1 and B0 is erased. Two extra page writes that the user did not ask for have been made; this is called **write amplification**.

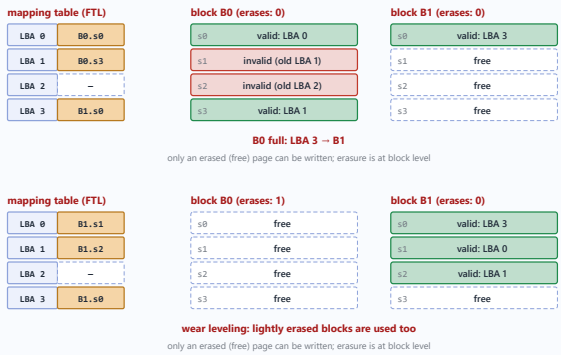


Figure 11.50: Two moments of the FTL. At the top, B0 is full and half of it is invalid; LBA 3 has been written to B1. At the bottom, after garbage collection: the valid pages have been moved to B1, and B0 is erased (erase count 1).

Without TRIM, the SSD would still consider the pages of a deleted file valid and would needlessly copy them during garbage collection too. Write amplification would rise, and both performance and lifetime would drop.

Definition 11.35

Wear leveling is the FTL policy that makes all flash blocks be erased roughly the same number of times. New writes are steered to lightly erased blocks; data that stays unchanged for a long time (cold data) is also moved occasionally so that the blocks it occupies can be used as well.

Without wear leveling, a few frequently updated blocks would wear out quickly, while the rest would stay almost unused. SSDs also set aside a spare area that is not shown to the user (over-provisioning). The spare area makes it easier for garbage collection to always find an empty block and reduces write amplification.

Common mistake

Defragmenting an SSD like a hard disk. On an SSD, having a file's pieces side by side brings no speedup, because there is no seek; defragmentation only adds needless writes and wear. The "Optimize" command in Windows does not defragment an SSD; it sends TRIM. What matters instead is aligning partitions to a 1 MiB boundary: logical blocks then sit on flash pages without being split, and a single write does not spill over into two pages.

11.11.3 NVMe and multiple queues

Definition 11.36

NVMe (NVM Express) is a command protocol designed for flash storage attached over PCIe. Each core can be given its own pair of submission (SQ) and completion (CQ) queues; the queues are circular buffers that live in memory.

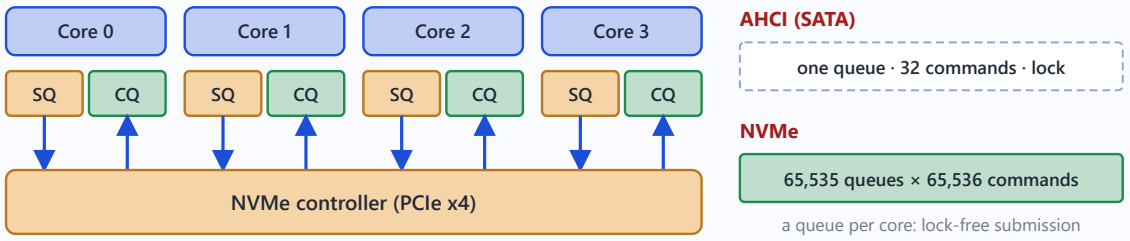


Figure 11.51: In NVMe each core has its own SQ/CQ pair and submission is lock-free. AHCI (SATA) has a single queue and 32 commands; all cores share the same lock.

SATA and its AHCI interface were designed for spinning disks. It has a single queue and at most 32 commands; this limit was enough for a hard disk, because the disk carries out only one request at a time anyway. Flash, on the other hand, can run hundreds of jobs in parallel across channels and dies. NVMe exposes this parallelism: it defines 64 K queues and 64 K commands per queue. Because each core writes to its own queue, there is no lock contention among cores; the completion interrupt also reaches the same core through MSI-X.

A few terms are often confused. NVMe is a protocol; M.2 is a card form (a form factor). An M.2 card can also be SATA. U.2/U.3 and E1.S/E3.S are cabled or slotted forms used in servers. NVMe driver support has been built into all common operating systems since 2013; the incompatibility with AHCI

is only at the protocol level. Latency on NVMe disks is between 10 and 100 µs; PCIe 5.0 x4 disks deliver about 14 GB/s and more than 2 million requests per second. NVMe-oF carries the same protocol over the network; ZNS (zoned namespaces) exposes the block structure of flash directly to software.

11.11.4 Linux blk-mq and I/O schedulers

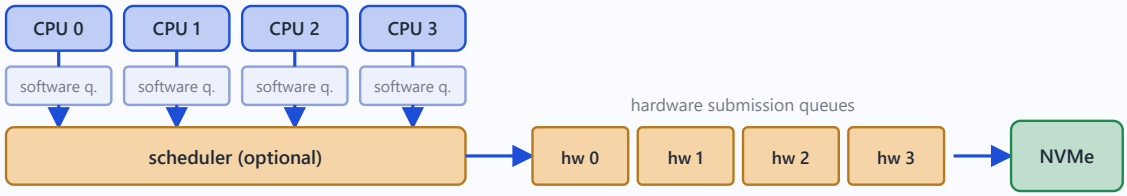


Figure 11.52: Linux blk-mq: per-CPU software queues, an optional scheduler, and the device's hardware submission queues.

Linux's block layer follows NVMe's multi-queue model. **blk-mq** (the multi-queue block layer) gives each CPU a software queue and maps them to the device's hardware queues. In the old single-queue design, at millions of IOPS all cores would wait on the same queue lock; blk-mq removes this contention. In Linux 5.0 the old single-queue block layer and its schedulers (noop, deadline, CFQ) were removed entirely. Today's **I/O schedulers** are summarized in Table 11.13.

Scheduler	Idea	Suited for
`none`	No ordering, dispatch directly	NVMe (default), fast SSDs
`mq-deadline`	Sorted by LBA + read/write deadline	SATA HDD/SSD (default in most distributions)
`bfq`	Per-process budget, fair sharing	Desktop, interactivity on slow devices
`kyber`	Limits read/write latency with targets	Fast multi-queue devices

Table 11.13: Linux blk-mq I/O schedulers.

A device's scheduler is read and changed through `sysfs`:

```
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ echo bfq | sudo tee /sys/block/sda/queue/scheduler
```

The name in square brackets is the active scheduler; on an NVMe disk this is `none`. The second command sets the scheduler of a SATA disk to `bfq`. `sudo tee` is used because the redirection (`>`) would be performed outside `sudo`, with the user's own privileges.

Why is no sorting done at all on NVMe? The reason arm scheduling exists is to reduce seek time. An SSD has no seek, and access time is almost independent of position. Multiple hardware queues already give every core a parallel path. Sorting only spends CPU and adds latency. On an SSD the scheduler's job is not to sort but, when needed, to preserve fairness and latency targets; for that, `mq-deadline`, `bfq`, or `kyber` can be chosen.

11.12 RAID and stable storage

11.12.1 RAID: combining disks

Definition 11.37

RAID (Redundant Array of Independent Disks) is an arrangement that presents many disks to the operating system as a single disk. **Striping** distributes consecutive blocks across different disks to provide parallelism. **Mirroring** keeps a copy of every block on another disk. **Parity** stores the XOR of several blocks in a separate block so that a lost block can be recomputed.

The idea of RAID goes back to the 1988 paper by Patterson, Gibson, and Katz. The paper proposed building an array out of inexpensive disks instead of a single large expensive disk (SLED). That is why the "I" in the acronym originally stood for *inexpensive*. Today it is read as "independent". The standard levels are 0 through 6. RAID can be built with a hardware card or in software: `mdadm` on Linux, RAID-Z in ZFS, Btrfs, and Windows Storage Spaces are examples of software RAID.

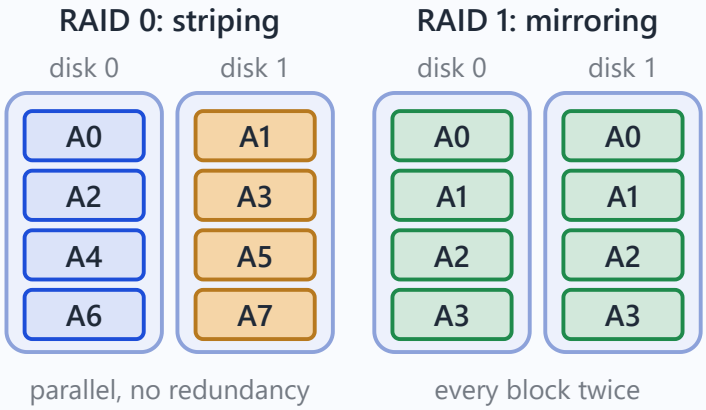


Figure 11.53: RAID 0 distributes consecutive blocks (A0, A1, ...) across the disks in turn: parallel but with no redundancy. RAID 1 writes every block to two disks at once.

Layout	Usable	Tolerates	Cost of a small random write
RAID 0 (striping)	16 TB	0 disks	1 I/O

Layout	Usable	Tolerates	Cost of a small random write
RAID 1 (4-way mirror)	4 TB	3 disks	4 I/Os (one to each copy)
RAID 10 (mirror + stripe)	8 TB	1 (2 if in different pairs)	2 I/Os
RAID 5 (distributed parity)	12 TB	1 disk	4 I/Os (2 reads + 2 writes)
RAID 6 (double parity)	8 TB	2 disks	6 I/Os (3 reads + 3 writes)

Table 11.14: RAID layouts with four 4 TB disks.

Common mistake

Treating RAID as a backup. RAID protects against disk failure, but not against user error, software bugs, or ransomware. A deleted file is deleted on all disks at the same time; a corrupted file is corrupted in every copy. A backup is a copy made on a separate medium and at a separate time.

11.12.2 RAID levels

Figure 11.54: At the top, RAID 0: stripes (s0, s1, ...) are spread over four disks in turn. In the middle, RAID 5: the parity blocks (P) sit on a different disk in each stripe. At the bottom, RAID 6: the P and Q parities are distributed over five disks.

- **RAID 0.** Consecutive stripes go to different disks. On large requests, n disks give n times the bandwidth. There is no redundancy: if one disk fails, everything in the array is lost.
- **RAID 1.** Every write is done twice. A read can be served from either disk; the load is spread across the disks. Half of the capacity goes to redundancy.
- **RAID 10.** Mirror first, then stripe. Writes are fast, and a failed disk is rebuilt quickly by copying from its mirror alone. It is common in databases.

- **RAID 2 and 3.** RAID 2 uses a Hamming code at the word level, and RAID 3 uses a single parity disk at the byte level. Both levels require all disks to spin in lockstep. They are not used today, because disks already do ECC internally.
- **RAID 4.** Block-level striping and a separate parity disk. Every small write updates the parity disk too; the parity disk becomes a bottleneck.
- **RAID 5.** Parity blocks are rotated across the disks and the bottleneck disappears. The capacity of n − 1 of the n disks is usable; it tolerates the loss of one disk.
- **RAID 6.** It keeps two parities: P (XOR) and Q (a Reed–Solomon code over the field $GF(2^8)$). It tolerates the loss of two disks. Because rebuilding takes hours on large disks, it is preferred to guard against a second failure during that time.

11.12.3 Parity and XOR

The foundation of parity is the XOR ($\oplus$) operation. If two bits differ, XOR gives 1; if they are the same, 0. On bit strings, XOR is applied separately at each bit position. Three properties of the operation carry all of RAID's calculations: $x \oplus x = 0$, $x \oplus 0 = x$, and the operation is commutative and associative.

Theorem 11.1 (Recovery with parity)

Let $P = D_0 \oplus D_1 \oplus \ldots \oplus D_{n-1}$. Any block D_k whose loss is known equals the XOR of the remaining data blocks and P: $D_k = P \oplus$ (the XOR of all D_i other than k).

XOR both sides of the definition of P with all the blocks other than D_k. Each D_i ($i \neq k$) appears twice on the right side and vanishes because $x \oplus x = 0$; only D_k is left. A single parity suffices because we know which disk has failed: the disk either does not respond at all or reports an ECC error. Parity does not find the bad block; it only recomputes a block whose location we already know.

Example 11.24

In a RAID 5 stripe, D0 = 1011 0010, D1 = 0101 1100, D2 = 1110 0111 (8-bit sample blocks). What is the parity? If disk 1 fails, how is D1 found? If 0000 1111 is written in place of D1, what is the new parity?

1. **Parity.** D0 $\oplus$ D1 = 1110 1110; $\oplus$ D2 = 0000 1001. P = 0000 1001. The number of 1s at each bit position is now even.
2. **Recovery.** D1 = D0 $\oplus$ D2 $\oplus$ P = 1011 0010 $\oplus$ 1110 0111 $\oplus$ 0000 1001. The first two give 0101 0101, and with P this gives 0101 1100: the original D1. All stripes are rewritten onto the new disk this way; this is called rebuilding.

3. **Small write.** D1' = 0000 1111. The old D1 and old P are read. P' = P ⊕ D1 ⊕ D1' = 0000 1001 ⊕ 0101 1100 ⊕ 0000 1111 = 0101 1010. Then D1' and P' are written: 4 I/Os (read-modify-write).
4. **Check.** D0 ⊕ D1' ⊕ D2 = 1011 0010 ⊕ 0000 1111 ⊕ 1110 0111 = 0101 1010 = P'. The correct parity was found without reading the whole stripe.

Figure 11.55: At the top, recovery: D1 = D0 ⊕ D2 ⊕ P = 0101 1100. At the bottom, a small write: when D1' = 0000 1111 is written, P' = P ⊕ D1 ⊕ D1' = 0101 1010.

Rebuilding carries a risk too. On consumer-class disks the rate of unrecoverable read errors (UREs) is about 1 in 10^{14} bits. To rebuild the failed disk, all of the remaining disks must be read.

Example 11.25

A disk in a RAID 5 array has failed; a total of 12 TB will be read from the remaining disks for the rebuild. The URE rate is 1 in 10^{14} bits. What are the expected number of errors during the read and the probability of at least one error?

1. **Number of bits.** 12×10^{12} B $\times 8 = 9.6 \times 10^{13}$ bits.
2. **Expected errors.** $9.6 \times 10^{13} / 10^{14} = 0.96$.
3. **At least one error.** If errors are independent and rare, their count is approximately Poisson distributed: $1 - e^{-0.96} \approx 0.62$. The probability that the rebuild hits at least one unreadable sector is about 62%.
4. **Interpretation.** In RAID 5, no redundancy is left at this point; the stripe of the sector that is hit cannot be recovered. This is why RAID 6 is chosen for large arrays: the second parity corrects this error during the rebuild as well.

11.12.4 Code trace: small write and recovery

```
void small_write(int d, uint8_t nv) {
  uint8_t ov = rd(d);      /* I/O 1 */
  uint8_t p = rd(P);       /* I/O 2 */
  p = p ^ ov ^ nv;
  wr(d, nv);               /* I/O 3 */
  wr(P, p);                /* I/O 4 */
}
uint8_t rebuild(int bad) {
  uint8_t x = 0;
  for (int d = 0; d <= P; d++)
    if (d != bad)
      x ^= rd(d);          /* incl. P */
  return x;
}
```

In the code, disks are numbered from 0; `P` is the number of the parity disk (here 3). `rd` and `wr` are single disk I/Os. `small_write` replaces the block on disk d with `nv`. First the old data and old parity are read (2 I/Os). `p ^ ov` removes the effect of the old data from the parity, since $x \oplus x = 0$; `^ nv` adds in the new value. Then the new data and new parity are written (2 I/Os). The other data disks of the stripe are never read. `rebuild` computes the lost block by XORing all the disks other than the failed one, parity included.

Step	Operation	Value	I/O
1	`ov = rd(1)`	0110 1001	1
2	`p = rd(P)`	0001 1110	2
3	`p = p ^ ov ^ nv`	0001 1110 $\oplus$ 0110 1001 $\oplus$ 0011 0011 = 0100 0100	2
4	`wr(1, nv)`	D1 = 0011 0011	3
5	`wr(P, p)`	P = 0100 0100	4
6	`rebuild(2)`, d = 0	x = 1011 0010	
7	d = 1	x = 1011 0010 $\oplus$ 0011 0011 = 1000 0001	
8	d = 2 is skipped; d = 3 (P)	x = 1000 0001 $\oplus$ 0100 0100 = 1100 0101	3 reads

Table 11.15: `small_write(1, 0011 0011)` followed by `rebuild(2)`. Initially D0 = 1011 0010, D1 = 0110 1001, D2 = 1100 0101, P = 0001 1110.

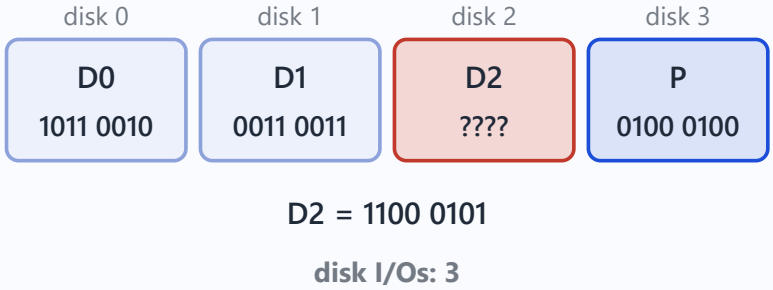


Figure 11.56: The result of `rebuild(2)`: x = 1100 0101, the same as the lost D2.

The recovered value equals the original D2. The new disk is rebuilt by doing 3 reads for every stripe, and the array is unprotected during this time. The small write has a weak point too: if the system crashes between the two writes in steps 4 and 5, the data is new while the parity is old, and the stripe becomes inconsistent. This is called the **write hole**. If a disk then fails, wrong data is computed from the wrong parity. Hardware RAID cards prevent this with a battery-backed cache, and ZFS by never updating stripes in place.

Example 11.26

In a stripe with four data disks, D0 = 1100 1010, D1 = 0011 0110, D2 = 1010 0001, D3 = 0101 1100. Find P; if the disk of D3 fails, recover D3; if 1111 0000 is written in place of D0, what is the new P?

1. **P.** D0 ⊕ D1 = 1111 1100; ⊕ D2 = 0101 1101; ⊕ D3 = 0000 0001. P = 0000 0001.
2. **Recovery.** D0 ⊕ D1 ⊕ D2 ⊕ P = 0101 1101 ⊕ 0000 0001 = 0101 1100: the original D3.
3. **Small write.** The changed bits are D0 ⊕ D0' = 1100 1010 ⊕ 1111 0000 = 0011 1010. P' = P ⊕ 0011 1010 = 0011 1011.
4. **Check.** D0' ⊕ D1 ⊕ D2 ⊕ D3 = 1111 0000 ⊕ 0011 0110 ⊕ 1010 0001 ⊕ 0101 1100 = 0011 1011. The result is the same.

11.12.5 RAID performance

The speed of a RAID layout is easily expressed with two measures of a single disk: the rate R (IOPS) on small random requests, and the rate S (MB/s) on large sequential transfers. The model of Arpaci-Dusseau and coauthors starts from these two numbers.

Example 11.27

A single disk has 7200 rpm, an 8 ms average seek, and a 200 MB/s media rate. The array consists of N = 4 disks. What are R and S? What are the rates of RAID 0, 1, 4, and 5?

1. **R.** A random 4 KiB read takes 12.19 ms; R = 1000 / 12.19 ≈ 82 IOPS (0.34 MB/s).
2. **S.** A sequential 10 MiB chunk: the transfer is 10,485,760 / (200 × 10^6) = 52.43 ms; the total is 8 + 4.17 + 52.43 = 64.6 ms. S = 10.49 MB / 64.6 ms ≈ 162 MB/s. This value assumes that seek and rotational latency are paid again for each chunk and that the media rate is constant; in a real long sequential read the seek is made once and S approaches the media rate (200 MB/s).
3. **RAID 0.** Each request goes to a single disk and the four disks work in parallel: sequential 4 · S ≈ 649 MB/s, random reads and writes 4 · R ≈ 328 IOPS.
4. **RAID 1 (two mirrored pairs).** The capacity is 2 disks. A random read can be served from either copy: 4 · R ≈ 328. Every write goes to two disks: 2 · R ≈ 164. Sequential reads, however, reach only 2 · S ≈ 325 MB/s.
5. **RAID 4.** The capacity is 3 disks. Sequential 3 · S ≈ 487 MB/s; random reads 3 · R ≈ 246. On a small write, the parity disk does 1 read and 1 write for every write: R / 2 ≈ 41 IOPS.
6. **RAID 5.** The capacity is 3 disks. Sequential 3 · S ≈ 487 MB/s; random reads use all the disks: 4 · R ≈ 328. The 4 I/Os of a small write are spread across the disks: (4 / 4) · R ≈ 82 IOPS.

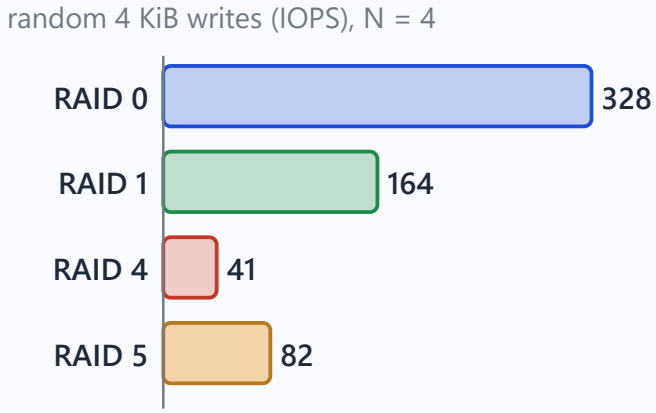


Figure 11.57: Random small-write rate on four disks: RAID 0 328, RAID 1 164, RAID 4 41, RAID 5 82 IOPS.

Layout	Capacity	Sequential read/write	Random read	Random write	Write latency
RAID 0	N = 4	N·S ≈ 649 MB/s	N·R ≈ 328	N·R ≈ 328	D
RAID 1 (2-way mirror)	N/2 = 2	(N/2)·S ≈ 325 MB/s	N·R ≈ 328	(N/2)·R ≈ 164	D (slower copy)
RAID 4	N − 1 = 3	(N−1)·S ≈ 487 MB/s	(N−1)·R ≈ 246	R/2 ≈ 41	2D
RAID 5	N − 1 = 3	(N−1)·S ≈ 487 MB/s	N·R ≈ 328	(N/4)·R ≈ 82	2D

Table 11.16: Rates of arrays of N disks (R: random IOPS of a single disk, S: its sequential rate, D: the latency of a single request). The numbers are for N = 4, R ≈ 82, S ≈ 162 MB/s.

Three rows of the table call for explanation. Why is **sequential read on a mirror** only (N/2)·S? The two copies can share the blocks: one disk reads the even-numbered blocks and the other the odd-numbered ones. But each disk must still rotate past the blocks it skips; skipped blocks save no time. **A small write in RAID 4** requires one read and one write on the parity disk every time. The array's rate falls to half that of a single disk, and adding disks does not help at all. **RAID 5** spreads the same 4 I/Os over all the disks, so the small-write rate is N·R/4 and grows with the number of disks. When a full stripe is written, the old data is not read; the parity is computed from the new data and the rate is (N − 1)·S.

Example 11.28

A single disk delivers 150 random IOPS. In an array of 8 disks, what is the small random write rate for RAID 0, RAID 10 (4 mirrored pairs), RAID 4, RAID 5, and RAID 6? If the number of disks rises to 16, which one does not speed up at all?

1. **RAID 0.** 1 I/O per write: 8 × 150 = 1200 IOPS.
2. **RAID 10.** Two copies, 2 I/Os: 1200 / 2 = 600.
3. **RAID 4.** 4 I/Os, but two of them always on the parity disk: 150 / 2 = 75.
4. **RAID 5.** 4 I/Os, spread across the disks: 1200 / 4 = 300.
5. **RAID 6.** 6 I/Os (3 reads + 3 writes): 1200 / 6 = 200.
6. **16 disks.** RAID 4 stays at 75; the bottleneck is the single parity disk. RAID 5 rises to 600.

11.12.6 Stable storage

RAID protects against bad sectors and failed disks, but it has no answer to what happens if there is a crash during a write. If power is cut while writing to two mirrored disks at the same time, both copies can be left half-written.

Definition 11.38

Stable storage is a storage arrangement that guarantees that a write either completes correctly or leaves the old data in place. Tanenbaum's model rests on three assumptions: the probability of the same block going bad on two disks at the same time is neglected; faulty data is detected by the ECC; if the CPU crashes, the write in progress stops.

The arrangement is built from two disks and three operations. **Stable write**: the block is first written to disk 1, read back and verified; if necessary it is retried up to n times, and failing that it is written to a spare sector. Once disk 1 is written successfully, the same is done for disk 2. **Stable read**: the block is read from disk 1; if it cannot be read with a valid ECC, it is read from disk 2. **Recovery**: after a crash, the blocks on the two disks are compared and any inconsistency is resolved.

Figure 11.58: Recovery after a crash. At the top the crash happened while writing disk 1: disk 1 is bad (ECC), and the old value is restored from disk 2. At the bottom disk 1 was written and the crash came before disk 2 was written: the new value is copied to disk 2.

Example 11.29

What does recovery do if a crash occurs at each of the five moments of a stable write?

1. **Before writing disk 1.** Both copies are old. Nothing is done; the write is treated as if it never happened.
2. **While writing disk 1.** The block on disk 1 is half-written and gives an ECC error. The old value is copied from disk 2 to disk 1.
3. **Disk 1 written, disk 2 not started.** Both copies are valid but different. The rule: disk 1 wins. The new value is written to disk 2 and the write is completed.
4. **While writing disk 2.** Disk 2 is bad (ECC); the new value is copied from disk 1.
5. **Both copies new.** The write is complete; nothing is done.

In every case the result is either entirely old or entirely new. What ensures this is the fixed order of the writes. At any moment at least one copy is either entirely old or entirely new and valid; recovery spreads that copy to the other. RAID 1 does not ensure this by itself, because it writes to both disks at the same time and keeps no order that says which is “correct”.

The same idea lives on today in other forms. File systems first write changes to a log (journaling) and apply them to their real places only once the log is durable. Write barriers and FUA make it possible to be sure that certain writes are durable on the disk. In enterprise SSDs, power-loss protection capacitors store enough energy to write the data in the cache to flash when the power goes out.

11.13 Summary

- The operating system talks not to the device but to the **device controller**. The controller has status, command, data-in, and data-out registers: writing a register issues a command, and reading it learns the status.

- Registers are reached with the special `IN` / `OUT` instructions in **port-mapped** I/O and with ordinary load and store instructions in **memory-mapped** I/O. In MMIO the register page must be mapped uncacheable and the pointer must be `volatile`. Today PCIe devices use MMIO through BARs.
- There are three transfer methods. In **polling** the CPU waits on the status register in a loop; in **interrupt-driven** I/O an interrupt arrives for every unit; in **DMA** the CPU handles only the setup and the completion interrupt. In the 4096-character example the CPU costs are 4.096 s, 8.19 ms, and about 7 µs.
- When an interrupt arrives, the CPU stops at an instruction boundary, the hardware saves the PC and PSW, the vector becomes an index into the IDT, the handler acknowledges the device and wakes the waiter, the scheduler chooses, and control returns with `iret`. Modern devices send the interrupt as a memory write with **MSI-X**. The handler is split into a top half and a bottom half.
- **Scatter-gather** writes to noncontiguous pages in a single DMA operation; the **IOMMU** translates device addresses and isolates devices from one another.
- I/O software is layered: the user library, device-independent software (on Linux the VFS, page cache, file system, and block layer), the device driver, and the interrupt handler. Drivers offer a common function table (`file_operations`).
- **Buffering** covers speed mismatch, size mismatch, and copy semantics. A **circular buffer** holds at most N − 1 bytes in N slots: head = tail means empty, (tail + 1) mod N = head means full. With a single producer and a single consumer no lock is needed.
- A **blocking** call does not return until data arrives; a **nonblocking** call returns whatever is there, or `EAGAIN`; an **asynchronous** call starts the job and reports later. epoll reports readiness and io_uring reports completion.
- Clock hardware decrements a counter on every pulse and raises an interrupt at zero. The **tickless kernel** stops the tick while idle. Pending timers are kept in a **delta list**: O(1) work per tick, O(n) for insertion.
- Disk access time = seek + rotational latency + transfer. At 7200 rpm, an 8 ms seek, and 200 MB/s, a random 4 KiB read takes 12.19 ms (≈ 82 IOPS) and a sequential 1 MiB read takes 17.41 ms (≈ 60 MB/s).
- Total head movements on the common disk-scheduling queue (head at 53): FCFS 640, SSTF 236, SCAN 236, C-SCAN 382 (183 without the return), LOOK 208, C-LOOK 322 (153 no jump). SSTF can cause starvation; the sweep algorithms cannot. **SPTF** accounts for rotational latency too and is implemented inside the disk with NCQ.
- An **SSD** has no seek and cannot update in place. The FTL does out-of-place updates, garbage collection, and **wear leveling**; TRIM reduces write amplification. NVMe offers a queue per core; Linux blk-mq follows this model, and the default scheduler on NVMe is `none`.
- **RAID** combines disks with striping, mirroring, and parity. Parity is XOR; a lost block is the XOR of the rest. In RAID 5 a small write needs 4 I/Os, in RAID 6 it needs 6. In RAID 4 the parity disk is a bottleneck. RAID is not a backup.

- **Stable storage** writes to two disks in a fixed order and guarantees, after any crash, either entirely old or entirely new data. Today the same idea lives on in journaling, FUA, and power-loss protection.

11.14 Key terms

bus device controller block / character device memory-mapped I/O polling busy waiting programmed I/O interrupt interrupt vector interrupt handler exception trap pipeline interrupt-driven I/O direct memory access (DMA) device driver buffer / buffering page cache spooling blocking / nonblocking / asynchronous I/O timer hard disk drive (HDD) seek time / rotational latency disk scheduling solid-state drive (SSD) wear leveling I/O scheduler RAID parity journaling

---

11.15 Test Yourself

1. The cylinders are 0–199; the head is at 100 and moving upward. The requests in arrival order are 55, 58, 39, 18, 90, 160, 150, 38, 184. What is the total movement of FCFS, SSTF, LOOK, and C-LOOK (count the jump)?

   Answer

   FCFS: 45 + 3 + 19 + 21 + 72 + 70 + 10 + 112 + 146 = 498. SSTF: in the order 90, 58, 55, 39, 38, 18, 150, 160, 184: 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = 248. LOOK: 100 → 184 (84), 184 → 18 (166): 250. C-LOOK: 100 → 184 (84), jump 184 → 18 (166), 18 → 90 (72): 322.

2. A disk spinning at 5400 rpm has an average seek time of 10 ms and a media rate of 100 MB/s (1 MB = 10^6 B). How long does a random 8 KiB read take on average? About how many random reads per second can the disk perform?

   Answer

   One revolution is 60,000 / 5400 ≈ 11.11 ms; the average rotational latency is 5.56 ms. The transfer is 8192 / 10^8 s ≈ 0.08 ms. The total is 10 + 5.56 + 0.08 ≈ 15.64 ms; 1000 / 15.64 ≈ 64 reads per second.

3. A circular buffer with N = 4 slots is empty. The calls in order are: put x, put y, get, put z, put w, put

v, get, put u. Which call fails? What are head and tail at the end?

Answer

x is written to slot 0 and y to slot 1 (tail 2). get reads x (head 1). z is written to slot 2 and w to slot 3; the tail wraps around (0). At put v, (0 + 1) mod 4 = 1 = head: the buffer is full and −1 is returned. get reads y (head 2). u is written to slot 0 (tail 1). Result: head = 2, tail = 1; the buffer holds z, w, u.

4. The delta list is A 5, B 2, C 6. If X, which expires in 9 ticks, is inserted, what does the list become? How many nodes does `add` pass?

Answer

A's delta 5 ≤ 9 → t = 4; B's delta 2 ≤ 4 → t = 2; C's delta 6 > 2, the loop stops. X goes in before C with delta 2, and C's delta becomes 6 − 2 = 4. The list is A 5, B 2, X 2, C 4. Two nodes are passed. C's absolute time stays at 13 ticks from now.

5. In a RAID 5 stripe, D0 = 0110 1101, D2 = 1001 0011, P = 1010 0110. The disk of D1 has failed. What is D1?

Answer

D1 = D0 ⊕ D2 ⊕ P. D0 ⊕ D2 = 1111 1110; ⊕ P = 0101 1000. D1 = 0101 1000.

6. A network card receives a burst of 2000 packets. The interrupt-driven method needs a 3 µs interrupt for each packet; a DMA ring needs 10 µs of setup and a single 3 µs interrupt at the end of the burst. What are the CPU costs? Which is cheaper for single-packet traffic?

Answer

Interrupt-driven: 2000 × 3 = 6000 µs = 6 ms. DMA: 10 + 3 = 13 µs; about 460 times less. For a single packet, interrupt-driven costs 3 µs and DMA 13 µs: the fixed setup cost does not pay for itself on a small transfer.

7. A program wants to watch a file on disk with epoll and read it when it is “ready”. Why does this not give asynchronous reading? Which interface does?

Answer

epoll only reports readiness. Disk files always look ready, because the kernel does not refuse a disk read with `EAGAIN`; it completes it by blocking. The read still waits inside `read`. io_uring (or IOCP on Windows), which starts the job and reports completion later, provides true asynchrony.

8. On an SSD the file system does not send TRIM. After the user deletes a large file, why does the disk slow down as it fills up again, and why does it wear out sooner?

   Answer

   The SSD still considers the pages of the deleted file valid. When garbage collection cleans a block, it copies these pages to another block too. Every page copied is an extra write: write amplification rises, writes wait for garbage collection, and cells are erased and written needlessly.

9. A driver contains a wait loop that reads the status register through an `int *status` pointer. When optimization is turned on, the loop never ends. Why? What are the two fixes?

   Answer

   The compiler assumes `*status` does not change in the loop and performs the read once. The pointer must be made `volatile int *`. In addition, the register page must be mapped uncacheable (`ioremap` on Linux); otherwise the CPU reads the old value from the cache.

10. In stable storage, both copies return valid ECC but their values differ. What does recovery do? At which moment of the crash does this situation point?

    Answer

    Disk 1 wins; its value is copied to disk 2 and the write is completed. This situation shows that the write to disk 1 had finished and been verified, while the write to disk 2 had not yet started. If the crash had come while writing disk 2, the block on disk 2 would have given an ECC error.

11.16 Further reading

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters “I/O Devices”, “Hard Disk Drives”, “Redundant Arrays of Inexpensive Disks”, and “Flash-based SSDs”: a short, quantitative treatment of disk scheduling, the RAID performance model, and the FTL; free at ostep.org.
- D. A. Patterson, G. Gibson, and R. H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, SIGMOD 1988: the short paper that first defined the RAID levels, still readable today.
- J. Axboe, “Efficient IO with io_uring” (kernel.dk): the document in which io_uring's designer describes the interface, the rings, and the SQPOLL mode.
- J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed., O'Reilly, 2005: the classic source on character drivers, interrupt handling, and DMA, with examples; some of the kernel interfaces are out of date, but the concepts remain valid.

11.17 References

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 11: Mass-Storage Structure; Chapter 12: I/O Systems.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 5: Input/Output.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "I/O Devices", "Hard Disk Drives", "RAID", "Flash-based SSDs".
4. D. A. Patterson, G. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", SIGMOD 1988.
5. NVM Express Base Specification 2.x; PCI-SIG PCIe 5.0/6.0; USB4 Specification; UEFI Specification (GPT).
6. Linux kernel documentation: "Multi-Queue Block IO Queueing Mechanism (blk-mq)", "NO_HZ: Reducing Scheduling-Clock Ticks", io_uring (J. Axboe, "Efficient IO with io_uring"); J. Corbet, A. Rubini, G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed.

PART IV

Advanced Topics

Chapters 12–15

Multiprocessor Systems

Even an ordinary laptop today has eight or sixteen cores. On servers the number runs into the hundreds, and in a data center thousands of computers work together on a single job. An operating system designed for a single processor cannot exploit this hardware. How cores share the same memory, how caches stay coherent, how locks are implemented in this environment, and how threads are distributed over processors are new questions. The notes first describe the hardware of shared-memory multiprocessors: the bus, the crossbar and omega networks, NUMA, directory-based coherence, the MESI protocol, and false sharing. They then move on to multiprocessor operating system models, locks, and scheduling. Multicomputers that do not share memory, message passing, remote procedure call, and load balancing follow. The last sections cover virtualization, containers, middleware in distributed systems, and the cloud.

LEARNING OUTCOMES

Hardware
Compare UMA (bus, crossbar, omega) and NUMA architectures; trace directory-based coherence on one address.

Cache coherence
Apply MESI state transitions to a sequence of reads and writes; find and fix false sharing.

Operating system and locks
Evaluate the separate-OS, master-slave, and SMP models; the TSL, ticket, TTAS, and MCS locks; and spinning versus switching.

Scheduling
Apply the limits of a shared ready queue, space sharing, and gang scheduling to an example.

Multicomputers
Compute the diameter of topologies; explain message passing, RPC, DSM, and load-balancing methods.

Virtualization and distributed systems
Recognize the hypervisor types, VT-x and EPT, containers, the kinds of middleware, and the cloud.

12.1 Why multiple processors?

The oldest way to speed up a computer was to raise the clock speed. From the 1980s to the early 2000s, processor clock speeds doubled every few years. This road hit a wall around 2005. Even today the clock speed of desktop processors is between 3 and 5 GHz. Semiconductor makers continue to fit more transistors onto a chip every year. But these transistors are no longer spent on a single faster core; they are spent on more cores. This section describes the three basic organizations of parallel systems and the limit on the gain that parallelism brings.

12.1.1 Three parallel organizations

There are three main ways to put several processors to work together (Figure 12.1). In the first, all processors are attached to a single shared memory. Every processor can read and write every memory word with ordinary `LOAD` and `STORE` instructions. This is called a **shared-memory multiprocessor**. In the second, each processor has its own memory, and the processors are connected to one another by a fast interconnect. One processor cannot read another's memory directly; data can only be sent as messages. This arrangement is called a **multicomputer** or a cluster. In the third, complete computers, that is, machines with their own disks, peripherals, and operating systems, are linked by a wide-area network such as the Internet. This is a **distributed system**.

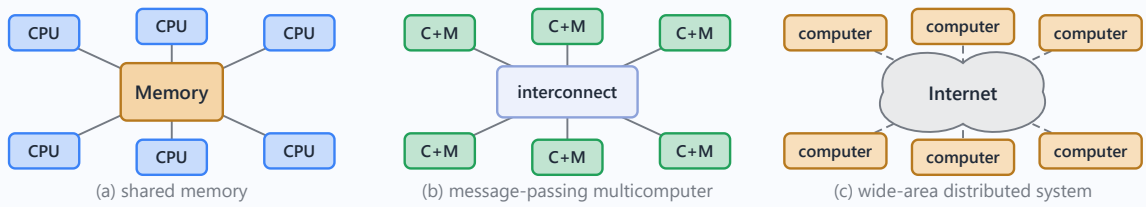


Figure 12.1: Three parallel organizations. (a) A shared-memory multiprocessor, (b) a message-passing multicomputer (C+M: CPU and memory), (c) a wide-area distributed system.

The difference among the three is how tightly they are coupled. In a multiprocessor, communication is a memory access at the nanosecond level. In a multicomputer, a message takes a few microseconds. In a distributed system, messages take milliseconds. As the coupling loosens, the system can be scaled up, but it becomes harder to program and manage.

Shared-memory multiprocessors themselves divide in two. In a **symmetric multiprocessor** (SMP), all processors have equal rights to memory and to input/output (I/O) devices. Kernel code can run on any processor. In an **asymmetric multiprocessor** (AMP), the processors are not equal. Either only one of them runs the operating system, or each processor runs separate software. All of today's desktop and server operating systems are SMP. The asymmetric arrangement is met mostly in embedded chips.

A multiprocessor puts new jobs on the operating system. Which thread will run on which processor? What happens if two processors touch the same kernel table at the same time? How is data carried between processors? How are the copies of shared data kept consistent? The rest of this chapter answers these questions.

12.1.2 The clock-speed limit and Amdahl's law

To understand why clock speed stopped, we have to look at power consumption. For many years, as transistors shrank, the operating voltage fell too. So power per unit area stayed constant. This observation is called **Dennard scaling**. In the mid-2000s the voltage could no longer be lowered. Raising the clock speed now meant overheating the chip. Manufacturers began to spend the transistor budget on more cores.

Increasing the number of cores does not speed up every program in the same proportion. Part of a program can only be executed sequentially: reading input, combining results, protecting a shared structure with a lock. This sequential part finishes in the same time no matter how many cores there are.

Theorem 12.1 (Amdahl's law)

If a fraction p of a program's running time on a single processor can be parallelized and the fraction 1 − p stays sequential, then the speedup on n processors is

> S(n) = 1 / ((1 − p) + p / n)

As n goes to infinity, p / n approaches zero; the speedup can never exceed 1 / (1 − p).

It is easy to see where the formula comes from. Let the time on a single processor be 1 unit. The sequential part takes 1 − p on n processors too. If the parallel part is divided evenly among n processors, it takes p / n. The new time is the sum of the two; the speedup is the ratio of the old time to the new time.

Example 12.1

95% of a program can be parallelized (p = 0.95). What is the speedup on 8 and on 64 processors? What is the maximum speedup possible? How many processors are needed for a 10-fold speedup?

1. **n = 8.** The parallel part takes 0.95 / 8 = 0.11875. The total time is 0.05 + 0.11875 = 0.16875. The speedup is 1 / 0.16875 ≈ 5.93. Eight processors give a gain of about 6 instead of the expected 8.
2. **n = 64.** 0.05 + 0.95 / 64 = 0.05 + 0.0148 = 0.0648. The speedup is ≈ 15.42. The number of processors rose eightfold, and the speedup did not even triple.

3. **The limit.** As n grows, the time approaches 0.05. The speedup cannot exceed 1 / 0.05 = 20. Even on 1024 processors the speedup is ≈ 19.64.
4. **For 10×.** We need 0.05 + 0.95 / n = 0.1. From this 0.95 / n = 0.05 and n = 19.

The saying “performance grows as you add processors” is true only if the sequential part is very small. Even a 5% sequential part caps the gain at 20.

Amdahl's law assumes a fixed problem size. In reality, as the number of processors grows, the problem people want to solve grows too: a bigger simulation, more data. In that case the share of the sequential part shrinks, and the gain is greater than Amdahl predicted. This view is called Gustafson's law. Still, Amdahl's lesson holds: as the number of cores grows, shrinking the sequential part of the software becomes more important than adding cores. The operating system's own locks are part of this sequential part. A single kernel lock puts all processors in a queue and limits how far the system scales.

Common mistake

Computing the speedup as p × n. A program that is 95% parallel does not speed up by 0.95 × 64 ≈ 61 times on 64 processors. The sequential part takes the same time at every processor count and stays constant in the denominator; the correct value is about 15.4.

12.2 Multiprocessor hardware

To understand how the operating system should behave on a multiprocessor, we first need to know how the hardware is built. This section describes the ways of connecting processors to memory, why memory access time is not the same everywhere, and how caches are kept coherent.

12.2.1 UMA: the bus-based architecture

Definition 12.1

In a **UMA** (Uniform Memory Access) multiprocessor, every CPU reaches every memory word in the same time. In a **NUMA** (Non-Uniform Memory Access) multiprocessor, on the other hand, the access time varies with which memory the word is attached to.

UMA is the most comfortable arrangement for the programmer. Where the data is placed does not change performance; a program can be written for a multiprocessor just as it is written for a single processor. The simplest UMA machine attaches all processors and the memory to a single **bus** (Figure

12.2 (a)). When a CPU wants to read from memory, it first checks whether the bus is free. If it is free, it puts the address on the bus, asserts a few control signals, and waits until memory puts the response on the bus. If the bus is busy, it waits for it to become free.

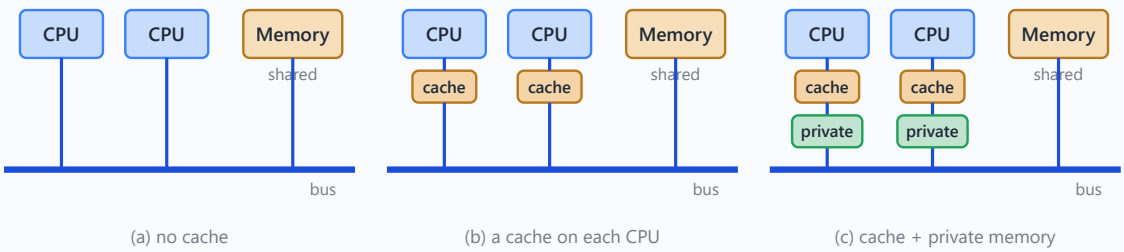


Figure 12.2: Bus-based UMA multiprocessors: (a) without caches, (b) a cache on each CPU, (c) caches and private memory.

This arrangement works well with two or three processors. As the number of processors grows, the bus saturates. With thirty-two or sixty-four processors, the CPUs spend most of their time waiting for the bus; adding a processor does not speed the system up. In arrangement (b), a cache is added to each CPU. Most reads are served from the local cache and do not go out on the bus; traffic drops sharply. The price is a new problem: a copy of the same memory line can exist in several caches. If one CPU changes its own copy, the other copies become stale. Keeping these copies coherent is the job of the hardware, and it is described in detail in later sections.

In arrangement (c), each CPU also has **private memory**. The compiler puts the program's code, constants, stack, and local variables in this private memory. Shared memory is used only for variables that are truly shared. This reduces bus traffic still further. But the compiler has to know which data is shared, and that makes programming harder.

In today's chips there is no longer a single shared bus. Cores, cache slices, and memory controllers are connected inside the chip by a ring or mesh network. Intel's “mesh” and AMD's Infinity Fabric are examples. The idea is still the same: a common communication medium, and large caches to protect that medium's limited bandwidth.

12.2.2 The crossbar switch

The most direct way out of the single bus's bottleneck is to connect every CPU to every memory module by a separate path. The **crossbar switch** does this with a grid (Figure 12.3). The horizontal lines represent the CPUs and the vertical lines the memory modules. At every crosspoint there is a small electronic switch. When the switch closes, a direct path is established between that CPU and that memory module.

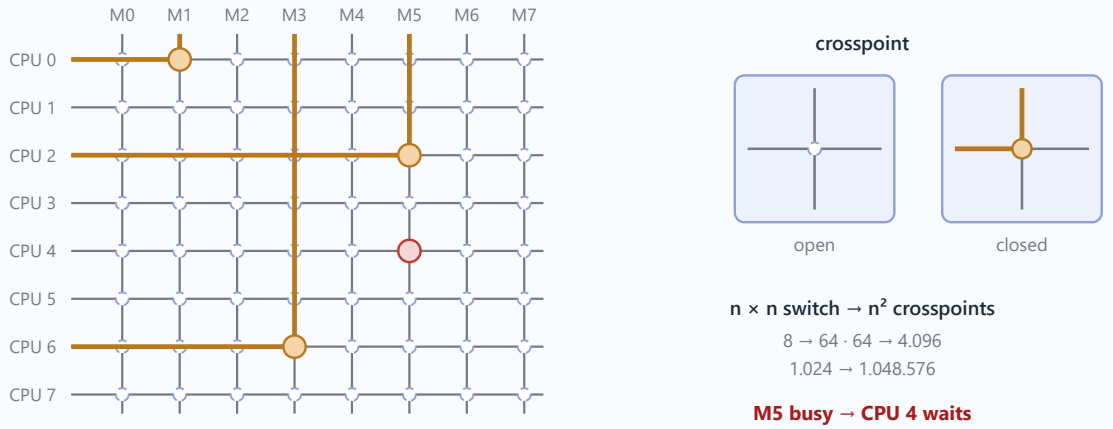


Figure 12.3: An 8 × 8 crossbar. The connections CPU 0 → M1, CPU 2 → M5, and CPU 6 → M3 are established at the same time; because M5 is busy, CPU 4 waits.

In the figure, CPU 2 first wants to access M5: the crosspoint (2, 5) closes and the path is established. Then the connections CPU 0 → M1 and CPU 6 → M3 are established at the same time. Requests going to different memory modules never wait for one another. For this reason the crossbar is a **nonblocking** network. If CPU 4 also wants M5, there is a path in the network, but the module itself is busy. The only point of contention is the memory module, not the network.

The cost of the crossbar is its size. For n CPUs and n memory modules, n^2 crosspoints are needed. An 8 × 8 switch has 64 crosspoints; a 64 × 64 one has 4096, and a 1024 × 1024 one has 1,048,576. That is why large crossbars are expensive. The old Sun Fire E25K server used an 18 × 18 crossbar. Today crossbars are used mostly inside the chip, at small scale, between the cores and the L3 cache slices.

12.2.3 The omega network

To escape the n^2 cost, the network can be built in stages from small switches. The building block is a 2 × 2 switch: it has two inputs and two outputs. A message arriving at one input is routed to one of the two outputs. The message carries four fields: the destination memory module (Module), the address within the module, the opcode (read or write), and the value to write. The switch looks only at the Module field (Figure 12.4).

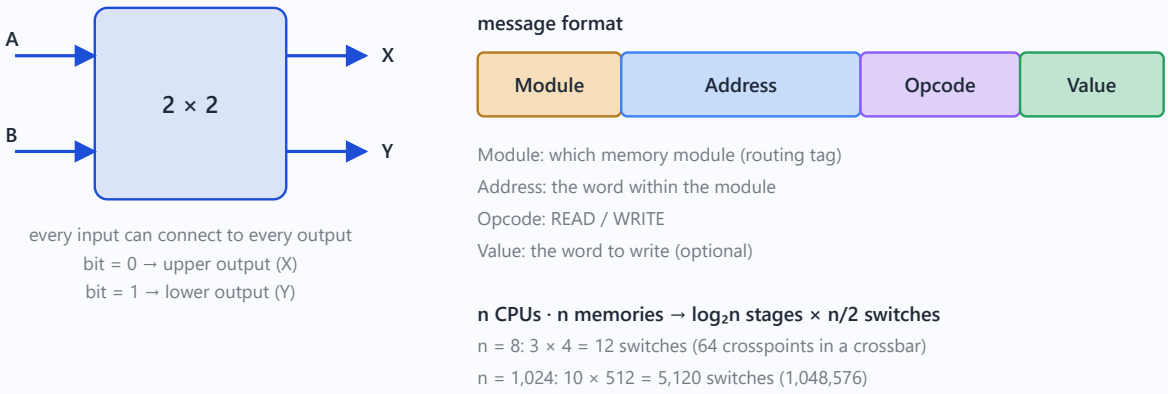


Figure 12.4: A 2 × 2 switch and the fields of a message. The switch makes its routing decision from the Module field alone.

The **omega network** arranges these switches in $\log_2 n$ stages. For 8 CPUs and 8 memories, 3 stages are needed; each stage has four switches (1A–1D, 2A–2D, 3A–3D). The wiring between stages is the **perfect shuffle**. The number of an output is rotated left by one bit to find the number of the input in the next stage. For example, line 011 is connected to input 110.

The routing rule is very simple. The switch in stage k looks at the k-th bit of the destination address from the left. If the bit is 0, it sends the message to the upper output; if 1, to the lower output. The source address plays no part in routing.

Example 12.2

In an 8 × 8 omega network, CPU 011 wants to read memory 110. Which switches does the message pass through?

1. **Arriving at the input.** Line 011 is connected by the shuffle to input 110. This input belongs to switch 1D (the inputs are 110 and 111).
2. **Stage 1.** 1D looks at bit 1 of the destination: 1. The message leaves by the lower output. The number of this output is 111.
3. **Stage 2.** 111 goes through the shuffle to 111 again; the message enters 2D. Bit 2 of the destination is 1; the message leaves by the lower output of 2D.
4. **Stage 3.** The message enters 3D. Bit 3 of the destination is 0; the message leaves by the upper output and reaches memory 110.

The path is 1D (lower) → 2D (lower) → 3D (upper). (These paths and the conflict below were also verified with a short Python simulation that applies the shuffle rule.)

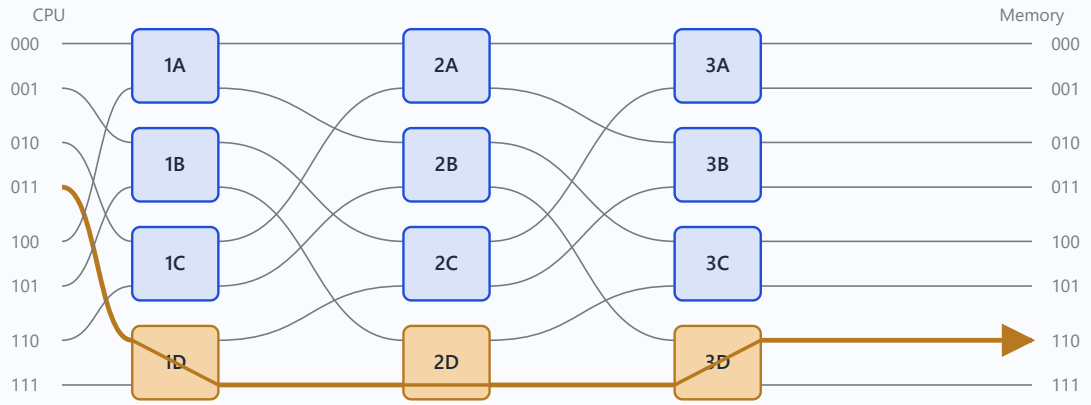


Figure 12.5: The request from CPU 011 to memory 110. Each stage uses one bit of the destination address.

The response returns through the same network. For this, each switch, as it passes the message on, writes into the Module field which input it received it from, that is, one bit of the source address. By the time the message reaches memory, the Module field has accumulated the source address, 011. Memory sends the response back with this tag, and the response reaches CPU 011 by the same rule.

An omega network can carry many transfers at the same time, but not always. Suppose that at the same moment CPU 001 wants to go to memory 111. This request leaves 1B by the lower output and, after the shuffle, enters 2D. Since bit 2 of the destination is 1, it too wants the lower output of 2D. The two requests land on the same output and one has to wait (Figure 12.6). This is why the omega network is a **blocking** network.

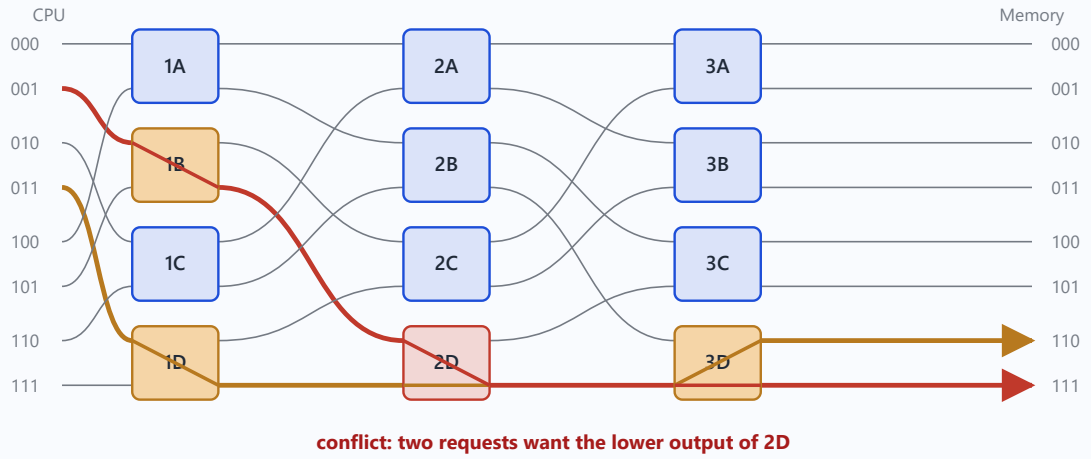


Figure 12.6: The requests 011 → 110 and 001 → 111 conflict at the lower output of switch 2D; one waits.

An omega network uses $n / 2 \times \log_2 n$ switches. With 8 inputs, $4 \times 3 = 12$ switches are needed; with 64 inputs, $32 \times 6 = 192$; with 1024 inputs, $512 \times 10 = 5120$ switches. Compared with the more than one million crosspoints of a 1024×1024 crossbar, this is very little hardware. The price is twofold:

requests wait when they conflict, and latency grows because every request passes through $\log_2 n$ stages.

n	Crossbar: n^2 crosspoints	Omega: $(n/2)\cdot\log_2 n$ switches	Omega stages
8	64	12	3
64	4.096	192	6
1.024	1.048.576	5.120	10

Table 12.1: The hardware cost of the crossbar and the omega network.

Common mistake

Doing the routing in an omega network with the bits of the source address. On the way out, the switches look only at the destination address; the bits of the source address are recorded along the path only so that the response can find its way back. Another mistake is to think the omega network is nonblocking: if two requests want the same switch output, one waits.

12.2.4 NUMA multiprocessors

UMA machines built with a bus or a switch network cannot be scaled beyond a few dozen processors. For machines with hundreds of processors, a compromise is needed: the wish to reach every memory word in the same time is given up. These machines are called NUMA multiprocessors (Definition 12.1).

In a NUMA machine, memory is divided into nodes. Each node has a few CPUs and a memory close to them. In today's two-socket server, each socket is a node (Figure 12.7). The socket's cores read from the DRAM attached to their own memory controller in about 80 ns. The memory of the other socket is reached over the inter-socket link (UPI on Intel, Infinity Fabric on AMD), in 140 ns for example. These latencies are example values; real values vary from machine to machine.

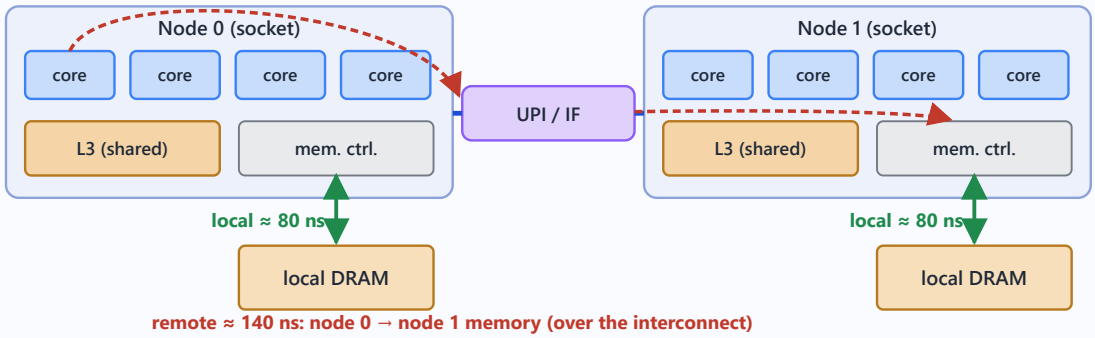


Figure 12.7: A two-node NUMA server. Each socket reaches its own memory fast and the other socket's memory more slowly over the interconnect.

NUMA machines have three common properties. First, all CPUs see a single physical address space. Second, remote memory is also reached with ordinary `LOAD` and `STORE` instructions; the program does not send messages. Third, remote access is slower than local access. NUMA machines divide in two according to whether they have cache coherence. Those without it are called **NC-NUMA** and those with it **CC-NUMA** (cache-coherent NUMA). All of today's servers are CC-NUMA, and they mostly maintain coherence with a directory.

For the operating system, NUMA means a new responsibility: keeping data next to the processor that uses it. The average access time depends on what fraction of accesses are local.

> average access = $f \cdot t_{\text{local}} + (1 - f) \cdot t_{\text{remote}}$

Here f is the local fraction of accesses.

Example 12.3

Local access takes 80 ns and remote access 140 ns. What is the average access time when 90% and when 50% of accesses are local?

1. **f = 0.9.** $0.9 \cdot 80 + 0.1 \cdot 140 = 72 + 14 = 86$ ns.
2. **f = 0.5.** $0.5 \cdot 80 + 0.5 \cdot 140 = 40 + 70 = 110$ ns.
3. **Comparison.** $110 / 86 \approx 1.28$. When the local access ratio falls from 90% to 50%, the program slows down by about 28%.

To avoid this loss, the operating system does three things. The first is the **first-touch** policy: a page is allocated from the memory of the node whose CPU uses it first. Second, the scheduler runs a thread, as far as possible, on the node where its data lives. Load balancing is done first within a node, and moves across nodes only if necessary. Third, Linux's **automatic NUMA balancing** samples which node accesses which pages of a process. It moves pages next to processes or processes next to pages. On the application side, too, a process and its memory can be bound to a particular node with the `numactl` tool.

NUMA is not specific to multi-socket servers. Large single-socket chips can be NUMA internally too. For example, the NPS (nodes per socket) setting on AMD EPYC and SNC (sub-NUMA clustering) on Intel split a socket into several NUMA nodes.

12.2.5 Directory-based CC-NUMA

With a small number of processors, caches can be kept coherent by listening to a common bus. With hundreds of nodes, broadcasting every request to all nodes chokes the network. The solution is to keep a record of which caches hold each memory line. This record is called a **directory**. For a given line, messages are sent only to the nodes holding a copy of that line.

Tanenbaum's example makes the idea concrete. The system has 256 nodes. Each node has a CPU and 16 MiB of memory; the total memory is 256 × 16 MiB = 2^{32} bytes. Cache lines are 64 bytes. A 32-bit physical address splits into three fields (Figure 12.8):

- the leftmost 8 bits are the **node** number (256 nodes);
- the next 18 bits are the **block** (line) number within that node's memory: 16 MiB / 64 B = 2^{18} lines;
- the last 6 bits are the **offset** within the line (64 bytes).

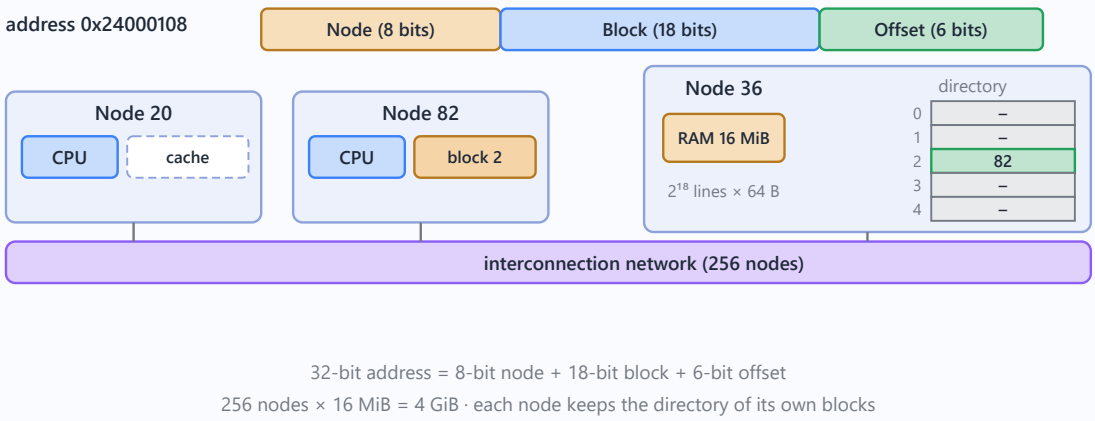


Figure 12.8: A 256-node directory-based CC-NUMA. Each node keeps the directory of the lines in its own memory.

Each node keeps a directory entry for each of the 2^{18} lines in its own memory. In this simple design, if the line is currently in a cache, the entry stores the node number of that cache. A line can be cached in only one node at a time.

Example 12.4

The CPU at node 20 executes `LOAD 0x24000108` and then `LOAD 0x24000080`. Initially, in node 36's directory, entry 2 holds 82 and entry 4 is empty. What happens at each request?

1. **Decoding the address.** The leftmost byte of 0x24000108 is 0x24 = 36: the line is in node 36's memory. The remaining 24 bits are 0x000108 = 264; 264 = 4 · 64 + 8, so block 4, offset 8.
2. **Request.** The address is not local. Node 20's memory controller sends node 36 the message “I want block 4”.
3. **Directory.** Node 36 looks at its directory: entry 4 is empty, so the line is in no cache. The current copy is in node 36's own RAM.
4. **Reply.** Node 36 sends the line to node 20 and writes 20 in entry 4: “a copy of this line is at 20”.

5. **Second request.** In 0x24000080 the node is again 36; since 0x80 = 128 = 2 · 64 + 0, it is block 2, offset 0. In the directory, entry 2 says 82: the line is in node 82's cache.
6. **Invalidation.** Node 36 sends node 82 the message “invalidate the line”. If the line has been modified at 82, it is written back first. Then the line is sent to node 20 and 20 is written in entry 2.

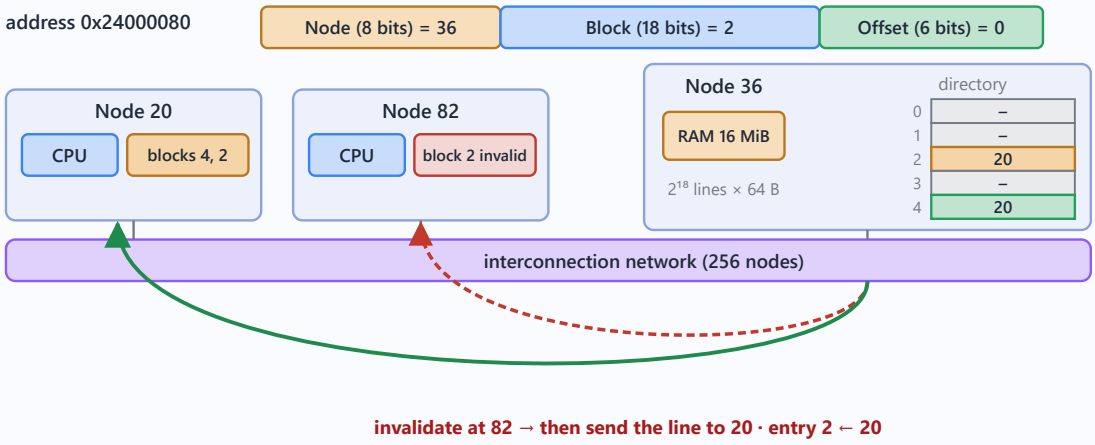


Figure 12.9: The request for 0x24000080. Node 36 first invalidates the copy at 82, then sends the line to 20; directory entry 2 now points to 20.

All this communication is done with point-to-point messages. No bus has to be listened to by all the caches. The cost of the directory is memory. If each entry holds an 8-bit node number and a valid bit, it comes to 9 bits. A node's directory is $2^{18} \times 9$ bits = 294,912 bytes. This is an overhead of 9 bits for every 64-byte line, that is, about 1.76% of memory.

It is not realistic for a line to be cacheable at only one node; a line that is read often should have many copies. Real systems keep the copy set in a **bit vector** with one bit per node, or in a list of a limited number of pointers. Inside multicore chips, structures called “snoop filters” do the same job: they track which line is in which core's private caches.

12.2.6 The cache coherence problem

Each core has its own private cache, and these caches hold copies of the same memory. Figure 12.10 shows the problem. In memory, x = 5. Both cores have read x and brought it into their caches. Core 1 writes 7 to x. The write changes only the copy in its own cache. If core 0 now reads x, it sees the old value 5 in its own cache. From the program's point of view this is wrong: the last write to x was 7.

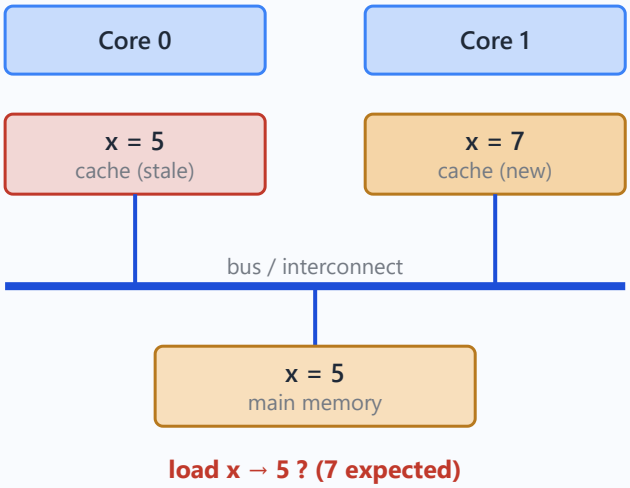


Figure 12.10: Cache incoherence. Core 1 made x 7; core 0 reads the stale 5 in its own cache.

Definition 12.2

Cache coherence is the condition that writes to the same address are seen in the same order by all cores, and that a read returns the value most recently written to that address.

Hardware provides coherence in two ways. In the **snooping** method, all cache controllers listen to the common bus continuously. If someone else is about to write a line that a cache holds a copy of, the cache sees this on the bus and updates or invalidates its copy. This method suits a small number of cores. In the **directory** method, a list of copies is kept and messages go only to the nodes concerned. This method scales to multicore chips and NUMA machines.

There are two basic policies for writes. In the **write-invalidate** policy, a core invalidates the other copies before writing; they ask for the line again on their next read. In the **write-update** policy, the new value is sent to all copies. On repeated writes to the same line, updating produces traffic every time, while invalidation does so only on the first write. That is why nearly all of today's processors use invalidation.

Coherence concerns a single address. In what order writes to different addresses are seen by other cores is a separate matter. This topic is called the **memory consistency model**. The x86 model (x86-TSO) is fairly strong; ARM and RISC-V use weaker models. Lock implementations therefore include memory barriers: they force the writes in the critical section to be visible to everyone before the lock is released.

12.2.7 The MESI protocol

The most common of the write-invalidate protocols is MESI. It takes its name from the initials of the four states a cache line can be in (Table 12.2).

State	Meaning
M (Modified)	The only copy of the line is in this cache and it is dirty: the value in memory is stale. The line is written back to memory when it is evicted from the cache.
E (Exclusive)	The only copy is here and it is clean: it matches memory. A write turns the line into M without going out on the bus.
S (Shared)	Clean copies may be in several caches. To write, the other copies must first be invalidated.
I (Invalid)	The copy is invalid; an access is a cache miss.

Table 12.2: The MESI states.

The state of a line changes with two kinds of events: its own core's reads and writes, and another core's requests seen on the bus. Figure 12.11 shows all the transitions together. Three kinds of requests circulate on the bus. `BusRd` asks to read a line. `BusRdX` asks exclusively, with the intent to write a line; the other copies are invalidated. `BusUpgr` asks only for invalidation so that a line in state S can be written; since the data is already at hand, the line is not transferred.

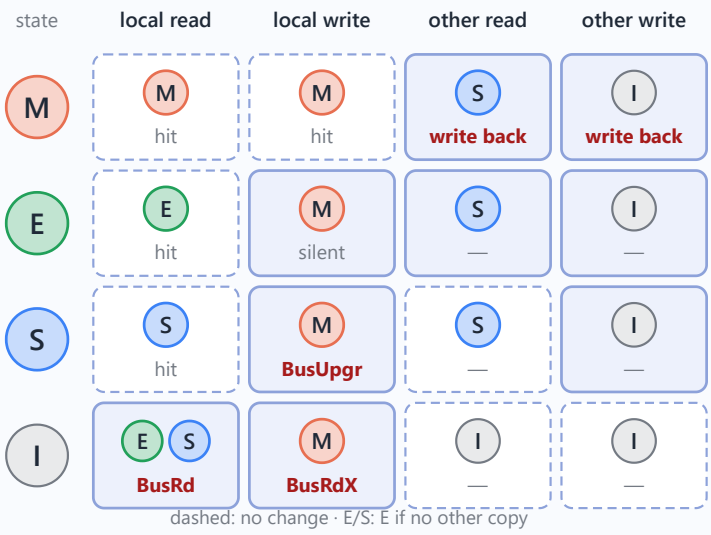


Figure 12.11: The MESI transition table. Rows are the current state, columns are events: local read, local write, another core's read (other read), another core's write (other write). A cell gives the new state and, if needed, the bus operation.

The table reads as follows. If a line in state I is read locally, `BusRd` is sent. If another cache says "I have it", the line becomes S; if nobody does, it becomes E. If a line in state I is to be written, `BusRdX` is sent and the line becomes M. Writing a line in state E is silent: since it is known that there is no other copy, nothing goes out on the bus. Writing a line in state S requires `BusUpgr`. If another core wants to

read a line in state M, its owner puts the data on the bus (*Flush*), also writes it back to memory, and makes the line S. If another core wants to write, the owner again supplies the data and makes the line I.

The reason the E state exists is to make a common case cheaper: a thread first reads its private data and then writes it. Without E, the read would make the line S, and the following write would generate a needless `BusUpgr`. The protocol has commercial derivatives too. In Intel's MESIF, the F (Forward) state decides which of the shared copies answers requests. In AMD's MOESI, the O (Owned) state denotes a line that is dirty but shared; the line can be given to other caches without being written back to memory.

12.2.8 MESI step by step

How the protocol works is best seen on a sequence of operations. In a two-core system, x is initially only in memory and its value is 0. The line of x is in state I in both caches.

Example 12.5

C0 and C1 perform the following operations in turn: C0 reads x, C1 reads x, C1 writes x = 1, C0 reads x, C0 writes x = 2, C0 writes x = 3. After each step, what are the states of the line in the two caches, the value in memory, and the bus message?

Step	Operation	C0	C1	Memory	Bus	Explanation
0	—	I	I	0	—	x only in memory
1	C0 reads	E	I	0	BusRd	no other copy: exclusive and clean
2	C1 reads	S	S	0	BusRd	C0 snoops the request and says “I have it”
3	C1: x = 1	I	M	0	BusUpgr	invalidation to write from S; memory is now stale
4	C0 reads	S	S	1	BusRd, Flush	C1 supplies the dirty line to both C0 and memory
5	C0: x = 2	M	I	1	BusUpgr	invalidation again
6	C0: x = 3	M	I	1	—	already M: no bus traffic

Six operations produce five bus operations. At the end of the sequence the current value of x is 3, and it is only in C0's cache; memory still says 1. (The table was produced with a Python simulation that applies the MESI rules.)

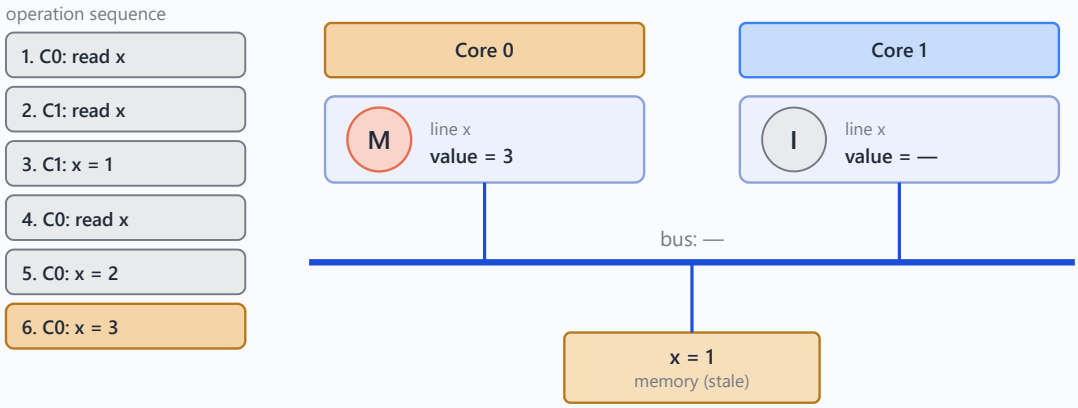


Figure 12.12: The last step of the sequence. Because C0 holds the line in state M, the new write does not go out on the bus; the value in memory is stale.

The last step shows an important result. Repeated writes by one core to data that only it uses are cheap: as long as the line stays in state M, it generates no messages. What is expensive is different cores writing to the same line in turn. Each write moves the line from one cache to another. False sharing in the next section rests on this observation.

Common mistake

Confusing the E and S states. Both are clean, but E means “no other copy”; writing to a line in E silently makes it M. To write to a line in S, the other copies must first be invalidated. Another mistake is to think memory is not updated at step 4: when a line in state M is handed to another cache, it is also written back to memory.

12.2.9 False sharing

Coherence protocols track cache lines, not variables. In today's processors a line is 64 bytes. If two completely independent variables fall on the same line, the hardware treats them as shared data.

Definition 12.3

False sharing is needless coherence traffic caused by different variables used by different cores being in the same cache line.

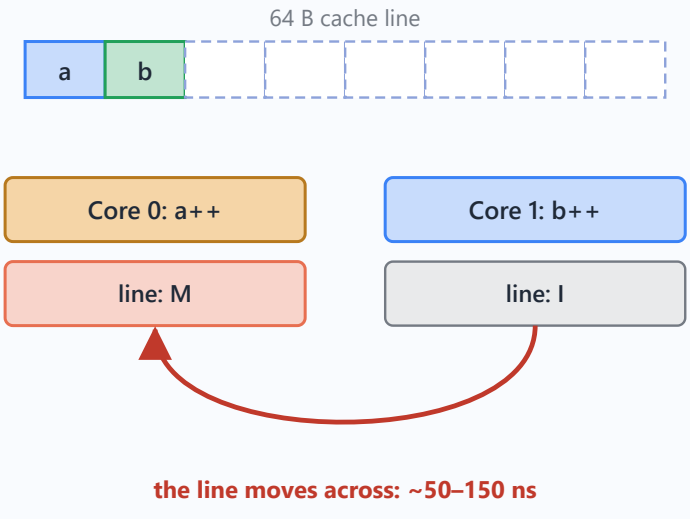
In the code below, two threads increment two separate counters. In logic there is no sharing whatsoever and no lock is needed.

```
struct counters {      /* bad */
  long a;   /* offset 0 */
  long b;   /* offset 8 */
};

struct padded {        /* good */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* thread 0: for(..) s.a++;
   thread 1: for(..) s.b++; */
```

In `struct counters`, `a` is at byte 0 and `b` at byte 8; both fall in the same 64-byte line. Thread 0 runs `s.a++` in a loop on C0, and thread 1 runs `s.b++` on C1. The events unfold as follows (Figure 12.13):

1. C0 does `a++`. It gets the line in state M; C1's copy becomes invalid.
2. C1 does `b++`. The line moves to C1 and becomes M there; the copy in C0 becomes invalid.
3. C0 does `a++` again and pulls the line back.

When both cores write often, back to back, the line goes from one core to the other every time. This is called “ping-pong”. Moving a line takes between tens and a hundred and fifty nanoseconds; the increment itself is a few cycles. As a result, the two-thread program can run even slower than the single-thread version doing the same work.



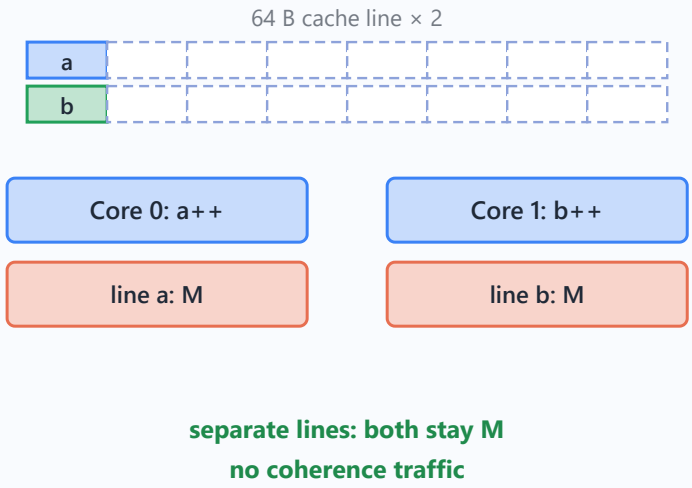


Figure 12.13: At the top, a and b are in the same line: every write moves the line to the other core. At the bottom, the padded structure: each counter is in its own line, both stay in state M, and no coherence traffic arises.

The fix is to put each counter in a separate line. In `struct padded`, C11's `_Alignas(64)` specifier aligns each field to a 64-byte boundary. The size of the structure rises to 128 bytes and 56 bytes per counter are wasted. In return the ping-pong disappears entirely; the gain is often a many-fold speedup. In Java the same job is done with the `@Contended` annotation, and in the Linux kernel with the `____cacheline_aligned` macro. The kernel's per-CPU counters are likewise kept in separate lines for each CPU for this reason.

Common mistake

Mistaking false sharing for a race condition and adding a mutex. There is no race on separate variables; the results are correct, only the program is slow. Adding a mutex does not solve the problem; on the contrary, the lock's line too starts bouncing between cores. The right fix is to place the data on separate lines.

The same problem appears on a larger scale too. In distributed shared memory the unit of coherence is not a line but a page. If two independent variables are on the same 4 KiB page, the page goes back and forth between machines over the network.

12.2.10 Multicore and heterogeneous chips

Most of today's processors host several independent cores on a single chip. These chips are called **multicore** or chip multiprocessors (CMP) (Figure 12.14). In a typical layout each core has private L1 and L2 caches. The L3 cache is shared and is distributed in slices over the on-chip network. On server sockets the number of cores runs into the hundreds: the AMD EPYC 9005 series has 192 cores and the Intel Xeon 6900E series has 288 efficiency cores.

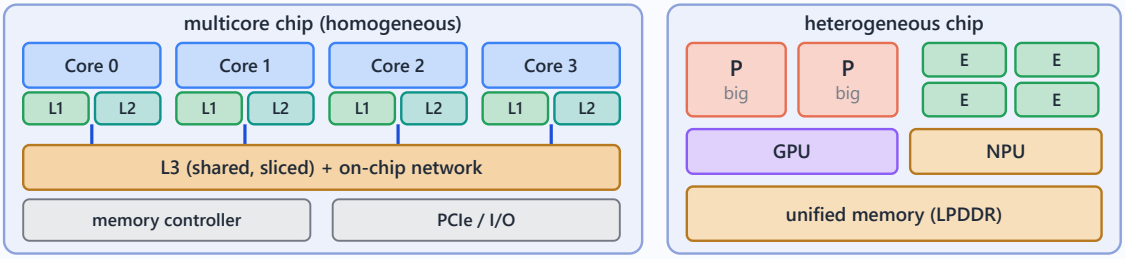


Figure 12.14: On the left, a homogeneous multicore chip: private L1/L2, a shared L3, a memory controller. On the right, a heterogeneous chip with big (P) and small (E) cores, a GPU, and an NPU.

Multicore chips have four properties that concern the operating system:

- **Chiplets:** Large processors are no longer a single piece. Groups of cores are placed on separate dies and combined in one package. Even within a single socket, a core's distance to the L3 and to memory is not the same everywhere. The scheduler tries to keep cooperating threads on the same chiplet.
- **Big and little cores:** ARM's big.LITTLE design, Apple's chips, and Intel's P/E cores combine fast but power-hungry cores with slow but efficient ones. The operating system places work according to energy. On Linux, energy-aware scheduling (EAS) does this job. Intel's Thread Director hardware also gives Windows hints about which thread suits which core.
- **Accelerators:** A GPU and, for AI workloads, an NPU sit on the same chip and share the same memory. These are not scheduled directly; the operating system sends them work through a driver.
- **Shared resources:** The L3 cache and memory bandwidth are common. A memory-intensive job on one core can slow down a job on another core.

12.2.11 Simultaneous multithreading

A superscalar core can execute several instructions every cycle. But a single thread rarely fills these slots. A cache miss or a dependency between instructions makes the core wait, and execution units sit idle (Figure 12.15 (a)).

Definition 12.4

In **simultaneous multithreading** (SMT), a single physical core executes the instructions of two or more hardware threads in the same cycle. The core appears to the operating system as several logical CPUs.

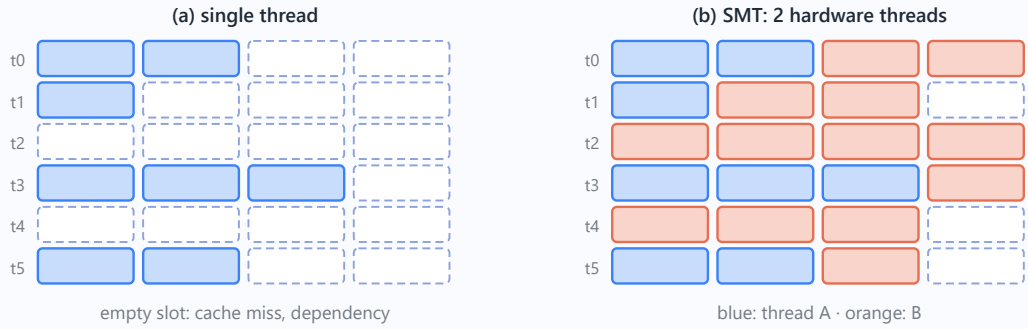


Figure 12.15: (a) Execution slots left empty with a single thread; (b) with SMT, two threads (blue A, orange B) share the same slots.

Intel calls this Hyper-Threading; AMD uses the name SMT. The logical CPUs have separate register sets. The execution units and the L1 and L2 caches, however, are shared. The gain is typically between 10% and 30%. If two threads want the same unit, for example the floating-point unit, there may be no gain at all. For this reason the scheduler places jobs on empty physical cores first. It uses the two logical CPUs of one physical core together only when all physical cores are busy.

SMT has a security side as well. Two threads that share the same core can infer each other's data through the shared caches. Side-channel attacks such as L1TF and MDS exploit this. Linux's **core scheduling** puts only mutually trusting threads on the same physical core. Some vendors have begun to give up SMT altogether; the P cores of Intel's Lunar Lake (2024) chip have no SMT.

12.2.12 GPUs and heterogeneous computing

CPUs and GPUs are designed for different purposes (Figure 12.16). A CPU consists of a small number of large, complex cores; with large caches and mechanisms such as branch prediction, it tries to minimize the latency of a single thread. A GPU contains thousands of simple lanes and maximizes total throughput. The lanes execute the same instruction in groups of 32. NVIDIA calls these groups *warps*, AMD *wavefronts*. This design suits data-parallel jobs in which the same operation is applied to every data element: graphics, AI, and scientific computing.

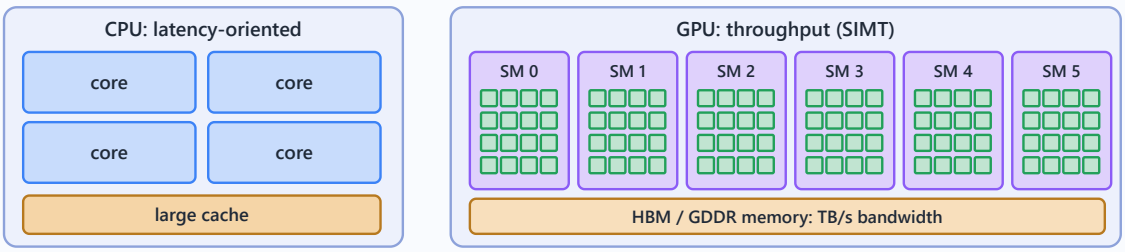


Figure 12.16: A latency-oriented CPU and a throughput-oriented GPU. The GPU's streaming multiprocessors (SMs) are attached to high-bandwidth memory.

GPU programs are written in CUDA, ROCm/HIP, OpenCL, or SYCL. The CPU launches the function that will run on the GPU (this too is called a kernel, not to be confused with the operating system kernel), moves data to GPU memory, and waits for the result. In the meantime the operating system and the driver manage GPU memory and time-share the GPU contexts of different processes. Today's systems also offer a unified or shared virtual memory between CPU and GPU. When the GPU accesses a page, a page fault can occur through the IOMMU and the page is moved to the GPU. In large AI clusters each node has 4–8 GPUs. The GPUs are connected to one another by NVLink and the nodes to one another by InfiniBand or RoCE, and data is moved directly with RDMA.

12.3 Multiprocessor operating system types

Once the hardware connects several processors to the same memory, one has to decide how the operating system will use those processors. Three models have emerged in history. This section describes the three in turn and compares them.

12.3.1 Each CPU has its own operating system

The simplest way is to divide memory into as many partitions as there are CPUs and run an independent copy of the operating system on each CPU in its own partition (Figure 12.17). The operating system's code is read-only and so can be shared; its data is separate in each copy. When a process makes a system call, the call is trapped on its own CPU and that CPU's operating system handles it.

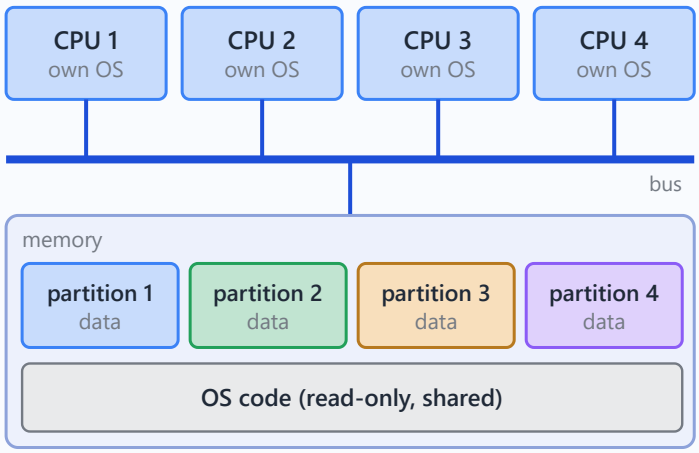


Figure 12.17: Each CPU has its own operating system. Memory is divided into four partitions; the operating system code is shared, its data is separate.

The appeal of this model is its simplicity. A single-processor operating system can be used almost unchanged. Because the partitions are independent of one another, no locks are needed between them either. But the absence of sharing creates four problems:

- Processes are not shared. Each operating system has its own process table. While one CPU is overloaded, another may sit idle; work cannot easily be moved from one to the other.
- Pages are not shared. Memory may run out in one partition while another partition still has free pages.
- Each operating system keeps its own buffer cache. The same disk block can be in two caches in different states. If both copies are modified, the disk block becomes inconsistent. The only way to avoid this problem is to remove the buffer caches, which lowers performance.
- One CPU's operating system does not know what the others are doing; common scheduling cannot be done.

This model lives on today under other names. The hardware partitioning of large servers and the separate virtual machines running on a hypervisor rest on the same idea. In research, the Barrelfish operating system uses it as a deliberate design. In this “multikernel” design, each core is treated as a separate node that communicates by messages.

12.3.2 The master-slave model

In the second model, a single copy of the operating system and all its tables are on one CPU only; this CPU is called the **master** (Figure 12.18). The other CPUs, the **slaves**, run only user processes. When a process on a slave CPU makes a system call, the call is redirected to the master and handled there.

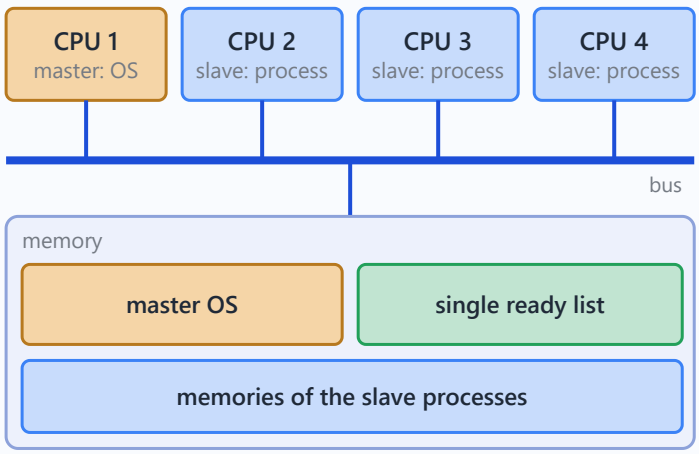


Figure 12.18: The master-slave model. The operating system and the single ready list are on the master; the slaves run user processes.

The master-slave model solves most of the problems of separate operating systems. There is a single ready list. When a slave becomes idle, it asks the master for a new process; the load is thus balanced automatically. Memory pages are handed out from a single pool, and since there is a single buffer cache, no inconsistency arises. Because only the master touches kernel data, no locks are needed either.

The model's weakness is that it does not scale. All system calls are handled on a single CPU. Under a system-call-heavy workload, the master becomes the bottleneck. As the number of slaves grows, calls queue up in front of the master and adding new CPUs gains little. If the master crashes, the whole system stops too. For this reason the master-slave model is useful in systems with few CPUs.

This is the classic form of asymmetric multiprocessing (AMP). Today the asymmetric arrangement is met mostly in embedded chips. For example, next to the Cortex-A cores running Linux there is a Cortex-M core running a real-time operating system. Linux starts this core through the `remoteproc` infrastructure and communicates with it over OpenAMP.

12.3.3 Symmetric multiprocessors

The third model removes the master's bottleneck. There is a single copy of the operating system in memory, but every CPU can run it (Figure 12.19). A system call is handled on the CPU that makes it. Load and memory are balanced dynamically, as in master-slave. This model is called **symmetric multiprocessing** (SMP).

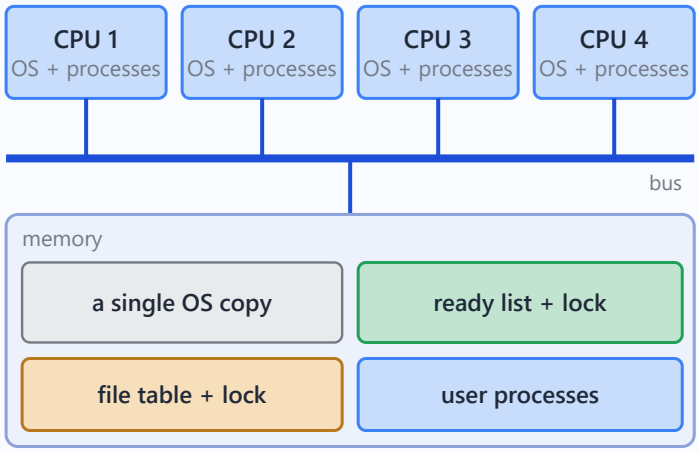


Figure 12.19: The SMP model. All CPUs run the single copy of the operating system; shared structures such as the ready list and the file table are protected by locks.

The price of SMP is concurrency. Two CPUs can touch the same kernel table at the same time: one adds a process to the ready list while another removes a process from the same list. To prevent these races, the kernel's shared structures must be protected with locks. The easiest first step is to protect the whole kernel with a single lock. A CPU takes this lock when entering the kernel and releases it when leaving. Linux 2.0 started out exactly this way; this lock is called the “Big Kernel Lock” (BKL). A kernel with a single lock is really a somewhat more flexible form of the master-slave model: only one CPU can be in the kernel at a time.

To scale, the kernel is divided into mutually independent critical regions, and each region is protected by its own lock. Linux whittled the BKL down step by step over more than a decade and removed it entirely in version 2.6.39 (2011). Today the kernel has thousands of fine-grained locks. Alongside

them are used RCU (read-copy-update), a mechanism in which readers take no lock at all, and per-CPU data structures, in which each CPU touches only its own copy.

Fine-grained locks have their own danger. If one code path takes locks A and B in that order and another path takes them in reverse order, two CPUs can wait for each other forever. To prevent this, kernel developers give the locks an order and require every code path to take locks in the same order. Linux's `lockdep` tool catches violations of this rule at run time.

12.3.4 Comparing the three models

Table 12.3 puts the three models side by side.

	Separate OS	Master-slave	SMP
OS copies	one per CPU	one (on the master)	one (everyone runs it)
Where is a system call handled?	local CPU	master CPU	local CPU
Ready list	per CPU	single, on the master	single or per CPU + balancing
Load balancing	none	good	good
Scalability	good but no sharing	the master is a bottleneck	depends on lock design, best
Complexity	lowest	low	highest
Today	virtual machines, partitioning	embedded AMP, legacy systems	Linux, Windows, macOS, BSD

Table 12.3: Comparison of multiprocessor operating system models.

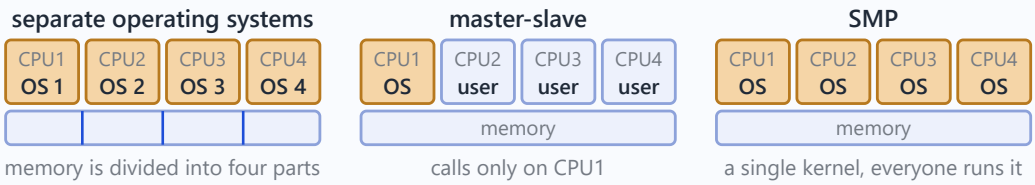


Figure 12.20: The three models together: with separate operating systems, memory is divided into four; in master-slave, calls are handled only on CPU1; in SMP, everyone runs the single kernel.

These three models are for shared-memory multiprocessors. Its difference from multicomputer operating systems must be kept in mind. In a multiprocessor, all CPUs see the same physical memory. In a multicomputer, each node has its own memory and its own copy of the operating system; the nodes communicate only by messages.

Common mistake

Attributing problems to the wrong model. Inconsistent buffer caches and unbalanced load are problems of the "each CPU has its own operating system" model. The master being a bottleneck is a problem of the master-slave model, and races in kernel tables and lock-order bugs are problems of SMP.

12.4 Synchronization

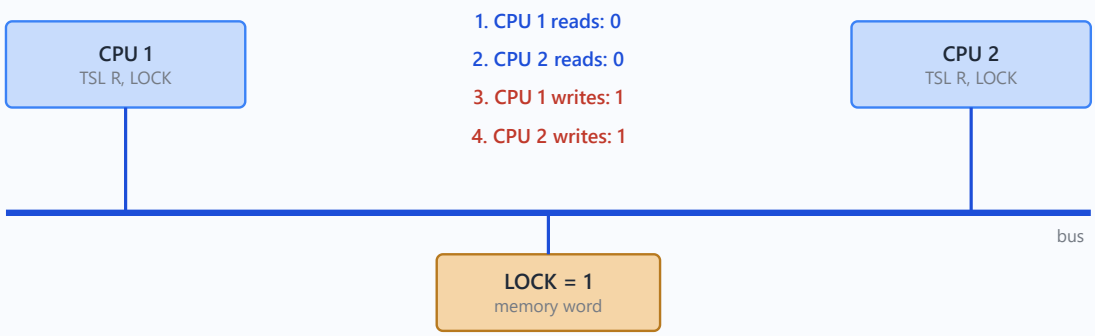
An SMP kernel protects its shared structures with locks. On a single processor, a lock can be implemented easily by disabling interrupts or with an indivisible instruction. On a multiprocessor, both approaches raise new problems. This section describes the hardware foundation of the lock, more scalable kinds of locks, and the choice between spinning and sleeping while waiting.

12.4.1 TSL and bus locking

The **TSL** (test and set lock) instruction reads a memory word, puts its old value in a register, and writes 1 to the word. On a single processor this instruction is indivisible: no interrupt arrives in the middle of the instruction. If the lock word is 0 the lock is free, and if 1 it is held. If TSL returns 0 as the old value, the lock has been acquired.

On a multiprocessor, the indivisibility of the instruction alone is not enough. The read and the write of TSL are two separate bus cycles. Another CPU can slip in between these two cycles (Figure 12.21):

1. CPU 1 reads the lock word: 0, that is, free.
2. Before CPU 1 writes, CPU 2 reads too: again 0.
3. CPU 1 writes 1 to the word; CPU 2 writes 1 too.
4. Both think "the lock is mine" and mutual exclusion is broken.



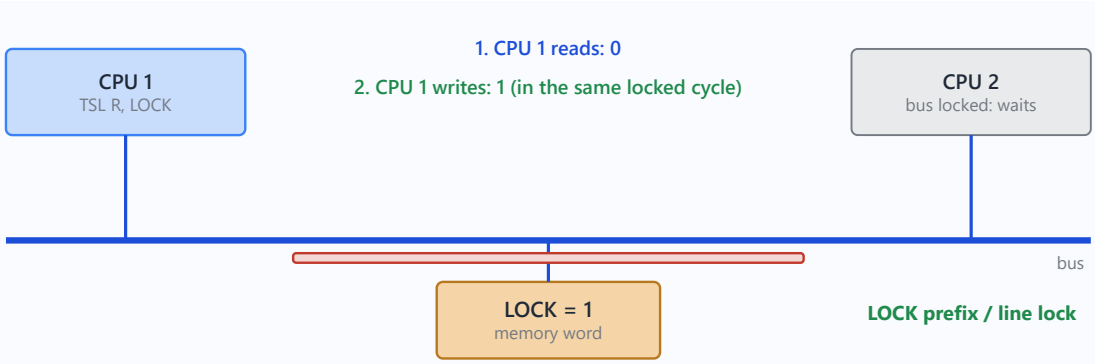


Figure 12.21: At the top, TSL without locking the bus: both CPUs read 0 and both acquire the lock. At the bottom, the fix: TSL locks the bus (or the cache line) for the duration of the read and the write; CPU 2 waits.

Disabling interrupts does not help either. Disabling interrupts protects only the CPU that executes the instruction; the other CPUs keep accessing memory. The fix is in the hardware: TSL locks the bus for the duration of the read and the write. During this time no other CPU can access memory. On today's x86 processors, the `LOCK` prefix mostly locks not the whole bus but only the cache line concerned. The line is held in state M for the duration of the operation and the requests of other cores are held off. ARM does the same job in two ways: the `LDXR` / `STXR` instruction pair (if someone else wrote in the meantime, the write fails and the loop repeats) or single-instruction atomic operations such as `CAS` in the LSE extension.

12.4.2 The ticket lock

A correctly working TSL lock still has two shortcomings. First, it is not fair: when the lock becomes free, which of the waiters takes it is a matter of chance, and one can keep losing and starve. Second, every attempt does a write and generates cache traffic. The **ticket lock** solves the first problem completely and the second to a large extent.

The idea is the take-a-number machine at a bank. There are two counters: `next` hands out the next ticket, and `serving` shows whose turn it is. A thread that wants the lock atomically increments `next` and takes the old value as its own ticket. Then it waits until `serving` equals its ticket. The thread that releases the lock increments `serving` by one. The Java program in the slides shows two threads protecting a shared counter with this lock:

```
import java.util.concurrent.atomic.*;
public class Ticket {
  static AtomicInteger next = new AtomicInteger();
  static volatile int serving = 0;
  static int counter = 0;
  static void lock() {
    int my = next.getAndIncrement();
    while (serving != my) Thread.onSpinWait();
  }
  static void unlock() { serving++; }
  public static void main(String[] a)
      throws Exception {
    Runnable job = () -> {
      for (int i = 0; i < 100_000; i++) {
        lock(); counter++; unlock(); } };
    Thread t0 = new Thread(job);
    Thread t1 = new Thread(job);
    t0.start(); t1.start(); t0.join(); t1.join();
    System.out.println("counter = " + counter);
  }
}
```

The key lines of the code are these. `next` is an `AtomicInteger`; `getAndIncrement()` increments the counter indivisibly and returns the old value. On x86 this is a single `LOCK XADD` instruction. Even if two threads call it at the same time, they get different tickets. `serving` is `volatile`. Every read sees the real value. The write in `unlock()` also makes the writes in the critical section visible to everyone. `Thread.onSpinWait()` gives the processor the hint "I am spinning"; on x86 it becomes the `PAUSE` instruction and reduces power consumption. The `serving++` in `unlock()` need not be atomic, because only the lock's owner writes to `serving`.

Example 12.6

T0 and T1 call `lock()` in turn; T0 releases when it finishes the critical section. How do the counters and the state of the threads change?

Step	Event	next	serving	T0.my	T1.my	State
0	start	0	0	—	—	lock free
1	T0 takes a ticket	1	0	0	—	serving == 0: T0 enters, `counter++`
2	T1 takes a ticket	2	0	0	1	serving ≠ 1: T1 spins
3	T1 spins	2	0	0	1	reads serving from the S copy in its own cache
4	T0 releases	2	1	—	1	the write invalidates T1's copy
5	T1 reads again	2	1	—	1	1 == my: T1 enters

At the end of 2 × 100,000 increments, the program prints `counter = 200000`. The program was run five times with JDK 21 and printed 200000 each time. When the lock calls were removed and only `counter++` was left, six runs gave values between 118,108 and 151,847: the increments overwrote one another.

Figure 12.22: At the end of the program the two counters equal 200000; every ticket handed out has been served.

The ticket lock is **fair**: threads enter the lock in ticket order, that is, in arrival order (FIFO). No thread starves. The waiters only read: they read `serving` from the S copy in their own caches and do not occupy the bus. When the lock is released, a single coherence message invalidates all the copies. It still has one weakness. After the release, all waiters read the line again, but only one can enter. With many waiters, this wave of reads is expensive.

12.4.3 Spinlocks and cache traffic

Locks that spin in a loop while waiting for the lock are called **spinlocks**. On a multiprocessor, the cost of a spinlock is the cache traffic it generates while waiting. Figure 12.23 compares three designs.

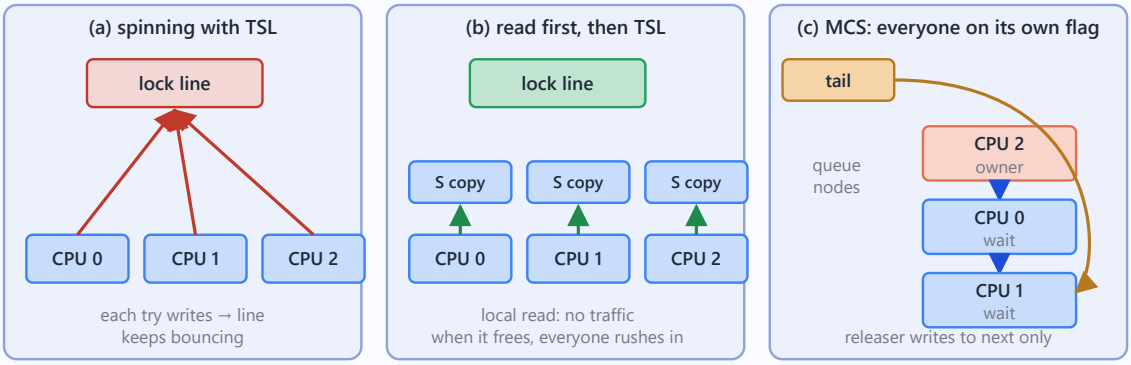


Figure 12.23: (a) A pure TSL loop: every attempt is a write, and the lock line bounces between CPUs. (b) Read first, then TSL: waiters read their own S copies. (c) MCS: each waiter spins on its own flag.

(a) A pure TSL loop. The waiting CPU executes TSL nonstop. Every TSL is a write: it pulls the line in state M and invalidates the other copies. The lock line is moved constantly among the waiters. When the lock's owner wants to access this line, it too gets in the queue. As a result the waiters slow down the lock's owner, and the release of the lock is delayed too.

(b) Read first, then TSL. This method is called TTAS (test-and-test-and-set). The waiting CPU first tries the lock with an ordinary read. If the lock is held, the read is served from the copy in its own cache and nothing goes out on the bus. When the lock looks free, it tries TSL. In addition, **exponential backoff** can be used: after a failed TSL, the CPU waits a while and doubles this time at each failure. This reduces the waiters rushing in at the same moment.

(c) The MCS lock. In the lock proposed by Mellor-Crummey and Scott in 1991, the waiters form a linked list. Each waiter spins on a flag in its own node, and each node is in a separate cache line. The lock is managed with a single `tail` pointer that points to the end of the list. The releaser writes to the flag of the next in the list. This is a single remote write and wakes only the next CPU. MCS is both fair and generates constant traffic independent of the number of waiters. Linux's current `qspinlock` is based on this idea.

None of these designs is made to prevent data inconsistency; the pure TSL lock works correctly too. The aim is to reduce coherence traffic, that is, lines being moved needlessly between caches.

12.4.4 Code trace: the TTAS lock

The implementation of the TTAS lock in C is short. The call `tsl(k)` stands for the hardware instruction that indivisibly writes 1 to `*k` and returns the old value.

```
void lock(int *k) {
  for (;;) {
    while (*k == 1)   /* spin locally */
      ;
    if (tsl(k) == 0)  /* looked free */
      return;         /* acquired */
  }
}
void unlock(int *k) {
  *k = 0;
}
```

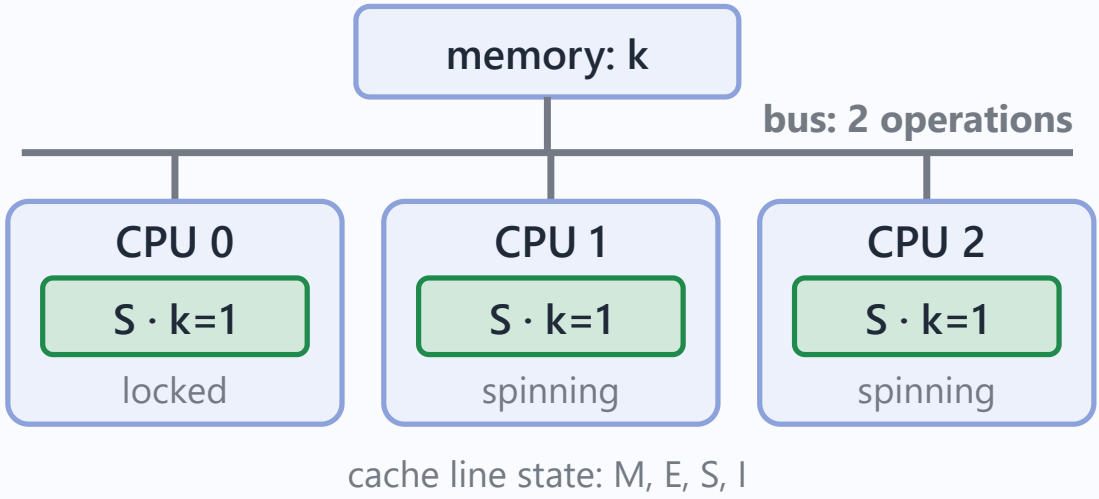
`lock()` is an endless loop. The inner `while` spins as long as the lock is 1; this is a read, and if the copy is in the cache it does not go out on the bus. When the lock looks 0, `tsl()` is tried. If the old value is 0, the lock has been acquired and the function returns. If another CPU slipped in and took the lock first, the old value is 1; the loop goes back to the local read. `unlock()` writes 0 to the lock word.

Example 12.7

CPU 0 holds the lock (k = 1, the line is in CPU 0's cache in state M). CPU 1 and CPU 2 call `lock()`. CPU 0 releases the lock and the two waiters race. How many bus operations occur?

Step	Event	CPU 0	CPU 1	CPU 2	Bus operation (total)
0	start	M, k=1, lock held	I	I	0
1	CPU 1 reads `*k` (miss)	S	S, k=1	I	1
2	CPU 2 reads (miss)	S	S	S, k=1	2
3	both spin locally	S	S	S	2
4	CPU 0 `unlock()` : k = 0	M, k=0	I	I	3
5	CPU 1 reads (miss), sees k = 0	S	S	I	4
6	CPU 2 reads (miss), sees k = 0	S	S	S	5
7	CPU 1 `tsl` : old value 0, lock acquired	I	M, k=1	I	6
8	CPU 2 `tsl` : old value 1, fails	I	I	M, k=1	7
9	CPU 2 spins locally	I	I	M	7

Seven bus operations occur in total. No matter how long the spinning in step 3 lasts, this number does not grow; in a pure TSL loop, every attempt would generate one more operation. (The states were verified with the same MESI simulation.)



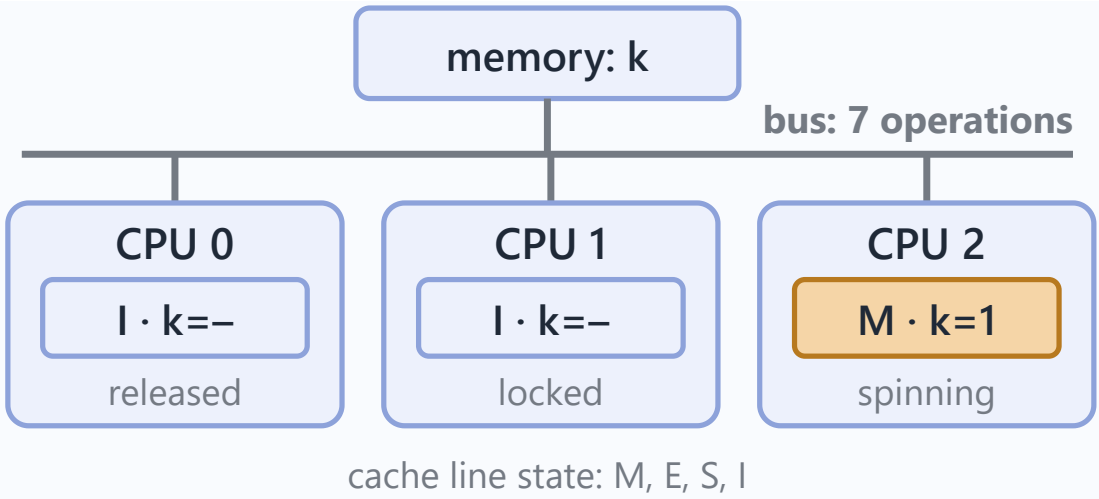


Figure 12.24: At the top, the end of step 2: three S copies, 2 bus operations. At the bottom, the end of the trace: CPU 1 holds the lock, and CPU 2 spins on the M copy pulled in by its failed TSL; 7 operations in total.

There is an interesting detail in step 8. CPU 2's failed TSL is also a write. The lock does not change (1 is written over 1), but the line moves from CPU 1 to CPU 2. When CPU 1 leaves the critical section and calls `unlock()`, it has to pull the line back. The price of the race is these extra operations. As the number of waiters grows, on every release all the waiters read and try TSL; this is why the MCS lock scales better.

> **Common mistake**
>
> Thinking the inner read loop in TTAS suffices and removing the TSL. Between the read and the write another CPU can take the lock; at steps 5 and 6 both CPUs saw k = 0. The atomic TSL ensures the lock's correctness; the inner read only reduces traffic.

12.4.5 Spin or switch?

When a thread cannot get a lock, it can do one of two things. It can **spin**: it keeps polling the lock without giving up the CPU. The CPU is wasted in the meantime, but the thread continues as soon as the lock is free. Or it can **switch** (block): it sleeps and the CPU is given to another thread. The price of this is two context switches: one when it goes to sleep and one when it wakes. The awakened thread also meets a cold cache.

Figure 12.25 compares the two options on a timeline. The lock becomes free at t = 3 µs, and a context switch takes C = 5 µs. The spinning thread waits 3 µs and enters the critical section. The switching thread leaves at 5 µs and the CPU moves to other work. The thread comes back at 5 µs and enters the critical section only at the 15th microsecond.

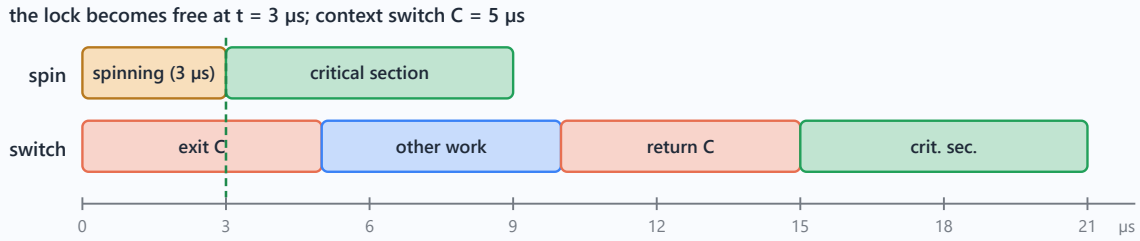


Figure 12.25: When the lock becomes free after 3 µs, spinning costs 3 µs and switching costs 2C = 10 µs.

The decision rule follows from this. If the expected wait is shorter than the cost of switching, that is, 2C, spinning is cheaper; otherwise sleeping is. For short critical sections spinning is right, and for regions that involve disk I/O or last long, sleeping is.

The problem is that the wait time is not known in advance. For this there is a well-known compromise: spin first for as long as the cost of switching, that is, for 2C, and if the lock still cannot be acquired, sleep. In the worst case the cost of this strategy does not exceed twice the cost of an ideal algorithm that knows the wait time in advance. An online strategy with such a guarantee is called a **competitive** strategy.

Example 12.8

C = 5 µs, so the cost of switching is 2C = 10 µs. The thread first spins for 10 µs and then sleeps. How does the cost of this strategy compare with the ideal one when the wait is t = 3, 10.1, and 50 µs?

1. **t = 3.** It spins for 3 µs and the lock is acquired; the cost is 3 µs. The ideal cost is also min(3, 10) = 3 µs; the ratio is 1.
2. **t = 10.1.** It spins for 10 µs and then sleeps: 10 + 10 = 20 µs. The ideal is to sleep from the start: 10 µs. The ratio is 2; this is the worst case.
3. **t = 50.** Again 10 + 10 = 20 µs is spent; the ideal is 10 µs. The ratio is 2.

Had the threshold been chosen as C, then when t slightly exceeded C the cost would be C + 2C = 3C and the ideal about C; the ratio would rise to 3. (The ratios were verified by computing them in Python for thousands of values of t.)

Real systems apply this idea with adaptive locks. Linux's `futex`-based mutex spins briefly if the lock's owner is running on another CPU, and sleeps immediately if the owner is asleep. On Windows, a spin count can be set for `CRITICAL_SECTION`. The Java virtual machine also keeps uncontended locks cheap and converts them into an operating system lock when contention rises; this is called lock inflation.

Inside the kernel one more rule applies: one cannot sleep in a region where interrupts are disabled, because the interrupt that would wake the sleeping code cannot arrive. In such regions only a spinlock can be used.

> **Common mistake**
>
> Spinning on a spinlock in a single-CPU system. If the lock's owner is waiting in the ready queue on the same CPU, the owner can never make progress unless the spinning thread gives up the CPU. Spinning makes sense only when the lock's owner is running on another CPU.

12.5 Multiprocessor scheduling

On a single processor, the scheduler answers just one question: which thread is next? On a multiprocessor a second question is added: on which CPU will this thread run? If the threads are independent of one another, the second question is easy. If the threads of the same application communicate with one another, the two questions must be answered together. This section describes three approaches: time sharing, space sharing, and gang scheduling.

12.5.1 Time sharing: a shared ready queue

For mutually independent processes, the simplest approach is for all CPUs to take work from a single common ready queue (Figure 12.26). The queue is organized by priority levels, as in single-processor systems. When a CPU becomes free, it locks the queue, removes the highest-priority thread, releases the lock, and runs that thread. In the figure, 14 of the 16 CPUs are running; CPU 3 and CPU 7 are idle. Whichever of the idle CPUs locks the queue first will take A, at the head of the priority 7 queue.

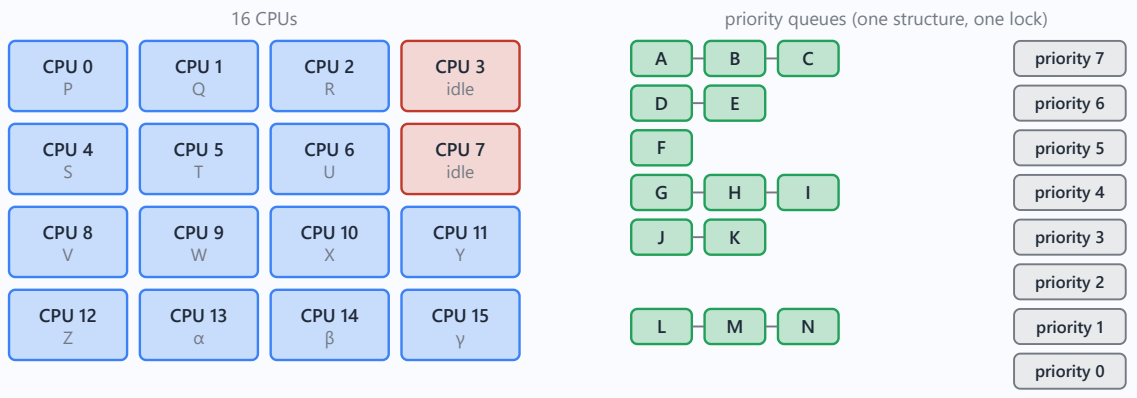


Figure 12.26: Time sharing: 16 CPUs take work from a single priority ready queue. The queue is a single structure and is protected by a single lock.

The biggest advantage of this approach is that load is balanced automatically. While one CPU waits for work, work does not pile up on another CPU. It has three drawbacks:

- **The single lock is a bottleneck.** As the number of CPUs grows, contention for the queue's lock grows too. In a system with hundreds of CPUs, the CPUs wait for one another to take the next job.
- **The cache stays cold.** If a thread has been running on CPU 4 for a while, its cache and TLB are full of its data. When it comes out of the queue again it may land on any idle CPU, for example CPU 12. There the cache is empty, and the thread spends its first milliseconds on misses. To prevent this, the scheduler tries to run a thread, if possible, on the CPU where it last ran. This tendency is called **processor affinity**.
- **Preemption while holding a lock.** If a thread's time slice ends while it holds a spinlock, it is preempted. The other CPUs that want the lock spin in vain until the owner runs again.

For the last problem, the idea of **smart scheduling** described by Tanenbaum can be used. A thread that acquires a spinlock sets a flag visible process-wide and clears the flag when it releases the lock. The scheduler does not preempt a thread with the flag set. It gives it a little extra time to finish its critical section. A similar mechanism is used in virtual machines today: the problem of a virtual CPU that holds a lock being preempted while the other virtual CPUs spin in vain is called “lock holder preemption”.

Today's operating systems do not use a single common queue. Linux and Windows keep a separate ready queue for each CPU. Lock contention goes away and a thread automatically stays on the same CPU. Imbalance between the queues is corrected by a separate load-balancing mechanism. A CPU with little load pulls work from another CPU's queue, or the load balancer occasionally pushes jobs from loaded CPUs to idle ones. Linux does this balancing with a hierarchy called “scheduling domains”. It balances first among the SMT siblings of the same physical core, then among cores that share the same L3, and last among NUMA nodes.

12.5.2 Space sharing

For threads that communicate tightly with one another, another approach is more suitable: running all the threads of the same application at the same time, on separate CPUs. This is called **space sharing**. When a job starts, the number of CPUs it asks for is allotted to it. It holds these CPUs until the job ends. There is no multiprogramming per CPU, and so no context switching. If there are not enough free CPUs, the job waits until enough CPUs become free. The simplest ordering is first-come, first-served (FCFS).

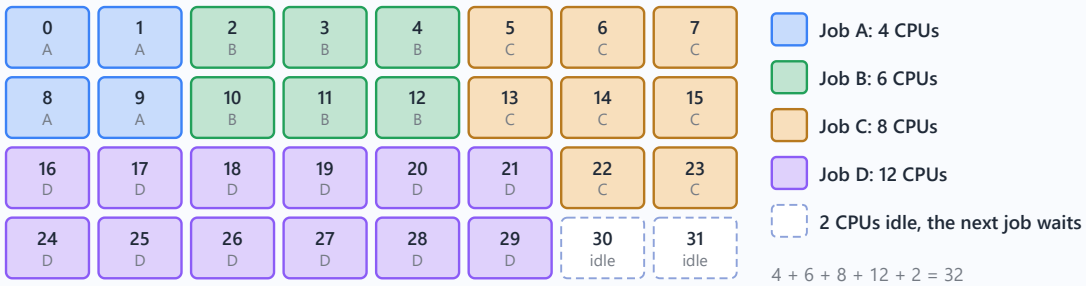


Figure 12.27: A 32-CPU system divided among four jobs: A uses 4, B 6, C 8, D 12 CPUs; 2 CPUs are idle. If the next job wants more than 2 CPUs, it waits.

In the example of Figure 12.27, 32 CPUs are distributed among four jobs: 4 + 6 + 8 + 12 = 30 CPUs are in use, 2 are idle. If the next job in the queue wants 3 CPUs, it waits until one of these jobs ends. In the meantime two CPUs stay idle. This is the main drawback of space sharing. When a thread blocks for I/O, its CPU is not given to another job and sits idle. Today the job schedulers in HPC clusters (Slurm, PBS) partition nodes in exactly this way: a job gets the nodes it asks for and keeps them until it finishes.

12.5.3 Gang scheduling

An example shows why communicating threads slow down under time sharing (Figure 12.28). There are two CPUs, the time slice is 100 ms, and two jobs run with time sharing. The two threads of job A, A0 and A1, send messages to each other. On CPU 0 the order is A0, B0, A0, B0...; on CPU 1 it is B1, A1, B1, A1.... A0 sends a request to A1 at t = 0. A1 is not running at that moment and sees the request only at t = 100. It sends the reply at once, but this time A0 is not running. A0 receives the reply at t = 200. A single request-reply has taken 200 ms.

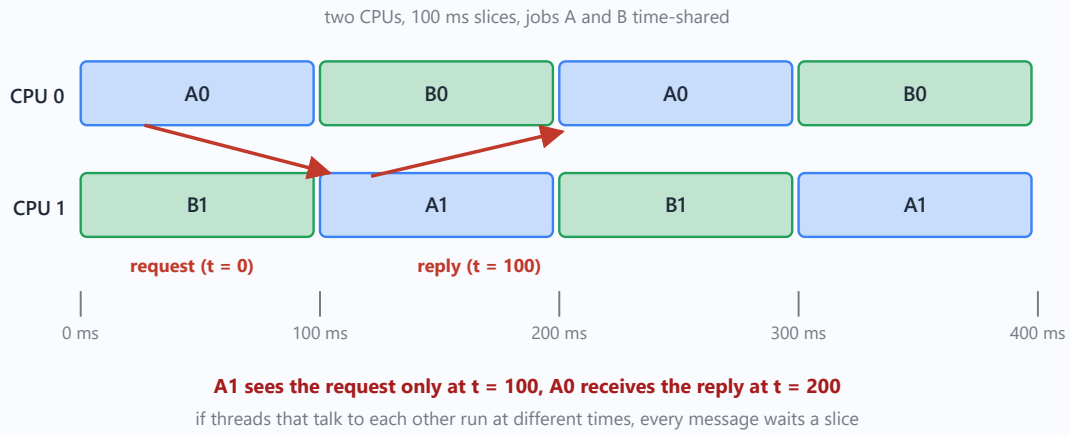


Figure 12.28: Without gangs: A0 and A1 run in offset slices; every message waits a slice, and a request-reply takes 200 ms.

Definition 12.5

Gang scheduling is running all the members of a group (a gang) of interdependent threads together in the same time slice, on different CPUs. All the CPUs start their slices at the same time and end them at the same time.

Gang scheduling combines time sharing and space sharing. It has three parts. Interdependent threads are grouped as a gang. All members of a gang run at the same time, on different CPUs. All gang members begin their slices together and end them together. This way a message sent by one member is seen at once by its receiver, and the reply comes back in the same slice.

Example 12.9

In a system with 6 CPUs there are five jobs: A has 6 threads, B 3, C 3, D 5, and E 7. How are they placed into slices with gang scheduling?

1. **Slice 0.** A's 6 threads fill the 6 CPUs.
2. **Slice 1.** B (3) and C (3) together fill the 6 CPUs.
3. **Slice 2.** D's 5 threads and E0. E's 7 threads do not fit in a single slice; its first member runs here.
4. **Slice 3.** E's remaining 6 threads (E1–E6).
5. **Cycle.** After four slices the order repeats from the start.

In total 6 + 3 + 3 + 5 + 7 = 24 threads fit exactly into 4 slices × 6 CPUs = 24 slots; utilization is 100%. In Tanenbaum's example, E is a large gang and has been split across two slices. Under a stricter definition no member of a gang runs apart from the others; then this arrangement could not be made, because E does not fit in one slice.

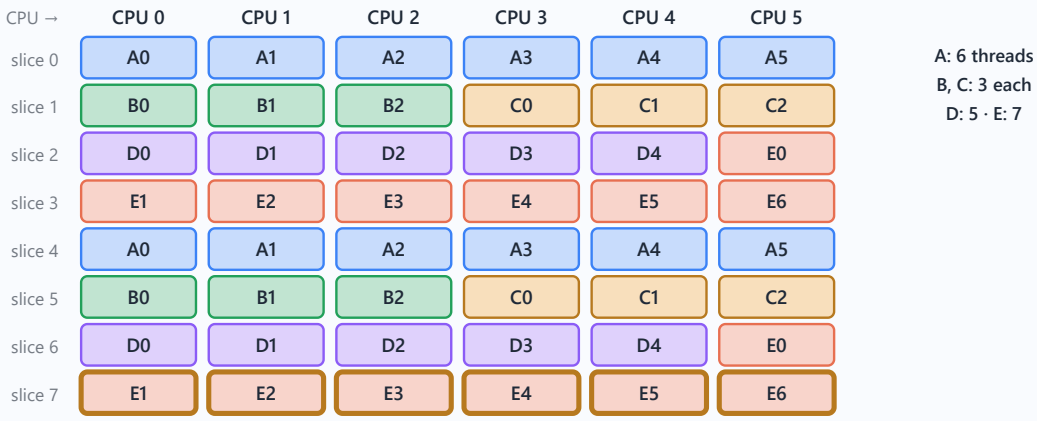


Figure 12.29: Gang scheduling: 6 CPUs and 8 slices. The slices A, B+C, D+E0, and E1–E6 repeat in a cycle of four slices.

The two conditions of gang scheduling can be summarized briefly: the gang members start at the same time and are all on different CPUs. This idea appears today in virtual machines as well. The vCPUs of a virtual machine with many virtual CPUs wait for one another through locks. If the hypervisor schedules them together (co-scheduling), a vCPU is prevented from spinning in vain while waiting for a preempted sibling.

Common mistake

Keeping in mind only the “at the same time” condition of gang scheduling. If the members ran at the same time but in turn on the same CPU, they would still wait for one another. Both conditions are needed together: the same slice and different CPUs.

12.6 Multicomputers

Shared-memory multiprocessors cannot be scaled beyond a few hundred CPUs, and scaling them is expensive. Larger systems are built from computers that do not share memory. This section describes the hardware of multicomputers, how messages are carried across the network, and the programming and load-distribution methods used in these systems.

12.6.1 Nodes and network interfaces

Definition 12.6

A **multicomputer** is a system in which nodes, each consisting of a CPU, RAM, and a network interface, are joined by a fast interconnect. Memory is not shared; each node runs its own copy of the operating system, and the nodes communicate only by messages.

Multicomputers are also called clusters. A node is often an ordinary server; what makes it special is the network interface and the interconnect. Large-scale simulations, high-performance computing, and AI training run on these systems. The systems on the TOP500 list, which ranks the world's fastest computers, are multicomputers with tens of thousands of nodes.

In a multicomputer, the biggest obstacle to performance is the copying of messages (Figure 12.30). In the worst case a message is copied five times:

1. from the sending process's user buffer to a kernel buffer,
2. from the kernel buffer to the memory of the network interface card (NIC),
3. from the NIC over the wire to the receiving NIC,
4. from the receiving NIC's memory to the receiver's kernel buffer,
5. from the kernel buffer to the receiving process's user buffer.

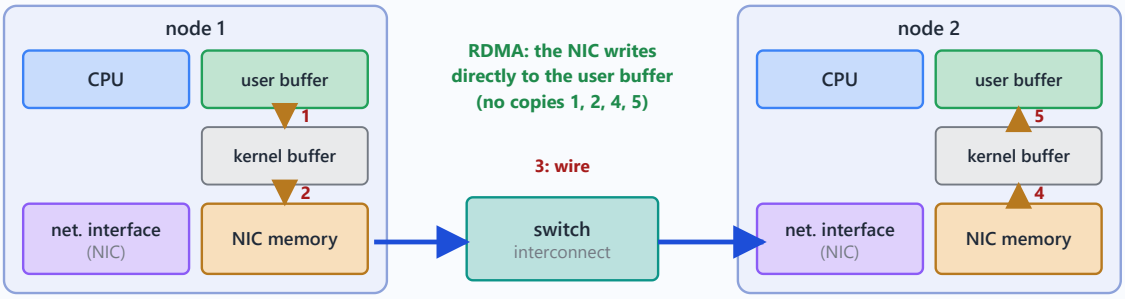


Figure 12.30: The copies of a message. Copy number 3 is the wire; with RDMA the NIC writes directly to the user buffer and copies 1, 2, 4, 5 disappear.

Each copy takes microseconds, and the network itself can be faster than that. The ways to reduce copies are these. The NIC memory can be mapped directly into user space; the process writes the message there itself. The NIC can read and write main memory directly with DMA. The most advanced solution is **RDMA** (remote direct memory access). On InfiniBand and RoCE networks, the NIC writes to the memory of a process on the remote machine without the kernel being involved at all. A message thus reaches the other side in under a microsecond instead of a few microseconds.

12.6.2 Interconnection topologies

How the nodes are connected to one another is called the **topology**. There are two basic measures for evaluating a topology. The **diameter** is the number of hops on the shortest path between the two farthest nodes; it determines the worst-case latency. The **number of links**, on the other hand, determines the cost. Figure 12.31 shows six common topologies.

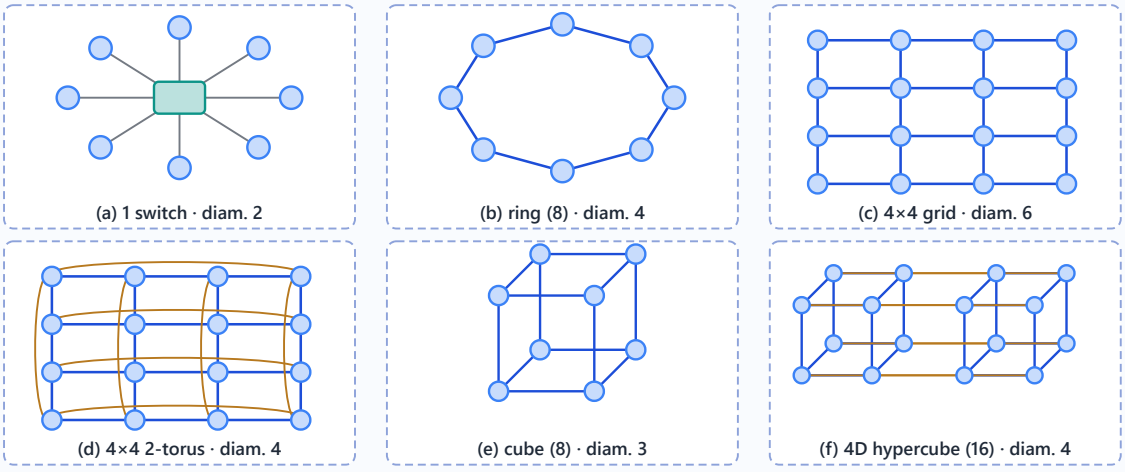


Figure 12.31: Interconnection topologies and their diameters: (a) single switch, (b) ring, (c) grid, (d) double torus, (e) cube, (f) four-dimensional hypercube.

- **Single switch** (star): every node is attached to the switch in the center. The diameter is 2, but the switch is a single point of failure and cannot be scaled up.
- **Ring**: each node has two neighbors. In a bidirectional ring of n nodes the diameter is n / 2.
- **Grid**: nodes are laid out in a square arrangement. In a k × k grid there are 2(k − 1) hops between the two farthest corners.
- **Torus**: the edges of the grid are wrapped around to the opposite edge. In a k × k torus the diameter is 2 · ⌊k / 2⌋.
- **Hypercube**: a d-dimensional hypercube has 2^d nodes and each node is attached to d neighbors. The diameter is $d = \log_2 n$.

Example 12.10

For a system of 1,024 nodes, what are the diameter and the number of links of a ring, a 32 × 32 grid, a 32 × 32 torus, and a 10-dimensional hypercube?

1. **Ring.** Diameter 1024 / 2 = 512, links 1024.
2. **Grid.** Diameter (32 − 1) + (32 − 1) = 62. There are 31 horizontal links in each row and 31 vertical links in each column: 2 × 32 × 31 = 1984.
3. **Torus.** With the wraparound links, at most 16 hops are needed in each dimension: diameter 32. 4 links leave each node and each link is shared by two nodes: 1024 × 4 / 2 = 2048.
4. **Hypercube.** Diameter $\log_2 1024 = 10$. Each node has 10 links: 1024 × 10 / 2 = 5120.

The values are collected in Table 12.4. The diameters of the small 16-node topologies were also computed with a breadth-first search and compared with the values in the figure.

n = 1,024	Ring	Grid 32 × 32	Torus 32 × 32	Hypercube (10D)
Diameter	512	62	32	10
Links	1.024	1.984	2.048	5.120

Table 12.4: Diameter and number of links of four topologies at 1,024 nodes.

The table shows a tradeoff. The hypercube has the smallest diameter, but also the largest number of links. Moreover, in a hypercube the number of links per node grows with the dimension; when the number of nodes doubles, one more link must be added to every node. Today's supercomputers mostly use *fat-tree* and *dragonfly* topologies. Japan's Fugaku supercomputer, on the other hand, uses a six-dimensional torus network (Tofu).

12.6.3 Routing in a hypercube

In a hypercube the nodes carry d-bit numbers, and two nodes are neighbors only if they differ in a single bit. A 4-dimensional hypercube has 16 nodes; each node has 4 neighbors. For example, the neighbors of 0000 are 1000, 0100, 0010, and 0001. This structure makes routing very easy. The length of the shortest path between two nodes equals the number of bits in which their numbers differ, that is, the **Hamming distance**. At each step the message advances by correcting one of the differing bits.

E-cube routing corrects these bits always in the same order, from left to right. For a message going from 0000 to 1011, source $\oplus$ destination = 1011; three bits differ. The first differing bit on the left is corrected: 1000. This step moves the message from the lower 3-dimensional cube to the upper one. Then it comes to 1010, and finally to 1011 (Figure 12.32). The number of hops is 3; in a 4-dimensional hypercube the worst case is $\log_2 16 = 4$ hops.

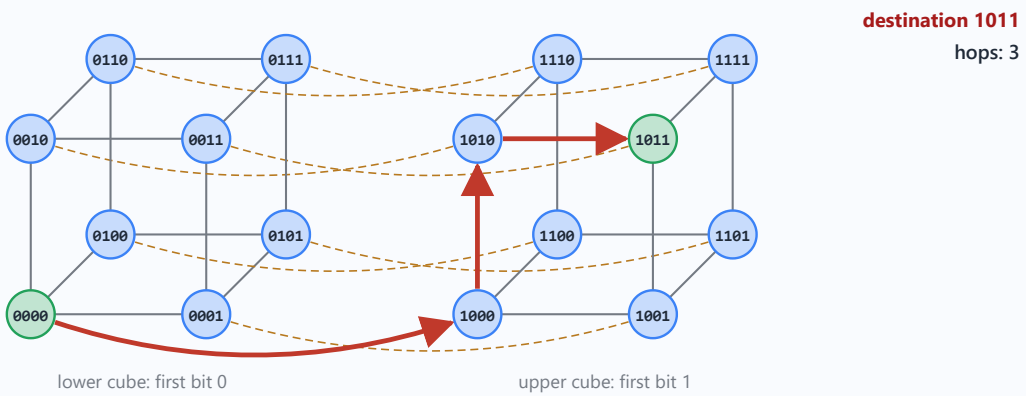


Figure 12.32: The path 0000 → 1000 → 1010 → 1011 in a 4-dimensional hypercube. In the left cube the first bit is 0 and in the right cube it is 1; the dashed edges join the two cubes.

An important property of the rule is that it is local. Each node decides which neighbor to send the message to by looking only at its own number and the destination's number. No routing table is needed. Because the bits are always corrected in the same order, the path never loops either.

12.6.4 Code trace: e-cube routing

E-cube routing is a few lines in C. `src` is the source and `dst` the destination node, and `d` is the dimension of the hypercube; the function prints each node passed.

```
void route(int src, int dst, int d) {
  int u = src;
  for (int b = d - 1; b >= 0; b--) {
    int m = 1 << b;
    if ((u ^ dst) & m) {  /* bit differs */
      u ^= m;             /* go to neighbor */
      printf("%d ", u);
    }
  }
}
```

`u` is the node where the message currently is. The loop starts at the leftmost bit (`b = d - 1`) and moves to the right. `m = 1 << b` is the mask with only bit b set to 1. If `(u ^ dst) & m` is nonzero, u and the destination differ in this bit. `u ^= m` flips that bit; this means the message moves to the neighbor in that dimension. The loop runs d times; the number of hops equals the number of differing bits.

Example 12.11

How does the call `route(6, 13, 4)`, that is, the message going from 0110 to 1101, proceed step by step?

b	m	u (before)	u ^ dst	Bit differs?	u (after)	Output
—	—	0110	1011	—	0110	—
3	1000	0110	1011	yes	1110	14
2	0100	1110	0011	no	1110	—
1	0010	1110	0011	yes	1100	12
0	0001	1100	0001	yes	1101	13

The output is `14 12 13`. The message follows the path 0110 → 1110 → 1100 → 1101 in 3 hops. The initial u ⊕ dst = 1011 said from the start that three bits differ. (The loop was transcribed one-to-one into Python and run.)

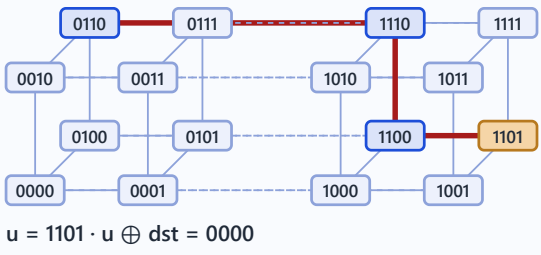


Figure 12.33: The end of the trace: u = 1101 and u ⊕ dst = 0000; the message has reached its destination.

Common mistake

Computing the number of hops from the difference of the node numbers. The difference between 6 and 13 is 7, but the number of hops is the number of differing bits: 0110 ⊕ 1101 = 1011, that is, 3. Adjacency is not numerical closeness but a single-bit difference.

12.6.5 Store-and-forward or cut-through

A packet travels from source to destination through several switches. How the switches forward the packet changes the total latency considerably. As an example, consider a 4 KiB packet and 10 Gb/s links. It takes 4096 × 8 bits / 10^{10} bit/s ≈ 3.28 µs for the packet to cross one link.

In **store-and-forward**, each switch takes the whole packet into its buffer, checks it, and only then sends it on to the next link. A packet going from A to B through two switches crosses three links. On each link the whole packet is waited for: 3 × 3.28 = 9.83 µs. Latency grows linearly with the number of hops.

In **cut-through**, a switch sets up the path as soon as it has read the packet's header and starts forwarding without waiting for the rest of the packet. The packet flows like a pipe; its front part arrives at the destination while its back part is still leaving the source. **Wormhole** routing is the form of this idea used with packets divided into small pieces (flits). The total latency is about one packet time plus the small header delays of the switches. If each switch processes the header in 0.05 µs, the total is 3.28 + 2 × 0.05 ≈ 3.38 µs (Figure 12.34).

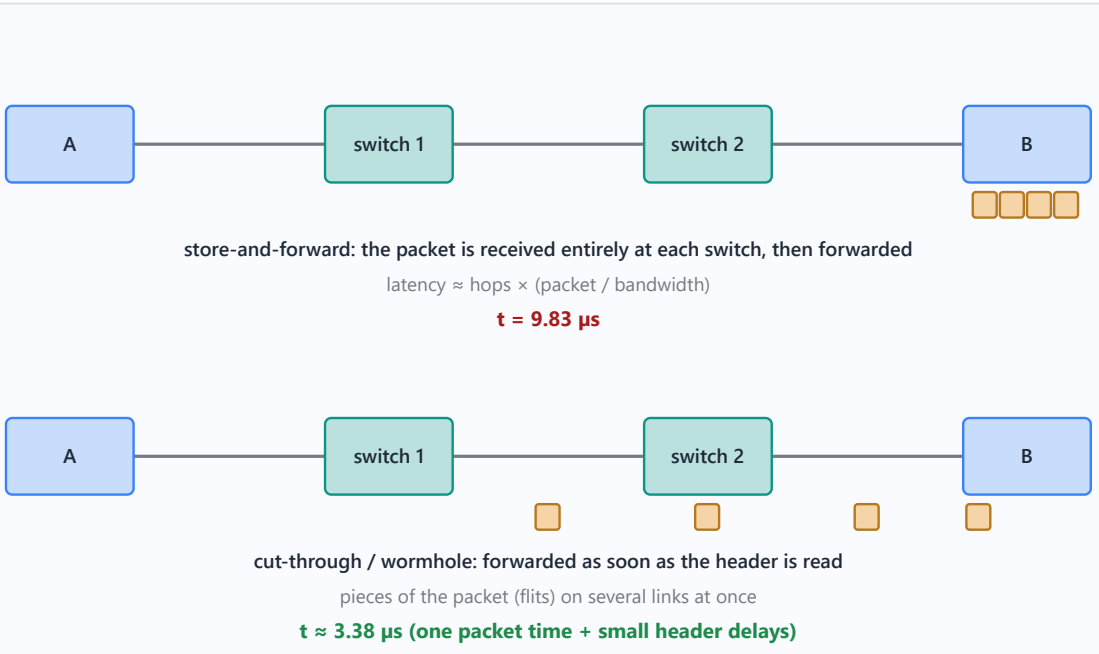


Figure 12.34: At the top, store-and-forward: the packet is received entirely at each switch, for a total of 9.83 µs. At the bottom, cut-through: pieces of the packet are on several links at once, for a total of ≈ 3.38 µs.

The gap widens as the number of hops grows. The latency of store-and-forward is proportional to the number of hops; in cut-through each hop adds only a small header delay. That is why high-performance computing networks and data-center switches use cut-through or wormhole routing.

12.6.6 Blocking and nonblocking send

In a multicomputer, processes communicate with `send` and `receive` calls. The most important design question for these calls is whether the sender waits for the message to be delivered.

In **blocking** or synchronous send, the `send` call does not return until the message has been delivered (Figure 12.35 (a)). The process traps into the kernel, the message is copied and sent; only after that does the process continue. This method is simple and safe: when `send` returns, the buffer can be reused. The price is that the CPU waits idle for the whole transmission.

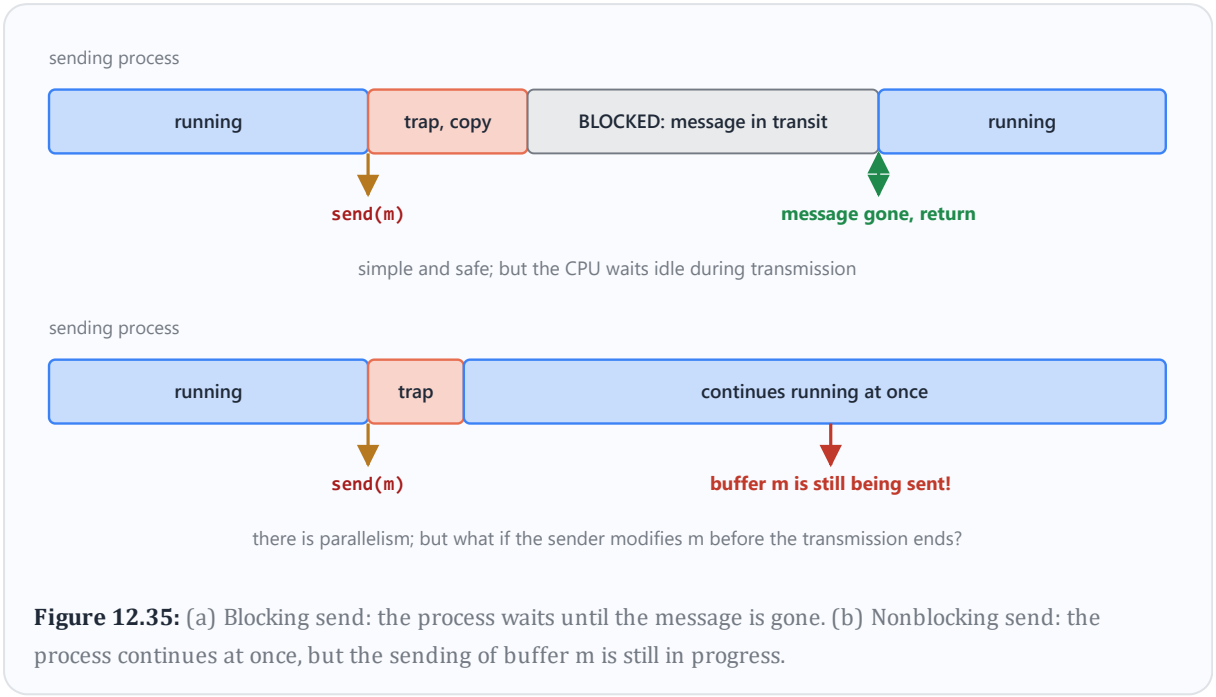


Figure 12.35: (a) Blocking send: the process waits until the message is gone. (b) Nonblocking send: the process continues at once, but the sending of buffer m is still in progress.

In **nonblocking** or asynchronous send, `send` returns immediately and the process keeps running (b). Computation and communication proceed in parallel. But a new problem arises: if the sender modifies `m` in the buffer before the transmission ends, what data goes to the other side? Tanenbaum lists four answers to this problem (Table 12.5).

Option	Cost
Blocking send	CPU idle during transmission
Nonblocking + copy to the kernel	CPU time for the extra copy
Nonblocking + interrupt on completion	hard to program (race conditions)
Copy on write (CoW)	an extra copy again if the buffer is written

Table 12.5: Send options and their costs.

In the second option, the kernel copies the message into its own buffer and the process continues at once; the buffer is now free. In the third, when the transmission is finished an interrupt or signal reaches the sender; the process must not touch the buffer until then. In the fourth, the page of the buffer is marked read-only; a copy of the page is made only if the process tries to write to the buffer.

There are similar options on the receiver side. A blocking `receive` waits until a message arrives. With polling, the process looks now and then to see whether a message has come. A third way is to start a new thread when the message arrives, or to run the code inside the message directly (an *active message*). The present-day counterparts of these concepts are familiar too. In the MPI library, `MPI_Send` / `MPI_Recv` are blocking calls, and `MPI_Isend` / `MPI_Irecv` with `MPI_Wait` are nonblocking ones. On Linux, `io_uring` and zero-copy `MSG_ZEROCOPY` are modern answers to the same problem.

12.6.7 Remote procedure call

`send` and `receive` are an I/O model; they force the programmer to build the messages. In 1984 Birrell and Nelson proposed a more familiar model: calling a procedure on another machine as if it were a local procedure.

> **Definition 12.7**
>
> A **remote procedure call** (RPC) is a program on one machine calling a procedure on another machine with the syntax of an ordinary local call. The construction and sending of messages are hidden from the programmer.

The hiding is done by small procedures called **stubs**. When the client writes `result = add(3, 4)`, it is really calling the client stub. A call goes through these steps (Figure 12.36):

1. The client calls the client stub like a local procedure. The parameters are pushed on the stack as usual.
2. The stub packs the parameters into a message and makes a system call to the kernel. The packing is called **marshalling**.
3. The kernel sends the message across the network to the server machine.
4. The server's kernel hands the message to the server stub.
5. The server stub unpacks the parameters and calls the server procedure locally.

The reply follows the same path in reverse: server → stub → kernel → network → client kernel → client stub → client. From the client's point of view, `add` is an ordinary procedure; it is only a little slow.

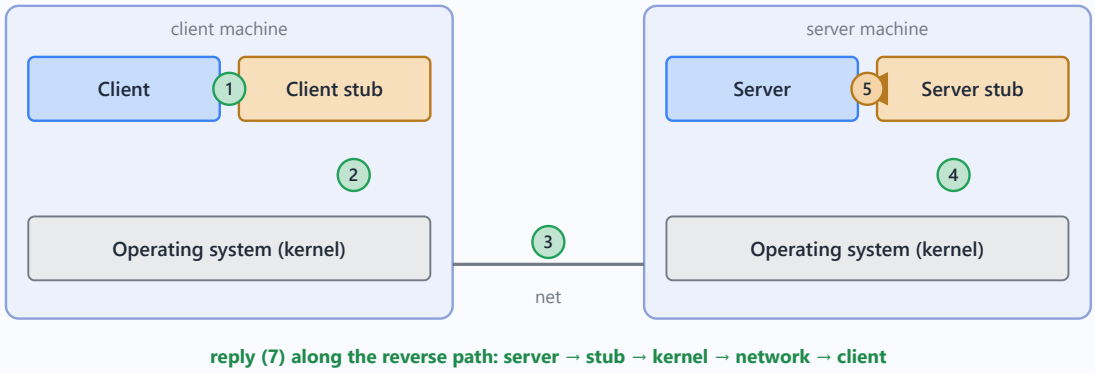


Figure 12.36: The steps of an RPC: client, client stub, kernel, network, server kernel, server stub, and server. The reply returns along the reverse path.

This apparent simplicity hides several difficulties:

- **Pointers.** An address on the client is meaningless in the server's address space. An array parameter is copied by value and sent, and the changes the server makes are copied back on return. This method is called **copy-restore**. Structures built with pointers, such as linked lists, cannot easily be sent this way.
- **Arrays of unknown length.** In C the length of an array is not stored in the array itself. The stub cannot know how many bytes to send; the length must be given as a separate parameter.
- **Global variables.** The server cannot see the client's global variables.
- **Partial failures.** In a local call, caller and callee crash together. In an RPC, the server or the network can crash alone. If the client, receiving no reply, repeats the call, the procedure may run twice. For this reason it must be stated whether an RPC system provides “at least once” or “at most once” semantics.
- **Data representation.** The two machines may differ in byte order or integer size; the stub converts the data to a common format.

Stubs are not written by hand; they are generated automatically from an interface definition. The most common examples today are gRPC and its interface language Protocol Buffers, Apache Thrift, and Java RMI.

Common mistake

Thinking of the stub as a network driver or a mechanism that runs remote code locally. The stub is just an ordinary procedure that packs the parameters into a message, sends it to the kernel, and unpacks the reply. The remote procedure runs on the server.

12.6.8 Distributed shared memory

RPC hides message passing behind a procedure call. A more ambitious idea is to create the illusion of shared memory on machines that do not share memory. This idea is called **distributed shared memory** (DSM). The layer that provides the illusion can be in hardware (NUMA), in the operating system, or in a user library. Page-based DSM, defined by Li and Hudak in 1989, uses the operating system's virtual memory mechanism.

All machines share the same virtual address space, but each page is physically in the memory of a single machine (Figure 12.37). When a CPU accesses a page that is not in its memory, the MMU raises a page fault. The operating system finds out, for example through a directory, which machine the page is on, and requests it. The page is moved over the network to this CPU, the page table is updated, and the instruction that caused the fault is executed again. From the process's point of view this is no different from an ordinary page fault served from disk; only the page comes not from disk but from another machine.

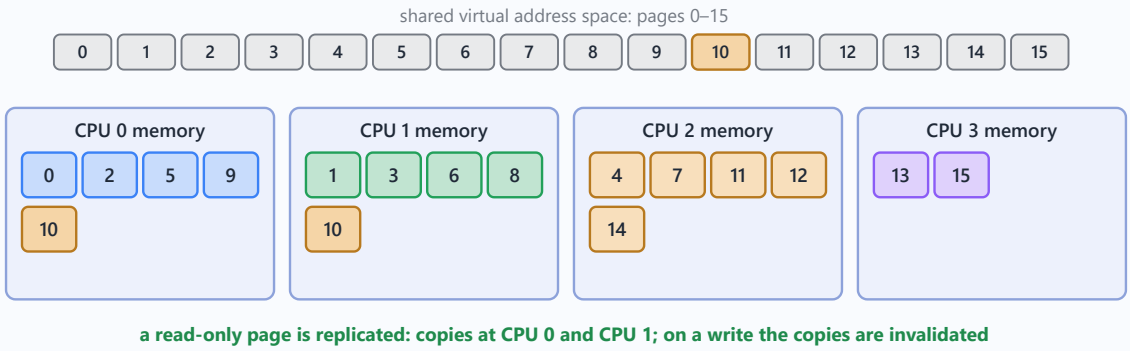


Figure 12.37: 16 pages distributed over four CPUs. Page 10 is replicated because it is read-only: CPU 0 and CPU 1 each have a copy.

Constantly moving frequently read pages is expensive. So a read-only page is **replicated**: page 10 in the figure has a copy at both CPU 0 and CPU 1, and both CPUs read locally. If a CPU wants to write to a replicated page, the other copies are invalidated. This is the page-sized counterpart of MESI. For the same reason, false sharing also appears at page size. If two independent variables used on different machines are on the same page, the page goes back and forth between the two machines.

Page-based DSM is not common today. Moving a 4 KiB page over the network is expensive, and a page is a coarse unit for sharing. High-performance computing turned to message passing (MPI) and PGAS languages (partitioned global address space). The present-day counterpart of the idea is in hardware: the CXL standard allows multiple machines to use a common memory pool.

12.6.9 Graph-theoretic deterministic allocation

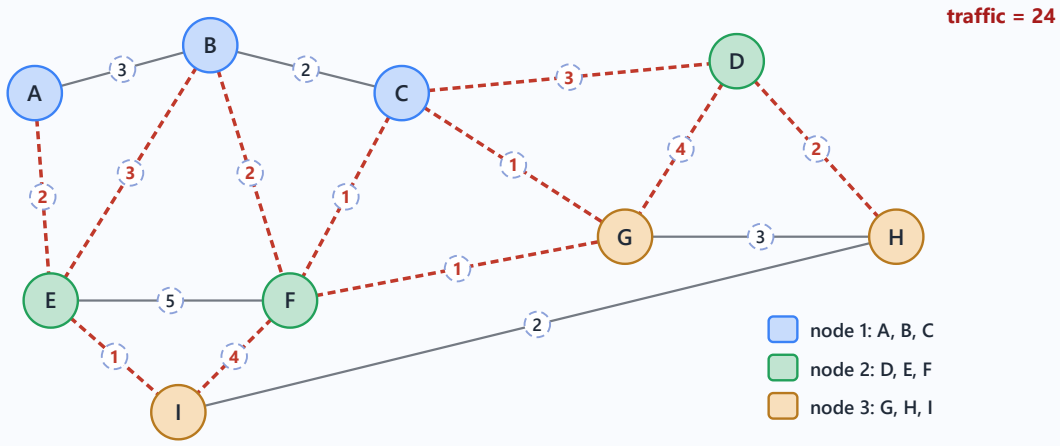
In a multicomputer, which node the processes run on also matters. If two processes that communicate a lot are put on separate nodes, every message of theirs crosses the network. If they are put on the same node, their messages stay local. If the traffic between processes is known in advance, the assignment can be solved as a graph problem. The processes are the vertices and the traffic between them the edge weights. The aim is to partition the vertices into equal-sized sets while minimizing the total weight of the edges that remain between the sets.

Example 12.12

Nine processes A–I are to be assigned to 3 machines, three each. The traffic between processes is the edge weights in Figure 12.38; the total traffic is 39. What is the network traffic of the alphabetical assignment and of the best assignment?

1. **Alphabetical assignment.** {A, B, C}, {D, E, F}, {G, H, I}. Within machines only the edges A–B (3), B–C (2), E–F (5), G–H (3), and H–I (2) remain: a total of 15. The network traffic is 39 − 15 = 24.
2. **A better assignment.** {A, B, C}, {D, G, H}, {E, F, I}. Within machines A–B (3), B–C (2), D–G (4), D–H (2), G–H (3), E–F (5), E–I (1), F–I (4) remain: a total of 24. The network traffic is 39 − 24 = 15.
3. **Is it the best?** Nine processes can be divided into three groups of three in $9!\,/\,(3!^3 \cdot 3!) = 280$ different ways. When all are tried, the smallest network traffic is 15, and only the assignment above gives this value.

(All 280 partitions were enumerated in Python and their cut weights computed.)



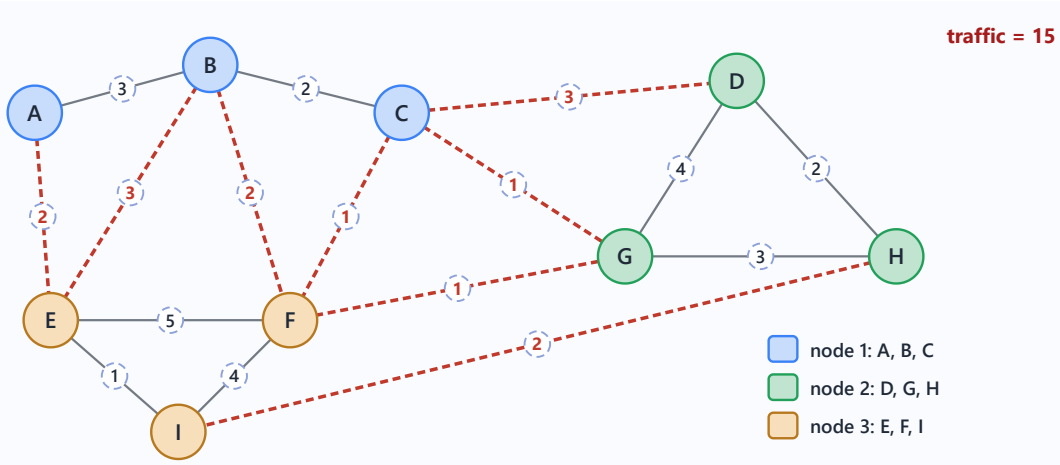


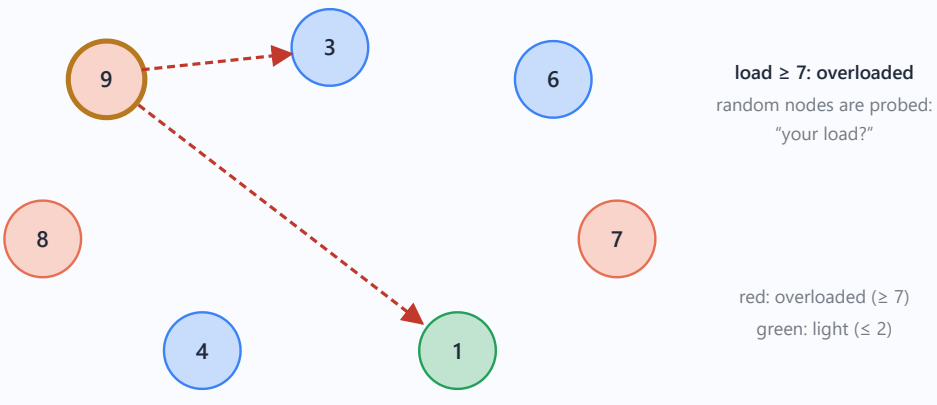
Figure 12.38: Nine processes and the traffic between them. At the top the alphabetical assignment (network traffic 24), at the bottom the best assignment (15). The dashed red edges are the traffic that remains between machines.

When the processes that communicate a lot are gathered on the same machine, the network traffic drops by more than a third. But this method has two limits. First, the traffic must be known in advance. Second, the problem is NP-hard in general: as the number of processes grows, the number of possible partitions explodes and they cannot all be tried. In practice, heuristic methods are used.

12.6.10 Distributed heuristic load balancing

If the traffic is not known, or if jobs are constantly coming and going, load is distributed with simple rules that the nodes run among themselves. There are two basic algorithms.

In the **sender-initiated** algorithm, a node that creates a new job looks at its own load. If its load exceeds a threshold, it asks another randomly chosen node "what is your load?" (Figure 12.39). Probing is limited, for example to at most N = 3 nodes. If a node below the threshold is found, the job is sent to it; if none is found, it runs locally. Eager and coauthors studied this algorithm in 1986. In the example in the figure the threshold is 7, and the node with load 9 probes two nodes with loads 3 and 1. It sends the job to the least loaded one, the node with load 1.



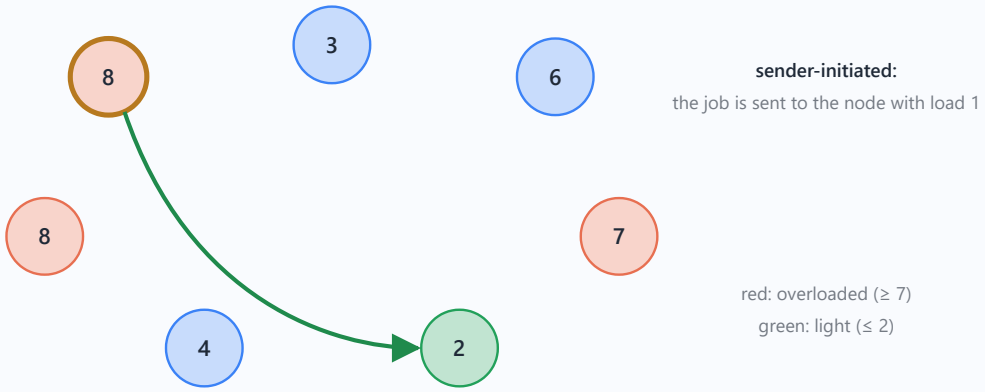


Figure 12.39: Sender-initiated load balancing. At the top the node with load 9 probes two random nodes; at the bottom the job has been sent to the node with load 1. Red: overloaded (≥ 7), green: light (≤ 2).

The drawback of this algorithm appears when the whole system is loaded. Every node tries to send out jobs, and every probe is wasted. The probes themselves also put extra load on the network and the nodes. As load grows, the system spends more of its time probing.

The **receiver-initiated** algorithm approaches the problem from the opposite direction. A node that has finished its work and become idle looks for work itself. It probes random nodes and, if it finds a loaded one, takes a job from it (Figure 12.40). The cost of probing falls on the node that is idle anyway. When the system is loaded, there are few idle nodes and so few probes. This is why the receiver-initiated algorithm works well under high load, and the sender-initiated one under low load. There are also algorithms that combine the two.

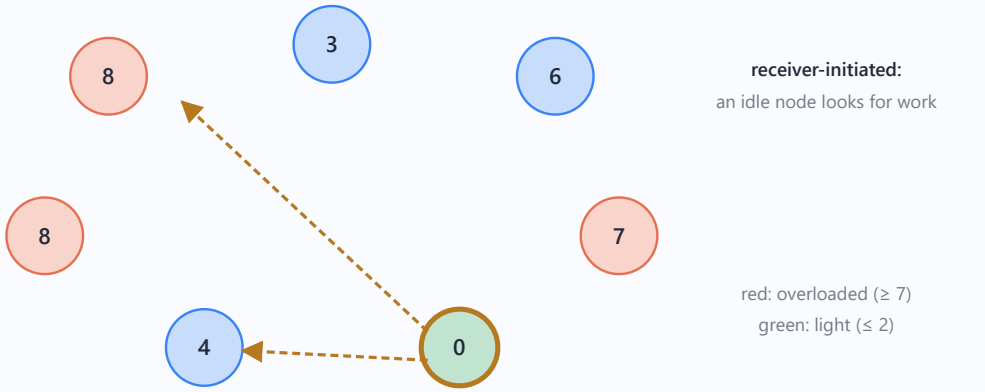


Figure 12.40: Receiver-initiated load balancing: a node that has finished its work and dropped to load 0 asks overloaded nodes for work.

The present-day counterpart of receiver-initiated balancing is **work stealing**. Each worker thread has its own work queue. A worker whose queue is empty steals work from the other end of another worker's queue. Cilk, Java's `ForkJoinPool`, and the runtimes of Go and Rust's Tokio use this method.

12.7 Virtualization

Most servers in a data center are lightly loaded most of the time. Allocating a separate physical machine to each service wastes hardware, energy, and space. Virtualization solves this by running several virtual machines on a single physical machine. Each virtual machine runs its own operating system and is isolated from the others. This section describes the software that runs virtual machines, how the processor and memory are virtualized, and containers.

12.7.1 Hypervisors: type 1 and type 2

Definition 12.8

A **hypervisor** or virtual machine monitor (VMM) is the software layer that creates and runs virtual machines. It presents each virtual machine with an environment that imitates the hardware of a real computer. The operating system running in a virtual machine is called the **guest**.

Hypervisors divide in two according to where they run (Figure 12.41). A **type 1** hypervisor runs directly on the hardware; there is no other operating system beneath it. It is also called a bare-metal hypervisor. VMware ESXi, Xen (today XCP-ng), and Microsoft Hyper-V are of this kind. Linux's KVM module turns the Linux kernel itself into a type 1 hypervisor. Most clouds use KVM or its derivatives; AWS's Nitro system and Google's cloud are examples. When Hyper-V is enabled on Windows, Windows too begins to run on top of the hypervisor, in a privileged “root partition”.

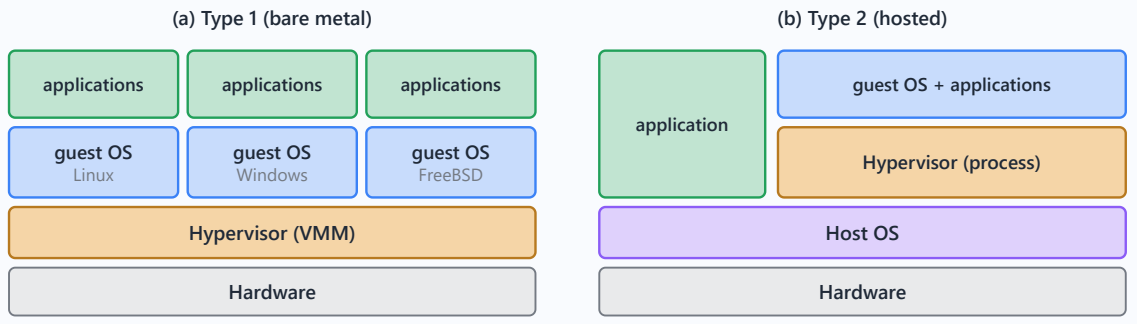


Figure 12.41: (a) A type 1 hypervisor runs directly on the hardware; (b) a type 2 hypervisor is a process on top of the host operating system.

A **type 2** hypervisor runs like an ordinary application on a host operating system. VirtualBox, VMware Workstation and Fusion, Parallels, and UTM are of this kind. It leaves device access to the host operating system; so the overhead in I/O is greater. A pure type 2 hypervisor interprets the guest's instructions or converts them by binary translation; QEMU without KVM works this way. Type

2 hypervisors in practice, however, load a kernel module and use the processor's virtualization support. What determines a hypervisor's type is not whether it uses hardware support but where it runs.

Popek and Goldberg set out in 1974 the condition required for a machine to be virtualized efficiently. Let us divide instructions into two groups. **Sensitive** instructions are those that affect the state of the machine, for example the page table or I/O, or behave differently depending on it. **Privileged** instructions are those that trap when an attempt is made to execute them in user mode.

> **Theorem 12.2 (Popek–Goldberg)**
>
> If all the sensitive instructions of a machine are privileged, that is, they trap in user mode, an efficient hypervisor can be built for that machine.

The meaning of the condition is this. The guest operating system is run in user mode. Its ordinary instructions are executed directly on the hardware. When it tries to execute a sensitive instruction, it traps and control passes to the hypervisor. The hypervisor emulates the effect of the instruction for the virtual machine. Classic x86 did not satisfy this condition. For example, the `POPF` instruction also changes the interrupt flag; but in user mode it silently ignores this change without trapping. The guest operating system thinks it has disabled interrupts, and the hypervisor has no idea. This problem was solved first by binary translation and then by hardware support.

The reasons for using virtualization are well known today. Many servers are consolidated onto one machine. A running virtual machine can be moved to another physical machine without interruption; this is called live migration. A snapshot of a virtual machine can be taken and it can be rolled back to it when needed. This is also the foundation of cloud computing.

12.7.2 Trap and emulate: VT-x and AMD-V

Intel added hardware virtualization support to x86 with VT-x in 2005, and AMD with AMD-V in 2006. This support gives the processor a new operating mode. The hypervisor runs in **root** mode and the guest in **non-root** mode. The guest operating system runs in the kernel mode of non-root mode and believes it is truly in kernel mode. But the real control is with the hypervisor.

When the guest kernel wants to execute a sensitive instruction, this happens (Figure 12.42):

1. The guest kernel executes a sensitive instruction: `mov cr3`, which changes the page table base, `out`, which writes to an I/O port, or `hlt`, which halts the processor.
2. The hardware does not execute the instruction. Instead it hands control to the hypervisor with a **VM exit**. The guest's state and the reason for the exit are written to a memory structure called the VMCS.
3. The hypervisor **emulates** the effect of the instruction. For example, it updates the shadow table or the EPT, or changes the state of a virtual device.

4. The hypervisor returns to the guest with a **VM entry**; the guest continues from the next instruction.

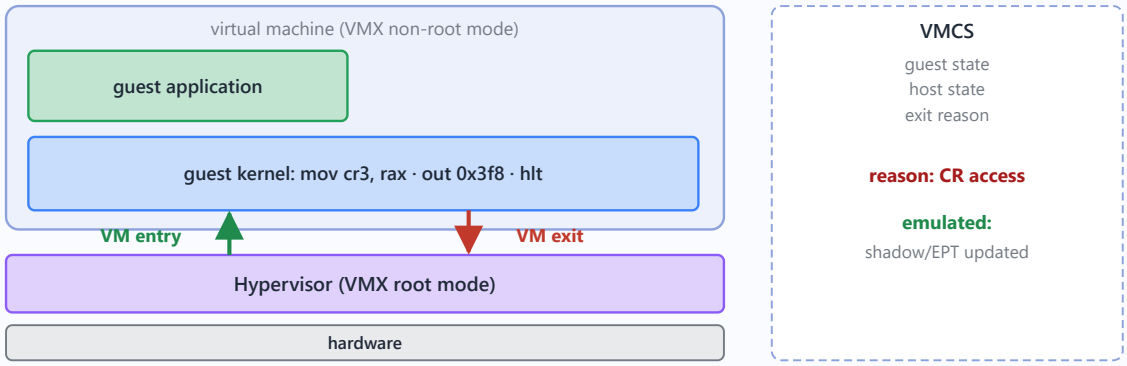


Figure 12.42: Trap and emulate. The guest's sensitive instruction causes a VM exit; the reason is written to the VMCS, the hypervisor emulates the instruction and returns to the guest with a VM entry.

Only the guest's sensitive instructions trap; the rest of its instructions run directly on the hardware, at full speed. Even so, a VM exit takes about a microsecond. A workload that exits often therefore slows down. Before hardware support, VMware (1999) solved this problem with **binary translation**. The code of the guest kernel is scanned before it runs, and sensitive instructions are replaced with safe code fragments that call the hypervisor. The translated code is cached and used again and again.

12.7.3 Paravirtualization

There is one more way to avoid the cost of trap and emulate: modify the guest operating system.

Definition 12.9

In **paravirtualization**, the guest operating system knows it is running on a hypervisor. Instead of sensitive instructions, it makes direct calls to the hypervisor; these calls are called **hypercalls**.

A hypercall is the analog of a system call one layer down. Just as an application requests service from the operating system, the guest operating system requests it from the hypervisor. Because there is no trap-and-emulate cost, the method is faster. The price is that it requires changes in the guest operating system's source code. Xen (2003) is the hypervisor that first popularized this approach; a modified version of Linux ran on Xen at nearly native speed (Figure 12.43).

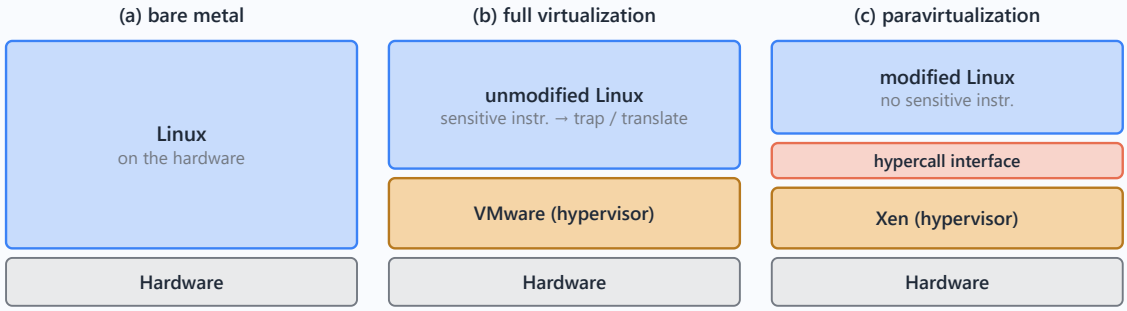


Figure 12.43: (a) Linux on bare metal, (b) unmodified Linux on VMware (sensitive instructions trap or are translated), (c) modified Linux on Xen using the hypercall interface.

It was a problem that every hypervisor had its own hypercall interface: separate versions of a kernel were needed. So that a single kernel could run both on bare metal and on different hypervisors, a common interface called VMI (Virtual Machine Interface) was proposed. When hardware support became widespread it was no longer needed, and VMI was removed from Linux in 2011. Today paravirtualization lives on mostly in drivers. `virtio` disk and network drivers, paravirtual clocks, and Hyper-V's "enlightenments" are examples. In today's clouds, CPU and memory are fully virtualized with hardware support. I/O is done either with paravirtual drivers such as `virtio` or by assigning a piece of a device directly to the virtual machine with SR-IOV.

12.7.4 Memory virtualization: shadow tables and EPT

Address translation in a virtual machine has two layers (Figure 12.44). A process in the guest produces a **guest virtual address** (GVA). The guest operating system's page tables translate it to a **guest physical address** (GPA). But the guest's "physical" memory is really just a piece of memory that the hypervisor has allocated. The hypervisor must translate the GPA to a real **host physical address** (HPA).

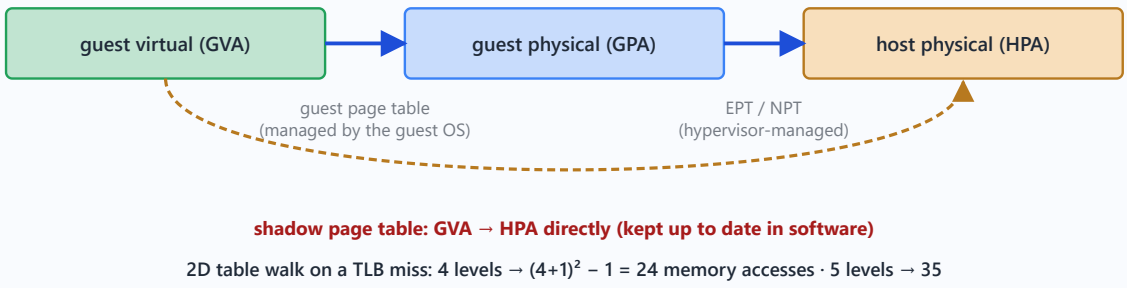


Figure 12.44: Two-level address translation: the guest page table GVA → GPA, EPT/NPT GPA → HPA. The shadow page table holds the GVA → HPA mapping directly.

The solution before hardware support was **shadow page tables**. For each guest process the hypervisor keeps a table that maps the GVA directly to the HPA, and gives the MMU this table. It makes the guest's own tables read-only. When the guest tries to change a table entry, it traps; the hypervisor reflects the change in the shadow table too. Translation is fast, but every table change by the guest means a VM exit. Workloads that create processes or take many page faults therefore slow down a lot.

Intel's EPT (Extended Page Tables, 2008) and AMD's NPT (Nested Page Tables) give this job to the hardware. The MMU walks the two tables together. The guest changes its own table without trapping; the hypervisor manages only the GPA → HPA table. The cost is in the TLB miss. Each entry of the guest table is a GPA; to be read, it must first go through the EPT.

Example 12.13

The guest and the host use 4-level page tables. On a TLB miss, with no table entry in the cache, how many memory accesses are needed in the worst case (excluding the final data access)?

1. **One level of the guest table.** The address of the entry is a GPA. It is first translated to an HPA by walking 4 EPT levels (4 accesses), then the entry itself is read (1 access): 5 accesses.
2. **Four levels.** $4 \times 5 = 20$ accesses. At the end the GPA of the data is obtained.
3. **The final translation.** The GPA of the data must go through the EPT as well: 4 more accesses. Total $20 + 4 = 24$.

The general formula is $(n + 1)(m + 1) - 1$, where n is the number of guest levels and m the number of EPT levels. For $n = m = 4$ it comes to $(4 + 1)^2 - 1 = 24$ accesses. If both sides have 5 levels, $(5 + 1)^2 - 1 = 35$ accesses are needed. Without virtualization, the same miss would need only 4 accesses.

There are two ways to reduce the impact of this long walk. Large pages (2 MiB or 1 GiB) reduce both the number of table levels and the TLB misses. Tagging TLB entries with the virtual machine's identifier (VPID on Intel, ASID on AMD), in turn, makes it unnecessary to flush the TLB when switching between virtual machines.

Hypervisors can also hand out more memory to guests than there is physical memory (overcommitment). Two mechanisms support this. The **balloon driver** is a driver running inside the guest. When the hypervisor wants memory, it allocates memory in the guest and gives these pages back to the hypervisor; the guest decides for itself what goes to disk through its own page replacement policy. Linux's **KSM** (Kernel Same-page Merging) mechanism finds pages with identical contents in different virtual machines and merges them into a single copy. If someone writes to the page, a separate copy is made by copy-on-write.

12.7.5 Containers and virtual machines

Each virtual machine runs its own kernel; it takes seconds to start and holds gigabytes of memory. For many applications this is excessive. Isolating applications from one another is often enough; a separate kernel is not needed.

Definition 12.10

A **container** is a group of processes that shares the host's single kernel and is isolated from other processes with the help of namespaces and cgroups.

Linux containers rest on three kernel mechanisms. **Namespaces** show a group of processes their own process numbers, file system tree, network interfaces, and host name; the group cannot see other processes. **Control groups** (cgroups) limit the amount of CPU, memory, and I/O the group can use. **seccomp**, for its part, restricts the system calls the group can make. A container holds the libraries and tools of a distribution, but the kernel is the host's (Figure 12.45).

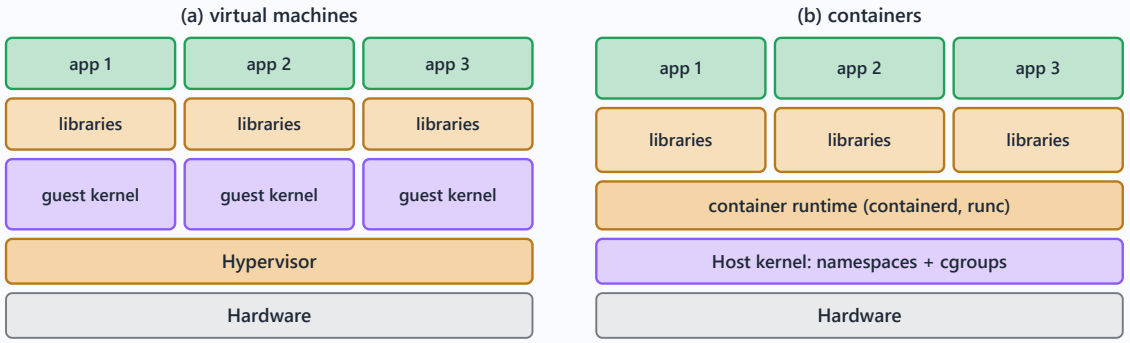


Figure 12.45: (a) Each virtual machine has its own guest kernel; (b) containers share the single host kernel.

	Virtual machine	Container
Isolation boundary	hardware level, separate kernel	same kernel: namespace, cgroup, seccomp
Startup / overhead	seconds, GB of memory	milliseconds, MB of memory
Different operating system	yes (Windows on Linux)	no (same kernel family)
Examples	KVM, Hyper-V, ESXi	Docker, Podman, Kubernetes pods

Table 12.6: Comparison of a virtual machine and a container.

A container's isolation is weaker than a virtual machine's. Because all containers share the same kernel, a vulnerability in the kernel affects all of them. Solutions in between are therefore spreading. Firecracker, used by AWS Lambda, creates small virtual machines (micro-VMs) that start within milliseconds. gVisor serves a container's system calls in a kernel in user space. Kata Containers runs each container in a lightweight virtual machine. WSL 2 on Windows also runs a real Linux kernel in a lightweight Hyper-V virtual machine; Linux containers on Windows run in this virtual machine.

Common mistake

Thinking a container has its own kernel. On a server running Linux 6.8, in a container based on Ubuntu 20.04, `uname -r` still shows 6.8; only the user space in the container is Ubuntu 20.04. A virtual machine on the same server, by contrast, boots its own guest kernel and shows its version.

12.8 Distributed systems

The nodes of a multicomputer are in the same room, under the management of the same organization, and connected by a private network. In a distributed system all of these assumptions are relaxed. This section describes distributed systems, the network that holds them together, and the four main kinds of middleware.

12.8.1 Distributed systems and middleware

A distributed system consists of autonomous computers that communicate over a network. Each computer manages its own operating system, memory, and resources. The system nonetheless tries to look to the user like a single, consistent system. The nodes may run different hardware and different operating systems, may belong to different organizations, and may be spread around the world.

Definition 12.11

Middleware is a software layer in a distributed system that sits between the operating systems and the applications and offers applications a model that is the same on all nodes.

Different operating systems can speak a common language only thanks to middleware (Figure 12.46). Middleware is based on one of four basic paradigms. In document-based middleware everything is a document; the Web is the example. In file-based middleware everything is a file; NFS and SMB are like this. In object-based middleware everything is an object; CORBA, Java RMI, and gRPC are of this kind. In coordination-based middleware, processes communicate through a shared space or a broker; Linda and publish-subscribe systems are of this kind.

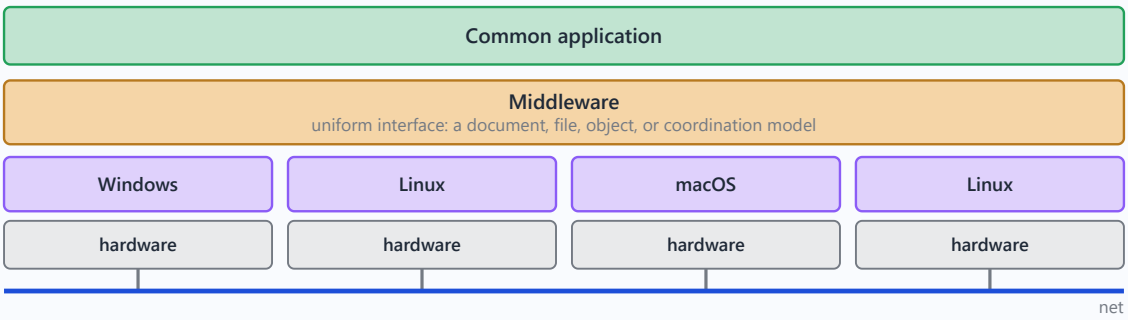


Figure 12.46: The position of middleware: a common layer on top of different operating systems and a common application on top of it.

True distributed operating systems that manage all the nodes like a single operating system were also tried. Amoeba and Plan 9 are the best known. These remained research systems. Today each node runs its own operating system; distribution is handled by middleware and cluster managers such as Kubernetes.

12.8.2 Comparing the three kinds of systems

This chapter has covered three parallel organizations: the multiprocessor, the multicomputer, and the distributed system. Table 12.7 puts them side by side following Tanenbaum's comparison.

Property	Multiprocessor	Multicomputer	Distributed system
Node structure	CPU	CPU, RAM, network interface	complete computer
Peripherals	all shared	shared (except perhaps disk)	full set at each node
Location	same rack	same room	possibly worldwide
Communication between nodes	shared RAM	dedicated interconnect	traditional network
Operating system	one, shared	many, the same	many, possibly different
File system	one, shared	one, shared	each node's own
Administration	one organization	one organization	many organizations

Table 12.7: Comparison of the three kinds of systems.

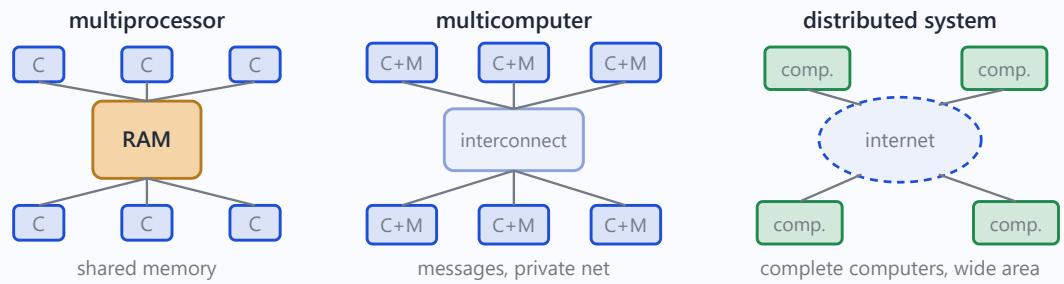


Figure 12.47: The three kinds of systems: CPUs attached to shared memory; nodes with their own memory

and a private interconnect; complete computers joined by a wide-area network.

12.8.3 Network hardware and encapsulation

A network binds the nodes of a distributed system. The most common kind of local network is Ethernet (Figure 12.48). In the classic Ethernet of 1976, all computers shared a single cable. A computer listened to the cable before sending and sent if it was idle. If two computers sent at the same time, a collision occurred, and both backed off for a random time and tried again. This method is called CSMA/CD. Today each computer is connected to a switch by its own cable. The links work at speeds between 1 and 400 Gb/s and in full duplex; there are no collisions.

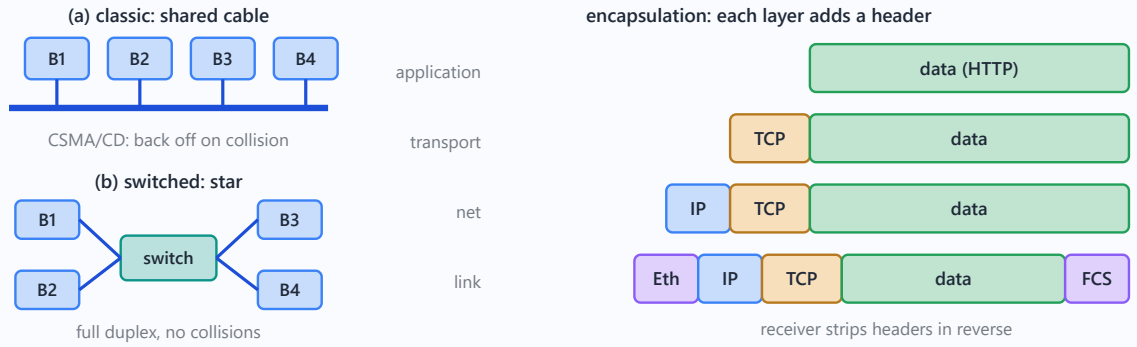


Figure 12.48: (a) Classic and (b) switched Ethernet; on the right, encapsulation, in which each layer adds a header to the packet.

Local networks are joined to one another with routers. Organizations' networks connect to Internet service providers, and they connect to the Internet backbone. The backbone consists of high-speed fiber links and exchange points where networks join one another.

Network software is organized in layers. When application data is sent, each layer adds its own header. The transport layer adds the TCP header, the network layer the IP header, and the link layer the Ethernet header and the trailing checksum (FCS). This is called **encapsulation**. On the receiving side the headers are stripped in reverse order and each layer interprets its own header.

12.8.4 Network protocols and service types

Network services are classified from two angles. In a **connection-oriented** service, a connection is first established, then data is sent, and finally the connection is terminated; it is like a telephone call. In a **connectionless** service there is no setup cost; each packet is sent independently, like mail. A **reliable** service guarantees that the data is delivered and in order; the receiver acknowledges each packet with an acknowledgment (ACK). An unreliable service does not give this guarantee. Table 12.8 summarizes the Internet's basic protocols.

Protocol	Layer	Service	Note
IP (v4/v6)	network	connectionless, unreliable	addressing and routing
TCP	transport	connection-oriented, reliable byte stream	acknowledgments, retransmission, flow and congestion control
UDP	transport	connectionless, unreliable datagram	has a checksum, no retransmission; DNS, games, audio
QUIC	transport (over UDP)	connection-oriented, reliable, encrypted	used by HTTP/3 (2022)
DNS	application	name → IP address	hierarchical, distributed, cached
HTTP/HTTPS	application	Web documents	HTTPS = HTTP + TLS, port 443
SMTP	application	email transfer	between servers; reading is by IMAP
FTP → SFTP	application	file transfer	FTP is unencrypted; SFTP/HTTPS has replaced it
SNMP	application	network device monitoring	v3 adds authentication and encryption

Table 12.8: The main Internet protocols.

UDP's checksum detects a corrupted packet, but does not correct it or resend it; the corrupted packet is simply dropped. Resending lost or corrupted packets is the job of TCP or the application.

12.8.5 Document-based middleware: the Web

On the Web everything is a document. Documents are linked to one another by links and form a giant directed graph. When the user types the address `www.example.org/index.html` into the browser, this happens (Figure 12.49):

1. The browser asks DNS, through the operating system's resolver, for the IP address of `www.example.org`.
2. The reply arrives: 203.0.113.10. The reply is kept in a cache for a while.
3. The browser opens a TCP connection to the server's port 443 and performs the TLS handshake. In HTTP/3 these two steps are merged into one by QUIC.
4. The browser sends the `GET /index.html` request.
5. The server returns the HTML file and the browser begins to display the text.
6. The images, CSS, and scripts that the HTML refers to are fetched over the same connection; HTTP/2 and HTTP/3 carry them in parallel streams. The connection stays open for a while.

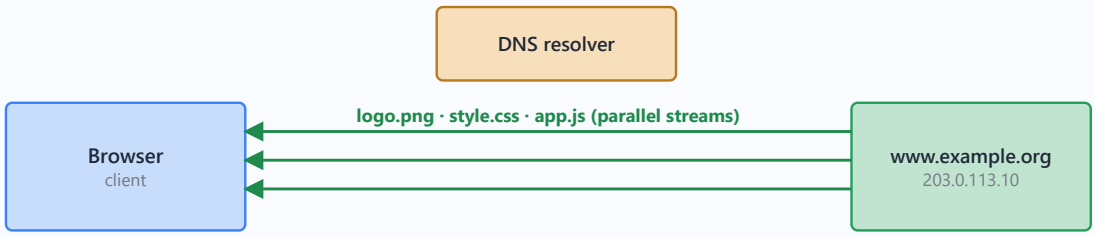


Figure 12.49: The last step of fetching a web page: `logo.png`, `style.css`, and `app.js` arrive over the same connection in parallel streams.

In old textbooks, connections are made to port 80 and a new TCP connection is opened and closed for every object. This belongs to the HTTP/1.0 era. Today the default is HTTPS (port 443); connections are persistent, and HTTP/2–3 use multiplexing.

12.8.6 File-system-based middleware

File-based middleware makes the distributed system look like a file system. The first design question is how the file will be accessed (Figure 12.50). In the **upload/download model**, the client copies the whole file to itself, reads and writes it locally, and when finished uploads the new version back to the server. This model is simple, but moving an entire large file to read a few bytes of it is expensive. FTP and the old AFS use this model. In the **remote access model** the file stays on the server. The client sends `read` and `write` requests over the network, and only the blocks it wants are moved. NFS and SMB use this model.

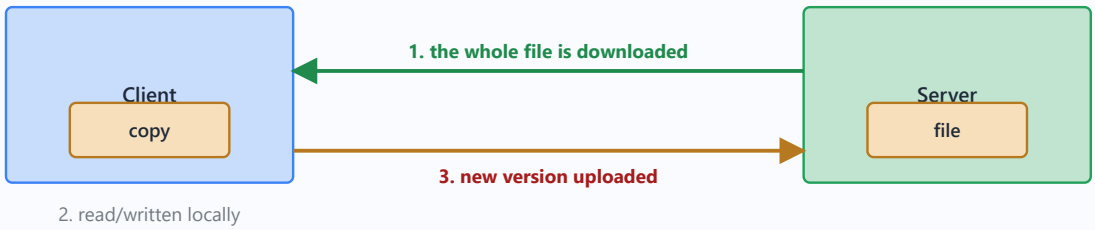


Figure 12.50: At the top, the upload/download model: the whole file is downloaded, used locally, and uploaded back. At the bottom, the remote access model: the file stays on the server and requests go over the network.

The second question is naming. In the simplest solution the file name includes the machine name: `machine:/path`. This name is not location transparent; if the file is moved to another machine, its name changes. A better solution is to mount the remote file system on a directory of the local tree. If all clients mount the servers at the same place, everyone sees a single, common name space. If clients mount different things, each sees a different view (Figure 12.51). The same path can point to different files on different clients. The most powerful solution is a single name space that is the same on all machines and contains no location of the file at all. This is called location independence. Present-day examples are NFSv4, SMB 3, and object storage systems such as Amazon S3.

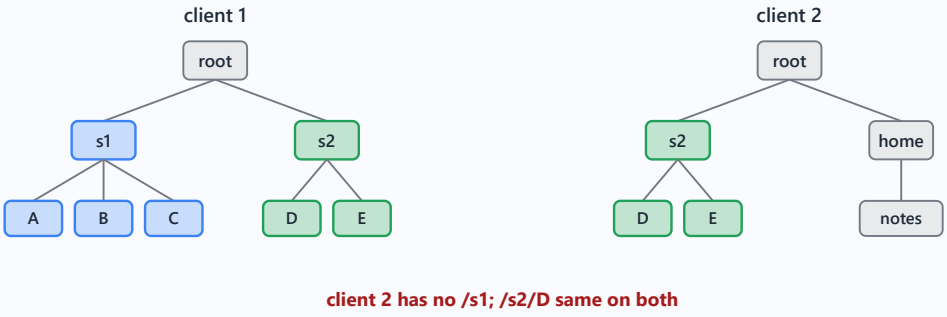


Figure 12.51: Two clients have mounted different things: on client 2 there is no `/s1`, while `/s2/D` is the same file on both clients.

12.8.7 The semantics of file sharing

When several clients share the same file, it must be defined when a write becomes visible to others. On a single machine the answer is clear. If process A appends "c" to a file and then B reads, B sees "abc". Every read sees the most recent write. This is called **UNIX semantics**, or sequential consistency (Figure 12.52, top).

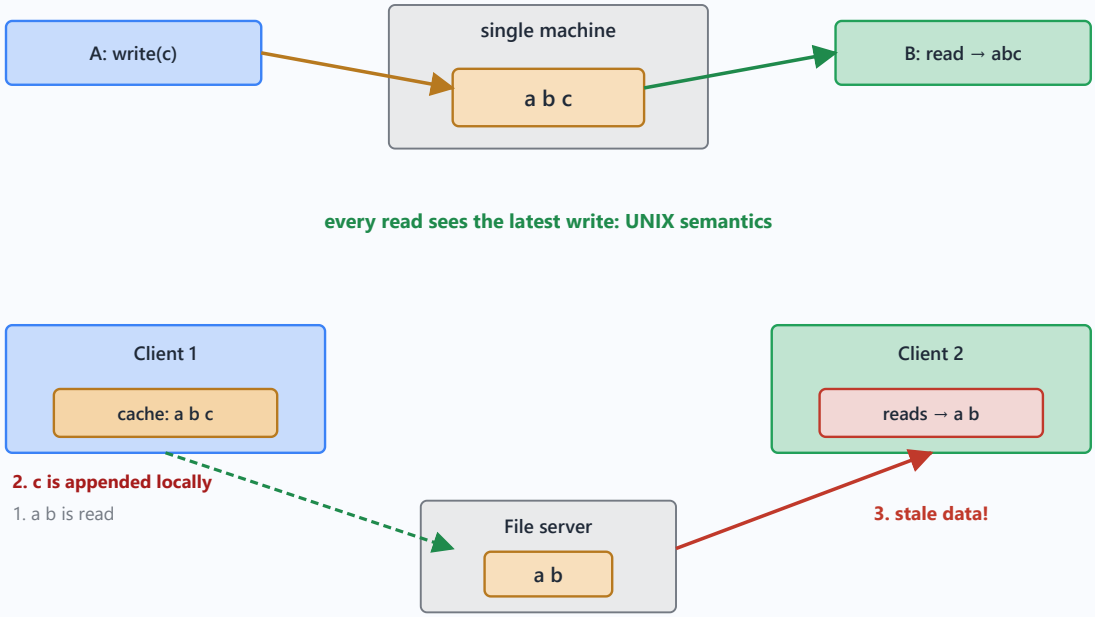


Figure 12.52: At the top, UNIX semantics on a single machine. At the bottom, client 1 has appended "c" in its own cache; the server does not know this, and client 2 reads the stale "ab".

In a distributed system, clients cache files for performance. Client 1 modifies the file in its cache, but the server does not yet know. Client 2 reads from the server and sees the stale "ab". The only way to preserve UNIX semantics is not to use a cache and to send every operation to the server; this is very slow. Distributed systems therefore use looser semantics (Table 12.9).

Approach	Rule	Examples
UNIX semantics	every write is visible to everyone at once	single machine; remote access without a cache
Session semantics	changes become visible when the file is closed	AFS; NFS's close-to-open consistency
Immutable files	a file cannot be written, a new one is created	object storage, Git objects
Transactions	all or nothing, serializable	databases

Table 12.9: File sharing semantics.

In **session semantics**, a client's changes are visible only to itself until it closes the file. After the file is closed, clients that open it see the new version. With **immutable files** the problem never arises, because a file is never modified; a change produces a new file. In **transactions**, a series of operations is done either entirely or not at all, and concurrent transactions give results as if they had been done in sequence. Each solution sits at a different point between consistency and performance.

12.8.8 Object-based middleware: CORBA

In object-based middleware, the distributed system is a collection of objects. A client calls a method of an object on another machine as if it were a local object's. The most comprehensive example of this approach is CORBA, defined by the OMG in 1991 (Figure 12.53).

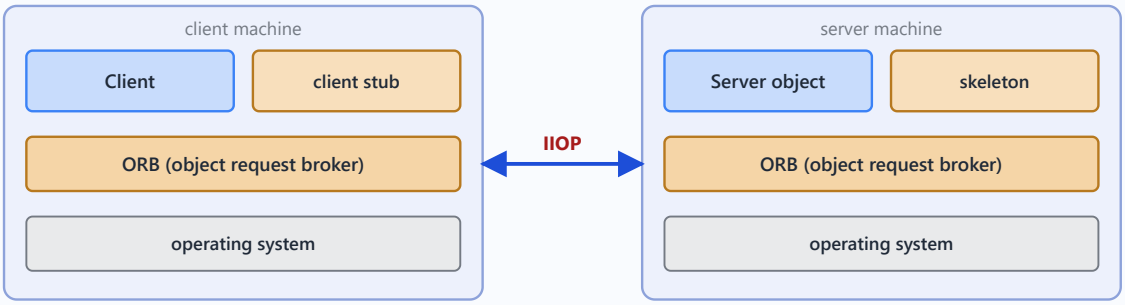


Figure 12.53: The basic elements of CORBA: the client and client stub, the server object and its skeleton, the ORBs on both sides, and the IIOP protocol between them.

- **IDL** (Interface Definition Language): the interfaces of objects are defined in a language independent of the programming language. The IDL compiler generates from this definition the client stub and the **skeleton** on the server side.
- **ORB** (Object Request Broker): it finds the object's location, packs the call, and carries it to the other side with the IIOP protocol. The client does not know which machine the object is on; this is called network transparency.
- **Static invocation** (SII): the client uses stubs generated from IDL at compile time. It is fast, and types are checked at compile time.
- **Dynamic invocation** (DII): the client discovers the interface it will call from an interface repository at run time and builds the call itself.

SII and DII are two separate mechanisms; SII is not a compiled version of DII. CORBA lives on today mostly in legacy systems. In new systems gRPC with Protocol Buffers, REST, and Java RMI are preferred. But ideas such as generating stubs from IDL, an ORB-like broker, and location-independent object references have passed into these systems.

12.8.9 Coordination-based middleware: Linda

In coordination-based middleware, processes do not know one another. Instead of communicating directly, they put data into a common environment and take data from it. The classic example of this approach is Linda, introduced by Gelernter in 1985.

Definition 12.12

The **tuple space** is, in Linda, a persistent set of tuples shared by all processes. A **tuple** is a record made of fields of various types: for example `("abc", 2, 9)`.

Linda has four basic operations (Figure 12.54):

- `out` puts a tuple into the space and does not block.
- `in` searches for a tuple that matches a **template**, removes it from the space, and assigns its fields to variables. If there is no matching tuple, the caller blocks and waits until a suitable tuple is put in.
- `rd` works like `in`, but does not remove the tuple from the space; it only reads it.
- `eval` starts a new process; when the process finishes, its result is put into the space as a tuple.

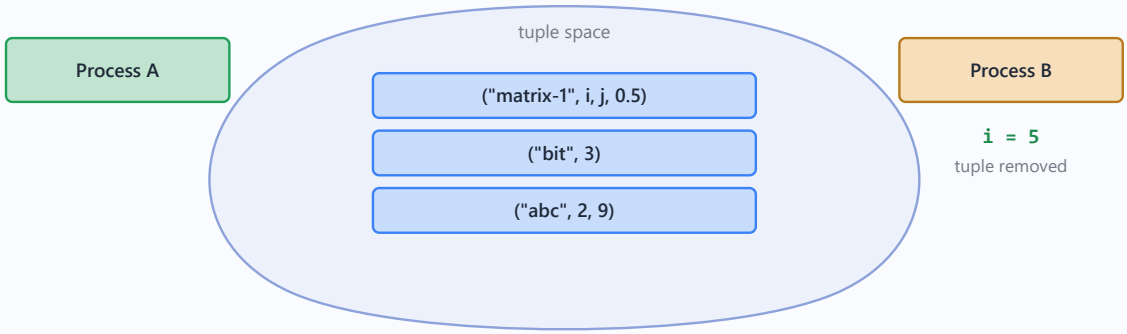


Figure 12.54: The tuple space. Process B called `in("abc", 2, ? i)`; the matching tuple `("abc", 2, 5)` was taken from the space and i became 5.

Three conditions are required for a template to match a tuple. The numbers of fields must be the same. Corresponding fields must be of the same type. The constant fields in the template must also equal those in the tuple. Fields of the form `? i` are **formal parameters**: they match any value of the right type, and that value is assigned to i. If more than one tuple matches, Linda does not define which will be chosen.

With these simple operations, synchronization comes for free. For example, a semaphore can be implemented with identical tuples put into the space: `out("sem")` is the V operation and `in("sem")` the P operation. A work pool is built in a similar way: a manager puts jobs in with `out`, and workers take them with `in`.

12.8.10 Code trace: Linda `in()`

The function below is the search part of `in()` in a simple Linda implementation that keeps the tuple space as a linked list. `s` is the template; `n` holds the number of fields, `type[k]` the type of the k-th field, and `fixed[k]` whether that field is a constant.

```
Tuple *in_find(Template *s) {
  for (Tuple *d = space; d; d = d->next) {
    if (d->n != s->n) continue;
    int ok = 1;
    for (int k = 0; ok && k < s->n; k++)
      if (d->type[k] != s->type[k] ||
          (s->fixed[k] && !equal(d, s, k)))
        ok = 0;          /* field mismatch */
    if (ok) { remove(d); return d; }
  }
  return NULL;           /* block */
}
```

The outer loop walks over the tuples in the space in turn. If the number of fields differs, the tuple is skipped at once (`continue`). Otherwise the inner loop compares the fields one by one. If the type differs, or the field is a constant and its value differs, `ok` becomes 0 and the inner loop stops. If all fields match, the tuple is removed from the space and returned. If no tuple matches, the function returns `NULL`; the calling process blocks.

Example 12.14

The space holds, in order, d1 = ("abc", 2), d2 = ("xyz", 2, 7), d3 = ("abc", 1, 4), d4 = ("abc", 2, 5), and d5 = ("abc", 2, 9). A process calls `in("abc", 2, ? int i)` three times.

Call	Tuple	Check	Result
1	d1	2 fields ≠ 3 fields	`continue`
1	d2	field 0: "xyz" ≠ "abc"	ok = 0
1	d3	field 0 matches; field 1: 1 ≠ 2	ok = 0
1	d4	constants equal, field 2 is an integer	d4 is removed, i = 5
2	d1, d2, d3	as above	skipped
2	d5	constants equal, field 2 is an integer	d5 is removed, i = 9
3	d1, d2, d3	as above	`NULL` : the process blocks

On the first call d5 would have matched too, but d4 comes first in the list, so it was taken. The third call proceeds only when someone puts a matching tuple with `out`. (The search loop was transcribed one-to-one into Python and run on the same space.)

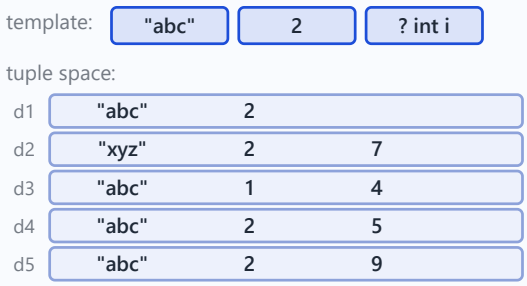


Figure 12.55: The start of the trace: the template and the five tuples in the space.

Common mistake

Looking only at values when matching. The template `in("abc", ? int x, 5)` does not match the tuple `("abc", 2.0, 5)`: the type of the second field is floating-point, not integer. The type check comes before the value check.

12.8.11 Publish-subscribe and Jini

The most common form of coordination today is the **publish-subscribe** model (Figure 12.56). A publisher publishes a message to a topic. A subscriber subscribes to the topics it is interested in. The **broker** in between delivers each message to all subscribers of that topic. The publisher and the subscriber do not know each other's identity; nor do they need to be running at the same time. These two properties are called decoupling in space and in time.

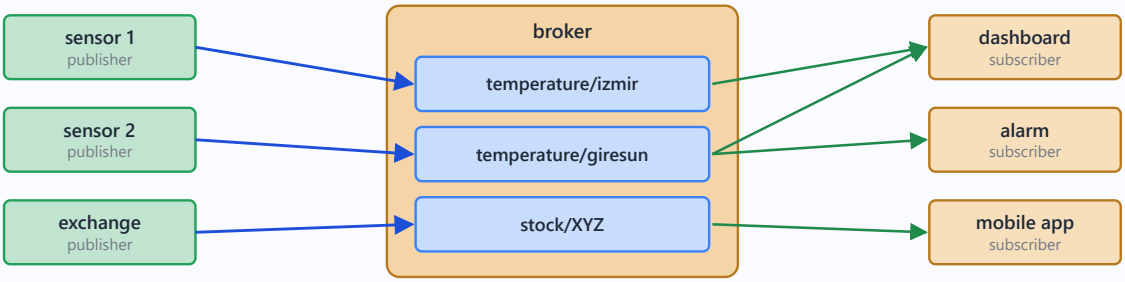


Figure 12.56: Publish-subscribe: sensors and the stock exchange publish to topics, and the broker distributes the messages to the dashboard, alarm, and mobile app subscribed to the topic.

Subscribers and topics can be added and removed at run time; the system is therefore flexible and scalable. The delivery guarantee depends on the broker's configuration: at most once, at least once, or storing messages in a persistent queue. Present-day examples are MQTT for the Internet of Things, Apache Kafka, RabbitMQ, and the queue services of cloud providers.

Jini is a coordination system introduced by Sun in 1998. Java devices that join the network find a lookup service without prior configuration and register the services they offer with it. A client finds a suitable service from the lookup service and uses it through Java RMI. Registrations are made with a **lease** period. If the lease is not renewed, the registration lapses by itself; this way crashed devices are cleaned out of the service list. Jini was continued under the name Apache River but is not active today. Its ideas live on in service discovery: mDNS/DNS-SD, Consul, and Kubernetes services.

12.8.12 The cloud and clusters

A **cluster** is a set of servers connected by a fast network and under single management; high-performance computing clusters and data centers are examples. The **cloud** is the offering of these resources over the network, on demand and pay-as-you-go. AWS, Azure, and Google Cloud are the largest cloud providers.

Cloud services are distinguished by how many of the layers the provider manages (Figure 12.57). In an on-premises system, the user manages every layer. In IaaS (Infrastructure as a Service), the provider manages the servers, storage, network, and virtualization; the user manages everything from the operating system up in a virtual machine. In PaaS (Platform as a Service), the user brings only the application and its data. In FaaS (Function as a Service), the user writes only individual functions; the provider runs them on every request. In SaaS (Software as a Service), the user uses a ready-made application and manages no layer.

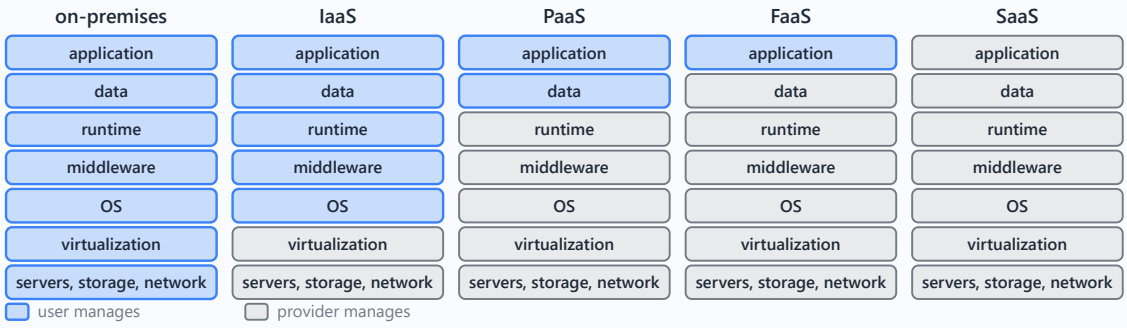


Figure 12.57: Cloud service models: on-premises, IaaS, PaaS, FaaS, and SaaS. The user manages the blue layers and the provider the gray layers.

The infrastructure of the cloud is the topics of this chapter coming together. The physical servers are multicore NUMA machines. On them hypervisors such as KVM or Nitro run virtual machines. On top of the virtual machines or the servers, containers run. Kubernetes schedules these containers (pods) onto nodes, restarts those that crash, and distributes the load; in a sense it is the present-day counterpart of the distributed operating system. The nodes are connected by RDMA-capable networks, workloads are distributed with load balancers, and services communicate through publish-subscribe queues.

12.9 Summary

- Parallel systems are built in three organizations: the shared-memory **multiprocessor**, the message-passing **multicomputer**, and the wide-area **distributed system**. Since clock speed has stood still since 2005, the transistor budget goes into the number of cores. According to **Amdahl's law**, the speedup is $1 / ((1 - p) + p / n)$ and cannot exceed $1 / (1 - p)$.
- In **UMA** machines every memory word is reached in the same time. A single bus saturates with a few CPUs. A **crossbar** is nonblocking but needs n^2 crosspoints. An **omega network** is built from $(n/2) \cdot \log_2 n$ switches, routes by the bits of the destination address, and is blocking.
- In **NUMA**, remote access is slower than local; the average access is $f \cdot t_{local} + (1 - f) \cdot t_{remote}$. The operating system keeps data next to the work with first touch, affinity, and automatic NUMA balancing. In CC-NUMA, coherence is maintained with a **directory**; messages go only to the nodes concerned.
- **Cache coherence** means writes to the same address are seen in the same order by everyone. In **MESI**, writing a line in E is silent, writing a line in S requires invalidation, and a line in M is also written to memory when it is given to someone else.
- In **false sharing**, independent variables sit in the same 64-byte line, so the line bounces between cores. The fix is alignment and padding.
- Multicore chips are made of chiplets, big and little cores, and accelerators. **SMT** presents one core as several logical CPUs; the gain is 10–30% and it carries a security risk.
- There are three operating system models. A separate operating system per CPU offers no sharing. In **master-slave**, the master is a bottleneck. In **SMP**, everyone runs the kernel and shared structures are protected by fine-grained locks.
- On a multiprocessor, **TSL** must lock the bus or the line for the duration of the read and the write. The **ticket lock** is fair; **TTAS** makes waiters spin on a local copy; **MCS** gives each waiter its own flag. If the expected wait is shorter than 2C, spinning is cheaper, and if longer, sleeping is; spinning for 2C and then sleeping gives twice the cost in the worst case.
- A shared ready queue balances load but suffers from a single lock and cold caches; today's systems use per-CPU queues and load balancing. **Space sharing** allots CPUs to a job until it finishes. In **gang scheduling**, interdependent threads run in the same slice on different CPUs.
- In a multicomputer, messages can be copied five times; RDMA removes the copies. The diameters of the topologies are $n / 2$ for a ring, $2(k - 1)$ for a $k \times k$ grid, and $\log_2 n$ for a hypercube. **E-cube** routing corrects the differing bits from left to right. **Cut-through** removes the latency of store-and-forward, which grows with the number of hops.
- Send can be blocking or nonblocking. **RPC** makes a remote call look like a local one with stubs; pointers and partial failures cause difficulties. **DSM** creates the illusion of shared memory with page faults. Process assignment is a graph partitioning problem; distributed load balancing can be sender-initiated or receiver-initiated.

- **Hypervisors** are divided into type 1 (bare metal) and type 2 (on a host). x86, which did not satisfy the Popek–Goldberg condition, was virtualized first with binary translation and then with VT-x/AMD-V. **Paravirtualization** uses hypercalls. **EPT** removes the VM exits of shadow tables, but a TLB miss rises to $(n + 1)(m + 1) - 1$ accesses. **Containers** share a single kernel.
- In a distributed system, **middleware** offers a document (Web), file (NFS, SMB), object (CORBA, gRPC), or coordination (Linda, publish-subscribe) model. In file sharing, UNIX semantics is expensive; session semantics, immutable files, and transactions are tradeoffs. The cloud offers all these layers as a service.

12.10 Key terms

Amdahl's law symmetric multiprocessor (SMP)
uniform / non-uniform memory access (UMA / NUMA) crossbar switch omega network
cache coherence snooping / directory MESI protocol false sharing multicore
simultaneous multithreading (SMT) master-slave model spinlock ticket lock processor affinity
load balancing space sharing gang scheduling multicomputer hypercube message passing
remote procedure call (RPC) distributed shared memory hypervisor paravirtualization
virtual machine container middleware tuple space publish-subscribe

12.11 Test Yourself

1. 90% of a program can be parallelized. What is the speedup on 16 cores, and what is the efficiency per core (speedup / n)? What is the upper limit of the speedup?

   Answer

   $S(16) = 1 / (0.1 + 0.9 / 16) = 1 / (0.1 + 0.05625) = 1 / 0.15625 = 6.4$. The efficiency is $6.4 / 16 = 0.4$: only 40% of the cores' power is used. The upper limit is $1 / 0.1 = 10$.

2. In an 8 × 8 omega network, CPU 110 accesses memory 001. Through which switches, and by which outputs, does the request pass? If CPU 000 wants to read memory 000 at the same time, is there a conflict?

   Answer

   110 comes by the shuffle to input 101, that is, to 1C. Since the bits of the destination 001 are 0, 0, 1, the path is 1C (upper) → 2A (upper) → 3A (lower). The request from CPU 000 follows the path 1A (upper) → 2A (upper) → 3A (upper). Both requests use the upper output of 2A; one waits. A conflict has occurred inside the network even though the destination modules are different.

3. In a NUMA machine, local access takes 90 ns and remote access 150 ns. If 80% of accesses are local, what is the average time? What must the local access ratio be at least so that the average does not exceed 100 ns?

   Answer

   $0.8 \cdot 90 + 0.2 \cdot 150 = 72 + 30 = 102$ ns. From the condition $90f + 150(1 - f) \le 100$ we get $150 - 60f \le 100$, that is, $f \ge 50 / 60 \approx 0.833$. At least about 83.3% of accesses must be local.

4. In a directory-based system with 256 nodes, 16 MiB per node, and 64-byte lines, which node's memory holds the line of address 0x5A0003C4? What are the block number and the offset?

   Answer

   The leftmost 8 bits are 0x5A = 90: the line is at node 90. The remaining 24 bits are 0x0003C4 = 964. Since $964 = 15 \cdot 64 + 4$, the block is 15 and the offset is 4. The request goes to node 90, and entry 15 of its directory is consulted.

5. In a three-core MESI system, x is initially only in memory and is 0. In turn, C0 reads x, C0 writes x = 5, C1 reads, C2 reads, C1 writes x = 6. At the end, what are the states of the line, how many bus operations have occurred, and what value is in memory?

   Answer

   C0 reads: E (BusRd). C0 writes: E → M, silent. C1 reads: BusRd; C0 supplies the data and writes it to memory, and both are S. C2 reads: BusRd, all three S. C1 writes: BusUpgr; C0 and C2 become I, C1 becomes M. The final states are I, M, I. Four bus operations occur. Memory holds 5; the current value 6 is only in C1.

6. The context switch cost is C = 2 μs. If the expected wait for a lock is 3 μs, and if it is 6 μs, which is cheaper: spinning or sleeping?

   Answer

   The cost of switching is $2C = 4$ μs. Since 3 μs < 4 μs, spinning is cheaper in the first case. Since 6 μs > 4 μs, sleeping is cheaper in the second. If the wait time is not known, spinning for 4 μs and then sleeping gives, in the worst case, twice the ideal cost.

7. On a system with 4 CPUs, four jobs are to run with gang scheduling: A has 3 threads, B 2, C 2, and

D 1. What is the minimum number of slices in a round, and what is the CPU utilization?

Answer

There are 8 threads in total and 4 CPUs per slice; at least $\lceil 8 / 4 \rceil = 2$ slices are needed. Placement: in slice 0 A (3) and D (1), in slice 1 B (2) and C (2). Since every slice is full, utilization is 100%.

8. Through which nodes does a message going from 0101 to 1010 pass with e-cube routing in a 4-dimensional hypercube? Why is this path the shortest?

Answer

$0101 \oplus 1010 = 1111$; all four bits differ. The bits are corrected from left to right: $1101 \to 1001 \to 1011 \to 1010$. Each hop corrects a single differing bit and no hop increases the difference; so the number of hops equals the Hamming distance, that is, 4. This is the diameter of the 4-dimensional hypercube.

9. A 1 KiB packet goes over 4 switches, that is, 5 links. The links are 1 Gb/s and each switch processes the header in 0.1 µs. What are the latencies of store-and-forward and cut-through?

Answer

Crossing one link takes $1024 \times 8 / 10^9$ s = 8.192 µs. In store-and-forward the whole packet is waited for on each link: $5 \times 8.192 + 4 \times 0.1 = 40.96 + 0.4 = 41.36$ µs. In cut-through the packet flows once and only the header delays are added: $8.192 + 4 \times 0.1 \approx 8.59$ µs.

10. A 32-bit guest operating system uses 2-level page tables; the host's EPT has 4 levels. On a TLB miss, with no entry in the cache, how many memory accesses are needed in the worst case (excluding the final data access)?

Answer

For each level of the guest table, the GPA of the entry is translated with 4 EPT accesses and the entry is read with 1 access: $2 \times 5 = 10$. The GPA of the data is also translated with 4 accesses. Total $10 + 4 = 14 = (2 + 1)(4 + 1) - 1$.

12.12 Further reading

- A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, Chapter 8 “Multiple Processor Systems” and Chapter 7 “Virtualization and the Cloud”. The book is the original source of the omega network, directory, gang scheduling, and load-balancing examples in these notes.

- R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, the chapters "Multiprocessor Scheduling", "Locks", and "Distributed Systems": a short and clear treatment of cache affinity, lock implementations, and RPC; the book is free at ostep.org.
- J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Chapter 5: examines snooping and directory-based coherence protocols, memory consistency models, and synchronization instructions in detail from the hardware side.
- J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM TOCS*, 1991: the classic paper that compares ticket and MCS locks with measurements.

12.13 References

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 8: Multiple Processor Systems; Chapter 7: Virtualization and the Cloud.
2. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Section 5.5 (Multi-Processor Scheduling), Chapter 18 (Virtual Machines), Chapter 19 (Networks and Distributed Systems).
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Multiprocessor Scheduling", "Locks", "Distributed Systems", "Sun's NFS", "AFS", appendix: "Virtual Machine Monitors".
4. J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2017 — Chapter 5: Thread-Level Parallelism (coherence, MESI, directory protocols).
5. J. M. Mellor-Crummey, M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM TOCS*, 1991; A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls", *ACM TOCS*, 1984; G. J. Popek, R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures", *CACM*, 1974.
6. Linux kernel documentation: "CFS/EEVDF Scheduler", "Scheduler Domains", "NUMA Memory Policy", "Core Scheduling"; *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 3 (VMX, EPT).

Security

The operating system is the guardian of every resource on a computer. It decides which process may read which file, which user may log in, and which code runs in kernel mode. If those decisions are wrong, even the best scheduler or the fastest file system is worthless: data leaks, gets tampered with, or becomes unavailable. These notes first define the goals and threats of security. They then present the cryptographic tools an operating system relies on: encryption, hashing, signatures, certificates, and TLS. The protection section covers the access matrix, access control lists, capabilities, and the principle of least privilege. Formal models and covert channels follow. The final sections treat authentication, memory-safety vulnerabilities, malware, and today's defenses: sandboxes, containers, mandatory access control, and secure boot.

LEARNING OUTCOMES

Goals and threats
Distinguish the goals of confidentiality, integrity, and availability, the types of attack, and the profiles of attackers.

Cryptography
Explain symmetric and asymmetric encryption, hash functions, digital signatures, certificates, and the TLS handshake; work through a Diffie-Hellman exchange step by step.

Protection
Read the access matrix in its ACL and capability forms; apply least privilege, setuid, and Linux capabilities.

Formal models
Apply the Bell-LaPadula and Biba rules; explain why covert channels cannot be closed completely.

Authentication
Compare salted slow password hashes, MFA, TOTP, and FIDO2 passkeys.

Attacks and defenses
Recognize memory-safety errors, TOCTOU, and malware; recognize canaries, NX, ASLR, CFI, sandboxing, and secure boot.

13.1 Security goals and threats

Before protecting a system, we have to say what is being protected, against whom, and for what purpose. This section introduces the three basic goals of security, the kinds of events that violate them, and the attackers behind those events. It ends with the overall architecture of defense: layered defense and the trusted computing base.

13.1.1 Security and protection

Operating-systems textbooks distinguish two closely related notions. **Protection** consists of mechanisms inside the system: rules that determine which process may access which object with which right, and that enforce those rules. File permission bits, the read-only bit in a page table, and the split between kernel and user mode are all protection mechanisms. **Security** is a broader question. How well does the system withstand deliberate attacks and accidents, from inside and outside? Even if the protection mechanisms work flawlessly, the system is not secure when a password is stolen, an administrator is tricked, or no backup was made.

This distinction has a practical consequence. Protection is a subsystem that the operating system implements directly and whose correctness can be proved. Security covers people, networks, hardware, and every bug in the software; it can never be fully achieved, and we can only raise the cost of an attack. These notes treat the two together. Goals and threats come first, then protection mechanisms. At the end we describe the attacks that try to get past those mechanisms and the defenses developed against them.

13.1.2 The CIA triad

The goals of security are usually summed up in three words. Because of their initials, the three are called the **CIA triad** (Figure 13.1).

Definition 13.1

Confidentiality means that data can be read only by authorized parties. **Integrity** means that data can be modified only by authorized parties and only through authorized channels. **Availability** means that legitimate users can reach the data and the service whenever they need them.

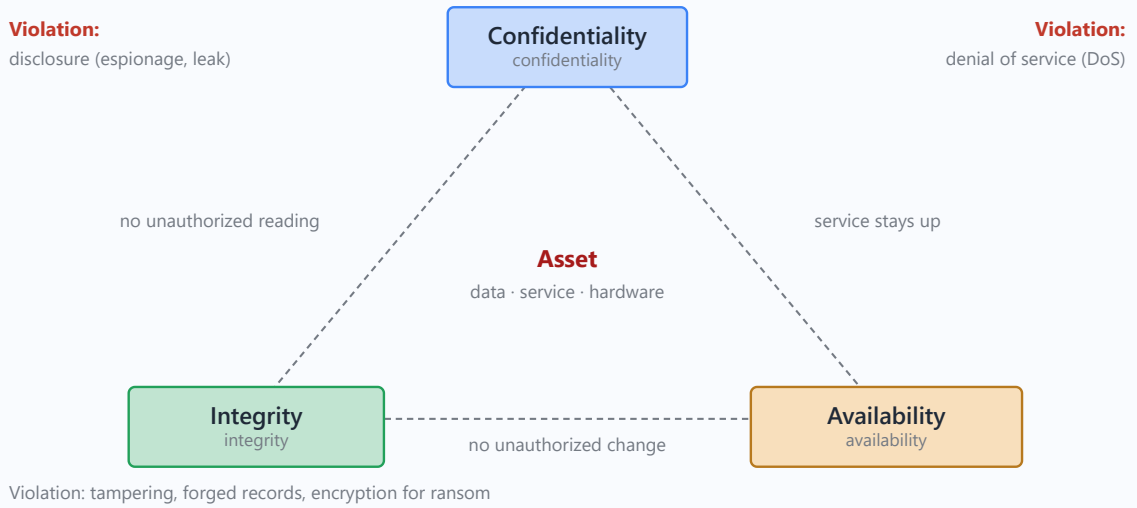


Figure 13.1: The CIA triad. The protected asset may be data, a service, or hardware; violating each goal is a distinct kind of harm.

Each goal has a matching kind of violation. A breach of confidentiality is *disclosure*: a company's customer database is leaked, or spyware copies e-mail. A breach of integrity is *tampering*: a student changes his grade record, or an attacker replaces a system program with his own version. A breach of availability is *denial of service*: a web server is swamped with fake requests, or ransomware encrypts a disk and makes it unusable.

The three goals are not independent, and sometimes they conflict. Keeping copies of data in three countries improves availability, but it also raises the number of places where a leak can occur to three. Encrypting a disk improves confidentiality; if the key is lost, the data becomes inaccessible forever. Security design is therefore a matter of balancing the goals.

Two more goals are often added to the triad. **Authenticity** means that a message or a user can be verified to come from the claimed source. **Accountability** means that it can be proved afterwards who performed an action. Accountability has two tools: audit logs and non-repudiation. With non-repudiation, a person cannot deny a signed document later by saying “I did not sign it.” This is the real benefit of a digital signature.

Example 13.1

Consider four events involving the records on a hospital's file server. Which goal does each violate?

1. **An employee opens and reads the test results of a celebrity out of curiosity.** The data did not change and the service did not stop, but an unauthorized person read it. This is a confidentiality violation.

2. **An attacker changes a patient's blood type.** The data that gets read is now wrong, and nobody notices. This is an integrity violation, and it can be more dangerous than a confidentiality violation.
3. **Ransomware encrypts all files; doctors cannot reach the records for hours.** The data did not leave the building and its content was not corrupted, but it cannot be used. This is an availability violation.
4. **A doctor later claims not to have written a prescription, and the records contain no trace that would refute this.** Accountability is missing. If prescriptions were signed with the doctor's private key, denial would be impossible.

13.1.3 Threat, vulnerability, attack, and exploit

Four words are easily confused in the language of security, but each is a separate link in a chain (Figure 13.2).

Definition 13.2

A **threat** is a possible harm. A **vulnerability** is the weakness that makes the harm possible. An **attack** is an attempt to use the vulnerability. An **exploit** is the code or technique that carries out that attempt.

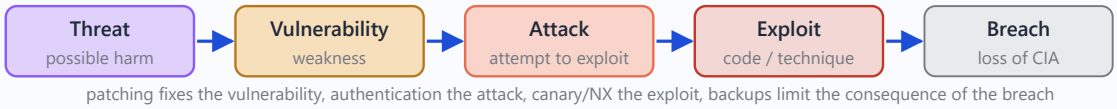


Figure 13.2: The chain from threat to breach. Each link has its own defense: patching addresses the vulnerability, authentication the attack, canaries and NX the exploit, and backups limit the consequence of the breach.

An example makes the chain concrete. “Theft of customer data” is a threat. An unchecked input field in a web application is a vulnerability. The attacker's writing a specially crafted query into that field is an attack. The query itself, that is, the string or program that triggers the vulnerability, is the exploit. If the database is finally exfiltrated, confidentiality has been violated.

The value of the chain view is that a defense can be placed at every link. Patching removes the vulnerability. Strong authentication makes it harder for the attacker to get into the system. Measures such as stack canaries and non-executable pages stop the exploit from working. Backups limit the consequence of a breach even when everything else fails. Even if the defense at one link is broken, the later links remain in play.

As in the slides, breaches can be grouped into seven categories (Table 13.1). The first three correspond directly to the CIA goals. The next four undermine those goals indirectly.

Category	Definition	Example
Breach of confidentiality	Unauthorized reading of data	Leak of a credit-card database
Breach of integrity	Unauthorized modification of data	Record tampering in a grading system
Breach of availability	Destruction of data or service	Ransomware encrypting a disk
Theft of service	Unauthorized use of resources	Mining cryptocurrency on a compromised server
Denial of service (DoS/DDoS)	Blocking legitimate use	SYN flood from a botnet
Masquerading	Pretending to be someone else	Password stolen by phishing
Privilege escalation	Gaining more authority than one is entitled to	Becoming root through a bug in a setuid program

Table 13.1: Categories of breach.

In a **denial of service** (DoS) attack the goal is not to read or modify data. The attacker keeps the system so busy that it cannot answer legitimate requests. A TCP server keeps a half-open connection record for every connection request. The attacker sends thousands of fake requests (SYNs) per second and never completes the handshake. The table fills up and real users cannot connect. If the attack comes from thousands of compromised machines at once, it is called a **distributed denial of service** (DDoS) attack. Blocking a single source address then does not help either.

In **masquerading**, the attacker takes on the identity of another user or machine. **Privilege escalation** is when an ordinary user or process reaches administrator rights by exploiting a bug. For an operating system this is the most dangerous category, because with administrator rights all the other protections can be switched off.

13.1.4 Network attacks

Almost every computer today is connected to a network, and the channel through which the operating system talks to the outside world is part of the attack surface. Security textbooks traditionally describe these attacks with three characters. Alice and Bob are two parties who want to talk. Eve (the eavesdropper) is a passive attacker who only listens. Mallory is an active attacker who can modify messages. If the channel is unencrypted and unauthenticated, all four attacks below are possible.

Eavesdropping
: The passive attacker takes a copy of the traffic and reads it. Because it does not touch the messages, it is very hard to detect. It breaks confidentiality. The remedy is encryption: today this is mostly done by TLS.

Replay
: The attacker captures a valid message and later resends it unchanged. The message may be encrypted and signed: the attacker does not need to read or modify it. If a “pay 1000 TL” message is processed twice, the damage doubles. The remedy is to attach a value that can be used only once to every message: a nonce, a timestamp, or a sequence number.

Man-in-the-middle (MITM)

The attacker steps in between the two parties and appears to each as the other. Alice thinks she is talking to Bob; in fact she is talking to Mallory. Mallory reads, modifies, and forwards the messages. Encryption alone is not a remedy: the party with whom Alice sets up the encrypted channel may be Mallory himself. The remedy is to authenticate the other party, which means certificates.

Session hijacking

After a user authenticates once, the server gives the user a session token, usually a cookie. If an attacker steals this token, he enters the session without knowing the password. The remedy is to send the token only over an encrypted channel (`Secure`), to keep scripts from reading it (`HttpOnly`), and to keep its lifetime short.

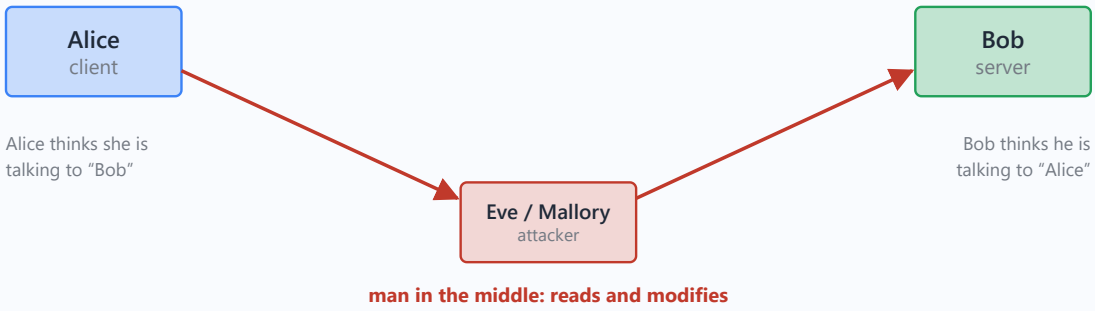


Figure 13.3: Man-in-the-middle: Mallory appears to each party as the other; he reads and modifies the messages.

Figure 13.4: Session hijacking: if the cookie issued after authentication is stolen, the attacker enters the session without a password.

Common mistake

Thinking that encryption prevents replay. The attacker can resend an encrypted message without decrypting it. If the receiver cannot tell that it has seen the same message before, it treats the second copy as valid too. Replay is prevented only by a single-use value attached to the message and by the receiver keeping track of those values.

13.1.5 Attackers and accidental data loss

The strength of a defense is tuned to who the attacker is. Protecting a machine in a student lab against a curious friend is not the same job as protecting a bank's server against a state-sponsored group. Attackers fall roughly into four groups.

The curious insider
: Has little technical knowledge and wants to “take a look” at someone else's e-mail or files. Usually exploits a permission mistake. The behavior is unethical and often illegal as well. Correct file permissions are mostly enough against this group.

The malicious employee
: Has a legitimate account and authority on the system and abuses it. Can leak data, commit sabotage, or leave a logic bomb that triggers if he is fired. Sits inside the walls built against outside attackers, which makes this the hardest group to detect.

The cybercriminal
: The motive is profit: ransomware, theft of identity and card data, renting out a network of compromised machines (a botnet). Today this is the most common threat. This group mostly uses known vulnerabilities that already have patches, and scans unpatched systems with automated tools.

State-sponsored groups
: The literature calls them advanced persistent threats (APT). Their goals are military or commercial espionage and sabotage of critical infrastructure. They can stay inside silently for years. They use vulnerabilities that nobody yet knows (zero-days) and supply-chain attacks. Stuxnet, which appeared in 2010, was such a piece of software, aimed at uranium-enrichment centrifuges.

Attackers are not the only cause of data loss. Accidents lose far more data than attacks do (Table 13.2). A disk fails, an administrator formats the wrong disk, a laptop is left on a train. In Tanenbaum's wry example, even mice gnawing on backup tapes make the list.

Cause	Example	Countermeasure
Natural events	fire, flood, earthquake	geographically separate backup
Hardware and software errors	disk failure, CPU error, software bug	RAID, checksums, versioning

Cause	Example	Countermeasure
Human error	wrong command, formatting the wrong disk, lost device	3-2-1 backup, encrypted disk

Table 13.2: Causes of accidental data loss and countermeasures.

The common backup rule is called **3-2-1**: keep three copies of the data on two different media, with one copy kept in another location. Today a condition is added: the remote copy should be *immutable*. Ransomware also encrypts or deletes the backups it can reach over the network. An immutable copy survives this attack. In this way the same rule works against both accidents and attacks.

13.1.6 Defense in depth

No defense is flawless. A firewall rule is written wrongly, an employee clicks a phishing e-mail, a new vulnerability is found in a library. So we do not rely on a single defense; defenses are arranged as nested layers. This principle is called **defense in depth**. It resembles the layout of medieval castles with a moat, a wall, an inner wall, and a keep: even if an attacker gets past one layer, he is stopped by the next (Figure 13.5).

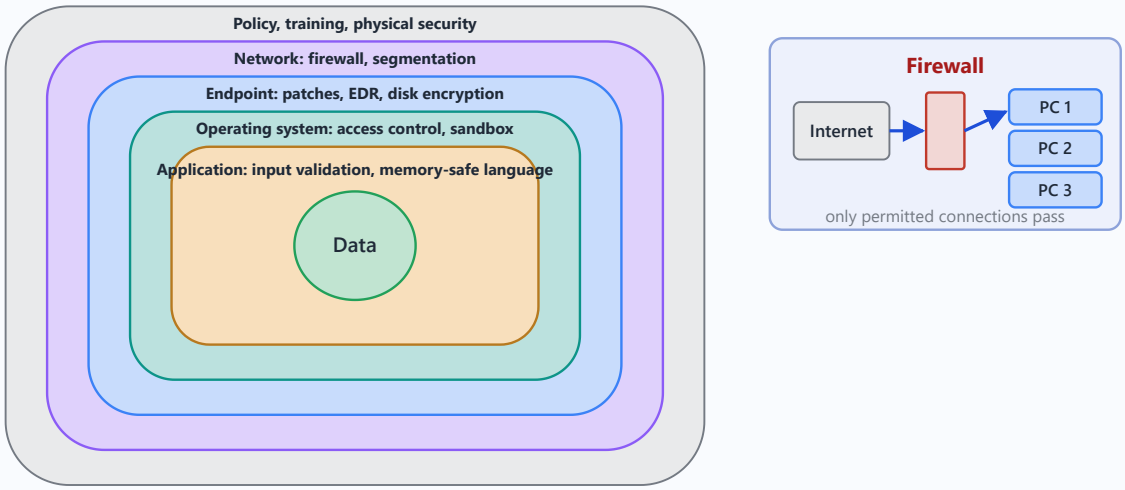


Figure 13.5: The layers of defense in depth (left) and a firewall in front of a network (right).

The layers run from the outside in as follows. At the outermost level are policies, user training, and physical security: the lock on the server-room door belongs to this layer. At the network layer, a **firewall** lets only permitted connections through; network segmentation also keeps the compromise of one segment from spreading to others. The endpoint layer has patches, endpoint detection and response (EDR) tools, and disk encryption. The operating-system layer provides access control and the sandbox. The application layer has input validation and memory-safe languages. At the innermost point stands the protected asset, the data.

Good practices are spread across these layers as well. Security updates are applied quickly, because most attacks use vulnerabilities whose patches came out long ago. Strong authentication and access control working with least privilege are the basic rule. Logs are kept and monitored; otherwise a breach may go unnoticed for months. Regular penetration tests look for the weak points of the defense in advance. Users are trained against phishing.

The **zero trust** approach, which has become common in recent years, is one step beyond defense in depth. In the traditional arrangement, the inside of the firewall was considered trusted: a machine inside could reach the other machines freely. Zero trust rejects this assumption. A request coming from inside the network goes through authentication and authorization just like one from outside. This makes lateral movement harder for an attacker who has slipped in.

13.1.7 The trusted computing base

Every security claim rests on the assumption that some components work correctly. File permissions mean something only if the kernel code that checks them is free of bugs. The sum of these components is called the trusted computing base (Figure 13.6).

Definition 13.3

The **trusted computing base** (TCB) is the set of all hardware, firmware, and software components responsible for enforcing the system's security policy. A bug in the TCB can break the security of the whole system; a bug outside the TCB cannot violate the policy.

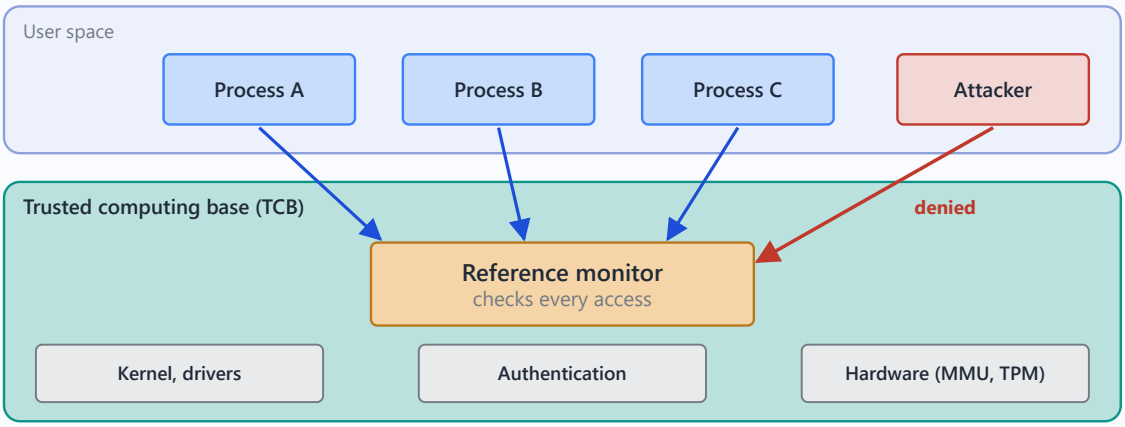


Figure 13.6: Every access by user processes passes through the reference monitor inside the TCB. The attacker's request passes through it as well and is denied.

The heart of the TCB is the **reference monitor**. This component receives every access request, consults the policy, and decides to allow or deny. Three conditions must hold for a reference monitor to be trustworthy. First, it must be invoked on every access; a check that can be bypassed is as bad as

no check at all. Second, it must be tamper-proof; user processes must not be able to modify its code or data. Third, it must be small enough for its correctness to be verified.

This third condition has an important consequence for operating-system design. The bigger the TCB, the higher the chance that it contains bugs. The Linux kernel has tens of millions of lines, and all of it, device drivers included, runs in kernel mode, that is, inside the TCB. This is why microkernels attract interest: by moving the file system and the drivers to user mode, they shrink the TCB. The seL4 microkernel goes further. Its roughly ten thousand lines of code have been formally proved to behave according to its specification.

Common mistake

Reading the TCB as "the components proved to be trustworthy." The definition says the opposite: the TCB is the set of components that one *has to* trust. If a printer driver runs in kernel mode, it is part of the TCB, however sloppily it was written. The goal of secure design is to keep this set as small as possible.

13.2 Cryptography

The operating system uses cryptography everywhere: when storing passwords, encrypting disks, verifying the signatures of update packages, protecting network connections, and checking the boot chain. This section introduces the basic tools needed to understand these uses. The goal is not to design ciphers. It is enough to know what each tool provides and what it does not.

13.2.1 Basics of encryption

The purpose of encryption is to turn a readable message into a form that someone without the key cannot understand. The readable message is called **plaintext** (P), and its transformed form **ciphertext** (C). The function that performs the transformation is encryption E, and the function that reverses it is decryption D. Both use a key K (Figure 13.7).

Definition 13.4

Encryption makes data unreadable without the key. In **symmetric** or **secret-key encryption**, encryption and decryption use the same key:

$$C = E(P, K), \quad P = D(C, K)$$

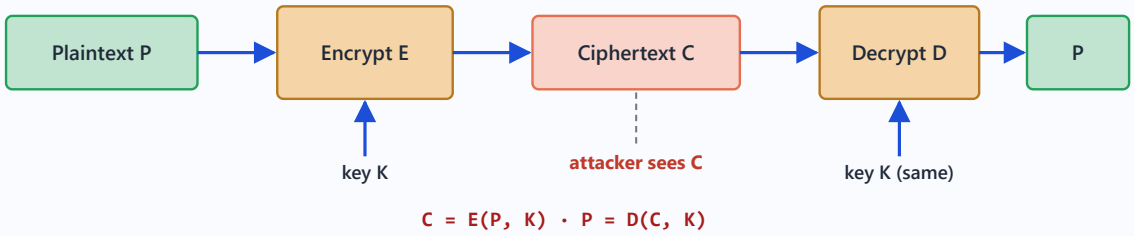


Figure 13.7: The symmetric encryption model. The attacker can see the ciphertext; without knowing the same K, he cannot obtain the plaintext.

The model assumes that the attacker can listen to the channel and see C. Does the attacker also know the encryption algorithm? By the principle Auguste Kerckhoffs set out in 1883, we must assume he does. **Kerckhoffs's principle** says that the security of a cryptosystem should rest only on the secrecy of the key. The algorithm may be public, and should be. A secret algorithm is sooner or later revealed by reverse engineering. A public algorithm, by contrast, is attacked for years by cryptographers all over the world; if it survives, it is considered trustworthy. This is why all of today's standard ciphers are open to everyone.

One consequence of Kerckhoffs's principle is the importance of the size of the key space. An attacker who knows the algorithm can try every key one by one. This method is called a **brute-force** attack. The old DES standard used a 56-bit key. In 1998, this space of 2^{56} keys could be searched in a few days with special-purpose hardware. A 128-bit key space, by contrast, cannot be searched even with the computing power of the whole world. But a large key space alone is not enough; the next example shows why.

13.2.2 Substitution cipher and frequency analysis

One of the oldest ciphers is the **monoalphabetic substitution cipher**. Each letter of the alphabet is replaced by a fixed other letter. The key is a permutation of the 26 letters. The key in the slides lists the letters starting from the top row of the keyboard: A → Q, B → W, C → E, and so on (Table 13.3).

plain	A	B	C	D	E	F	G	H	I	J	K	L	M
cipher	Q	W	E	R	T	Y	U	I	O	P	A	S	D
plain	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
cipher	F	G	H	J	K	L	Z	X	C	V	B	N	M

Table 13.3: The key of the substitution cipher.

Example 13.2

Let us encrypt the word MERHABA with this key.

1. The letter under M is D: M → D.

2. E → T, R → K, H → I.

3. A → Q, B → W, A → Q.

Result: MERHABA → DTKIQWQ. To decrypt, the table is read in the reverse direction: above the D in the ciphertext stands M. All three A's in the word have become Q.

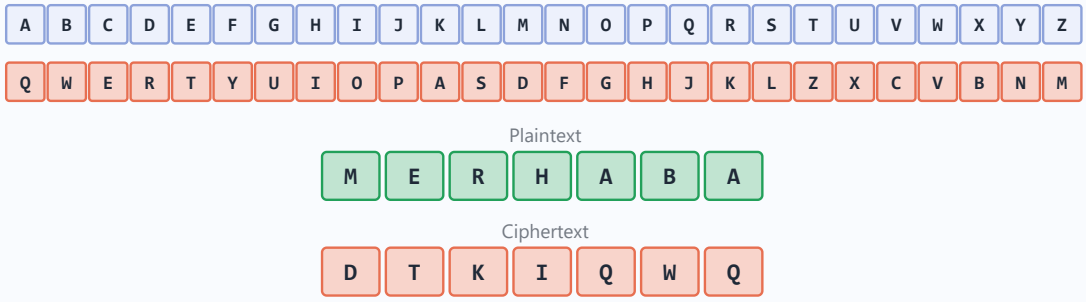


Figure 13.8: Encrypting MERHABA. The same plain letter always goes to the same cipher letter.

The key space of this cipher is huge: the number of permutations of 26 letters is $26! \approx 4 \times 10^{26}$, or about 2^{88}. A computer trying a billion keys per second would need billions of years to search this space. Even so, the cipher can be broken with a few lines of text, with pencil and paper. Why?

The fact that all three A's in the example became Q exposes the weakness. The cipher changes letters but does not change letter frequencies. In a Turkish text the most frequent letters are A, E, and İ (in English, E, T, and A). The most frequent letter in the ciphertext is very likely the image of A. Then the frequent two-letter and three-letter patterns ("ve", "bir", "ler") are matched. After a few guesses the key emerges step by step. This method is called **frequency analysis**. It was first described in the ninth century by the Arab mathematician al-Kindi.

Common mistake

Equating a large key space with security. The key space is only a lower bound against brute force. If the structure of the cipher leaks the statistics of the plaintext, the attacker does not try keys one by one but uses the structure directly. This is why the 2^{88} space is no help at all for the substitution cipher.

13.2.3 Modern symmetric ciphers

Modern symmetric ciphers use one of two constructions to hide frequency information. **Block ciphers** split the data into fixed-size blocks and scramble each block over many rounds. The standard block cipher is AES (Advanced Encryption Standard, 2001). It uses 128-bit blocks and keys of 128, 192, or 256 bits. Today's processors have dedicated instructions for AES (AES-NI), and encryption

reaches speeds of gigabytes per second. **Stream ciphers** instead generate a long random-looking byte sequence from the key and XOR it with the data. ChaCha20 is a common example of this kind; it is fast on phones that have no AES hardware.

It is not enough for a cipher to provide confidentiality alone. If an attacker flips bits of the ciphertext, the receiver gets a corrupted plaintext and may not notice. For this reason encryption today is always used together with an integrity check. This combined construction is called **authenticated encryption**. AES-256-GCM and ChaCha20-Poly1305 are two standard examples: they append a short authentication tag to the ciphertext. If even a single bit changes, the tag fails to match and the receiver rejects the message.

13.2.4 Public-key cryptography

Symmetric encryption has a fundamental problem: the two parties must already share the same key. How do we share this key with a site we are connecting to for the first time on the Internet? Sending the key over the network amounts to giving it to everyone who is listening. In 1976, Diffie and Hellman gave a new answer to this question: two separate keys.

Definition 13.5

In **public-key** or **asymmetric encryption**, every user has a key pair. The **public key** is distributed to everyone; only the user knows the **private key**. A message encrypted with the public key can be decrypted only with the matching private key.

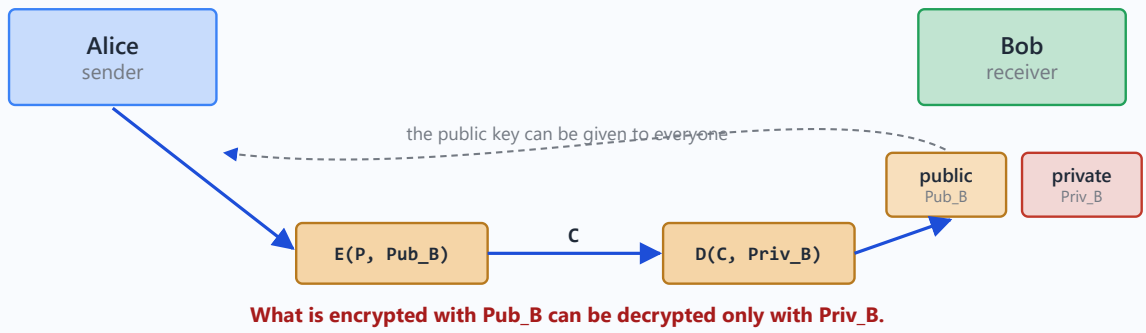


Figure 13.9: Alice encrypts with Bob's public key Pub_B; only Bob, who knows Priv_B, can decrypt the message.

When Alice wants to send Bob a secret message, she obtains Bob's public key and encrypts the message with it (Figure 13.9). Eve, who listens to the ciphertext, also knows Bob's public key, but she cannot decrypt the message with it. Decryption is possible only with the private key, and that key never leaves Bob's machine. So no previously shared secret is needed.

The mathematical basis of this arrangement is **one-way trapdoor functions**. Such a function is easy to compute in one direction and very hard to compute in the other. But if a secret piece of information (the trapdoor) is known, the reverse direction becomes easy too. In RSA (1977) the function is the product of large primes: multiplying two 1024-bit primes takes an instant, but there is no known fast way to factor the product. In elliptic-curve cryptography (ECC) the hard problem is the discrete logarithm on the curve.

Example 13.3

Let us follow RSA with toy-sized numbers. In real systems the numbers have hundreds of digits; the steps are the same.

1. **Key generation.** Bob picks two primes: $p = 3$, $q = 11$. Their product is $n = 33$. $\varphi = (p - 1)(q - 1) = 2 \times 10 = 20$. The public exponent is chosen as $e = 3$ (it has no common divisor with 20). The private exponent d is the number satisfying $e \cdot d \equiv 1 \pmod{20}$: $3 \times 7 = 21 \equiv 1$, so $d = 7$. The public key is $(n, e) = (33, 3)$ and the private key is $d = 7$.
2. **Encryption.** Alice sends the message $m = 4$: $c = m^e \bmod n = 4^3 \bmod 33 = 64 \bmod 33 = 31$.
3. **Decryption.** Bob computes $c^d \bmod n$: $31^7 \bmod 33 = 4$. The original message comes back.

Eve knows $n = 33$ and $e = 3$. To find d she needs φ, that is, she must know p and q. Factoring 33 is easy. Factoring a 2048-bit n, however, is beyond what today's computers can do. The values were verified with Python.

The price of asymmetric encryption is speed: it is a hundred to a thousand times slower than symmetric encryption (Table 13.4). So real systems use the two together. The asymmetric method does only a short job: it establishes a session key between the two parties. The actual data is then encrypted symmetrically under this session key. TLS, SSH, and disk encryption all use this hybrid arrangement.

	Symmetric	Asymmetric (public-key)
Key	Single, shared	Public + private pair
Speed	Very fast (AES-NI)	100–1000 times slower
Use	Bulk data	Key exchange, signatures
Example	AES, ChaCha20	RSA, ECDH/X25519, Ed25519, ML-KEM

Table 13.4: Comparison of symmetric and asymmetric encryption.

The last algorithm in the table is a response to a new threat. A sufficiently large quantum computer could solve both factoring and discrete logarithms quickly with Shor's algorithm. No such machine exists today. But traffic recorded today could be decrypted years from now, so the transition has already begun. In 2024 the US standards agency NIST published the lattice-based ML-KEM algorithm as a quantum-resistant key-establishment standard (FIPS 203).

13.2.5 Hash functions

The third basic tool of cryptography is not encryption; it uses no key and cannot be reversed.

Definition 13.6

A **cryptographic hash function** produces a fixed-size output, the **digest**, from an input of any length, and has the following properties:

- **One-wayness:** given a digest, finding an input that produces it is practically impossible.
- **Collision resistance:** finding two different inputs that give the same digest is practically impossible.
- **Avalanche effect:** changing a single bit of the input changes about half of the bits of the digest.

Today's standards are SHA-256 and SHA-3. SHA-256 produces a 256-bit digest, that is, 64 hexadecimal digits, from any input. To see the avalanche effect, let us compare the digests of two words.

Example 13.4

"MERHABA" and "MERHABa" differ only in the last letter. In ASCII, A (0x41) and a (0x61) differ in a single bit. The SHA-256 digests are:

```
MERHABA  cc44cc878588056ef8376b36fc7a2ba5124656c38597eb31291704e0af986112
MERHABa  57821b9db50b174af36bc659771a47e0eaa6aef8144b1a9e5b803c93e3280b18
```

There is no similarity between the two digests. Compared bit by bit, 124 of the 256 bits differ, which is very close to the expected half (128). The digests were computed with Python's `hashlib` library.

Hash functions appear in many places in an operating system. The digest of a downloaded file is compared with the published digest to see that the file was not corrupted. Passwords are stored not in the clear but as digests. Integrity-checking tools store digests of system files and find the ones that change. Git names every object by its digest.

Breaking a hash function means that finding collisions becomes practical. Collisions were produced in practice for MD5 in 2004 and for SHA-1 in 2017. Neither should be used for security purposes any more.

Common mistake

Treating a digest as compression, or as proof by itself. The input cannot be recovered from a digest, so it is not compression. An unsigned digest does not prove integrity either: an attacker who modifies the document can recompute and replace the digest that accompanies it. The digest must come over a trusted channel or be signed.

13.2.6 Digital signatures

Public-key cryptography can also be used in the reverse direction. An operation performed with the private key can be checked by anyone with the public key, but only the owner of the private key can perform it. The digital signature builds on this observation.

Definition 13.7

A **digital signature** is the value obtained by processing the digest of a document with the sender's private key. Anyone can verify the signature with the sender's public key. A valid signature proves two things: the document has not changed (integrity) and it came from the owner of the private key (authenticity, non-repudiation).

Signing and verification proceed in the following steps (Figure 13.10).

1. **Digest.** The sender, Alice, computes the digest of the document D: H(D). Signing the digest rather than the document itself is both fast and works on a fixed-size input.
2. **Signature.** Alice signs the digest with her private key Priv_A and obtains the signature S. Today's algorithms are RSA-PSS, ECDSA, and Ed25519.
3. **Sending.** The document, the signature, and the certificate carrying Alice's public key are sent together. If confidentiality is not wanted, the document is not encrypted.
4. **Verification.** The receiver, Bob, recomputes the digest of the document himself and verifies the signature with Pub_A. If verification says that the signature was made over exactly this digest with Priv_A, the document is intact and came from Alice.

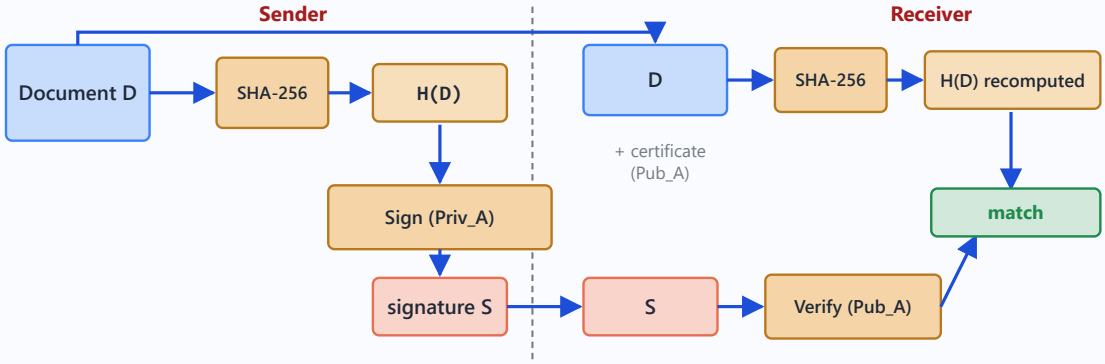


Figure 13.10: The flow of a digital signature: the sender signs the digest with her private key; the receiver recomputes the digest and verifies the signature with the public key.

What happens if a single byte of the document changes on the way? Bob computes the digest of the modified document D′. Because of the avalanche effect, this digest is completely different from H(D), and the signature does not match it (Figure 13.11). The attacker cannot produce a new valid signature either, because he does not have Priv_A.

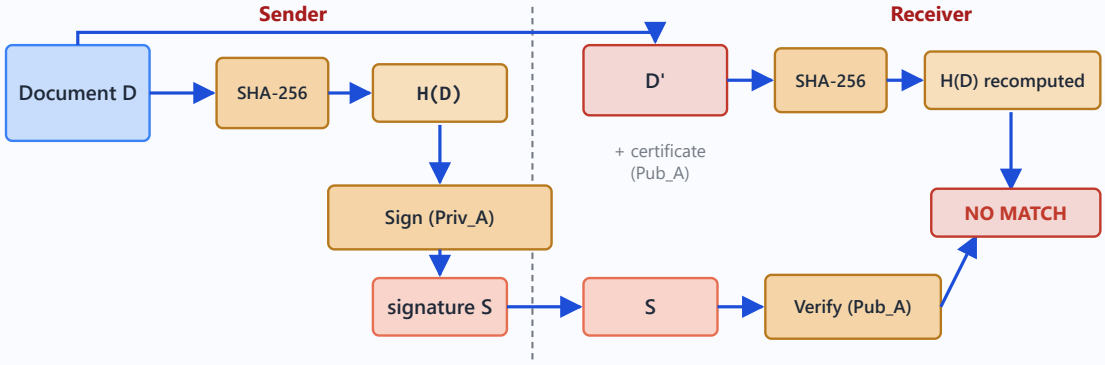


Figure 13.11: If the document becomes D′ in transit, the recomputed digest does not match the signature and verification fails.

Example 13.5

Let us follow signing with the toy RSA key (n = 33, e = 3, d = 7). Let the digest of the document be h = 4.

1. Bob signs with his private key: $s = h^d \bmod n = 4^7 \bmod 33 = 16$.
2. The receiver verifies with the public key: $s^e \bmod n = 16^3 \bmod 33 = 4$. The result equals the digest he computed himself; the signature is valid.
3. An attacker modifies the document and the digest becomes 5. $16^3 \bmod 33$ is still 4 and is not equal to 5; verification fails. To produce a signature matching the new digest one would need to know d = 7.

Signing is very easily confused with encryption. The two operations use the keys in opposite directions. In encryption, the sender uses *the receiver's public* key, and only the receiver can decrypt. In a signature, the sender uses *her own private* key, and anyone can verify. A message authentication code (MAC) produced with a shared secret key provides integrity but not non-repudiation: since the receiver also knows the key, the receiver could have produced the same code.

13.2.7 Certificates and public-key infrastructure

Public-key cryptography leaves one question open. How does Bob know that the public key he holds really belongs to Alice? If Mallory distributes his own public key as "Alice's key," he can read all messages sent to Alice. This is exactly the man-in-the-middle attack. The solution is a document, signed by a trusted third party, that binds the key to an identity.

Definition 13.8

A **certificate** (in the X.509 format) is a document that binds a public key to the identity of its owner, for example a domain name. It also contains a validity period and the signature of a **certificate authority** (CA) that vouches for the binding. The whole of the institutions and rules that produce, distribute, and revoke certificates is called a **public key infrastructure** (PKI).

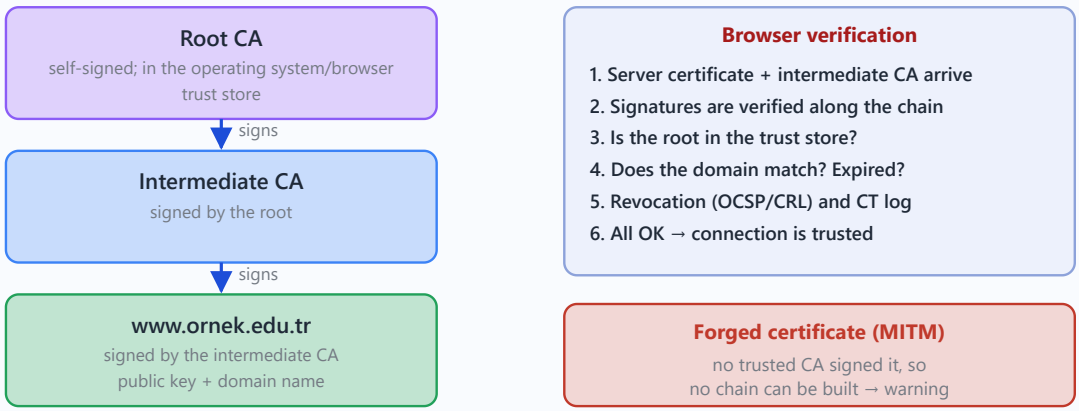


Figure 13.12: The certificate chain: the root CA signs the intermediate CA, and the intermediate CA signs the server certificate. On the right, the browser's verification steps.

Certificates form a chain (Figure 13.12). At the head of the chain is the **root CA**. The root certificate is self-signed; nobody signs it. Trust in it comes from its being preinstalled in the *trust store* of the operating system or the browser. Root CAs keep their keys offline and leave the daily work to intermediate CAs. The server certificate signed by an intermediate CA binds, for example, the domain name `www.ornek.edu.tr` to the server's public key.

When a browser connects to an HTTPS site, it performs the following checks. The server sends its own certificate and the intermediate CA certificate. The browser verifies the signatures along the chain and checks that the chain reaches a root in its trust store. It checks that the domain name in the certificate matches the name being connected to and that the validity period has not expired. It checks whether the certificate has been revoked (via OCSP or CRL) and whether it appears in the publicly visible Certificate Transparency (CT) logs. If everything passes, the connection is considered trusted. Mallory's forged certificate, on the other hand, does not carry the signature of a trusted CA, so no chain can be built and the browser shows a warning.

The world of certificates has changed rapidly over the last decade. Let's Encrypt made obtaining certificates free and automatic through the ACME protocol; its certificates last 90 days. A short lifetime limits the damage of a stolen key and forces renewal to be automated. A joint body of browser vendors and CAs has decided to cut the lifetime of certificates gradually to 47 days by 2029.

13.2.8 The TLS 1.3 handshake

All the earlier tools come together in a single protocol: TLS (Transport Layer Security). HTTPS, secure e-mail, and many other protocols run on top of TLS. The current version is TLS 1.3 (RFC 8446, 2018). The **handshake** at the start of a connection does three jobs together: it establishes a shared session key, it authenticates the server, and it confirms that the handshake has not been tampered with (Figure 13.13).

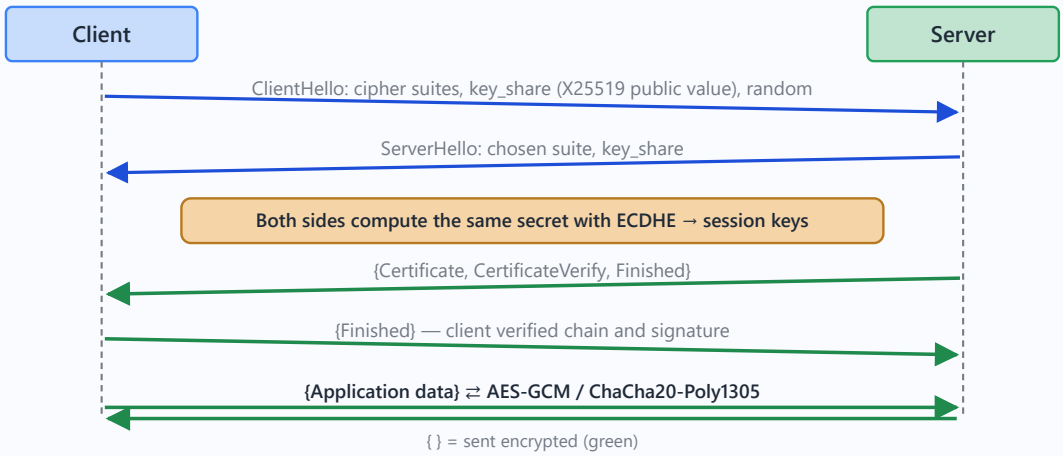


Figure 13.13: The TLS 1.3 handshake (simplified). Messages in curly braces are sent encrypted.

1. **ClientHello.** The client sends the cipher suites it supports, a random value, and an ephemeral Diffie-Hellman public value. Today this value is mostly from the X25519 curve.
2. **ServerHello.** The server chooses a cipher suite and sends its own Diffie-Hellman public value. Everything up to here goes in the clear.

3. **Shared secret.** Each side computes the same shared secret from the other side's public value and its own secret value. This secret never travels over the network. The session keys are derived from it, and every message from now on is encrypted.
4. **The server's identity.** The server sends its certificate. It then signs the digest of the handshake so far with its private key (CertificateVerify) and adds the Finished message. An attacker in the middle cannot produce this signature, because he does not have the server's private key.
5. **The client's confirmation.** The client verifies the certificate chain and the signature. With its own Finished message it confirms that the handshake has not been modified. The whole procedure ends in a single round trip (1-RTT).
6. **Application data.** Data now flows under symmetric authenticated encryption: AES-GCM or ChaCha20-Poly1305.

Ephemeral Diffie-Hellman values bring an important property: **forward secrecy**. The key of each session is derived from ephemeral values generated for that session, and these values are deleted when the session ends. The server's long-lived private key is used only for signing. Even if this key is stolen years later, the attacker's recorded old traffic cannot be decrypted.

13.2.9 Code trace: the Diffie-Hellman key exchange

The Diffie-Hellman exchange at the heart of TLS can be traced by hand with small numbers. The two parties first agree on two public numbers: a prime p and a base g. Each party chooses a secret exponent, raises the base to it, and sends the result in the clear. Then it raises the other party's public value to its own secret exponent. Thanks to the commutativity of exponentiation, both parties arrive at the same number:

> $A = g^a \bmod p, \quad B = g^b \bmod p, \quad s = B^a = A^b = g^{ab} \bmod p$

The code in the slides contains the `modpow` function, which computes large exponents efficiently, and the four computations of the exchange.

```c
long modpow(long g, long e, long p) {
    long r = 1;
    while (e > 0) {
        if (e & 1) r = r * g % p;
        g = g * g % p;
        e >>= 1;
    }
    return r;
}
int main(void) {
    /* public: p = 23, g = 5 */
    long A  = modpow(5, 6, 23);
    long B  = modpow(5, 15, 23);
    long sA = modpow(B, 6, 23);
    long sB = modpow(A, 15, 23);
}
```

`modpow` uses the "square and multiply" method. The bits of the exponent `e` are scanned from right to left. In each round, if the bit is 1, the result `r` is multiplied by the current base `g`. Then the base is squared to bring it to the weight of the next bit: g, g^2, g^4, g^8 ... Because the result is reduced mod p after every multiplication, the numbers never grow. If the exponent has n bits, the loop runs n rounds. For 5^{15}, the naive method needs 14 multiplications, while this method finishes in 4 rounds. For 2048-bit exponents the difference is the difference between possible and impossible.

Alice has chosen the secret a = 6. The computation A = modpow(5, 6, 23) proceeds as in Table 13.5. The binary representation of 6 is 110.

Round	e & 1	r	g (after)	e (after)	Explanation
1	0	1	5 · 5 mod 23 = 2	3	bit 0: r unchanged
2	1	1 · 2 = 2	2 · 2 = 4	1	bit 1: r = r · g
3	1	2 · 4 = 8	4 · 4 = 16	0	bit 2: r = r · g; the loop ends

Table 13.5: Trace of `modpow(5, 6, 23)`. Initially r = 1, g = 5, e = 6. The values were recomputed with Python.

The function returns 8: A = 8. This value is sent over the network in the clear; a = 6 never leaves Alice's machine. Bob runs the same function with his secret b = 15 (Table 13.6). The binary representation of 15 is 1111; since all four bits are 1, a multiplication happens in every round.

Round	e & 1	r	g (after)	e (after)
1	1	1 · 5 = 5	25 mod 23 = 2	7
2	1	5 · 2 = 10	4	3
3	1	10 · 4 = 40 mod 23 = 17	16	1
4	1	17 · 16 = 272 mod 23 = 19	256 mod 23 = 3	0

Table 13.6: Trace of `modpow(5, 15, 23)`. Initially r = 1, g = 5, e = 15.

B = 19 travels over the network to Alice. Now each party raises the other's public value to its own secret exponent. Alice computes sA = modpow(19, 6, 23): the rounds pass through r = 1, 16, 2 and the result is 2. Bob computes sB = modpow(8, 15, 23): r = 8, 6, 12, 2, and the result is again 2. Both parties have computed $5^{6 \cdot 15} \bmod 23 = 5^{90} \bmod 23$.

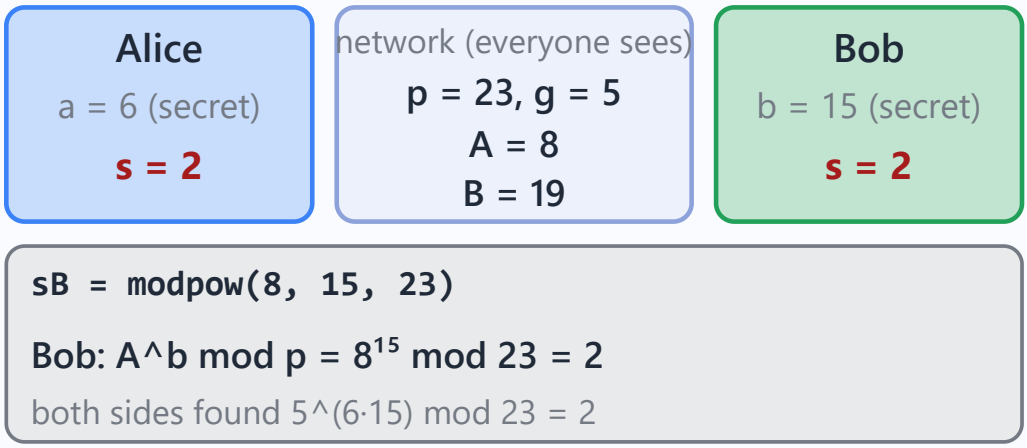


Figure 13.14: The end of the exchange: Alice and Bob reach the shared secret s = 2; only p, g, A, and B crossed the network.

Eve, who listens to the network, sees p = 23, g = 5, A = 8, and B = 19. To find the shared secret she needs a or b. Finding a means solving the equation $5^a \bmod 23 = 8$. This is called the **discrete logarithm problem**. In this small example Eve tries 22 possibilities and finds a = 6 at once. In practice p is 2048 bits long, or the X25519 curve is used. No known algorithm can solve a problem of this size in reasonable time.

Common mistake

Believing that Diffie-Hellman alone prevents the man-in-the-middle attack. The exchange is secure only against an Eve who merely listens to the network. If Mallory steps in and performs one exchange with Alice and a separate one with Bob, she establishes a separate shared secret with each party and reads all the traffic. This is why in TLS the server signs the handshake with the key in its certificate. Unauthenticated Diffie-Hellman is not secure.

13.3 Protection

Cryptography protects data outside the system, on the network or on disk. Inside the system a different question is asked: which objects can a running process access, and with which rights? This section presents the model for this question (the protection matrix), the two ways of implementing the model (access control lists and capabilities), and ways of reducing rights to a minimum.

13.3.1 Protection domains

A computer system contains **objects** that need protection: files, memory segments, printers, semaphores, processes. Each object has operations of its own: a file is read, written, and executed; a printer is only written to. At a given moment, a process can access some of these objects with certain rights. The set of these (object, right) pairs is called a domain (Figure 13.15).

> **Definition 13.9**
>
> A **protection domain** is a set of (object, rights) pairs. Every process runs in some domain at every moment and can perform only the operations that domain permits.

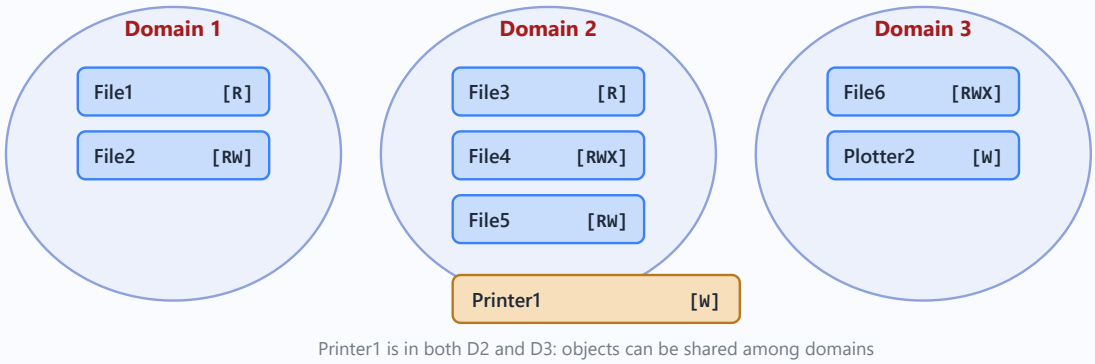


Figure 13.15: Three protection domains. Printer1 appears in both the second and the third domain: objects can be shared among domains.

In Figure 13.15, the first domain can read File1 and read and write File2. The second domain reads File3, uses all rights on File4, reads and writes File5, and writes to Printer1. The third domain has File6 (RWX), Plotter2 (W), and Printer1 (W) again. Domains need not be disjoint: the same object can appear in several domains, even with different rights.

In UNIX and Linux, a domain is determined by the set of a user ID (UID) and group IDs (GIDs). Two processes running with the same UID and GIDs access the same objects with the same rights. In addition, kernel mode and user mode are two separate domains. Code running in kernel mode can access all memory and all devices.

Processes can change domains while running. A system call moves the process from user mode to kernel mode; when the call ends, the process returns to its old domain. When a process executes a program with the `setuid` bit, it moves into the domain of the program's owner.

13.3.2 The protection matrix

All the rights of all domains can be shown in a single table. This table, proposed by Lampson in 1971, is called the protection matrix or access matrix.

> **Definition 13.10**
>
> In the **protection matrix** (access matrix), each row stands for a domain and each column for an object. The cell at the intersection of row i and column j holds the rights that a process in domain i has over object j.

	File1	File2	File3	File4	File5	Printer1	File6	Plotter2	D1	D2	D3
D1	R	RW								Enter	
D2			R	RWX	RW	W					
D3						W	RWX	W			

Table 13.7: A protection matrix in which domains also appear as objects. The “Enter” right permits moving into another domain.

Table 13.7 adds an interesting feature to the matrix: domains are objects too. The **Enter** right in row D1, column D2 says that a process running in D1 may switch to D2. In this way, switching domains also comes under the control of the matrix. The same approach can model the rights to modify the matrix itself: copying a right to someone else, or editing rights as the owner of an object.

Every state of the matrix can be compared with the security policy. If the policy says that Eve must never be able to read a file, the cell for that file in Eve's row must never show R. If R appears in that cell after a sequence of legitimate commands, the system has gone from an **authorized** state to an **unauthorized** one. A natural question arises: given an initial state and the permitted commands, can we compute in advance whether an unauthorized state can be reached? In 1976 Harrison, Ruzzo, and Ullman showed that this question is **undecidable** in the general case. There is no algorithm that answers “can a right leak?” for every matrix. This result explains why models such as Bell-LaPadula and Biba, which restrict the rules from the start, were developed.

The practical problem with the matrix is its size. In a system with thousands of users and millions of files, the matrix has billions of cells, and the vast majority of them are empty. The matrix is very **sparse**. Even in the small example of Table 13.7, counting only the eight object columns, 15 of the 24 cells are empty. So the whole matrix is never stored. Either the columns or the rows are stored, and the empty cells are skipped.

13.3.3 Two forms of the same matrix: ACLs and capabilities

Storing the matrix column by column means putting next to each object a list answering “who can access this object, and with which rights?” Storing it row by row means giving each domain a list answering “which objects can this domain access, and with which rights?” (Figure 13.16 and Figure

13.17).

Definition 13.11

An **access control list** (ACL) is a column of the matrix. It lists which domains can access an object and with which rights, and it is stored with the object. A **capability list** (C-list) is a row of the matrix: it lists the objects a domain can access and its rights over them. Each entry in the list is called a **capability**.

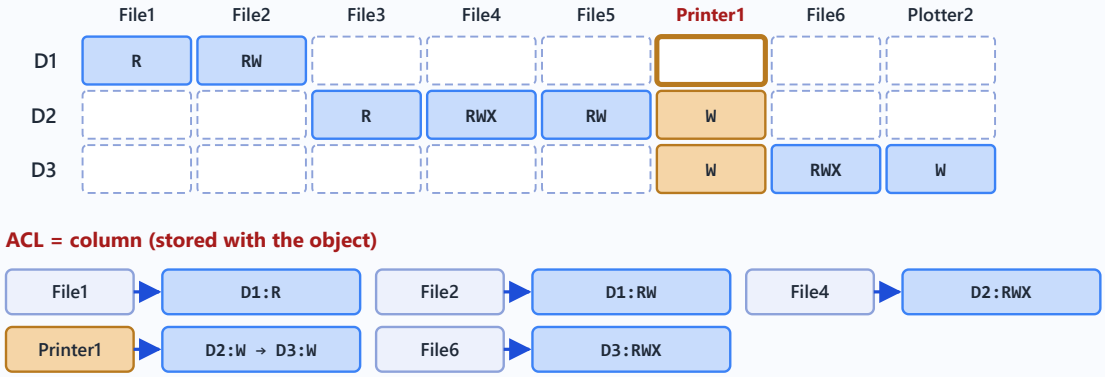


Figure 13.16: The non-empty columns of the matrix become ACLs. For example, the ACL of Printer1 is D2:W and D3:W.

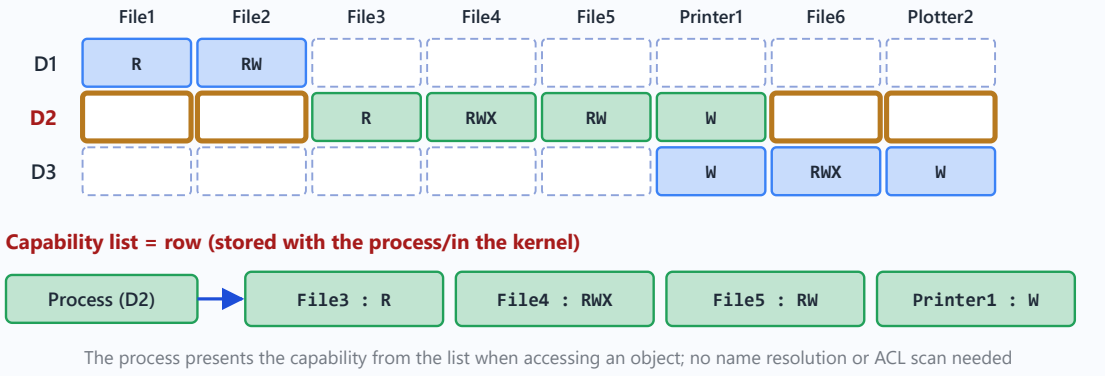


Figure 13.17: A row of the same matrix: the capability list of D2 is File3:R, File4:RWX, File5:RW, and Printer1:W.

In the ACL scheme, when a process wants to access an object, the kernel opens the object's ACL and looks for the process's identity in this list. In the capability scheme, the process presents the capability it holds when it accesses. It is enough for the kernel to check that the capability is valid; no name resolution or list scan is needed.

The two representations carry the same information, but make different questions easy to answer (Table 13.8).

Question	ACL (column)	Capability (row)
Where is it stored?	Next to the object	With the domain (in the process or in the kernel)
“Who can access this object?”	Easy: look at the list	Hard: scan all processes
“What can this process access?”	Hard: scan all objects	Easy: look at the list
Revoking a right	Easy: delete one entry	Hard: all copies must be found
Delegating a right	The object's list must change	Easy: copy the capability
Cost at access time	List scan	Validity check of the capability

Table 13.8: Comparison of ACLs and capability lists.

Example 13.6

In the matrix of Table 13.7, we want to revoke D3's right to Printer1 immediately. In addition, D2 wants to delegate its read right on File5 to D1. In which representation is each job easy?

1. **Revocation.** In the ACL scheme, Printer1's list is {D2:W, D3:W}; the D3 entry is deleted and the job is done. In the capability scheme, the capability in D3's list must be deleted. But D3 may have copied this capability to other processes earlier. To find all the copies, every capability list in the system must be scanned.
2. **Delegation.** In the capability scheme, D2 hands D1 a copy of its File5:R capability; File5 itself is not touched. In the ACL scheme, a new D1:R entry must be added to File5's list; only the owner of the object can do this.

Real systems often combine the two representations. When opening a file, UNIX checks the permission bits, which are a kind of ACL. For the opened file it gives the process a **file descriptor**. This descriptor is in fact a capability: on the subsequent `read` and `write` calls the permission is not checked again. Even if the file's permissions are changed in the meantime, the open descriptor keeps working.

13.3.4 UNIX permissions and POSIX ACLs

Classic UNIX uses a very compressed form of the ACL idea. Every file has an owner and a group. The file's i-node holds three bits for each of three classes: owner (u), group (g), and others (o). Each triple has read (r), write (w), and execute (x) bits. The nine bits in total are written as three digits in octal: r = 4, w = 2, x = 1.

```
$ ls -l report.txt
-rw-r----- 1 ali accounting 4096 report.txt
 rw-  owner   r--  group  ---  other

$ setfacl -m u:ayse:rw report.txt
$ getfacl report.txt
user::rw-
user:ayse:rw-
group::r--
mask::rw-
other::---
```

After the first character, the string `-rw-r-----` in the output splits into three triples. The owner ali can read and write (rw- = 4 + 2 = 6). Members of the group accounting can only read (r-- = 4). Others have no rights (--- = 0). The file's octal mode is 640, and it can be set with the command `chmod 640 report.txt`.

The permissions of newly created files are determined by the **umask** value. A program usually asks for 666 for a file and 777 for a directory. The kernel clears from this value the bits that are 1 in the umask. With the common umask 022, files get 644 (rw-r--r--) and directories 755 (rwxr-xr-x). With the stricter umask 077, files are open only to the owner: 600.

The nine-bit scheme is fast and fits in the i-node, but it is coarse-grained. If Ali wants to make his file writable not by the whole accounting group but only by Ayşe, nine bits are not enough. POSIX ACLs fill this gap. The command `setfacl -m u:ayse:rw report.txt` adds a named-user entry to the list. In the `getfacl` output we see this entry and a new line: `mask`. The mask limits the maximum rights that the named-user and group entries can reach. Even if Ayşe's entry is rw-, if the mask is set to r--, Ayşe can effectively only read. Files that carry an ACL are recognized in `ls -l` output by a `+` sign at the end of the permission string.

Example 13.7

A script file has mode 754. What can the owner, group members, and others do? What permissions do a new file and a new directory get when the umask is 027?

1. 7 = 4 + 2 + 1: the owner reads, writes, and executes (rwx).
2. 5 = 4 + 1: the group reads and executes (r-x).
3. 4: others only read (r--). The string is `rwxr-xr--`.
4. umask 027 clears write (2) from the group and everything (7) from others. File: 666 → 640 (rw-r-----). Directory: 777 → 750 (rwxr-x---).

The NTFS file system of Windows uses a richer ACL model. Each object's discretionary access control list (DACL) consists of ordered entries. Each entry carries a security identifier (SID), a rights mask, and a type, allow or deny. Explicit deny entries come before allows. The ACL of a folder passes to the files inside it through inheritance. On the command line, the `icacls` tool displays and edits these lists. The semantics of POSIX, NFSv4, and NTFS ACLs do not match exactly. For this reason, rights sometimes change in unexpected ways when files are moved between systems.

The advantage of the ACL scheme is that it is easy to understand. Who can access an object is visible at a glance, and a right is easily revoked per object. Its drawbacks are that administration gets harder in large systems and that the list is scanned on every access. The scanning cost is eased by caching: in UNIX, the permission check is made only at `open` time.

Example 13.8

Users are divided into four groups and an administrator role, and there are four objects (A, B, C, D) (Table 13.9). What is the ACL of object C? Which objects can Group 4 access?

Group	A	B	C	D
Group 1	R	RWX	—	—
Group 2	W	—	—	—
Group 3	—	—	X	R
Group 4	—	—	—	—
Admin	RWX	RWX	RWX	RWX

Table 13.9: Protection matrix for groups and objects.

The ACL of C consists of the non-empty cells of its column: {Group 3: X, Admin: RWX}. Group 4's row is entirely empty, so it appears in no ACL. By the safe-default rule, this group cannot access any object.

13.3.5 Code trace: an access control list

We can trace how a kernel scans an ACL with a small C function. The ACL of file A is column A of the protection matrix: Group 1 reads, Group 2 writes, and the administrator can do everything. Rights are encoded as bits: R = 4, W = 2, X = 1.

```
/* ACL of file A: R=4 W=2 X=1 */
struct entry { int group, rights; } acl[] =
    {{G1, 4}, {G2, 2}, {ADM, 7}};

int access(int group, int req) {
    for (int i = 0; i < 3; i++)
        if (acl[i].group == group) {
            int h = acl[i].rights;
            return (h & req) == req;
        }
    return 0;   /* not listed: denied */
}
```

The array has only three entries; the cells of Group 3 and Group 4 on A are empty, so they are not stored. The `access` function scans the list from the beginning. At the first entry that names the requesting group it decides and returns. The decision expression is `(h & req) == req`. If all the requested rights are present in the entry, the AND yields exactly the request. If even one is missing, the result differs and access is denied. If the group is not in the list, the loop completes and the function returns 0. Table 13.10 traces four calls.

Call	i examined	Matching entry	h & req	Result
`access(G2, W)`	0, 1	i = 1, h = 2	2 & 2 = 2	allow
`access(G1, W)`	0	i = 0, h = 4	4 & 2 = 0 ≠ 2	deny
`access(G4, R)`	0, 1, 2	none	—	deny (0)
`access(ADM, R|X)`	0, 1, 2	i = 2, h = 7	7 & 5 = 5	allow

Table 13.10: Four calls of the `access` function. The values were recomputed with Python.

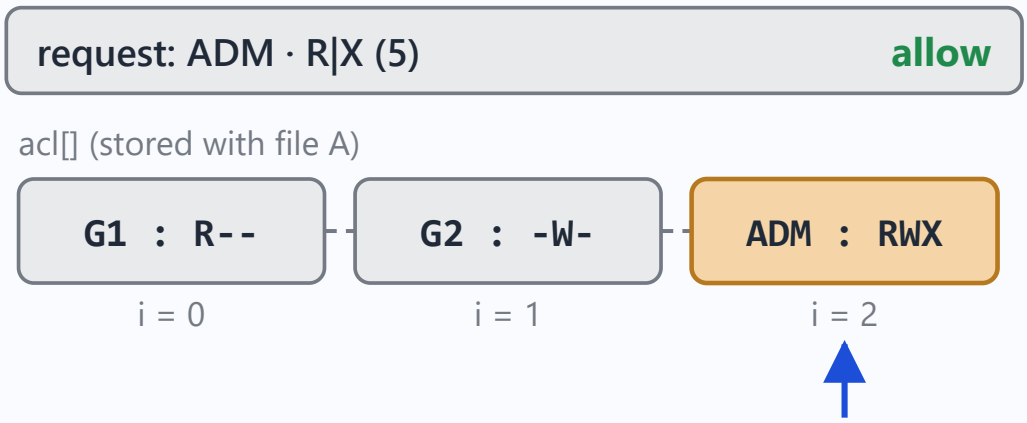


Figure 13.18: The last call: the administrator asks for read and execute together (5); a match is found at the third entry, and since 7 & 5 = 5, access is allowed.

The trace shows three lessons clearly. First, the decision is made at the first matching entry: Group 1's write request is denied and the list is not scanned any further. This is also why NTFS puts deny entries first. Second, for a request from a group that is not in the list, the whole list is scanned. This is the cost of every access with long ACLs: in a list of n entries, O(n) comparisons in the worst case. Third, the safe-default principle is built into the code. Everything not explicitly allowed is forbidden; the last line of the function says so.

Common mistake

Writing the decision expression as `(h & req) != 0`. This expression allows access if *any one* of the requested rights is present. If Group 1 asks for read and write together (6), the result is $4 \,\&\, 6 = 4 \neq 0$, and access is allowed although the group has no write right. The correct expression requires *all* of the requested bits to be present in the entry.

13.3.6 Capability lists

In the capability scheme, a capability is a ticket that gives its holder the right to access an object. The most important property of the ticket is that it cannot be forged. If a process could mint a "capability to write File7" on its own, the scheme would protect nothing. There are three ways to protect capabilities from forgery.

Keeping them in the kernel
: The capability list lives in kernel memory. The process knows only the index of an entry in the list and passes this index to the system call: `read(cap 2, …)`. UNIX file descriptors work exactly this way (Figure 13.19).

Tagged hardware
: An extra bit is attached to every word in memory to show whether it is a capability. This bit can be changed only by special instructions. The CHERI architecture from the University of Cambridge brings this approach to current processors.

Cryptographic protection
: The capability stays in the user's hands but carries a check field. This field is obtained by passing the object and the rights, together with a secret value known only to the server, through a one-way function (for example HMAC). If the user changes the rights, the check field no longer matches. The Amoeba distributed operating system used this method.

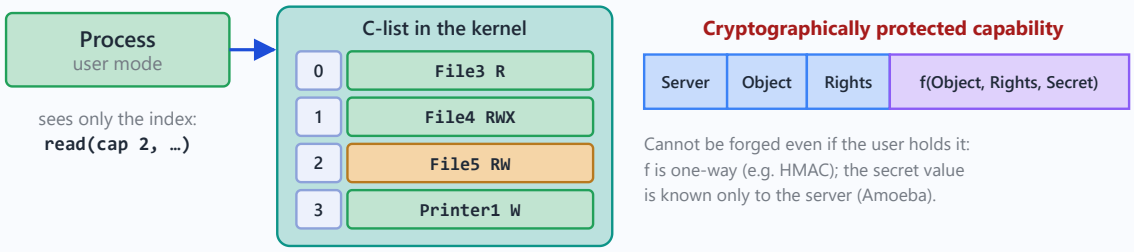


Figure 13.19: On the left, the C-list kept in the kernel: the process sees only an index. On the right, the cryptographically protected capability format.

Four basic operations are defined on capabilities. *Copying a capability* creates a new capability for the same object; this is delegation. *Copying an object* creates a new object and a new capability for it. *Removing a capability* deletes an entry from the list; the object stays in place. *Destroying an object* permanently deletes the object and its capability.

Pure capability systems live today in research and in specialized niches: the formally verified seL4 microkernel and Google's Fuchsia work through capabilities from top to bottom. FreeBSD's Capsicum framework, on the other hand, adds a capability mode to an ordinary UNIX. A process that enters this mode can no longer open new files by name; it can only use the descriptors it already holds.

13.3.7 Least privilege and setuid

The famous 1975 paper by Saltzer and Schroeder lists the principles of secure system design. The best known of them is the following.

> **Definition 13.12**
>
> The **principle of least privilege**: every component should run with the smallest set of rights that suffices for its job, and should hold those rights only for as long as they are needed. If the component is compromised, the harm it can do is limited to this set.

UNIX has a classic situation that conflicts with this principle. The digests of passwords are in the file `/etc/shadow`. The file is owned by root and its mode is `-rw-r-----`: ordinary users cannot even read it. So how does Ali change his own password? The solution is the **setuid** bit. The program `/usr/bin/passwd` is owned by root and shows an s instead of x in its permissions: `-rwsr-xr-x`. When this bit is set, the program runs with the rights of the file's owner, not of the person who runs it.

To understand this behavior, one must know the three user IDs of a process. The **real ID** (ruid) is the user who started the process. The **effective ID** (euid) is the ID used in permission checks. The **saved ID** (suid) stores an ID the process may return to later. Normally all three are equal. When a program

with the setuid bit is started by `execve`, the effective and saved IDs are changed to the file's owner; the real ID does not change.

```
/* /usr/bin/passwd  -rwsr-xr-x root */
int main(void) {
    uid_t me = getuid();
    int fd = open("/etc/shadow", O_RDWR);
    update_own_entry(fd, me);
    close(fd);
    setresuid(me, me, me);
    /* ... rest of the work ... */
    fd = open("/etc/shadow", O_RDWR);
    if (fd < 0) perror("open");
    else close(fd);
    return 0;
}
```

The code is a sketch of a `passwd` that uses its privilege only as long as necessary. `getuid()` returns the real ID. From it the program learns on whose behalf it runs and changes only that user's line. In the `open` call the kernel checks access against the effective ID. Since euid is 0, the file opens. As soon as the job is done, `setresuid(me, me, me)` sets all three IDs to 1000. Because the saved ID changes too, there is no way back. The second `open` at the end now fails with EACCES and returns −1; the program checks the return value, and `perror` prints the error as “open: Permission denied” (Table 13.11).

Step	ruid	euid	suid	Result
shell (ali)	1000	1000	1000	unprivileged
`execve` (setuid root)	1000	0	0	effective ID becomes the file owner
`getuid()`	1000	0	0	returns 1000
`open("/etc/shadow")`	1000	0	0	allowed (euid 0)
`setresuid(1000, 1000, 1000)`	1000	1000	1000	privilege dropped permanently
`open("/etc/shadow")`	1000	1000	1000	EACCES, fd = −1

Table 13.11: Tracing the IDs when Ali (uid 1000) runs `passwd`. The values were verified with a Python simulation.

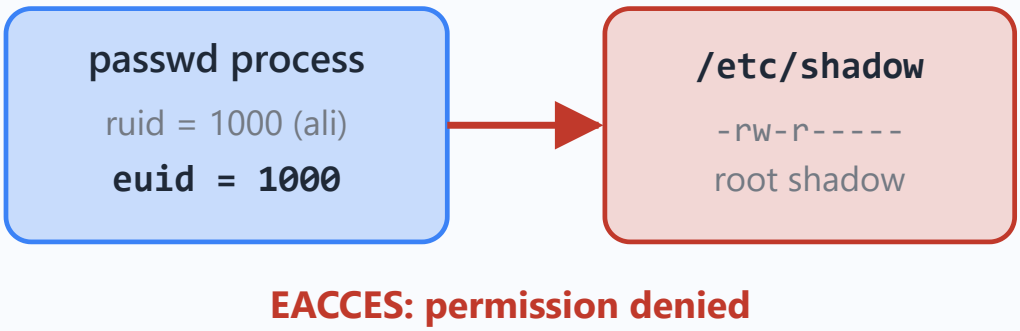


Figure 13.20: After the privilege is dropped, euid is 1000 and access to `/etc/shadow` is denied.

The value of dropping privilege this early shows when a bug appears in the rest of the program. Suppose the later part of the program has a buffer overflow and an attacker exploits it to run his own code. If the privilege had not been dropped, the attacker's code would run as root. Because it was dropped, it runs with Ali's rights and can do no more than Ali could already do. This is why setuid-root programs are the classic target of privilege-escalation attacks. Each must be examined carefully, and their number should be kept to a minimum.

Common mistake

Thinking that `seteuid(getuid())` drops the privilege. This call changes only the effective ID; the saved ID stays 0. Because an unprivileged process can set its effective ID to its real or saved ID, the attacker's code becomes root again with `seteuid(0)`. To drop permanently, the saved ID must be changed too: `setresuid(me, me, me)`.

13.3.8 Linux capabilities and design principles

In traditional UNIX, root can either do everything or nothing. Ports below 1024 are privileged. So a web server that wants to listen on port 80 has to be root. The web server thereby also gains, needlessly, the rights to bypass file permissions, change the clock, and load kernel modules. Linux splits root's single block of authority into about 41 separate pieces and calls them **capabilities** (Figure 13.21). The word “capability” here is not the same as in the capability-list scheme; it means “a piece of privilege.”

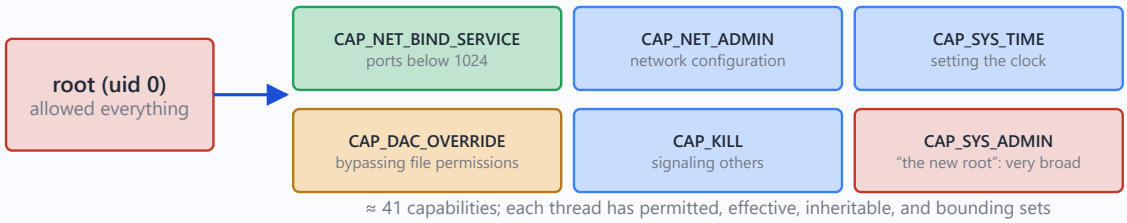


Figure 13.21: The division of root's authority into Linux capabilities. CAP_SYS_ADMIN is so broad that it is called "the new root."

For example, CAP_NET_BIND_SERVICE is the right to open ports below 1024. CAP_DAC_OVERRIDE bypasses file permissions, CAP_KILL sends signals to processes of other users, and CAP_SYS_TIME sets the system clock. Every thread has four capability sets: permitted, effective, inheritable, and bounding. A capability can also be assigned to a file. The commands below give the web server only the right to open low-numbered ports.

```
$ sudo setcap cap_net_bind_service=+ep /usr/local/bin/websrv
$ getcap /usr/local/bin/websrv
/usr/local/bin/websrv cap_net_bind_service=ep     # may bind port 80, no other root rights
```

The suffix `=ep` says that the capability is added to the permitted (p) and effective (e) sets. The program can now listen on port 80 without being root. Even if it is compromised, it cannot bypass file permissions or kill other processes.

The list by Saltzer and Schroeder does not end with least privilege. The other principles find counterparts in today's systems as well.

Fail-safe defaults
: Everything not explicitly allowed is forbidden. The `return 0` on the last line of the ACL scan is this principle.

Complete mediation
: Every access is checked; a permission once granted is not treated as valid forever in a cache. This is the first condition of the reference monitor.

Open design
: Security does not depend on the design being kept secret. This is the general form of Kerckhoffs's principle.

Separation of privilege
: A critical operation is made to depend on more than one condition or person. A safe with two keys and approval by two people are examples of this principle.

Psychological acceptability
: A security mechanism must not exasperate the user. Otherwise users find ways to get around it: they paste the password on the screen, they set permissions to 777.

Today's software applies these principles through **privilege separation**. The OpenSSH server is a good example. A small root process does only the jobs that need privilege, such as authentication. Risky jobs, such as parsing data that arrives from the network, run in a child process with no privileges at all. systemd likewise limits which capabilities a service may use with `CapabilityBoundingSet=` in service files. `NoNewPrivileges=` prevents a service from gaining new privileges through setuid programs.

13.4 Formal security models

The protection matrix can express any policy, but whether a right can leak cannot be computed in the general case. Formal models solve this problem by restricting the rules from the start. The rules are so simple that, as long as they are obeyed, a particular harm can be proved impossible. This section covers two classic models and a leakage path that even they cannot close: covert channels.

In both models it is the system, not the user, that does the enforcing. With ordinary UNIX permissions the owner of a file can change the permissions as he likes; this is called **discretionary access control** (DAC). In formal models, the policy is set by the administrator and no process can loosen it; this is called **mandatory access control** (MAC).

13.4.1 The Bell-LaPadula model

In military organizations, documents are divided into classification levels. In the arrangement used in the slides, these levels from highest to lowest are Top Secret (4), Secret (3), Confidential (2), and Unclassified (1). Every person also has a clearance level. The model defined by Bell and LaPadula in 1973 is the computer counterpart of this arrangement. It has a single goal: to prevent information from leaking from a high level to a low level.

Definition 13.13

In the **Bell-LaPadula model** (BLP), every process and every object carries a confidentiality level. The model imposes two rules:

- **Simple security property:** a process at level k may read only objects whose level is less than or equal to k (*no read up*).
- *** property** (star property): a process at level k may write only to objects whose level is greater than or equal to k (*no write down*).

The first rule agrees with intuition: a person cleared for Secret cannot read a Top Secret document. The second rule looks odd at first sight. Why should a process at the Secret level not be able to write to an Unclassified file? The answer is that a process can carry the information it reads to where it writes. If a process that reads a Secret document copies its contents into an Unclassified file, the

information has leaked downward. Writing up, on the other hand, does not break confidentiality: if Secret information moves into a Top Secret file, it is protected more tightly. This kind of write is usually a “blind” append; the process cannot read the file it writes.

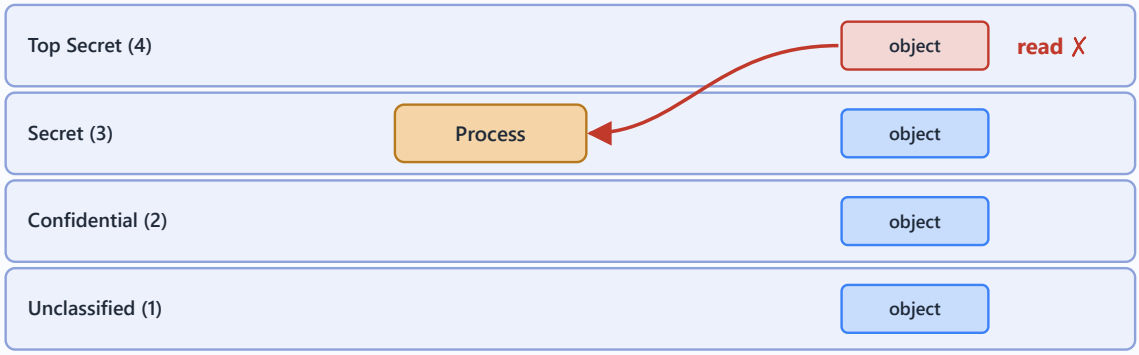


Figure 13.22: No read up: a process at the Secret level cannot read a Top Secret object.

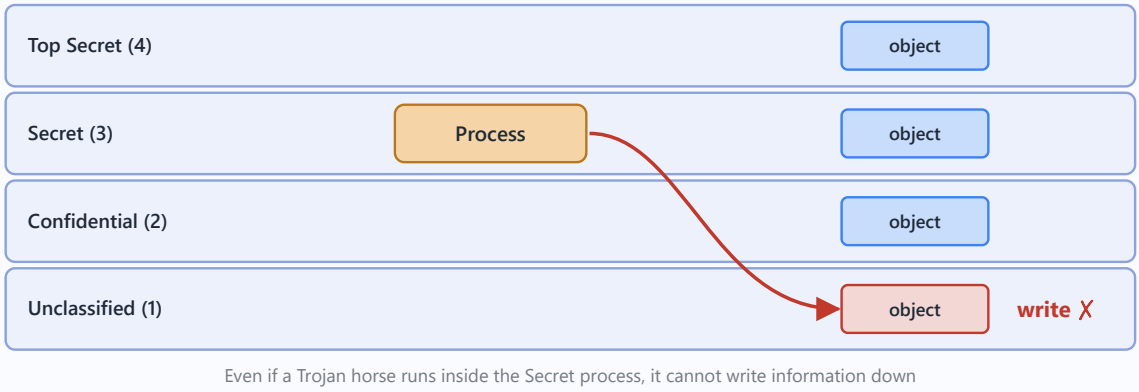


Figure 13.23: No write down: even a Trojan horse inside the process cannot write information to the Unclassified level.

Example 13.9

A process running at the Secret (3) level makes read and write requests to objects at the four levels. Which ones does BLP allow?

1. **Reading.** The object level must be ≤ 3. Top Secret (4): denied. Secret, Confidential, Unclassified: allowed.
2. **Writing.** The object level must be ≥ 3. Top Secret and Secret: allowed. Confidential and Unclassified: denied.

The process both reads and writes at its own level. It only reads below and only writes above.

The real gain of BLP is that its rules do not depend on the good will of the user. Suppose a user cleared for Secret runs a program with a **Trojan horse** hidden inside it. The program runs with the user's rights and can read Secret documents. Under DAC the program could write what it read into a publicly readable file. Under BLP this write is a write down, so the system denies it. The rule works regardless of what the program is trying to do.

13.4.2 The Biba model

BLP protects only confidentiality; it says nothing about integrity. An Unclassified process may write to a Top Secret file and corrupt it. In 1977 Biba proposed a model that protects integrity. Biba is the mirror image of BLP: this time the levels show not confidentiality but *trustworthiness*. In the example in the slides the levels are High (system), Medium (user), and Low (Internet).

Definition 13.14

In the **Biba model**, every process and object carries an integrity level. The model imposes two rules:

- **Simple integrity property:** a process at level k may read only objects whose level is greater than or equal to k (*no read down*).
- *** integrity property:** a process at level k may write only to objects whose level is less than or equal to k (*no write up*).

The meaning of the no-write-up rule is clear: user processes cannot modify system files. The no-read-down rule is subtler. A process that reads untrusted data from the Internet and acts on it becomes "tainted." Malicious input can change the behavior of the process, and the process can then use this behavior to write to more trusted objects. Biba stops this at the source.

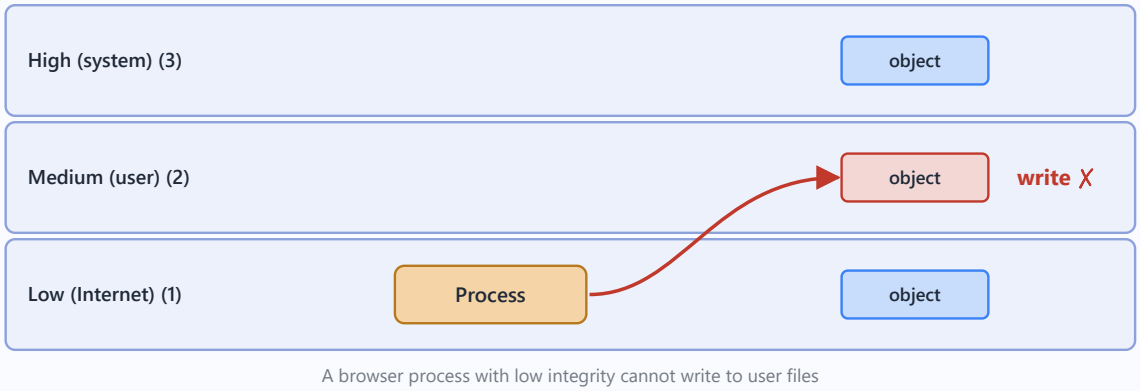


Figure 13.24: No write up in Biba: a browser process with Low integrity cannot write to the user files at the Medium level.

A real application of Biba is Windows's **Mandatory Integrity Control** (MIC). In Windows, processes and objects carry one of the integrity levels Low, Medium, High, and System. Ordinary applications run at the Medium level. The tab processes of browsers run at the Low level. Even if an exploit in a web page takes over the browser process, it cannot write to the user's documents at the Medium level. This is exactly the no-write-up rule.

13.4.3 Comparing the two models

The two models use the same structure in opposite directions (Table 13.12).

	Bell-LaPadula (1973)	Biba (1977)
Protects	Confidentiality	Integrity
Reading	Down free, up forbidden	Up free, down forbidden
Writing	Up free, down forbidden	Down free, up forbidden
Prevents	Information leaking downward	Tainted data spreading upward
Today	SELinux MLS, military systems	Windows MIC, FreeBSD and Linux integrity policies

Table 13.12: Bell-LaPadula and Biba.

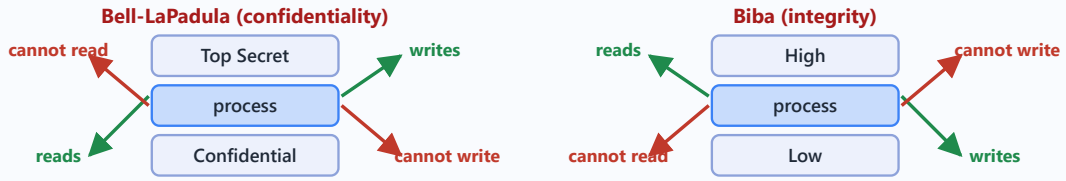


Figure 13.25: The directions in which a process can read and write under the two models. Instead of memorizing the arrows, remember what each model prevents.

Example 13.10

A process is at the Medium level. What do the two models say about the requests to read High, write High, read Low, and write Low? If both are applied together, what can the process access?

Request	BLP (confidentiality)	Biba (integrity)
Read High (read up)	deny	allow
Write High (write up)	allow	deny
Read Low (read down)	allow	deny
Write Low (write down)	deny	allow

Table 13.13: Four requests by a Medium-level process. The decisions were verified with Python.

In every row, one of the models denies. If both models are applied together, the process can read and write only objects at its own level. One protects confidentiality, the other integrity, and their goals conflict.

The names are easy to confuse. In BLP, the simple security property forbids reading up and the * property forbids writing down. In Biba, the simple integrity property forbids reading down and the * integrity property forbids writing up. In both models the word “simple” marks the read rule and the star marks the write rule.

Common mistake

Remembering the BLP rules as “no write up.” In confidentiality the danger is information flowing *down*; the write that is forbidden is the one that goes downward. The way to derive the rule is this: first ask what the model wants to protect, then think about which direction of information flow would break that goal.

The two models share a limitation: they control only explicit read and write operations. Information can be transferred without being written to a file. The next topic deals with these paths.

13.4.4 Covert channels and steganography

In 1973 Lampson asked a question: can a program be confined completely, so that it does not leak the secret information it processes while providing its service? This is called the **confinement problem**. The scenario is as follows: a client gives a server its secret data to process. The server is malicious and wants to pass the data on to an accomplice outside. The operating system confines the server: it cannot write files, cannot connect to the network, and cannot exchange messages with the accomplice. Lampson's answer is negative.

Definition 13.15

A **covert channel** is a way of transferring information that the system designer did not intend for communication and that bypasses the security policy.

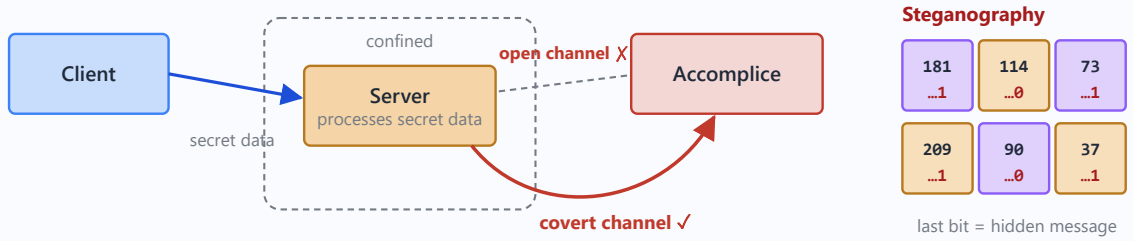


Figure 13.26: The confined server's open channel is closed, but it can leak data to the accomplice through a covert channel. On the right, steganography: a message hidden in the last bits of pixels.

Every shared resource can be a covert channel. Channels fall into three kinds.

Storage channel
: Information is transferred by a change in a shared state: whether a file exists, whether it is locked or open, how full the disk is.

Timing channel
: Information is transferred by the timing of events. If the server uses the CPU heavily it sends 1, if it stays idle it sends 0. The accomplice reads the bits by measuring how much its own work slows down. Cache latency is such a channel too.

Resource channel
: Information is transferred by how much of a shared resource is used: the page-fault rate, the volume of network traffic.

Closing every covert channel is practically impossible, because the state of every resource that processes share can be observed by the other processes. What can be done is to reduce the channels' **bandwidth**, that is, the number of bits per second they can carry. Adding noise, separating resources by level, and reducing the resolution of clocks are among these methods.

A close relative of the covert channel is **steganography**: hiding the existence of information instead of encrypting it. In a color image, the red, green, and blue components of each pixel are eight bits each. Changing the least significant bit of each component changes the color by at most 1/256. The eye cannot see the difference. The last bits of the six values in Figure 13.26 are 1, 0, 1, 1, 0, 1 in order. At one bit per channel, a 1920 × 1080 photograph can store 777,600 bytes. In Tanenbaum's well-known example, five Shakespeare plays are hidden inside a photograph with three zebras; the image does not change at all to the eye.

13.4.5 A covert channel through file locking

Let us build a concrete storage channel. The server and the accomplice agree in advance: in each time interval, if file F is locked, a 1 is considered sent; if it is free, a 0. To send a bit, the server locks or releases F. In the same interval the accomplice tries to lock F. If the attempt fails, the file is locked and the bit is 1; if it succeeds, the bit is 0. In this arrangement there is neither a read nor a write; the contents of F never change. No operation that BLP controls takes place, yet information flows.

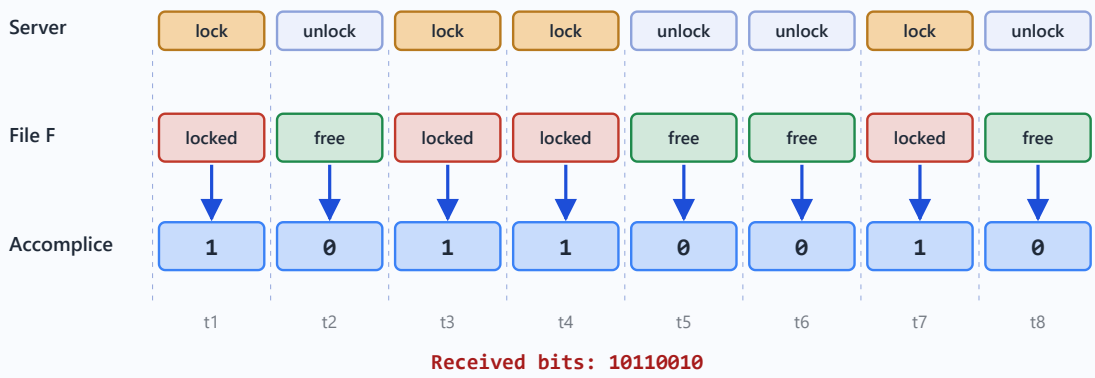


Figure 13.27: One byte over eight intervals: the accomplice reads the sequence 10110010 from its lock attempts.

In the timeline of Figure 13.27, over eight intervals the server switches F in the order lock, unlock, lock, lock, unlock, unlock, lock, unlock. The accomplice receives the sequence 10110010, that is, the value 0xB2 = 178. If the interval is 10 ms, there are 100 intervals per second, giving a channel of about 100 bit/s. A 256-bit encryption key leaks at this speed in 2.56 seconds.

Example 13.11

The server sends the value `secret = 'K'`, starting from the most significant bit:

```
for (int i = 7; i >= 0; i--) {
    if ((secret >> i) & 1) lock(F);
    else                   unlock(F);
    sleep_ms(10);
}
```

1. The ASCII code of 'K' is 0x4B; in binary 01001011.
2. From i = 7 down to 0 the bits are 0, 1, 0, 0, 1, 0, 1, 1. The server does unlock, lock, unlock, unlock, lock, unlock, lock, lock in turn.
3. In each interval the accomplice computes `bit = try_lock(F) ? 0 : 1` and after 80 ms obtains 01001011 = 0x4B = 'K'.

An administrator who finds the channel can lengthen the interval or add random delays to the lock state. These measures reduce the speed of the channel, but do not bring it to zero.

13.5 Authentication

All the decisions of the protection matrix rest on knowing which user a process belongs to. This is established when the user identifies himself as the system starts. If authentication is weak, even the best access control is useless: the attacker walks in under the identity of a legitimate user and uses all of that user's rights.

13.5.1 Authentication factors

Definition 13.16

Authentication means verifying that a user is who he claims to be. Deciding which operations will be permitted is a separate step, called **authorization**.

Authentication can rely on four kinds of evidence, that is, **factors**.

Something you know
: A password, a PIN, a security question. It is cheap and usable everywhere. But it can be stolen or guessed, and users reuse the same password on many sites.

Something you have
: A phone, a smart card, a FIDO2 security key, a TOTP generator. It can be stolen physically, but it cannot be copied remotely.

Something you are
: A fingerprint, a face, an iris. It is easy to use, but it cannot be changed: if it leaks, it is permanently "burned."

Where you are or how you behave
: Location, IP address, typing rhythm, device fingerprint. Usually used not alone but as an additional risk signal.

Multi-factor authentication (MFA) means asking for at least two factors from different categories together. The category condition matters. "Password + security question" is not MFA, because both are something you know, and the same phishing page can collect both at once. MFA is by itself the most effective measure against stolen passwords.

Older systems used a kind of **challenge-response** verification. The user was asked questions at registration, and one of them was chosen at random at login. Typical questions were the mother's maiden name or the name of the first pet. Because most of these answers can be found easily on social media today, the method is considered weak. The strong form of the challenge-response idea is cryptographic: the server sends a random number, and the user's device processes it with a secret key. FIDO2 uses this idea.

13.5.2 Password storage: salt and slow hashing

Passwords are never stored in the clear. If they were, an attacker who got hold of the password file, or an administrator who could read the file, would learn all the passwords at once. Since users use the same password elsewhere, the harm would not stay limited to one system. The system stores the digest of the password instead. At login, the digest of the password the user typed is computed and compared with the stored one.

But a plain digest is not enough (Figure 13.28). People often choose the same passwords. Since the same password gives the same digest, an attacker looking at the digest file immediately sees which users chose the same password. Worse, the attacker can compute the digests of millions of common passwords once in advance and put them into a table. Such tables are called **rainbow tables**. Every digest in a leaked file is looked up in this table, and thousands of passwords are cracked in one pass.

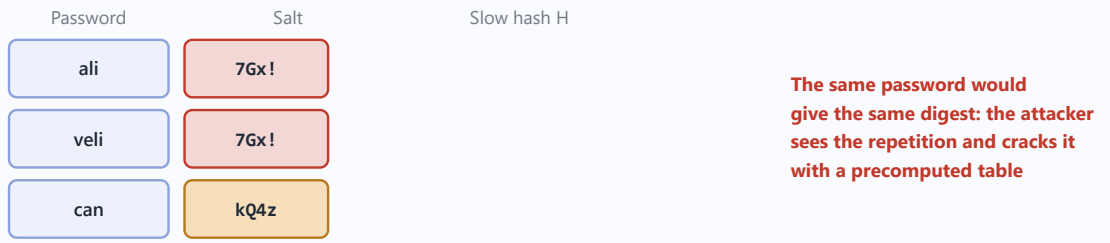


Figure 13.28: With unsalted digests, ali and veli have the same password and so the same digest; the attacker sees the repetition.

Definition 13.17

A **salt** is a random value generated separately for each user and added to the password before hashing. The salt is not secret; it is stored in the clear next to the digest.

The effect of the salt is simple: the same password is now a different input. Even if ali and veli have the same password, their digests are completely different because their salts differ. Precomputed tables are useless too, because a separate table would be needed for every possible salt. Even 1970s UNIX used a 2-character, that is 12-bit, salt. This meant each password could be stored in 4096 different ways and that the table size would grow 4096-fold.

A salt alone is not enough. General-purpose hash functions such as SHA-256 are designed to be fast. A current graphics card can compute billions of SHA-256 hashes per second. In password storage the opposite is wanted: each guess should be expensive. For this reason passwords are hashed with deliberately slow, memory-hungry functions. The current recommendation is **Argon2id** (RFC 9106); scrypt and bcrypt are also used. One verification takes a few milliseconds and the user does not notice. Billions of guesses, however, take years.

```
# Registration / password change
salt = os.urandom(16)
h = argon2(password, salt)
store(user, salt, h)

# Login attempt
salt, h = lookup(user)
g = argon2(attempt, salt)
if constant_eq(g, h): ok()
```

The pseudocode in the slides shows two operations. At registration or password change, a 16-byte random salt is generated. The password is passed through Argon2 with this salt, and the user name, salt, and digest are stored together. At login, the user's salt and digest are looked up, the attempt is hashed with the same salt, and the two digests are compared.

Doing the comparison with `constant_eq` is not a detail. An ordinary comparison stops at the first differing byte. If the attacker can measure the response time very precisely, he can deduce from the time how many leading bytes are correct. A constant-time comparison always compares all the bytes and prevents this timing leak.

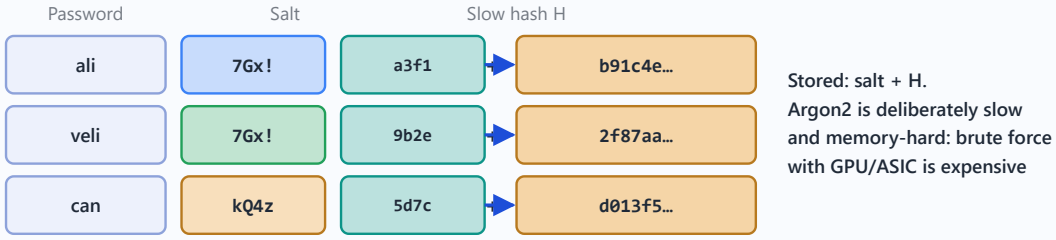


Figure 13.29: Each user has a separate salt; the same password gives different digests. What is stored is the salt and the slow digest.

Common mistake

Thinking that the salt must be kept secret, or that a salt makes weak passwords safe. The salt prevents only mass attacks and precomputation. If the attacker targets a single user, he knows that user's salt and tries dictionary passwords one by one. Against a weak password, a slow hash and a long password help. After a leak, passwords must be reset in every case.

13.5.3 Brute-force time

It is instructive to see the separate contributions of the salt and of slowness in numbers.

Example 13.12

Passwords consist of 8 lowercase letters (a 26-letter alphabet). The attacker wants to crack the digest of a single user in a leaked database. How long does it take in the worst case?

1. **Number of possible passwords.** There are 26 choices for each position: $26^8 = 208{,}827{,}064{,}576 \approx 2.09 \times 10^{11}$.
2. **Plain SHA-256.** At 10^{10} guesses per second: $2.09 \times 10^{11} / 10^{10} \approx 21$ seconds.
3. **Argon2id.** Since each guess is expensive, let only 10^4 guesses per second be possible: $2.09 \times 10^{11} / 10^4 \approx 2.09 \times 10^7$ seconds ≈ 242 days. The slow hash has made each guess 10^6 times more expensive.
4. **A longer password.** With 10 characters and 62 symbols (uppercase, lowercase, and digits), the space becomes $62^{10} \approx 8.4 \times 10^{17}$. Even with plain SHA-256 this takes about 2.66 years, and with Argon2id about 2.66 million years.

All values were computed with Python.

Where is the salt in this calculation? The salt does not change the time for a single user. The effect of the salt shows up when the attacker wants to crack all the users. In the unsalted arrangement, the attacker computes the digest of each candidate once and compares it with the digests of a million users at the same time: the work is the same as for a single user. In the salted arrangement he has to start over for each user. With a million users and plain SHA-256, the 21-second job grows to $10^6 \times 21$ seconds, about 242 days. The salt multiplies the work by the number of users; slowness raises the price of each guess. Neither can replace the other.

13.5.4 MFA and TOTP

The most common second factor added next to the password is the phone. The phone either receives a code from the server or generates the code itself (Figure 13.30).

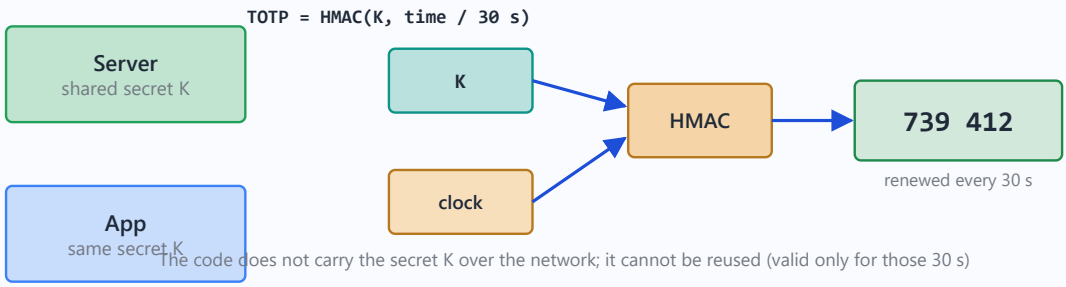


Figure 13.30: TOTP: the server and the app know the same secret K and the same time; the code is derived from HMAC(K, time / 30 s).

SMS code

Common but weak. The attacker can trick the carrier into moving the number to his own SIM card (SIM swapping) or redirect messages through weaknesses in the SS7 protocol of the phone network. It is still better than having no second factor at all.

TOTP generator app

Time-based one-time password (TOTP, RFC 6238). At registration the server and the app share a secret key, usually through a QR code. From this key and the clock, the app generates a new code every 30 seconds. It works offline. But it is open to phishing: the user can type the code into a fake site as well.

Push notification approval

The user taps a login approval notification on the phone. It is easy, but vulnerable to *approval fatigue* attacks: the attacker sends dozens of requests in a row and the user finally gets tired and approves one. For this reason *number matching* is added, in which the user types a number shown on the screen into the phone.

Note that a TOTP code does not carry the secret key over the network. The code is a one-way function of the key and the time. Even if an attacker sees a code, he cannot recover the key from it. The code is also valid only within its own 30-second window.

13.5.5 Code trace: generating a TOTP code

How the phone and the server produce the same code can be seen in the C function in the slides. The trace uses the test key of RFC 6238: the 20 bytes of the ASCII string “12345678901234567890”.

```
uint32_t totp(uint8_t *K, uint64_t t) {
    uint64_t T = t / 30;    /* 30 s */
    uint8_t h[20];
    hmac_sha1(K, T, h);
    int o = h[19] & 0x0F;   /* offset */
    uint32_t v = (h[o] & 0x7F) << 24
               | h[o+1] << 16
               | h[o+2] << 8
               | h[o+3];
    return v % 1000000;     /* 6 digits */
}
```

The function works in four steps.

1. **Time step.** The Unix time t (seconds since 1970) is divided by 30: `T = t / 30`. Because this is integer division, every moment in the same 30-second window gives the same T. Seconds 0–29 fall into T = 0, seconds 30–59 into T = 1, and seconds 60–89 into T = 2. For t = 59, T = 1.
2. **HMAC.** T, as an 8-byte big-endian number, is passed through HMAC-SHA1 together with K. The result is a 20-byte digest: `75 a4 8a 19 d4 cb e1 00 64 4e 8a c1 39 7e ea 74 7a 2d 33 ab`. Someone who does not know K cannot produce this value.

3. **Dynamic truncation.** 4 of the 20 bytes must be chosen. The digest itself determines where: the low half of the last byte is the offset. h[19] = 0xab; 0xab & 0x0F = 0xb = 11. So o = 11.
4. **The code.** The bytes h[11] … h[14], that is c1 39 7e ea, are combined into a 32-bit number. The top bit of the first byte is cleared with `& 0x7F` (0xc1 → 0x41). The reason is to keep the number from coming out negative on systems where it is interpreted as signed. v = 0x41397eea = 1094287082. The last six digits are v mod 1,000,000 = 287082.

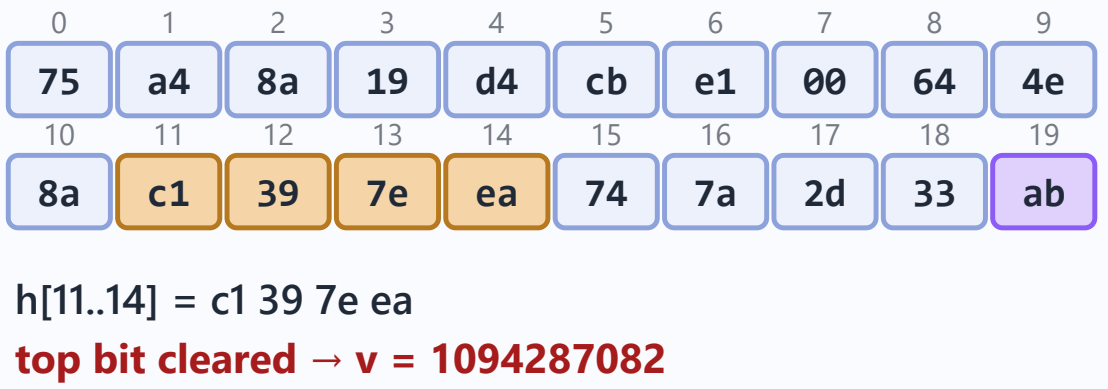


Figure 13.31: h[19] = 0xab selects the offset 11; h[11..14] = c1 39 7e ea, and with the top bit cleared, v = 1094287082.

The phone shows 287082; the server makes the same computation and compares it with the value the user typed. After 30 seconds, everything changes (Table 13.14). At t = 75, T = 2. A T that is larger by just one gives a completely different digest. This time the last byte is 0x44, the offset is 4, and the code is 359152.

t	T	h[19]	o	h[o..o+3]	v	Code
59	1	0xab	11	c1 39 7e ea	1094287082	287082
75	2	0x44	4	08 2f ef 30	137359152	359152

Table 13.14: The TOTP computation at two moments. The values were recomputed with Python's `hmac` library; the 8-digit result (94287082) is the same as the test vector of RFC 6238.

The phone's clock can drift a few seconds from the server's. Servers therefore usually accept the codes of the previous and the next step as well: T − 1, T, and T + 1. This means that a code is valid over a window of about 90 seconds. The probability of hitting the right code by random guessing is $3 / 10^6$ per attempt. As long as the server limits the number of attempts, this probability is negligible.

Example 13.13

What is T at t = 1,700,000,000? Over what interval is the same code valid? If the server accepts T − 1 … T + 1, over what interval could a code entered right now have been generated?

1. 1,700,000,000 / 30 = 56,666,666.67; the integer part gives T = 56,666,666.
2. The window starts at T · 30 = 1,699,999,980 and lasts 30 seconds: 1,699,999,980 … 1,700,000,009.
3. The start of T − 1 is 1,699,999,950 and the end of T + 1 is 1,700,000,039. The accepted window is 90 seconds.

13.5.6 FIDO2, WebAuthn, and passkeys

TOTP is strong against password theft, but weak against phishing. The user types both the password and the current code into a fake login page. The attacker relays these two to the real site within a few seconds and gets in while the code is still valid. The root of the problem is that no mechanism checks which site the user is giving the code to. FIDO2 and its web interface WebAuthn close this hole (Figure 13.32).

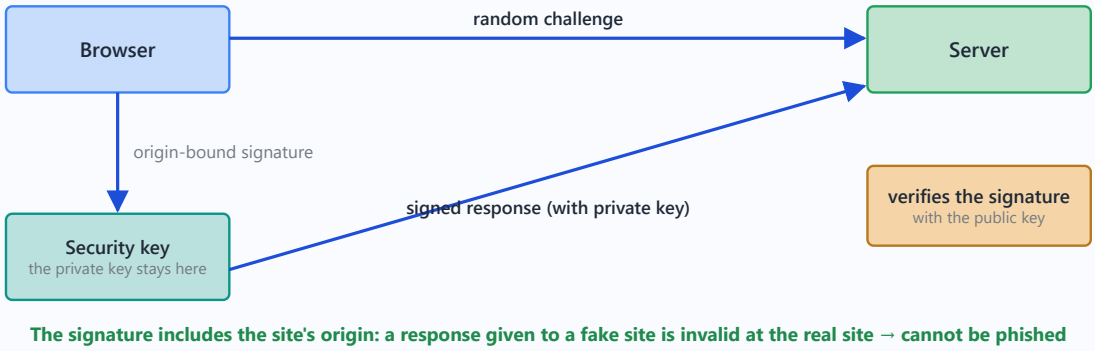


Figure 13.32: FIDO2: the security key signs the server's random challenge, bound to the origin of the site; the server verifies the signature with the public key.

The scheme rests on public-key cryptography. When the user registers with a site, the device generates a new key pair for that site. The private key never leaves the device's secure hardware, that is, the TPM, the Secure Enclave, or the USB security key. Only the public key is sent to the server. At login the server sends a random **challenge**. The browser adds to this challenge the **origin** of the site it is connected to, that is, the triple of protocol, domain name, and port. The device signs the two with the private key, and the server verifies the signature with the stored public key.

The resistance to phishing comes from this origin. If the user visits a fake site such as `example-login.com`, the browser puts that fake origin into the signature. Moreover, the device finds no registered key for the fake site. Even if the attacker relays the response to the real site, the signature

is invalid because the origin does not match. The user's carelessness does not change the outcome; the browser does the checking.

The other gains of the scheme matter too. The server stores only public keys; even if the database leaks, nobody can log in with these keys. There is no shared secret, and the same key is never reused on another site, because a separate pair is generated for each site. **Passkeys** are the form of FIDO2 credentials that can be synchronized securely across a user's devices. They are designed to replace passwords.

13.5.7 Biometric authentication

Biometrics measures a physical or behavioral characteristic of a person: fingerprint, face, iris, voice. At first sight it looks ideal: it cannot be forgotten, lost, or handed over. But every measurement is slightly different, and the system compares the measured value with the enrolled template against a threshold. So there are two kinds of error (Table 13.15).

Measure	Description
False acceptance rate (FAR)	Rate of accepting a stranger: a security risk
False rejection rate (FRR)	Rate of rejecting the real user: a usability problem
Liveness test	Check against a photograph, a mask, or a fingerprint mold
Irrevocability	If a fingerprint leaks it cannot be changed; a password can

Table 13.15: Measures of biometric systems.

FAR and FRR are linked. If the threshold is tightened, fewer strangers are accepted, but the real user is rejected more often. If it is loosened, the opposite happens. Even a small FAR matters at large numbers. In a system with a FAR of 1/100,000, the probability that at least one of a thousand different strangers' attempts is accepted is $1 - (1 - 10^{-5})^{1000} \approx 1\%$.

The fundamental limit of biometrics is this: it is good for an identity but bad for a secret. Faces and fingerprints are not secret; people leave fingerprints on the surfaces they touch every day and photographs on social media. The right use is therefore local. Biometrics works as a factor that unlocks secure hardware inside the device; for example, Face ID opens access to the keys in the Secure Enclave. The raw template never leaves the device. Alone it is not suitable for remote authentication, and it is usually combined with a device in the user's hand. The first devices that measured finger length are historical today. Today's systems use the minutiae of fingerprint ridges, the three-dimensional shape of the face, and the iris pattern. A hardware-backed liveness test is also essential.

13.5.8 How do attackers get into a system?

The weak points of authentication are the entry paths that attackers use most often (Table 13.16).

Path	Description	Countermeasure
Phishing	Collecting passwords with a fake login page	FIDO2, user training
Credential stuffing	Trying leaked passwords on other sites	Unique passwords, MFA
Brute force / dictionary	Guessing a weak password	Slow hashing, rate limiting, lockout
Unpatched vulnerability	Exploiting a known vulnerability	Fast patch management
Social engineering	Tricking the help desk	Process and verification
Default or weak credentials	admin/admin, a port left open	Hardening

Table 13.16: Common initial-access paths and countermeasures.

A real attack is seldom a single step; it is usually a chain (Figure 13.33). The attacker first obtains *initial access* through phishing or a vulnerability. This access is usually an ordinary user account. Then, with a local vulnerability, he performs *privilege escalation* and becomes root. Next he spreads to other machines on the same network by *lateral movement*. Finally he achieves *persistence* by installing a backdoor or a rootkit: he stays inside even if the machine is rebooted.

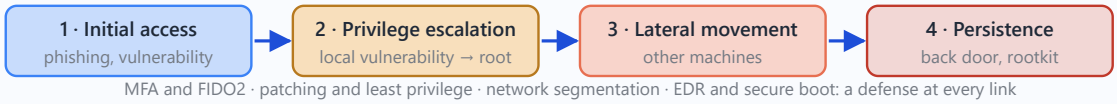


Figure 13.33: The attack chain and the defense at each link: MFA and FIDO2, patching and least privilege, network segmentation, EDR and secure boot.

The chain view shows once again that the defense must be layered as well. MFA makes initial access harder. Patching and least privilege block escalation. Network segmentation stops lateral movement. Endpoint monitoring and secure boot catch persistence. The failure of a single link must not lead to the takeover of the whole system.

13.6 Software vulnerabilities

Even if authentication and access control were perfect, bugs in programs would open new doors for attackers. A bug becomes a security vulnerability when it makes the program do something its designer never foresaw. The most dangerous are memory errors that let the attacker run code of his own choosing. This section covers the most classic of them, the stack buffer overflow, the defenses developed against it, and TOCTOU, a race-condition vulnerability.

13.6.1 Stack buffer overflow

When a function is called in C, a **frame** is opened for it on the stack. The frame holds the function's local variables, the caller's saved frame pointer, and the **return address**. The return address is where the processor will continue executing when the function ends. On the x86 family the stack grows from

high addresses to low addresses. So the return address of the called function is pushed first, and local arrays sit at lower addresses than it.

Definition 13.18

A **buffer overflow** is writing more data into a buffer than its size, so that the excess bytes spill over into adjacent memory. If a buffer on the stack overflows, the most dangerous victim is the return address.

```
/* small, deliberately buggy example */
void A(char *in) {
    char buffer[8];
    strcpy(buffer, in);   // NO bounds check
    return;
}
A(argv[1]);               // in: user input
```

This small, deliberately buggy function copies a string that comes from the user into an 8-byte array. `strcpy` does not know the size of the destination; it keeps copying until the terminating `\0` in the source. As long as the input is 7 characters or fewer there is no problem: at most 8 bytes are written, terminator included, and the array does not overflow. If the input is longer, the excess bytes are written into the cells above the array. Because the array sits at a low address, the return address at a high address, and copying proceeds toward increasing addresses, the overflow heads straight for the return address (Figure 13.34).

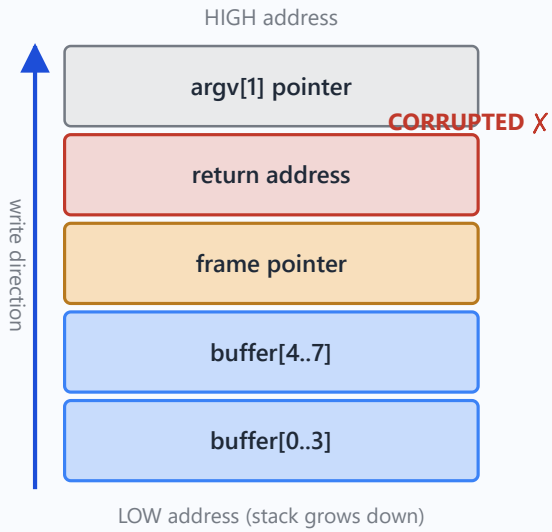


Figure 13.34: The stack frame. The writing direction is upward; a long input corrupts first the frame pointer and then the return address.

When the function executes `return`, the processor reads the return address from the stack and jumps there. If the address has been corrupted with random bytes, the program most likely crashes with a segmentation fault (SIGSEGV). But an attacker who chooses the input can put an address of his own choosing in place of the return address. The flow of execution then goes wherever the attacker wants. In the classic attack, this address points to machine code that the attacker has placed in the buffer with the same input. If the program is setuid root or a network service, the attacker obtains a shell with root rights. The paper “Smashing the Stack for Fun and Profit” by Aleph One, published in 1996, described this technique in detail and began a generation of attacks.

Example 13.14

On a 64-bit machine, “Giresun-Ordu-Samsun” is copied with `strcpy` into the array `char name[8]`. There is no canary and no padding: the frame holds the 8-byte `name`, the 8-byte saved frame pointer (rbp), and the 8-byte return address. How many bytes are written, how many overflow, and how many bytes of the return address are corrupted?

1. The string has 19 characters. `strcpy` writes 19 + 1 = 20 bytes, terminator included.
2. The first 8 bytes, “Giresun-”, fit the array exactly. Since no `\0` has come yet, copying continues.
3. The next 8 bytes, “Ordu-Sam”, are written over the saved rbp.
4. The remaining “sun” and `\0` are written into the first four bytes of the return address: 73 75 6e 00. Because the machine is little-endian, these bytes are the low four bytes of the address; the address becomes …006e7573.

Of the 20 bytes in total, 20 − 8 = 12 have overflowed; 4 bytes of the return address are corrupted. When the function returns, the processor jumps to this invalid address and the program crashes. The values were verified with Python.

The fix is to limit the copy to the size of the destination. `snprintf` copies at most `sizeof buffer - 1` characters and always adds a terminator.

```
void A(const char *in) {
    char buffer[8];
    snprintf(buffer, sizeof buffer, "%s", in);   /* at most 7 chars + '\0' */
}
```

Common mistake

Not counting the terminator. A string of 8 characters does not fit into an 8-byte array: `strcpy` writes 9 bytes and one byte overflows. Even a single overflowing byte can be exploited, for example by zeroing the lowest byte of the saved frame pointer. Bugs of this kind are called *off-*

by-one errors.

13.6.2 Defenses against buffer overflows

Buffer overflows have been known for more than forty years, and many layers of defense have been developed against them. Each defense targets a different step of the attack.

Stack canary
: The compiler places a randomly chosen value, picked at process start, just below the return address. Before the function returns, it checks whether this value has been corrupted. An overflow that wants to reach the return address must first trample the canary. If corruption is seen, the program is stopped. In GCC this feature is known as stack-smashing protection (SSP).

DEP / NX
: Data pages are marked non-executable in the page table (the NX bit on x86-64). Code injected into the stack does not run even if the processor jumps there; a page fault occurs. The rule that a page must not be both writable and executable is also called W^X.

ASLR
: Address space layout randomization: the stack, the heap, and the libraries are loaded at different addresses on every run. The attacker cannot know which address to jump to. If the program itself is compiled as position-independent (PIE), its address is randomized too. On Linux, library placement in 64-bit processes carries 28 bits of randomness; about 268 million possibilities.

CFI
: Control-flow integrity: indirect jumps and calls are allowed to go only to the valid targets that the compiler determined in advance.

Shadow stack
: A copy of the return address is kept on a separate stack that the program cannot modify with ordinary write instructions. At return, the two addresses are compared. Intel CET and ARM GCS provide this in hardware. ARM's pointer authentication (PAC) and branch target identification (BTI) are separate protections with a similar aim.

Memory-safe languages
: In languages such as Rust, Go, Java, and C#, array bounds and object lifetimes are checked by the compiler or the runtime. Buffer overflows cannot arise in these languages in the first place. This is the most fundamental solution.

As defenses developed, attacks changed too. When NX stopped code injected into the stack from running, attackers stopped bringing in new code. Instead they began to use code that already exists in memory. In a **return-to-libc** attack, the return address is redirected to the `system()` function in libc and given the argument "/bin/sh". The more general form is **return-oriented programming** (ROP). The attacker finds small pieces of code (*gadgets*) ending in a `ret` instruction inside the program and its libraries. He writes the addresses of these pieces onto the stack one after another. Each piece executes a few instructions and moves on to the next with `ret`. With enough pieces, any

computation can be performed. Because all this code already sits in executable pages, NX cannot stop this attack. ASLR hides the addresses of the pieces, and CFI and the shadow stack keep returns from going to unexpected places. This is why defenses are used together; none of them is enough alone.

13.6.3 Code trace: the stack canary

Let us follow how the canary works byte by byte. In the code below, the canary lines are added not by the programmer but by the compiler; in real compiled code these lines are the function's prologue and epilogue instructions.

```
/* the compiler adds the canary lines */
void copy_in(const char *input) {
    long canary = __stack_chk_guard;
    char buf[8];
    strcpy(buf, input);
    if (canary != __stack_chk_guard)
        __stack_chk_fail();
}
/* input: "AAAAAAAABBBBBBBB" */
```

The function is called with the input “AAAAAAAABBBBBBBB”: 16 characters and a terminator. `buf` is 8 bytes. In the frame, from the low address toward the high one, are `buf`, the canary, the saved rbp, and the return address. The trace proceeds through the steps of Table 13.17.

Step	buf[8]	canary	saved rbp	return address
call	(undefined)	(undefined)	f0 c4 20 3a fd 7f 00 00	96 11 40 00 00 00 00 00
prologue	(undefined)	00 5f b8 d4 e2 71 3c 9a	same	same
first 8 bytes	A A A A A A A A	same	same	same
next 8 bytes	A × 8	B B B B B B B B	same	same
`\0`	A × 8	B × 8	**00** c4 20 3a fd 7f 00 00	96 11 40 00 00 00 00 00
exit check	0x4242424242424242 ≠ 0x9a3c71e2d4b85f00 → `__stack_chk_fail()`			

Table 13.17: The state of the frame in the call `copy_in("AAAAAAAABBBBBBBB")`. The bytes are written in memory order (from the low address to the high one). The values were verified with Python.

1. **Call.** The `call` instruction pushes the return address (0x401196) onto the stack. The function's prologue saves the caller's frame pointer (0x7ffd3a20c4f0). In little-endian layout the lowest byte of an address comes first: 0x401196 appears in memory as 96 11 40 00 ...
2. **Canary.** The prologue that the compiler adds places the value `__stack_chk_guard`, chosen at random at process start, between `buf` and the saved rbp. In this example the value is 0x9a3c71e2d4b85f00. That its first byte, the lowest byte in memory, is 0x00 is a deliberate choice. Because string functions stop at this byte, the canary cannot be leaked by a string read and cannot be written verbatim by `strcpy`. The remaining 56 bits are random: 2^{56} possibilities.

3. **First 8 bytes.** `strcpy` writes "AAAAAAAA" into `buf`. So far there is no problem.
4. **Overflow.** "BBBBBBBB" is written outside `buf`, over the canary. The canary is now 0x4242424242424242 (the ASCII code of B is 0x42).
5. **Terminator.** `\0` is written into the first byte of the saved rbp: f0 → 00. If the input were a little longer, the return address would be next. But the only way to reach the return address is to pass over the canary.
6. **Exit check.** Before the function returns, the epilogue compares the canary on the stack with the value from process start. The two differ.
7. **Abort.** `__stack_chk_fail` prints the message "*** stack smashing detected ***: terminated" and ends the process with SIGABRT. The corrupted frame is never returned to.

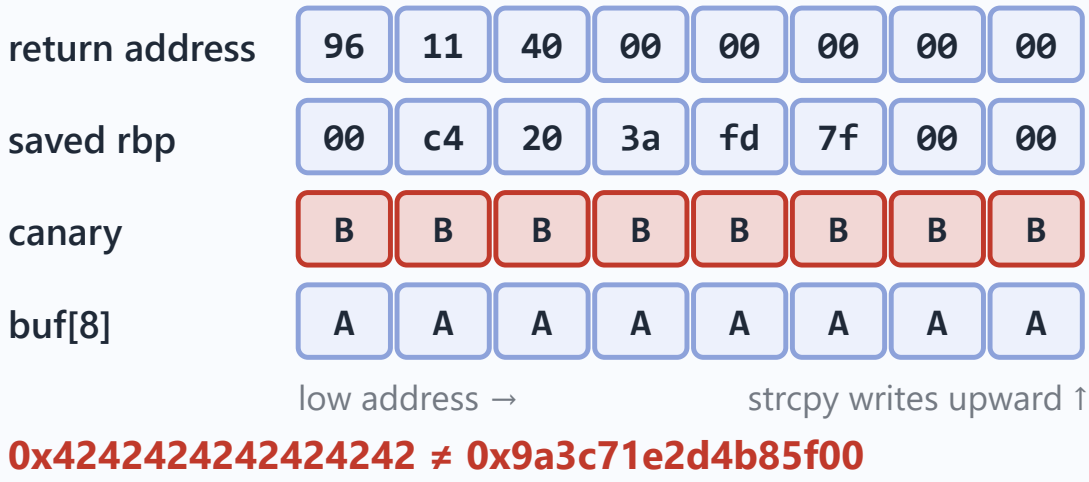


Figure 13.35: The exit check: the canary on the stack (the B's) does not match the value from process start.

The gain of the canary is that it turns a code-execution attack into a crash. That is still a loss, because the service stops and availability is damaged. But the attacker cannot run his code. The canary has limits too. If the attacker can read the canary's value through another bug, he can write it back unchanged. If the overflow corrupts a function pointer instead of the return address, the canary never comes into play.

13.6.4 Other memory and injection bugs

The buffer overflow is only one member of a broad family of bugs (Table 13.18).

Vulnerability	Why it is dangerous	Countermeasure
Format string	`printf(user)` : reads memory with %x, writes with %n	`printf("%s", user)`
Integer overflow	`len = a + b` wraps around and leads to a small allocation	Checked arithmetic, size types

Vulnerability	Why it is dangerous	Countermeasure
Use-after-free	Access to freed memory (UAF)	Ownership (Rust), smart pointers, quarantine
Double free	The same block is `free`d twice; the heap is corrupted	NULL after `free`, hardened allocator
Out-of-bounds read	Memory leak such as Heartbleed	Bounds checking, fuzzing
Command/SQL injection	Input mixes into code or a query	Parameterized queries, escaping, allowlists

Table 13.18: Memory and injection bugs.

In a **format-string** bug, the programmer passes the user's input directly to `printf` as the format string. If the user writes `%x` in the input, `printf` reads non-existent arguments from the stack. The contents of memory are thus dumped to the screen. `%n`, on the other hand, *writes* the number of characters printed so far to an address; with it the attacker can modify memory. In an **integer overflow**, the sum of two lengths exceeds a 32-bit variable and wraps around to a small number. The program allocates memory with this small value, then copies large data into that area, and an overflow occurs on the heap.

In **use-after-free**, the program keeps accessing a block it has freed through an old pointer. If the allocator has meanwhile given the block to another object, the attacker, by controlling the contents of that object, determines how the program interprets the old object. A large share of browser vulnerabilities are of this kind. The famous example of an **out-of-bounds read** is Heartbleed, found in OpenSSL in 2014. The server copied as many bytes into its reply as the length the client declared, without checking it. With every request, up to 64 KiB of its memory leaked, possibly including private keys and passwords.

The common root of these bugs is that data gets mixed with code or control information and that untrusted input is used unchecked. In the format string, data is interpreted as a format command. In SQL injection, a quote character typed by the user changes the structure of the query. The defense is common too: validate input, separate data from code, use memory-safe languages. **Fuzzing** tools, which feed the program random malformed inputs, also find bugs before attackers do.

13.6.5 The TOCTOU race condition

Some vulnerabilities are not memory errors but timing errors. A program checks a condition and, some time later, performs an operation based on that check. If the condition can change between the two moments, the program is now acting on invalid information.

Definition 13.19

A **TOCTOU** (time of check to time of use) vulnerability is a race condition. An attacker can change the resource between the moment it is checked and the moment it is used.

```
/* TOCTOU: a setuid-root helper */
if (access(path, W_OK) == 0) {   // CHECK
    fd = open(path, O_WRONLY);     // USE
    write(fd, data, n);
}
```

The code is part of a setuid-root helper program. The program wants to prevent itself from writing, with root rights, to a file that the user who runs it has no write permission for. It uses the `access` call for this. `access` checks permission not against the effective ID but against the *real* ID: it answers the question "can ali write to this file?" If the answer is yes, the program opens the file with `open`. `open`, however, uses the effective ID, that is, root.

The problem is that the two calls resolve the file name separately (Figure 13.36).

1. **Check.** `/tmp/f` is an ordinary file owned by Ali. `access` says the permission is there.
2. **Window.** There is a small time gap between the check and the use. The scheduler may take the process off the CPU at exactly this point.
3. **Attack.** In this gap the attacker deletes `/tmp/f` and puts in its place a symbolic link pointing to `/etc/passwd`: `ln -sf /etc/passwd /tmp/f`. The file name is the same, but the target has changed.
4. **Use.** `open("/tmp/f")` resolves the name again and, following the link, opens `/etc/passwd` with root rights. `write` corrupts this critical file.

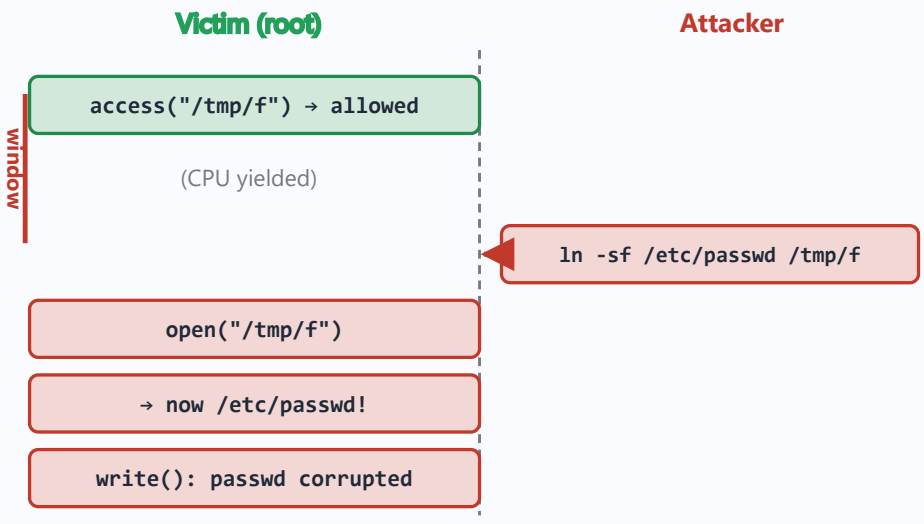


Figure 13.36: The TOCTOU timeline: in the window between check and use, the attacker replaces the file name with a symbolic link.

The window may be very short, but the attacker repeats the attempt thousands of times. On a multicore machine he improves his chances by running his own process on a separate core that keeps swapping the link. Hitting it once is enough.

The principle of the solution is not to resolve the name twice. The file is opened once, and all later checks are made on the same file descriptor. A descriptor stays bound to whichever file it was bound to at the moment it was opened; a later change of the name does not affect it. In addition, before opening, the program drops its privilege to the user's identity. Then the permission check is made not by `access` but by `open` itself, with the right identity. The `O_NOFOLLOW` flag refuses to open if the last component is a symbolic link.

```
/* fixed: the name is resolved once, the check is made on the same descriptor */
uid_t ruid = getuid();
seteuid(ruid);                                  /* drop privilege first */
int fd = open(path, O_WRONLY | O_NOFOLLOW);     /* the kernel checks permission */
if (fd >= 0) {
    struct stat st;
    if (fstat(fd, &st) == 0 && S_ISREG(st.st_mode))
        write(fd, data, n);
    close(fd);
}
```

Programs working in a secure directory use the `openat` family to resolve the path relative to a directory descriptor; this also prevents the intermediate components of the path from being replaced.

Common mistake

Treating a smaller window as a solution. Because `access` and `open` are written one after the other, the window is measured in microseconds; but a small window only raises the number of attempts the attack needs. The solution is to close the window: the check and the use must be performed on the same object, with a single name resolution.

13.7 Insider attacks

Some attacks come not from outside the system but from inside. The attacker is someone who develops or operates the system: a programmer, an administrator, a contractor. Because he has legitimate access, he has already got past all the outer defenses. This group has three classic methods.

Logic bomb

: Malicious code that is triggered under a certain condition. The condition may be a date, or an event such as the programmer's name being removed from the payroll. Often a disgruntled employee plants it: the code runs every day, checks whether the programmer is still with the company, and if not, deletes files.

Back door

A hidden entry that bypasses authentication (Figure 13.37). Back doors can also be slipped into the supply chain. In 2024, a back door was noticed at the last moment in the xz compression library used by Linux distributions. The back door had been added by a contributor who had earned trust over years; it affected SSH authentication.

Login spoofing

The attacker runs a program that looks like the real login screen and walks away from the terminal. The user who arrives types his user name and password. The program records them, shows a "wrong password" message, and hands over to the real login screen. The user gets in on the second attempt and suspects nothing.

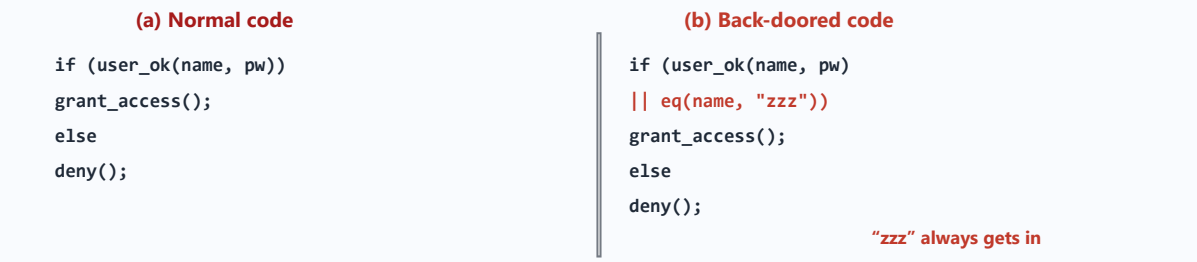


Figure 13.37: (a) Normal login code; (b) a back-doored version that always grants login for the user name "zzz."

Against login spoofing, a **secure attention key** is used. In Windows, no user program can capture the Ctrl+Alt+Del combination; these keys always directly summon the operating system's real login screen. If users are trained to press these keys before logging in, the real screen comes to the front even when a fake program is running on screen.

The defense against back doors is both organizational and technical. In code review, every change is seen by at least one other person. For critical changes, approval by two people is required. **Reproducible builds** let everyone obtain a byte-for-byte identical binary from the same source code. In this way it can be verified independently that the distributed program was built from the published source. Ken Thompson's 1984 Turing Award lecture shows the limit of this subject. Thompson describes a compiler that re-inserts the back door when it compiles itself and leaves no trace in the source code. In the end, every chain of trust begins somewhere with trusting a tool we did not write ourselves.

13.8 Malware

Malware is the general name for programs that run against the wishes of the system's owner. These programs are designed to do harm, steal data, or consume resources. The vulnerabilities of the earlier sections are the doors these programs use to get into the system.

13.8.1 Types

Malware is classified by how it spreads and by its purpose (Table 13.19).

Type	Spread	Purpose / behavior
Virus	Attaches itself to another program; infects when the user runs that program	Corrupts files, spreads
Worm	On its own over the network; no user needed	Spreads fast, consumes resources
Trojan horse	Inside a program that looks useful	Opens a back door, steals data
Ransomware	Phishing, vulnerabilities, RDP	Encrypts files, demands a ransom
Rootkit	Installed after privilege is gained	Hides itself and other malware
Spyware	Trojan horse, bundled software, fake add-on	Collects data, sends it away
Bot / botnet	Worm + back door	DDoS, spam, mining on remote command

Table 13.19: Types of malware.

The difference between a virus and a worm is whom the spread depends on. A **virus** needs a carrier: it settles in another program or document and becomes active when that file is run or opened. A **worm** is an independent program. It copies itself from machine to machine through an unpatched service on the network, and the user does not need to do anything. The Morris worm of November 1988 affected a significant share of the machines connected to the Internet of the time within a few hours. WannaCry in 2017 was a worm that carried ransomware, and it spread through a vulnerability in Windows's file-sharing protocol.

The **Trojan horse** takes its name from the wooden horse of the Trojan War. It is a game, add-on, or “free” piece of software that looks useful and that the user installs of his own will. Because it runs with the user's rights, it reaches everything that user can reach. **Spyware** monitors the user's activities and sends them away. It can change the browser's home page or search engine, add unwanted toolbars, and open pop-up ads that cannot be stopped.

Today the boundaries are blurred. A modern piece of malware usually combines several types: it spreads like a worm, hides with a rootkit, connects to a botnet, and finally runs a ransomware payload.

13.8.2 Parasitic viruses

The most common kind of classic virus is the **parasitic virus**, which settles in executable files. When infecting a file, the virus does two things: it adds its own code to the file and changes the program's entry point. When the program is run, the virus runs first, infects new files, and then jumps to the actual program. The user sees the program working normally and suspects nothing. Viruses fall into three kinds depending on where in the file they put their code.

Prepending
: The virus code is added to the beginning of the program and the entry point is turned to the virus.

Appending
: The virus is added to the end of the program; a jump to the virus is placed at the beginning of the program. This is the most common kind, because the rest of the program does not need to be shifted.

Cavity
: Executable files contain unused gaps because of alignment. The virus splits itself into pieces and scatters them into these gaps. Because the file size does not change, detection becomes harder (Figure 13.38).

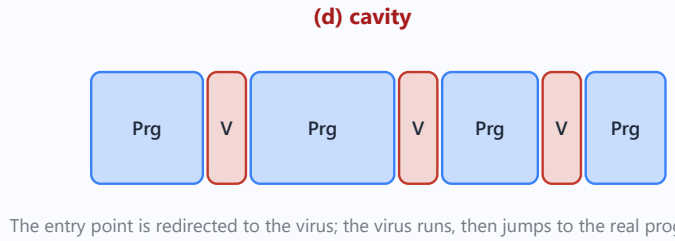


Figure 13.38: A cavity virus: the pieces are scattered over the unused areas of the program; the file size does not change.

There are other kinds. A *companion virus* does not touch the real program: in MS-DOS a .com file of the same name ran before the .exe, so it settles next to prog.exe under the name prog.com. *Memory-resident viruses* stay in memory and take over interrupt vectors. *Boot-sector viruses* run before the operating system. *Device-driver viruses* run in kernel mode. *Macro viruses* settle in the macros inside documents, and *source-code viruses* in source files before they are compiled. Most of these are historical today. Modern malware uses scripts, in-memory (*fileless*) techniques that never write a file to disk, and supply-chain paths.

13.8.3 Ransomware

Ransomware is malware that encrypts the victim's data and demands money in return for the decryption key. Today it is one of the most destructive threats organizations face. A typical attack proceeds in five steps (Figure 13.39).

1. **Initial access.** Usually through an attachment to a phishing e-mail, an unpatched service exposed to the Internet, or a stolen remote desktop (RDP) password.
2. **Spread.** The malware runs and reaches as many machines as possible by lateral movement on the network. Attackers may spend days or weeks at this stage.
3. **Deleting backups.** To prevent recovery, shadow copies and backups reachable over the network are deleted. This is why an offline or immutable backup is essential.

4. **Encryption.** Files are locked with a strong symmetric cipher, for example AES. The symmetric key is encrypted with the attacker's public key or sent to the attacker's server. The victim is left with no copy of the key.
5. **Ransom note.** Payment instructions are displayed on screen. In recent years, attackers also copy the data out before encrypting and threaten to publish it if payment is not made.

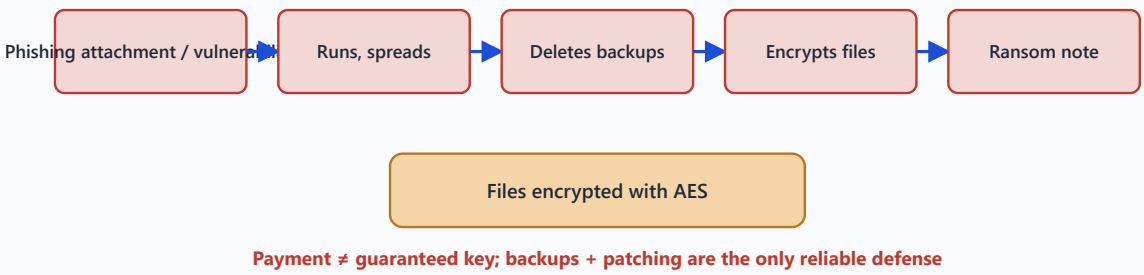


Figure 13.39: The ransomware attack chain. Payment does not guarantee that the key will be handed over; backups and patching are the only reliable defense.

Ransomware shows how sound cryptography can be used for a bad purpose. Used correctly, AES cannot be decrypted without the key. So once encryption has started there is little left to do; the defense must be prepared beforehand. Payment does not guarantee that the key will be handed over, and it encourages new attacks. The layers of defense are well-known tools: an immutable backup that follows the 3-2-1 rule, fast patching, MFA, network segmentation, and endpoint monitoring (EDR) that catches suspicious behavior.

13.8.4 Rootkits and concealment

A **rootkit** is a collection of programs that an attacker who has settled in a system uses to hide his presence. It takes its name from the root account in UNIX. A rootkit hides files, processes, and network connections from the operating system and its tools. There are kinds depending on the layer it hides in (Figure 13.40).

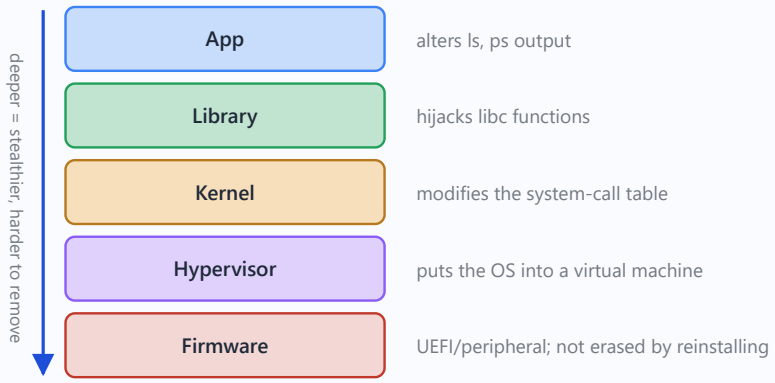


Figure 13.40: The layers in which rootkits can hide. The lower it goes, the stronger the concealment and the harder the removal.

Application level
: Tools such as `ls` and `ps` are replaced with versions that remove the malware's files and processes from their output. Easily caught by an integrity check.

Library level
: Functions in libc are hijacked. For example, the function that reads directories skips certain names. All programs that use this library are deceived.

Kernel level
: The rootkit is loaded as a kernel module and modifies the system-call table or the kernel's internal data structures. Every question asked of the kernel gets a false answer.

Hypervisor level
: The rootkit slips the running operating system into a virtual machine without being noticed and runs beneath it. Seen from inside the operating system, no trace is visible.

Firmware level
: It hides in UEFI firmware or in the firmware of a peripheral. It is not erased even if the operating system is reinstalled or the disk is replaced.

The deeper a rootkit settles, the more powerful it is, but also the harder it is to install. Scanning from inside is unreliable against rootkits in and below the kernel, because the rootkit answers the questions the scanner asks. The defense comes from several directions. Mandatory signed drivers ensure that only trusted modules are loaded into the kernel. Secure boot prevents a modified bootloader or kernel from running. Measured boot and the TPM record the digests of boot components and expose changes. As a last resort, the disk is scanned offline from a clean system.

13.8.5 Antivirus and evasion techniques

Antivirus software and virus writers have been in a race for a long time. Every detection method has given rise to a new evasion technique.

The oldest method is **signature scanning**. A characteristic byte sequence, a signature, is extracted from the code of every known virus. The scanner looks for these sequences in files. It is fast and has few false alarms, but it catches only known viruses. Virus writers first turned to **compression**: when an infected program is compressed, the signature becomes invisible. Scanners responded by unpacking files and scanning their insides. Then **encrypted viruses** appeared: the body of the virus is encrypted with a different key in each copy, and only a small decryptor stays in the clear. Scanners then started looking for the signature of the decryptor.

The answer to this is **polymorphic** viruses (Figure 13.41). These viruses change the decryptor in every copy too. Different instruction sequences that do the same job are produced: no-op instructions are inserted, registers are swapped, the order of instructions is shuffled. Since no common byte sequence remains between two copies, the signature method does not work.

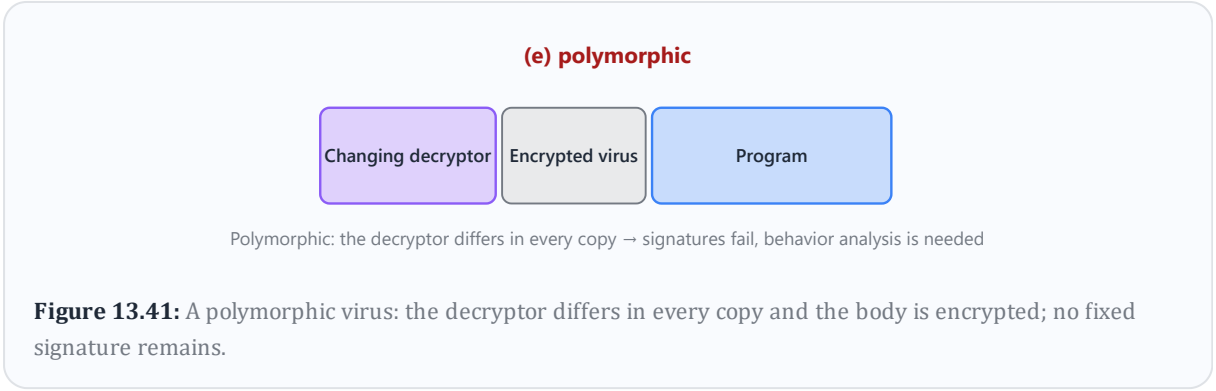


Figure 13.41: A polymorphic virus: the decryptor differs in every copy and the body is encrypted; no fixed signature remains.

Today's security software therefore uses several methods together.

Signature scanning
: Catches known patterns quickly and with a low false-alarm rate; misses new ones.

Integrity checking
: Stores digests of system files and notices any change.

Behavioral and heuristic analysis
: Looks not at what a program is but at what it does. Encrypting hundreds of files in a short time, injecting code into other processes, and deleting shadow copies are suspicious actions.

Sandbox and cloud
: A suspicious file is run in an isolated environment and its behavior is observed. The result is shared with all customers through the cloud.

13.9 Defense and isolation

If we accept that bugs and vulnerabilities will never disappear completely, two paths of defense remain. The first is to ensure that only code from trusted sources enters the system. The second is to limit the harm that untrusted or compromised code can do. This section presents code signing for the first path, and sandboxes, containers, and mandatory access control for the second.

13.9.1 Code signing

Code signing is the application of digital signatures to programs (Figure 13.42). The goal is to verify, before running a program, that it has not been modified and that it comes from the publisher it claims.

1. **Digest.** The publisher computes the digest of the program.
2. **Signature.** The digest is signed with the publisher's private key. The signature is distributed with the program.
3. **Distribution.** The user receives the program and the signature. The publisher's certificate carries its public key and chains to a certificate authority.
4. **Verification.** The operating system recomputes the digest and verifies the signature with the public key. If they match, the program has not been modified and is from that publisher.

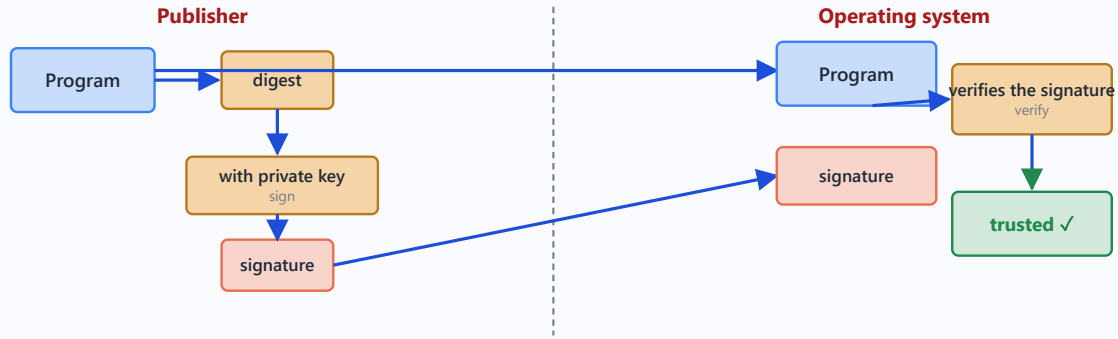


Figure 13.42: Code signing: the publisher signs the digest with its private key, and the operating system verifies it with the public key.

Most of today's systems rely on this foundation. Mobile app stores distribute only signed applications. In its 64-bit versions, Windows requires kernel drivers to be signed. The package managers of Linux distributions (`apt`, `dnf`) check the signature of every package against the distribution's key. Some platforms go even further and allow only signed code to run.

It is just as important what code signing does not prove. A signature shows who a program came from and that it was not modified on the way; it does not show that the program is harmless. If the publisher's signing key is stolen, or the publisher's build server is compromised, malicious code carries a valid signature too. Supply-chain attacks target exactly this point.

13.9.2 Sandboxes

Definition 13.20

A **sandbox** is a method of running an untrusted program with limited rights, in an environment isolated from the rest of the system. Even if the program is compromised, the harm it can do stays limited by the boundaries of the sandbox.

The idea is a practical answer to Lampson's confinement problem. Figure 13.43 uses an old analogy: the program is in a prison, and a guard checks every system call it makes. Calls that are not allowed are blocked. The `chroot` jails of early UNIX and FreeBSD's `jail` mechanism were the first implementations of this thought.

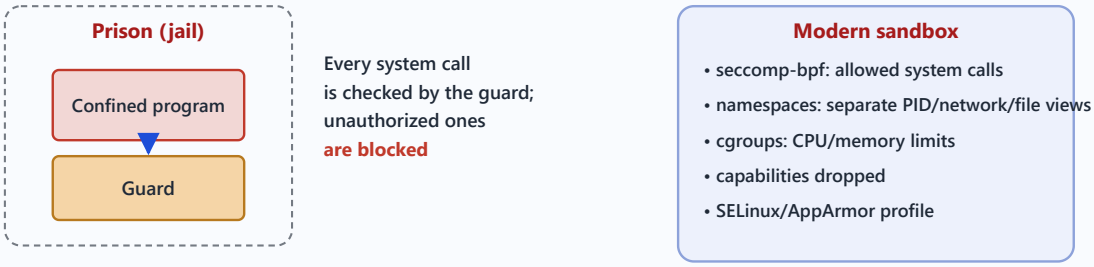


Figure 13.43: On the left, a confined program and its guard; on the right, the building blocks of a modern sandbox on Linux.

Today browser tabs, mobile applications, and many server services run in sandboxes. On Linux, a sandbox is a combination of several kernel mechanisms.

seccomp-bpf
: Limits the system calls a process may make with a filter. For example, an image decoder may call only `read`, `write`, and `exit`. If it tries to make any other call, the kernel kills the process.

Namespaces
: Give the process a separate view of the system. A process in a PID namespace sees only the processes in its own namespace; a network namespace has its own network interfaces, and a mount namespace has its own file-system tree.

cgroups
: Limit the amount of CPU, memory, and I/O that a group of processes can use. They prevent denial of service through resource exhaustion.

Dropping capabilities
: Even a process that starts as root drops all the Linux capabilities it does not need.

Mandatory access control

An SELinux or AppArmor profile separately restricts the files and resources the process can access.

13.9.3 Containers

The structure that emerges when namespaces and cgroups come together is called a **container** (Figure 13.44). An application in a container sees a file system, a network stack, and a process tree of its own; it runs as if it were alone on a machine. In reality, however, all containers share the same kernel. Tools such as Docker and Kubernetes are built on these kernel features.

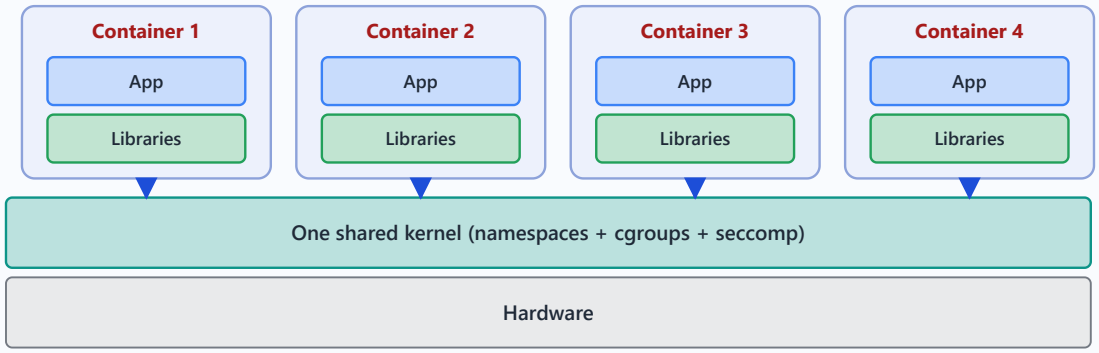


Figure 13.44: Four containers: each has its own application and libraries, but all share a single kernel.

Comparing a container with a virtual machine is instructive from a security point of view (Table 13.20). In a virtual machine, each guest runs its own kernel, and the guests are separated from each other by the hypervisor and by hardware. In a container, isolation rests entirely on the kernel's own features.

	Container	Virtual machine
Isolation	Kernel features	Hardware + hypervisor
Overhead	Low, starts in a second	High, separate kernel
Attack surface	Kernel vulnerability = escape risk	Stronger boundary

Table 13.20: Isolation of containers and virtual machines.

The consequence of this difference is clear. If a vulnerability is found in the shared kernel, an attacker in a container can use it to escape the container and reach all the other containers on the same machine. In a virtual machine, a vulnerability in the guest kernel affects only that guest; escaping requires a second vulnerability in the hypervisor. For this reason cloud providers usually run the containers of different customers in separate virtual machines or in lightweight virtual machines (for example Firecracker).

Common mistake

Considering it harmless to run as root inside a container. If a user namespace is not used, root in the container is real root (uid 0) as far as the kernel is concerned. A process that gets out of the container through a configuration mistake or a kernel vulnerability takes over the whole machine. Container processes should also run with least privilege.

13.9.4 Mandatory access control: SELinux and AppArmor

UNIX permissions and ACLs are discretionary access control (DAC): the permissions of a file are set by its owner, who can change them at any time. This has a weakness. A piece of malware running under Ali's account can also change the permissions of Ali's files, because as far as the kernel is concerned it is Ali. Mandatory access control (MAC) closes this weakness: the policy is set by the system administrator, and no process, not even root, can loosen its own restriction.

SELinux
: Gives every subject and object a security label. Rules are written with **type enforcement**: "processes of type httpd_t may read files of type httpd_sys_content_t." Very detailed and powerful, but complex. It also supports BLP-style multi-level security (MLS).

AppArmor
: Uses per-application profiles based on file paths: "/usr/sbin/nginx may read only under /var/www/." Easier to write and understand.

On Linux both systems use the kernel's **Linux Security Modules** (LSM) framework. The kernel places hooks in front of critical operations. When a process calls, for example, `open`, the usual DAC check is made first. If DAC allows, the LSM hook is called and the MAC policy is consulted (Figure 13.45). Both must say "yes" for the operation to go through. If DAC denies, MAC is never asked.

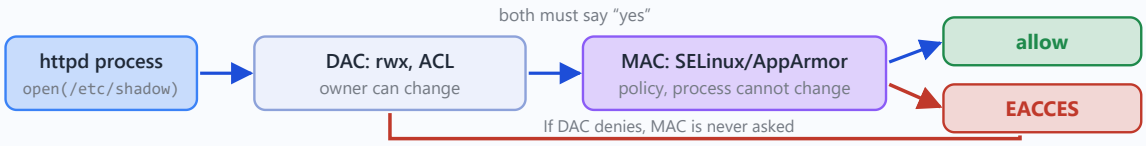


Figure 13.45: An `open()` request passes first through the DAC permission bits and then through the MAC policy in the LSM hook.

Example 13.15

A web server has been compromised through a vulnerability. The attacker can run commands with the server's user account. Two observations are made: (1) the attacker can freely change the permissions of a file belonging to the server with `chmod`; (2) he cannot access the files under `/home` even where the permission bits would allow it. Which check causes each behavior?

1. (1) is DAC. The owner of the file is the server account, and under DAC the owner, even a malicious one, can change the permissions.
2. (2) is MAC. The SELinux or AppArmor profile restricts the server process to the web files only. Even if the permission bits say “yes,” MAC says “no,” and the result is EACCES.

MAC restricts even a compromised process to the operations its profile allows. The attacker can reach only the files that profile defines. Android also runs SELinux fully in enforcing mode to separate applications from one another.

13.10 Hardware-level security

All the earlier sections assumed that the operating system works correctly. But what if the operating system itself has been modified? Or if the hardware leaks information despite all the checks of the software? This section deals with the very bottom of the chain of trust.

13.10.1 Secure boot and the TPM

When a computer is switched on, firmware, bootloader, kernel, and operating system run in turn. If an attacker can modify any link of this chain, he runs his own code before the operating system. When the operating system starts, it is this code that controls everything, and it is impossible to notice it from inside. Malware that settles in the boot process in this way is called a **bootkit**.

Definition 13.21

Secure boot means that every layer of the boot chain refuses to run the next one without verifying its digital signature. The chain rests on an immutable **root of trust** in hardware.

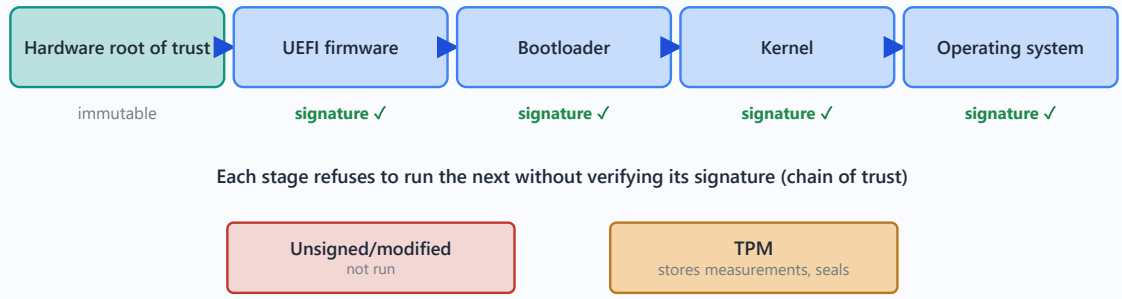


Figure 13.46: The chain of trust: each stage verifies the signature of the next. An unsigned or modified component is not run. The TPM stores the measurements.

The chain works as follows (Figure 13.46). The root of trust in hardware verifies the signature of the UEFI firmware. UEFI checks the signature of the bootloader on disk against the keys in its own database. The bootloader checks the signature of the kernel, and the kernel that of the drivers it loads. If a bootkit has modified the bootloader on disk, UEFI cannot verify the signature and does not run the bootloader.

The complement of secure boot is the **TPM** (Trusted Platform Module). The TPM is a small security chip built into the motherboard or the processor. It contains **platform configuration registers** (PCRs). In **measured boot**, the digest of every component is taken before it runs and is extended into a PCR. Extending means hashing the old value of the register together with the new digest. The final value of a PCR is therefore a fingerprint of the whole chain together with its order, and it cannot be changed retroactively.

The TPM uses these measurements in two ways. The first is **sealing**: the disk-encryption key (for example a BitLocker or LUKS key) is sealed to particular PCR values. The TPM releases the key only if the system started in the expected state. If the bootloader has been modified, the measurement comes out different and the key is not unlocked. The second is **remote attestation**: the TPM signs the measurements with its own key and sends them to a remote server. The server can verify that the machine started with trusted software.

Common mistake

Treating secure boot and measured boot as the same thing. Secure boot *does not run* an unsigned component. Measured boot runs everything, but *records* what ran and does not unlock the key on an unexpected measurement. The TPM does not repair or block anything; it only measures, stores, and releases keys accordingly.

13.10.2 Spectre, Meltdown, and side channels

In a covert channel, two processes cooperate on purpose. In a side channel, the victim is unaware of anything; the attacker measures indirect traces of its behavior.

Definition 13.22

A **side channel** is an attack path that leaks data through the indirect physical effects of computing it. These effects can be time, power consumption, cache state, electromagnetic emission, or sound.

A classic example is a function that compares a password character by character and stops at the first difference. An attempt that starts out right takes slightly longer; by measuring the times, the attacker finds the password character by character. This is the reason for constant-time comparison. The

cache is also a powerful side channel. Whether a piece of data is in the cache can be told from the time it takes to access it: from the cache it arrives in a few nanoseconds, from main memory in about a hundred.

Meltdown and **Spectre**, announced in early 2018, combined this side channel with the processors' **speculative execution** (Figure 13.47). To gain speed, today's processors execute in advance instructions whose results are not yet certain. For example, before a branch condition is computed, the instructions in the predicted direction are executed. If the prediction turns out wrong, the processor rolls back the architectural results of these instructions, that is, the register values. But the data that the instructions brought into the cache stays in the cache.

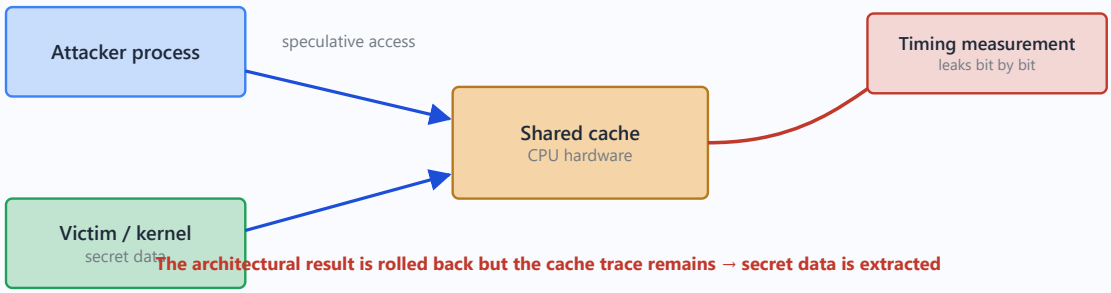


Figure 13.47: The architectural result of a speculative access is rolled back, but the trace it leaves in the shared cache remains and is read through timing measurements.

The attacker uses this as follows. He steers the processor into speculatively reading a secret byte that he has no right to access. The speculative code uses the value of this byte as an index into a large array and brings the line at that index into the cache. The processor notices the error and rolls the result back; the attacker never sees the secret byte. But he then accesses all the lines of the array and measures the times. The index of the line that comes back fast is the value of the secret byte. In this way Meltdown allowed a process in user mode to read kernel memory. Spectre made it possible to read the memory of another process, or of a protected region within the same process.

The countermeasures lie in both software and hardware. Against Meltdown, operating systems introduced **kernel page-table isolation** (KPTI): while in user mode, most of kernel memory is removed from the page table altogether. Against Spectre, microcode updates and speculation barriers added by the compiler are used. Most of these measures come at a performance price; the slowdown is noticeable in workloads heavy in system calls. These vulnerabilities also taught an important lesson: the correctness of protection mechanisms depends not only on the architectural specification but also on the implementation details of the hardware.

13.11 Summary

- The three basic goals of security are **confidentiality**, **integrity**, and **availability**; authenticity and accountability are added to them. A **threat** is a possible harm, a **vulnerability** is a weakness, an **attack** is an attempt to use it, and an **exploit** is the code that does so.
- Network attacks include eavesdropping, replay, man-in-the-middle, and session hijacking. Their remedies are, respectively, encryption, single-use values, certificate-based authentication, and secure cookies.
- Defense is built in **depth**; security claims rest on a **TCB** that should be kept small and on a **reference monitor** that checks every access.
- **Symmetric** encryption is fast but needs a shared key; **public-key** encryption solves key distribution but is slow. Real systems use both together. Security rests on the key, not on the secrecy of the algorithm (Kerckhoffs).
- A **hash function** is one-way and collision-resistant. In a **digital signature**, the digest is signed with the private key and verified with the public key. **Certificates** bind a public key to an identity. **TLS 1.3** combines ephemeral Diffie-Hellman, certificates, and authenticated encryption in a single round trip.
- In the **Diffie-Hellman** example ($p = 23$, $g = 5$, $a = 6$, $b = 15$), $A = 8$, $B = 19$, and the shared secret is 2. Square-and-multiply computes an n-bit exponent in n rounds.
- The **protection matrix** is sparse; stored by columns it gives **ACLs**, by rows **capability lists**. Revocation is easy with ACLs, delegation with capabilities. UNIX permissions are a nine-bit ACL; a file descriptor is a capability.
- By the **least privilege** principle, setuid programs drop their privilege permanently with `setresuid` as soon as their work is done. Linux divides root into about 41 **capabilities**.
- **Bell-LaPadula** protects confidentiality: no read up and no write down. **Biba** protects integrity: no read down and no write up. Neither can close **covert channels**; only their bandwidth can be reduced.
- Passwords are stored with a **salted** and **slow** hash (Argon2id): the salt makes mass attacks costly, and slowness makes each guess costly. **MFA** asks for two factors from different categories. A **TOTP** code is derived from $\mathrm{HMAC}(K, \lfloor t / 30 \rfloor)$ by dynamic truncation. A **FIDO2** signature is bound to the origin of the site and is therefore phishing-resistant.
- A **stack buffer overflow** corrupts the return address. Defenses are layered: the **canary** catches the corruption at return, **NX** blocks injected code, **ASLR** hides addresses, **CFI** and the shadow stack make ROP harder; the most fundamental solution is memory-safe languages.
- In a **TOCTOU** vulnerability the resource changes between check and use; the solution is to resolve the name once and work on the same descriptor.
- Insider attacks are logic bombs, back doors, and login spoofing. **Malware** includes viruses, worms, Trojan horses, ransomware, rootkits, spyware, and bots; polymorphic viruses defeat signature scanning.

- **Code signing** verifies the source. **Sandboxes** and **containers** limit the harm; because a container shares the kernel, it is a weaker boundary than a virtual machine. **MAC** (SELinux, AppArmor) restricts even a compromised process to its profile.
- **Secure boot** does not run unsigned components; the **TPM** stores measurements and seals keys to them. **Spectre** and **Meltdown** use the trace that speculative execution leaves in the cache as a side channel.

13.12 Key terms

confidentiality / integrity / availability (CIA) threat, vulnerability, exploit denial of service man-in-the-middle attack defense in depth trusted computing base (TCB) reference monitor encryption public-key cryptography cryptographic hash function digital signature certificate, public key infrastructure forward secrecy protection domain access matrix access control list (ACL) capability principle of least privilege discretionary / mandatory access control covert channel authentication salt multi-factor authentication buffer overflow stack canary address space layout randomization (ASLR) malware ransomware sandbox container, namespace secure boot side channel

13.13 Test Yourself

1. How is the word GIRESUN encrypted with the substitution key of Table 13.3? What does the ciphertext ZTLZ hide?

   Answer

   G → U, I → O, R → K, E → T, S → L, U → X, N → F: UOKTLXF. In the reverse direction Z → T, T → E, L → S, Z → T: TEST. Both Z's go to T; the frequencies are preserved again.

2. In Diffie-Hellman, p = 23 and g = 5. Alice chooses a = 4 and Bob chooses b = 3. What are A, B, and the shared secret?

   Answer

   A = 5^4 mod 23 = 625 mod 23 = 4. B = 5^3 mod 23 = 125 mod 23 = 10. Alice: 10^4 mod 23 = 10000 mod 23 = 18. Bob: 4^3 mod 23 = 64 mod 23 = 18. The shared secret is 18.

3. A friend says, “If Alice sends the document with an HMAC under a secret key she shares with Bob,

that is a signature too." Does this scheme provide integrity and non-repudiation?

Answer

It provides integrity: someone who does not know the key cannot modify the document and produce a valid code. It does not provide non-repudiation: since Bob also knows the key, Bob could have produced the same code. Alice can say "Bob produced this." Non-repudiation needs a private key that only Alice knows.

4. With the ACL of the code trace, what do `access(G2, R|W)` and `access(ADM, W)` return? If the decision expression were `(h & req) != 0`, would the result of the first call change?

Answer

G2's entry has h = 2; the request is 6. 2 & 6 = 2 ≠ 6, so deny (0). For ADM, h = 7, 7 & 2 = 2, so allow (1). With the faulty expression, 2 & 6 = 2 ≠ 0 and G2 would be allowed although it has no read right.

5. A file has mode 754. Can group members modify the file? With umask 027, what modes do a new file and a new directory get?

Answer

754 = rwxr-xr--. The group's right is r-x; there is no write, so they cannot modify it. With umask 027 a file becomes 666 → 640 and a directory 777 → 750.

6. For a process at the Secret level, which operation does BLP allow: reading an Unclassified announcement, or writing to a Confidential file? If the same process were at the Medium level in Biba, could it read an Internet file at the Low level?

Answer

In BLP, reading down is free; reading the Unclassified announcement is allowed. Writing to a Confidential file is a write down and is forbidden. In Biba, reading down is forbidden; a process at the Medium level cannot read the Low file.

7. In a covert channel using file locking, the interval is reduced to 5 ms. What is the speed of the channel? How long does it take to leak a 2048-bit RSA private key?

Answer

1000 / 5 = 200 intervals per second, that is 200 bit/s. 2048 / 200 = 10.24 seconds.

8. Passwords consist of 12 lowercase letters and are stored with plain SHA-256. At 10^{10} guesses per second, how long does it take in the worst case to find a single user's password? Does adding a salt change this time?

   Answer

   $26^{12} \approx 9.54 \times 10^{16}$; divided by 10^{10} this is $\approx 9.54 \times 10^{6}$ seconds, about 110 days. A salt does not change the time for a single user; when all users are targeted, it multiplies the work by the number of users.

9. At t = 1,000,000,000, what is the TOTP counter T? What 30-second window corresponds to this code?

   Answer

   1,000,000,000 / 30 = 33,333,333.3; T = 33,333,333. The window runs from T · 30 = 999,999,990 to 1,000,000,019.

10. On a 64-bit machine, a function without a canary copies a 30-character string with `strcpy` into the array `char name[16]`. How many bytes overflow? How many bytes of the saved rbp and of the return address are corrupted?

    Answer

    With the terminator, 31 bytes are written; 31 − 16 = 15 bytes overflow. The first 8 corrupt the whole saved rbp, and the remaining 7 (including the last character and the `\0`) corrupt 7 bytes of the return address.

13.14 Further reading

- J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems”, 1975: the classic paper that first lists the principles of least privilege, fail-safe defaults, and complete mediation together. It is still relevant today.
- R. Anderson, *Security Engineering*, 3rd ed.: treats security as an economic and human problem as much as a technical one. Its chapters on passwords, access control, and formal models contain real-world incident examples.
- Aleph One, “Smashing the Stack for Fun and Profit”, Phrack 49, 1996: the historic text that shows the exploitation of a stack buffer overflow step by step. It helps one understand why today's defenses are needed.
- P. Kocher et al., “Spectre Attacks: Exploiting Speculative Execution”, and M. Lipp et al., “Meltdown”, 2018: explain from the original source how speculative execution combines with the cache side channel.

13.15 References

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 9: Security.
2. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 16: Security; Chapter 17: Protection.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — the security chapters.
4. R. Anderson, *Security Engineering*, 3rd ed., Wiley, 2020.
5. J. H. Saltzer, M. D. Schroeder, “The Protection of Information in Computer Systems”, 1975.
6. D. E. Bell, L. J. LaPadula (1973); K. J. Biba (1977) — formal security models. B. W. Lampson, “A Note on the Confinement Problem”, 1973.
7. RFC 8446 (TLS 1.3), RFC 6238 (TOTP), RFC 9106 (Argon2); FIDO2/WebAuthn (W3C); NIST FIPS 203 (ML-KEM).
8. Aleph One, “Smashing the Stack for Fun and Profit”, Phrack 49, 1996; Kocher et al., “Spectre”; Lipp et al., “Meltdown”, 2018.

Operating-System Design

Taken one at a time, processes, memory, file systems, and I/O each look like a problem solved by its own algorithms. A team designing an operating system from scratch faces questions of a different kind. Which abstractions should it offer? How many system calls should there be, and how general should each be? Should drivers run inside the kernel or outside it? At what moment should a name be bound to an object? Where should code be made fast, and where left simple? These questions have no definitive answers; what exists is a body of principles and trade-offs accumulated through experience. The notes first describe what makes design hard and the four basic goals of an operating system. They then turn to interface design, the options for system structure, and design principles such as the separation of mechanism and policy, orthogonality, naming, and binding time. Indirection, reentrancy, the space-time trade-off, caching, hints, and optimizing the common case are traced step by step on short C programs. The last sections cover the management of large software projects, how kernel code is tested and debugged, and current trends in operating systems. All traces and numbers were recomputed in Python and checked.

LEARNING OUTCOMES

Challenges and goals

Explain what makes operating-system design hard and state the four basic goals of an operating system.

Interface design

Discuss the principles of simplicity, completeness, and efficiency, and the execution and data paradigms, using system-call examples.

System structure

Evaluate the structural options through the isolation–performance trade-off; discuss layer ordering, safe extensibility, and kernel threads as design decisions.

Design principles
Apply the separation of mechanism and policy, orthogonality, two-level naming, binding time, and static versus dynamic structures.

Techniques and performance
Use indirection, reentrancy, the space-time trade-off, caching, hints, and optimization of the common case on concrete examples.

Projects, testing, and trends
Interpret Brooks's law; choose kernel testing and debugging tools; assess the trends of virtualization, many cores, Rust, eBPF, and unikernels.

14.1 Design challenges and goals

A team writing a compiler or a database knows fairly well what it is building. The compiler's input is fixed by a language definition and its output by an instruction set. The job of an operating system is not drawn so sharply. Which programs will run on it, on what hardware, and for how many years is not known in advance. Designing an operating system is therefore less a matter of choosing an algorithm than of balancing demands that conflict with one another.

14.1.1 Why is designing an operating system hard?

Tanenbaum lists eight factors that make operating-system design harder than other software projects (Table 14.1). Some of them stem simply from size, others from the operating system's position: it sits between the hardware and all the applications and must survive changes in both.

Factor	Description
1. Sheer size	The Linux kernel is about 40 million lines; no single person understands all of it.
2. Concurrency	Devices, interrupts, and threads run at the same time; race conditions and deadlocks arise.
3. Hostile users	Some users want to break the system or steal data; every input is suspect.
4. Sharing despite isolation	Users must be kept apart, yet able to share the information they choose.
5. Long life	An operating system lives 20–30 years; future hardware and applications must be anticipated.
6. Generality	How the system will be used is unknown: anywhere from a phone to a supercomputer.
7. Portability	It must run on different processors and platforms: x86-64, ARM64, RISC-V.
8. Backward compatibility	Old programs must keep working; new ideas are limited by old constraints.

Table 14.1: Tanenbaum's eight factors that make operating-system design difficult.

Size alone is a serious problem. Most of the forty million lines are driver code, but the rest is still far more than one person can keep in mind. In such a system it is hard to predict what a change will affect. Well-designed interfaces and a structure that limits dependencies are therefore as important as the code itself.

Concurrency is a difficulty peculiar to operating systems. An application programmer can often think in terms of a single thread. The kernel, by contrast, is concurrent from the start: it deals with interrupts, with several cores, and with system calls arriving at the same moment. A bug may show up only when a particular timing occurs, and reproducing it can take days.

Hostile users and sharing despite isolation are two demands that pull against each other. The system must protect every user from the others, yet users also want to share files and memory in order to work together. Every door opened for sharing is also an attack surface.

The last four factors concern time. A long life requires the system to adapt to hardware that does not yet exist. When UNIX was designed in the 1970s, there were neither multicore processors nor SSDs. Generality prevents the designer from optimizing for a single way of using the system. Portability forces hardware-dependent code to be confined. Backward compatibility makes every published interface permanent. The need for long maintenance and repair is a natural consequence of a long life and the obligation to stay compatible.

> **Common mistake**
>
> Reducing all the difficulties to the problem of “large software.” Size is present in every big project. What sets an operating system apart is that concurrency, hostile users, and backward compatibility come together in the same code. A bug in an application brings down that application; a bug in the kernel brings down everything running on top of it.

14.1.2 The four basic goals

Faced with these difficulties, we must state clearly what an operating system is for. Tanenbaum names four basic goals. They can be pictured as layers: applications on top, hardware at the bottom, and the operating system in between (Figure 14.1).

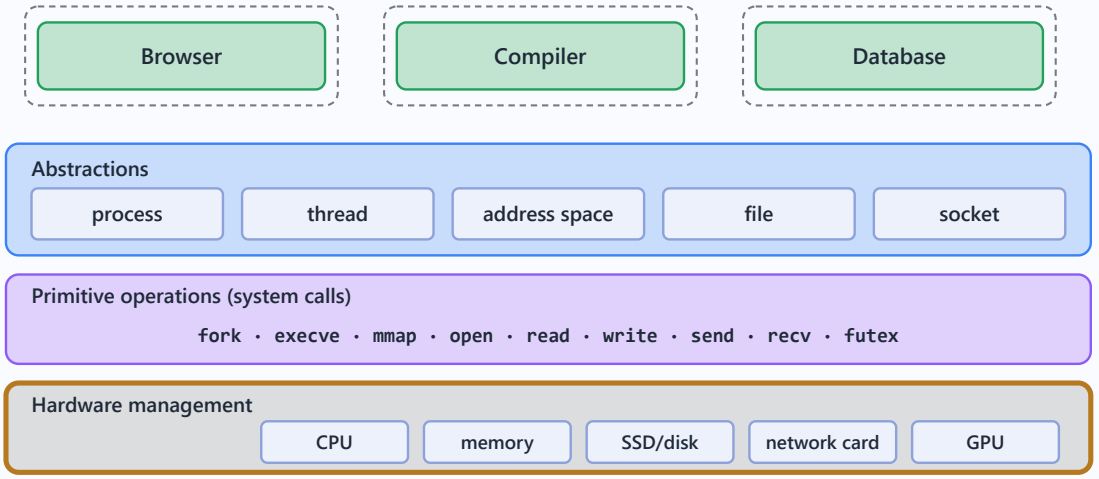


Figure 14.1: The four goals of an operating system. Applications (browser, compiler, database) use abstractions; the primitive operations on the abstractions are system calls; isolation keeps each process in its own address space; at the bottom, the hardware is managed.

Definition 14.1

An **abstraction** is a concept that hides the complexity of the hardware and gives the programmer a more convenient model: process, thread, address space, file, socket. A **primitive** is a basic operation on an abstraction; in an operating system, the primitives are the system calls.

1. **Defining abstractions.** A disk is really an array of numbered blocks. A file, on the other hand, is an object with a name, a size, and permissions, read byte by byte. The operating system closes the gap between the two. The hardest part of design is finding the right abstraction. One that is too detailed does not hide the hardware enough; one that is too coarse is too narrow for some applications.
2. **Providing primitives.** An abstraction is useful only when operations can be performed on it. For files these operations are `open`, `read`, `write`, and `close`; for processes, `fork` and `execve`; for address spaces, `mmap`; for synchronization, `futex`.
3. **Providing isolation.** Users and processes must not interfere with one another. Each process runs in its own address space; if one crashes, the others are unaffected. Even so, controlled ways of sharing, such as file permissions and shared memory, must remain open.
4. **Managing the hardware.** Interrupts, timers, the MMU, and device controllers are the subject of this goal. Most low-level work is done in this layer, and all the goals above rest on it.

The four goals are not independent of one another. Isolation can be achieved only through hardware management, that is, through the MMU and privilege levels. Abstractions without primitives are mere names. The rest of these notes is largely an answer to the question: with what structure and what principles, and at what cost, are these goals realized?

Common mistake

Counting the compilation of programs among the goals of an operating system. Compilation is the job of a toolchain, that is, of the compiler. The operating system loads the compiled program into memory, creates a process for it, and runs it. The compiler is itself an ordinary application running on top of the operating system.

14.2 Interface design

The face an operating system shows to the outside world is its system-call interface. Applications are written against this interface, and once it has been published it can hardly be changed in practice. How the interface is designed is therefore a longer-lived decision than how the inside of the kernel is written.

14.2.1 Guiding principles

There is no precise recipe for a good interface, but three principles guide the designer: simplicity, completeness, and efficiency.

Definition 14.2

Simplicity: the interface should consist of a small number of well-thought-out concepts. **Completeness**: the interface should be able to do everything users need. **Efficiency**: every feature should be implementable efficiently, and the programmer should be able to sense the approximate cost of a call.

The saying most often cited for simplicity is Antoine de Saint-Exupéry's: perfection is reached not when there is nothing left to add, but when there is nothing left to take away. A small number of concepts is easy to learn, implement, and verify. Every new concept also brings its interactions with all the other concepts.

Completeness is the counterweight to simplicity. A remark attributed to Einstein sums it up: everything should be as simple as possible, but no simpler. If an interface is so spare that it cannot do a needed job, applications look for ways around it, and the result is more complicated. The ideal is to get a lot done with few mechanisms. The MINIX 3 kernel offers only a handful of primitives for interprocess communication: `send`, `receive`, `sendrec`, and `notify`. The file system, the drivers, and memory management are built on these few primitives.

The efficiency principle says two things. First, a feature that cannot be implemented efficiently probably is not worth adding. Second, the programmer should be able to sense the approximate cost of a call. `lseek` merely changes the file offset and is cheap; `read` may need a disk access and can be

expensive. If a call that looks cheap is secretly expensive, the programmer makes wrong decisions.

A practical rule follows from these three principles: every system call should do one thing well. Adding a feature is easy; removing one is almost impossible. Because of backward compatibility, a wrong decision is carried for years.

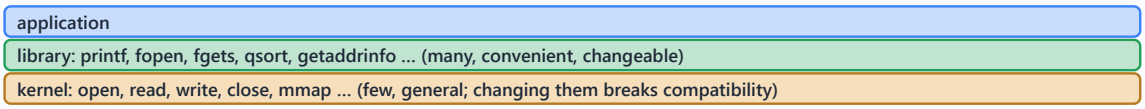


Figure 14.2: Convenience functions (`printf` , `fopen` , `qsort` , `getaddrinfo`) live in the library; they are numerous and may change. The calls in the kernel (`open` , `read` , `write` , `mmap`) are few and general; changing them breaks compatibility.

Example 14.1

Can we give an example from the UNIX interface for each principle?

1. **Simplicity.** In UNIX, files, devices, and pipes are all used with the same four calls: `open` , `read` , `write` , `close` . The programmer learns a single model.
2. **Completeness.** There are six different `exec` library functions, but all of them reduce to a single `execve` call. Because `execve` takes a path, an argument array, and an environment array, it meets every need.
3. **Efficiency.** `lseek` only changes a number, whereas `read` may read from disk. The difference in cost between the two calls can be sensed from their names and definitions.

14.2.2 Small is beautiful

The most concrete measure of the simplicity principle is the number of system calls. A few general calls are better than many special ones. Real systems differ greatly on this measure (Figure 14.3).

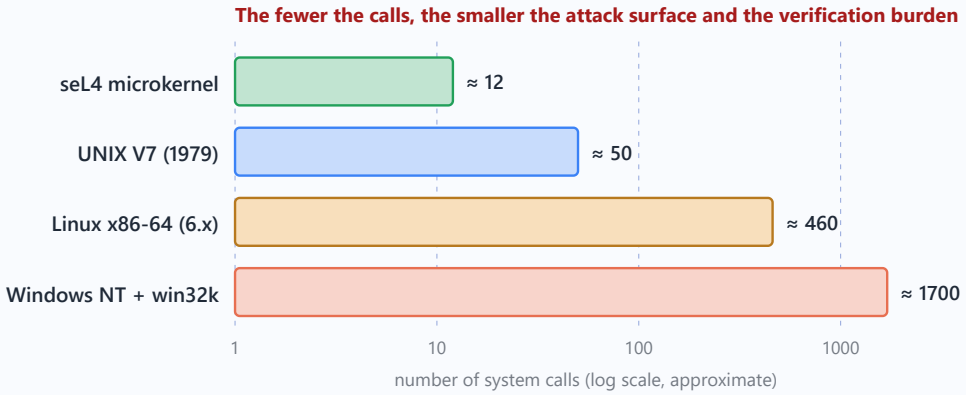


Figure 14.3: Approximate numbers of system calls, on a logarithmic scale: seL4 ≈ 12, UNIX V7 (1979) ≈ 50, Linux x86-64 (6.x) ≈ 460, Windows NT and win32k together ≈ 1700.

Because the scale is logarithmic, the differences between the bars can deceive the eye. Windows has about 140 times as many calls as seL4, and Linux has more than nine times as many as UNIX V7. As the number of calls grows, so does the attack surface: every call is a door through which malicious input can reach the kernel.

The seL4 microkernel is the extreme example of this principle. Its kernel is about 10 thousand lines and has been formally verified: it has been proved that the C code conforms to a mathematical specification. For a kernel of millions of lines, such a proof is not possible with today's methods.

In Linux, by contrast, the number of calls grows by a few every year. The recently added `io_uring_*`, `landlock_*`, and `mseal` calls are examples. Windows follows a different path: the documented interface is not the kernel calls but the Win32 API. The kernel calls may change from release to release, because applications do not use them directly. In this arrangement, the library layer in between preserves compatibility.

Common mistake

Taking a small interface to mean a “system with few features.” The fifty or so calls of UNIX V7 were enough to build a rich programming environment with pipes, redirection, and process creation. The point of smallness is not to cut features but to obtain them from the combination of a few general mechanisms.

14.2.3 User-interface paradigms

The user interface is often not part of the kernel. Even so, the paradigm chosen affects the design of the whole system. Four paradigms live side by side today.

- **Command line** (CLI, command-line interface). The shell, text commands, and pipes. It remains indispensable for automation and remote administration.

- **WIMP / GUI.** Windows, icons, menus, and a pointer (windows, icons, menus, pointer). The meaning of the mouse buttons should be the same in every application.
- **Touch and mobile.** Gestures such as swiping and pinching, full-screen applications, a notification center.
- **Voice, chat, and agents.** Voice assistants and natural-language interfaces open a new entry point to operating-system services.

The effect of the paradigm on the kernel is clearest in the graphical interface. A GUI requires an event-driven execution model, a fast window system, and a consistent application framework. A click follows a path from the device to the application (Figure 14.4).

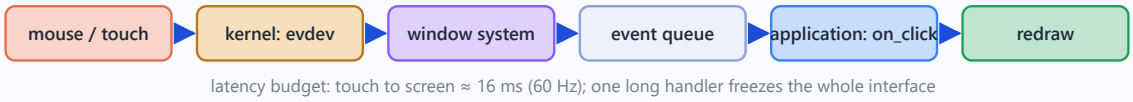


Figure 14.4: The path of a click: the input device, the evdev driver in the kernel, the window system, the application's event queue, and the `on_click` handler; finally the screen is redrawn. On a 60 Hz display, the budget from touch to screen is about 16 ms.

The 16 ms budget comes from the calculation 1000 ms / 60 ≈ 16.7 ms. Within this time the event must pass from the kernel to the window system and then to the application, be processed, and a new frame must be drawn. A single long-running handler freezes the whole interface. The principle common to all the paradigms is consistency: the same action should give the same result everywhere.

14.2.4 Execution paradigms

Who determines the flow of a program also determines what it will ask of the operating system. There are two basic paradigms.

Definition 14.3

In **algorithmic code**, the program determines its own flow and calls the operating system only to obtain services. In **event-driven code**, external events determine the flow: a key, a click, a network packet, a timer. The program waits in a loop for the next event and calls the matching handler.

The typical form of algorithmic code is similar to the one below. The program starts, does things in order, reads and writes when needed, and finally ends. Compilers, sorting programs, and batch jobs have this structure.

```
int main(void)
{
    int x, y;
    init();
    do_something();
    read(fd, &x, sizeof x);
    do_something_else();
    write(fd, &y, sizeof y);
    keep_going();
    exit(0);
}
```

Event-driven code, on the other hand, is built around a loop. `get_message` fetches the next event, and `switch` routes it to the relevant code according to its type. GUIs, network servers, and microkernel servers have this structure.

```
int main(void)
{
    msg_t msg;
    init();
    while (get_message(&msg)) {
        switch (msg.type) {
        case 1: ... ; break;
        case 2: ... ; break;
        case 3: ... ; break;
        }
    }
}
```

The choice of paradigm also shapes the system calls. The algorithmic model wants a blocking `read`: wait until data arrives, then continue from where the flow left off. The event-driven model wants a call of the "give me the next event" kind: `get_message`, `epoll_wait`, or `io_uring_enter`. An operating system must support both paradigms.

14.2.5 Code trace: the event loop

How an event loop works can be followed on concrete code. The program below has three kinds of events: key (`KEY`), click (`CLICK`), and timer (`TIMER`). Each goes to its own handler.

```
int main(void) {
    msg_t m;
    init();
    while (get_message(&m)) {
        switch (m.type) {
        case KEY:   on_key(&m);   break;
        case CLICK: on_click(&m); break;
        case TIMER: on_timer(&m); break;
        }
    }
    return 0;
}
```

`init()` prepares the window and the resources. The `while` condition is the return value of the call `get_message(&m)`. The call returns 1 if it delivers an event, and 0 on the `QUIT` event, which signals that the program should shut down. The `switch` calls the handler `on_key`, `on_click`, or `on_timer` according to the event type. When the handler finishes, the loop returns to the top. The trace takes eight steps (Table 14.2).

Step	Queue	get_message	Code running
1	empty	not called yet	`init()`
2	CLICK, KEY, TIMER	waiting for an event	the window system adds the events
3	KEY, TIMER	returned CLICK	`on_click`
4	TIMER	returned KEY	`on_key`
5	empty	returned TIMER	`on_timer`
6	empty	blocked	the process sleeps
7	QUIT	waiting for an event	the process wakes up
8	empty	returned 0	the loop ends, `return 0`

Table 14.2: Trace of the event loop. CLICK, KEY, and TIMER arrive in the queue in that order, and QUIT comes last.

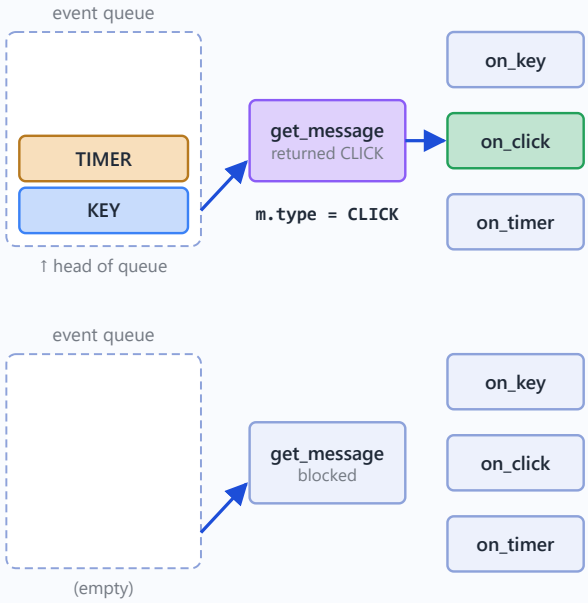


Figure 14.5: (a) `get_message` has returned the CLICK at the head of the queue, and the `switch` has routed it to `on_click`. (b) When the queue empties, `get_message` blocks; the process sleeps and uses no CPU.

Two points of the trace matter. The first is step six. When the queue is empty, `get_message` does not return immediately; it puts the process to sleep. The process uses no CPU meanwhile; when a new event arrives, the kernel wakes it up. The loop is not a "busy wait" but a structure built around a blocking call.

The second is the duration of the handlers. The loop is single-threaded; while one handler runs, the other events in the queue wait. `on_timer` does a short job such as blinking the cursor. But if a handler sets about reading a large file, the interface freezes for that time. Long jobs are therefore handed to a separate thread. The same pattern is used with `epoll` in servers such as nginx and Node.js, and with message queues in microkernel servers.

Example 14.2

Let the handlers keep a counter: `KEY` increases the counter by 1, `CLICK` by 10; `TIMER` resets the counter if it is greater than 10. If the events arrive in the order KEY CLICK KEY TIMER CLICK KEY KEY QUIT, what is the counter at the end of the loop?

1. **KEY, CLICK, KEY.** 0 → 1 → 11 → 12.
2. **TIMER.** Because 12 > 10, the counter is reset: 0.
3. **CLICK, KEY, KEY.** 0 → 10 → 11 → 12.
4. **QUIT.** `get_message` returns 0 and the loop ends. The result is 12.

The result is determined not by the text of the program but by the order of the events. Had the same events arrived in a different order, the counter would have come out differently.

Common mistake

Thinking that an event loop keeps the CPU constantly busy when the queue is empty. `get_message` is a blocking call; if there is no event, the process goes to sleep. The problem is not the loop itself but long-running handlers.

14.2.6 Data paradigms

The execution paradigm determines the flow; the data paradigm determines how resources are named and used. There are two great traditions (Figure 14.6).

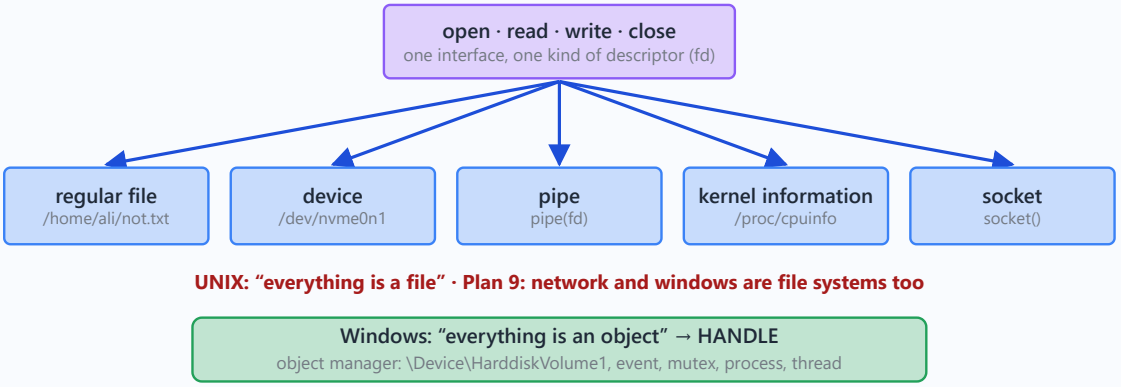


Figure 14.6: Data paradigms. In UNIX a single file interface (`open` , `read` , `write` , `close`) applies to a regular file, a device, a pipe, `/proc` information, and a socket. In Windows everything is an object and is used through a HANDLE.

The UNIX principle is summed up as “everything is a file.” The regular file `/home/ali/not.txt` , the NVMe disk `/dev/nvme0n1` , a pipe, the kernel information `/proc/cpuinfo` , and a socket are all used through the same kind of descriptor, the file descriptor (fd). Plan 9 pushed this idea further, offering the network and windows as file systems too. In Windows, everything is an object. The object manager keeps files, events, mutexes, processes, and threads in a single namespace under names such as `\Device\HarddiskVolume1` ; a program reaches them through a HANDLE.

The gain from a common data paradigm is that the same tools work on every resource. `cat` , `dd` , and shell redirection work on a file, on a device, and on kernel information alike. The command `cat /proc/loadavg` shows the system's load average with no special tool needed.

There is a price as well. Some devices do not fit the file model. Setting a camera's resolution or changing a network card's settings cannot be expressed naturally with `read` and `write` . Everything then piles up in “escape hatches” such as `ioctl` . `ioctl` takes a command number whose meaning changes from device to device, and hundreds of different operations hide behind a single call. The complexity has been lifted off the surface of the interface, but it has not been lost inside.

14.2.7 The system-call interface

The most concrete application of the interface principles is the relationship between system calls and the library. The kernel holds few, general calls, and the library holds convenience functions (Figure 14.7).

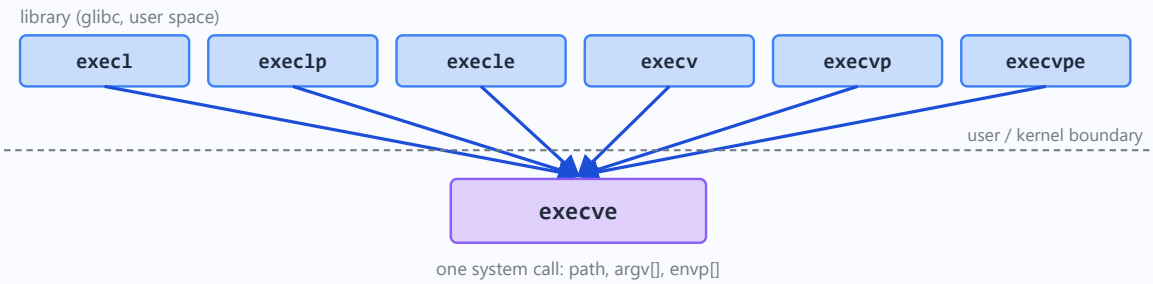


Figure 14.7: The six `exec` library functions (`execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe`) live in user space; all of them reduce to the single `execve` system call.

The six functions differ in how the arguments are given (a list or an array), whether the program is searched for in `PATH`, and whether the environment is passed explicitly. All of these differences are resolved in user space. The kernel sees only a path, an argument array, and an environment array. Adding a new convenience does not change the kernel; only the library grows.

The second principle is independent components. UNIX splits process creation into two calls: `fork` and `execve`. In between, the child process can prepare its own environment: it can redirect input and output, change its working directory, and close unneeded files. In Windows, by contrast, `CreateProcess` takes ten parameters in a single call. That call is powerful, but every new need means a new parameter or flag.

Modern trends move in two directions. `posix_spawn` creates a process in a single step, to avoid the cost of copying the entire address space with `fork`. `io_uring` is a batched-call interface: many requests are submitted in a single user–kernel transition. Both trends arise from the principle that cost should be predictable.

> **Common mistake**
>
> Taking `printf` or `fopen` to be a system call. They are C library functions; when necessary, they make a `write` or `openat` call inside. `printf` often just writes to a buffer in user space and never enters the kernel at all.

14.3 System structure

The interface is the face seen from outside; the structure describes how the inside of the kernel is divided. Which code runs in which address space and with what privilege? How does one component call another? The answers to these questions determine both the speed of the system and its resilience to faults.

14.3.1 Structural options: isolation versus speed

The structural options are usually listed under names such as "monolithic," "layered," and "microkernel." Behind each name is really a design decision. Every decision gains something and costs something (Table 14.3).

Design decision	Gained	Paid	Example
Calling everything directly in a single address space	Speed: one trap, direct function calls	No fault isolation: one driver bug brings the system down	Linux, FreeBSD
Restricting dependencies to one direction (layers)	Layer-by-layer testing; changes stay local	Choosing the layer order; the cost of every crossing	THE, MULTICS
Putting services in separate address spaces	A crashed server is restarted; small, verifiable kernel	IPC, context switching, and copying on every request	MINIX 3, QNX, seL4, Zircon
Separating protection from management	Application-specific abstractions (libOS)	Sharing between applications is hard; libOS complexity	Aegis/ExOS, unikernels
Loading code at run time	Adaptation without recompiling	Safety of the extension: needs a verifier or a safe language	SPIN, Linux eBPF
Compromising the pure model	Microkernel layout, monolithic speed	Part of the isolation is lost	Windows NT, macOS XNU

Table 14.3: System-structure options as design decisions.

Every row of the table is a different point on the same trade-off. The basic trade-off is between isolation and performance. Each time an isolation boundary is crossed, a price is paid: a trap, a context switch, or a data copy. The more boundaries there are, the better faults are contained, but the slower the system becomes.

Definition 14.4

In a **monolithic kernel**, all services, including the file system, drivers, and memory management, run in a single privileged address space and call one another directly. In a **microkernel**, the kernel does only the most basic jobs, such as IPC, scheduling, and memory mapping. The other services run in user space as separate server processes and communicate by messages.

The difference between the two structures is plain in a single file read (Figure 14.8). In a monolithic kernel, the application enters the kernel with one trap; the VFS, ext4, and the disk driver call one another directly as function calls. In a microkernel, the application's request goes as a message to the file server, and from there by another message to the disk driver; the reply returns along the same path. Each message means at least one kernel crossing and a context switch.

Figure 14.8: On the left, a monolithic kernel: one address space, one trap, and direct calls. On the right, a microkernel: the application, the file server, and the disk driver are separate processes; the kernel does only IPC, scheduling, and memory mapping.

Example 14.3

A team is considering taking the network driver out of the kernel and running it as a separate server process in user space. When the driver has a NULL-pointer bug, what happens in the two designs, and what is the cost of the user-space design?

1. **Monolithic kernel.** The driver is in the kernel address space. The faulty access is a kernel fault (an oops); most of the time the system is left unstable or crashes entirely.
2. **User-space driver.** The fault kills only the driver process. In MINIX 3, the reincarnation server notices this and restarts the driver; the other processes keep running.
3. **Cost.** Every packet is now carried between the kernel and the driver process by messages. Each message adds a context switch and often a data copy; the packet rate drops.

The options are not mutually exclusive. Linux is monolithic, but it is extensible while running through loadable modules and eBPF; it also uses kernel threads. Windows NT and macOS XNU started with microkernel ideas but, for performance, moved most services back into the kernel address space. These hybrid structures show that both pure models have had to compromise in practice.

Common mistake

Assuming that a microkernel is slower in every case. In modern microkernels such as L4 and seL4, IPC has been brought down to a few hundred cycles. The real cost often lies not in a single message but in the number of messages a request needs and in how many times the data is copied.

14.3.2 Layered systems

A layered structure is the oldest way of restricting dependencies to one direction. Dijkstra's THE system (1968) was built on this idea. Each layer uses only the abstractions offered by the layer below it and knows nothing of the one above. Tanenbaum proposes a seven-layer design for a modern operating system (Figure 14.9).

Definition 14.5

In a **layered system**, the system is divided into ordered layers, each of which uses only the services of the layer below it. A layer can be tested on its own if the layers beneath it work correctly.

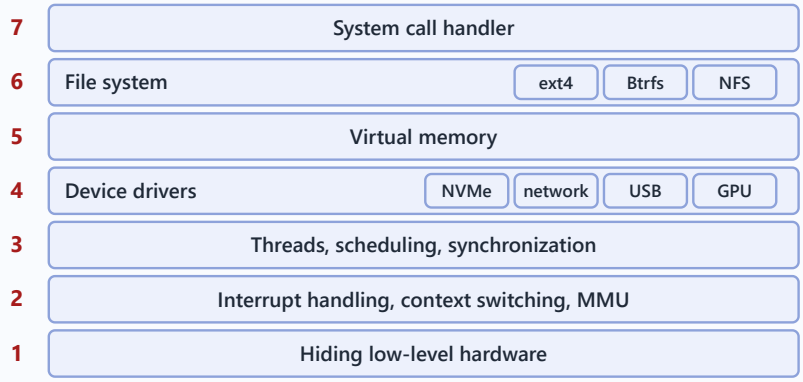


Figure 14.9: Tanenbaum's seven-layer design. 1: hiding the low-level hardware; 2: interrupt handling, context switching, MMU; 3: threads, scheduling, synchronization; 4: device drivers; 5: virtual memory; 6: file systems; 7: system-call handler.

The order of the layers is not arbitrary; it follows from the dependencies. Virtual memory sits above the device drivers because it writes pages to disk and reads them back. The file system sits above virtual memory because it allocates frames for the page cache. Threads sit above layer 2 because they depend on interrupt handling and context switching. Layer 1, which hides the hardware, provides portability: when porting to a new processor, this layer is rewritten first.

The passage of a `read` call through the layers shows the structure in action.

1. **Layer 7.** The application calls `read(fd, buf, n)` and the trap arrives at the system-call handler. The parameters are checked here: is `fd` valid, and is the buffer in user space?
2. **Layer 6.** The file system (for example ext4) translates the file offset to a disk block; the block to be read is 4711. If the block is not in the cache, it is requested from the lower layers.
3. **Layer 5.** Virtual memory allocates the page-cache frame where the data will be placed and validates the mapping of the user buffer.
4. **Layer 4.** The NVMe driver writes the read command into the device's queue.

5. **Layer 3.** The calling thread blocks until the I/O finishes; the scheduler runs another thread.
6. **Layer 2.** When the device finishes, it raises an interrupt. The interrupt handler makes the sleeping thread ready. The result travels back up through the layers, and `read` returns the number of bytes read.

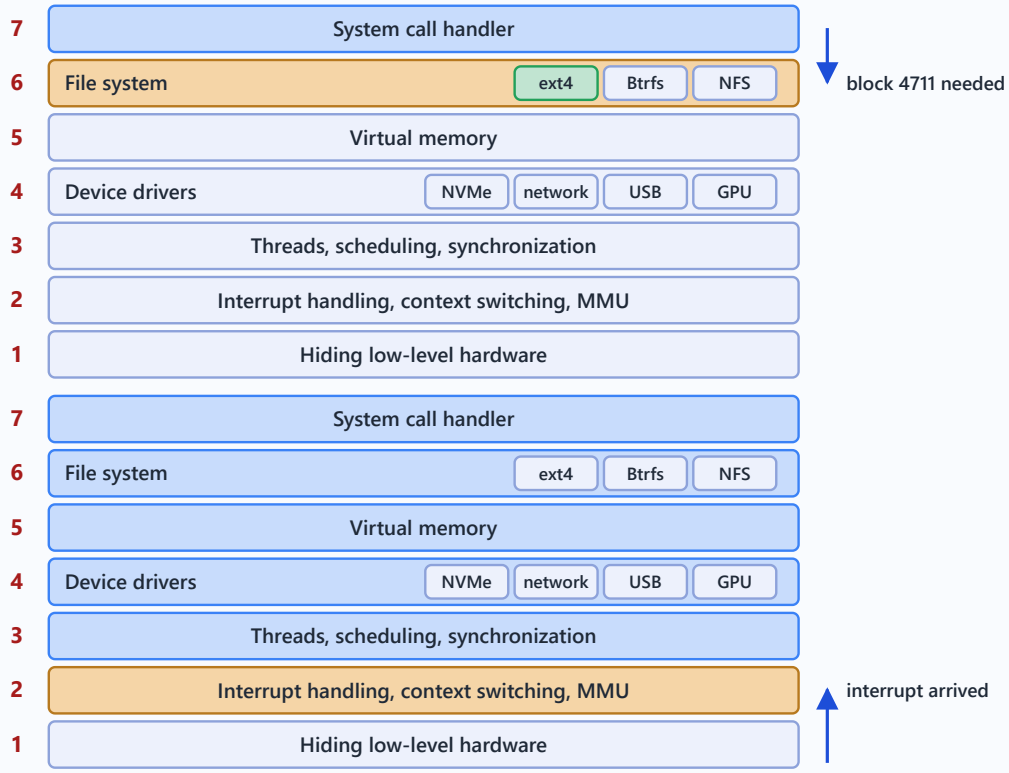


Figure 14.10: The path of the `read` call. (a) In the file-system layer, the offset is translated to block 4711. (b) When the device finishes, the interrupt arrives at layer 2 and the result returns upward.

The price of a layered structure is the cost of crossing each layer. In a purely layered system, a request must pass through every layer one by one, and each crossing adds a function call and sometimes a copy of the parameters. The real difficulty, though, is choosing the layer order. In real systems, dependencies are not always one-directional. Virtual memory uses drivers, but a driver may also ask for memory for its own buffers. Today's systems therefore use the idea of layers as a thinking tool, not as a rigid rule.

Common mistake

Reading the layer number as an order of importance. Layer 7 is not “more important” than layer 1; it merely depends on more layers below it. The order shows dependency: a layer works only if everything beneath it is ready.

14.3.3 Extensible systems

As needs change, new capabilities must be added to the kernel. Recompiling the kernel and rebooting the machine every time is impractical. In extensible systems, the kernel provides the basic functions and new components are added while it runs.

> **Definition 14.6**
>
> An **extensible system** is one that allows new code to be loaded into the kernel while it runs. **eBPF** (extended Berkeley Packet Filter) is the mechanism in Linux that lets small, verified programs be attached to kernel hooks and run.

The biggest problem of extension is safety. When third parties can add features to the system, it must also be ensured that the added code does not break the kernel. A classic loadable module (`insmod`) has full privileges: a bug in the module is no different from a bug in the kernel itself and can crash the system. The SPIN system of 1995 tried to solve this problem with a safe language, having extensions written in Modula-3 so that memory errors were prevented at compile time.

eBPF in Linux is today's answer (Figure 14.11). The program is written in C and compiled to BPF bytecode with `clang`. User space hands this bytecode to the kernel with the `bpf()` system call. The **verifier** in the kernel examines every path of the program. Are the loops bounded, and is every memory access safe? If the program fails a condition, loading is rejected. A program that passes is translated to native machine code by the JIT compiler and attached to a hook.

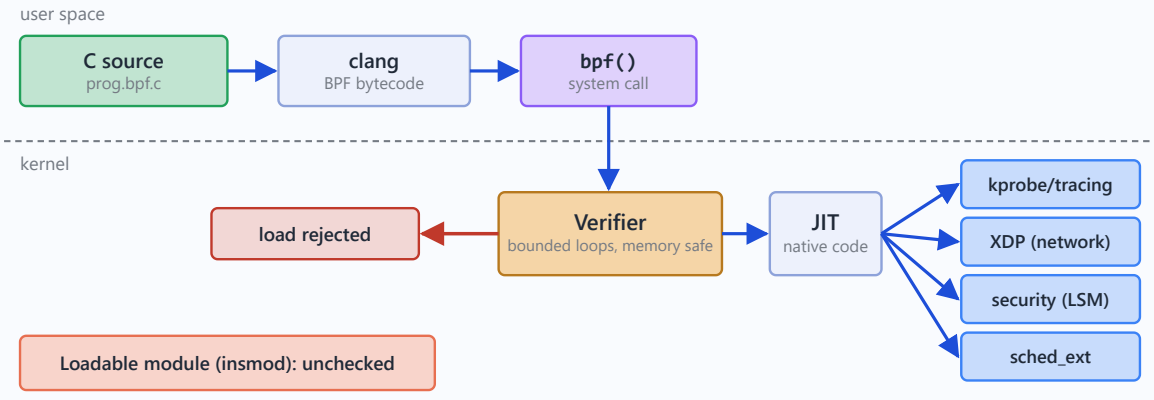


Figure 14.11: The path of an eBPF program: the C source is compiled to bytecode with `clang`, reaches the kernel through the `bpf()` call, passes the verifier, is translated to native code by the JIT, and is attached to hooks (tracing, XDP, LSM, sched_ext). A loadable module, in contrast, is unchecked.

Today eBPF hooks are used in four areas: tracing (kprobes and tracepoints), networking (processing packets at the driver level with XDP), security (LSM hooks), and, since Linux 6.12, scheduling (sched_ext). The importance of eBPF is that it provides isolation and extensibility at the same time.

The code is fast because it runs in the kernel address space, and thanks to the verifier it cannot crash the kernel.

14.3.4 Kernel threads

Some of the kernel's work is not tied to a system call or an interrupt. Reclaiming pages when memory runs low, writing the journal to disk, and finishing jobs too long to do in an interrupt are examples. For these jobs the kernel uses its own threads (Figure 14.12).

> **Definition 14.7**
>
> A **kernel thread** is a thread created and managed by the kernel that runs only in the kernel address space. It has no user address space. In Linux it is created with `kthread_create`.

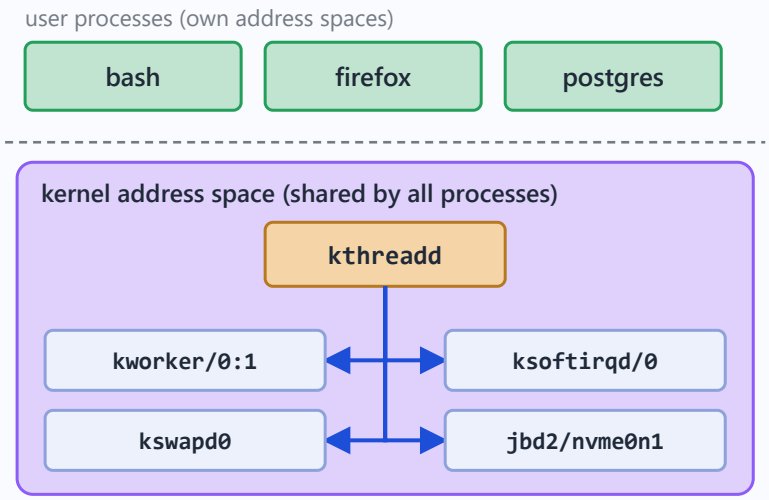


Figure 14.12: Each of the user processes (bash, firefox, postgres) has its own address space. The kernel threads (kthreadd, kworker, ksoftirqd, kswapd, jbd2) run in the kernel address space, which is common to all processes.

Kernel threads are scheduled like ordinary threads and can block and sleep. Sleeping in an interrupt handler is forbidden; long jobs that cannot be done in an interrupt are therefore deferred to a kernel thread. Context switching is also cheap, because the address space, that is, the page table, does not change. Typical examples in Linux are the following:

- `kworker`: runs deferred work (workqueue).
- `ksoftirqd`: handles soft interrupts, such as the processing of network packets.
- `kswapd`: reclaims pages when free memory runs low.
- `jbd2`: writes ext4's journal to disk.

The ancestor of them all is `kthreadd`. In the output of `ps -ef` they appear in square brackets, for example `[kswapd0]`. User processes cannot create kernel threads. System calls can hand work only to threads the kernel is already running; the worker pool of `io_uring` is an example. Kernel threads make the structure one step more flexible: even a monolithic kernel can divide its work into independent flows that are scheduled separately.

14.4 Design principles

Once the structure is chosen, the designer meets hundreds of small decisions. A few principles prove useful in these decisions again and again. They are not a single algorithm but a way of thinking; the same principle turns up in memory management and in file systems alike.

14.4.1 Separating mechanism from policy

Definition 14.8

A **mechanism** determines how something is done; a **policy** decides what is to be done. The mechanism stays fixed in the kernel; the policy can be changed and is often set in user space.

The value of this separation is that different behaviors can be obtained without changing the kernel. The kernel cannot foresee every situation. But if it offers a good mechanism, the party that knows the situation best, the application or the administrator, chooses the policy itself. Scheduling is the clearest example of this principle (Figure 14.13).

Example 14.4

A database process has four threads: T1 UI (priority 5), T2 query (3), T3 backup (8), T4 log (1). The kernel's mechanism is “run the highest-priority ready thread.” Which thread runs, and what happens when the policy changes?

1. **Initially.** The highest priority is 8; the CPU goes to T3, the backup. The queries wait.
2. **Policy.** The database is the party that knows its own threads best. It decides that queries matter and the backup can wait, and says so with a system call: `sched_setattr(T2, prio=9)`, `sched_setattr(T3, prio=2)`.
3. **Result.** The priorities are now 5, 9, 2, 1. Without any change to kernel code, it picks T2.
4. **Another policy.** An interactive application puts the UI first: the priorities become 9, 6, 2, 1 and T1 runs.

The same mechanism has produced different behavior under different policies.

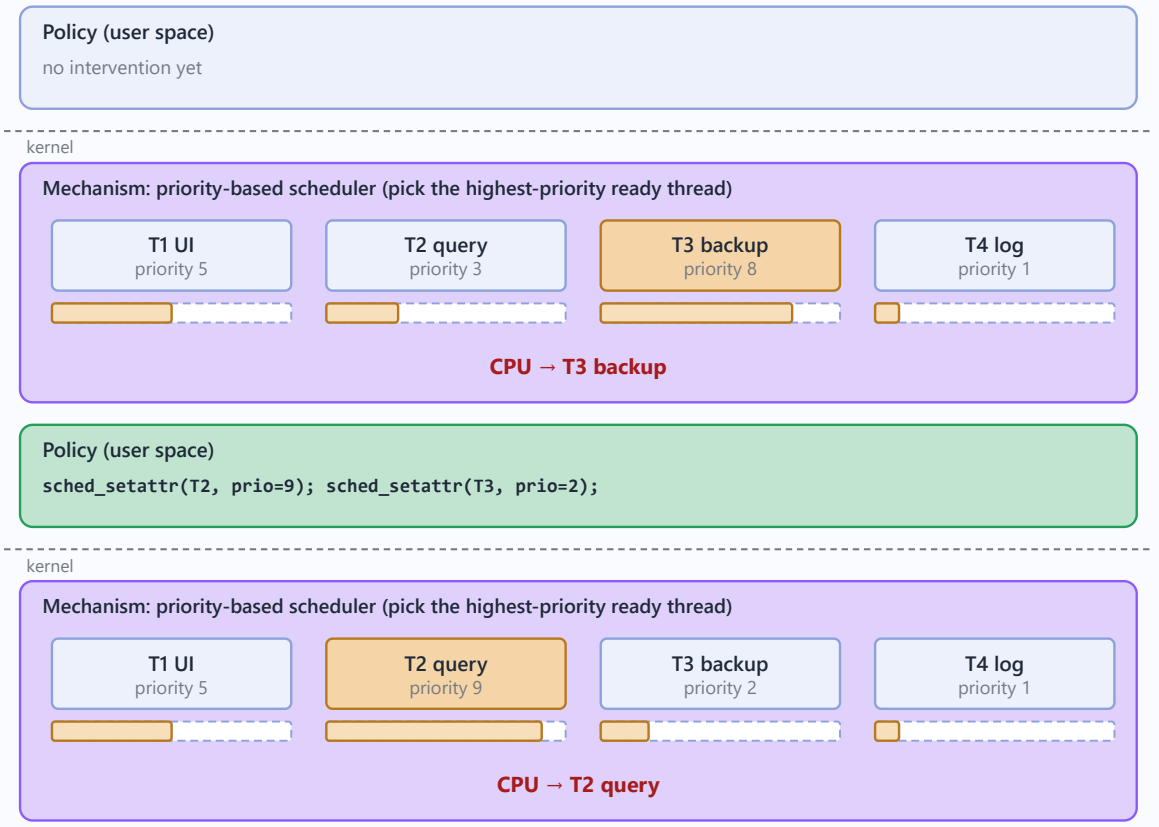


Figure 14.13: Mechanism and policy. (a) User space has not yet intervened; the kernel picks T3 (backup), which has the highest priority. (b) The database has changed the priorities; the same scheduler now picks T2 (query).

The sched_ext facility in Linux 6.12 takes this principle a step further. Not just priorities but the entire scheduling policy can be loaded as an eBPF program. The kernel provides the task queues, the context switching, and the distribution to CPUs; the loaded program decides which task runs when.

The same separation appears everywhere in an operating system:

- **Paging.** The mechanism is the MMU and frame allocation; the policy is the rule that decides which page to evict (LRU, clock).
- **CPU frequency.** The mechanism is the cpufreq driver; the policy is the `schedutil` or `powersave` governor.
- **Security.** The mechanism is the LSM hooks; the policy is the SELinux rules.

> **Common mistake**
>
> Counting all kernel code as mechanism and all user-space code as policy. The distinction is made by the kind of work, not by where the code lives. The page-replacement rule runs in the kernel, but it is a policy: the clock algorithm can be chosen instead of LRU. The MMU's address

translation or the saving of registers in a context switch, on the other hand, are mechanisms independent of any policy.

14.4.2 Orthogonality

Definition 14.9

Orthogonality is the ability to combine separate concepts independently of one another. In an orthogonal design, changing one concept does not affect another.

The term comes from geometry: moving in one direction along two perpendicular axes does not change the position on the other axis. In design, orthogonality makes it possible to obtain many combinations from a small number of concepts. The C language offers a simple example. The lifetime of a variable (`static`) and its type are chosen independently: `static int`, `static char *`, and `static struct proc` are all valid.

The best example in operating systems is Linux's `clone` call (Figure 14.14). When a new execution unit is created, what is shared is chosen with separate flags. `CLONE_VM` shares the address space, `CLONE_FS` the root and working directory, `CLONE_FILES` the open-file table, `CLONE_SIGHAND` the signal handlers, and `CLONE_THREAD` the thread group.

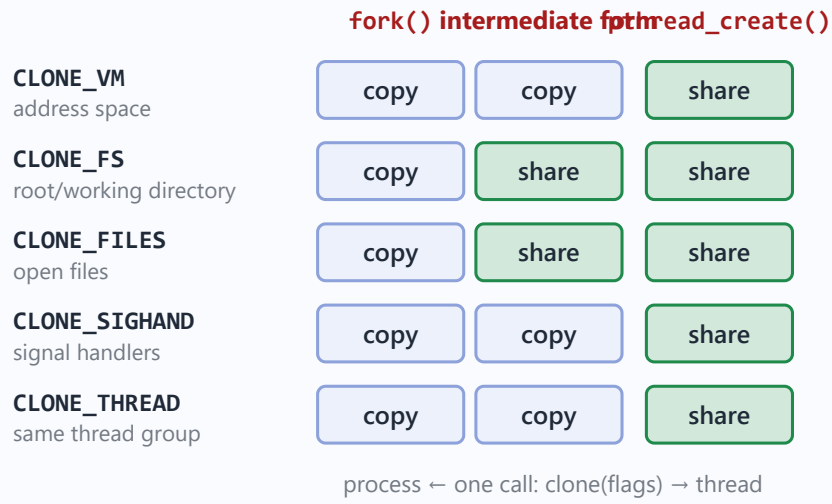


Figure 14.14: Independent flags in the Linux `clone` call. `fork()` shares nothing, `pthread_create()` shares everything; every combination in between is possible.

If none of the flags is given, the result is `fork`: everything is copied. If all are given, the result is `pthread_create`: everything is shared. Process and thread are thus not two separate concepts but the two ends of the same mechanism. Containers are also built with the same call, using the `CLONE_NEW*` namespace flags such as `CLONE_NEWPID` and `CLONE_NEWNET`.

Windows approaches the same problem with a different orthogonal design. A process is a resource container: the address space, the handles, and the security information belong to the process. A thread is the unit of scheduling. The two concepts are separate from the start, and every process contains one or more threads.

14.4.3 Naming: two-level names

An operating system manages thousands of objects: files, processes, devices, memory pages, network endpoints. People want to refer to these objects by memorable names. Inside the system, a fast and unique number is more useful. The solution is two-level naming.

Definition 14.10

An **external name** is the name people use, usually an ASCII or Unicode string. An **internal name** is a fast, unique number used inside the system. A table or a service performs the mapping between the two.

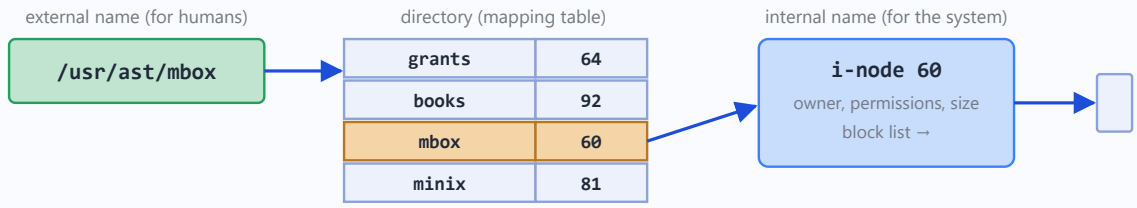


Figure 14.15: Two-level naming. The external name `/usr/ast/mbox` is mapped through the directory (the mapping table) to the internal name, i-node 60; the i-node holds the owner, the permissions, the size, and the block list.

The same pattern recurs in many places in the operating system (Table 14.4).

External name	Mapped by	Internal name
`/usr/ast/mbox`	directory	i-node number
`www.giresun.edu.tr`	DNS	IP address → (ARP) MAC address
file descriptor 3	process file table	`struct file` in the kernel
virtual address	page table	physical frame
`\Device\Harddisk0`	object manager	kernel object (HANDLE)

Table 14.4: Examples of two-level naming.

The biggest gain from the mapping in between is that an object can be moved without changing its name. When a website is moved to another server, only the DNS record changes; users go on typing the same name. When a page is written out to disk and later restored to a different frame, only the

page table is updated; the program uses the same virtual address. The DNS row in the table is interesting: there the mapping is done twice. The name is first translated to an IP address, and the IP address to a MAC address by ARP.

14.4.4 Name resolution step by step

Two-level naming has a cost: every time a name is used, the mapping must be performed. In a file path, this mapping proceeds component by component, and each component may need a disk access.

Example 14.5

The kernel receives the call `open("/usr/ast/mbox")` and nothing is in the cache. How many disk reads are needed to reach the file's i-node?

1. **Root directory.** The i-node of the root directory (i-node 1) is read; it tells us the directory data is in block 1. Block 1 is read, and a linear search finds `usr` → i-node 6. Total: 2 reads.
2. **/usr.** i-node 6 is read, then block 132 of the `/usr` directory. `ast` → i-node 26. Total: 4 reads.
3. **/usr/ast.** i-node 26 is read, then directory block 406. `mbox` → i-node 60. Total: 6 reads.
4. **The file.** Finally, the file's own i-node (60) is read. Total: 7 reads.

The rule is: two reads for every directory on the path (its i-node and its directory block), and one read at the end for the file's i-node. This path with three directories needs 2 × 3 + 1 = 7 reads.

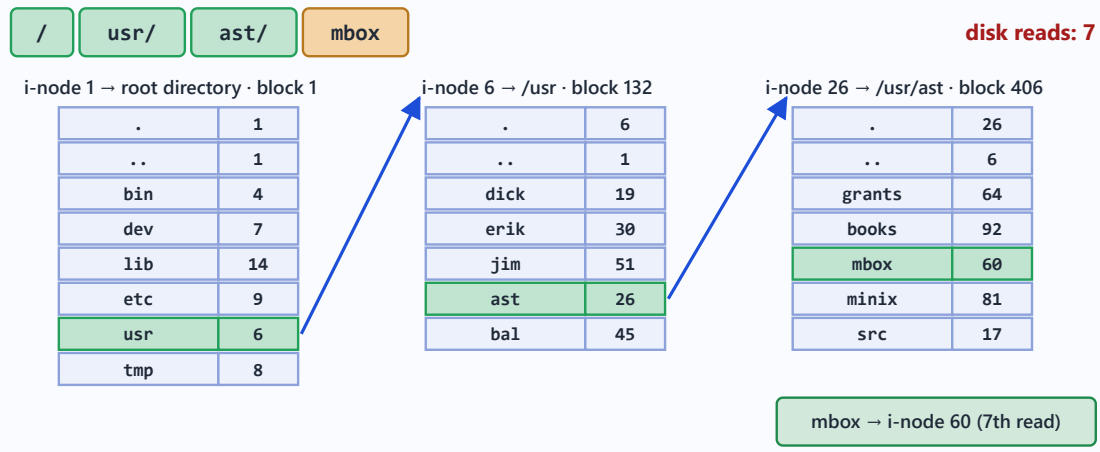


Figure 14.16: Resolving the path `/usr/ast/mbox` component by component. Reading the root directory (i-node 1, block 1), `/usr` (i-node 6, block 132), and `/usr/ast` (i-node 26, block 406) yields i-node 60.

Seven disk accesses were needed just to find the file. On a hard disk, where each access takes milliseconds, this means a delay of tens of milliseconds; not a single byte of the file has been read yet. The flexibility of two-level naming becomes cheap only if the mappings are kept in a cache.

Common mistake

Counting only the directory blocks, or forgetting the file's own i-node. To read the contents of a directory, that directory's i-node must be read first, because it says in which block the directory data lies. The path `/home/ali/ders/notlar.txt`, with four directories, therefore needs 2 × 4 + 1 = 9 reads.

14.4.5 Binding time

Definition 14.11

Binding time is the moment at which a name is bound to the actual object. In **early binding**, the association is fixed in advance; in **late binding**, it is made at the latest possible moment.

A name can be bound at many different moments. Memory addresses are a good example (Figure 14.17). An absolute address embedded in the code while the program is written is the earliest binding. At compile time, a fixed load address can be chosen. At link time, static libraries are added to the program. At load time, relocation and ASLR determine the addresses. At run time, virtual memory, `dlopen`, and DNS leave the binding to the last moment.

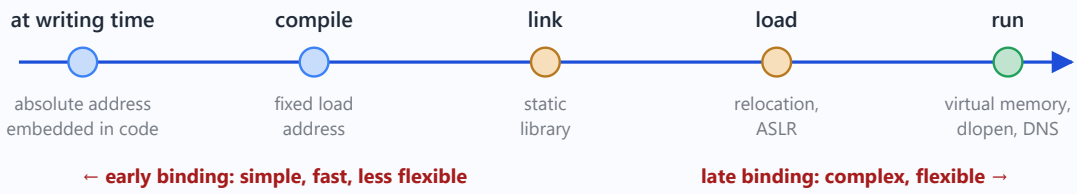


Figure 14.17: The binding-time line: at writing time, compile time, link time, load time, and run time. Flexibility grows from left to right; early binding is simple and fast, late binding is complex but flexible.

Early binding is simple and fast, but less flexible. The program runs only at a particular address; when a library is updated, the program must be recompiled. Late binding is more complex, because it brings an extra indirection and a run-time cost. In return, flexibility is gained: the program runs wherever in memory the page is loaded, and it does not change when the library is updated.

Operating systems generally choose late binding. Virtual memory is the extreme example: the physical location of a page is resolved anew on every access. This cost is bearable thanks to the TLB. The price of late binding is paid, but hardware help is enlisted to reduce it.

14.4.6 Static versus dynamic structures

The size of kernel data structures is also a design decision. Should the process table, the open-file table, or the buffer pool be kept as a fixed-size array at compile time, or as a structure that grows and shrinks at run time?

> **Definition 14.12**
>
> A **static structure** is one whose size is fixed at compile time or boot time, for example a fixed-size array. A **dynamic structure** grows and shrinks at run time by allocating memory, for example a linked list or a hash table.

In old UNIX kernels, the process table was a fixed-size array, and a process was looked up like this:

```
found = 0;
for (p = &proc_table[0];
     p < &proc_table[PROC_TABLE_SIZE];
     p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

`found` is cleared and the pointer `p` is set to the first element of the table. The loop continues until `p` reaches `&proc_table[PROC_TABLE_SIZE]`, the address at the end of the table. On each iteration, the entry's `proc_pid` field is compared with the `pid` being sought. When there is a match, `found = 1` is set and `break` leaves the loop. `p++` advances the pointer not by one byte but by the size of one `struct proc`; in C, pointer arithmetic is done in units of the size of the pointed-to type.

Pid 42 is being sought in a table of six entries. The pids in the table are, in order, 1, 314, 0, 42, 977, 0; 0 means an empty entry (Table 14.5).

Iteration	p	p->proc_pid	Comparison	found
1	`&proc_table[0]`	1	1 ≠ 42	0
2	`&proc_table[1]`	314	314 ≠ 42	0
3	`&proc_table[2]`	0 (empty)	0 ≠ 42	0

Iteration	p	p->proc_pid	Comparison	found
4	`&proc_table[3]`	42	equal, `break`	1

Table 14.5: Linear search for pid 42 in a static process table.

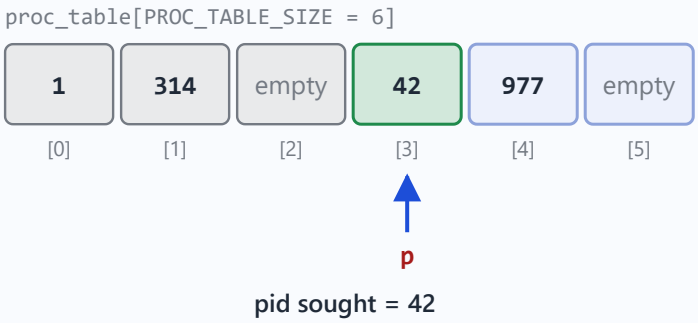


Figure 14.18: A static process table of six entries (`PROC_TABLE_SIZE = 6`). The pointer `p` finds pid 42 at the fourth comparison.

Empty entries are scanned one by one too; that is the cost of linear search. In the worst case the whole table is scanned and the search takes O(n). But for a small, fixed table, this simple code is enough.

The advantages of a static table are simplicity and predictability: no memory allocator is needed, and the table is never fragmented. The disadvantage is that once the table is full, no new process can be created. In old UNIX this limit was the constant `NPROC`. A dynamic structure, such as a linked list, a hash table, or Linux's `xarray`, removes this limit; a hash table and an `xarray` also bring the search close to O(1). The price is a memory allocator, fragmentation, and more complicated locking. There is a middle way too: tables whose size is tunable. In Linux, the largest pid value is read from the file `/proc/sys/kernel/pid_max` and can be changed.

14.4.7 Top-down versus bottom-up implementation

Once a system has been designed, from which end should writing begin? There are two approaches (Figure 14.19).

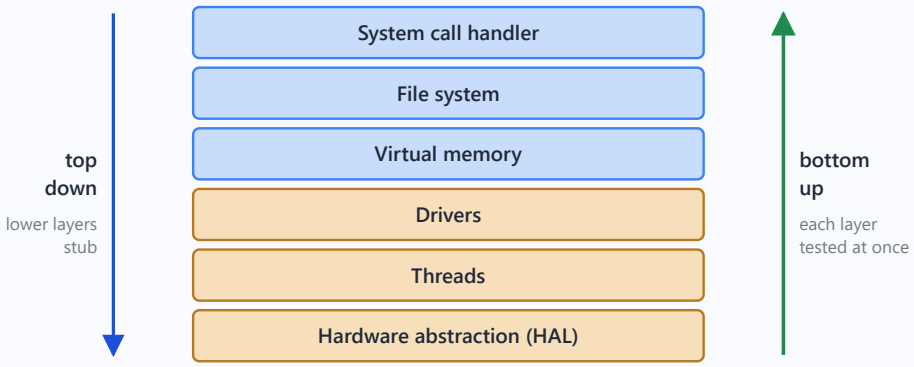


Figure 14.19: In top-down implementation, the lower layers are left as stubs; in bottom-up implementation, one starts from the hardware abstraction and tests each layer as soon as it is finished.

In **top-down** implementation, the system-call handlers are written first. The lower layers are left as **stubs**. A stub is a function declared with the correct signature but with an empty body or returning a constant value. The interface becomes clear early. But realistic testing is hard until the whole system is done, because stubs do not imitate real behavior.

In **bottom-up** implementation, the layer that hides the hardware is written first, then interrupts, the timer, and threads. Each layer is tested when it is finished, because everything beneath it is real. Tanenbaum recommends this route for implementation. The design itself, however, should be thought through top-down: first the interface is determined, and then the layers that will provide that interface are built from the bottom up.

14.4.8 Synchronous versus asynchronous communication

Definition 14.13

In **synchronous** communication, after sending a request the sender blocks until the reply arrives. In **asynchronous** communication, the sender keeps running; completion is reported to it later by a signal, a callback, or a completion queue.

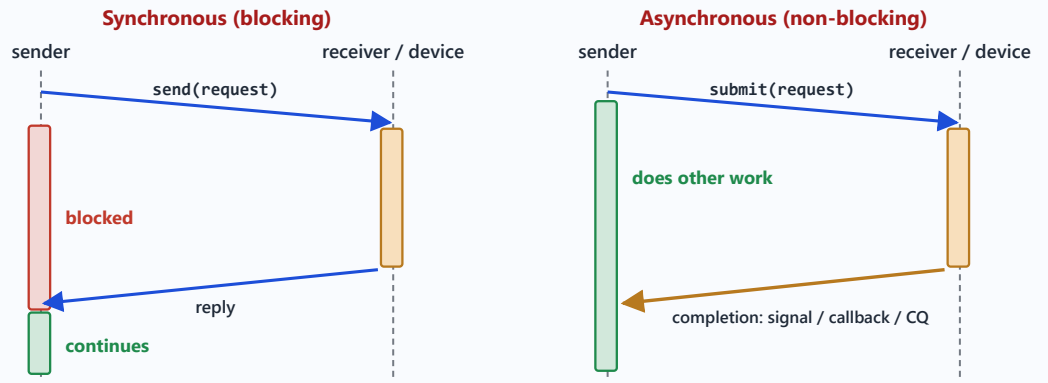


Figure 14.20: On the left, synchronous communication: after `send(request)` the sender blocks until the reply arrives. On the right, asynchronous communication: after `submit(request)` the sender does other work; completion is reported by a signal, a callback, or the CQ.

Synchronous communication is easy to program. The flow reads sequentially: request, reply, next line. The price is that a thread sits idle for every pending request. In microkernels, IPC is usually synchronous; L4 and seL4 follow this route. Because the sender waits for the reply, the kernel does not need to buffer the message, and the message can be transferred directly to the receiver. This makes IPC very fast.

In asynchronous communication, a single thread can have thousands of requests in flight at once. But race conditions, error handling, and buffering become harder. The failure of a request arrives when the code that sent it has long since moved on to something else. `io_uring` in Linux is a current example of this model. The submission (SQ, submission queue) and completion (CQ, completion queue) rings are shared between user space and the kernel. With the right setup, requests can be submitted even without a system call, because a kernel thread watches the submission ring.

Common mistake

Thinking that asynchronous communication is always faster. The latency of a single request is about the same in the two models; the asynchronous model only lets other work be done while waiting. The gain appears when there are many concurrent requests. The price is code that is more complicated and harder to debug.

14.5 Useful techniques

Principles say what to aim for; techniques say how to get there. The techniques in this part are used everywhere in operating-system code, often without being named. Each has a gain and a cost.

14.5.1 Hiding the hardware

An operating system must run on more than one processor architecture. Each architecture has different startup code, register width, and interrupt layout. Letting these differences spread through all the code kills portability. The solution is to gather the hardware-dependent code in a few files. In C, the oldest tool for this is conditional compilation.

Definition 14.14

Conditional compilation means that, with the preprocessor directives `#if`, `#elif`, and `#endif`, only those parts of the source code that satisfy certain conditions are compiled. Lines that do not satisfy the condition never reach the compiler.

The first example selects different startup code according to the CPU type. The file `config.h` defines the constant `CPU`, and the preprocessor lets only the matching block through to compilation.

```
#include "config.h"
void init(void)
{
#if (CPU == X86_64)
    /* x86-64 init code */
#endif
#if (CPU == ARM64)
    /* ARM64 init code */
#endif
#if (CPU == RISCV64)
    /* RISC-V init code */
#endif
}
```

The second example selects a register type according to the word length. On a 32-bit machine `Register` becomes an `int`, on a 64-bit machine a `long`. The last branch is today's portable solution: `uintptr_t` from `<stdint.h>` is an integer type large enough to hold a pointer on every platform.

```
#include "config.h"
#include <stdint.h>
#if (WORD_LENGTH == 32)
typedef int Register;
#elif (WORD_LENGTH == 64)
typedef long Register;
#else  /* today: portable */
typedef uintptr_t Register;
#endif
Register R0, R1, R2, R3;
```

The listings are an updated version of Tanenbaum's Pentium–UltraSPARC example. In today's systems the same idea is applied at the directory level. In Windows, the hardware-dependent code is gathered in the HAL (Hardware Abstraction Layer). In Linux it lives in directories such as `arch/x86`, `arch/arm64`, and `arch/riscv`. When porting to a new processor, only this part is rewritten.

Example 14.6

In `config.h`, the constant `CPU` is defined as `ARM64` and `WORD_LENGTH` as 64. Which startup block does the code compile, and what is `sizeof(Register)`? What changes if the same code is compiled on 64-bit Windows?

1. **Startup.** Only the `#if (CPU == ARM64)` block is compiled. The x86-64 and RISC-V blocks are removed in the preprocessor; the compiler never sees them.
2. **Register.** Because `WORD_LENGTH == 64`, `Register` is a `long`. 64-bit Linux uses the LP64 model; in this model `long` is 8 bytes, so `sizeof(Register)` is 8.
3. **Windows.** 64-bit Windows uses the LLP64 model; there `long` is 4 bytes. The same source code produces a register type of a different size. This is why `uintptr_t` is preferred in portable code.

Common mistake

Scattering conditional compilation everywhere. `#ifdef` blocks sprinkled into every function make the code unreadable, and since some combinations are never compiled, bugs go unnoticed for years. The right approach is to gather the hardware-dependent code in a few files and offer the rest of the code a common interface.

14.5.2 Indirection

Definition 14.15

Indirection is reaching an object not directly but through an intervening table, pointer, or name. By changing the intermediate step, behavior can be changed without touching the parties at the two ends.

A remark attributed to David Wheeler sums up the power of the technique: every problem in computer science can be solved by another level of indirection. Keyboard input is a good example. When a key is pressed, the keyboard does not send a character; it sends a **scan code** that gives the position of the key. What turns this code into a character is the operating system's **key map** (keymap).

Example 14.7

A user presses the key to the right of U on a US keyboard. Which character is produced with three different keymaps?

1. **Hardware.** The keyboard sends scan code 0x17 in set 1. On USB keyboards what arrives is an HID usage code; the driver translates it to a common code.
2. **US (QWERTY) map.** 0x17 → `i`.
3. **Turkish Q map.** Only the table is changed: the same key now produces `ı`.
4. **Turkish F map.** The same key becomes `n`.

In all three cases the hardware and the application are the same; only the table in between changes.

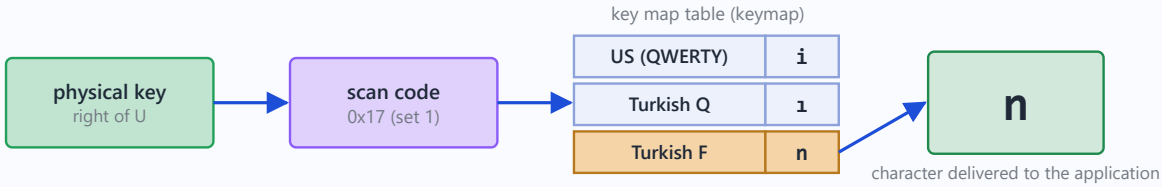


Figure 14.21: Indirection in keyboard input. The physical key produces scan code 0x17; the keymap translates it to `i` in the US layout, to `ı` in Turkish Q, and to `n` in Turkish F.

The gain of the extra step is flexibility, and its cost is a table lookup. The cost is usually negligible, but it can add up on a hot path. In operating systems, indirection is everywhere: the page table in virtual memory, the process file table for file descriptors, another path in symbolic links, the i-node in file systems, device classes in drivers. Two-level naming and late binding are also examples of indirection.

14.5.3 Reuse and reentrancy

In a large system, the same kinds of problems recur in different places. Keeping track of free disk blocks, free memory frames, and unused PIDs is the same problem: recording which elements of a large set are free. Writing a single **bitmap** module for these three is better than writing three separate pieces of code. Linux follows this principle by offering common data structures: the doubly linked list in `list.h`, the red-black tree, and the `xarray` are used throughout the kernel.

Code that is to be reused must meet one condition: it must be callable from several places at the same time.

Definition 14.16

A piece of code is **reentrant** if it works correctly even when the same code is called again while one of its calls is still in progress. The rule: there is no static variable shared between calls; the state is kept in a place supplied by the caller, and if it must be shared, it is protected by a lock.

In the kernel, reentrancy is needed for two reasons. The first is interrupts: while a function is running, an interrupt arrives, and the interrupt handler may call the same function. The second is multiple cores: another CPU core may run the same code at the same time. `strtok` in the standard C library is a classic example of code that fails this condition.

```
char *strtok(char *s, const char *d)
{
    static char *last; /* shared state! */
    ...
}

/* reentrant version: state in the caller */
char *strtok_r(char *s, const char *d,
               char **saveptr);
```

`strtok` splits a string into pieces at delimiters. On the first call the string is passed; on later calls `NULL` is passed and the function continues from where it left off. It remembers where it left off in its internal `static char *last` variable. This variable is not on the stack but in the data segment, shared by all threads. The reentrant version, `strtok_r`, keeps the same information in the `saveptr` pointer supplied by the caller.

14.5.4 Code trace: strtok

To see how the problem arises, we can trace a simplified version of `strtok`. `my_strtok` takes a one-character delimiter and keeps its state in the variable `static char *rest`.

```
char *my_strtok(char *s, char sep) {
    static char *rest;      /* shared by all! */
    if (s) rest = s;
    if (!rest || !*rest) return NULL;
    char *tok = rest;
    while (*rest && *rest != sep) rest++;
    if (*rest) *rest++ = '\0';
    return tok;
}
/* main:  my_strtok(a, ',')  a = "ali,veli" */
/* irq:   my_strtok(b, ';')  b = "x;y"      */
/* main:  my_strtok(NULL, ',')              */
```

The function works line by line as follows. If `s` is given, `rest` is set to its start. If `rest` is `NULL` or points to the end of the string, no pieces are left and `NULL` is returned. Otherwise the start of the piece is saved in `tok`. The `while` loop advances `rest` to the delimiter or to the end of the string. If a delimiter was found, `'\0'` is written over it and `rest` moves to the next character. The function returns `tok`; the returned string ends at the `'\0'` that was written.

The scenario is as follows. `main` is splitting the string `a = "ali,veli"` at commas. Right after it gets the first piece, an interrupt arrives, and the interrupt handler uses the same function to split the string `b = "x;y"` at semicolons. Then `main` continues from where it left off (Table 14.6).

Caller	Call	State of the strings	rest	Returns
main	`my_strtok(a, ',')`	a: `ali\0veli\0`	`&a[4]` (v)	`"ali"`
interrupt	`my_strtok(b, ';')`	b: `x\0y\0`	`&b[2]` (y)	`"x"`
main	`my_strtok(NULL, ',')`	unchanged	`&b[3]` (\0)	`"y"`

Table 14.6: Trace of `my_strtok`. The `rest` column shows the value after the call.

- **First call.** `rest = a`. The loop advances to the comma at `a[3]`; `'\0'` is written over the comma and `rest` moves to `a[4]`, the start of "veli." It returns `"ali"`.
- **Interrupt.** The handler's call arrives with `s = b` and overwrites the shared `rest`. The semicolon becomes `'\0'`, `rest` points to `b[2]`, and it returns `"x"`. The place where `main` left off is lost.
- **main continues.** `main` passes `NULL` and assumes the function is still working on `a`. But `rest` points into `b[2]`. The loop advances to the end of the string; there is no delimiter, so `rest` stays at `b[3]`. It returns `"y"`; the expected result was `"veli"`.

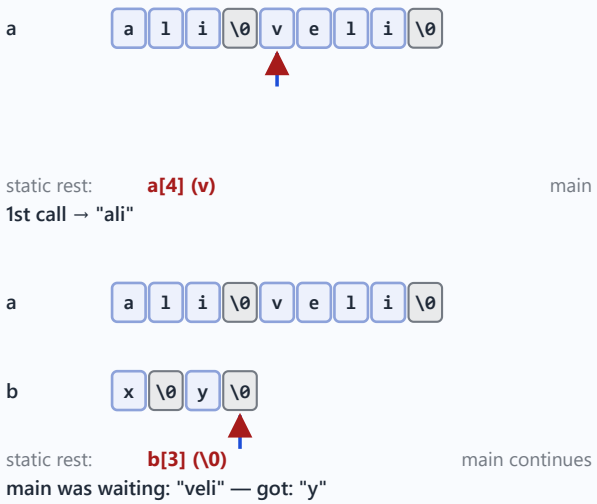


Figure 14.22: (a) After the first call, `rest` points to `a[4]` (v). (b) When `main` continues after the interrupt handler's call, `rest` is inside `b`; `main` expects "veli" but gets "y."

The bug shows up rarely and depending on timing. If the interrupt does not fall exactly between two calls, the program works correctly. Catching this kind of bug by testing is therefore hard. The root cause is clear: the state between calls is kept in a shared `static` variable. The fix is clear too: carry the state to the caller, as in `strtok_r(s, d, &rest)`. Then every caller has its own `rest`, and the interrupt cannot touch the state of `main`.

Common mistake

Thinking that a `static` variable is kept on each thread's own stack. Local `static` variables live in the data segment, and all the threads in the program share them. If two threads use `strtok` at the same time, the pieces may come from the wrong string. A single-core machine is not immune to this bug either: an interrupt or preemption creates the same interleaving.

14.5.5 Brute force and checking for errors first

Definition 14.17

Brute force means preferring the simplest and most obvious solution, even if it seems slow. One moves to a complicated algorithm only when measurement shows it is needed.

For a table of fifty entries, linear search can be both faster and less error-prone than a hash table. Computing the hash function, the collision chains, and resizing cost more in a small table than they gain. A complicated algorithm is needed only if n is large or the code is on a frequently executed path. Sorting is similar: insertion sort suits small arrays, and merge sort or heap sort suits large ones.

The second technique concerns where error checks go. A system-call handler receives a size and a buffer address from the user; it also needs a lock and memory. A bad ordering looks like this:

```
lock(&tbl_lock);
buf = kmalloc(n, GFP_KERNEL);
if (n > MAX || !access_ok(u, n)) {
    kfree(buf);
    unlock(&tbl_lock);
    return -EINVAL;
}
```

The code first takes the lock, then allocates the memory, and only after that checks whether the parameters are valid. If a parameter is bad, it has to give back everything it took, in reverse order: `kfree`, `unlock`, then the error code. This code also never checks whether `kmalloc` failed. A good ordering looks like this:

```
if (n > MAX || !access_ok(u, n))
    return -EINVAL; /* check first */
buf = kmalloc(n, GFP_KERNEL);
if (!buf)
    return -ENOMEM;
lock(&tbl_lock);
...
```

The parameters are checked first; if they are bad, `-EINVAL` is returned without any resource having been acquired. Then the memory is allocated and the success of the allocation is checked. The lock is taken last. Nothing is left to undo on the error path. One of the most common bugs in real kernels is forgetting an `unlock` on an error path. Checking for errors first eliminates this class of bug.

Common mistake

Taking brute force to mean “sloppy code.” The principle is to choose the simple solution knowingly and by measurement. A brute-force scan of a table is a sound design decision as long as *n* stays small. Only when *n* grows, or when a profile shows this code to be a bottleneck, should it be rethought.

14.6 Performance

Every operating system wants to be fast. But speed is meaningful only once correctness is secured, and every speedup makes the code more complicated. This part first covers how to decide what to speed up, and then the most commonly used techniques.

14.6.1 Why is it slow, and what should be sped up?

Hardware has become thousands of times faster over the decades, but the speed the user feels has not grown at the same rate. A large part of the gain was consumed by the growth of software. There are three reasons. The first is feature bloat: every release adds new features and cannot remove the old ones. The second is layers of backward compatibility. The third is security measures. The patches added against the Spectre and Meltdown vulnerabilities, such as separating the kernel and user page tables, raised the cost of every system call. The indirection added for generality also takes a little time at every layer.

Measurement decides what to speed up. Donald Knuth's saying is often cited here: premature optimization is the root of all evil. First write correct code, then fast code. Speedups are applied only to the parts shown by measurement to be bottlenecks. The principle of "good enough" also belongs here: making the code twice as complicated for a 5% gain is usually not worth it. In Linux, the measurement tools are `perf`, the eBPF-based `bpftrace`, and flame graphs.

A simple calculation shows how to read a profile (Figure 14.23).

Example 14.8

A profile shows that 70% of the running time is spent in the function `f()` and 30% in `g()`. Which pays off more, making `f()` twice as fast or `g()` twice as fast?

1. **Starting point.** Take the total time to be 100 units: `f` 70, `g` 30.
2. **f twice as fast.** `f` drops to 35 units; the total is 35 + 30 = 65. The speedup is 100 / 65 ≈ 1.54×.
3. **g twice as fast.** `g` drops to 15 units; the total is 70 + 15 = 85. The speedup is 100 / 85 ≈ 1.18×.
4. **Result.** For the same effort, speeding up `f` pays off far more. Even if `g` were infinitely fast, the total time would not drop below 70.

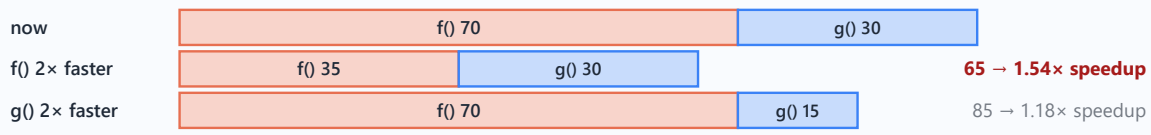


Figure 14.23: A profile: 70% of the time is in `f()`. Making `f()` twice as fast brings the total down to 65 (1.54×); making `g()` twice as fast brings it to 85 (1.18×).

The general form of this calculation is called Amdahl's law.

Theorem 14.1 (Amdahl's law)

If a fraction p of the running time is made k times faster, the overall speedup is

$$S = 1 / ((1 - p) + p / k)$$

no matter how large k is, the speedup cannot exceed $1 / (1 - p)$.

The values in the example fit the formula: for $p = 0.7$ and $k = 2$, $S = 1 / (0.3 + 0.35) \approx 1.54$; for $p = 0.3$, $S = 1 / (0.7 + 0.15) \approx 1.18$. Even if `f` were infinitely fast, the speedup could not exceed $1 / 0.3 \approx 3.33$. The law shows numerically that optimization effort should be spent on the common case.

There are five classic techniques to turn to after measuring: the space-time trade-off, caching, hints, exploiting locality, and optimizing the common case.

14.6.2 Space-time trade-off: counting bits

Definition 14.18

A **space-time trade-off** means shortening the running time by using more memory or, conversely, saving memory by running longer.

The classic example is counting the 1 bits in a byte. This operation is common in the bitmaps of memory allocators and in permission masks. The first version tests each bit one by one in a loop:

```
#define BYTE_SIZE 8
int bit_count(int byte)
{
    int i, count = 0;
    for (i = 0; i < BYTE_SIZE; i++)
        if ((byte >> i) & 1)
            count++;
    return count;
}
```

The loop counts `i` from 0 to 7. On each iteration the byte is shifted right by `i` bits and the lowest bit is masked with 1: `(byte >> i) & 1`. If the result is 1, the counter is incremented. The example byte is 0xB7: 183 in decimal, 1011 0111 in binary (Table 14.7).

i	0	1	2	3	4	5	6	7
(byte >> i) & 1	1	1	1	0	1	1	0	1

i	0	1	2	3	4	5	6	7
count	1	2	3	3	4	5	5	6

Table 14.7: Trace of `bit_count(0xB7)`. 0xB7 = 1011 0111.

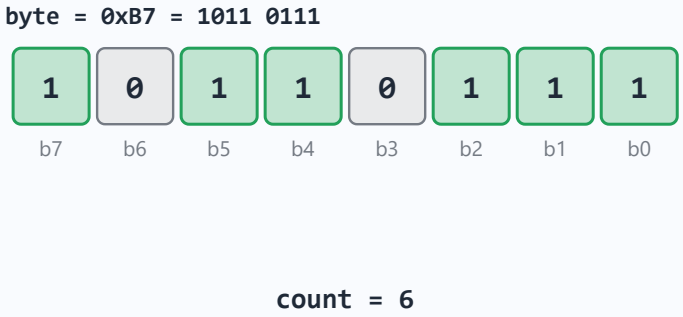


Figure 14.24: The eight bits of 0xB7 are counted one by one in a loop; the result is 6.

Bits 3 and 6 are zero; on those iterations the branch is not taken and the counter does not change. The loop runs eight times and returns 6. Memory use is negligible. But there are eight iterations per byte, and on each a shift, a mask, a comparison, and a branch. When branch prediction fails, the processor's pipeline is flushed, which adds cost.

Two faster versions buy time with space:

```
/* (b) macro: no loop */
#define bit_count(b) ((b&1) + ((b>>1)&1) + \
    ((b>>2)&1) + ... + ((b>>7)&1))

/* (c) table: 256 bytes */
char bits[256] =
    {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...};
#define bit_count(b) (int) bits[b]
```

The macro (b) unrolls the loop: eight terms are summed directly. There is no loop and no branch, but about 24 instructions are needed. Version (c) uses a table of 256 entries. Entry *i* of the table is the number of 1 bits in the binary representation of *i*. Counting bits comes down to a single memory read (Figure 14.25). The methods are compared in Table 14.8.

Figure 14.25: The first 32 values of the 256-entry bit-count table and `bits[0xB7] = bits[183] = 6`.

Method	Memory	Work per byte
(a) loop	~0	8 iterations: shift, mask, branch
(b) macro	~0	~24 instructions, no branch
(c) 256-entry table	256 B	1 memory read
16-entry table (nibble)	16 B	2 reads + an addition
POPCNT instruction	0	1 instruction (`__builtin_popcount`)

Table 14.8: Space and time cost of the bit-counting methods.

The last row of the table is today's reality. x86-64 and ARM64 processors have a special instruction that counts 1 bits, and the compiler turns a call to `__builtin_popcount` into that instruction. Even so, the space-time trade-off keeps its importance, because the same idea is used in hundreds of other computations where the hardware does not help.

14.6.3 Code trace: the nibble table

Between the 256-byte table and the loop there is a middle ground. If a byte is split into two nibbles (nibble), a table of 16 entries is enough. The table, too, can be built by a loop instead of being written by hand:

```
char nib[16];            /* 16 bytes */
void build_table(void) {
    nib[0] = 0;
    for (int i = 1; i < 16; i++)
        nib[i] = (i & 1) + nib[i >> 1];
}
int bit_cnt(unsigned char b) {
    return nib[b & 0x0F] + nib[b >> 4];
}
```

The observation behind `build_table` is this. The number of 1 bits in i is the value of its lowest bit plus the number of 1 bits in $i >> 1$. Since $i >> 1$ is always smaller than i, that value has already been computed when the loop proceeds from small to large. This is a dynamic-programming recurrence.

Example 14.9

With which values does `build_table()` fill the table?

1. **Start.** `nib[0] = 0`; there are no bits in 0.
2. **i = 1 … 3.** `nib[1] = 1 + nib[0] = 1`, `nib[2] = 0 + nib[1] = 1`, `nib[3] = 1 + nib[1] = 2`.
3. **i = 4 … 7.** 1, 2, 2, 3. For example, `nib[7] = 1 + nib[3] = 3`.
4. **i = 8 … 15.** 1, 2, 2, 3, 2, 3, 3, 4. For example, `nib[15] = 1 + nib[7] = 4`.

The whole table: 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4.

Once the table is ready, `bit_cnt` splits the byte into two halves. `b & 0x0F` gives the low four bits and `b >> 4` the high four. The two table values are added. There is no loop and no branch (Table 14.9).

b	Binary	Low half	High half	Sum
0xB7	1011 0111	`nib[7]` = 3	`nib[B]` = 3	6
0x5C	0101 1100	`nib[C]` = 2	`nib[5]` = 2	4

Table 14.9: Calls to `bit_cnt`.

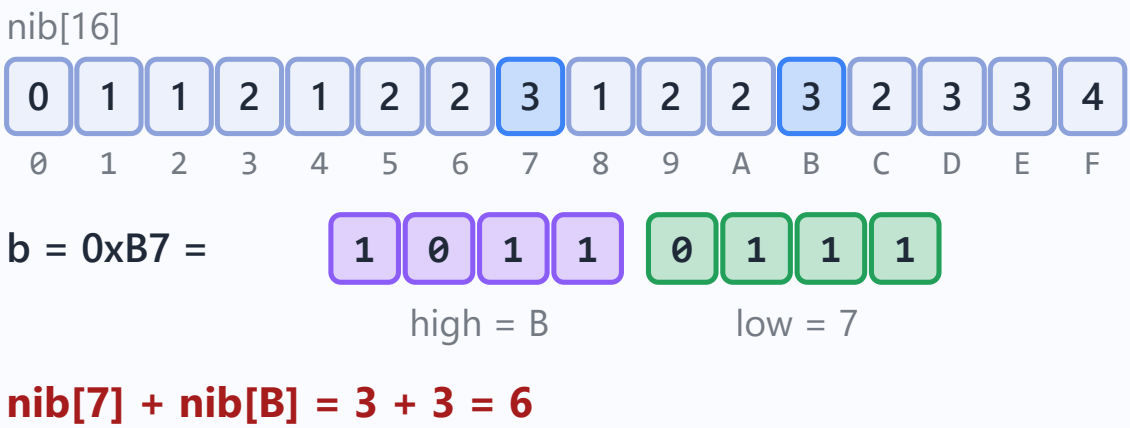


Figure 14.26: The 16-entry `nib` table, filled in. `bit_cnt(0xB7)`: two reads, for low half 7 and high half B, give 3 + 3 = 6.

Two memory reads and an addition suffice for 0xB7; the loop version had gone around eight times for the same byte. The nibble table takes 16 bytes, one-sixteenth of the 256-entry table. In exchange, there is one extra read and one extra addition. We see that the same question can be answered in several different ways depending on how much space is spent.

Common mistake

Assuming that the table method is always the fastest. The table lives in memory, and if it is not in the cache, a single read can take longer than a whole loop. A 256-byte table occupies four cache lines. As long as it is used constantly on a hot path, it stays in the cache and is fast. If it is used rarely, it may miss every time. So here, too, the decision is made by measurement.

14.6.4 Space-time trade-off: the color palette

The trade-off works in the opposite direction too: by spending a little time, a great deal of space can be saved. The color palette is the classic example (Figure 14.27).

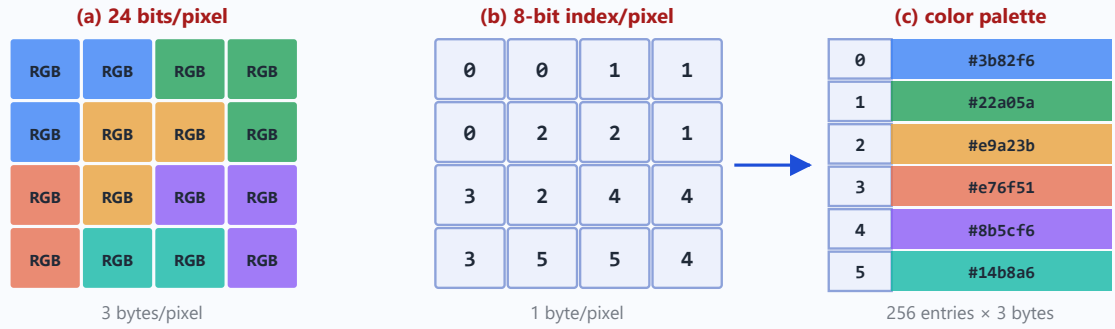


Figure 14.27: (a) 24 bits per pixel (3-byte RGB). (b) An 8-bit palette index per pixel (1 byte). (c) A color palette of 256 entries; each entry is 3 bytes.

In an image, the red, green, and blue values can be stored directly for each pixel. That is 3 bytes per pixel. If the image uses at most 256 different colors, there is another way. The colors used are collected in a palette table, and for each pixel only a 1-byte index is stored. When the image is displayed, each index is translated to the real color through the palette.

Example 14.10

A 1024 × 768 image is stored in two formats. What are the sizes and the ratio?

1. **24 bits/pixel.** 1024 × 768 × 3 = 2,359,296 bytes; 2,359,296 / 2^{20} = 2.25 MiB.
2. **8-bit index + palette.** The indices take 1024 × 768 × 1 = 786,432 bytes; the palette 256 × 3 = 768 bytes. Total: 787,200 bytes.
3. **Ratio.** 2,359,296 / 787,200 ≈ 3.0. The space drops to about a third.

Format	Size (bytes)	Ratio
(a) 24 bits/pixel	2,359,296 (2.25 MiB)	1

Format	Size (bytes)	Ratio
(b)+(c) 8-bit index + palette	786,432 + 768 = 787,200	≈ 1/3

Table 14.10: Size of a 1024 × 768 image in two formats.

The space gained costs in two ways. The time cost is the table lookup made as each pixel is displayed. The constraint cost is the limit of 256 colors; an image with thousands of colors, such as a photograph, does not fit this format without loss. GIF and PNG-8 use this method; GIF also compresses with LZW. The palette itself is 768 bytes, and as the image grows its share of the total becomes negligible.

Common mistake

Leaving out the size of the palette, or counting the palette per pixel. The palette is stored once per image. In an 800 × 600 image the indices are 480,000 bytes and the palette 768 bytes; the total is 480,768 bytes. The 24-bit format takes 1,440,000 bytes, and the ratio is again about 3.

14.6.5 Caching

Definition 14.19

A **cache** is a structure that stores the result of an expensive computation or a slow access so that it can be supplied quickly when the same request comes again. The value in the cache must always be correct; when the source changes, the entry is invalidated.

The general rule is simple: if the same result is computed again and again, store it. Name resolution is tailor-made for this. Finding the path `/usr/ast/mbox` without a cache took seven disk reads. If the kernel keeps the result of every path it resolves in a table, then when the same path is asked for a second time, the disk is not visited at all (Table 14.11).

Path	i-node
`/usr`	6
`/usr/ast`	26
`/usr/ast/mbox`	60
`/usr/ast/books`	92
`/usr/bal`	45
`/usr/bal/paper.ps`	85

Table 14.11: A path → i-node cache.

Three cases are distinguished. With no cache, finding `/usr/ast/mbox` reads the root i-node, the root directory, the `/usr` i-node, the `/usr` directory, the `/usr/ast` i-node, and the `/usr/ast` directory. That is six disk reads; with the file's own i-node it makes seven. If the full path is in the cache, a single memory lookup suffices and the disk accesses are zero. If only `/usr/ast` is in the cache, the search continues from there: a partial hit is also a gain.

In Linux this job is done by several caches. The **directory-entry cache** (dcache, dentry cache) maps path components to i-nodes. The i-node cache holds the i-nodes themselves, and the page cache holds file contents. This is why free RAM in Linux often looks “full”: unused memory is given to these caches and taken back when needed.

The price of a cache is **consistency**. If a file is deleted or renamed, its entry must be invalidated; otherwise the cache points to a file that does not exist. This burden comes from the requirement that the value in the cache always be correct. In an operating system, caches exist at every layer: the TLB, the CPU caches, the page cache, the dcache, the ARP and DNS caches.

14.6.6 Code trace: the path cache

A path resolver that also exploits partial hits can be written in a few lines. `c[]` holds the components of the path and `n` the number of components. The function finds the longest prefix in the cache and resolves only the remaining components from disk.

```
/* c[]: "usr","bal","thesis.tex"; n = 3 */
int resolve(char *c[], int n) {
    int k = n, ino = ROOT, reads = 0;
    while (k > 0 && !in_cache(c, k, &ino))
        k--;              /* longest prefix */
    for (int j = k; j < n; j++) {
        ino = dir_lookup(ino, c[j]);
        reads += 2;       /* i-node + block */
        cache_add(c, j + 1, ino);
    }
    return ino;
}
```

The function has two phases. In the first, `k` starts at `n` and decreases. `in_cache(c, k, &ino)` checks whether the path made of the first `k` components is in the cache; if it is, it writes the i-node number to `ino`. The loop stops at the first hit or when `k` falls to 0. If `k` falls to 0, `ino` stays at its initial value, the root i-node (`ROOT` = 1). In the second phase, `dir_lookup` is called for every remaining component. This call reads the directory's i-node and the directory block, so `reads` grows by 2. The result found is added to the cache with `cache_add`.

Initially the cache holds five paths: `/usr` (6), `/usr/ast` (26), `/usr/ast/mbox` (60), `/usr/bal` (45), and `/usr/bal/paper.ps` (85). Three lookups are made in turn (Table 14.12).

Path	k (prefix)	Initial ino	Resolved from disk	reads	Result
`/usr/bal/thesis.tex`	2 (`/usr/bal`)	45	`thesis.tex` → 131	2	131
`/usr/bal/thesis.tex`	3 (full path)	131	none	0	131
`/etc/passwd`	0	1 (root)	`etc` → 14, `passwd` → 212	4	212

Table 14.12: Trace of `resolve()`.

- **First lookup.** The full path `/usr/bal/thesis.tex` is not in the cache, so `k` drops to 2. `/usr/bal` is in the cache: `ino = 45`. There was no need to start from the root; two components came for free. For the one remaining component, the i-node of 45 and its directory block are read: `thesis.tex` → 131. The result is added to the cache.
- **The same path again.** The full path is now in the cache. The first loop stops at once, and the second loop never runs. There are 0 disk reads.
- `/etc/passwd`**.** Neither the full path nor `/etc` is in the cache; `k` drops to 0 and resolution starts from the root i-node. The root i-node and the root directory block are read: `etc` → 14; `/etc` enters the cache. Then the `/etc` i-node and directory block: `passwd` → 212. Total: 4 reads.

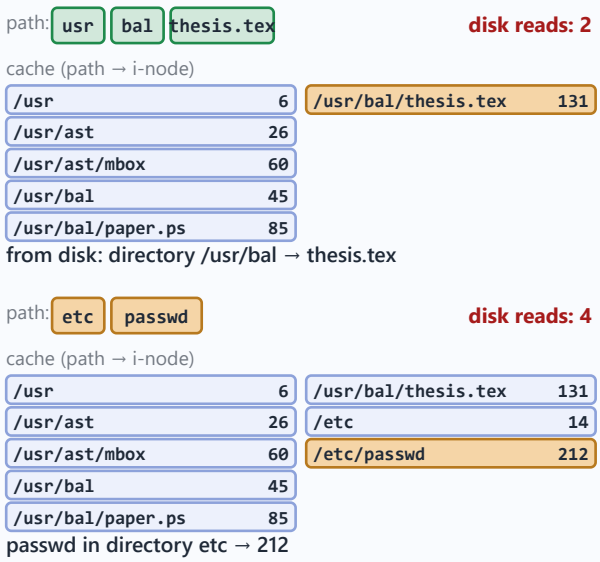


Figure 14.28: (a) `/usr/bal` comes from the cache; only `thesis.tex` is resolved from disk (2 reads) and added to the cache. (b) `/etc/passwd` is resolved from the root (4 reads); at the end the cache holds eight paths.

Without a cache, two reads are needed per component; if the longest prefix is in the cache, only the remaining components are paid for. The worst case is when no prefix is in the cache, and then the cost is proportional to the number of components. In Linux this cache is the dcache, which keeps the components separately; the logic is the same. The reads the code counts do not include the file's own i-node; the function returns only the i-node number.

Common mistake

Using the cache only for full hits. If the full path is not in the cache, starting the search from the root throws away the cached knowledge of `/usr/bal`. The longest-prefix rule exploits partial hits as well.

14.6.7 Hints

Definition 14.20

A **hint** is a piece of information that is probably right but may be wrong. A hint is not invalidated; instead, before it is used, it is verified by a cheap check.

A hint resembles a cache, but the difference is important. The value in a cache must always be correct; when the source changes, the cache must therefore be updated, and this brings a consistency burden. A hint carries no such burden. If the hint turns out to be wrong, the system notices at the moment of use and falls back on the slow path (Figure 14.29).

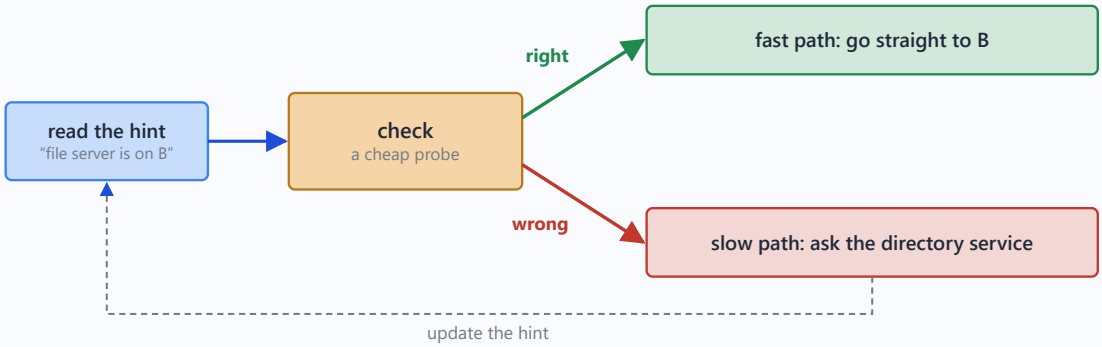


Figure 14.29: Using a hint. The hint is read ("the file server is on B") and verified by a cheap probe. If it is right, the fast path goes straight to B; if wrong, the slow path asks the directory service and the hint is updated.

An example from a distributed file system explains the technique. A client keeps, as a hint, which server a file lives on. The file may be moved from one server to another, and the client may not know. The client first asks server B, named in the hint. If B really holds the file, the job is done. If not, the client consults the directory service, learns the file's new location, and updates the hint. When a file is moved, no client needs to be told.

Other examples:

- **Redirect cache on the Web.** The place a page has moved to is remembered; if it is wrong, the server redirects again.

- **Branch prediction.** The processor guesses the direction of a branch and starts running down that path; if the guess is wrong, it discards the results.
- `madvise(MADV_SEQUENTIAL)`. The application tells the kernel it will read a memory region sequentially. The kernel does more aggressive read-ahead. If the hint is wrong, only performance suffers; correctness is not affected.

> **Common mistake**
>
> Regarding the possibility that a hint is wrong as a flaw. That is where its whole value lies: because it is allowed to be wrong, there is no need for invalidation messages, locks, or consistency protocols. The requirement is one thing only: a wrong hint must not lead to a wrong result, only to the slow path.

14.6.8 Exploiting locality

> **Definition 14.21**
>
> **Temporal locality**: data used recently will soon be used again. **Spatial locality**: when one item is used, nearby items are used too.

Temporal locality is the basis of the cache: if what was used recently is kept, the next request is likely to hit. Spatial locality is the basis of read-ahead and large pages: when one block is requested, several blocks after it are fetched too.

One way to exploit locality is to place data that is used together side by side (Figure 14.30). The BSD Fast File System (FFS) divides the disk into cylinder groups, and ext4 into block groups. A file's i-node and its data blocks are put in the same group, so after reading the i-node the disk arm does not have to travel far to reach the data. The same principle holds in the processor. If a thread is always scheduled on the same CPU core, that core's L1 and L2 caches stay “warm.” On NUMA machines, memory is allocated from the node where the thread runs.

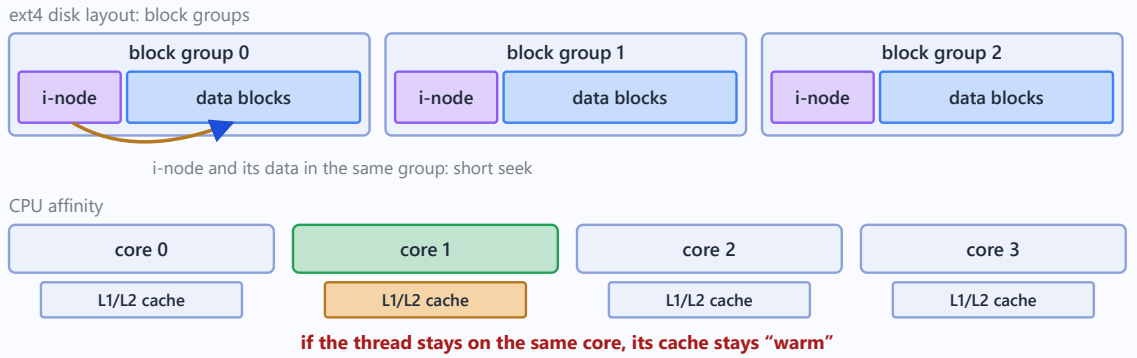


Figure 14.30: Exploiting locality. Top, ext4 block groups: the i-node and its data are in the same group, so the seek is short. Bottom, CPU affinity: if the thread stays on the same core, its L1/L2 cache stays warm.

14.6.9 Optimizing the common case: the futex

One piece of advice from Butler Lampson on system design is this: make the common case fast and the rare case correct. Amdahl's law shows why; the gain comes mostly from speeding up the part that takes the most time. Locks are the best example of this principle.

Entering the kernel every time to take a lock is expensive: a system call, a mode switch, and queue operations. Yet measurements show that the great majority of locks are taken without contention. The thread asking for the lock is usually alone. If this is the common case, it is what needs to be made fast.

Definition 14.22

A **futex** (fast userspace mutex) is the Linux synchronization mechanism in which the state of the lock is kept in an integer in user space and the kernel is entered only to wait and to wake. The lock word takes the value 0 (free), 1 (held), or 2 (held, with waiters).

The atomic **compare-and-swap** (CAS, compare-and-swap) operation is used on the lock word. CAS replaces a memory word with a new value if it equals the expected value, in a single indivisible step, and reports whether it succeeded. A scenario in which two threads ask for the lock in turn is traced in four steps (Table 14.13).

Step	Event	Lock word	Wait queue	System call
1	The lock is free, A wants the lock	0	empty	—
2	A: CAS 0 → 1 succeeds	1	empty	none
3	B: CAS fails, sets the value to 2	2	B	`futex(FUTEX_WAIT)`
4	A releases; old value 2	0	empty	`futex(FUTEX_WAKE)`

Table 14.13: Trace of a futex-based lock.

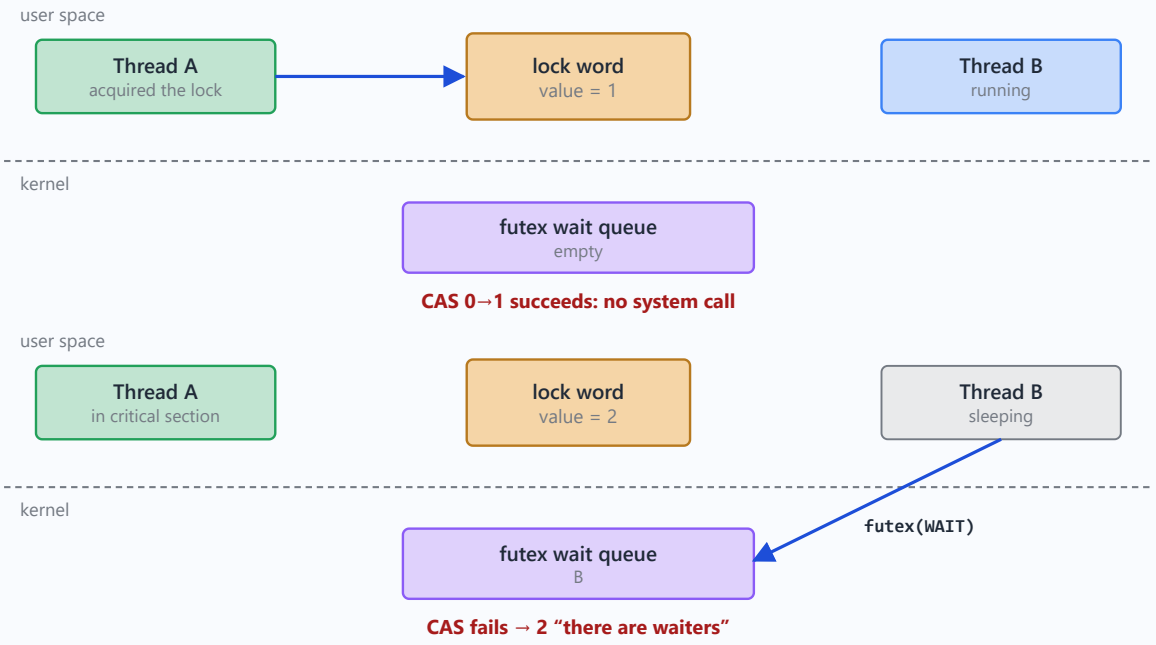


Figure 14.31: A futex-based lock. (a) No contention: A takes the lock in user space with CAS 0 → 1; no system call is made. (b) The rare case: B's CAS fails, it sets the value to 2 and sleeps in the kernel with `futex(WAIT)`.

- **Fast path.** A turns 0 into 1 with CAS. The kernel is never entered; the operation takes a few nanoseconds.
- **Slow path.** B also wants the lock; because the lock word is not 0, its CAS fails. B sets the value to 2, that is, puts down the "there are waiters" mark, and sleeps in the kernel with `futex(FUTEX_WAIT)`. The kernel is entered only here.
- **Release.** When A releases the lock, it sets the word to 0 and looks at the old value. If the old value is 2, it calls `futex(FUTEX_WAKE)`, and B wakes up and takes the lock. Had the old value been 1, it would have made no call at all.

An uncontended lock acquire and release thus finishes without making any system call. The kernel steps in only when waiting is really needed. The distinction between the values 1 and 2 is part of the common-case principle too: if nobody is waiting, the kernel is not visited even on release.

There are other examples of the same principle. The frequently used system calls have a short fast path in the kernel. Some calls, such as `clock_gettime`, are served by the vDSO without entering the kernel at all, read from a page mapped into user space. A TLB hit follows the same logic: the common case of address translation completes without a page-table walk.

Common mistake

Thinking that a futex is a lock in itself. A futex is not a lock but a wait–wake mechanism. The lock itself is the integer in user space and the CAS operations applied to it. Library locks such as `pthread_mutex` combine the two parts.

14.7 Project management

An operating system is the work of hundreds, sometimes thousands, of people over many years. At this scale, how people work together decides the outcome as much as technical decisions do. The lessons Fred Brooks drew from IBM's OS/360 project are still valid today.

14.7.1 The mythical man-month and Brooks's law

Definition 14.23

A **man-month** is one person's work for one month, used to measure the size of a project. **Brooks's law**: adding staff to a late software project makes it later.

The man-month is a misleading unit, because it assumes that work can be divided freely among people. A job of 12 man-months is not finished in 1 month by 12 people. People must coordinate with one another, and the number of communication paths grows quadratically as the team grows.

Example 14.11

If there is one communication path between every pair, how many paths are there in a team of n people? What do we see if we compute it for 2, 4, 6, 8, and 10 people?

1. **Formula.** Each of the n people is connected to the other $n - 1$; since each path is counted twice, the total is $n(n - 1) / 2$.
2. **Values.** 1 path for 2 people, 6 for 4, 15 for 6, 28 for 8, 45 for 10.
3. **Interpretation.** While the number of people grows fivefold from 2 to 10, the number of paths rises from 1 to 45. Adding 3 people to a team of 5 raises the headcount by a factor of 1.6 but the number of paths from 10 to 28, a factor of 2.8.

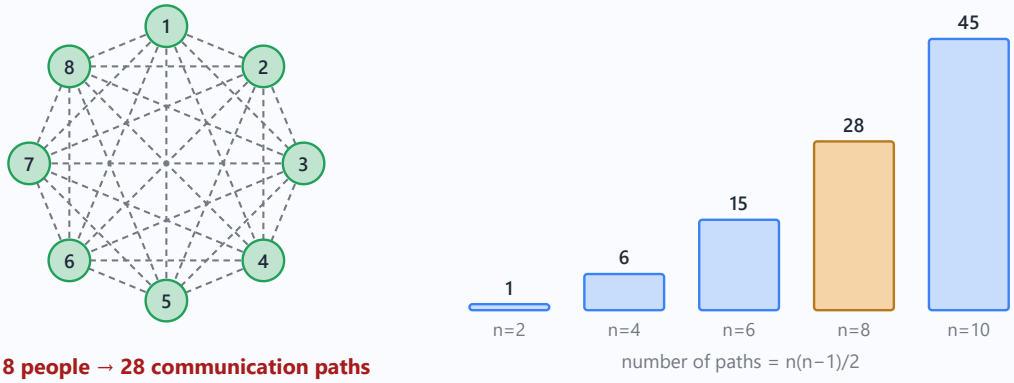


Figure 14.32: 28 communication paths in a team of 8. The bars on the right show the quadratic growth of the number of paths as $n(n-1)/2$: 1, 6, 15, 28, 45.

Every new person must coordinate with everyone: meetings, documents, and interface agreements multiply. A person added to a late project must also be trained. And those who will train the newcomer are the people who already know the project, the most valuable ones. In the short run, every person added lowers the team's total productivity. Brooks's law is the combination of these two effects.

Based on his OS/360 experience, Brooks also proposes a distribution of time: 1/3 for planning, 1/6 for coding, 1/4 for module testing, 1/4 for system testing. In a 12-month project, that means 4 months for planning, 2 for coding, 3 for module testing, and 3 for system testing. Writing code is only a sixth of the work; half the time goes to testing.

Common mistake

Reading Brooks's law as “a team should never be enlarged.” The law is about people added late to a late project. In the planning stage, on a project whose work can be split into independent parts, enlarging the team can help. The problem is adding people at a moment when the work cannot be divided and the cost of learning is high.

14.7.2 Team structure, experience, and the silver bullet

The way to reduce the communication burden is to divide the team into small, strong teams. In the **surgical team** model proposed by Harlan Mills and adopted by Brooks, the chief programmer designs and codes, and the others support him: one is an assistant, one manages the documents, one prepares the tools. Conceptual integrity stays in the hands of a few. According to Brooks, the conceptual integrity of a system is the most important quality of its design.

The development process of Linux is a present-day, large-scale form of this idea (Figure 14.33). The patches of thousands of contributors flow first to driver and subarea maintainers, then to subsystem maintainers, and finally to Linus Torvalds. At every level there is a small group that preserves

conceptual integrity.

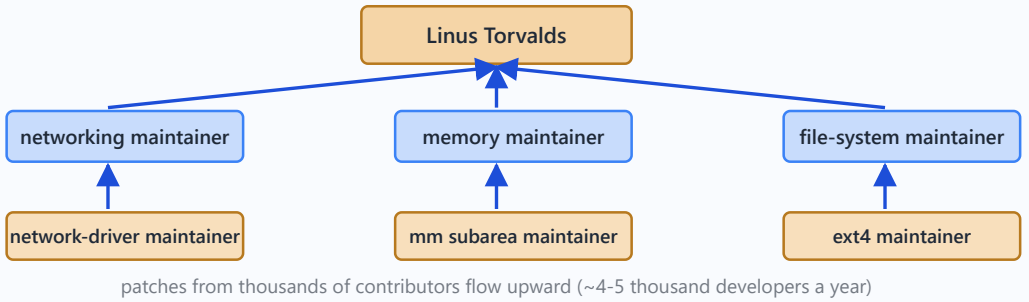


Figure 14.33: The Linux development hierarchy: contributors' patches flow to driver and subarea maintainers (network driver, mm, ext4), from there to subsystem maintainers (networking, memory, file system), and to Linus Torvalds. About 4–5 thousand developers contribute each year.

Experience plays a large role too. A good programmer is many times more productive than average; this difference can matter more than the number of people on the team. Two of Brooks's warnings belong here. The first is his saying “plan to throw one away”: the first version will most likely be thrown away, so it should be thought of from the start as a prototype. The second is the **second-system effect**: a designer who has successfully finished a first system adds to the second all the embellishments left out of the first, and the system bloats.

In his 1986 essay “No Silver Bullet,” Brooks reaches a more general conclusion: there is no single “silver bullet” that will by itself raise productivity in software development tenfold. He divides complexity into two kinds. **Accidental complexity** (accidental complexity) comes from tools and languages; better compilers, editors, and languages reduce it. **Essential complexity** (essential complexity) lies in the problem itself, and no tool can eliminate it. An operating system's concurrency, security requirements, and compatibility burden are essential. AI-assisted coding tools do not change this picture: they speed up the writing of code, but they do not remove the responsibility for design.

14.8 Testing and debugging

In Brooks's distribution of time, testing takes half. In operating systems the share is perhaps even larger, because a bug in the kernel affects the whole system. This part describes why kernel code is hard to test, which kinds of tests and tools are used, and by what means a bug is found.

14.8.1 Why is operating-system code hard to test?

In an application, a failing test usually produces just an error message. In the kernel, five factors make the job harder.

- **Privileged mode.** There is no higher layer to catch the bug. In the kernel, a single bad pointer produces an oops or a panic and brings down the whole system, test harness included.

- **Concurrency.** Interrupts, preemption, and multiple cores produce countless interleavings. A bug appears only under a rare timing.
- **Nondeterminism.** Some bugs do not repeat on every run. These are jokingly called **heisenbugs**: even adding a `printk` can change the timing and hide the bug.
- **Variety.** There are thousands of drivers, many architectures (x86-64, arm64, RISC-V), and tens of thousands of Kconfig options. Trying all the combinations is impossible.
- **Long-lived state.** Memory leaks and counter overflows show up only after the system has run for days.

The kinds of tests are layered according to these difficulties (Table 14.14).

Kind	What does it test? (kernel example)
Unit	A single function or module, in isolation (KUnit)
Integration	Subsystems working together, the user-space interface (kselftest)
System	The whole operating system, on a real or virtual machine (LTP, boot tests)
Regression	Has a fixed bug come back? Rerun on every change (KernelCI)
Stress / load	Overload, long durations, resource scarcity (stress-ng, rcutorture)

Table 14.14: Kinds of tests and their counterparts in the kernel.

In the kernel, a failed test often means losing the machine itself. Tests are therefore run in virtual machines; every test has a timeout, and if the machine crashes it is restarted automatically.

14.8.2 The kernel test pyramid

The test tools in Linux form a pyramid (Figure 14.34). At the base are many fast unit tests; at the top are a few slow but realistic system tests.

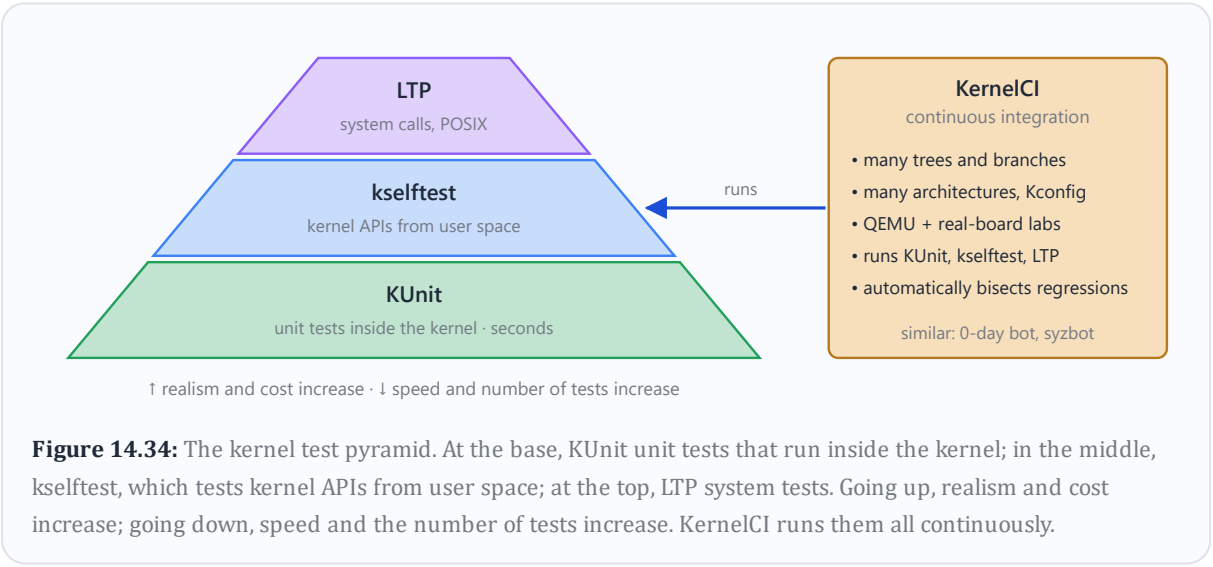


Figure 14.34: The kernel test pyramid. At the base, KUnit unit tests that run inside the kernel; in the middle, kselftest, which tests kernel APIs from user space; at the top, LTP system tests. Going up, realism and cost increase; going down, speed and the number of tests increase. KernelCI runs them all continuously.

- **KUnit.** The tests are compiled together with the kernel and call internal functions directly. They run in seconds under UML (User-Mode Linux) or QEMU; they are started with the `kunit.py run` command. The output is in KTAP format.
- **kselftest.** It lives under `tools/testing/selftests`. It tests the running kernel's system calls and interfaces such as cgroup, bpf, and networking from the outside, from user space; it is run with `make kselftest`.
- **LTP.** The Linux Test Project consists of thousands of tests: the semantics of system calls, POSIX conformance, file-system and stress scenarios. Distributions run LTP before a new release.

KernelCI runs these tests as continuous integration on many trees and branches, across many architectures and Kconfig combinations. Besides QEMU, labs of real boards are also used. When a regression is found, KernelCI automatically narrows it down to a single commit by binary search. The 0-day bot and syzbot are similar continuous-testing services.

14.8.3 Dynamic checkers

A test catches a bug only if it can see some symptom of it. Most memory bugs give no symptom: writing one byte past the end of an array often silently corrupts another object, and the system crashes much later in an unrelated place. Dynamic checkers make these silent bugs visible at the moment of the first invalid access (Table 14.15).

Tool	Bug it catches	How it works
KASAN `CONFIG_KASAN`	Use after free; heap, stack, and global array overflow; double free	The compiler inserts a check before every memory access. Shadow memory (1 byte per 8 bytes) records which bytes are valid. The report also gives the allocation and free stack traces.
KCSAN `CONFIG_KCSAN`	Data race: concurrent, lock-free accesses, at least one of which is a write	On random accesses it sets a watchpoint and waits briefly; if another core touches the same address meanwhile, it reports.
UBSAN `CONFIG_UBSAN`	Undefined behavior: signed overflow, division by zero, invalid shift, array bounds	Run-time checks inserted by the compiler.
lockdep `CONFIG_PROVE_LOCKING`	Lock-order violation (AB-BA), wrong lock use in interrupt context	Maintains the “before–after” graph among lock classes; on seeing a cycle, it warns before the deadlock actually happens.
kmemleak `CONFIG_DEBUG_KMEMLEAK`	Memory leak	Scans memory like a garbage collector; reports allocated blocks that no pointer points to.

Table 14.15: The kernel's dynamic checkers.

A typical KASAN report begins with two lines:

```
BUG: KASAN: slab-use-after-free in mydrv_release+0x54/0x90
Read of size 8 at addr ffff888012a4c010 by task cat/1432
```

The first line gives the type of the bug (slab-use-after-free: an access to a freed slab object) and its location: the instruction at byte 0x54 of the function `mydrv_release`. The second line gives the size and address of the access and which task made it: process 1432 of the `cat` command made an 8-byte read. The rest of the report brings two stack traces showing where the object was allocated and where it was freed; the object's lifetime can be read off from them.

The checkers are costly. KASAN slows the kernel down several times over and sets aside about one-eighth of memory for the shadow. KCSAN and lockdep also add noticeable overhead. They are therefore enabled in development, testing, and continuous-integration kernels. In production systems, the sampling, low-cost KFENCE can be used: it checks only occasionally chosen allocations.

14.8.4 Code trace: KASAN shadow memory

Definition 14.24

Shadow memory is the 1 byte of state that KASAN keeps for every 8-byte region of memory. If the value is 0, all 8 bytes of the region are valid. A value k between 1 and 7 means that only the first k bytes are valid. Negative values mean a red zone or freed memory.

The check KASAN makes before every access looks, in simplified form, like this:

```
/* 1 shadow byte per 8-byte region:
   0: all valid, 1..7: first k bytes,
   < 0: redzone or freed */
void kasan_check(void *a, int size) {
    int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
                           + SHADOW_OFS);
    if (s == 0) return;            /* fast path */
    if ((int)((uintptr_t)a & 7) + boy - 1 >= s)
        report(a, s);
}
/* p = kmalloc(13); p[3]; p[12]; p[13]; */
/* kfree(p); p[0]; */
```

The function first finds the shadow byte. The address is shifted right by 3 bits, that is, divided by 8, and a fixed offset is added to the result. If the shadow byte is 0, the whole region is valid and the check ends with a single comparison. This is the fast path, and the great majority of accesses go through it. If the value is not zero, the position of the access's last byte within the region is computed: `(a & 7) + size - 1`. If this position is greater than or equal to the shadow value, the access spills

beyond the valid bytes and a report is produced. It matters that the comparison is signed: because the position is converted to an integer with `(int)`, `s` is also compared as a signed value. A negative shadow value then satisfies this condition at every position; Linux KASAN also makes this check with a signed byte. Without the cast, the comparison would become unsigned because of `uintptr_t`, −5 would turn into a very large number, and freed memory would never be reported.

In the scenario, `p = kmalloc(13)` is called. The kernel's allocator gives a 16-byte slot for a 13-byte request. The shadow is set up as follows: the first 8 bytes are fully valid (0); the first 13 − 8 = 5 bytes of the second region are valid (5); the region after it is a red zone (0xfc, that is, −4). Then `p[3]`, `p[12]`, and `p[13]` are read in turn; after that come `kfree(p)` and `p[0]` (Table 14.16).

Access	Region (a >> 3)	Shadow s	(a & 7) + size − 1	Result
`p[3]`	0	0	—	fast path, valid
`p[12]`	1	5	4	4 ≥ 5 is false, valid
`p[13]`	1	5	5	5 ≥ 5, report
`kfree(p)`	0, 1	0xfb (−5)	—	shadow is marked “freed”
`p[0]`	0	−5	0	0 ≥ −5, report (use after free)

Table 14.16: The KASAN check. One-byte reads (`size` = 1).

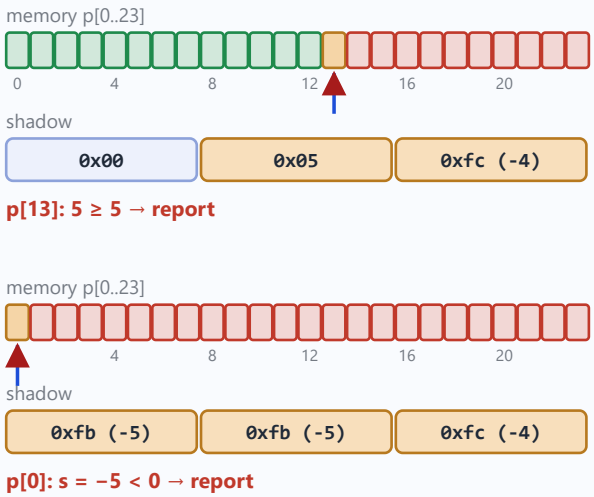


Figure 14.35: (a) `p[13]`: one byte past the end of the array; because 5 ≥ 5, a report is produced. (b) After `kfree(p)`, the object's shadow is 0xfb; the read of `p[0]` is reported as a use after free.

`p[12]` is the last valid byte of the 13-byte object and passes without trouble. `p[13]`, however, is one byte past the end of the array. In a real kernel this byte might belong to the slot's padding or to another object; without KASAN the bug would stay silent. KASAN reports it at the first invalid access.

`kfree(p)` marks the object's shadow as 0xfb (freed). The slot is held in **quarantine** for a while; if it were reused immediately, the new object's shadow would be valid and an access through the old pointer would be hidden. When `p[0]` is read, the shadow is −5, and the report comes with the stack

traces of where the object was allocated and freed.

Common mistake

Thinking that KASAN catches only code that crashes. KASAN's value is just the opposite: the read of `p[13]` gives no symptom at all on a kernel without KASAN. The checker catches, at the first moment, a bug that would have stayed silent. Another mistake is forgetting multi-byte accesses: a 4-byte read can spill into invalid bytes even if its start address is valid. That is why the formula has the term `size - 1`.

14.8.5 Code trace: the lockdep lock-order graph

The most common cause of deadlocks is two locks being taken in opposite orders in different places. If one thread holds A and wants B while another holds B and wants A, the two may wait forever. lockdep finds this situation before the deadlock occurs.

Definition 14.25

lockdep is the Linux kernel's lock validator. It tracks lock classes, not lock instances, and accumulates the relation "L was acquired while H was held" as an edge H → L in a directed graph. If a new edge closes a cycle in the graph, it warns of a possible deadlock.

```
void lock_acquire(task *t, int L) {
    for (int i = 0; i < t->n; i++) {
        int H = t->held[i];
        if (path_exists(L, H))   /* L → … → H */
            warn("possible deadlock", H, L);
        add_edge(H, L);    /* H before L */
    }
    t->held[t->n++] = L;
}
void lock_release(task *t) { t->n--; }
/* T1: A,B   T2: B,C   T3: C,A */
```

`lock_acquire` is called when a task acquires lock `L`. For each lock `H` that the task is holding at that moment, two things are done. First, the graph is checked for a path from `L` to `H`. If such a path exists, then somewhere `H` has been acquired (perhaps indirectly) while `L` was held, and now the reverse is being done: a warning is given. Then the edge `H → L` is added. Finally `L` is put in the list of held locks. `lock_release` removes the last lock from the list.

Three tasks run at different times: T1 takes A and then B, T2 takes B and then C, T3 takes C and then A (Table 14.17).

Event	Held	Path check	Edge added	Graph
T1: acquires A	—	loop does not run	—	empty
T1: acquires B	A	B → … → A? none	A → B	A → B
T2: acquires C (while holding B)	B	C → … → B? none	B → C	A → B → C
T3: acquires C	—	loop does not run	—	same
T3: acquires A (while holding C)	C	A → B → C exists: **warning**	C → A	A → B → C → A (cycle)

Table 14.17: Trace of lockdep.

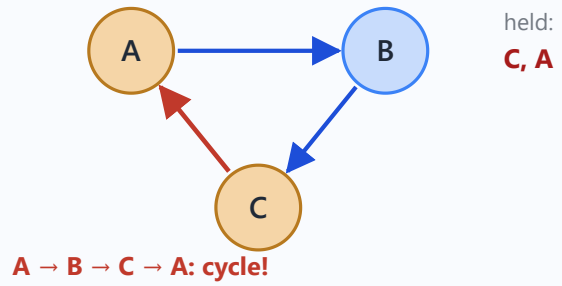


Figure 14.36: T3 asks for A while holding C. The graph has the path A → B → C; the new edge C → A closes a cycle and lockdep warns.

The three tasks never ran at the same time, and no real deadlock occurred. But if one day T1 holds A, T2 holds B, and T3 holds C at once, each waits for the next one's lock: T1 for B, T2 for C, T3 for A. This circular wait is a deadlock. lockdep shows it in a single run, before it happens. Its power comes from this: it turns a rare, timing-dependent bug into a deterministic graph check.

Common mistake

Looking only for the two-lock AB-BA order. The cycle in the example consists of three locks, and no task takes two locks in reverse order: T1 takes A–B, T2 takes B–C, T3 takes C–A. The problem is seen only when the three relations are combined. This is why lockdep's path search in the graph is necessary.

14.8.6 Fuzzing: the syzkaller loop

Definition 14.26

Fuzzing means looking for crashes and bugs by running a program on a great many random or semi-random inputs. In **coverage-guided** fuzzing, inputs that open up a new code path are saved and become seeds for later attempts.

The most productive bug finder for the Linux kernel is syzkaller (Figure 14.37). The loop works in six steps.

1. **Descriptions.** syzkaller knows the signatures of the system calls from syzlang descriptions: which argument is a file descriptor, which is a flag, which is a pointer. The corpus is initially empty or consists of seed programs.
2. **Generation and mutation.** The generator picks a program from the corpus and modifies it, or produces a new one; for example `openat` → `ioctl` → `mmap` → `close`. Because the arguments are chosen to suit their types, the calls are not immediately rejected with `EINVAL`.
3. **Execution.** The program runs in QEMU virtual machines running a test kernel with KASAN, KCSAN, and lockdep enabled. Even if the VM crashes, the host is unaffected; syz-manager restarts the VM.
4. **Coverage.** KCOV records the basic blocks (PC values) passed in the kernel. This feedback turns random trial into a guided search.
5. **Corpus.** A program that opens a new code path is added to the corpus and becomes a seed for later mutations. The loop turns without stopping, with hundreds of programs per second on every VM.
6. **Report.** If a crash or a KASAN report arrives, syzkaller reproduces the program, reduces it to its smallest form, and often produces a C reproducer. syzbot reports this to the developers automatically.

Figure 14.37: The syzkaller loop: corpus, generation and mutation (syzlang call descriptions), execution in a QEMU VM on a KASAN-enabled test kernel, KCOV coverage feedback. If a crash occurs, it is reproduced, minimized, and reported.

The fuzzer and the checker complement each other. A fuzzer alone sees only crashes. A bug that corrupts memory but does not crash looks, to the fuzzer, like a successful run. Checkers such as KASAN make these silent corruptions visible at the first invalid access and turn them into crashes. This is why syzkaller is run with a KASAN-enabled kernel.

14.8.7 Observation and tracing tools

Once a bug has been found, you need to see inside the system to understand its cause. Linux offers tools layer by layer for this (Table 14.18).

Tool	What it provides	Example
`printk`	Log levels 0 (EMERG) … 7 (DEBUG); written to a ring buffer, and those above the threshold are printed on the console	`pr_warn("queue full: %d\n", n);`
`dmesg`	Reads the ring buffer; if `kernel.dmesg_restrict=1`, only holders of `CAP_SYSLOG` can read it (prevents address leaks)	`dmesg -w --level=err,warn`
dynamic debug	Turns on `pr_debug` lines at run time without recompiling	`echo 'module e1000e +p' > …/dynamic_debug/control`
ftrace	In-kernel tracer: function and function-graph tracing, tracepoint events (tracefs)	`trace-cmd record -p function_graph -g vfs_read`
perf	Hardware counters and sampling profiler: hot spots, cache misses, scheduler latency	`perf record -g -- ./app`, `perf report`
eBPF / bpftrace	Small programs that pass the verifier are attached to kprobes and tracepoints; safe observation in production	one-line example in the text
kdump / crash	On a panic, switches to a backup kernel with kexec and saves a memory dump (`/proc/vmcore`); the dump is examined offline later	`crash vmlinux vmcore`

Table 14.18: Observation and tracing tools.

With bpftrace, a one-line program shows live which process is opening which file on the system:

```
# which process opens which file?
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm,
str(args.filename)); }'
```

The program attaches to the `sys_enter_openat` tracepoint. It runs every time an `openat` system call begins and prints the process name (`comm`) and the name of the file to be opened. There is no need to recompile the kernel or reboot the machine; because the program passes the verifier, it can be used safely even on a running production system.

The general rule in choosing a tool is observation before guessing: logs and traces let you form a hypothesis. But a heavy `printk` can change the timing and hide a race bug. Low-overhead tracers, ftrace and eBPF, are therefore preferred.

14.8.8 Kernel debugging with QEMU and gdb

Sometimes traces are not enough and the kernel must be run step by step. The easiest way is to run the kernel in QEMU and attach gdb from outside.

```
# terminal 1: -s → gdb server :1234
#             -S → CPU halted at start
qemu-system-x86_64 -kernel bzImage \
  -append "console=ttyS0 nokaslr" \
  -nographic -s -S
# terminal 2
gdb vmlinux
(gdb) target remote :1234
(gdb) hbreak start_kernel
(gdb) continue
(gdb) bt
(gdb) lx-dmesg  # kernel gdb script
```

In the first terminal, QEMU starts the kernel. The `-s` option opens a gdb server on port 1234; `-S` halts the CPU at the very beginning, so that one can attach before even the first instruction. `console=ttyS0` redirects the kernel's output to the serial line, that is, to the terminal. `nokaslr` turns off kernel address space randomization, so that the symbols in gdb match the real addresses. In the second terminal, gdb is opened with the uncompressed `vmlinux` file that contains symbol information (`CONFIG_DEBUG_INFO`). `target remote :1234` connects to QEMU. `hbreak start_kernel` puts a hardware breakpoint at the kernel's entry point in C. `continue` runs up to it, and `bt` shows the call stack. `lx-dmesg` is one of the kernel's own gdb scripts and prints the log buffer. The same session can also be set up on real hardware over a serial line with kgdb (`kgdboc=ttyS0,115200`).

In a real bug, the first thing to look at is the oops message:

```
BUG: kernel NULL pointer dereference,
     address: 0000000000000010
Oops: 0000 [#1] PREEMPT SMP NOPTI
CPU: 2 PID: 1432 Comm: cat
Tainted: G           O  6.12.0
RIP: 0010:mydrv_read+0x2a/0x80 [mydrv]
Call Trace:
 vfs_read+0xa4/0x2f0
 ksys_read+0x63/0xe0
 do_syscall_64+0x82/0x170
```

The message is read line by line. The first line gives the type of the bug: a NULL-pointer access at address 0x10. This is typically an access, through a NULL structure pointer, to a field 16 bytes from the start of the structure. `CPU: 2 PID: 1432 Comm: cat` says on which core and by which process the bug was triggered. `Tainted: G O` shows that the kernel is “tainted”; `O` means an out-of-tree module is loaded. `RIP: mydrv_read+0x2a/0x80` is the location of the faulty instruction: 0x2a bytes from the start of the function `mydrv_read` , whose size is 0x80. `scripts/faddr2line` finds the line in

the source file from this information. The call trace is read from bottom to top: `do_syscall_64` → `ksys_read` → `vfs_read` → the driver. That is, a `read` system call came through the VFS to the driver and crashed in the driver.

An oops kills only that process; the system may keep running, but it is no longer trustworthy. The setting `panic_on_oops=1` turns an oops directly into a panic. In the Windows world the counterpart is WinDbg: debugging is done in kernel mode over a network, USB, or a virtual machine, and blue-screen dumps are examined with `!analyze -v`. Driver Verifier (`verifier.exe`) stresses drivers with special-pool, IRQL, and resource-leak checks and so catches bugs early.

14.8.9 Hunting regressions: git bisect

If a bug is absent in v6.11 and appears in v6.12, there is a systematic way to find the change that brought it. The commits between the two releases form a sequence, and after some point in the sequence the bug is present. Finding this point is a binary-search question.

Definition 14.27

A **regression** is the breaking of a feature that used to work correctly after a new change. `git bisect` is the command that finds the first commit that brought in the bug by binary search between a known good and a known bad commit.

```
git bisect start
git bisect bad v6.12
git bisect good v6.11
# #500: build, boot, test
git bisect bad
git bisect good   # #250
git bisect good   # #375
git bisect bad    # #437
git bisect run ./test.sh
git bisect reset
```

The commands do the following. `git bisect start` begins the search. `bad v6.12` and `good v6.11` declare the two ends. Git checks out the commit in the middle of those between them. At each step the kernel is built, booted, and tested; depending on the result, `git bisect bad` or `git bisect good` is entered. After some point the testing can be left to a script: `git bisect run ./test.sh` carries out the remaining steps by itself. When it finishes, `git bisect reset` returns to the starting branch.

Example 14.12

There are 1000 commits between v6.11 (good) and v6.12 (bad), and the first bad commit is #412. At each step Git tests the middle of the range, ⌊(good + bad) / 2⌋. Which commits are tested?

1. **#500.** The bug is present: bad. The first bad commit is #500 or earlier; candidate 500.
2. **#250.** The bug is absent: good. The first bad commit is between 251 and 500; candidate 250.
3. **#375.** Good; candidates 376–500, that is, 125.
4. **#437.** Bad; candidates 376–437, that is, 62.
5. **git bisect run.** The script carries out the rest: 406 good, 421 bad, 413 bad, 409 good, 411 good, 412 bad.
6. **Result.** "#412 is the first bad commit." 10 tests in total.

Figure 14.38: git bisect: 1000 commits between v6.11 (good) and v6.12 (bad). Each test halves the candidate range; the first bad commit, #412, is found in 10 tests.

Each answer halves the number of candidates: 1000 → 500 → 250 → 125 → 62 → … The number of tests needed therefore grows with the logarithm of the number of commits.

Theorem 14.2 (Test bound of binary search)

Among n candidate commits, the first bad commit is found by binary search in at most $\lceil \log_2 n \rceil$ tests.

For 1000 commits, $\lceil \log_2 1000 \rceil = 10$, because $2^9 = 512 < 1000 \le 1024 = 2^{10}$. Even with a million commits, only $\lceil \log_2 10^6 \rceil = 20$ tests would be needed. For 12,000 commits the bound is 14: $2^{13} = 8192 < 12{,}000 \le 16{,}384 = 2^{14}$.

`git bisect run` looks at the script's exit code. If the script exits with 0, the commit is considered good; if it exits with a code between 1 and 127 (except 125), bad. 125 has a special meaning: "this commit cannot be tested, skip it." For example, if the kernel does not build at that commit, the script returns 125 and git tries a neighboring commit. For performance regressions, the measurement is noisy, so the script should average several measurements.

Common mistake

Returning 1 from the script for a commit that cannot be tested. If a commit that does not build is counted as "bad," the binary search goes into the wrong half and in the end the wrong commit is blamed. For a version that does not build or does not boot, the right answer is 125.

14.9 Trends in operating systems

Because operating systems live long, today's decisions must also serve the hardware and applications of decades from now. This part gathers the trends that most affect design today.

14.9.1 The big picture

Trend	Effect on the operating system
Virtualization and the cloud	The guest kernel cooperates with the hypervisor through virtio; in a multi-tenant cloud, the isolation boundary moves to the hypervisor. Containers are light but weakly isolated; micro-VMs (Firecracker) offer VM isolation and millisecond startup.
Multicore chips	Scalable locks, RCU, NUMA awareness, heterogeneous cores
Wide address space	64 bits, 5-level page tables; single-address-space system ideas
Seamless data access	Transparent access among the network, cloud storage, and local disk; NVMe, CXL memory
Battery-powered systems	DVFS, "finish fast and sleep" (race-to-idle), Android Doze, energy-aware scheduling (EAS)
Embedded and sensor nodes	Small real-time kernels: Zephyr, FreeRTOS, Contiki-NG; kilobytes of memory
Safe languages and programmable kernels	Rust in Linux (since 6.1), eBPF and sched_ext, unikernels

Table 14.19: Trends and their effects on the operating system.

Networking, parallel and distributed systems, and multimedia, which were separate trends in Tanenbaum's older editions, are now ordinary parts of every system. They are therefore considered not as separate trends but within the relevant rows of the table.

The virtualization row is a redrawing of the isolation–performance trade-off. In a multi-tenant cloud, the code of different customers runs on the same machine, and the real boundary between them is no longer the operating-system kernel but the hypervisor. Containers are light and fast because they share the same kernel, but a hole in the kernel affects all containers. Micro-VMs such as Firecracker choose a point between the two worlds by giving each workload a separate, very small virtual machine.

In battery-powered systems the goal is energy. DVFS (dynamic voltage and frequency scaling) changes the processor's voltage and frequency according to the load. In the “finish fast and sleep” strategy, the work is finished at top speed and the processor is immediately put into a deep sleep state. Android Doze defers applications' background work to batched windows while the screen is off.

14.9.2 Many cores and wide address spaces

Today servers have 128–192 cores per socket, and phones have chips mixing big and little cores (big.LITTLE). A single big kernel lock does not work at this scale: if all cores wait for the same lock, every added core is wasted. The tools needed are fine-grained locks, RCU (read-copy-update), in which readers take no lock at all, and separate data structures for each core. With heterogeneous cores, the scheduler also decides which kind of core each job goes to.

64-bit addresses open up new possibilities too (Figure 14.39).

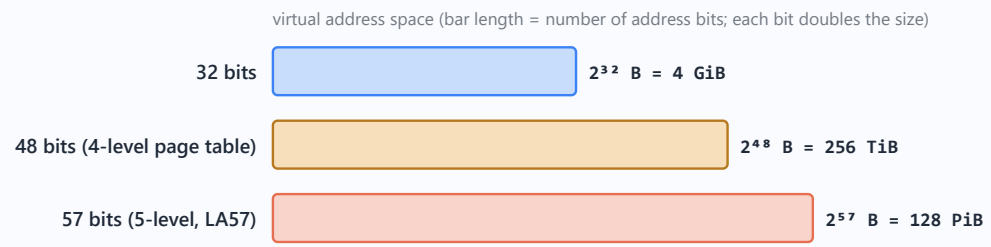


Figure 14.39: Virtual address space sizes: 32 bits 4 GiB, 48 bits (4-level page table) 256 TiB, 57 bits (5-level, LA57) 128 PiB. Each bit doubles the size.

Example 14.13

How large are 48-bit and 57-bit virtual address spaces, and how do these bit counts follow from the page-table levels?

1. **48 bits.** 2^{48} B = $2^{8} \times 2^{40}$ B = 256 TiB.
2. **57 bits.** 2^{57} B = $2^{7} \times 2^{50}$ B = 128 PiB.
3. **Levels.** In a 4 KB page the offset is 12 bits. Each table level has 512 entries and resolves 9 bits. 4 levels: 4 × 9 + 12 = 48 bits; 5 levels: 5 × 9 + 12 = 57 bits.

In an address space this wide, it becomes possible to map all files into memory (`mmap`) or to build systems in which all processes share a single address space. The price is a deeper page-table walk: in a 5-level table, a TLB miss needs five memory reads. A wide address space also increases the pressure on the TLB.

14.9.3 Rust, eBPF, and unikernels

The last trend concerns the language in which the kernel is written and in what form. Three developments stand out.

The kernel in Rust. Rust support entered the mainline with Linux 6.1 (December 2022), followed by the first Rust drivers. The aim is to prevent memory bugs such as use after free and overflow at compile time. A significant share of kernel vulnerabilities are bugs of this kind. Rust's ownership and borrowing rules let the compiler check an object's lifetime. This means preventing, while the code is being compiled, the bugs that KASAN catches at run time. The Windows kernel also has components written in Rust.

eBPF. Extending the kernel with small verified programs without recompiling it; it is used for tracing, networking (XDP), security, and scheduling (sched_ext). The idea of the extensible system has gone into production at large scale for the first time with eBPF.

Unikernel. This is the cloud version of the exokernel's library operating system (libOS) idea. A single application is built, together with the operating-system libraries it needs, as a single image and runs in a single address space on a hypervisor (Figure 14.40). MirageOS and Unikraft are examples of this approach.

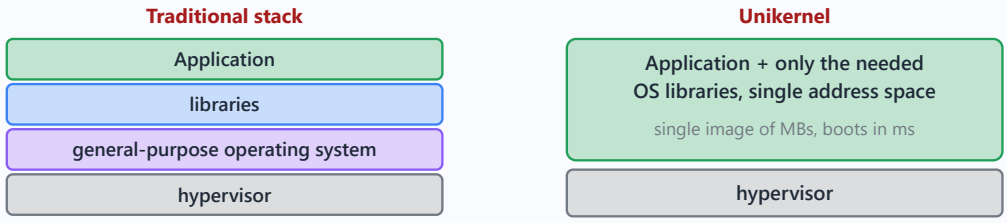


Figure 14.40: In the traditional stack, the application, the libraries, and a general-purpose operating system sit on top of the hypervisor. A unikernel combines the application with only the operating-system libraries it needs into a single image, with one address space, of a few megabytes; booting takes milliseconds.

The gain of a unikernel is smallness and speed. Because the image contains only the code that is needed, the attack surface is small and booting takes milliseconds. Because there is no user–kernel transition, system calls become ordinary function calls. It suits single-application microservices. The price is harder debugging and the impossibility of running more than one application in the same image. Isolation is now provided by the hypervisor.

14.10 Summary

- Eight factors make operating-system design hard: sheer size, concurrency, hostile users, sharing despite isolation, long life, generality, portability, and backward compatibility.
- The four basic goals of an operating system are to define abstractions, to provide primitive operations (system calls) on those abstractions, to provide isolation, and to manage the hardware.
- A good interface is simple, complete, and efficient. The kernel holds few, general calls, and the library holds convenience functions: the six `exec` functions reduce to a single `execve`. The number of system calls is ≈ 12 in seL4 and ≈ 460 in Linux.
- In algorithmic code the program determines the flow; in event-driven code, external events do. The event loop is built around a blocking `get_message`; one long handler freezes the whole interface. In UNIX everything is a file, in Windows everything is an object.
- The basic trade-off in system structure is between isolation and performance. A monolithic kernel is fast but has no fault isolation; a microkernel contains faults but pays IPC on every request. The layer order follows from the dependencies. eBPF provides safe extension thanks to the verifier.
- A mechanism determines how something is done, a policy what is done; sched_ext loads the whole policy as an eBPF program. Orthogonality combines separate concepts independently: the `clone` flags give every combination between `fork` and `pthread_create`.
- Two-level naming maps an external name to an internal number and lets an object be moved without changing its name. Resolving `/usr/ast/mbox` without a cache needs 7 disk reads. Operating systems generally choose late binding.
- Static tables are simple but cannot grow when full; dynamic structures are flexible but need an allocator and locking. Implementation is done bottom-up, design top-down. Synchronous communication is easy, asynchronous communication scales.
- Techniques: gathering hardware-dependent code in a few files, indirection, reuse, reentrancy (no shared `static` state; `strtok_r`), brute force, and checking for errors first.
- Measurement decides what to speed up; Amdahl's law gives the limit of the gain. The space-time trade-off (bit-count table, color palette), caches (always correct), hints (may be wrong, verified before use), locality, and optimizing the common case (futex) are the classic techniques.
- Brooks's law: adding staff to a late project makes it later; the number of communication paths grows quadratically as $n(n-1)/2$. Writing code is only a sixth of the work, and there is no silver bullet.
- Kernel testing is layered with KUnit, kselftest, and LTP. KASAN, KCSAN, UBSAN, lockdep, and kmemleak make silent bugs visible; syzkaller triggers them with coverage-guided fuzzing. ftrace, perf, and eBPF provide observation, QEMU and gdb step-by-step debugging, and `git bisect` hunts regressions in at most $\lceil \log_2 n \rceil$ tests.
- Trends: virtualization and the cloud, many cores, wide address spaces, energy efficiency, embedded systems, Rust, eBPF, and unikernels.

14.11 Key terms

abstraction primitive isolation system call simplicity, completeness, efficiency event-driven code event loop monolithic kernel microkernel layered system exokernel extensible system eBPF (extended Berkeley Packet Filter) kernel thread policy / mechanism orthogonality external name, internal name binding time stub synchronous / asynchronous communication conditional compilation indirection reentrancy brute force space-time trade-off cache hint locality futex (fast userspace mutex) Brooks's law shadow memory fuzzing regression unikernel

14.12 Test Yourself

1. The handlers in an event loop keep a counter: `KEY` adds 1, `CLICK` adds 10; `TIMER` resets the counter if it is greater than 10. If the events arrive in the order CLICK CLICK TIMER KEY KEY KEY TIMER QUIT, what is the counter when the loop ends?

   Answer

   3. The counter goes 10, 20, 0 (20 > 10), 1, 2, 3. At the last `TIMER`, 3 > 10 does not hold, so the counter does not change. At `QUIT`, `get_message` returns 0 and the loop ends.

2. The path cache has the entry `/var` → 30 and no other relevant entries. How many disk reads does `resolve()` count for `/var/log/syslog`? How many would it count if the cache were completely empty? What are the totals if the file's own i-node is also read?

   Answer

   With `/var` in the cache, it stops at `k` = 1; `log` and `syslog` are resolved from disk: 2 + 2 = 4 reads. With the cache empty, `k` drops to 0 and the three components are resolved from the root: 6 reads. If the file's i-node is counted too, it becomes 5 and 7.

3. With the `nib` table, which two entries does `bit_cnt(0xE3)` read, and what does it return?

   Answer

   0xE3 = 1110 0011. The low half is 3 and the high half is E. `nib[3]` = 2 and `nib[E]` = 3; the result is 5. The loop version reaches the same result in eight iterations.

4. A 1920 × 1080 image is stored (a) with 24-bit RGB per pixel, (b) with a 4-bit index per pixel and a

16-color palette (3 bytes per color). What are the sizes and the ratio?

Answer

(a) 1920 × 1080 × 3 = 6,220,800 bytes. (b) Two pixels fit in one byte: 1920 × 1080 / 2 = 1,036,800 bytes, and the palette is 16 × 3 = 48 bytes; the total is 1,036,848 bytes. The ratio is about 6. The price is a table lookup for every pixel and the limit of 16 colors.

5. 40% of a program's running time is spent in the function `compress()`. One option is to make this function 4 times faster, the other to make the rest of the program 2 times faster. Which pays off more?

Answer

They are equal. In the first, the time becomes 60 + 40/4 = 70; in the second, 40 + 60/2 = 70; the speedup in both cases is 100 / 70 ≈ 1.43. By Amdahl's law: 1 / (0.6 + 0.1) = 1 / (0.4 + 0.3) ≈ 1.43. How much a part can be sped up matters as much as its share.

6. A team of 12 adds 4 more people to a late project. From how many to how many does the number of communication paths rise? What do we see when we compare this with the increase in headcount?

Answer

It rises from 12 · 11 / 2 = 66 paths to 16 · 15 / 2 = 120 paths: about 1.82 times. The headcount has grown only 16 / 12 ≈ 1.33 times. Training the newcomers also takes time from the existing team; these are the two causes of Brooks's law.

7. For `git bisect`, #0 is good and #64 is bad; the first bad commit is #45. At each step Git tests ⌊(good + bad) / 2⌋. Which commits are tested? In a range of 5000 commits, how many tests are needed at most?

Answer

#32 good, #48 bad, #40 good, #44 good, #46 bad, #45 bad: 6 tests, $\lceil \log_2 64 \rceil = 6$. For 5000 commits, $\lceil \log_2 5000 \rceil = 13$, because $2^{12} = 4096 < 5000 \le 8192 = 2^{13}$.

8. `p = kmalloc(20)` yields a 32-byte slot. What do the KASAN shadow bytes become? Which of the

4-byte reads `p[8]`, `p[16]`, and `p[18]` is reported?

Answer

The shadow: 0, 0, 4, 0xfc. The first two regions are fully valid, the first 20 − 16 = 4 bytes of the third region are valid, and the fourth is a red zone. `p[8]`: shadow 0, fast path. `p[16]`: (16 & 7) + 4 − 1 = 3, 3 ≥ 4 is false, valid. `p[18]`: 2 + 4 − 1 = 5 ≥ 4, report: the read covers bytes 18–21, and 20 and 21 lie outside the object.

9. Four tasks run at different times: T1 takes A and then B, T2 takes C and then D, T3 takes B and then C, T4 takes D and then A. At what moment does lockdep warn? What would happen if T4 took A and then D?

Answer

The edges are A → B, C → D, B → C in turn. When T4, holding D, asks for A, the path A → B → C → D exists, and the new edge D → A closes a cycle: the warning comes when T4 takes A. If T4 took A first and then D, the edge A → D would be added; no cycle would form and no warning would be given.

10. Which of the following are mechanisms and which are policies? (a) The cpufreq driver changing the processor frequency, (b) the `schedutil` governor choosing the frequency according to load, (c) the LRU replacement rule, (d) the MMU walking the page table on a TLB miss, (e) SELinux rules.

Answer

Mechanisms: (a) and (d); they determine how something is done and are independent of which decision is made. Policies: (b), (c), and (e); they decide what is to be done and can be changed on top of the same mechanism.

14.13 Further reading

- B. W. Lampson, “Hints for Computer System Design”, ACM SOSP, 1983: the classic collection of short pieces of advice on interfaces, implementation, and performance; the “make the common case fast” rule and the idea of a hint come from here.
- F. P. Brooks Jr., *The Mythical Man-Month*, 20th anniversary edition, 1995: Brooks's law, the surgical team, the second-system effect, and the essay “No Silver Bullet” are collected in a single book.
- U. Drepper, “Futexes Are Tricky”, 2011: explains step by step how to write a futex-based lock correctly, why the states 0–1–2 are used, and the common mistakes.
- Linux kernel documentation, `docs.kernel.org/dev-tools/`: gives, in the kernel's own documentation, how to enable and use KASAN, KCSAN, UBSAN, kmemleak, KUnit, and kselftest.

14.14 References

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 12: Operating System Design.
2. B. W. Lampson, "Hints for Computer System Design", ACM SOSP, 1983 (and the 2020 update, "Hints and Principles for Computer System Design").
3. F. P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary edition, Addison-Wesley, 1995 — including "No Silver Bullet" (1986).
4. J. H. Saltzer, M. F. Kaashoek, *Principles of Computer System Design: An Introduction*, Morgan Kaufmann, 2009.
5. D. R. Engler, M. F. Kaashoek, J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", SOSP 1995.
6. G. Klein et al., "seL4: Formal Verification of an OS Kernel", SOSP 2009; J. N. Herder et al., MINIX 3 reliability studies.
7. Linux kernel documentation: Rust (docs.kernel.org/rust), BPF, and sched_ext; H. Franke et al., "Fuss, Futexes and Furwocks", 2002.

Linux

How do the general concepts of the course (processes, threads, scheduling, virtual memory, file systems, and protection) look in a real system? This chapter treats Linux as a case study. It does not redefine the concepts; it concentrates on the structures and decisions specific to Linux. It starts with the history from UNIX to Linux and with how the shell builds pipelines. Then it descends into the kernel: `task_struct`, the `clone()` flags, signal delivery, fair scheduling from CFS to EEVDF, four- and five-level page tables, the buddy and SLUB allocators, the page cache, and the OOM killer. For I/O and file systems it follows device files, blk-mq, the VFS, and ext4 journaling. The security section covers permission arithmetic, capabilities, and namespaces and cgroups, the foundation of containers. The chapter closes with Android.

LEARNING OUTCOMES

History and family
Place the line from UNIX through BSD, MINIX, GNU, and Linux; distributions, the release model, and Android.

Interfaces and the shell
Distinguish the library, system-call, and kernel layers; trace how redirections and pipelines are built with `pipe`, `fork`, `dup2`, and `exec`.

Processes and scheduling
Explain the kernel steps of `fork` and `execve`, signal delivery, `task_struct`, the `clone()` flags, and NPTL; compute eligibility and virtual deadlines in EEVDF.

Memory
Interpret the address-space layout and VMAs, the four-level page-table walk, zones, the buddy and SLUB allocators, the page cache, and the OOM killer.

I/O and files
Recognize device files, blk-mq, and sockets; explain the VFS objects, ext4's extent and journaling design, the link count, and `/proc`.

Security
Compute rwx, setuid, and umask; apply the first-matching-class rule; recognize capabilities, namespaces, cgroups, seccomp, LSMs, and the Android sandbox.

15.1 The roots of Linux

Throughout the course, the concepts of process, thread, scheduling, virtual memory, file system, and protection were presented in general terms. This last chapter is a **case study**: it looks at how the same concepts are implemented in a widely used real system. The system is Linux, the most widely deployed kernel, from servers to phones and from supercomputers to embedded devices. Its source code is open, so the name and field of every structure can be read directly. For that reason this chapter does not redefine concepts. It focuses on Linux-specific detail: `task_struct`, the `clone()` flags, the EEVDF scheduler, the SLUB allocator, namespaces, and cgroups.

15.1.1 From UNIX to Linux

The story of Linux begins with UNIX (Figure 15.1). In 1969 Ken Thompson and Dennis Ritchie wrote a small operating system on a PDP-7 at Bell Laboratories. The system was ported to the PDP-11 in 1971, and in 1973 the kernel was rewritten in C. That step was unprecedented: operating systems had always been written in assembly language. UNIX written in C could be moved to a new machine simply by porting the compiler. Portability is the real reason UNIX spread.

AT&T, bound by an antitrust decree, was not allowed to sell computers. It licensed the source code to universities for a small fee. The University of California at Berkeley improved this code and distributed it as **BSD** (Berkeley Software Distribution). Virtual memory and the TCP/IP stack entered UNIX through BSD. AT&T's own line became System V. In the 1980s every vendor shipped its own UNIX: SunOS, AIX, HP-UX, Xenix. As these versions diverged, running one program on all of them became hard. In 1988 the IEEE published the **POSIX** standard. POSIX is not an implementation; it defines the common interface of system calls and library functions.

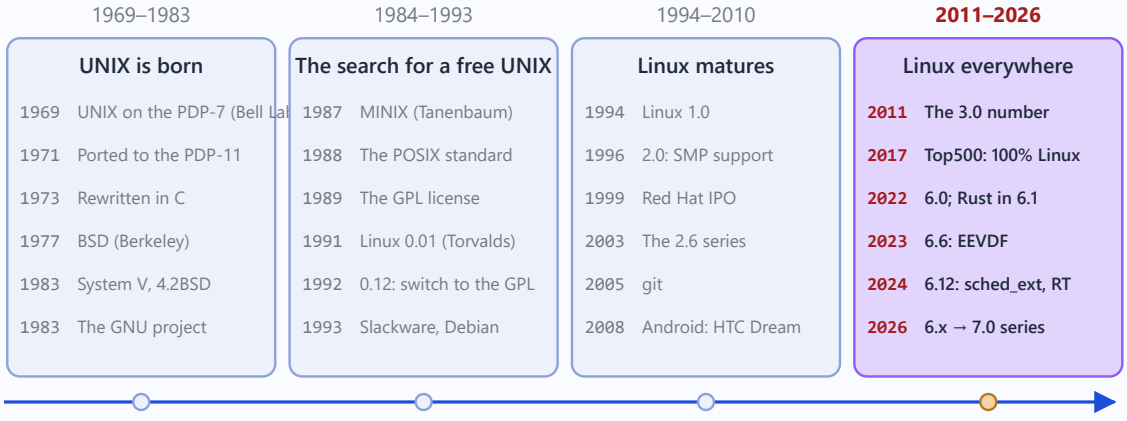


Figure 15.1: Four eras from UNIX to Linux: the birth of UNIX, the search for a free UNIX, the maturing of Linux, and Linux everywhere.

The second half of the 1980s was spent searching for a free UNIX. Richard Stallman started the **GNU** project in 1983. The goal was a completely free, UNIX-compatible system. The project produced the compiler (GCC), the shell (bash), the C library, and hundreds of tools, but it never finished a kernel. In 1987 Andrew Tanenbaum wrote **MINIX** for teaching. MINIX was a small microkernel, and its code was published together with a textbook. In 1991 Linus Torvalds, a student at the University of Helsinki, began developing his own kernel on top of MINIX. Linux 0.01 was released the same year. In 1992, with version 0.12, Torvalds moved the kernel to the GPL license. The GPL requires that modified code be distributed with the same freedoms. When the GNU tools and the Linux kernel were combined, a usable, completely free system emerged. The first distributions, such as Slackware and Debian, appeared in 1993.

Linux contains no UNIX code. It implements the POSIX interface with freshly written code. Its design, however, is the opposite of MINIX's. Torvalds chose a **monolithic** kernel instead of a microkernel. In 1992 this choice led to a famous e-mail debate between Tanenbaum and Torvalds. Tanenbaum considered the monolithic design obsolete; Torvalds defended simplicity and speed. Today's Linux is monolithic, but it gained flexibility through a modular structure.

Over the next fifteen years the kernel matured. Linux 1.0 appeared in 1994. Version 2.0 (1996) supported multiprocessor machines, that is, symmetric multiprocessing (SMP). The 2.6 series (2003) brought the preemptible kernel and new schedulers. In 2005 the license of BitKeeper, the commercial version-control tool used by kernel developers, became a problem. Torvalds wrote `git` in a few weeks; most of the software world uses it today. The first Android phone, the HTC Dream, went on sale in 2008. Since version 3.0 in 2011, version numbers carry no technical meaning; they merely grow. The release after 6.19 is 7.0, published in April 2026. Since November 2017, all of the 500 fastest supercomputers in the world run Linux. The 6.x series brought Rust support (6.1), the EEVDF scheduler (6.6), and the extensible scheduler `sched_ext` together with the real-time patch (6.12).

15.1.2 GNU, distributions, and the release model

Strictly speaking, the word “Linux” names only the kernel. What a user installs on a computer is a **distribution** (Figure 15.2). A distribution adds layers on top of the kernel: the GNU tools and the C library (mostly glibc), the init program that starts the system (today mostly systemd), a package manager, and applications. This is why the name “GNU/Linux” is also used for the complete system. Android uses the same kernel but not the GNU tools; its C library is Bionic.

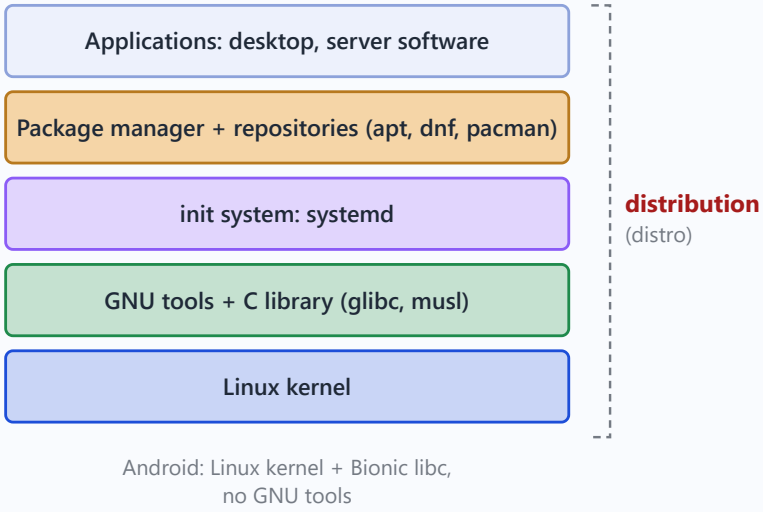


Figure 15.2: The layers of a distribution: kernel, GNU tools and C library, init system, package manager, and applications.

Distributions are grouped into families by package format (Table 15.1). Members of the same family use the same package manager. Pardus, developed in Türkiye, belongs to the Debian family. Alpine, on the other hand, uses the small musl library instead of glibc and BusyBox tools bundled in a single binary, which is why it is popular for container images.

Family	Examples	Package format / tool
Debian	Debian, Ubuntu, Mint, Pardus	.deb / apt
Red Hat	Fedora, RHEL, Rocky, Alma	.rpm / dnf
Arch	Arch, Manjaro	pacman
SUSE	openSUSE, SLES	zypper
Lightweight	Alpine (musl + BusyBox)	apk

Table 15.1: The main distribution families.

The kernel's own release model is regular. When a release is published, a two-week **merge window** opens. During this period the new features accumulated by the subsystem maintainers are pulled into the mainline tree. Weekly release candidates follow: rc1, rc2, and so on, usually up to rc7. Only bug fixes are accepted in these weeks. A new major release thus appears about every 9–10 weeks. The last major release of each year receives long-term support (LTS): 6.6 (2023), 6.12 (2024), and 6.18 (2025) are such releases. Distributions usually pick an LTS kernel and follow its security patches for years.

15.1.3 Design principles

UNIX was designed by programmers for themselves. The principles that come from this origin continue in Linux. The system is **simple**: it rests on a small number of well-defined concepts, namely the process, the file, and the pipe. It is **consistent**: a disk file, a terminal, a pipe, and a network socket are all used with the same `read` and `write` calls (Figure 15.3). The saying "everything is a file" expresses this consistency. The system is **flexible**: tools combine over text streams without knowing each other. It shows the programmer little "magic"; a successful command usually prints nothing.

Figure 15.3: Everything is a file: a disk file, a terminal, a pipe, and a socket are all used through the same file-descriptor interface.

Doug McIlroy, who gave UNIX its pipelines, summed up this thinking in three rules. Each program should do one thing well. Programs should be written to work together. Programs should handle text streams, because text is a universal interface. POSIX standardizes this interface. The same source code can be compiled and run on Linux, BSD, and macOS.

15.2 Interfaces and the shell

15.2.1 Three interfaces: the journey of a `printf`

A Linux system can be thought of as a stack of layers (Figure 15.4). Hardware is at the bottom, with the kernel above it. Above the kernel sits the standard C library, and above that the standard utility programs such as the shell, the editor, and the compiler. Users are at the top. There are three interfaces between the layers. The **system-call interface** lies between the library and the kernel. The **library interface** consists of the functions that programs call, such as `printf`, `open`, and `fork`. The **user interface** is the shell and the programs themselves. POSIX defines the library interface, not the system calls. Which function is a system call can vary from system to system.

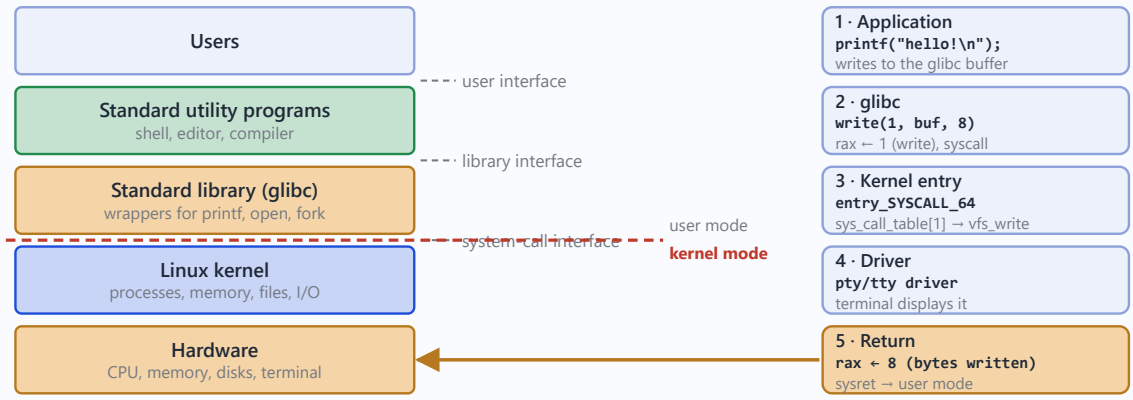


Figure 15.4: The UNIX/Linux layers and the five-step journey of the call `printf("hello!\n")`.

It is instructive to see these layers by following the trace of a single line. Suppose the program calls `printf("hello!\n")`.

1. **Application.** `printf` is not a system call. glibc writes the formatted text into a buffer in user space. If the output is attached to a terminal, the buffer is flushed at the end of the line.
2. **glibc.** When the buffer is flushed, glibc's `write(1, buf, 8)` wrapper runs. On x86-64 the system-call number is placed in the `rax` register; the number of `write` is 1. The arguments go into the `rdi`, `rsi`, and `rdx` registers. Then the `syscall` instruction executes.
3. **Kernel entry.** The CPU switches to kernel mode and jumps to the entry code `entry_SYSCALL_64`. The entry code uses the number as an index into the `sys_call_table` array and calls the handler of `write`. The handler passes to the VFS layer: it finds the object behind file descriptor 1.
4. **Driver.** In an interactive session, descriptor 1 is attached to a pseudo-terminal (pty). The tty layer passes the bytes to the terminal emulator. Had the output been redirected to a disk, the path would have gone to the page cache and the block layer.
5. **Return.** The number of bytes written (8) is returned in `rax`. The `sysret` instruction switches back to user mode. Had there been an error, the kernel would have returned a negative error code; glibc stores it in `errno` and returns −1 to the caller.

There are about 460 system calls on Linux x86-64. All of them enter through the same door: the number in `rax`, the arguments in registers, and entry with `syscall`. Which calls a program makes can be observed with `strace`.

Common mistake

Assuming every library function is a system call. `printf`, `malloc`, `opendir`, and `readdir` are library functions. Some make no call at all; others occasionally rely on an underlying call (`write`, `brk` / `mmap`, `getdents64`). Calling `printf` 1000 times in a loop may produce only a few `write` calls, thanks to buffering.

15.2.2 Commands and redirection

Most commands on Linux are small, single-purpose programs (Table 15.2). Programs called filters read from standard input and write to standard output, which is why they can be chained in a pipeline. Some names are misleading. `touch` updates a file's timestamp; if the file does not exist, it creates an empty one. `sync` writes the kernel's dirty buffers to disk; the tool that synchronizes between two machines is `rsync`. `/proc` is not a command but a pseudo file system: `cat /proc/meminfo` reads the kernel's memory counters. The networking tools have also been renewed: `ip addr`, `ip route`, `ss`, and `nft` replaced the old `ifconfig`, `route`, `netstat`, and `iptables`.

Category	Commands
File / directory	`ls cd pwd mkdir rm -r cp mv touch find tar`
Text (filters)	`cat less head tail grep sort uniq wc echo`
Process / system	`ps top kill chmod chown df du sync`
Network	`ssh scp ping curl wget traceroute ip ss`
Information	`man uname free lsblk`

Table 15.2: Frequently used commands.

Every process finds three open file descriptors when it is born: 0 is standard input, 1 is standard output, and 2 is standard error. Before running the program, the shell can redirect these descriptors to other files:

```
# redirection: 0 = stdin, 1 = stdout, 2 = stderr
sort < input.txt > output.txt 2> errors.log   # < reads, > writes (overwrites)
echo "line" >> log.txt                        # >> appends
make 2>&1 | tee build.log &                   # stderr→stdout, pipe, background
```

On the `sort` line, the shell sets up `sort`'s descriptor table (Figure 15.5). It connects 0 to `input.txt`, 1 to `output.txt`, and 2 to `errors.log`. Then it runs `sort`. `sort` is unaware of the redirection; it simply reads from 0 and writes to 1. `>` overwrites the file, and `>>` appends to it. `2>&1` means “copy 1's current target to 2.” The trailing `&` starts the command in the background.

Figure 15.5: The descriptor table the shell builds for `sort < input.txt > output.txt 2> errors.log`.

Redirections are applied from left to right. The shell turns each one into a `dup2` call. The order therefore changes the result.

Example 15.1

The directory `/nodir` does not exist. Where does the error message go in the commands `ls /nodir /tmp > c.txt 2>&1` and `ls /nodir /tmp 2>&1 > c.txt` ?

1. First command. `> c.txt` : 1 → c.txt. `2>&1` : 2 is copied from 1's current target, that is, c.txt. Both the listing and the error go to the file.
2. Second command. `2>&1` : 1 is the terminal at that moment, so 2 is also attached to the terminal. `> c.txt` : only 1 changes. The listing goes to the file, the error to the screen.
3. `2>&1` does not create a "link"; it copies the current target. Even if 1 changes afterwards, 2 stays at the old target.

15.2.3 The pipeline

A **pipeline** is a chain of commands in which the output of one program is connected to the input of the next. Consider the line `cat log.txt | grep error | sort | uniq -c` (Figure 15.6). The shell parses the line and creates 3 pipes and 4 processes. The four processes run at the same time. Each knows only its own standard input and output.

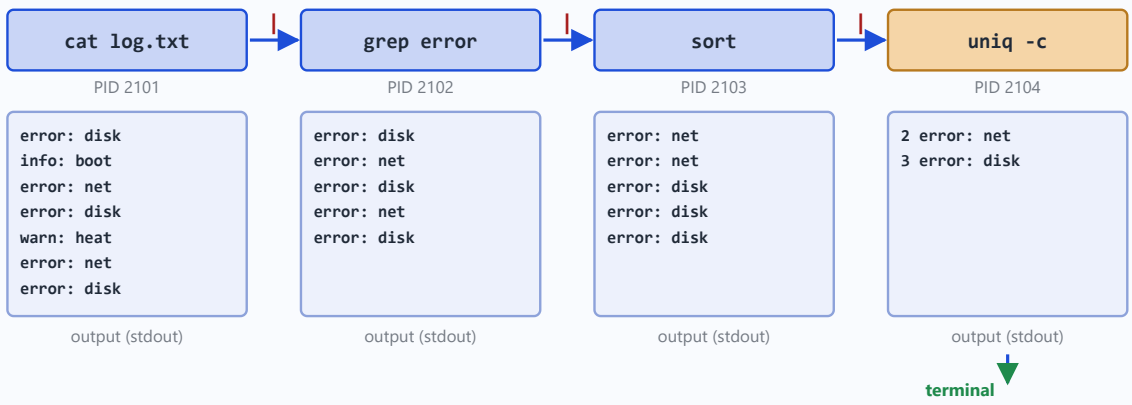


Figure 15.6: `cat log.txt | grep error | sort | uniq -c` : seven lines, five matches, five sorted lines, and two counts.

`cat` writes the 7 lines of the file into the pipe. A pipe's buffer in the kernel is bounded; on Linux the default is 64 KiB. If the buffer fills, the writing process blocks, and it continues as the reading process drains the buffer. This is a ready-made kernel solution to the producer–consumer problem. `grep` reads line by line and forwards the 5 matching lines immediately. A filter is stateless and does not hold the stream back. `sort` is different. It cannot write even a single line before it sees the end of input (EOF), because the last line may move to the first position in the sorted order. `uniq -c` counts only *consecutive* repeats, which is why it needs `sort` in front of it. The result is two lines: 2 × "error: net" and 3 × "error: disk".

Common mistake

Assuming `uniq` merges all repeats. In five unsorted lines (disk, net, disk, net, disk), no line equals the one before it. `uniq -c` prints five separate lines, each preceded by 1. A Python simulation gives the same result.

15.2.4 Code trace: how does the shell build a pipeline?

The shell builds the line `ls | wc -l` with four system calls: `pipe`, `fork`, `dup2`, and `exec`. A simplified version of the shell doing this job looks like this:

```
int fd[2];
pipe(fd);              /* 3: read, 4: write */
if (fork() == 0) {    /* child 1 */
    dup2(fd[1], 1);
    close(fd[0]); close(fd[1]);
    execlp("ls", "ls", NULL);
}
if (fork() == 0) {    /* child 2 */
    dup2(fd[0], 0);
    close(fd[0]); close(fd[1]);
    execlp("wc", "wc", "-l", NULL);
}
/* shell: close both ends */
close(fd[0]); close(fd[1]);
wait(NULL); wait(NULL);
```

The first line of the code declares an array of two elements. `pipe(fd)` creates a buffer and two ends in the kernel. The kernel always hands out the smallest free descriptor number. Since 0, 1, and 2 are taken, the read end becomes 3 and the write end becomes 4. The first `fork` creates the child process. The child receives a copy of the parent's descriptor table; both processes point to the same pipe. With `dup2(fd[1], 1)` the child redirects descriptor 1 to the write end of the pipe. It closes 3 and 4, which are no longer needed. `execlp` replaces the process image with `ls`. The descriptor table is preserved across `exec`. `ls` writes to 1, unaware of anything, and the data flows into the pipe. The second child does the same job for the read end: `dup2(fd[0], 0)`, closes both ends, and becomes `wc -l`. Finally the shell closes both ends and waits for the two children.

Step	Shell 0 1 2 3 4	Child 1	Child 2
`pipe(fd)`	tty tty tty r w	—	—
1. `fork`	tty tty tty r w	tty tty tty r w	—
`dup2(4, 1)` + two `close` calls	tty tty tty r w	tty **w** tty – –	—
`execlp("ls")`	same	tty w tty – – (now `ls`)	—
2. `fork`, `dup2(3, 0)`, `exec("wc")`	tty tty tty r w	same	**r** tty tty – –

Step	Shell 0 1 2 3 4	Child 1	Child 2
The shell closes both ends	tty tty tty – –	same	same

Table 15.3: Descriptor tables while `ls | wc -l` is being set up (tty: terminal; w: write end of the pipe; r: read end).

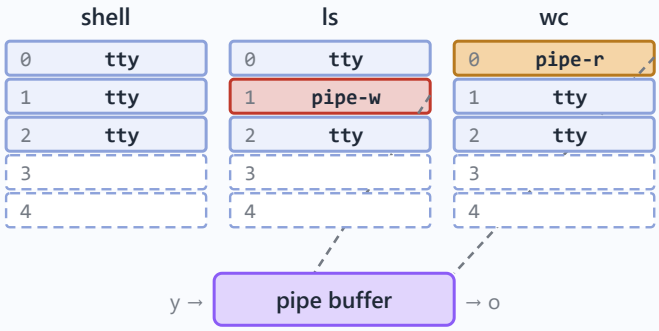


Figure 15.7: Final state: the write end of the pipe is open only in `ls`, and the read end only in `wc`.

The two `close` calls in the second-to-last line look insignificant, but the correct operation of the system depends on them. The reader of a pipe sees EOF only when *all* write ends are closed. Had the shell not closed 4, the write end would have stayed open. Even after `ls` finished and closed its own end, `wc` would assume more data might arrive and wait forever. Since the shell waits for `wc`, the two would hang together. In the opposite case, when all read ends are closed, a process writing to the pipe receives the `SIGPIPE` signal and, by default, terminates.

Common mistake

Forgetting to close the extra ends in the children. If the child that runs `wc` does not close 4, it never sees its own EOF because of the write end it holds itself. The rule is short: every process closes every pipe end it does not use.

15.3 The structure of the kernel

15.3.1 Components of the monolithic kernel

The Linux kernel is divided into three main components (Figure 15.8). The **I/O component** manages terminals, the network, and file systems. **Memory management** runs virtual memory, paging, and the page cache. **Process management** creates processes and threads, handles signals, and schedules the CPU. The system-call interface sits above all of them, and the interrupts and the dispatcher below. The dispatcher is the code that performs the context switch. The classic 4.4BSD kernel was divided into the same three components.

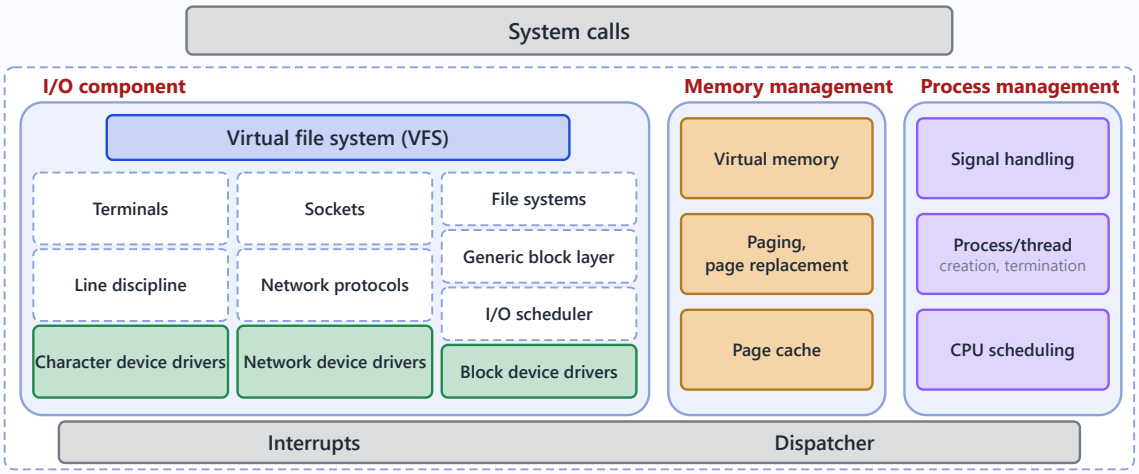


Figure 15.8: The structure of the Linux kernel: I/O, memory, and process management; system calls above, interrupts and the dispatcher below.

The I/O component is itself divided into three columns. Character devices pass through terminals and the line discipline. Network devices pass through the socket layer and the protocols. Block devices pass through the file systems, the generic block layer, and the I/O scheduler. The three columns meet at the top in the **virtual file system** (VFS). All the boxes in the figure run in a single address space, in kernel mode. One component can call another's functions directly; no message passing or context switch is needed. That is what makes Linux **monolithic**. The price is also clear: a bug in any component, for example an invalid pointer in a driver, can crash the entire kernel.

15.3.2 Loadable kernel modules

A monolithic kernel that included the driver for every conceivable device from the start would be very large. Linux solves this problem with modules.

> **Definition 15.1**
>
> A **loadable kernel module** (LKM) is an object file with the `.ko` extension that can be added to and removed from the kernel while the system is running. A loaded module runs in the kernel's address space with kernel privilege.

Most drivers, and some of the file systems and network protocols, are modules. The loading of a module can be followed in six steps (Figure 15.9). As an example, let the Intel network-card driver `e1000e` be loaded.

1. **Request.** The administrator types `modprobe e1000e`. The more common path is automatic loading: when a device is plugged into the system, the kernel generates an event, and `udev` in user space finds and loads the module that matches the device's ID.

2. **Dependencies.** `modprobe` reads from the file `/lib/modules/$(uname -r)/modules.dep` which modules the module needs and loads those first. The more primitive `insmod` does not do this; it tries to load the single file it is given.
3. **Copy and signature.** The `finit_module()` system call copies the module file into kernel memory. The caller must hold the `CAP_SYS_MODULE` capability. When Secure Boot is on, the module's signature is verified; an unsigned module is rejected.
4. **Linking.** The kernel performs the module's relocations. It looks up every external symbol the module uses in the kernel's symbol table. Only kernel functions exported with `EXPORT_SYMBOL` are visible to the module.
5. **Initialization.** The module's function marked with `module_init` runs. The driver registers itself with the relevant subsystem; for a PCI card this is the `pci_register_driver` call.
6. **Matching.** The PCI subsystem compares the IDs the driver supports with the devices in the system. For each matching device, the driver's `probe()` function is called and a network interface such as `eth0` comes into being.

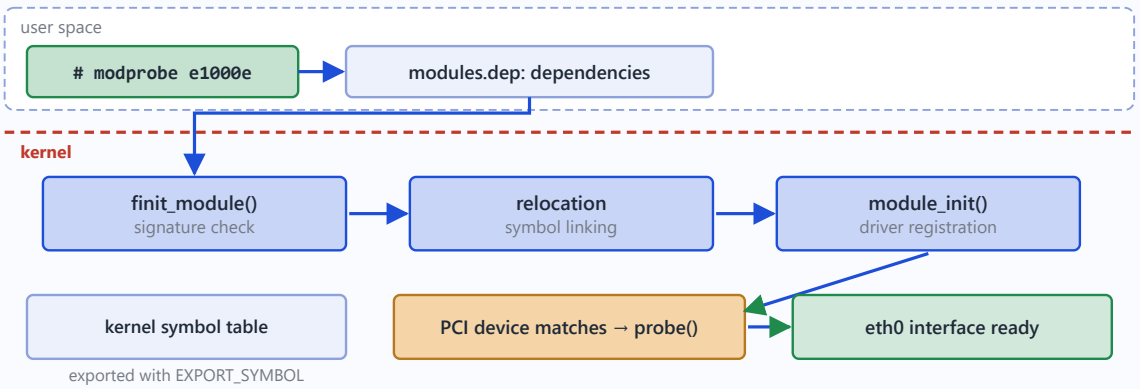


Figure 15.9: Loading a module: `modprobe`, `finit_module()`, signature check, relocation and symbol linking, `module_init()`, and `probe()`.

Loaded modules are listed with `lsmod` and removed with `rmmod`. The module system makes the kernel flexible but brings no protection. A module is as privileged as the kernel itself. For this reason the right to load modules is tightly controlled.

The kernel source tree passed 40 million lines in early 2025. Most of those lines are drivers. In recent years two innovations have aimed at improving kernel security. The first is Rust. Rust support entered the kernel in 6.1; the first Rust driver, a network PHY driver, was added in 6.8. Rust's type system catches some of the memory errors common in C drivers at compile time. The second is **eBPF**. eBPF lets small programs written by the user run safely inside the kernel. Before a program is loaded, it passes through a verifier. The verifier does not allow loading until it has proved that the program terminates and accesses only permitted memory. With eBPF, network packets are filtered and kernel events are traced; the kernel's behavior is extended without writing a module.

15.4 Processes and threads

15.4.1 `fork` and `execve` in the kernel

On Linux a new program starts in two steps (Figure 15.10). The shell (PID 501) first copies itself with `fork()`; the copy receives PID 748 and still runs the shell's code. The copy then replaces its own image with `ls` using `execve()`. The PID does not change; what changes is the program the process runs.

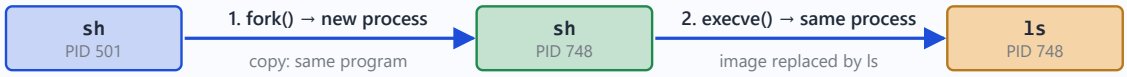


Figure 15.10: `fork()` produces a new process (same program); `execve()` produces a new program in the same process.

In the kernel, `fork()` does the following:

1. It allocates a new `task_struct`, fills it in from the parent's, and assigns a new PID.
2. It allocates a separate kernel stack for the child. It copies the parent's saved registers, setting only the return-value register to 0 in the child. This is where the “double return” of `fork` comes from.
3. It copies the page tables and marks writable pages read-only on both sides. This is **copy-on-write** (COW): a page is copied only when one of them tries to write to it.
4. It shares the open files. The child's descriptor table is a copy of the parent's; each entry points to the same `struct file` object, and the object's reference count is incremented.
5. It copies the signal handlers, the working directory, and the credentials.

`execve()`, on the other hand, discards the old image. First it reads the file's ELF header. It drops the `mm_struct` that describes the process's old address space and builds a new one. It maps the program's code and data segments from the file with `mmap`; it reads no page immediately. If the file's setuid or setgid bit is set, it prepares the process's new credentials (`cred`). Finally it sets the program counter. In a dynamically linked program, the first code to run is not the program itself but the dynamic linker `ld.so`, which maps the shared libraries and jumps to the program's entry point. A successful `execve` never returns, because the code that called it no longer exists. If it fails, it returns −1. In that case the child traditionally exits with `_exit(127)`; this is where the shell's “command not found” exit status 127 comes from.

Old UNIX systems really copied the process's data and stack segments on `fork`. Linux copies only the page tables. Since most `fork`s are immediately followed by an `exec`, very few pages end up being copied. Even if the shell's address space holds thousands of pages, `fork` takes microseconds.

15.4.2 Process-management calls

The life of a process is managed with a few system calls (Table 15.4). `clone` and `clone3` are the general creation calls; `fork` is a special case of them. The difference between `exit` and `_exit` is small but important: `exit` is a library function that flushes the stdio buffers and calls the `atexit` functions; `_exit` goes directly to the kernel.

Call	Function
`fork()`	Creates a new process by copying the process (COW)
`clone()`, `clone3()`	A new task whose shared parts are chosen with flags (thread, container)
`execve()`	Replaces the process image with a new program
`waitpid()`	Waits for a child to finish, obtains its exit status, and reaps the zombie
`exit()` / `_exit()`	Terminates the process (`exit` flushes the stdio buffers)
`kill()`	Sends a signal to a process; a signal need not terminate it
`sigaction()`, `sigprocmask()`	Installs a handler; reads and changes the set of blocked signals
`sigpending()`, `sigsuspend()`	Returns the pending signals; temporarily changes the mask and waits for a signal
`rt_sigreturn()`	Restores the saved context on return from a handler
`alarm()` / `pause()`	Generates `SIGALRM` after n seconds / sleeps until a signal arrives

Table 15.4: Process-management system calls.

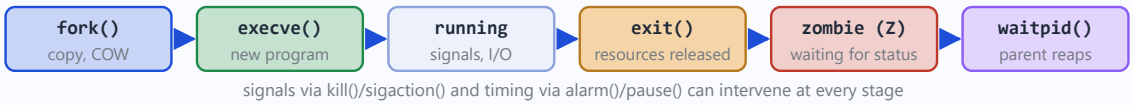


Figure 15.11: The life of a process: `fork`, `execve`, running, `exit`, zombie, and reaping with `waitpid`.

A process that terminates does not vanish at once (Figure 15.11). It releases its memory and files, but its `task_struct` remains, carrying the exit status. A process in this state is called a **zombie**; `ps` shows it with the letter Z. When the parent reads the status with `waitpid`, the zombie is removed completely. If the parent dies first, its children are re-parented to PID 1 (or to a process that has declared itself a "subreaper"), which reaps them. The process tree can be seen with `pstree -p` or `ps -ef --forest`.

15.4.3 The process descriptor: `task_struct`

The Linux counterpart of the textbook process control block (PCB) is `struct task_struct` (Figure 15.12). The structure is large; on x86-64 it takes a few kilobytes and varies from version to version. But most of what it contains is not data directly but pointers to other structures.

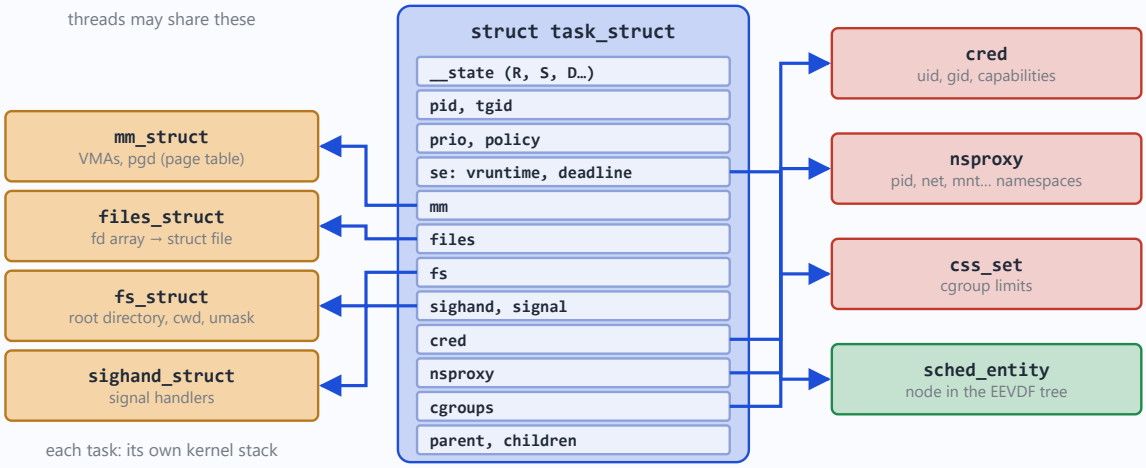


Figure 15.12: `task_struct` and the structures it points to. Those on the left can be shared among threads; each task has its own kernel stack.

The fields the structure holds itself describe the task's identity and scheduling state. `__state` is the task's state. `pid` and `tgid` are identifiers. `prio` and `policy` give the priority and the scheduling policy. The `se` field is an embedded `sched_entity` structure; the `vruntime` and `deadline` values used by the fair scheduler live here. `parent` and `children` build the process tree. The remaining fields are pointers:

- `mm` → `mm_struct`: the address space. The list of memory regions (VMAs) and the address of the top-level page table (`pgd`) are here. Kernel threads have a null `mm`.
- `files` → `files_struct`: the file-descriptor table. Each entry points to a `struct file`.
- `fs` → `fs_struct`: the root directory, the working directory, and the umask.
- `sighand` and `signal`: the signal handlers and the process-wide signal state.
- `cred`: uid, gid, and capabilities.
- `nsproxy`: the namespaces the task lives in (PID, network, mount, and so on).
- `cgroups` → `css_set`: the control groups the task belongs to and their limits.

The consequence of this layout is Linux's most surprising design decision: **there is no structural difference between a process and a thread**. Both are a task, that is, a `task_struct`. The difference is whether the substructures are shared. The `mm` pointers of two separate processes point to different `mm_struct`s. The `mm` pointers of two threads of the same process point to the same object.

Definition 15.2

On Linux, the unique in-kernel identifier of every task is `pid`. The identifier common to the threads of one process is the **thread group ID** (`tgid`). The value a user sees as the "PID" is the `tgid`: `getpid()` returns the `tgid`, and `gettid()` returns the task's own `pid`.

Example 15.2

A single-threaded process with PID 900 calls `pthread_create` twice. The kernel gives the new tasks the IDs 901 and 902 in turn. What do `getpid()` and `gettid()` return in each task?

1. Main thread: `pid` = 900, `tgid` = 900. `getpid()` = 900, `gettid()` = 900. The group leader is the task whose ID equals the group ID.
2. The new threads are born with `CLONE_THREAD`, so they inherit the group's `tgid`: `getpid()` is 900 in both. `gettid()` is 901 and 902.
3. `ps` shows a single process, 900. `ps -L` or the directory `/proc/900/task/` lists the three tasks separately.

The values of the `__state` field appear as letters in the STAT column of `ps`. R means running or ready to run; Linux does not distinguish these two states. S is interruptible sleep: the task waits for an event and can be woken by a signal. D is **uninterruptible sleep**: the task is usually waiting for a disk operation to finish, and signals do not wake it. T means stopped and Z means zombie. A task in state D does not die immediately even with `SIGKILL`. An unresponsive network file system could therefore produce processes that resist `kill -9`. To reduce this problem, the kernel now uses `TASK_KILLABLE` in most waits: the task can be woken only by a fatal signal.

15.4.4 The `clone()` flags and NPTL

Since a process and a thread are represented by the same structure, the same call can create both. That call is `clone()`.

Definition 15.3

`clone(flags, …)` is the Linux system call that creates a new task. Each flag chooses whether one substructure of the parent is *shared* with the child or the child gets a *new* copy of it. A structure for which no flag is given is copied.

The sharing flags map one to one onto the pointers of `task_struct` (Table 15.5). If `CLONE_VM` is given, the child's `mm` points to the same object as the parent's; if not, the address space is copied with COW. `CLONE_FILES` shares the descriptor table, `CLONE_FS` the root and working directories, and `CLONE_SIGHAND` the signal handlers. `CLONE_THREAD` puts the child into the parent's thread group: the child gets the same `tgid` and looks like the same process from the outside.

Flag	If given	If not given
`CLONE_VM`	the address space (`mm`) is shared	copied with COW

Flag	If given	If not given
`CLONE_FILES`	the descriptor table is shared	the table is copied (same `struct file` s)
`CLONE_FS`	root, working directory, and umask are shared	copied
`CLONE_SIGHAND`	signal handlers are shared (requires `CLONE_VM`)	copied
`CLONE_THREAD`	same thread group, same `tgid` (requires `CLONE_SIGHAND`)	new process, new `tgid`
`CLONE_NEWPID`, `CLONE_NEWNS`, `CLONE_NEWNET`, `CLONE_NEWUTS`, `CLONE_NEWIPC`, `CLONE_NEWUSER`, `CLONE_NEWCGROUP`	the child starts in a new namespace	stays in the parent's namespace

Table 15.5: The main `clone` flags and the structures they affect.

Three typical calls show the power of the flags (Figure 15.13).

1. `fork()`. In glibc, `fork` is a `clone` called with only the `SIGCHLD` flag. There are no sharing flags; the child gets a copy of every structure, and memory is shared with COW. Separate PID, separate TGID: a new process. Here `SIGCHLD` is not a sharing flag but the number of the signal to be sent to the parent when the child terminates.
2. `pthread_create()`. Linux's thread library NPTL (Native POSIX Thread Library) passes these flags: `CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID`. Everything is shared, and the new task joins the same TGID. The last three flags set up the thread's local storage and notify `pthread_join` of its termination.
3. **Container.** The namespace flags give the child a new view of the world. A child born with `CLONE_NEWPID` is the first process of its own PID namespace: inside, it believes it is PID 1; outside, it carries an ordinary PID. This is the foundation of containers.

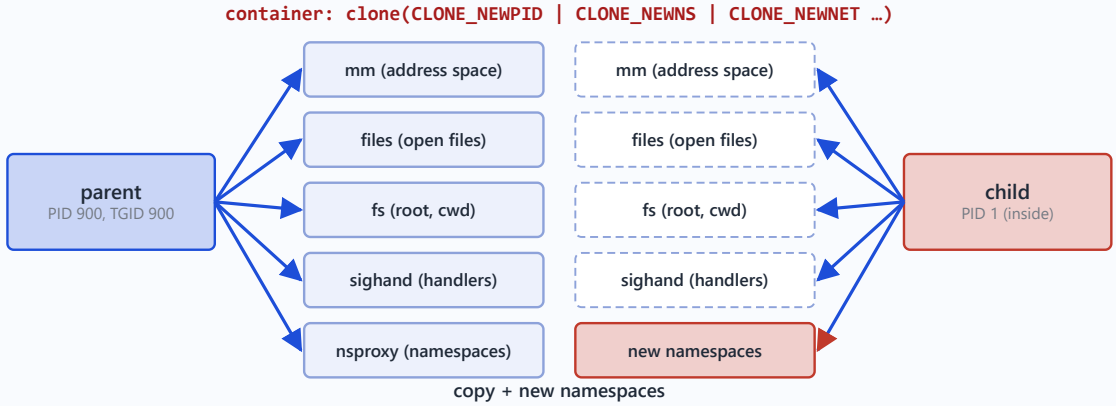


Figure 15.13: `clone` with namespace flags: the child gets copies of the structures and new namespaces; inside, it is PID 1.

The model NPTL builds is **1:1**: every user thread corresponds to one kernel task. The kernel schedules threads directly; when one blocks, the others keep running, and on a multicore machine they truly run in parallel. NPTL arrived in 2003 with the 2.6 series. LinuxThreads, which came before it, was also 1:1, but every thread carried a separate PID and signal handling did not conform to POSIX. `CLONE_THREAD` and the TGID concept were added to solve these problems.

Because the flags can be combined independently, "half-process, half-thread" tasks are also possible. A task born with `CLONE_VM | CLONE_FILES` but without `CLONE_THREAD` appears as a separate process; it has its own TGID. Yet it uses the same memory and the same descriptor table as its parent. A file opened by one is open in the other. Classic UNIX has no such intermediate form. `clone3()` (5.3) does the same job with a wider argument structure; new flags are now added to it.

Common mistake

Assuming the PID in `ps` refers to a thread. A multithreaded server is a single line in `ps`, because `ps` lists TGIDs. Sending a signal to one thread requires `tgkill` and the task ID; `kill` sends the signal to the whole group, and a suitable thread within the group receives it.

15.5 Signals

15.5.1 POSIX signals

Definition 15.4

A **signal** is a software interrupt that notifies a process of an event. Every signal has a number and a default action. For each signal, the process can choose the default action, ignoring it, or its own handler.

Signals arise in two ways. **Synchronous** signals are the result of the process's own instruction: an invalid memory access produces `SIGSEGV`, and integer division by zero produces `SIGFPE`. **Asynchronous** signals come from outside: Ctrl-C at the keyboard produces `SIGINT`, and the `kill` command produces `SIGTERM` by default. A handler's usual jobs are cleaning up resources, writing a log, or re-reading the configuration.

Signal	No.	When is it generated?	Default
`SIGHUP`	1	Terminal connection lost; in daemons, "re-read the configuration"	terminate
`SIGINT`	2	Ctrl-C (to the foreground process group)	terminate
`SIGQUIT`	3	Ctrl-\	core dump
`SIGABRT`	6	`abort()`	core dump

Signal	No.	When is it generated?	Default
`SIGFPE`	8	Arithmetic error	core dump
`SIGKILL`	9	`kill -9` ; cannot be caught or ignored	terminate
`SIGUSR1` / `SIGUSR2`	10 / 12	User-defined	terminate
`SIGSEGV`	11	Invalid memory access	core dump
`SIGPIPE`	13	Writing to a pipe with no reader	terminate
`SIGALRM`	14	`alarm()` timer expired	terminate
`SIGTERM`	15	Default of `kill` ; a polite request to terminate	terminate
`SIGCHLD`	17	Child terminated or stopped	ignore
`SIGCONT` / `SIGSTOP`	18 / 19	Continue / stop (cannot be caught); Ctrl-Z generates `SIGTSTP` (20)	continue / stop

Table 15.6: Commonly encountered signals (numbers are for x86 and ARM Linux).

`SIGKILL` and `SIGTERM` are both sent with the `kill` command. The difference is catchability. A process can catch `SIGTERM` ; it closes its open files, deletes its temporary files, and exits gracefully. `SIGKILL` and `SIGSTOP` , on the other hand, cannot be caught, blocked, or ignored. They are designed this way so that the administrator can always stop a runaway process. The right order is `SIGTERM` first and, if necessary, `SIGKILL` . The numbers between `SIGRTMIN` and `SIGRTMAX` (34–64 in glibc) are reserved for real-time signals. Unlike classic signals, these are queued: if the same signal arrives three times, it is delivered three times.

15.5.2 The delivery path of a signal

Generating a signal and handling it are separate moments (Figure 15.14). In between, the signal is **pending**. The Ctrl-C example shows the path from start to finish.

1. **Generation.** The user presses Ctrl-C. The line discipline of the tty driver recognizes the character and generates `SIGINT` for every process in the foreground process group. The only thing the kernel does is set the bit of signal 2 in the target task's pending-signal set.
2. **Check.** The signal is not handled immediately. When a process returns to user mode from a system call or an interrupt, the kernel asks, “is there a pending, unblocked signal?” A sleeping process is woken by the signal and goes through this check on its return path.
3. **Frame.** If the process has installed a handler with `sigaction` , the kernel writes a **signal frame** onto the process's user stack. The frame holds the registers of the interrupted moment and a return trampoline. The kernel sets the program counter to the address of the handler.
4. **Handler.** The handler runs in user mode, as part of the process's own code.
5. **Return.** When the handler returns, the trampoline makes the `rt_sigreturn` system call. The kernel restores the registers from the frame.
6. **Resume.** The process continues from the instruction at which it was interrupted (PC = 0x401a2c in the figure). Had no handler been installed, the default action would have been applied; for `SIGINT` , that means the process terminates.

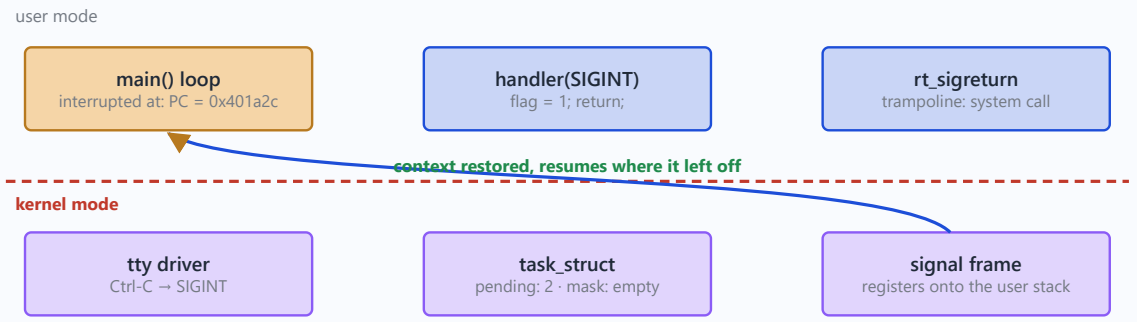


Figure 15.14: A signal is generated, waits, and is delivered to the handler. The return is made with the `rt_sigreturn` call.

The handler runs in the middle of the interrupted code, in the same thread. The process may have been interrupted right inside `malloc`, while holding the heap's lock. If the handler also calls `malloc` or `printf`, it tries to take the same lock a second time and the process deadlocks. For this reason only **async-signal-safe** functions may be used in a handler, such as `write` and `_exit`. The most robust pattern is for the handler to set just a flag of type `volatile sig_atomic_t` to 1 and for the main loop to do the real work by checking the flag.

> **Common mistake**
>
> Assuming `sigreturn` is the call that enters the handler. Entering the handler is not a system call; the kernel "sends" the process to the handler by changing the program counter. `rt_sigreturn` performs the *return* from the handler, and it is the trampoline, not the programmer, that calls it.

15.5.3 Code trace: selecting pending signals

The kernel keeps two bit sets for every task: the **pending** signals and the **blocked** signals. Signal number n sits at bit (n − 1). A simplified version of the check on return to user mode looks like this:

```
/* on return to user mode (kernel) */
void check_signals(task *t) {
    unsigned long h = t->pend & ~t->blk;
    while (h) {
        int sig = __builtin_ctzl(h) + 1;
        t->pend &= ~(1UL << (sig - 1));
        deliver(t, sig); /* handler/default */
        h = t->pend & ~t->blk;
    }
}
/* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
/* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```

The third line computes the set that is ready for delivery: ready = pending & ~blocked. The loop runs until this set is empty. `__builtin_ctzl(h)` counts the trailing zero bits of the number, that is, it gives the position of the lowest set bit; +1 converts that to a signal number. The lowest-numbered ready signal is therefore chosen first. The chosen signal's bit is cleared from the pending set and the signal is delivered. After delivery the set is recomputed, because running the handler may change the sets.

The scenario is as follows. Before entering a critical section, the process blocks `SIGINT` with `sigprocmask(SIG_BLOCK, {SIGINT})`. While it sleeps, three signals arrive: Ctrl-C (2), `kill -USR1` (10), and a child's termination (17). A handler is installed for `SIGUSR1`.

Event	pending	blocked	ready	Result
`SIG_BLOCK {SIGINT}`	{}	{2}	{}	—
2, 10, 17 arrive	{2, 10, 17}	{2}	—	only bits are set
Return to user mode	{2, 10, 17}	{2}	{10, 17}	lowest: 10
10 delivered	{2, 17}	{2}	{17}	`SIGUSR1` handler runs
17 delivered	{2}	{2}	{}	`SIGCHLD`: default, ignore
`SIG_UNBLOCK {SIGINT}`	{2}	{}	{2}	same check on return
2 delivered	{}	{}	{}	no handler: the process terminates

Table 15.7: Trace of `check_signals` (the bit operations were re-run in Python).

In bit notation, after the three signals arrive, the pending set is 0x10202: signal 2 is bit 1 (0x2), signal 10 is bit 9 (0x200), and signal 17 is bit 16 (0x10000). The blocked set is 0x2. The ready set is 0x10202 & ~0x2 = 0x10200. The lowest set bit is 9, that is, signal 10.

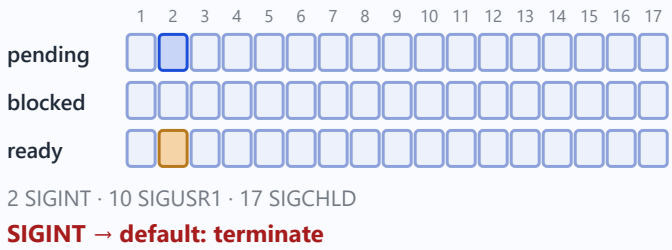


Figure 15.15: End of the trace: once the block is lifted, `SIGINT` enters the ready set and the default action terminates the process.

Two lessons come out of the trace. First, **blocking is deferring, not ignoring**. `SIGINT` was not lost during the critical section; it stayed in the pending set and was delivered as soon as the block was lifted. Ignoring (`SIG_IGN`), on the other hand, discards the signal at the moment it arrives. Second, classic signals are not counted. The pending set is a bit set. Had Ctrl-C been pressed three times during the critical section, the bit would still be set only once and the signal would be delivered only once.

15.6 Scheduling

15.6.1 Scheduling classes

The Linux scheduler is not a single algorithm but a set of **scheduling classes** arranged in priority order. Every task belongs to a class. When a CPU becomes free, the kernel asks the classes in order, and the first class that yields a task to run wins. The order is as follows:

1. **stop**: an internal class the kernel uses to stop a CPU or to migrate tasks.
2. **deadline**: the `SCHED_DEADLINE` policy. A task declares a runtime, a deadline, and a period; the scheduler runs the earliest deadline first.
3. **rt**: `SCHED_FIFO` and `SCHED_RR`, fixed priorities from 1 to 99. A FIFO task runs until it yields voluntarily; RR rotates among tasks of the same priority with a time slice.
4. **fair**: `SCHED_NORMAL` (formerly `SCHED_OTHER`), `SCHED_BATCH`, and `SCHED_IDLE`. All ordinary processes are here. Since 6.6 this class is governed by EEVDF.
5. **idle**: the idle task that runs when the CPU has no other work.

`sched_ext`, which arrived with 6.12, allows the scheduling policy to be loaded as an eBPF program. A company can try a scheduler suited to its own workload without recompiling the kernel. If the program misbehaves, the kernel removes it and gives the tasks back to the fair class. The mechanism is in the kernel, the policy in user space.

Every CPU has its own **run queue** (`struct rq`). Each class keeps its own data structure in this queue. Tasks are balanced between CPUs from time to time. The load balancer sees the CPUs as a hierarchy of scheduling domains: the hardware threads of the same core, the cores on the same chip, the chips in the same NUMA node. A task is migrated at the nearest level first, because its cache and memory are warmer there.

15.6.2 From O(n) to CFS

The selection rule of the fair class changed three times in twenty years (Table 15.8). The scheduler in the 2.4 kernel scanned all ready tasks at every selection. This cost, which grows linearly with the number of tasks, became a problem on servers with hundreds of tasks. The **O(1) scheduler**, which arrived with 2.6.0, kept a separate list for each priority level: an array of 140 lists and a bitmap showing which lists were non-empty. The highest-priority task was selected by a single instruction that finds the first set bit in the bitmap. A task whose time slice expired moved to a second, “expired” array, and when the active array emptied, the roles of the two arrays were swapped. The speed problem was solved, but the heuristic rules used to recognize interactive tasks became complicated and produced unpredictable behavior.

Version	Selection rule	Data structure
2.4 (2001)	O(n): all ready tasks are scanned at every selection	single list

Version	Selection rule	Data structure
2.6.0–2.6.22 (2003)	O(1): a list per priority, active and expired arrays	array of 140 lists + bitmap
2.6.23–6.5 (2007)	CFS: smallest `vruntime`	red-black tree
6.6+ (2023)	EEVDF: earliest virtual deadline among eligible tasks ($v \le V$)	the same tree, augmented with subtree information

Table 15.8: Evolution of the fair class's selection rule.

In 2.6.23 came Ingo Molnár's **Completely Fair Scheduler** (CFS). The idea of CFS is to emulate an ideal multitasking processor. On the ideal processor, n tasks would run at the same time, each at 1/n speed. A real processor cannot do this, but it can track which task has fallen furthest behind its share. This tracking is done with each task's **virtual runtime**.

15.6.3 Weights and virtual runtime

Definition 15.5

Every task has a **weight** w derived from its nice value; the weight of nice 0 is 1024. When a task runs for a real time Δt, its **virtual runtime** (`vruntime`) grows by:

$$\Delta v = \Delta t \times 1024 / w$$

The nice value lies between −20 and +19; a smaller value means a higher priority. The weights are read from the kernel's `sched_prio_to_weight` table. At each nice step the table changes the weight by a factor of about 1.25: nice 1 → 820, nice 0 → 1024, nice −1 → 1277. The result is that a task whose nice value increases by one loses roughly 10% of its CPU share. The extremes are 88,761 for nice −20 and 15 for nice 19.

The formula agrees with intuition at both ends. The virtual clock of a task with a large weight runs slowly; even if it runs a lot in real time, its virtual time grows little. The virtual clock of a task with a small weight runs fast. The rule of CFS is to run the task with the smallest `vruntime`. Tasks are ordered by `vruntime` in a red-black tree, and the leftmost node is chosen. Insertion and deletion are O(log n). Because the leftmost node is cached, selection is O(1).

Example 15.3

Three tasks with nice 0 (w = 1024), nice 5 (w = 335), and nice −5 (w = 3121) run for 6 ms. How far does each one's `vruntime` advance? In what ratio do a continuously ready nice 0 task and a nice 5 task share a single CPU?

1. nice 0: 6 × 1024 / 1024 = 6 ms.

2. nice 5: 6 × 1024 / 335 ≈ 18.34 ms. Its virtual clock runs three times as fast.
3. nice −5: 6 × 1024 / 3121 ≈ 1.97 ms.
4. In the long run the `vruntime` s of the two tasks stay close to each other. For that, the real run times must be proportional to the weights: 1024 / (1024 + 335) ≈ 75.3% for nice 0, and 335 / 1359 ≈ 24.7% for nice 5.

What happens to a task that has just woken up? A task that has slept for hours has a `vruntime` far behind. If it were put on the queue as it is, it would monopolize the CPU for a long time. CFS therefore pulled the woken task's `vruntime` to slightly behind the queue's minimum value. Adjustments of this kind multiplied over time: the latency target (`sched_latency`), the minimum slice, the wakeup granularity. Each solved one problem while creating a new one in another workload. The real shortcoming of CFS was theoretical. CFS determined well *how much* CPU a task would get, but gave no guarantee about *when* it would get it. A latency-sensitive task, such as audio or a game, needs the CPU at short intervals even when its share is small.

15.6.4 EEVDF: eligibility and virtual deadline

Linux 6.6 (2023) replaced the selection rule of the fair class with **EEVDF** (Earliest Eligible Virtual Deadline First). The algorithm is based on the 1995 work of Ion Stoica and Hussein Abdel-Wahab; Peter Zijlstra adapted it to the kernel. EEVDF adds two concepts: eligibility and virtual deadline.

Definition 15.6

The **average virtual time** of a queue is the weighted average of the tasks' `vruntime` s: $V = \Sigma\, w_i v_i \,/\, \Sigma\, w_i$. A task's **lag** is $\text{lag}_i = w_i\,(V - v_i)$. A task whose lag is zero or positive, that is, which satisfies $v_i \le V$, is **eligible**. Every task has a **virtual deadline**: $d_i = v_i + \text{slice} \times 1024 / w_i$. EEVDF chooses, among the eligible tasks, the one with the earliest virtual deadline.

Lag is the difference between the service a task would have received on the ideal processor and the service it actually received. A positive lag shows that the task has fallen behind its share; the scheduler owes it. A negative lag shows that the task has received more than its share; the task is not eligible and waits until its turn comes. The sum of the lags of all tasks is zero at every moment. The eligibility condition prevents a task from monopolizing the CPU by exceeding its share. The virtual deadline, in turn, answers the question of *when*. A task with a short slice has a near deadline and moves ahead as soon as it is eligible.

Not a few things stay unchanged in the move from CFS to EEVDF. The `vruntime` in `sched_entity` , the nice-to-weight table, and the red-black tree in the run queue remained as they were. The tree is still ordered by `vruntime` . Each node additionally stores the earliest deadline in its own subtree. With this extra information the scheduler finds the “eligible task with the earliest deadline” in a single descent of the tree, in O(log n) steps. What changed are CFS's tuning knobs. The latency target

and the wakeup granularity were removed; a single **base slice** took their place. Its value can be seen in the file `/sys/kernel/debug/sched/base_slice_ns`. The default is a base value of a fraction of a millisecond scaled by the number of CPUs; on machines with eight or more CPUs it is a few milliseconds. The examples in this chapter take the slice as 3 ms.

Example 15.4

On a single CPU there are two continuously ready tasks: A with nice 0 (w = 1024) and B with nice 5 (w = 335). Both start from v = 0, and the slice is 3 ms. Each task, once selected, runs until it finishes its slice. What do the selections look like over the first 24 ms?

1. t = 0. V = 0; both are eligible. Deadlines: $d_A = 0 + 3 = 3$, $d_B = 0 + 3 \times 1024 / 335 \approx 9.17$. A is chosen.
2. t = 3. $v_A = 3$, $d_A = 6$. $V = (3 \times 1024 + 0 \times 335) / 1359 \approx 2.26$. A is not eligible ($3 > 2.26$). The only eligible task is B; it is chosen.
3. t = 6. $v_B \approx 9.17$, $d_B \approx 18.34$. $V \approx 4.52$. A is eligible ($3 \le 4.52$), B is not. A is chosen.
4. t = 9 and t = 12. V is 6.78 and 9.04, respectively; v_A is 6 and 9. A is eligible two more times and is chosen. Since B's v is 9.17, V has not yet passed it.
5. t = 15. $v_A = 12$, $V \approx 11.30$. A is not eligible, B is eligible ($9.17 \le 11.30$). B is chosen. After that the pattern repeats.

In 24 ms, A runs six slices (18 ms) and B runs two slices (6 ms): 75% and 25%. This is the ratio of the weights ($1024 : 335 \approx 3 : 1$).

t (ms)	v_A / d_A	v_B / d_B	V	Eligible	Chosen
0	0 / 3	0 / 9,17	0	A, B	A
3	3 / 6	0 / 9,17	2,26	B	B
6	3 / 6	9,17 / 18,34	4,52	A	A
9	6 / 9	9,17 / 18,34	6,78	A	A
12	9 / 12	9,17 / 18,34	9,04	A	A
15	12 / 15	9,17 / 18,34	11,30	B	B
18	12 / 15	18,34 / 27,51	13,56	A	A
21	15 / 18	18,34 / 27,51	15,82	A	A

Table 15.9: v, d, and V at the decision points (A nice 0, B nice 5, slice 3 ms; produced with a Python simulation).

The share is determined by the weights, as in CFS. The difference of EEVDF appears when the two tasks have different slices. A latency-sensitive task can request a shorter slice with the `sched_setattr()` call; the call's `sched_runtime` field is interpreted as a slice request in the fair class. This facility was completed in 6.12. A short slice means an earlier deadline, and therefore being chosen more often. The share, however, does not change.

Example 15.5

A and B are both nice 0. A's slice is 3 ms; B has asked for 1 ms. What is the selection order over the first 12 ms?

1. t = 0: both are eligible; $d_A = 3$, $d_B = 1$. B is chosen and runs for 1 ms.
2. t = 1: $v_B = 1$, V = 0.5. Only A is eligible; A runs for 3 ms.
3. t = 4: $v_A = 3$, V = 2. B is eligible, $d_B = 2$. B is chosen. At t = 5 (V = 2.5, $d_B = 3$) and at t = 6 (V = 3, both eligible; $d_B = 4 < d_A = 6$), B is chosen again.
4. t = 7: $v_B = 4 > V = 3.5$. A runs (7–10). Then B gets two more slices.

The order is B AAA BBB AAA BB. In 12 ms both tasks ran for 6 ms; the shares are equal. But B got its first slice immediately and never waited longer than 3 ms.

Common mistake

Confusing EEVDF with "CFS, which picks the smallest `vruntime`." The task with the smallest `vruntime` is always eligible, but it may not be the one chosen. For example, with V = 12, A (v = 10, d = 14) and B (v = 12, d = 13) are eligible. EEVDF chooses B, which has the earlier deadline; CFS would have chosen A.

15.7 Booting

From the moment the power button is pressed to the login screen, a series of programs hand control to one another (Figure 15.16).

1. **UEFI.** The firmware on the motherboard tests the hardware. It finds the EFI system partition (ESP) on the disk and loads the EFI application there. If Secure Boot is on, each stage verifies the signature of the next.
2. **Bootloader.** GRUB2 or systemd-boot loads the kernel image (`vmlinuz`) and the `initramfs` into memory and passes command-line parameters to the kernel. The kernel can also be started directly as an EFI application. When the kernel, the initramfs, and the command line are combined in a single signed file, it is called a unified kernel image (UKI).
3. **Kernel.** The compressed image unpacks itself. `start_kernel()` sets up memory management, interrupts, and the scheduler. The first task to run this code becomes PID 0 and eventually turns into the idle task. The kernel creates `kernel_init` (PID 1) and `kthreadd` (PID 2), the ancestor of kernel threads.
4. **initramfs.** The drivers needed to mount the root file system may not be in the kernel: NVMe, RAID, an encrypted disk. The initramfs provides a temporary root in RAM. The scripts in it load the necessary modules, mount the real root, and switch to it with `switch_root`.

5. **systemd.** PID 1 is now systemd. It starts services in parallel according to the dependencies in unit files and reaches a target: `multi-user.target` or `graphical.target`.
6. **Session.** On the console `agetty` → `login` → shell, on the desktop gdm, and remotely sshd greets the user. A `systemd --user` instance is also started for the user's session.

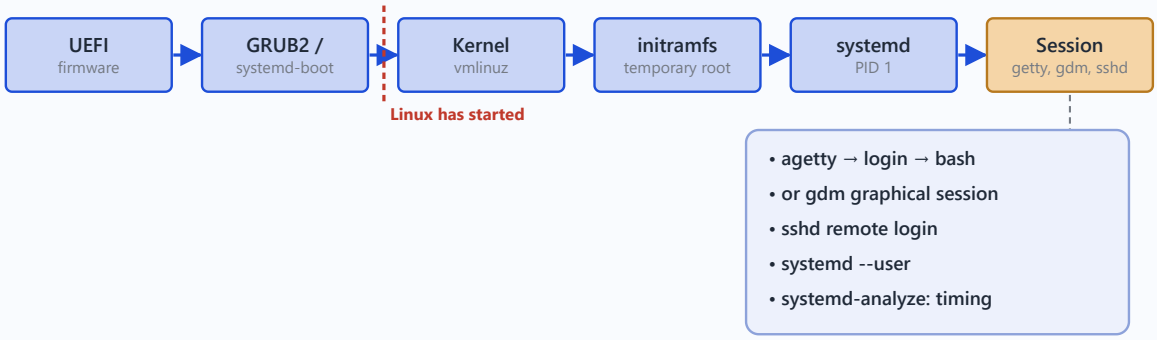


Figure 15.16: The Linux boot stages: UEFI, bootloader, kernel, initramfs, systemd, and session.

The `init → getty → login → sh` chain of classic UNIX is still recognizable today. The difference is that PID 1 is now a large program that starts services in parallel according to a dependency graph, tracks them in cgroups, and restarts a service that crashes. How long each stage of boot takes is measured with `systemd-analyze`.

Common mistake

Assuming PID 1 is an ordinary process. If PID 1 terminates, the kernel panics. Orphaned processes are re-parented to it, and it reaps their zombies. In a container, the PID 1 inside takes on these duties; if an application that does not reap zombies is run as PID 1, zombies accumulate in the container.

15.8 Memory management

15.8.1 The process address space and VMAs

With four-level paging on x86-64, virtual addresses are 48 bits. Linux divides this space into two halves (Figure 15.17). The lower half, 2^{47} B = 128 TiB, belongs to the user. The upper half, again 128 TiB, belongs to the kernel. The kernel half is the same in all processes and cannot be accessed in user mode. Between the two halves lies a huge unusable hole. With five-level paging (LA57), user space grows to 2^{56} B = 64 PiB.

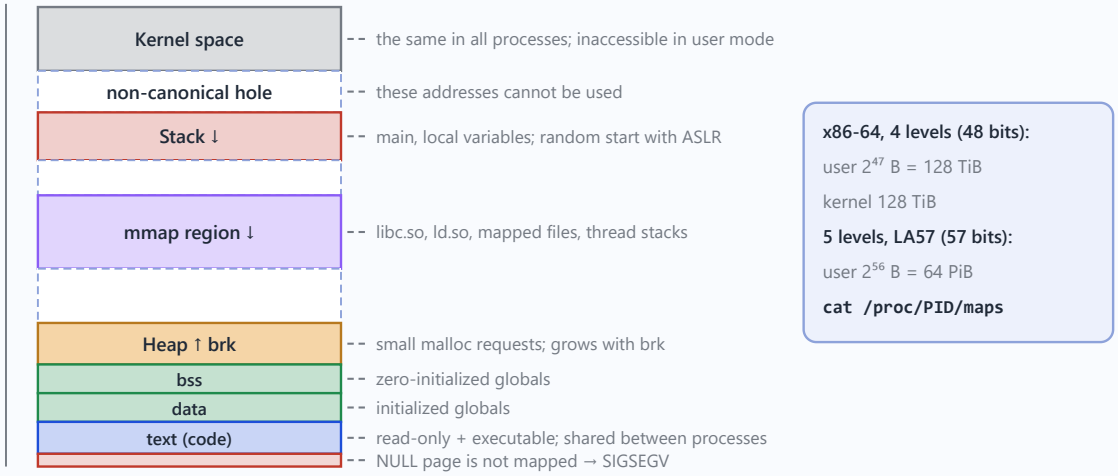


Figure 15.17: The virtual address space of a process on x86-64 Linux: code, data, bss, heap, mmap region, stack, and kernel space.

The layout of the user half, from bottom to top, is as follows. The lowest page is never mapped; a program that accesses a NULL pointer receives `SIGSEGV`. Next comes the program code (text): read-only and executable, and shared among all processes running the same program. Then come the initialized global variables (data) and the zero-initialized globals (bss). The heap starts right above bss and grows upward with `brk`. At the top end of the address space is the stack, which grows downward. Below the stack is the mmap region, which also grows downward: shared libraries (`libc.so`, `ld.so`), mapped files, and thread stacks are placed here. ASLR (address space layout randomization) changes the starting points of the stack, the mmap region, and the heap on every run, making it harder for an attacker to guess addresses. The gap between the heap and the mmap region is only virtual and consumes no physical memory.

Definition 15.7

A **virtual memory area** (VMA, `struct vm_area_struct`) is a contiguous region of the address space that carries the same permissions and the same backing. Each VMA holds its start and end addresses, its permissions (r, w, x, shared or private), and, if there is one, the file it maps.

A process's VMAs are kept in its `mm_struct`. Since 6.1 they are stored in a **maple tree**. This tree is a cache-friendly kind of B-tree suited to range lookups; it replaced the earlier red-black tree and linked list. On a page fault, the kernel's first question is “which VMA is this address in?” If the address is in no VMA, or the VMA does not permit the access, the process is sent `SIGSEGV`. If the address is in a valid VMA, the page is either read from the file or given as a fresh zeroed frame. A process's VMAs can be seen line by line with `cat /proc/PID/maps`.

15.8.2 Sharing, mapped files, and memory calls

When two processes run the same program, they keep the text pages in physical memory only once (Figure 15.18). The page tables of the two processes point to the same frames. Two processes that map the same file as shared also see the same page-cache frames. The data and stack pages of each process, however, belong to it alone.

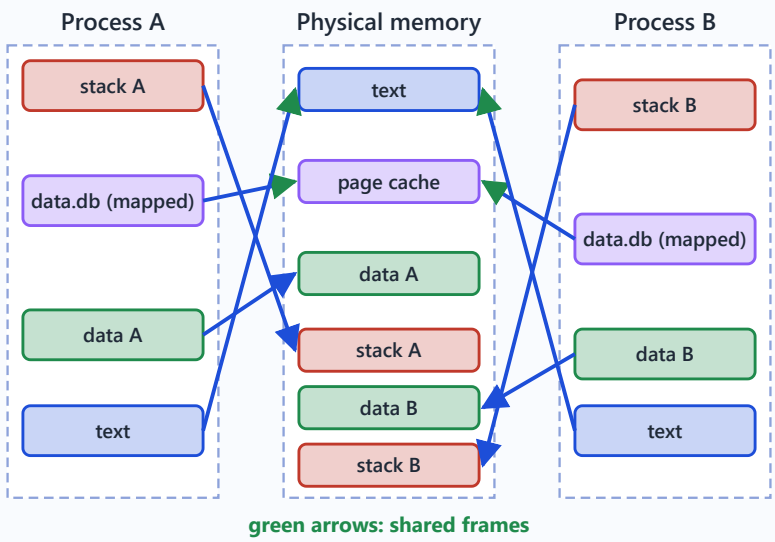


Figure 15.18: Processes A and B share the program text and the mapped file `data.db` in the same physical frames.

Call	Function
`brk` / `sbrk`	Grows or shrinks the heap by moving its end
`mmap`	Maps a file, a device, or anonymous memory into the address space
`munmap`	Removes the mapping and returns the memory to the system
`mprotect`	Changes the permissions (r/w/x) of a region
`madvise`	Gives the kernel a usage hint (e.g. `MADV_DONTNEED`)
`msync`	Writes the dirty pages of a shared mapping to the file

Table 15.10: Memory-management system calls.

`mmap` has two kinds of mapping. With `MAP_SHARED`, what is written is visible in the file and to other processes that map the same file. With `MAP_PRIVATE`, the page is copied on the first write and the file does not change; a program's data segment is mapped this way. A mapping that has no file is called an anonymous mapping; its pages come from zeroed frames on first touch. glibc's `malloc` serves small requests from the heap and grows the heap with `brk` as needed. Requests of 128 KiB and above it serves directly with an anonymous `mmap`. This threshold is a default, and glibc can adapt it at run time. Large blocks thus go straight back to the system with `munmap` at `free` time.

15.8.3 The page-table walk: four and five levels

On x86-64, a 4 KiB page needs a 12-bit offset. The remaining 36 bits of the 48-bit address are split among four tables. Each table is a 4 KiB page and holds 512 entries of 8 bytes; a table index is 9 bits. Linux's names for the tables are: PGD (bits 47–39), PUD (38–30), PMD (29–21), and PTE (20–12). Bits 63–48 must be a copy of bit 47. An address that satisfies this condition is called a **canonical** address. In user addresses these bits are 0; in kernel addresses they are 1.

Example 15.6

The virtual address is 0x00007f3a1c2b5e88. What are the four indices and the offset? If the frame number in the PTE is 0x3c5a1, what is the physical address?

1. The offset is the lowest 12 bits, that is, the last three hexadecimal digits: 0xe88.
2. PGD = address >> 39 = 0xfe = 254. Bits 63–48 are 0 and bit 47 is also 0; the address is canonical.
3. PUD = (address >> 30) & 0x1ff = 232. PMD = (address >> 21) & 0x1ff = 225. PTE = (address >> 12) & 0x1ff = 181.
4. Physical address = frame × 4096 + offset = 0x3c5a1 × 0x1000 + 0xe88 = 0x3c5a1e88. In hexadecimal, the multiplication is just appending three zeros to the frame number.

The walk itself goes as follows (Figure 15.19). The `CR3` register holds the physical address of the process's PGD table; on a context switch the kernel loads `CR3` with the new process's `mm->pgd`. PGD[254] points to a PUD table. PUD[232] points to a PMD table; a 1 GiB huge page can also end at this level. PMD[225] points to a PTE table; 2 MiB huge pages and transparent huge pages (THP) end here. PTE[181] is the actual entry: the present bit, the frame number, read/write (R/W), user/supervisor (U/S), no-execute (NX), dirty (D), and accessed (A) bits. If the present bit is 0, the hardware generates a page fault.

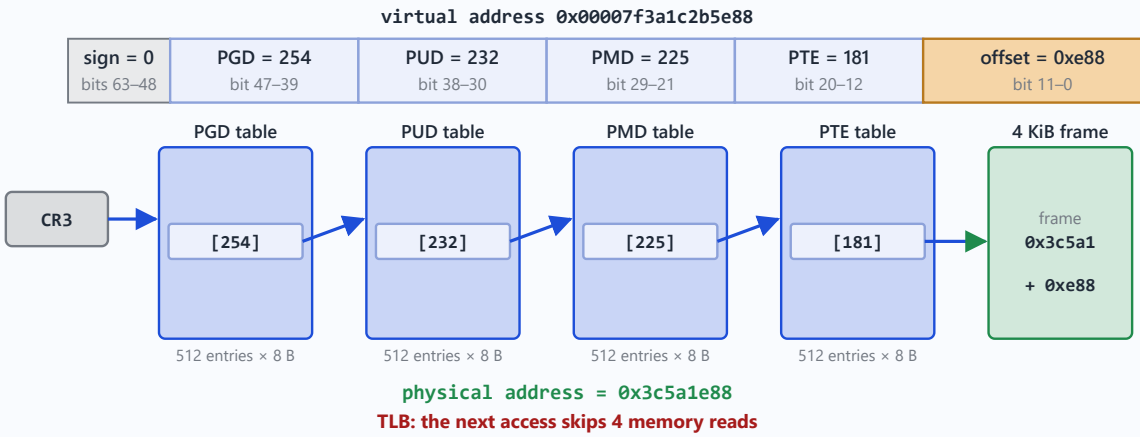


Figure 15.19: The four-level walk for 0x00007f3a1c2b5e88: PGD 254, PUD 232, PMD 225, PTE 181, and offset 0xe88.

Making four extra reads on every memory access is expensive. The TLB caches the translation; on the next access the four reads are skipped. On a TLB miss, the hardware does the walk; the kernel only builds the tables. Huge pages shorten the walk by one or two levels and cover a much larger area with a single TLB entry.

Five-level paging (LA57) has been supported since Linux 4.14. It adds a level called P4D between the PGD and the PUD and extends the virtual address to 57 bits. Linux's page-table code is generically five-level. If the hardware uses fewer, the intermediate levels are “folded”: on a four-level machine the P4D is treated as having a single entry and embedded in the PGD. The same code thus works from two-level 32-bit ARM to five-level x86-64.

> **Common mistake**
>
> Trying to read the indices directly from the hexadecimal digits of the address. The 9-bit fields are not aligned with the 4-bit digits. The right way is to shift and mask: (address >> k) & 0x1ff, with k = 39, 30, 21, 12. Only the 12-bit offset corresponds to the last three digits.

15.8.4 Representing physical memory

Linux describes physical memory with a three-level hierarchy: node → zone → page (Figure 15.20). On a multi-socket server, each processor socket has memory close to itself. This arrangement is called NUMA (non-uniform memory access). Each NUMA node is described by a `pglist_data` structure. A CPU allocates memory from its own node first, because accessing a remote node is slower.

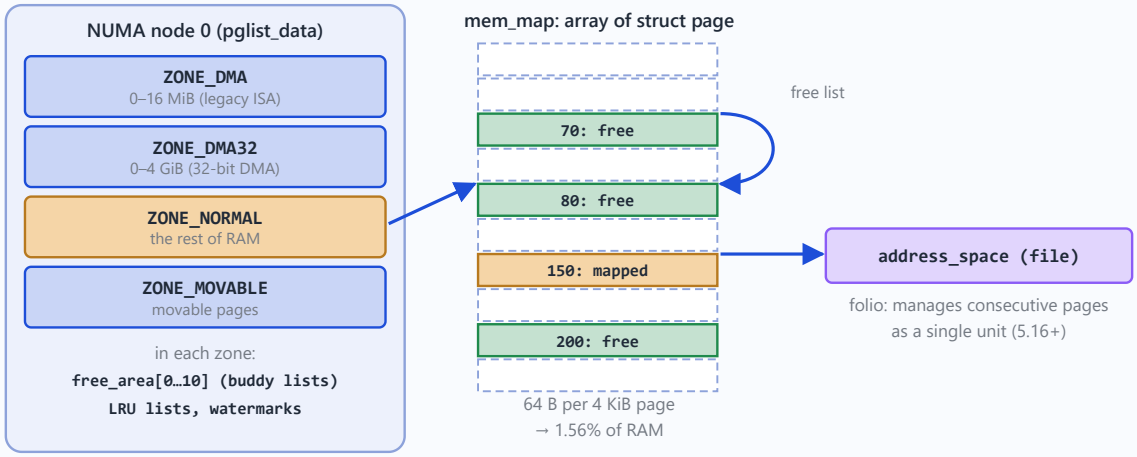


Figure 15.20: Physical memory: the NUMA node, the zones, the `struct page` array, and the buddy free lists.

Each node is divided into zones. `ZONE_DMA` is the first 16 MiB; it is for old ISA devices that can address only 24 bits. `ZONE_DMA32` is the first 4 GiB; it is for devices that can do 32-bit DMA. `ZONE_NORMAL` is the rest of RAM. `ZONE_MOVABLE` holds only movable pages; it can be emptied before a

memory module is removed and makes it easier to create huge pages. On 32-bit kernels there was also `ZONE_HIGHMEM` for high memory that the kernel could not map directly. On 64-bit systems this zone does not exist, because the kernel address space is large enough to map all of RAM directly.

For every physical page frame there is a `struct page` in the kernel. The structure is 64 bytes. These live in a large array called `mem_map`. 64 B for every 4 KiB page means 64 / 4096 ≈ 1.56% of RAM. On a 16 GiB machine this array takes 256 MiB. To manage consecutive pages as a single unit, the **folio** concept arrived with 5.16. A folio can cover multiple pages; the page-cache and reclaim code work folio by folio instead of page by page.

Each zone's `free_area` array holds the buddy allocator's free lists. List k contains free blocks of 2^k pages. k runs from 0 to 10: from 4 KiB to 4 MiB. A request is served from the smallest suitable block; a larger block is split in two, and when a freed block's "buddy" is also free, the two merge. The numbers of free blocks in the zones can be read from `/proc/buddyinfo`. Each zone also has three watermarks (min, low, high); reclaim works according to them.

15.8.5 The SLUB object allocator

The kernel allocates large numbers of small objects of the same size: a `task_struct` for every process, a `struct file` for every open file, a dentry (192 B) for every name, an inode for every file. Allocating a page for each of them would be both slow and wasteful.

> **Definition 15.8**
>
> **SLUB** is Linux's object allocator. For each object type it keeps an **object cache** (`kmem_cache`). The cache divides one or several consecutive pages it takes from the buddy allocator (a **slab**) into objects of the same size. Free objects are linked into a free list.

Some of the caches belong to specific types: `task_struct`, `dentry`, `inode_cache` (Figure 15.21). `kmalloc`, on the other hand, is general-purpose and uses caches divided into size classes: `kmalloc-8`, `kmalloc-16` … `kmalloc-8k`. A 100-byte request is served from `kmalloc-128`. Larger requests go directly to the buddy allocator. For large areas that need not be physically contiguous, `vmalloc` is used; it maps scattered frames into a virtually contiguous region.

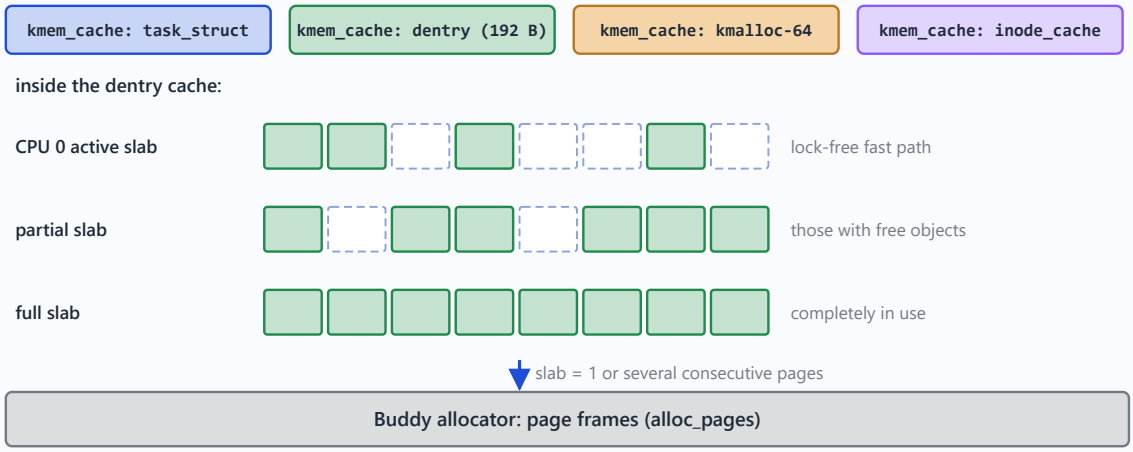


Figure 15.21: SLUB: object caches divide the pages they take from the buddy allocator into fixed-size objects. Each CPU has an active slab, and the cache has partial and full slabs.

A cache has three kinds of slabs. Each CPU has an **active slab**; most allocation and freeing happens here without taking a lock. **Partial** slabs contain both full and free objects. In **full** slabs, all objects are in use. When the active slab has no free objects left, the CPU takes a new slab from the partial list; if there is none, it asks the buddy allocator for pages. The state of the caches can be watched in `/proc/slabinfo` or with the `slabtop` command.

The name SLUB comes from history. For many years three allocators coexisted in Linux. SLAB was the first adaptation of the design Bonwick proposed for Solaris, and it kept complicated queues. SLOB was written for small embedded systems. SLUB arrived in 2.6.22 as a simplified version of SLAB and over time became the default. SLOB was removed in 6.4 and SLAB in 6.8. Today the only allocator is SLUB.

Example 15.7

A dentry object is 192 bytes. How many dentries fit in a single-page (4 KiB) slab, and how many bytes are left over?

1. 4096 / 192 = 21.33. The integer part is 21 objects.
2. 21 × 192 = 4032. The remainder is 4096 − 4032 = 64 bytes.
3. The waste is 1.6% of the page. Had a separate 4 KiB page been allocated for each dentry, the waste would have been 95%.

15.8.6 Code trace: the SLUB free list

The secret of SLUB's speed is that it uses no extra memory to hold the free list. The inside of a free object is unused anyway. SLUB writes the “next free object” pointer into the object itself. A simplified version of the fast path looks like this:

```
void *slab_alloc(kmem_cache *c) {
    void *obj = c->freelist;      /* head of list */
    if (!obj)
        obj = new_slab(c);        /* slow path */
    c->freelist = *(void **)obj;   /* next */
    return obj;
}
void slab_free(kmem_cache *c, void *obj) {
    *(void **)obj = c->freelist;   /* inside the object */
    c->freelist = obj;
}
/* a,b = alloc; free(a); c,d,e,f = alloc */
```

`slab_alloc` takes the head of the list. If the list is empty, it enters the slow path: `new_slab` asks the buddy allocator for a page, divides it into objects, and hands out the first one. Then it advances `c->freelist` to the pointer written in the first 8 bytes of the object just taken. The expression `*(void **)obj` means "read the 8 bytes at the start of the object as a pointer." `slab_free` does the opposite: it writes the old head of the list into the freed object and makes the object the new head. The list thus works like a stack: the last one freed is the first one handed out.

The trace is done in the `kmalloc-1k` cache. A slab is a single 4 KiB page and holds four 1 KiB objects: +0, +1024, +2048, +3072. At the start the list is +0 → +1024 → +2048 → +3072 → NULL. The program first allocates a and b, frees a, and then allocates c, d, e, and f.

Operation	Object returned	freelist afterwards	Explanation
a = alloc	+0	+1024	head of the list; the pointer inside it is +1024
b = alloc	+1024	+2048	fast path: no lock, no page allocation
free(a)	—	+0	the old head (+2048) is written into +0
c = alloc	+0	+2048	the "hot" object that a has just released
d = alloc	+2048	+3072	
e = alloc	+3072	NULL	slab full
f = alloc	2:+0	2:+1024	slow path: new page, four objects

Table 15.11: Free-list trace (simulated exactly in Python; "2:" is the second slab page).

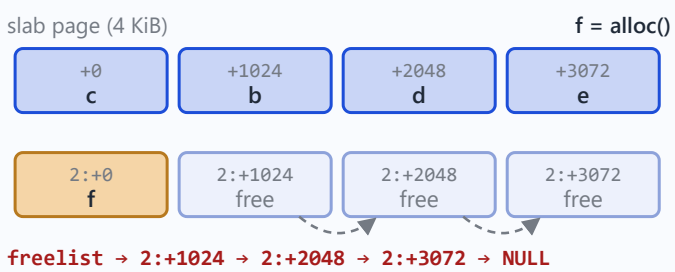


Figure 15.22: End of the trace: the first slab is full (c, b, d, e), and f is the first object of the second slab.

Three observations from the trace sum up the design of SLUB. First, c landing in a's place is no accident. The bytes of the most recently freed object are most likely still in the CPU cache; handing it out again immediately reduces cache misses. Second, the common case takes a few instructions: one load, one store, one return. The real SLUB does this step with per-CPU structures and an atomic compare-and-swap instruction instead of a lock. Third, the rare case, f, which needs a new page, is slow but correct. Because all objects are the same size, no external fragmentation arises inside a slab; every hole left behind fits the next request exactly.

> **Common mistake**
>
> Continuing to use an object that has been freed. The first bytes of a freed object are now the free list's pointer. Faulty code that writes to the object corrupts the list; the next allocation returns an arbitrary address. Bugs of this kind are a frequent source of kernel vulnerabilities. For this reason the kernel can store the pointer mixed with a secret value, and with debugging options it can fill freed objects with a pattern and check them.

15.8.7 The page cache, reclaim, and the OOM killer

Linux's memory philosophy can be summed up in one short sentence: free memory is wasted memory. Unused RAM is used for file data as the **page cache**. Once a file has been read, its pages stay in the cache; the second read does not go to the disk. This is why the “available” column in the output of `free -h` is larger than “free”: it also counts the reclaimable part of the cache. Suppose a 16 GiB machine starts with 5 GiB of application memory, 8 GiB of page cache, and 2 GiB of free memory (Table 15.12). The kernel itself and its buffers hold the remaining 1 GiB.

Stage	Anonymous	Cache	Free	Swap	Kernel's reaction
Start	5	8	2	0	—
The application grew	8,3	6	0,7	0	low watermark crossed: kswapd wakes up
Cold pages go	9	4,7	1,3	1	MGLRU, zswap → swap
Pressure continues	11,5	3,2	0,3	2	below min: direct reclaim
OOM	4,6	2	8,4	2	`mem_hog` (PID 7311) killed

Table 15.12: A 16 GiB machine under memory pressure (GiB).

Reclaim works in stages according to the watermarks (Figure 15.23).

1. **Background.** When free memory falls below the low watermark, each node's `kswapd` kernel thread wakes up. It drops clean cache pages immediately; dirty ones it writes to disk first. When free memory rises to the high level, it sleeps again. The application does not slow down noticeably in the meantime.

2. **Victim selection.** Which pages go is decided by the **multi-generational LRU** (MGLRU) that arrived with 6.1. Pages are divided into access generations; the coldest generation goes first. A cache page can be discarded, because its data is in the file on disk. Cold anonymous pages, on the other hand, have no file. zswap first compresses them in RAM and, if necessary, writes them to the swap area.
3. **Direct reclaim.** If the allocation rate outpaces reclaim and free memory falls below the min level, a process requesting memory cannot wait for pages. It performs the reclaim itself and stalls in the process. The application's latency rises noticeably.
4. **OOM.** If nothing is enough, the OOM killer (the mechanism that kills a process when memory runs out) steps in. It kills the process with the highest `oom_score` with `SIGKILL`. The score is mostly proportional to the process's memory usage. The administrator protects a process or singles it out by writing a value between −1000 and +1000 to `/proc/PID/oom_score_adj`. If a cgroup's memory limit is exceeded, the victim is chosen only from that group.

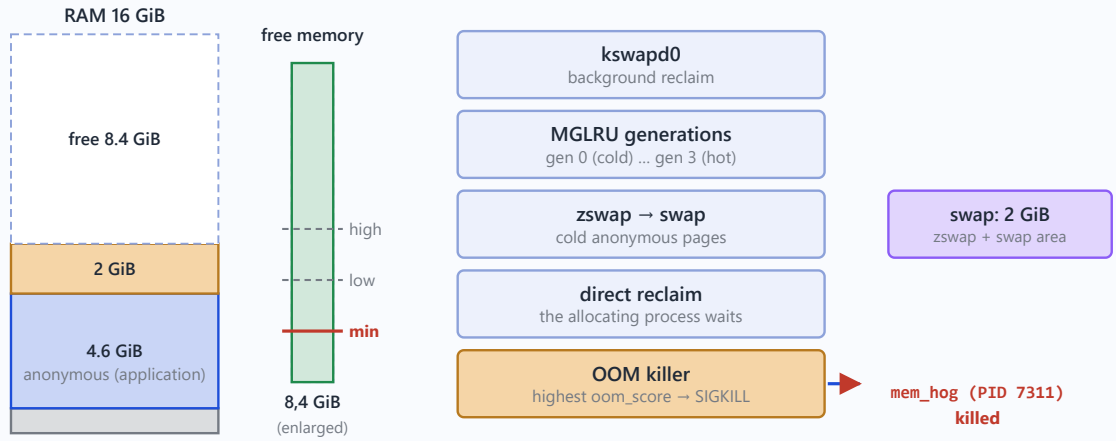


Figure 15.23: The last stage of memory pressure: the OOM killer killed `mem_hog`, and free memory rose to 8.4 GiB.

How eagerly the swap area is used is determined by the `vm.swappiness` setting; the default is 60. As the value falls, the kernel prefers dropping cache pages over sending anonymous pages to swap.

Common mistake

Mistaking the small “free” value in the `free` output for a memory shortage. On a Linux machine that has been running for a long time, free memory is always low, because RAM is full of cache. The real indicators are the “available” value, swap activity, and the memory-pressure (PSI) counters in `/proc/pressure/memory`.

15.9 Input/output

15.9.1 Devices are files: major and minor numbers

On Linux, disks and terminals are accessed through **device files** in the `/dev` directory. An application reaches a disk and a terminal with the same `open`, `read`, and `write` calls. For these files `ls -l` shows two numbers instead of a size:

```
$ ls -l /dev     # selected lines
brw-rw---- 1 root disk   8, 0 sda
brw-rw---- 1 root disk   8, 1 sda1
brw-rw---- 1 root disk 259, 0 nvme0n1
crw-rw-rw- 1 root root   1, 3 null
crw-rw-rw- 1 root tty    5, 0 tty
```

The first letter of the line is the device type. `c` is a **character device**: it provides a byte stream and is usually unbuffered; terminals, `/dev/null`, and the mouse are like this. `b` is a **block device**: it provides random access to fixed-size blocks and uses the page cache and the I/O scheduler. The two numbers in the size column are the major and minor numbers (Figure 15.24). The **major** number selects the driver: 8 is the SCSI disk driver `sd`; SATA and USB disks are attached to it as well. The **minor** number selects the unit that driver manages. The `sd` driver reserves 16 minors per disk: 0 is the whole disk, 1–15 are partitions. That is why `sda1` is 8,1; the second disk `sdb` starts at 8,16, and `sdb3` is 8,19. NVMe disks are attached to the extended block major 259 (`blkext`).

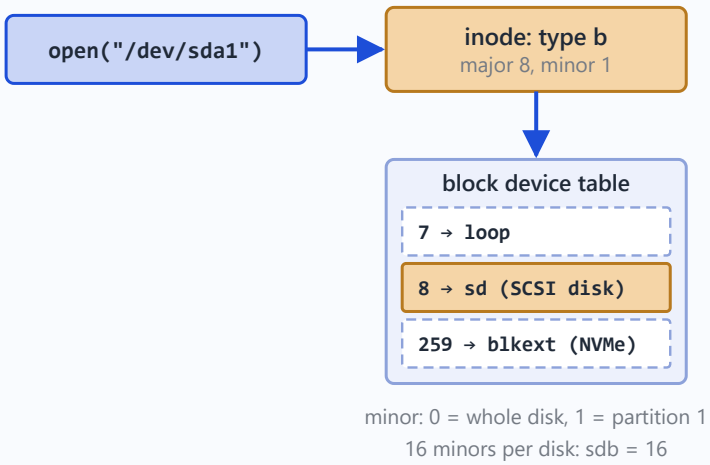


Figure 15.24: `open("/dev/sda1")`: the type and major number in the inode select the driver, and the minor number selects the partition.

A device file holds no data on disk. Its inode stores only the type and the two numbers. `open` looks up the kernel's device table with these numbers and finds the driver. Today there is no need to create the nodes in `/dev` by hand. When the kernel finds a device, it creates the node itself in the `devtmpfs` pseudo file system; `udev` sets the permissions and adds persistent names (`/dev/disk/by-uuid/…`). Network interfaces (`eth0` , `wlan0`), on the other hand, are not in `/dev` . They are reached through sockets. This is the best-known exception to the “everything is a file” principle.

15.9.2 The structure of the I/O system

The three device types descend through the kernel along three separate paths (Figure 15.25). Character devices go from the VFS to the tty layer and the line discipline, and from there to the character driver. The path to block devices passes through the file system (ext4, XFS, Btrfs), the page cache, the block layer, and the I/O scheduler. The network path starts at the socket layer: TCP or UDP, IP and the netfilter packet filter, the queueing discipline (qdisc), and the network driver. Network drivers use the NAPI interface: under heavy traffic, instead of generating an interrupt for every packet, they disable interrupts and poll for packets in batches.

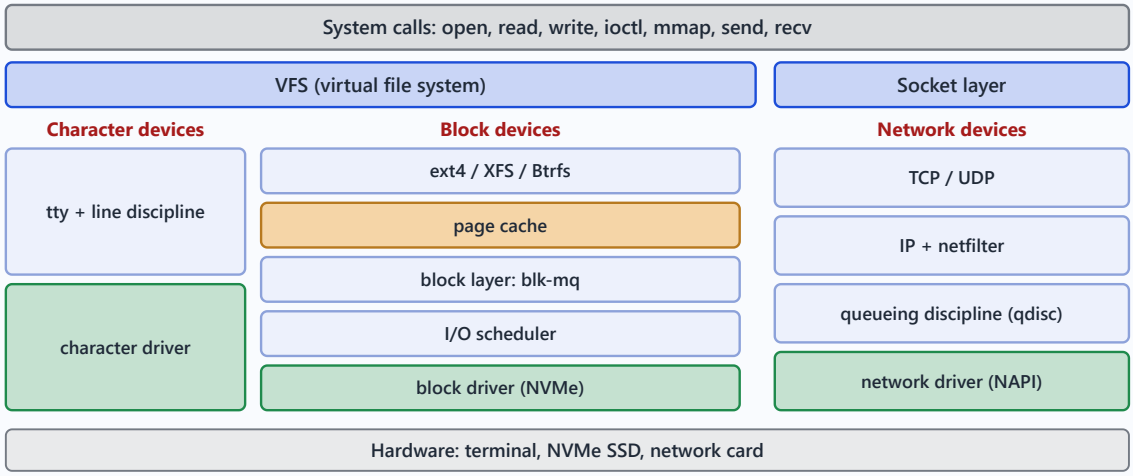


Figure 15.25: The Linux I/O system: separate layers for character, block, and network devices.

On the network path, data is carried in `sk_buff` structures. An `sk_buff` holds the packet's data and space reserved up front for headers; as each layer adds its header, it does not copy the data but only moves a pointer. With XDP and eBPF, packets can be filtered or redirected at the driver level, before entering the stack. Classic BSD had a similar three-way split. System V's module stack called STREAMS never entered the mainline Linux kernel.

15.9.3 The driver interface: `file_operations` and the terminal

Old UNIX textbooks list the separate entry points of each driver in a table: `k_open` and `k_read` for the disk, `tty_open` and `tty_read` for the terminal, and so on. Linux gathers these entry points in a single structure of function pointers:

```
static const struct file_operations my_fops = {
    .owner          = THIS_MODULE,
    .open           = my_open,
    .release        = my_release,     /* close */
    .read           = my_read,
    .write          = my_write,
    .unlocked_ioctl = my_ioctl,
    .poll           = my_poll,
    .mmap           = my_mmap,
};
cdev_init(&cdev, &my_fops);  /* major:minor → fops */
```

A character driver fills in this structure and attaches it to a major:minor range with `cdev_init`. When an application calls `read()` on the device file, the VFS routes the call to the `f_op->read` pointer of the corresponding `struct file`. The counterpart of `close` is `.release`, and it is called only when the last descriptor is closed. Device-specific commands arrive through `.unlocked_ioctl`. `.poll` lets the device be waited on with `select`, `poll`, and `epoll`. `.mmap` maps device memory directly into the process's address space. This is object-oriented design in C: the table of function pointers works like a virtual method table. The same pattern repeats throughout the VFS.

Terminal I/O is configured through the **termios** interface. `tcgetattr` and `tcsetattr` read and write the terminal parameters. In **canonical mode** (`ICANON`) the line discipline does line editing itself: backspace is handled, and the program receives the line only after Enter. In raw mode every key is delivered immediately; text editors and programs that ask for passwords use this mode. The `ECHO` flag determines whether what is typed is echoed to the screen. `cfsetispeed` and `cfsetospeed` set the input and output baud rates. Today this setting is meaningful mostly for real serial ports; terminal windows are pseudo-terminals (pty).

15.9.4 The block layer: blk-mq

The old single-queue block layer was designed for spinning disks that perform hundreds of operations per second. Every request went through a single queue and a single lock. On NVMe SSDs, which can perform millions of operations per second, this lock became a bottleneck. Today's block layer, **blk-mq**, is multi-queue. Each CPU writes its requests to its own software queue. The requests pass through an optional I/O scheduler and are distributed to the device's hardware queues. NVMe controllers offer many hardware queues; usually one queue falls to each CPU, and no lock is shared between CPUs.

There are four scheduler options. `none` does no ordering at all; on fast NVMe devices it is usually the best. `mq-deadline` orders reads and writes by deadlines and ensures that no request starves. `bfq` (budget fair queueing) shares bandwidth fairly among processes; on slow desktop disks it preserves interactivity. `kyber` adjusts the queue depth according to target latencies. The active scheduler is read per device from the file `/sys/block/<device>/queue/scheduler`; the one in square brackets is

selected. Writing a name to the file changes the scheduler. The newer asynchronous I/O interface `io_uring` is also built on top of this layer. The application and the kernel exchange requests and results through two shared ring buffers, which reduces the number of system calls.

15.9.5 The network: sockets and connection setup

A socket is the endpoint of network communication and is represented by a file descriptor. The establishment of a TCP connection can be followed through the calls on the two sides (Figure 15.26).

1. **The server prepares.** `socket()` creates a socket; it gets 3, the smallest free descriptor. `bind` attaches the socket to port 80. `listen(3, 128)` puts the socket into the listening state and opens a queue for pending connections.
2. **Waiting.** The server blocks in `accept`. The client creates its own socket.
3. **Handshake.** The client calls `connect`. The client's kernel sends SYN, the server's kernel replies with SYN-ACK, and the client completes with ACK. The two kernels carry out this three-way handshake; applications never see the packets.
4. **The new socket.** When the connection is established, `accept` returns a new socket (fd 4). fd 3 keeps listening. A single listening socket can thus give rise to thousands of connection sockets.
5. **Data and close.** Data flows with `write` and `read`. The kernel puts the data into `sk_buff`s and adds the TCP/IP headers; the driver writes the packet to the card's ring buffer. `close` sends FIN to the other side.

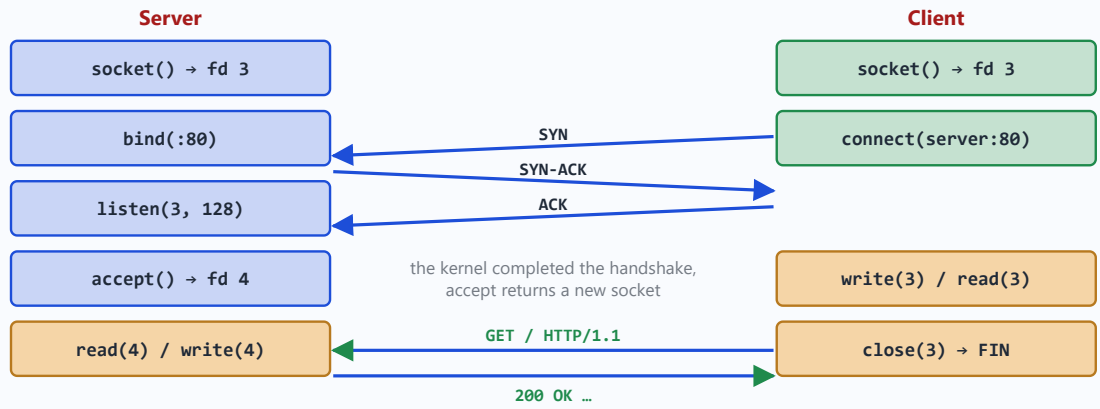


Figure 15.26: A client–server connection with TCP sockets: `socket`, `bind`, `listen`, `accept`, `connect`, the three-way handshake, data, and FIN.

A server with many clients can open a thread for every connection. With thousands of connections this becomes expensive. Linux's answer is `epoll`. The server registers the sockets it is interested in once. Then, with a single `epoll_wait` call, it gets only the list of sockets that are ready. The cost is proportional not to the total number of connections but to the number of ready sockets.

15.10 File systems

15.10.1 The directory tree and mounting

Unlike the drive letters of Windows, Linux has a single directory tree (Figure 15.27). Another file system is **mounted** on any directory of the tree. After the command `mount /dev/sdb1 /mnt/usb`, the root directory of the USB stick appears at `/mnt/usb`. The old contents of the mount point are hidden for as long as the mount lasts.

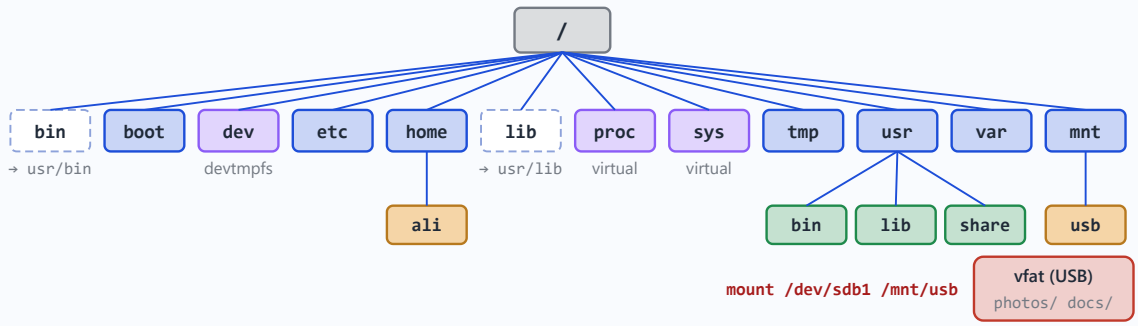


Figure 15.27: The Linux directory tree and a vfat file system mounted at the point `/mnt/usb`.

The roles of the directories are defined by the Filesystem Hierarchy Standard (FHS). `/usr` holds read-only, shareable software: commands in `/usr/bin`, shared libraries in `/usr/lib`, and kernel modules in `/usr/lib/modules`. `/etc` holds configuration files in text form. `/var` holds variable data (logs, queues, caches), `/home` the user directories, and `/root` the administrator's directory. The kernel and the initramfs are in `/boot`. `/tmp` is for temporary files and on most distributions is a tmpfs in RAM. `/dev`, `/proc`, and `/sys` are not on disk; they are pseudo file systems the kernel generates in memory. Most distributions have moved `/bin`, `/sbin`, and `/lib` under `/usr` with "usrmerge". The old paths remain as symbolic links for compatibility.

15.10.2 VFS objects

Linux uses dozens of file systems through the same calls: ext4, XFS, Btrfs, FAT, NFS, procfs. The **virtual file system** (VFS) makes this possible. The VFS defines four object types (Figure 15.28):

- **superblock**: a mounted file system. The block size, the root inode, and the file-system operations are here.
- **inode**: the file itself. Its size, permissions, times, and the location of its data. The in-memory `struct inode` is a copy of the on-disk inode.
- **dentry**: the link between a name and an inode. `"x.txt" → inode 1234`. It is kept in a cache to speed up path resolution.
- **file**: an open-file session. The file position (`f_pos`), the open flags, and the reference count.

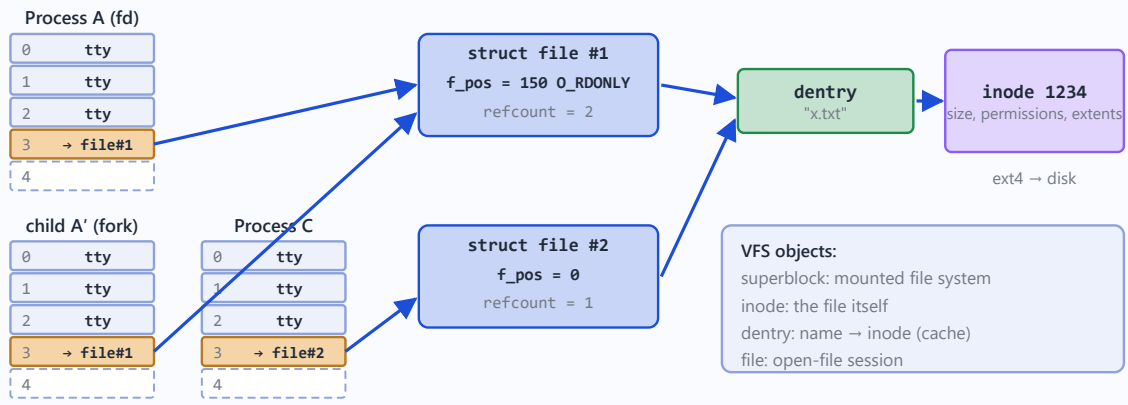


Figure 15.28: The relationship among the file-descriptor table, `struct file`, the dentry, and the inode. A and its child A′ share the same `struct file`; C has opened the same file separately.

The situation in the figure makes the three levels of open-file tables concrete. Process A opened `x.txt` and read 150 bytes. Then it called `fork`. Descriptor 3 of the child A′ points to the same `struct file #1`; its reference count is 2 and the position is shared. If A′ reads 10 more bytes, A's position also becomes 160. Process C opened the same file by itself. Its `struct file #2` is separate and its position is 0. Both `struct file`s point to the same dentry, and the dentry to the same inode. A process's descriptor table lives in `task_struct → files_struct`.

Every object has an operations table: `super_operations`, `inode_operations`, `file_operations`, `dentry_operations`. When the VFS does a job, it calls the function pointer in these tables. On ext4, `read` goes to ext4's read function; on NFS, to an RPC over the network; on procfs, to a function that turns kernel data into text on the spot. The calling program does not see the difference.

15.10.3 File and directory calls, `stat`

The system-call interface of the file system is small (Table 15.13). A few details are often confused. `rmdir` removes only an empty directory. `rename` is atomic: if the target name exists, it is replaced in one step; the safe way to update a file is to write the new contents to a temporary file and `rename` it. `opendir`, `readdir`, and `closedir` are not system calls; they are library functions built on `getdents64`.

Call	Function
`open`, `creat`, `close`	Open / create and return an fd; close the descriptor
`read`, `write`, `lseek`	Read / write at the position; move the position
`stat`, `fstat`, `lstat`	Get information (`lstat`: the link itself)
`pipe`, `dup2`, `fcntl`	Create a pipe; duplicate an fd; locks and flags
`mkdir`, `rmdir`	Create / remove a directory (if empty)
`link`, `unlink`, `symlink`	Add / remove a name; create a symbolic link
`rename`, `chdir`	Move (atomic); change the working directory

Call	Function
`getdents64`	Read directory entries

Table 15.13: File and directory system calls.

`stat` resolves a path, reads the inode, and copies its fields into `struct stat` (Figure 15.29). `st_ino` is the inode number, `st_mode` the type and permission bits, `st_nlink` the hard-link count, and `st_size` the size in bytes. `st_blocks` gives the space allocated on disk in 512-byte units; for a sparse file it can be much smaller than `st_size`. There are three timestamps: `st_atime` is the last access, `st_mtime` the last content change, and `st_ctime` the last inode change. `st_ctime` is *not* the creation time; `chmod` or `link` updates it as well. The creation time can be read only with the newer `statx()` call, from the `stx_btime` field.

Figure 15.29: `stat`: the path is resolved, the inode is read, and the fields are copied into `struct stat`.

15.10.4 File locking

Processes that share a file can lock byte ranges with `fcntl`. A lock is described with a `struct flock`: the type (`F_RDLCK` shared, `F_WRLCK` exclusive), the start (`l_start`), and the length (`l_len`). Shared locks may overlap one another; an exclusive lock may overlap no lock at all. This is the byte-level counterpart of the readers–writers lock.

Example 15.8

On a 16-byte file, A places a shared lock on bytes 4–7, B on 6–9, and C on 2–11, in that order. Then D asks for an exclusive lock on byte 9 alone. Which requests are granted?

1. A: `F_RDLCK`, `l_start` = 4, `l_len` = 4. There is no lock on the file; granted.
2. B: the range 6–9 overlaps A at 6–7. Both are shared; granted.
3. C: 2–11 covers all of A and B. Shared again; granted. Bytes 6 and 7 are now covered by three locks.
4. D: byte 9 is within the ranges of B and C. An exclusive lock conflicts with shared locks. If requested with `F_SETLK`, the call returns an error without waiting (`EAGAIN` or `EACCES`). If requested with `F_SETLKW`, D waits until B and C release their locks.

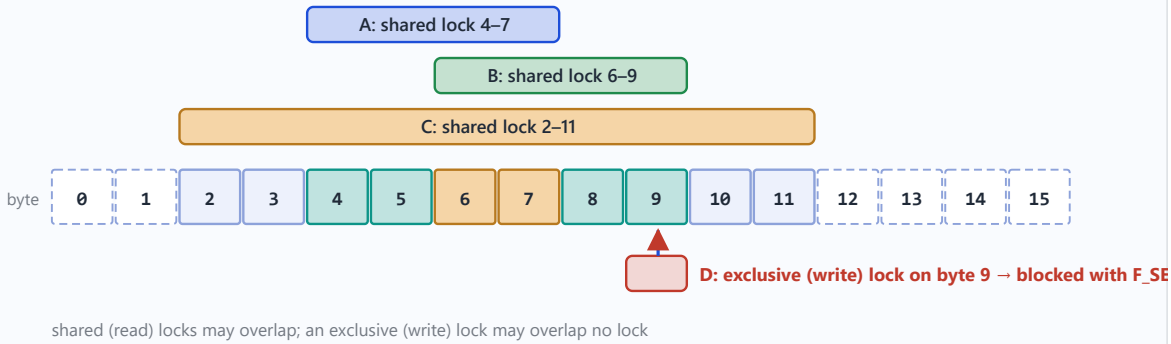


Figure 15.30: fcntl byte-range locks: three shared locks and a pending exclusive lock request for byte 9.

Two processes waiting with `F_SETLKW` may end up waiting for each other's locks. The kernel tracks the wait relationships; if it detects a cycle, it returns `EDEADLK` to the waiting call. This is a real example of deadlock detection. These locks are **advisory**: a process that does not ask for the lock can still write to the file. For a whole file there is the simpler `flock()`. An important flaw of `fcntl` locks is that they belong to the process. When the process closes any descriptor of the file, all its locks are dropped. Threads in the same process also cannot tell the locks apart. **OFD locks** (`F_OFD_SETLK`), tied to the open file description, fix this flaw.

15.10.5 The disk layout of ext2 and ext4

The layout of the classic UNIX file system was simple: a boot block, a superblock, an inode table, and data blocks. In this layout a file's inode could be at the start of the disk while its data was anywhere; the disk arm kept travelling between the two. ext2 divided the disk into **block groups** (Figure 15.31). Each group has its own block bitmap, inode bitmap, inode table, and data blocks. The kernel tries to place a file's inode and data in the same group, and a directory's files near the directory's group. The superblock and the group descriptors are backed up in some groups; if the first copy is damaged, it is recovered from the backup.

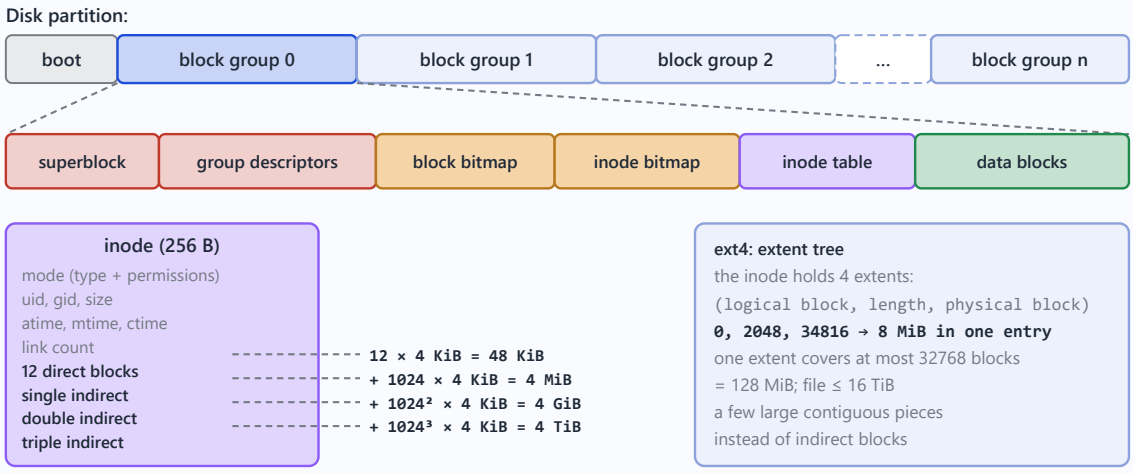


Figure 15.31: The ext2/ext4 disk layout: block groups, the inode's block pointers, and ext4 extents.

An ext2 inode holds 12 direct block pointers plus single, double, and triple indirect pointers. With 4 KiB blocks, one indirect block holds 1024 pointers. The reachable space grows multiplicatively: 12 × 4 KiB = 48 KiB, + 1024 × 4 KiB = 4 MiB, + 1024^2 × 4 KiB = 4 GiB, + 1024^3 × 4 KiB = 4 TiB. The theoretical limit with indirect blocks is thus about 4 TiB. The real limit in ext2, however, is 2 TiB: the 32-bit `i_blocks` field in the inode counts allocated space in 512-byte units and runs out at 2^{32} × 512 B = 2 TiB. The inode number of the root directory is always 2; 1 was formerly reserved for the list of bad blocks.

For large files, ext4 uses **extents** instead of indirect blocks. An extent describes a contiguous range of blocks with three numbers: logical block, length, and physical block. The extent `(0, 2048, 34816)` says that the first 2048 blocks of the file, that is, its first 8 MiB, sit contiguously on disk starting at block 34816. With indirect blocks the same information meant 2048 pointers and two indirect blocks. Four extents fit in the inode; if more are needed, the extents are placed in a tree rooted at the inode. A single extent covers at most 32,768 blocks, that is, 128 MiB with 4 KiB blocks. With 4 KiB blocks a file can be at most 16 TiB. ext4 also uses delayed allocation: it does not decide where the data blocks go until the moment the data is to be written to disk, so it can choose large contiguous extents.

15.10.6 Journaling in ext4

A `write()` followed by `fsync()` changes several blocks. The data itself (D) is written; alongside it the metadata changes: the inode (M1), the block bitmap (M2), and the extent (M3). If the power fails after some of these blocks have been written and before the others, the file system is left inconsistent. For example, the bitmap shows a block as in use but no inode points to that block. ext4 solves this problem with **journaling**. The journal is managed by the JBD2 layer (Figure 15.32).

1. **Data first.** In the default `data=ordered` mode, D is written first to its final place. The metadata therefore never points to a block containing garbage.

2. **The journal transaction.** M1, M2, and M3 are written to the journal as a single transaction (T42): a descriptor block, copies of the metadata blocks, and, at the end, a **commit** record. The moment the commit record reaches the disk, the transaction is durable.
3. **Checkpoint.** The metadata blocks are then written to their final places, without hurry. The space in the journal is reused; the journal is a circular buffer.
4. **Recovery.** After a crash, a full fsck scan is not performed at mount time. The journal is read. Transactions that have a commit record are replayed. Incomplete transactions without a commit are discarded, and the file system stays in its old consistent state. This takes seconds.

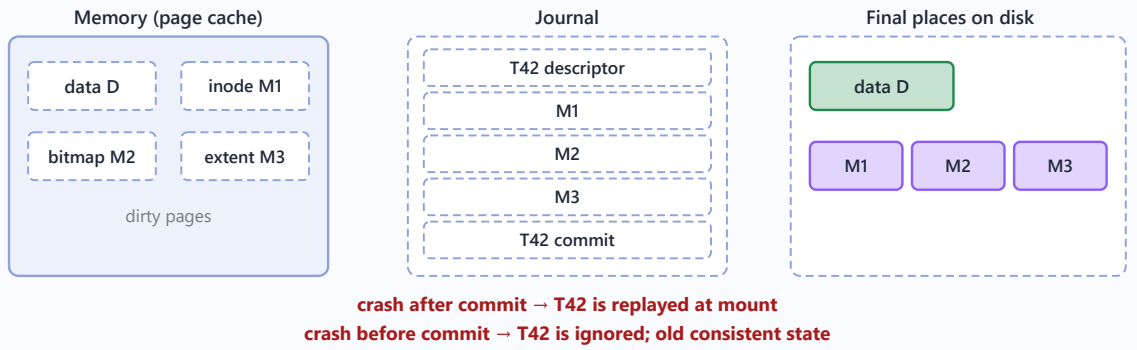


Figure 15.32: ext4 ordered journaling: data, journal transaction T42, commit, and checkpoint; two outcomes depending on whether the crash comes before or after the commit.

The other two modes offer different trade-offs. `data=journal` writes the data to the journal too: it is the safest, but the slowest, because every data block is written twice. `data=writeback` gives no ordering guarantee between data and metadata: it is fast, but after a crash a file may show old blocks or blocks containing garbage. By default ext4 commits about every five seconds. An append for which `fsync` was not called can be lost entirely if the power fails before the commit; but thanks to `data=ordered`, the file never shows garbage. XFS also uses metadata journaling. Btrfs, instead of a journal, stays consistent through copy-on-write.

> **Common mistake**
>
> Believing that journaling protects data. In the default mode the journal protects only metadata and guarantees only *consistency*. The application's latest writes can be lost without `fsync`. A program that wants to be sure data has reached the disk must call `fsync`.

15.10.7 Directories and path resolution

In ext2/ext4, a directory is an ordinary file made of entries. Each entry holds the inode number, the record length, the name length, the file type, and the name itself. A deleted entry is not destroyed; its space is added to the record length of the previous entry. In directories with thousands of entries,

linear search becomes slow. For such directories ext4 uses a tree (htree) ordered by the hash of the names.

Example 15.9

How is `open("/home/ali/notes.txt")` resolved? (Figure 15.33)

1. The path is absolute; the search starts at the root. The root directory's inode (2) is read. `home` is looked up among the entries in its data blocks: 131073. The x (search) permission is checked for the root.
2. `ali` is looked up in the data of the `home` directory (131073): 131074.
3. `notes.txt` is looked up in the `ali` directory (131074): 132010. Had the path been relative, the search would have started at the process's working directory.
4. Inode 132010 is read. The permissions are checked; if read permission is granted, a `struct file` is created and the smallest free descriptor, for example 3, is returned.

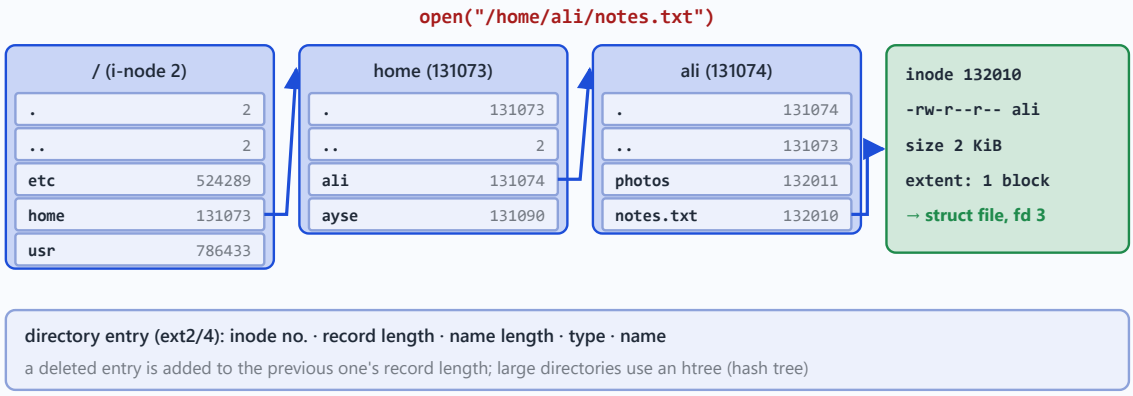


Figure 15.33: Resolving the path `/home/ali/notes.txt` : inode 2 → 131073 → 131074 → 132010.

Inode numbers are not random. The closeness of 131073 and 132010 reflects ext4's effort to put a directory's files in the same block group. If every step required a disk read, path resolution would be very slow. The **dentry cache** (dcache) keeps name → inode mappings in memory. Frequently used paths are resolved without ever going to disk. The dcache also stores negative results: the fact that "there is no such name in this directory" is cached too. A program that looks up a nonexistent file on every open therefore does not burden the disk.

15.10.8 Code trace: the link count and `unlink`

On Linux, a file's name and the file itself are separate things. The name is a directory entry; the file is the inode. Two kinds of links make use of this separation (Figure 15.34):

```
$ ln a.txt backup/b.txt     # hard
$ ln -s a.txt s            # symbolic
$ ls -li
500 -rw-r--r-- 2 ali ali 9 a.txt
777 lrwxrwxrwx 1 ali ali 5 s -> a.txt
```

A **hard link** is a second name given to the same inode. `a.txt` and `backup/b.txt` point to the same inode (500), and the link count is 2. A **symbolic link**, on the other hand, is a separate inode (777); its content is just the path of the target. `ls -l` shows it with type `l` and in the form `s -> a.txt`. Its size is 5, because `"a.txt"` is five characters. A hard link cannot cross a file-system boundary, because an inode number is meaningful only within its own file system. It also cannot be made to directories, because it could create a cycle in the tree. A symbolic link can point to another file system and to directories. The price is that the link **dangles** when the target is deleted.

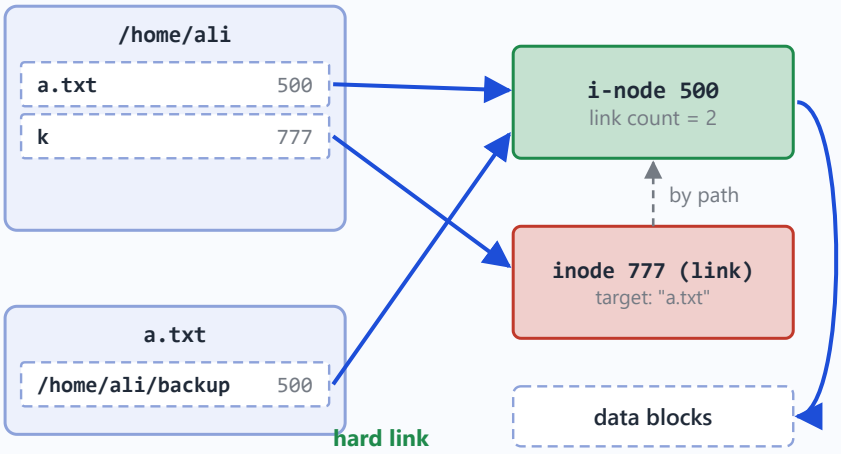


Figure 15.34: A hard link is a second name pointing to the same inode; a symbolic link is a separate inode that contains the path of the target.

An inode carries two counters. The **link count** counts the directory entries on disk; the **open-file count** counts the references in memory. The data is freed only when both drop to zero. The following program changes these two counters step by step:

```c
int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
write(fd, "data", 4);
link("a.txt", "b.txt");      /* hard link */
symlink("a.txt", "s.txt");   /* symbolic */
unlink("a.txt");
int r1 = open("b.txt", O_RDONLY);
int r2 = open("s.txt", O_RDONLY);
unlink("b.txt");
close(r1);
close(fd);
```

Line	Directory entries	812 links	812 open	Result
1–2 `open`, `write`	a.txt → 812	1	1	fd = 3
3 `link`	a.txt, b.txt → 812	2	1	the data is not copied
4 `symlink`	+ s.txt → 813 ("a.txt")	2	1	812 does not change
5 `unlink("a.txt")`	b.txt, s.txt	1	1	data reachable through b.txt
6 `open("b.txt")`	b.txt, s.txt	1	2	r1 = 4
7 `open("s.txt")`	b.txt, s.txt	1	2	r2 = −1, `ENOENT`
8 `unlink("b.txt")`	s.txt	0	2	no name, but the data remains
9 `close(r1)`	s.txt	0	1	fd 3 still works
10 `close(fd)`	s.txt	0	0	inode and blocks freed

Table 15.14: Link and open-file counters (verified in Python with a directory and inode simulation).

The meaning of the lines is as follows. `open` allocates a new inode (812) and adds the entry `a.txt → 812` to the directory. `link` only writes a new directory entry; a.txt and b.txt are equal, and the question of which one is the "original" has no answer. `symlink` creates a separate inode (813) and does not touch 812. `unlink("a.txt")` removes one name and lowers the link count to 1; "deleting" a name is not deleting the file. b.txt opens without trouble, and the smallest free descriptor, 4, is returned. When s.txt is opened, the kernel follows the path inside it, `"a.txt"`. That name no longer exists: the symbolic link dangles, and `open` returns −1. `unlink("b.txt")` removes the last name too. The link count is 0, but since two descriptors are open, the inode and the data are not yet freed. `ls` no longer shows the file, yet reading and writing through fd 3 work. When the last descriptor is closed, both counters drop to zero and the blocks are freed.

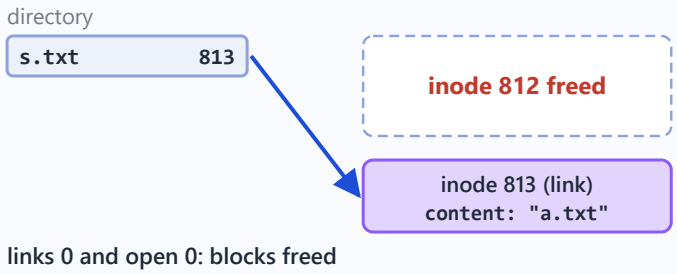


Figure 15.35: End of the trace: inode 812 is freed; only the dangling symbolic link s.txt (813) remains.

The well-known pattern for temporary files relies on this behavior. The program opens the file and immediately calls `unlink`. The file appears in no directory, and no other process can find it by name. Even if the program crashes, the kernel closes the descriptors and the space returns by itself. Linux also offers the `O_TMPFILE` flag for this pattern: the file is created nameless from the start.

Common mistake

Deleting a large log file to free disk space and expecting the space to come back at once. As long as the process writing to the file keeps the descriptor open, the blocks are not freed; `df` does not change. Such deleted but open files can be found with `lsof +L1`. The right way is to restart the process, or to truncate the file instead of deleting it.

15.10.9 Beyond ext4

The file systems Linux supports answer different needs (Table 15.15). ext4 is the default on most distributions because it is mature, fast, and has a reliable repair tool. XFS is strong in parallel I/O and with very large files; it is the default of RHEL. F2FS is a log-structured design suited to flash memory and is common on Android phones.

File system	Notable feature	Typical use
ext4	Journaling, extents, mature and fast fsck	Default on most distributions
XFS	Parallel I/O, large files, delayed allocation	RHEL default, large servers
Btrfs	Copy-on-write, snapshots, checksums, compression, subvolumes	Default on Fedora and openSUSE
F2FS	Flash-friendly log-structured design	Android phones
tmpfs	Entirely in RAM (page cache + swap)	`/tmp`, `/run`, `/dev/shm`
NFS, SMB	Network file systems	Shared home directories

Table 15.15: File systems commonly used on Linux.

Btrfs's method of staying consistent differs from journaling (Figure 15.36). No block is ever overwritten. When block B4 changes, the new content is written to a free place, B4′. The node pointing to B4 must also change; a new copy of it (node′) is written as well. This chain continues up

to the root, and a new root (v2) is created. The last step is for the superblock to point to the new root; this is a single atomic write. If the crash comes before this write, the old root (v1) is entirely valid. Keeping the old root gives a **snapshot** for free: the tree reached through v1 is the file system as it was before the change.

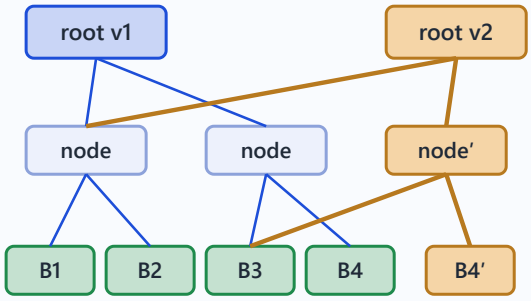


Figure 15.36: Copy-on-write in Btrfs: when B4 changes, only the path from the root to B4 is copied; the old root becomes a snapshot.

Thanks to the VFS, all of these file systems are used with the same system calls. Which file system is mounted on which directory is shown by `findmnt` or `df -T`.

15.10.10 `/proc` and `/sys`

Linux also presents the kernel's internal state as files. These files are not on disk; they are generated on the kernel side at the moment they are read.

```
$ cat /proc/loadavg
0.42 0.37 0.30 2/1234 98765
$ grep -E "MemTotal|MemAvailable" /proc/meminfo
MemTotal:       16303452 kB
MemAvailable:    9876544 kB
$ ls /proc/self
cmdline  cwd  environ  exe  fd  maps  status  …
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ sysctl vm.swappiness        # = /proc/sys/vm/swappiness
vm.swappiness = 60
```

`/proc` (procfs) carries two kinds of information. There is a directory for every process: `/proc/PID/status` shows the state and memory usage, `maps` the VMAs, and `fd/` the open descriptors. `/proc/self` is the directory of the reading process itself. Kernel statistics are here too. The first three numbers in the `/proc/loadavg` output are the load averages of the last 1, 5, and 15 minutes. `2/1234` gives the number of running and total tasks; the last number is the most recently

assigned PID. The `/proc/meminfo` lines in the example say that 9.42 GiB (60.6%) of the total 15.55 GiB of memory is available. `ps`, `top`, and `free` read their data from here. The files under `/proc/sys` are tunable kernel parameters. `sysctl vm.swappiness` actually reads the file `/proc/sys/vm/swappiness`.

`/sys` (sysfs) presents the kernel's device model. Every device, driver, and bus is a directory. Each file holds a single value. The file `/sys/block/nvme0n1/queue/scheduler` shows the blk-mq scheduler: `none`, in square brackets, is selected. `/sys/fs/cgroup` is the interface of cgroup v2; resource limits are set by writing to the files in this directory.

15.11 Security

15.11.1 `chmod` and `umask` arithmetic

On Linux every file has three permission bits each for the owner, the group, and others: r for read, w for write, x for execute. In front of them stand three special bits: setuid, setgid, and the sticky bit. The twelve bits are written with four octal digits. Each octal digit is three bits: 7 = 111 = rwx, 5 = 101 = r-x, 4 = 100 = r--. `chmod 754 script.sh` therefore gives `rwxr-xr--`.

The symbolic form of `chmod` modifies the existing bits. `g+w` is a bitwise OR (OR 020), and `o-r` is bitwise masking (AND ~004). When `chmod g+w,o-r` is applied to 0754, the group write bit is set and the others' read bit is cleared: the result is 0770, `rwxrwx---` (Figure 15.37).

Figure 15.37: `chmod g+w,o-r`: 0754 OR 020 AND ~004 = 0770.

Definition 15.9

The **umask** is the mask of permission bits to be removed from the files a process newly creates. A new file asks for 666 and a new directory for 777. The actual permission is request & ~umask.

Example 15.10

With umask 022 and 027, what permissions do `touch a` and `mkdir d` produce?

1. umask 022: file 666 & ~022 = 0644 (`rw-r--r--`), directory 777 & ~022 = 0755 (`rwxr-xr-x`). This is the default on most distributions.
2. umask 027: file 666 & ~027 = 0640 (`rw-r-----`), directory 777 & ~027 = 0750 (`rwxr-x---`). Others can see nothing.
3. Octal subtraction happens to give the right result in the first case; in the second it produces a value such as 666 − 027 = 637 (`rw--wxrwx`) that turns on wrong bits. umask clears bits; it does not subtract.

Each of the three special bits solves a different problem (Figure 15.38). **setuid** (4xxx), on an executable file, makes the effective user ID of the running process that of the file's owner. The `passwd` command has permission 4755 (`rwsr-xr-x`): when an ordinary user runs it, the process can write to `/etc/shadow` with root privilege. **setgid** (2xxx) does the same for the group; on a directory, it makes new files inherit the directory's group. The **sticky bit** (1xxx) is meaningful on directories. `/tmp` has permission 1777 (`rwxrwxrwt`): anyone can write, but a file can be deleted and renamed only by the file's owner, the directory's owner, or root (or a process with the `CAP_FOWNER` capability). An s or t that takes the place of x in the symbolic notation shows that the underlying x bit is also set. If x is off, an uppercase letter (S, T) is written.

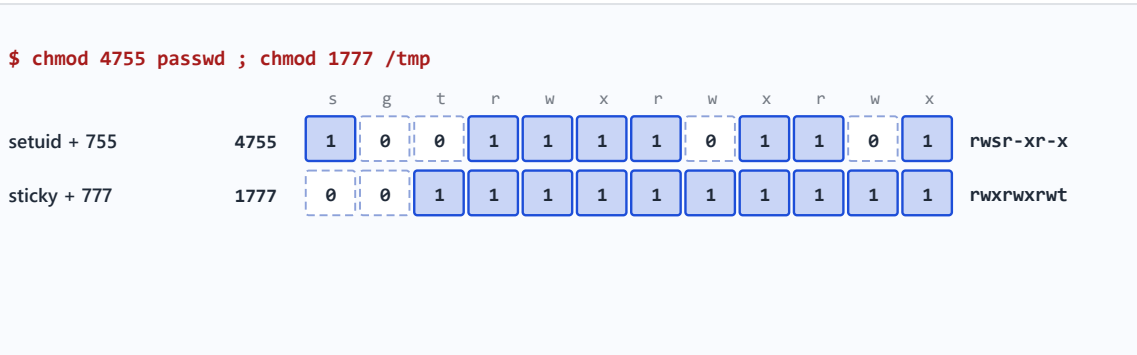


Figure 15.38: setuid + 755 = 4755 (`rwsr-xr-x`) and sticky + 777 = 1777 (`rwxrwxrwt`).

The results can be checked with `stat -c "%a %A" file` ; the command prints the octal and symbolic permissions together. `umask -S` shows the mask in symbolic form.

15.11.2 Access control, root, and capabilities

When a process accesses a file, the kernel looks at the process's effective identities and selects a single permission class:

1. If the effective uid is 0 (root), permission is granted. Only for an execute request is at least one x bit required on the file.

2. If the effective uid is the file's owner, only the owner bits are examined.
3. If the effective gid or one of the supplementary groups is the file's group, only the group bits are examined.
4. Otherwise the others bits are examined.

The first matching class wins; later classes are never looked at. That is why the owner cannot read a file with permission `----rw-rw-`, while group members and others can. The owner can, if they wish, give the permission back to themselves with `chmod`.

A process has two kinds of identities. The **real** uid is the user who started the process. The **effective** uid is the identity used in permission checks; in setuid programs the two diverge. The `access()` call deliberately tests access with the real uid. It lets a setuid program answer the question "can this user really read this file?" `setuid` and `setresuid` change identities; a well-written service uses these calls to drop its privileges once its privileged work is done. User and group IDs are kept in the files `/etc/passwd` and `/etc/group`. Password hashes are in `/etc/shadow`, which only root can read. `sudo` gives an authorized user root privilege on a per-command basis.

In the classic model, root can do everything and everyone else can do no privileged work. This "all or nothing" approach is dangerous. `ping` had to be setuid root just to open a raw socket. A bug inside it opened the whole system to an attacker.

Definition 15.10

A **capability** is a piece of root's privileges that can be granted and taken away independently. Linux divides root's power into more than 40 capabilities. Examples: `CAP_NET_RAW` is for opening raw sockets, `CAP_NET_BIND_SERVICE` for binding to ports below 1024, `CAP_SYS_MODULE` for loading modules, and `CAP_DAC_OVERRIDE` for bypassing file permissions.

After the command `setcap cap_net_raw+ep /usr/bin/ping`, `ping` can open raw sockets without being setuid root. Even if a vulnerability is found, the attacker can only open raw sockets; they cannot read files or load modules. Capabilities are kept as per-process sets and stored in the `cred` structure of `task_struct`.

15.11.3 Code trace: the file permission check

The kernel's permission check is done, roughly, by the following function. `req` is a bit mask: read is 4, write is 2, execute is 1.

```
int has_perm(cred *c, inode *f, int req) {
    int bit, m = f->mode;       /* r4 w2 x1 */
    if (c->euid == 0)           /* root */
        return !(req & 1) || (m & 0111);
    if (c->euid == f->uid)
        bit = (m >> 6) & 7;     /* owner */
    else if (in_group(c, f->gid))
        bit = (m >> 3) & 7;     /* group */
    else
        bit = m & 7;            /* other */
    return (bit & req) == req;
}
```

The return expression in the root branch is a little surprising. `!(req & 1)` is true if the request does not include execute; for read and write, the permission bits are never looked at. If execute is requested, `m & 0111` looks for at least one x bit among owner, group, or others. Even root cannot execute a file that has no x bit at all, because such a file is most likely not a program. The following `if` chain selects a single class. The three selected bits are taken by shifting and masking: `(m >> 6) & 7` for the owner, `(m >> 3) & 7` for the group, and `m & 7` for others. The last line tests whether all the requested bits are set.

Let the file `report.txt` be owned by ali (uid 1000), with group team and mode 0640 (`rw-r-----`). veli (1001) is in the team group; can (1002) is not.

Who, request	Mode	Class	Bits	Test	Result
ali, w (2)	0640	owner	6 (rw-)	6 & 2 == 2	granted
veli, w (2)	0640	group	4 (r--)	4 & 2 = 0 ≠ 2	denied
veli, r (4)	0640	group	4 (r--)	4 & 4 == 4	granted
can, r (4)	0640	other	0 (---)	0 & 4 = 0 ≠ 4	denied
root, r (4)	0640	root	—	no x in the request	granted
ali, r (4)	0066	owner	0 (---)	0 & 4 = 0 ≠ 4	denied

Table 15.16: Trace of `has_perm` (the function was translated to Python line by line and run).

Figure 15.39: With mode 0066, ali falls into the owner class, where there are no bits at all: denied (`EACCES`).

The last row is the most instructive case of the trace. With mode 0066 (`----rw-rw-`), can and veli can read the file. ali, being the file's owner, falls into the owner class; group membership is never looked at. Since the owner bits are empty, the request is denied. In the real kernel there are other layers before and after this check: access control lists (ACLs), capabilities, and security modules.

15.11.4 Containers: namespaces and cgroups

A container is not a separate object in the Linux kernel. It is ordinary processes isolated by two kernel features: namespaces and cgroups.

> **Definition 15.11**
>
> A **namespace** is a kernel feature that gives a group of processes a separate view of system resources. The processes in a namespace see only the PIDs, mount points, network interfaces, or user IDs of that view. A **control group** (cgroup) is a kernel feature that limits and measures the resource use of a group of processes, such as CPU, memory, I/O, and the number of processes.

A short rule separates the two: **a namespace determines what is seen; a cgroup determines how much is used**. Linux has eight namespace types (Table 15.17). Each is selected by a flag in the `clone` , `unshare` , or `setns` call. Which namespaces a process is in can be read from the links in the directory `/proc/PID/ns/` ; if two processes are in the same namespace, the links show the same number.

Namespace	Flag	Resource it isolates
PID	`CLONE_NEWPID`	Process IDs; the first process inside is PID 1
Mount	`CLONE_NEWNS`	Mount points, the file-system tree
Network	`CLONE_NEWNET`	Network interfaces, routing table, ports, firewall
UTS	`CLONE_NEWUTS`	Hostname and domain name
IPC	`CLONE_NEWIPC`	System V IPC, POSIX message queues
User	`CLONE_NEWUSER`	uid/gid mapping: root inside is an ordinary uid outside
cgroup	`CLONE_NEWCGROUP`	The visible root of the cgroup tree
Time	`CLONE_NEWTIME`	Boot-time and monotonic clocks (5.6+)

Table 15.17: Linux namespaces.

The mount namespace was the first historically. Since nobody then expected other types, the flag is named just `CLONE_NEWNS` . The user namespace is special: even an unprivileged process can create it and be root inside. The uid 0 inside is mapped to, for example, uid 100000 outside. The root inside has authority only over resources in its own namespace. “Rootless” containers rely on this feature.

What a container runtime (runc, crun) does can be followed in five steps (Figure 15.40).

1. **Namespaces.** New namespaces are created with `clone()` or `unshare()`: its own PID numbers, mount tree, network stack, hostname, IPC, user mapping, cgroup, and clock view.
2. **Root file system.** In the mount namespace, `pivot_root` switches to the image's root file system. The root is usually overlayfs: the image's read-only layers are stacked, and a writable layer private to the container is added on top. A hundred containers running from the same image share the lower layers.
3. **Resource limits.** Limits are written to the cgroup v2 directory: `cpu.max` = `"50000 100000"`, `memory.max` = 512M, `pids.max` = 100.
4. **Attack surface.** Unneeded capabilities are dropped. The `no_new_privs` flag is set, so even setuid programs cannot grant privilege. A seccomp filter that blocks dozens of risky system calls and an AppArmor or SELinux profile are applied.
5. **Launch.** Finally `execve` runs the application. Inside, the application is PID 1; on the host it is PID 48213.

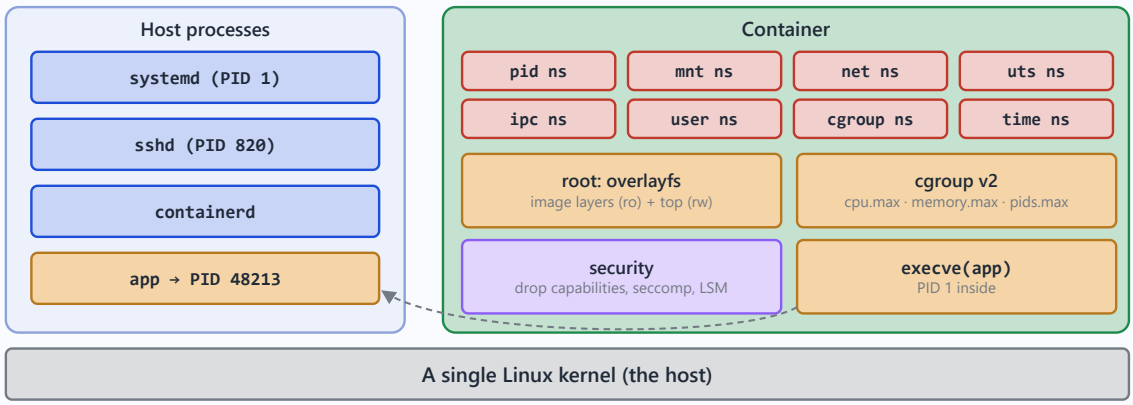


Figure 15.40: Setting up a container: eight namespaces, an overlayfs root, cgroup v2 limits, and security settings, all on a single host kernel.

Example 15.11

What does `"50000 100000"` in the `cpu.max` file mean? What is written to give a container exactly two CPUs?

1. The format is `"quota period"`, both in microseconds. In each period of 100,000 µs (100 ms), the group may use a total of 50,000 µs of CPU time.
2. 50,000 / 100,000 = 0.5: the group gets half a CPU's worth of time. If it exhausts the quota, it is not run for the rest of the period, even if there are idle CPUs on the machine.
3. For two CPUs the quota must be twice the period: `"200000 100000"`. The quota can be spread over several CPUs; the group's two threads can run on two cores at the same time.

In cgroup v2, all controllers (`cpu`, `memory`, `io`, `pids`) share a single hierarchy. In the old v1, each controller had its own tree, and a process could end up in inconsistent groups in different trees. If the memory limit is exceeded, the OOM killer chooses a victim only from that group. systemd also puts every service in its own cgroup. `systemd-cgls` shows the tree and `systemd-cgtop` the groups' live usage.

Containers all share the same kernel. This makes them much lighter than virtual machines: a container starts in milliseconds and spends no memory on a kernel of its own. The price is weaker isolation. A vulnerability in the kernel can affect all containers at once. With a virtual machine, an attacker must first break through the guest kernel and then the hypervisor.

Common mistake

Treating a namespace as a security boundary and a cgroup as a visibility tool. A PID namespace hides processes but does not limit memory use; an unlimited container can consume the host's memory. Conversely, a cgroup limits memory but does not hide the processes on the host. For a container, both are needed together.

15.11.5 seccomp and Linux security modules

A system call passes through several checks before reaching its object (Figure 15.41). As an example, let an application call `open("/etc/shadow")`.

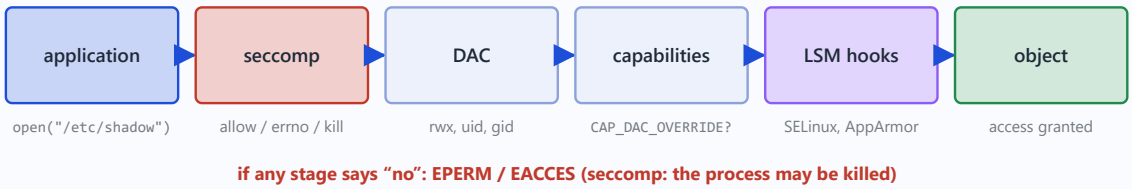


Figure 15.41: A system call passing through the seccomp, DAC, capability, and LSM checks. If any stage says “no,” the call returns with `EPERM` or `EACCES`.

1. **seccomp.** If the process has installed a seccomp-BPF filter on itself, the call passes through it first. The filter is a small BPF program that looks at the call's number and arguments. It can allow the call, return an error code, or kill the process. Chrome's renderer processes, systemd services, Docker, and Android applications use these filters.
2. **DAC.** Discretionary access control looks at the rwx bits, the uid, and the gid. It is called discretionary because the file's owner can change the permissions at will.
3. **Capabilities.** If DAC denies, the kernel checks whether the process has a capability that overrides the denial, for example `CAP_DAC_OVERRIDE`.

4. **LSM.** Finally the hooks of the Linux security modules run. These implement mandatory access control: the rules are set by the administrator, and even root cannot bypass them.

There are three main security modules. **SELinux** is label-based: every process and file has a type, and the policy says what each type may do to each other type. It is used on RHEL, Fedora, and Android. **AppArmor** uses path-based profiles: "this program may access only these files." It is easier to write and is common on Ubuntu, SUSE, and Debian. **Landlock** (5.13) lets an unprivileged process put itself in a sandbox: "from now on I can write only under `./data`." Once a process has set this restriction, it cannot lift it; its children inherit it too.

15.12 Android

Android is a system that runs on the Linux kernel but does not use the GNU user space (Figure 15.42). Its C library is Bionic, written by Google. Applications are written in Java or Kotlin and run on the Android Runtime (ART). The hardware abstraction layer (HAL) separates vendors' drivers from the framework. Android brought three important additions to the kernel: the Binder IPC mechanism, wakelocks for power management, and mandatory SELinux. Binder eventually entered the mainline kernel as well.

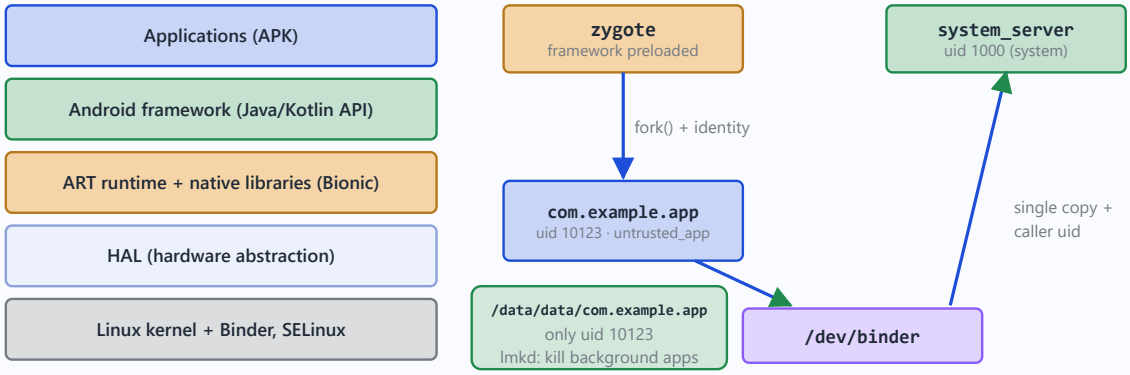


Figure 15.42: Android architecture, app launch through zygote, Binder IPC, and the application sandbox.

The launch of an application is an interesting use of `fork`. At boot, `init` starts a process called **zygote**. zygote loads ART and the framework's thousands of classes and resources once, and waits. When an application is opened, zygote calls `fork`. The loaded pages are shared with COW; each application need not reload the classes, and startup is faster. Memory is saved too, because the framework's pages are a single copy across all applications. The child process then switches to the application's unique uid, its SELinux context (`untrusted_app`), and its seccomp filter. No `exec` follows the `fork` here; the image that is already ready is used.

An application calls a system service such as location or the camera through **Binder**. Binder is a remote procedure call mechanism that works over the `/dev/binder` device. The kernel moves the data to the receiver's pre-mapped buffer with a single copy. It also passes along the caller's uid. The

application cannot forge this uid, because the kernel adds it. The service checks permission accordingly.

Android's sandbox rests on Linux's oldest protection mechanism: every application is a separate Linux user. The application in the figure runs with uid 10123, and the directory `/data/data/com.example.app` is open only to this uid. If an application tries to read another application's files, ordinary file permissions stop it. When memory runs low, `lmkd` in user space steps in instead of the kernel's OOM killer. lmkd watches memory pressure with PSI counters and kills unimportant background applications. When the user returns to that application, Android restarts it from its saved state.

15.13 The course's concepts in Linux

This case study found a concrete counterpart in Linux for each of the general concepts of the course (Table 15.18). The last row of the table sums up the direction Linux is taking today: while the mechanism stays in the kernel, the policy is increasingly left to the user.

Topic	Concept	Counterpart in Linux
Structures	System call, monolithic kernel	`syscall` , LKM, eBPF
Processes	PCB, creation, IPC	`task_struct` , fork/exec/wait, signals, pipes
Threads	Kernel-thread model	`clone()` , NPTL 1:1
Scheduling	Fair share, real time	EEVDF, FIFO/RR/DEADLINE, `sched_ext`
Synchronization	Lock, condition variable	futex, spinlock, RCU
Deadlock	Detection and prevention	lockdep, fcntl `EDEADLK`
Memory	Allocation, fragmentation	zones, buddy lists, SLUB
Virtual memory	Paging, replacement	4/5-level tables, COW, MGLRU, zswap, OOM
File systems	Inode, directory, journal	VFS, ext4, XFS, Btrfs
I/O	Driver, scheduling	device files, blk-mq, io_uring
Multiprocessor	Load balancing, NUMA	per-CPU queues, scheduling domains
Security	Access control, isolation	rwx, capabilities, namespaces, SELinux, seccomp
Design	Separation of mechanism and policy	cgroup, `sched_ext` : policy in user space

Table 15.18: The course's concepts and their counterparts in Linux.

Two mechanisms named in the table but not detailed in this chapter deserve a short explanation. The **futex** (fast user-space mutex) is the foundation of pthread locks. When the lock is free, acquiring and releasing it happen entirely in user space, with a single atomic instruction. The kernel is entered only when there is contention, that is, when a thread has to sleep. **RCU** (read-copy-update) is a synchronization method in which readers read the data structure without taking any lock. The writer prepares a new copy of the structure and swaps the pointer atomically. The old copy is not freed until

all readers that might still be reading it have finished. `lockdep` is the kernel's own deadlock checker: it tracks the order in which locks are taken and, the moment it sees a cycle in that order, it warns, before any deadlock has actually occurred.

15.14 Summary

- Linux is a **monolithic** kernel that implements the POSIX interface of UNIX with freshly written code. Combined with the GNU tools, it forms distributions. A release comes out about every 9–10 weeks; the last release of each year becomes an LTS.
- There are three interfaces between the layers: user, library, and system call. `printf` writes to a buffer; the real system call is `write`, which enters the kernel with the `syscall` instruction.
- The shell builds redirections from left to right with `dup2`, and pipelines with `pipe`, `fork`, `dup2`, and `exec`. If an unused write end is not closed, the reader cannot see EOF.
- **Loadable modules** give the monolithic kernel flexibility but run with kernel privilege. eBPF safely runs small verified programs in the kernel.
- On Linux a process and a thread are the same structure: `task_struct`. The difference is whether substructures such as `mm`, `files`, `fs`, and `sighand` are shared. The `clone()` flags select this; NPTL builds the 1:1 model. The PID the user sees is the `tgid`.
- Signals wait as a bit in the pending set and, on return to user mode, are delivered starting from the lowest number of the set ready = pending & ~blocked. Blocking is deferring; classic signals are not counted.
- The fair class has run with **EEVDF** since 6.6: among eligible tasks ($v \le V$), the one with the earliest virtual deadline is chosen. The share depends on weights (nice 0 → 1024); a short slice gets a task chosen more often.
- The address space consists of VMAs and is kept in a maple tree. On x86-64, a 48-bit address is translated by a four-level walk of 9+9+9+9+12 bits; LA57 adds a fifth level.
- Physical memory follows the node → zone → page hierarchy; the buddy allocator manages blocks of 2^0–2^{10} pages. **SLUB** keeps small objects in slabs; the free-list pointer lives inside the free object, and the fast path is lock-free.
- Free RAM goes to the page cache. Under pressure, kswapd, MGLRU, zswap/swap, direct reclaim, and, as a last resort, the OOM killer come into play.
- In device files the major number selects the driver and the minor number selects the unit. The block layer is multi-queue with blk-mq. Drivers attach to the VFS through the `file_operations` table.
- The four VFS objects are the superblock, inode, dentry, and file. ext4 uses block groups, extents, and `data=ordered` journaling. Data is freed when both the link count and the open-file count reach zero.

- In permission checks the first matching class wins; umask clears bits (request & ~umask). Capabilities split root into pieces. Container = **namespaces** (what is seen) + **cgroups** (how much is used) + seccomp and LSM.
- Android uses the same kernel with Bionic, ART, Binder, zygote, and a uid per application.

15.15 Key terms

monolithic kernel loadable kernel module system call shell pipe, pipeline file descriptor process control block (`task_struct`) thread group ID signal copy-on-write virtual runtime eligibility, virtual deadline bootloader virtual memory area (VMA) buddy system slab allocator page cache virtual file system (VFS) file locking journaling hard link / symbolic link capability namespace control group (cgroup) container

---

15.16 Test Yourself

1. On x86-64, 4 KiB pages and a four-level table are used. What are the four indices and the offset of the address 0x00007ffd4a3b2010? If the frame number in the PTE is 0x2f00a, what is the physical address?

   Answer

   PGD = address >> 39 = 255, PUD = 501, PMD = 81, PTE = 434, offset = 0x010. The physical address is 0x2f00a × 4096 + 0x010 = 0x2f00a010. This is a stack address: PGD 255 is the last entry of the user half.

2. Two continuously ready tasks, nice 0 (w = 1024) and nice 1 (w = 820), share a single CPU. How is the CPU shared in the long run? By how much does the `vruntime` of the nice 1 task grow when it runs for 4 ms?

   Answer

   The share is proportional to the weight: 1024 / 1844 ≈ 55.5% and 820 / 1844 ≈ 44.5%. One nice step shifts the share by about 10 points. Δv = 4 × 1024 / 820 ≈ 4.995 ms.

3. On a queue, V = 12 ms. There are tasks A (v = 10, d = 14), B (v = 12, d = 13), and C (v = 13, d = 12).

Which does EEVDF choose? Which would CFS have chosen?

Answer

The eligible tasks are those with v ≤ V: A and B. Although C has the earliest deadline, it is not eligible, because it has received more than its share. Between A and B, the one with the earlier deadline, B, is chosen. CFS would have chosen the smallest `vruntime`, that is, A.

4. A process has blocked `SIGUSR1` and installed a handler for `SIGINT`. While it sleeps, signals 17, 10, and 2 arrive in that order. On return to user mode, which signals are delivered, and in what order? Which one waits?

Answer

Pending {2, 10, 17}, blocked {10}; ready {2, 17}. The lowest number goes first: 2 (the handler runs), then 17 (`SIGCHLD`, default: ignore). 10 stays in the pending set and is delivered when the block is lifted. Arrival order plays no role in the selection.

5. On a fresh `kmalloc-512` slab (4 KiB, eight objects: +0, +512, …), the following operations are performed: a, b, c = alloc; free(b); free(a); d, e, f = alloc. Which addresses do d, e, and f get?

Answer

a = +0, b = +512, c = +1024; freelist +1536. free(b): +512 → +1536. free(a): +0 → +512 → +1536. The last one freed is handed out first: d = +0, e = +512, f = +1536.

6. How does a task created with `clone(CLONE_VM | CLONE_FILES | SIGCHLD, …)`, without `CLONE_THREAD`, appear in `ps`? If the child opens a new file, can the parent use it?

Answer

Because there is no `CLONE_THREAD`, the child has its own TGID; it appears in `ps` as a separate process and sends `SIGCHLD` to the parent when it ends. Even so, it shares the address space and the descriptor table with the parent. The number of the file the child opened is also valid in the parent; the parent can read with the same number.

7. With umask 077 and 002, what permissions do `touch a` and `mkdir d` produce?

Answer

umask 077: file 0600 (`rw-------`), directory 0700 (`rwx------`). umask 002: file 0664 (`rw-rw-r--`), directory 0775 (`rwxrwxr-x`). 002 is used in shared directories for group work.

8. The file `prog` is owned by ali, with group team and mode 0750. veli is in the team group; can is

not. Can veli execute `prog`? Can can read it? Can root execute a script whose mode is 0644?

> **Answer**
>
> veli falls into the group class: bit 5 (r-x), 5 & 1 = 1, granted. can falls into the others class: bit 0, denied. For root, the bits are not examined for read and write, but execution requires at least one x bit; 0644 has none, so it is denied (`EACCES`).

9. `echo x > f`, `ln f g`, and `ln -s f h` are run. A process opens `g` and keeps the descriptor. Then `rm f` and `rm g` are run. When are the data blocks freed? What does `cat h` print?

> **Answer**
>
> f and g are the same inode; the link count is 2. After the two `rm`s the link count is 0, but because an open descriptor exists, the blocks are freed only when the process closes the descriptor. h holds the path `"f"`; that name no longer exists. `cat h` gives the error “No such file or directory” (`ENOENT`).

10. A container's `cpu.max` file contains `"25000 100000"`. How much CPU can the container use? Does this limit affect the output of `ps` inside?

> **Answer**
>
> 25 ms in every 100 ms period, that is, a quarter of a CPU. The limit is a cgroup setting; it determines only how much is used. What `ps` shows is determined by the PID namespace, not by the cgroup.

15.17 Further reading

- R. Love, *Linux Kernel Development*, 3rd ed., Addison-Wesley, 2010: its chapters on process management, scheduling, memory, and the VFS show, in plain language, the kernel-code counterparts of the structures in this chapter, from `task_struct` to the slab allocator. It belongs to the CFS era.
- M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010: the most comprehensive programmer's reference for signals, process creation, pipes, file locking, and capabilities; the author's manual pages at man7.org are kept up to date.
- Linux kernel documentation, docs.kernel.org, the pages “EEVDF Scheduler” and “Control Group v2”: first-hand accounts of the concepts of EEVDF and of all the files of the cgroup v2 controllers.
- I. Stoica, H. Abdel-Wahab, “Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation”, Old Dominion University technical report, 1995: the theoretical foundation of the concepts of lag, eligibility, and virtual deadline.

15.18 References

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 10: Case Study 1: Unix, Linux and Android.
2. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 20: The Linux System.
3. R. Love, *Linux Kernel Development*, 3rd ed., Addison-Wesley, 2010.
4. M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010; man7.org manual pages (namespaces(7), capabilities(7), signal(7), fcntl(2), clone(2)).
5. The Linux Kernel Documentation, docs.kernel.org — scheduler (EEVDF, sched-ext), admin-guide/mm (MGLRU, zswap), block (blk-mq), filesystems (vfs, ext4), userspace-api (seccomp, landlock), admin-guide/cgroup-v2.
6. Android Open Source Project, source.android.com — architecture, Binder, application sandbox.
7. D. M. Ritchie, K. Thompson, “The UNIX Time-Sharing System”, *Communications of the ACM*, 17(7), 1974.
8. I. Stoica, H. Abdel-Wahab, “Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation”, technical report, Old Dominion University, 1995.

APPENDIX A

Glossary of Terms

The terms used in the book, each with a short definition. The number next to a term is the chapter where it is explained in detail. A searchable version of the glossary is on the course page.

A

abstraction 1
Hiding the complicated details of hardware and presenting concepts such as processes, files, and address spaces instead.

access control list (ACL) 13
A list that states who may access an object and with which rights.

access matrix 13
A protection model whose rows are domains and columns are objects, with the permitted rights stored in each cell.

address space 2
The set of memory addresses that a process can use.

address space layout randomization (ASLR) 13
Choosing the addresses of code, stack, and libraries at random on every run to make attacks harder.

aging 5
Raising the priority of a waiting process over time to prevent starvation; also a counter technique in page replacement.

Amdahl's law 4
The law stating that the serial part of a program limits the maximum speedup from parallelism.

application programming interface (API) 2
The library functions a programmer calls, which use system calls underneath.

atomic instruction 6
A hardware instruction that performs a read and a write without being interrupted (TSL, XCHG, CAS).

authentication 13
Verifying that a user is who they claim to be.

authorization 13
The step that decides which resources an authenticated user may access and with which rights.

B

backing store (swap space) 9
The disk area that holds pages not currently in memory.

banker's algorithm 7
An avoidance algorithm that knows maximum claims in advance and allows only safe states.

barrier 6
A synchronization tool that makes a group of processes or threads meet at a point; none may pass until all have arrived.

base and limit registers 8
Registers that hold the start and the size of a process's memory region.

batch system 1
An early type of system in which jobs run one after another without user interaction.

Belady's anomaly 9
The fact that with FIFO, increasing the number of frames can increase the number of page faults.

binding time 14
The moment when a name is bound to an object: compile time, load time, or run time.

bitmap 8
A structure that records free or used status with one bit per unit.

block / character device 11
Classes of devices that transfer data as addressable blocks or as a stream of bytes.

blocking / nonblocking / asynchronous I/O 11
I/O interfaces that make the caller wait, return immediately, or notify the caller when finished.

booting 2
The process that starts from firmware at power-on and loads the kernel into memory.

bootloader 15
A small program that runs after the firmware and loads the kernel (GRUB).

bounded waiting 6
The condition that a process waits only a bounded number of turns before entering its critical section.

Brooks's law 14
The observation that adding people to a late software project makes it later.

buddy system 9
An allocator that splits memory into power-of-two blocks and merges buddy blocks.

buffer / buffering 11
Holding transferred data temporarily in memory.

buffer cache 10
A cache that keeps disk blocks in memory.

buffer overflow 13
A write beyond the bounds of a buffer that corrupts adjacent memory, such as the return address.

bus 11
A shared communication line that connects components (PCIe, USB).

busy waiting 6
Waiting by repeatedly testing a condition in a loop; it consumes CPU time.

C

cache 8
A memory that holds frequently used data at a faster level.

cache coherence 12
Keeping copies of the same data in different caches consistent.

cache hit / miss 8
The wanted data is found in the cache (hit) or not found (miss).

cache line 8
The smallest block transferred between cache and memory (usually 64 bytes).

capability 13
An unforgeable token held by a process that grants access to an object with specific rights.

cascading termination 3
Terminating all children when a parent process terminates; in UNIX the children are instead adopted by a new parent.

certificate 13
A document that proves, with a trusted authority's signature, to whom a public key belongs.

certificate authority (CA) 13
An organization that vouches with its signature that the public key in a certificate belongs to its owner.

checkpoint 7
The moment when a process's state is saved to disk; after a deadlock the process can be rolled back to this point.

circular wait 7
Processes waiting for one another in a way that forms a cycle.

client–server model 2
A structure in which services run in server processes and exchange messages with clients.

clock algorithm 9
An implementation of second chance using a circular list and a hand (pointer).

clock cycle 8
One period of the processor clock; in a pipeline each stage takes one cycle.

Coffman conditions 7
Mutual exclusion, hold and wait, no preemption, and circular wait.

compaction 8
Shifting used regions together to merge the free space.

compare-and-swap (CAS) 6
An atomic instruction that writes a new value if the value at the address equals the expected one, and always returns the old value.

Completely Fair Scheduler (CFS) 15
Linux's scheduler that emulates an ideal multitasking processor by always choosing the task with the smallest virtual runtime.

compression (lossy / lossless) 9
Representing data with fewer bits, with or without losing information.

concurrency 3
Several activities making progress interleaved over time; they need not run at the same instant.

condition variable 6
An object on which a thread waits with wait until a condition holds and is woken with signal.

conditional compilation 14
Including or leaving out parts of the code at compile time according to preprocessor conditions.

confidentiality / integrity / availability (CIA) 13
The three basic goals of security.

container 12
A group of processes that share the same kernel but are isolated with namespaces and cgroups.

context switch 3
Saving the state and loading a new one in order to move the CPU from one process to another.

contiguous allocation 8
Giving each process (or file) a single, consecutive region.

control group (cgroup) 15
A Linux mechanism that limits and measures the CPU, memory, and I/O use of groups of processes.

convoy effect 5
Short jobs piling up and waiting behind a long one; it increases the average waiting time in FCFS scheduling.

cooperative multitasking 3
An arrangement in which the operating system can take back the processor only when the program yields it or makes a system call.

copy-on-write (COW) 9
Copying a shared page only when one of the sharers writes to it.

core dump 3
A file into which the memory image of a process killed by a signal is written, depending on system settings.

covert channel 13
A channel that leaks information along a path the security policy did not anticipate.

CPU burst 5
The time a process uses the CPU continuously between two I/O operations.

CPU scheduling 5
Deciding which of the ready processes gets the CPU.

CPU-bound / I/O-bound 3
A process that spends most of its time computing or waiting for I/O, respectively.

critical section (region) 6
The piece of code that accesses shared data and that only one process may be in at a time.

crossbar switch 12
An interconnect that joins every processor to every memory module with a separate crosspoint; it is conflict-free but expensive.

C-SCAN 11
A SCAN variant that serves in one direction only: on reaching the end, the head returns to the start of the disk without serving.

D

data / task parallelism 4
In data parallelism the data is split and each core performs the same operation on its part; in task parallelism each core runs a different task.

deadlock 7
A situation in which each process waits forever for a resource held by another process in the set.

deadlock avoidance 7
Granting each request only if it will not put the system into an unsafe state.

deadlock detection 7
Finding whether the system is deadlocked by using graphs or matrices.

deadlock prevention 7
Making one of the four conditions structurally impossible.

deadlock recovery 7
Resolving a deadlock by preempting resources, rolling back, or terminating processes.

defense in depth 13
The principle of using several independent layers of security so that the others still protect if one is breached.

defragmentation 10
Making the scattered blocks of files on a disk contiguous again.

demand paging 9
Bringing pages into memory only when they are first accessed.

denial of service 13
An attack that makes a system unable to serve legitimate users.

device controller 11
The hardware unit that drives a device directly and has registers and a buffer.

device driver 11
Software running in the kernel that talks to a particular controller.

digital signature 13
A digest signed with a private key and verified with the public key.

dining philosophers problem 6
A classic synchronization problem that illustrates deadlock and starvation with shared resources.

direct memory access (DMA) 11
A controller transferring data directly to memory without keeping the CPU busy.

directory 10
A special file that maps file names to files.

directory entry 10
A record in a directory that maps a name to an i-node number.

discretionary access control (DAC) 13
Access control in which the owner of an object decides who gets which permission.

disk scheduling 11
Choosing the order of pending disk requests (SSTF, SCAN, C-LOOK).

dispatch latency 5
The time it takes the dispatcher to stop one process and start another.

dispatcher 5
The module that gives the CPU to the selected process: context switching and mode change.

distributed shared memory 12
A method that makes the memories of separate machines appear as one shared address space at page granularity.

dual-mode operation 2
Hardware protection of the OS by separating kernel mode and user mode.

dynamic loading 8
Not loading a routine into memory until it is called; code that is never called is never loaded.

E

earliest deadline first (EDF) 5
A dynamic-priority algorithm that runs the task whose deadline is nearest.

Earliest Eligible Virtual Deadline First (EEVDF) 15
The selection rule of the fair class since Linux 6.6: among eligible tasks, run the one with the earliest virtual deadline.

eBPF (extended Berkeley Packet Filter) 14
A virtual machine in the Linux kernel in which small, verified programs run safely.

effective access time (EAT) 8
The average memory access time, taking TLB and page-fault probabilities into account.

eligibility 15
In EEVDF, a task having received less CPU than it deserves; only eligible tasks can be chosen.

encryption 13
Making data unreadable without a key.

entry / exit / remainder section 6
The code that asks permission before entering the critical section, the code that announces leaving it, and the rest of the code.

event loop 14
A loop that waits for events and dispatches each to its handler in order.

event-driven code 14
A program whose flow is determined by handlers attached to incoming events.

exception 11
An unusual condition that occurs while an instruction executes (division by zero, page fault).

executable file 1
A program file in a format the loader can load into memory and run (ELF, PE).

exokernel 2
A kernel that only allocates resources with protection and leaves abstraction to libraries.

exploit 13
Code or a method that exploits a vulnerability so that an attacker reaches a goal.

exponential average 5
A method that predicts the next CPU burst as an average of past bursts whose weights decrease.

exponential backoff 12
Waiting after a failed lock attempt and doubling the wait time with every failure.

extensible system 14
An operating system to whose kernel new functionality can be added at run time.

external name 14
A human-readable name given by a user to an object (for example a file path).

F

fair-share scheduling 5
Scheduling in which the processor share is given to users or groups rather than to processes.

false sharing 12
Unnecessary coherence traffic caused by different variables being in the same cache line.

fetch–decode–execute cycle 1
The basic working cycle that the CPU repeats for every instruction.

FIFO page replacement 9
A replacement algorithm that keeps pages in a queue in load order and evicts the page loaded earliest when space is needed.

file allocation table (FAT) 10
A structure in linked allocation that keeps block chains in a single table at the start of the disk; entry i is the number of the block that follows block i.

file attributes (metadata) 10
The information about a file other than its contents: name, size, owner, permissions, timestamps, and so on.

file descriptor 10
A small integer in the process table that represents an open file.

file locking 15
Locking a file or a region of it against simultaneous access.

file system 10
Software that organizes, names, and protects files and directories on a disk.

firewall 13
A defense component at the network layer that lets through only permitted connections.

first-come, first-served (FCFS) 5
A nonpreemptive algorithm that runs processes in order of arrival.

flash translation layer (FTL) 10
The layer in an SSD controller that redirects every write to a new, free page and maps the logical block address to that page.

forward secrecy 13
The property that old sessions cannot be decrypted even if a long-term key is later compromised.

frame (page frame) 8
A block of physical memory of the same size as a page.

futex (fast userspace mutex) 14
A fast lock that waits in user space when there is no contention and in the kernel when there is.

fuzzing 14
A technique that gives a program random or semi-random inputs to look for crashes and bugs.

G

gang scheduling 12
Running related threads together in the same time slice.

global / local replacement 9
In global replacement the victim is chosen among all frames; in local replacement a process chooses the victim only from its own frames.

graphical user interface (GUI) 1
An interface that provides visual interaction with windows, icons, menus, and a pointer.

guest operating system 2
The operating system that runs inside a virtual machine.

GUID partition table (GPT) 10
A partitioning format that is part of the UEFI standard, uses 64-bit LBAs, and identifies each partition with a GUID.

H

happens-before 4
An ordering relation that guarantees one thread's writes are visible to another thread; for example, join provides it.

hard / soft real-time 5
In a hard real-time system, missing a deadline is a failure; in a soft one, critical tasks have priority but there is no guarantee.

hard disk drive (HDD) 11
A secondary storage device that stores data on rotating magnetic platters.

hard link / symbolic link 10
A second name that points to the same i-node, or a special file that contains another path.

hash function (cryptographic) 13
A function that produces a fixed-size, irreversible digest from data.

hashed page table 8
A page table searched by the hash of the virtual page number.

hierarchical page table 8
Keeping the page table in a multilevel tree form by paging the page table itself.

hint 14
Cached information that need not be correct; if it is wrong it can be cheaply checked and corrected.

hit (cache) 8
The wanted data being found in the cache.

hold and wait 7
The condition that a process holding resources waits for new ones.

hole 8
A free block of memory left between processes in variable partitioning.

hybrid kernel 2
A structure that combines microkernel ideas with a monolithic kernel (Windows, macOS).

hypercall 12
In paravirtualization, a direct call that the guest operating system makes to the hypervisor instead of executing sensitive instructions.

hypercube 12
An interconnect in the shape of an n-dimensional cube; each of the 2^n nodes is linked to n neighbors.

hypervisor 12
The software layer that creates and runs virtual machines.

I

i-node 10
A structure that holds a file's attributes and block addresses.

indexed allocation 10
Keeping all block addresses of a file in one index block (the i-node).

indirection 14
Reaching an object not directly but through an intermediate name or table; it provides flexibility.

internal / external fragmentation 8
Unusable free space inside an allocated region or between regions.

internal fragmentation 9
Memory wasted in the unused inner part of an allocated block (for example, a page).

internal name 14
The name by which the system knows an object internally (for example, an i-node number).

interprocess communication (IPC) 3
The methods processes use to exchange data and coordinate.

interrupt 11
A device or event stopping the CPU's flow and switching it to a handler.

interrupt handler 11
The routine that runs in the kernel when an interrupt arrives.

interrupt vector 11
A table entry that gives the handler address for an interrupt number.

interrupt-driven I/O 11
An I/O method in which the device alerts the CPU with an interrupt when it is ready.

inverted page table 8
A single, system-wide table that keeps one entry per physical frame.

isolation 14
The property that a fault or data in one component does not affect the others.

J

journaling 10
Surviving crashes by first writing changes to a log and then applying them in place.

K

kernel 1
The central part of the operating system that runs in kernel mode and accesses hardware directly.

kernel mode 1
The privileged mode in which all instructions and hardware can be used.

kernel thread 14
A thread that runs only in kernel mode and that the kernel creates for its own work.

L

layered system 2
An OS structure in which each layer uses only the services of the layer below.

least recently used (LRU) 9
An algorithm that evicts the page that has gone unaccessed for the longest time.

lightweight process (LWP) 4
The old name for threads; they are called lightweight because creating one is much cheaper than creating a new process.

limited direct execution 3
The program runs directly on the processor at full speed, but asks the kernel for privileged work and gives the processor back by a timer.

linked allocation 10
Linking the blocks of a file together with pointers.

linker 2
The tool that combines object files and static libraries, resolves symbols, and produces an executable file.

livelock 7
A situation in which processes keep changing state but cannot make progress.

load balancing 5
Distributing work evenly among processors (push/pull migration).

loadable kernel module (LKM) 15
A piece of code that can be added to and removed from the kernel while it runs (drivers).

loader 2
The kernel component that reads the header of an executable file and maps its code and data segments into the process's address space.

locality 9
The tendency of programs to access a small address region within a short time.

lock-free 6
Structures written with atomic instructions such as CAS instead of locks, in which the system as a whole always makes progress.

logical / physical address 8
The address generated by the CPU and the address seen by the memory unit, respectively.

log-structured file system (LFS) 10
A file system design that treats the whole disk as a single log, buffering writes in memory and appending them to the disk in one long sequential write.

lottery scheduling 5
Proportional-share scheduling that gives processes tickets and distributes the CPU by a random drawing.

M

mailbox 3
In indirect message passing, an object with a unique identity into which messages are sent and from which they are received (a port).

main memory (RAM) 8
The memory that the CPU accesses directly and in which programs and data are held while running.

malware 13
Harmful software such as viruses, worms, Trojan horses, and ransomware.

man-in-the-middle attack 13
An attack in which an attacker gets between two parties and reads or changes their messages.

mandatory access control (MAC) 13
Access control that system policy enforces regardless of the owner's wishes (SELinux).

marshalling 12
In RPC, the stub packing a procedure's parameters into a message to send over the network.

master boot record (MBR) 10
The old partitioning format that holds boot loader code and four partition entries in the first sector of the disk; it cannot address beyond 2 TiB.

master-slave model 12
A multiprocessor arrangement in which the operating system runs on only one processor and the others run only user programs.

measured boot 13
Hashing each boot component before it runs and extending the digest into a PCR; the final value summarizes the whole chain.

memory barrier (fence) 6
An instruction that forces earlier memory operations to become visible before later ones.

memory consistency model 12
Rules that determine in which order writes to different addresses are seen by other cores (for example x86-TSO).

memory hierarchy 1
The layers of memory from fast, small, and expensive levels to slow, large, and cheap ones.

memory management 8
The tasks of allocating, protecting, and reclaiming memory for processes.

memory management unit (MMU) 8
The hardware that translates logical addresses into physical addresses at run time.

memory-mapped file 9
Mapping a file's pages into the address space so that it is read and written like memory (mmap).

memory-mapped I/O 11
Mapping device registers into the memory address space.

MESI protocol 12
A coherence protocol in which a cache line is in one of the states Modified, Exclusive, Shared, or Invalid.

message 3
The unit of data sent between processes or machines.

message passing 3
Processes exchanging messages through the kernel with send and receive.

message-signaled interrupts (MSI) 11
Delivering an interrupt not over a wire but as a memory write that contains the vector number (the extended form is MSI-X).

microkernel 2
A structure that leaves only IPC, scheduling, and address-space management in the kernel.

middleware 12
A software layer that gives applications a common model in a distributed system.

monitor 6
A construct that wraps shared data and the procedures that use it with automatic mutual exclusion.

monolithic kernel 2
A structure in which all OS services run in a single address space in kernel mode.

mount 10
Attaching a file system to a point in the directory tree.

multi-factor authentication 13
Authentication using at least two of something you know, something you have, and something you are.

multicomputer 12
A tightly coupled collection of computers in which each node has its own memory and the nodes communicate by messages.

multicore 1
Several independent processor cores on a single chip.

multilevel feedback queue (MLFQ) 5
Scheduling that moves processes between queues according to their behavior.

multilevel queue (MLQ) 5
Scheduling that divides processes into fixed classes with separate queues.

multiprogramming 1
Keeping several jobs in memory and running another while one waits for I/O.

multithreaded process 4
A process that runs several threads sharing the same address space.

mutex 6
A binary lock with an owner, used for mutual exclusion.

mutual exclusion 6
The condition that at most one process is in the critical section at a time.

N

namespace 15
A Linux feature that gives a group of processes a separate view of resources such as PIDs, network, and mounts.

network file system (NFS) 10
A protocol that gives access to files on a remote machine as if they were local.

no preemption 7
One of the four conditions for deadlock: a resource cannot be taken from a process that holds it unless the process releases it.

nonpreemptable resource 7
A resource that would be corrupted if taken back from its owner (for example a printer or a lock).

not recently used (NRU) 9
An algorithm that classifies pages into four classes by their R and M bits and, on a fault, evicts a page from the lowest nonempty class.

NUMA (non-uniform memory access) 12
A multiprocessor arrangement in which each processor accesses local memory faster than remote memory.

O

omega network 12
A staged, cheaper processor-memory interconnect built from a small number of 2×2 switches.

OOM killer (out-of-memory killer) 3
The Linux kernel mechanism that, when memory runs out, picks one of the processes using the most memory and terminates it with SIGKILL.

open-file table 10
A kernel table that holds the position and status information of open files.

operating system (OS) 1
The basic software that manages the hardware and provides abstractions to applications.

operating system services 2
The functions the OS provides to users and programs, such as program execution, I/O, files, and communication.

optimal page replacement (OPT) 9
The algorithm that evicts the page that will not be used for the longest time; it is an unimplementable yardstick.

orphan process 3
A process whose parent has terminated before it.

orthogonality 14
The ability to combine independent concepts freely and without side effects.

P

page 8
A fixed-size block of the logical address space.

page cache 10
A cache that keeps file data in memory in page units.

page fault 9
A trap that occurs when a page not in memory is accessed.

page locking (pinning) 9
Setting a lock bit on a frame to keep it out of the replacement candidates; for example, for a buffer in use by DMA.

page offset 8
The lower n bits (d) of a logical address, giving the byte position within the page.

page replacement 9
Choosing which page to remove from memory when there is no free frame.

page table 8
The table that records in which frame each page lies.

page table entry (PTE) 8
An entry that holds the frame number and the valid, protection, dirty, and reference bits.

page-table base register (PTBR) 8
The register that holds the starting address of the page table in main memory (CR3 on x86); only this register changes on a context switch.

paging 8
Dividing logical memory into pages and physical memory into frames and mapping between them.

paging daemon 9
A kernel thread that wakes periodically, frees pages in advance when free frames are few, and writes dirty pages to disk.

parallelism 3
Several tasks truly running at the same time on separate processors.

paravirtualization 12
The guest OS cooperating with the hypervisor through direct calls (hypercalls).

parent / child process 3
The process that creates a process with fork is the parent; the created process is the child.

parity 11
The XOR of the data; it is used to rebuild a lost block when a disk fails.

partition 10
A part of a disk that can hold a separate file system, swap space, or encrypted container; its list is kept in a table at the start of the disk.

path name (absolute / relative) 10
A name that points to a file starting from the root or from the working directory.

permission bits (rwx) 10
The read, write, and execute rights for the owner, the group, and others.

Peterson's algorithm 6
A solution that provides mutual exclusion for two processes using only reads and writes.

pipe 3
A one-way byte stream buffered in the kernel that connects two processes.

pipeline (CPU) 11
A technique that overlaps the stages of instructions so that one instruction finishes every cycle.

pipeline (shell) 15
A chain of commands in which the output of one is connected to the input of the next.

plaintext / ciphertext 13
In encryption, the readable message (P) and its form transformed with a key (C).

policy / mechanism 1
The mechanism decides how something is done; the policy decides what will be done.

polling 11
The CPU repeatedly reading a device's status register until it is ready.

port-mapped I/O 11
An arrangement in which device registers live in an address space separate from memory (I/O ports) and are accessed with special IN/OUT instructions.

position-independent code (PIC) 8
Code that uses no absolute addresses, jumps relative to the PC, and reaches global data through the GOT; shared libraries are compiled this way.

preemptable resource 7
A resource that can be taken back from its owner without harm (for example CPU or memory).

preemptive / nonpreemptive scheduling 5
A running process can be taken off the CPU by force (preemptive) or runs until it gives it up itself (nonpreemptive).

prepaging 9
Bringing in pages expected to be needed in advance, in a single bulk read, without waiting for faults.

primitive 14
A basic operation that the operating system provides and on which other operations are built.

principle of least privilege 13
Giving each component only the minimum rights it needs for its job.

priority ceiling 6
A solution to priority inversion: the task that takes a lock is immediately raised to the priority of the highest-priority task that may use that lock.

priority inheritance 6
The lock-holding process temporarily taking on the priority of the waiting process.

priority inversion 6
A high-priority process waiting for a lock held by a low-priority process.

priority scheduling 5
An algorithm that runs the ready process with the highest priority.

privilege escalation 13
An ordinary user or process gaining administrator rights by exploiting a flaw.

process 3
A program in execution; it has its own address space and resources.

process control block (PCB) 3
The record the kernel keeps about a process's state, registers, and resources.

process state 3
The process's position in its life cycle: new, ready, running, waiting, terminated.

processor affinity 5
The tendency to keep a process on the same processor so that its cache stays warm.

producer–consumer problem 6
The problem of coordinating a producer and a consumer that share a bounded buffer.

program counter (PC) 1
The register that holds the address of the next instruction.

program status word (PSW) 1
The register that holds the condition bits, the processor mode (kernel or user), and whether interrupts are enabled.

programmed I/O 11
I/O in which the CPU itself transfers data to the device byte by byte.

progress 6
A condition for critical-section solutions: if the region is free, only the processes that want to enter decide who enters, and the decision cannot be postponed indefinitely.

protection domain 13
The set of (object, right) pairs that a process can access.

public key infrastructure 13
The whole of authorities and rules that issue, distribute, and revoke certificates.

public-key cryptography 13
A method in which data encrypted with the public key can be decrypted only by the private key.

publish-subscribe 12
A communication model in which a sender publishes a message to a topic and those subscribed to the topic receive it.

push / pull migration 5
Load balancing on multiprocessors: in push migration work is pushed away from an overloaded core, in pull migration an idle core pulls work from a loaded one.

Q

quota 10
A limit on the number of blocks and files each user may use.

R

race condition 6
A faulty situation in which the result depends on the timing of accesses to shared data.

ransomware 13
Malicious software that encrypts data and demands a ransom.

rate-monotonic scheduling (RM) 5
A real-time algorithm that gives the task with the shorter period a higher fixed priority.

RAID 11
Arrangements that combine several disks for performance and reliability.

readers–writers problem 6
An access problem that allows many readers together but a writer alone.

ready queue 3
The queue of ready processes waiting for the CPU.

real-time operating system (RTOS) 1
An operating system that guarantees a result is produced within a given time.

reentrancy 14
The ability of code to be called again while it is itself executing.

reentrant / thread-safe 4
Code that can be executed safely by several calls or threads at the same time.

reference bit / dirty bit 9
Bits that show that a page has been accessed (R) and that it has been modified (M).

reference monitor 13
The component that checks every access request and allows or denies it according to the rules.

reference string 9
The sequence of page accesses used to compare algorithms.

register 1
A very fast, small storage unit inside the processor.

regression 14
A previously working feature breaking after a new change.

relocation 8
Shifting addresses so that a program can be loaded at a different place in memory.

relocation register 8
A register whose value is added to the logical address to give the physical address; it is the same as Tanenbaum's base register.

remote direct memory access (RDMA) 12
Network access in which the NIC writes directly into a process's memory on another machine without involving the kernel.

remote procedure call (RPC) 12
Calling a procedure on another machine as if it were a local call.

resource allocation graph 7
A directed graph that shows requests and allocations with process and resource nodes.

response time 5
The time from a request until the first response (the first run) begins.

return-oriented programming (ROP) 13
An attack technique that builds attack code by arranging on the stack the addresses of small code fragments (gadgets) ending in ret within the program and its libraries.

round robin (RR) 5
A preemptive algorithm that runs the ready processes in turn, one time slice each.

runqueue 3
In Linux, a ready-process queue assigned separately to each processor core; cores do not wait for one another when choosing.

S

safe / unsafe state 7
The situation in which an order exists in which all processes can finish, or does not exist.

salt 13
A random value added to a password before it is hashed.

sandbox 13
Running a program in an isolated environment with limited rights.

SCAN (elevator algorithm) 11
An algorithm in which the head moves to the end of the disk in one direction serving requests on the way, and reverses direction on reaching the end.

scatter-gather 11
In DMA, giving the device a list of (address, length) pairs; the device writes to or reads from noncontiguous pages in one operation.

scheduler 5
The kernel component that makes the scheduling decision.

second chance 9
A FIFO variant that gives a page whose reference bit is 1 one more chance.

secure boot 13
Verifying the signature of each piece of software in the boot chain before it is loaded.

seek time / rotational latency 11
The time for the arm to move to the track and the time for the sector to come under the head.

segmentation 8
Dividing the address space into variable-size logical segments such as code, data, and stack.

semaphore 6
A synchronization object with an integer counter, used with down/up (P/V) operations.

sequential consistency (SC) 6
The property that the memory operations of all threads can be arranged in a single common order that preserves each thread's program order.

shadow memory 14
A separate memory region that records whether each byte of memory is accessible; KASAN checks it.

shadow page table 12
The hypervisor's table for each guest process that maps guest virtual addresses directly to host physical addresses.

shared library 8
A library loaded into memory once and used by many processes (.so, .dll).

shared memory 3
A common memory region mapped into the address spaces of several processes.

shell 1
The command interpreter through which a user interacts with the operating system by typing commands.

shortest job first (SJF) 5
An algorithm that chooses the process whose next CPU burst is shortest.

shortest remaining time first (SRTF) 5
The preemptive form of SJF.

shortest seek time first (SSTF) 11
A disk arm scheduling algorithm that at each step chooses the pending request nearest to the head.

side channel 13
Leaking secret information through an indirect path such as timing, power, or cache behavior.

signal 15
An asynchronous software notification that tells a process about an event (SIGKILL, SIGINT).

simultaneous multithreading (SMT) 12
A core executing two hardware threads at the same time.

slab allocator 9
An allocator that provides pre-prepared caches for frequently used kernel objects.

snooping (cache coherence) 12
A coherence method in which each cache watches writes on the shared bus and updates or invalidates its own copy.

socket 3
A communication endpoint between two ends, over the network or locally.

solid-state drive (SSD) 11
A storage device based on NAND flash with no moving parts.

space sharing 12
Scheduling in which related threads are placed on separate processors at the same time.

space–time trade-off 14
Gaining time by spending memory, or the reverse.

speedup 4
The ratio of the old running time to the new (parallel) running time.

spinlock 6
A lock that waits in a loop until the lock is released; for short critical sections.

spooling 1
Queuing jobs for a slow device on disk and sending them in order (for example, printing).

spurious wakeup 6
A thread waiting on a condition variable waking up without any signal; for this reason the condition is checked with a while loop.

stable storage 11
A storage arrangement that guarantees a write either completes correctly or leaves the old data in place.

stack canary 13
A random value placed before the return address; if it has changed, an overflow is detected and the program is stopped.

stack pointer (SP) 1
The register that points to the top of the stack holding local variables and return addresses in function calls.

starvation 5
A process constantly losing out to others and never making progress.

striping / mirroring 11
In RAID, striping spreads consecutive blocks across different disks to gain parallelism; mirroring keeps a copy of each block on another disk.

stub 14
Temporary code that stands in for a component not yet written and imitates only its interface.

superblock 10
The block that holds a file system's basic information such as its size and number of blocks.

swapping 8
Moving a process or pages between memory and disk.

symmetric multiprocessor (SMP) 12
A model in which a single copy of the OS runs on all processors with equal rights.

synchronization 6
Regulating the access of concurrent processes to shared data.

synchronous / asynchronous communication 14
Communication in which the sender continues after waiting for the reply or without waiting for it.

system call 2
The interface a program uses to request a service from the kernel.

T

test and set lock (TSL) 12
An indivisible instruction that reads a memory word, puts its old value in a register, and writes 1 to the word; on a multiprocessor it locks the bus.

thrashing 9
The system spending most of its time moving pages.

thread 4
An independent flow of execution within a process; it has its own PC, registers, and stack.

thread control block (TCB) 4
A structure that holds a thread's identity, state, saved PC and registers, stack pointer, and priority.

thread group ID (TGID) 15
In Linux, the identifier shared by all threads of a process; it is the PID that users see.

thread pool 4
Threads created in advance that take tasks from a queue.

thread-local storage (TLS) 4
Variables of which each thread keeps its own copy.

threat 13
A possible event or attacker that could violate one of a system's security goals.

throughput 5
The number of processes completed per unit of time.

ticket lock 12
A fair spinlock that gives waiters ticket numbers and hands out the lock in that order.

time of check to time of use (TOCTOU) 13
A race-condition vulnerability in which an attacker can change a resource between the moment it is checked and the moment it is used.

time sharing 12
Sharing the CPU among users in short slices to provide interactive use.

time slice (quantum) 5
The longest CPU time a process may use before being preempted.

timer 2
A hardware counter that generates an interrupt after a set time; it returns the CPU to the OS.

TLB reach 8
The amount of memory the TLB covers: number of entries × page size.

TLB shootdown 9
Clearing stale entries from the other cores' TLBs when a page mapping changes.

top half / bottom half 11
Splitting interrupt handling in two: the main handler kept short (top half) and the deferred work that runs with interrupts enabled (bottom half).

trade-off 5
Giving up one property to improve another.

translation lookaside buffer (TLB) 8
A small, fast cache that holds recently used page-to-frame mappings.

trap 2
The program entering the kernel by its own request (a system call) or by an error; a software interrupt.

Trojan horse 13
Software that the user installs willingly, which looks useful but secretly does harmful work.

trusted computing base 13
All the hardware and software responsible for enforcing the security rules and which must be trusted to work correctly.

Trusted Platform Module (TPM) 13
A small security chip built into the motherboard or processor that contains platform configuration registers (PCRs).

tuple space 12
A shared associative memory in which processes drop tuples and retrieve them by matching (Linda).

turnaround time 5
The time from a process's arrival to its completion.

two-phase locking (2PL) 7
A protocol in which all locks are first acquired and then released; once a lock has been released, no new one is acquired.

U

UMA (uniform memory access) 12
A multiprocessor arrangement in which every processor reaches every memory address in the same time.

UMA / NUMA 12
An architecture in which memory access time is the same for all processors, or varies with proximity.

unikernel 14
A structure in which the application and the kernel functions it needs are compiled into a single address space as a single image.

use-after-free 13
A vulnerability that arises when a program keeps accessing a block it has freed through an old pointer.

user mode 1
The mode in which privileged instructions are forbidden and applications run.

user-level / kernel-level thread 4
Threads managed by a library or by the kernel, respectively.

V

valid–invalid bit 8
A bit that shows whether a page is in the address space and in memory.

virtual address 9
The address a process sees, which the MMU translates into a physical address.

virtual deadline 15
In EEVDF, the deadline, computed from the request size, used to choose among the eligible tasks.

virtual file system (VFS) 10
A kernel layer that unifies different file systems behind a common interface.

virtual machine (VM) 2
A machine created by software that behaves like a real computer.

virtual memory 9
A technique that gives processes a contiguous address space larger than physical memory.

virtual memory area (VMA) 15
In Linux, a contiguous region of a process's address space that shares the same permissions and backing.

virtual runtime 15
In CFS and EEVDF, the accumulated CPU time of a task scaled by its priority.

virtualization 1
Presenting a single physical resource as several virtual resources.

vulnerability 13
A design or implementation flaw that an attack can exploit.

W

wait queue 3
A queue that holds the processes waiting for a particular event (such as I/O completing or a child terminating).

wait-for graph 7
A resource allocation graph with the resource nodes removed; a cycle means deadlock.

waiting time 5
The total time a process spends in the ready queue.

wear leveling 11
Extending lifetime by spreading writes evenly among flash blocks.

work stealing 4
An idle thread taking tasks from another thread's queue.

working set 9
The set of pages a process has used in its last Δ accesses.

worm 13
A program that copies itself from machine to machine, independent of the user, by exploiting an unpatched service on the network.

write-invalidate / write-update 12
Cache coherence policies: before a write the other copies are invalidated, or the new value is sent to all copies.

write-through / write-back 10
Write-through sends every change to disk immediately; write-back accumulates dirty blocks in memory and writes them later.

Z

zero trust 13
An approach that does not trust even the inside of the firewall; requests from inside the network also go through authentication and authorization checks.

zero-fill-on-demand 9
Serving the first access to BSS and new heap pages with a zeroed frame; zeroing is required for security.

zombie process 3
A process that has finished but whose exit status has not been collected by its parent.

I

I/O scheduler 11
The kernel layer that orders and merges block requests (mq-deadline, bfq).

APPENDIX B

Prerequisites for Operating Systems

OPERATING SYSTEMS · PREREQUISITES

Operating systems bring together software, hardware, algorithms and low-level programming. A solid grounding in the areas below makes it much easier to understand the behavior of the kernel, the execution of processes, memory management, file systems and interaction with devices.

Essential preparation

The most useful preparation is hands-on experience with programming, data structures, computer architecture, memory, command-line tools and basic networking.

FOUNDATIONS · PROGRAMMING · HARDWARE

Core Background

These topics form the technical foundation for processes, scheduling, synchronization, virtual memory, file systems, I/O, networking and protection.

Algorithms

Problem solving · efficiency · scheduling

An operating system continuously makes decisions about CPU time, memory, I/O (input/output) and resource allocation.

WHAT TO KNOW

Data Structures

Representation · lookup · resource tracking

The kernel keeps track of processes, memory regions, open files, devices, timers and network state with data structures.

WHAT TO KNOW

Complexity, trade-offs in scheduling, search, sorting, recursion and tracing the state of a program are useful foundations.

Complexity **Scheduling** **Search**

In short: Complexity, trade-offs in scheduling, search, sorting, recursion and tracing the state of a program are useful foundations.

You should be comfortable with arrays, linked lists, stacks, queues, trees, hash tables and bitmaps.

Lists **Queues** **Trees** **Hash tables**

In short: You should be comfortable with arrays, linked lists, stacks, queues, trees, hash tables and bitmaps.

Mathematical Foundations

Number systems · graphs · probability

The size of a page table, splitting an address into a page number and an offset, deadlock graphs, the matrices of the banker's algorithm, effective access time and key exchange all rest on basic discrete mathematics and probability.

WHAT TO KNOW

Powers of two and the units KiB/MiB/GiB, binary and hexadecimal notation, logarithms, modular arithmetic, sets, matrices, directed graphs and cycles, weighted averages and simple probability should be familiar.

Powers of two **Hexadecimal**

Graphs **Probability**

In short: Powers of two and the units KiB/MiB/GiB, binary and hexadecimal notation, logarithms, modular arithmetic,

Computer Architecture

CPU · registers · privilege · memory

An operating system runs directly on the hardware. It relies on the mechanisms the architecture provides for protection and control.

WHAT TO KNOW

You should know registers, instruction execution, privilege levels, interrupts, exceptions, caches and address translation.

Registers **Interrupts** **MMU**

Privilege

In short: You should know registers, instruction execution, privilege levels, interrupts, exceptions, caches and address translation.

sets, matrices, directed graphs and cycles, weighted averages and simple probability should be familiar.

Programming in C

Pointers · memory · systems programming

Most classic kernel interfaces and many operating system implementations are closely tied to C-style systems programming.

WHAT TO KNOW

Pointers, arrays, structs, bit operations, manual memory management and function calls should be well understood.

Pointers **Structs** **Memory** **Bit operations**

In short: Pointers, arrays, structs, bit operations, manual memory management and function calls should be well understood.

Compilation and Reading Assembly

Compiler · linker · machine instructions

The course examples compile C code with gcc, link it with libraries and show system calls, traps and atomic operations with a few lines of assembly.

WHAT TO KNOW

You should know at a basic level the path from a source file to an object file and an executable, static and dynamic libraries, how compiler optimizations can change the code, and how a short assembly listing corresponds to the C code.

gcc **Linking** **Executables** **Assembly**

In short: You should know at a basic level the path from a source file to an object file and an executable, static and dynamic libraries, how compiler optimizations can change the code, and how a short assembly listing corresponds to the C code.

Process and Thread Basics

Execution · concurrency · state

Before studying scheduling and synchronization, it helps to know how programs are executed and how threads share the resources of a process.

Memory Fundamentals

Addresses · stack · heap · virtual memory

An operating system manages physical memory and presents each process with a controlled address space.

WHAT TO KNOW

You should know at a basic level the difference between a program, a process and a thread, and what the stack, the heap and the execution state are.

Processes **Threads** **Stack** **Heap**

In short: You should know at a basic level the difference between a program, a process and a thread, and what the stack, the heap and the execution state are.

WHAT TO KNOW

Addresses, bytes, the difference between the stack and the heap, locality and the idea that virtual addresses must be translated should be familiar.

Addresses **Stack** **Heap** **Locality**

In short: Addresses, bytes, the difference between the stack and the heap, locality and the idea that virtual addresses must be translated should be familiar.

Digital Logic and Hardware

Bits · devices · buses · interrupts

An operating system talks to processors, memory and devices through interfaces defined by the hardware.

WHAT TO KNOW

Basic knowledge of binary numbers, buses, device registers, interrupts and direct memory access (DMA) makes low-level I/O much easier to understand.

Binary numbers **Devices** **Interrupts** **DMA**

In short: Basic knowledge of binary numbers, buses, device registers, interrupts and direct memory access (DMA) makes low-level I/O much easier to understand.

Command Line and Linux Tools

Shell · files · processes · diagnostics

Many operating system concepts become concrete when explored on the command line.

WHAT TO KNOW

It is useful to practice with commands such as ps, top, chmod, df and dmesg for files, permissions, processes, memory, disks and logs.

Shell **Processes** **Permissions** **Logs**

In short: It is useful to practice with commands such as ps, top, chmod, df and dmesg for files, permissions, processes, memory, disks and logs.

Networking Basics

Sockets · TCP/IP · ports

Security Fundamentals

Protection · authentication · permissions

Modern operating systems include a network stack and offer communication through socket APIs.

WHAT TO KNOW

You should know IP addresses, ports, the difference between TCP and UDP, client/server behavior and basic socket concepts.

TCP/IP **Sockets** **Ports**

In short: You should know IP addresses, ports, the difference between TCP and UDP, client/server behavior and basic socket concepts.

An operating system protects the boundaries between users, processes, files and devices.

WHAT TO KNOW

You should know the ideas behind authentication, authorization, permissions, the principle of least privilege and basic threat models.

Permissions **Authentication**

Protection

In short: You should know the ideas behind authentication, authorization, permissions, the principle of least privilege and basic threat models.

Most important before the course

Programming in C, pointers, data structures, processes, memory and basic computer architecture.

Can be learned alongside the course

Shell tools, inspecting processes on Linux, networking commands, permissions and system diagnostics.

No kernel experience needed

System calls, scheduling, synchronization, virtual memory, file systems and I/O are covered from the ground up in the course.

Readiness Checklist

If most of the statements below sound familiar, you are well prepared.

✓ I can read and write basic C programs, and compile and run them with gcc.

✓ I understand pointers, arrays, structs, the stack and the heap.

✓ I know queues, linked lists, trees and hash tables.

✓ I can calculate with binary and hexadecimal numbers, powers of two and simple probability.

✓ I know what CPU registers and interrupts are.

✓ I can use the terminal and basic Linux commands.

✓ I understand files, processes, permissions and basic networking concepts.

APPENDIX C

Pioneers of Operating Systems

OPERATING SYSTEMS · A HISTORICAL VIEW

Operating systems are the product of decades of accumulated work. Research on time sharing, concurrency, memory management, interactive use, portability, networking, virtualization, security and open systems shaped the structure we have today.

From the first computers to modern operating systems

The people, systems and ideas that influenced the development of operating systems and of modern computer science.

PEOPLE · SYSTEMS · ENGINEERING

Important People in Operating Systems

People who made major contributions to operating system architecture, concurrency, memory management, file systems, virtualization, security, networking and systems software.

John von Neumann

Stored-program model · 1940s

Helped formalize the stored-program model, in which instructions and data reside in the same memory.

John McCarthy

The idea of time sharing · 1950s–1960s

Argued that many users should be able to use one powerful computer at the same time through time sharing. He set

WHY IT MATTERS FOR ENGINEERING

Architecture, memory, execution, privileged control and program loading all rest on this model.

Architecture **Memory** **Execution**

Why it matters: Architecture, memory, execution, privileged control and program loading all rest on this model.

1940s

Fernando Corbató

CTSS and Multics · 1960s

Led influential time-sharing projects at MIT, including CTSS and Multics. He received the Turing Award in 1990 for this work.

WHY IT MATTERS FOR ENGINEERING

Multics advanced protection rings, the hierarchical file system, dynamic linking and multi-user computing.

CTSS **Multics** **Protection**

Why it matters: Multics advanced protection rings, the hierarchical file system, dynamic linking and multi-user computing.

1960s

out this proposal in a 1959 memo at MIT.

WHY IT MATTERS FOR ENGINEERING

Time sharing created the need for scheduling, timer interrupts, context switching and multi-user protection.

Time sharing **Scheduling**

Multi-user

Why it matters: Time sharing created the need for scheduling, timer interrupts, context switching and multi-user protection.

1950s–60s

Edsger W. Dijkstra

Synchronization and operating system design · 1960s

Introduced the semaphore and designed the layered THE system. He laid the foundations of synchronization and concurrent programming.

WHY IT MATTERS FOR ENGINEERING

The critical section, mutual exclusion, deadlock and starvation remain core problems of operating systems today.

Semaphore **Synchronization**

Deadlock

Why it matters: The critical section, mutual exclusion, deadlock and starvation remain core problems of operating systems today.

Peter J. Denning

Virtual memory and the working set · 1960s

Developed the working-set model to explain locality and the performance of virtual memory.

WHY IT MATTERS FOR ENGINEERING

The model explains page faults, memory pressure and thrashing.

Virtual memory **Working set** **Thrashing**

Why it matters: The model explains page faults, memory pressure and thrashing.

1960s

Dennis Ritchie

C and UNIX · Bell Labs · 1970s

Created the C language and co-developed UNIX with Ken Thompson, which made portable systems software possible.

WHY IT MATTERS FOR ENGINEERING

Rewriting most of UNIX in C in 1973 showed that an operating system could be moved to different hardware.

C **UNIX** **Portability**

Why it matters: Rewriting most of UNIX in C in 1973 showed that an operating

1960s

Ken Thompson

UNIX · Bell Labs · from 1969

Co-developed UNIX with Dennis Ritchie. He established a compact design philosophy built on simple tools that can be combined with one another.

WHY IT MATTERS FOR ENGINEERING

UNIX shaped the concepts of the process, the file descriptor, the pipe, the shell and the hierarchical file system.

UNIX **Processes** **File systems**

Why it matters: UNIX shaped the concepts of the process, the file descriptor, the pipe, the shell and the hierarchical file system.

1969+

Per Brinch Hansen

Concurrent programming · 1970s

Developed influential abstractions for concurrent programming and operating system structure. The RC 4000 kernel and the Concurrent Pascal language came out of this work.

WHY IT MATTERS FOR ENGINEERING

Synchronization based on the monitor made access to shared data more disciplined.

Monitors **Concurrency** **System structure**

system could be moved to different hardware.

1970s

C. A. R. Hoare

Monitors and CSP · 1970s

Contributed monitors and the Communicating Sequential Processes (CSP) model to the study of concurrency.

WHY IT MATTERS FOR ENGINEERING

This work offers explicit models for synchronization, channels and message passing.

CSP **Monitors** **IPC**

Why it matters: This work offers explicit models for synchronization, channels and message passing.

1970s

Richard Stallman

The GNU Project · from 1983

Started the GNU Project to build a UNIX-compatible free software system.

WHY IT MATTERS FOR ENGINEERING

GNU compilers, libraries, shells and tools became central to UNIX-like systems.

GNU **Free software** **Toolchain**

Why it matters: Synchronization based on the monitor made access to shared data more disciplined.

1970s

Butler Lampson

Systems design · Xerox PARC · 1970s

Contributed to influential personal computer projects such as the Alto and to work on distributed systems.

WHY IT MATTERS FOR ENGINEERING

His systems work brings together protection, naming, networking and the design of reliable interfaces.

Protection **Distributed systems**

Alto

Why it matters: His systems work brings together protection, naming, networking and the design of reliable interfaces.

Andy Hertzfeld

1970s+ **Macintosh system software · 1980s**

Worked on the system software and the Macintosh Toolbox of the original Macintosh.

WHY IT MATTERS FOR ENGINEERING

The original Macintosh software shaped expectations of the desktop graphical user interface (GUI) and of tightly integrated system services.

Why it matters: GNU compilers, libraries, shells and tools became central to UNIX-like systems.

1983+

Macintosh **GUI** **System software**

Why it matters: The original Macintosh software shaped expectations of the desktop graphical user interface (GUI) and of tightly integrated system services.

1980s

Andrew S. Tanenbaum

MINIX and operating system education · from the 1980s

Developed MINIX and wrote influential operating systems textbooks.

WHY IT MATTERS FOR ENGINEERING

MINIX makes the microkernel, interprocess communication (IPC), processes and the file system open to study in a complete working system.

MINIX **Microkernel** **Education**

Why it matters: MINIX makes the microkernel, interprocess communication (IPC), processes and the file system open to study in a complete working system.

1987+

Ken Sakamura

TRON and embedded systems · from the 1980s

Started the TRON project in 1984, which became influential in embedded and real-time systems.

WHY IT MATTERS FOR ENGINEERING

David Cutler

VMS and Windows NT · 1970s–1990s

Led major operating system projects at DEC, such as RSX-11M and VMS, and then Windows NT at Microsoft.

WHY IT MATTERS FOR ENGINEERING

Windows NT built strong abstractions for protected memory, security, asynchronous I/O (input/output) and portability.

VMS **Windows NT**

Kernel architecture

Why it matters: Windows NT built strong abstractions for protected memory, security, asynchronous I/O (input/output) and portability.

Linus Torvalds

The Linux kernel · from 1991

1970s–90s

Created the Linux kernel. Today it runs on servers, cloud systems, Android, embedded devices and supercomputers.

WHY IT MATTERS FOR ENGINEERING

ITRON emphasizes deterministic scheduling, interrupt latency and deadline-driven design.

TRON **RTOS** **Embedded**

Why it matters: ITRON emphasizes deterministic scheduling, interrupt latency and deadline-driven design.

1980s+

Linux shows how modular subsystems can be built inside a large monolithic kernel.

Linux **Kernel** **Open development**

Why it matters: Linux shows how modular subsystems can be built inside a large monolithic kernel.

1991+

J. C. R. Licklider

A vision of interactive computing · 1960s

Argued for interactive, networked use of computers, beyond stand-alone batch machines.

WHY IT MATTERS FOR ENGINEERING

Interactive use made low latency, time sharing and remote access important.

Interactive use **Time sharing**

Networking

Why it matters: Interactive use made low latency, time sharing and remote access important.

Douglas Engelbart

Interactive systems · 1960s

Led the NLS project. At the famous 1968 demonstration he introduced interactive use, hypertext, collaboration tools and the mouse.

WHY IT MATTERS FOR ENGINEERING

This work increased the need for event-driven I/O, displays and graphical environments that respond quickly.

Interaction **Input devices**

History of the GUI

Why it matters: This work increased the need for event-driven I/O, displays and graphical environments that respond quickly.

1960s

Douglas McIlroy

UNIX pipes and software tools · 1970s

Proposed UNIX pipes, which were added to UNIX in 1973. McIlroy also deeply influenced the “software tools” approach.

WHY IT MATTERS FOR ENGINEERING

Brian Kernighan

UNIX tools and systems programming · from the 1970s

1960s

Contributed to the UNIX software environment. With others he wrote influential tools such as AWK and

Pipes let independent programs be connected through their standard input and output streams.

Pipes **UNIX** **Software tools**

Why it matters: Pipes let independent programs be connected through their standard input and output streams.

1970s

programming books.

WHY IT MATTERS FOR ENGINEERING

This work shaped combining tools on the command line, scripting and portable systems programming.

UNIX **AWK** **Tools**

Why it matters: This work shaped combining tools on the command line, scripting and portable systems programming.

1970s+

John Lions

Teaching with UNIX source code · 1970s

Prepared an influential study that explains the source code of the UNIX Version 6 kernel line by line: *Lions' Commentary on UNIX 6th Edition*.

WHY IT MATTERS FOR ENGINEERING

Through this work students got to see process tables, traps, scheduling, i-nodes and system call paths.

UNIX V6 **Kernel internals**

Education

Why it matters: Through this work students got to see process tables, traps, scheduling, i-nodes and system call paths.

1970s

Bill Joy

BSD UNIX · 1970s–1980s

One of the principal developers of BSD UNIX. He wrote the vi editor and the csh shell.

WHY IT MATTERS FOR ENGINEERING

BSD had a major influence on networking, workstations and later UNIX derivatives.

BSD **UNIX** **Networking**

Why it matters: BSD had a major influence on networking, workstations and later UNIX derivatives.

1970s–80s

Marshall Kirk McKusick

BSD and the Fast File System · from the 1980s

Gerald J. Popek

Virtualization theory · 1970s

One of the leading BSD developers. He is best known for the Fast File System (FFS) and for his work on FreeBSD.

WHY IT MATTERS FOR ENGINEERING

FFS improved locality, block allocation and disk performance in the file system.

FFS **BSD** **File systems**

Why it matters: FFS improved locality, block allocation and disk performance in the file system.

1980s+

Together with Robert P. Goldberg, he set out in 1974 the formal criteria that determine when an architecture can support virtual machines efficiently.

WHY IT MATTERS FOR ENGINEERING

The Popek–Goldberg criteria became the foundation of hypervisors and of virtualization based on trap-and-emulate.

Virtualization **Hypervisors**

Isolation

Why it matters: The Popek–Goldberg criteria became the foundation of hypervisors and of virtualization based on trap-and-emulate.

1970s

Robert P. Goldberg

Virtual machine research · 1970s

Laid the foundations for the formal study of virtual machine monitors (VMM).

WHY IT MATTERS FOR ENGINEERING

This work identified equivalence, resource control and efficiency as the essential properties of a VMM.

VMM **Virtual machines**

Resource control

Why it matters: This work identified equivalence, resource control and efficiency as the essential properties of a VMM.

Leslie Lamport

Distributed systems · from the 1970s

Developed the basic methods for reasoning about the order of events, clocks and correctness in distributed systems.

WHY IT MATTERS FOR ENGINEERING

Logical clocks and the happens-before relation form the basis of distributed coordination and replication.

Logical clocks **Distributed systems**

Ordering

Why it matters: Logical clocks and the happens-before relation form the basis

1970s

of distributed coordination and replication.

1970s+

Barbara Liskov

Abstraction and distributed systems · from the 1970s

Pioneered abstraction in programming languages and the design of fault-tolerant distributed systems.

WHY IT MATTERS FOR ENGINEERING

This work strengthened modular interfaces, atomic transactions and reliable service design.

Abstraction **Distributed systems**

Fault tolerance

Why it matters: This work strengthened modular interfaces, atomic transactions and reliable service design.

Tom Kilburn

Atlas and virtual memory · 1950s–1960s

Led the computer projects at Manchester. One of them, Atlas (1962), is a milestone in the history of virtual memory.

WHY IT MATTERS FOR ENGINEERING

Atlas showed that paging, automatic movement of data between memory and storage, and address translation work in practice.

Atlas **Paging** **Virtual memory**

Why it matters: Atlas showed that paging, automatic movement of data between memory and storage, and address translation work in practice.

1970s

Maurice Wilkes

EDSAC and microprogramming · 1940s–1950s

Led the team that built EDSAC (1949). He proposed microprogramming for processor control in 1951.

WHY IT MATTERS FOR ENGINEERING

This work shaped the hardware–software interface and the practice of reusable systems software.

EDSAC **Microprogramming**

1950s–60s

David Wheeler

Subroutines and early system software · 1940s–1950s

Made important contributions to reusable subroutines and early programming methods on EDSAC.

WHY IT MATTERS FOR ENGINEERING

Subroutine calls eventually grew into calling conventions, stacks, libraries and runtime support.

Subroutines **EDSAC**

System software

System software

Why it matters: This work shaped the hardware–software interface and the practice of reusable systems software.

1940s–50s

Why it matters: Subroutine calls eventually grew into calling conventions, stacks, libraries and runtime support.

1940s–50s

Niklaus Wirth

The Oberon system · from the 1980s

Designed influential programming languages such as Pascal, Modula-2 and Oberon. With Jürg Gutknecht he developed the small and compact Oberon operating system.

WHY IT MATTERS FOR ENGINEERING

Oberon showed that a language and an operating system can be designed together, and demonstrated simplicity and the building of a consistent interactive system.

Oberon **System design** **Languages**

Why it matters: Oberon showed that a language and an operating system can be designed together, and demonstrated simplicity and the building of a consistent interactive system.

1980s+

Auguste Kerckhoffs

Kerckhoffs's principle · 1883

In his 1883 paper *La cryptographie militaire* he listed six rules for military ciphers. The best known says that a system must remain secure even if it falls

Robert Morris Sr.

UNIX security · from the 1970s

Worked on UNIX. He made influential contributions to password security and system protection; the best known is *Password Security: A Case History* (1979), written with Ken Thompson.

WHY IT MATTERS FOR ENGINEERING

This work links authentication, salted password hashing, privilege separation and threat modeling.

Security **UNIX** **Passwords**

Why it matters: This work links authentication, salted password hashing, privilege separation and threat modeling.

1970s+

Fred Brooks

OS/360 and *The Mythical Man-Month* · 1960s–1970s

At IBM, after the System/360 hardware, he took over management of the delayed OS/360 project in 1964. He described his experience in the book *The Mythical Man-Month*, published in 1975; its most famous observation is that

into the enemy's hands: security should depend only on the secrecy of the key, not on the secrecy of the algorithm.

WHY IT MATTERS FOR ENGINEERING

The principle explains why open design and a large key space matter; the encryption and authentication mechanisms that operating systems use are designed under this assumption.

Kerckhoffs's principle **Encryption**

Key

Why it matters: The principle explains why open design and a large key space matter; the encryption and authentication mechanisms that operating systems use are designed under this assumption.

1883

László Bélády

The OPT algorithm and Bélády's anomaly · 1960s

Worked at the IBM Watson Research Center. In a 1966 paper in the *IBM Systems Journal* he showed that the algorithm that evicts the page that will be used farthest in the future (OPT or MIN) gives the fewest page faults. In 1969, together with Nelson and Shedler, he found that with FIFO, increasing the

adding people to a late software project makes it later (Brooks's law). He received the Turing Award in 1999.

WHY IT MATTERS FOR ENGINEERING

OS/360 was one of the first large-scale attempts to manage the complexity of a big operating system; Brooks's lessons are still the starting point of discussions of teams, interfaces and conceptual integrity in operating system design.

OS/360 **Software engineering**

Brooks's law

Why it matters: OS/360 was one of the first large-scale attempts to manage the complexity of a big operating system; Brooks's lessons are still the starting point of discussions of teams, interfaces and conceptual integrity in operating system design.

1964–1975

Gene Amdahl

Amdahl's law · 1967

One of the chief designers of the IBM System/360 architecture. At the 1967 AFIPS conference he argued that the serial part of a program that cannot be parallelized puts a hard upper bound on speedup. This observation was later formulated as Amdahl's law: if the serial fraction is s, the speedup cannot exceed 1/s no matter how many cores are added.

number of frames can increase the number of page faults; this is called Bélády's anomaly.

WHY IT MATTERS FOR ENGINEERING

OPT cannot be implemented, but it is the lower bound against which other replacement algorithms are measured. The anomaly also shows the property that separates stack algorithms such as LRU from FIFO.

Page replacement **OPT**

Bélády's anomaly

Why it matters: OPT cannot be implemented, but it is the lower bound against which other replacement algorithms are measured. The anomaly also shows the property that separates stack algorithms such as LRU from FIFO.

1966–1969

WHY IT MATTERS FOR ENGINEERING

The law explains why adding more threads on a multicore processor yields less and less beyond a point, and shows why shortening serial sections guarded by locks matters.

Amdahl's law **Speedup**

Parallelism

Why it matters: The law explains why adding more threads on a multicore processor yields less and less beyond a point, and shows why shortening serial sections guarded by locks matters.

1967

Donald Knuth

Dynamic memory allocation and the 50-percent rule · 1968

In the first volume of *The Art of Computer Programming*, published in 1968, he analyzed dynamic memory allocation (first fit, best fit, the buddy system) in detail. The 50-percent rule comes out of this analysis: in a system running first fit, when there are N allocated blocks there are on average about 0.5 N free holes, so roughly one third of memory can become unusable because of fragmentation.

Edward G. Coffman Jr.

The four conditions for deadlock · 1971

In the paper "System Deadlocks," published with M. J. Elphick and A. Shoshani in *ACM Computing Surveys* in 1971, he listed the four necessary conditions for deadlock: mutual exclusion, hold and wait, no preemption and circular wait. These conditions are known by his name as the Coffman conditions. The paper also gives a

WHY IT MATTERS FOR ENGINEERING

The rule summarizes in a single number how large external fragmentation can get, and explains why methods that eliminate fragmentation, such as paging, are needed.

External fragmentation **First fit**

50-percent rule

Why it matters: The rule summarizes in a single number how large external fragmentation can get, and explains why methods that eliminate fragmentation, such as paging, are needed.

1968

Richard C. Holt

The resource allocation graph · 1971–1972

He presented "Some Deadlock Properties of Computer Systems" at SOSP in 1971 and published an extended version in *ACM Computing Surveys* in 1972. The graph model, which shows processes and resources as two kinds of nodes and requests and assignments as edges, is known today as the resource allocation graph.

WHY IT MATTERS FOR ENGINEERING

The graph makes visible how a deadlock arises and is the data structure on which detection algorithms operate: if each deadlock detection algorithm for resources with multiple instances of each type.

WHY IT MATTERS FOR ENGINEERING

Every deadlock prevention method works by breaking one of these four conditions; the detection algorithm is also the basis of the form taught in textbooks today.

Deadlock **Coffman conditions**

Detection

Why it matters: Every deadlock prevention method works by breaking one of these four conditions; the detection algorithm is also the basis of the form taught in textbooks today.

1971

C. L. Liu and James W. Layland

Rate-monotonic scheduling and EDF · 1973

Liu, of MIT Project MAC, and Layland, of the Jet Propulsion Laboratory, studied hard real-time scheduling of periodic tasks in a paper published in the *Journal of the ACM* in 1973. They proved that rate-monotonic (RM) priorities are optimal among fixed-priority methods and gave the utilization bound $n(2^{1/n} - 1)$ for n tasks. The same paper shows that any task set whose total utilization does not exceed 1 can be scheduled by earliest deadline first (EDF).

resource has a single instance, a cycle in the graph means deadlock.

Resource allocation graph **Deadlock**

Graph model

Why it matters: The graph makes visible how a deadlock arises and is the data structure on which detection algorithms operate: if each resource has a single instance, a cycle in the graph means deadlock.

1972

WHY IT MATTERS FOR ENGINEERING

The Liu–Layland test tells whether a task set can run without missing deadlines, without building the schedule; it is the theoretical basis of priority assignment in real-time operating systems.

Real-time scheduling **RM** **EDF**

Why it matters: The Liu–Layland test tells whether a task set can run without missing deadlines, without building the schedule; it is the theoretical basis of priority assignment in real-time operating systems.

1973

David E. Bell and Leonard J. LaPadula

The Bell–LaPadula model · 1973

At the MITRE Corporation in 1973 they developed a formal model for multilevel security. The model has two basic rules: by the simple security property, a process cannot read objects above its own level (no read up); by the * property, it cannot write to objects below its own level (no write down).

WHY IT MATTERS FOR ENGINEERING

The model uses rules to prevent information from leaking from a high confidentiality level to a low one; it is the computer counterpart of mandatory access control (MAC) and of military classification schemes.

Jerome Saltzer and Michael Schroeder

Principles of protection and secure design · 1975

In their 1975 paper "The Protection of Information in Computer Systems," published in the *Proceedings of the IEEE*, they listed eight principles for secure system design. Among them are keeping the mechanism simple, making the default deny, checking every access, open design, privilege separation and least privilege. Saltzer had also worked on the Multics project.

WHY IT MATTERS FOR ENGINEERING

The principles offer a common yardstick, still used today, for evaluating operating system mechanisms such as access

Multilevel security **Confidentiality**

Mandatory access control

Why it matters: The model uses rules to prevent information from leaking from a high confidentiality level to a low one; it is the computer counterpart of mandatory access control (MAC) and of military classification schemes.

1973

control, setuid programs and privilege separation.

Protection **Least privilege**

Design principles

Why it matters: The principles offer a common yardstick, still used today, for evaluating operating system mechanisms such as access control, setuid programs and privilege separation.

1975

Whitfield Diffie and Martin Hellman

Public-key cryptography · 1976

In their 1976 paper "New Directions in Cryptography," published in *IEEE Transactions on Information Theory*, they introduced the idea of public-key cryptography and the key exchange that bears their names. Two parties can produce a shared secret key by sending only public values over an insecure channel. They shared the 2015 Turing Award.

WHY IT MATTERS FOR ENGINEERING

This exchange, which needs no pre-shared key, underlies the TLS handshake and digital signatures; the network authentication and secure communication services of operating systems rely on it.

Public key **Key exchange** **TLS**

Kenneth J. Biba

The Biba integrity model · 1977

In the MITRE Corporation report "Integrity Considerations for Secure Computer Systems" (1977) he described a model for protecting integrity, based on rules that are the reverse of the Bell–LaPadula model. A process cannot read objects at a lower integrity level than its own (no read down) and cannot write to objects at a higher level (no write up).

WHY IT MATTERS FOR ENGINEERING

The model keeps untrusted data from corrupting trusted components; mandatory integrity controls such as integrity levels in Windows are applications of this idea.

Integrity **Multilevel security**

Mandatory access control

Why it matters: This exchange, which needs no pre-shared key, underlies the TLS handshake and digital signatures; the network authentication and secure communication services of operating systems rely on it.

1976

Gary L. Peterson

Peterson's algorithm · 1981

In “Myths About the Mutual Exclusion Problem,” published in *Information Processing Letters* in 1981, he gave a short mutual exclusion solution for two processes that works correctly with only reads and writes. The algorithm combines each process's “I want to enter” flag with a single variable that records whose turn it is.

WHY IT MATTERS FOR ENGINEERING

The algorithm is the classic example of a solution that satisfies the three requirements of the critical-section problem (mutual exclusion, progress, bounded waiting); the fact that it breaks on modern processors without a memory barrier also shows the importance of memory ordering.

Critical section **Mutual exclusion**

Memory ordering

Why it matters: The algorithm is the classic example of a solution that satisfies

Why it matters: The model keeps untrusted data from corrupting trusted components; mandatory integrity controls such as integrity levels in Windows are applications of this idea.

1977

Richard W. Carr and John L. Hennessy

WSClock · 1981

They presented the WSClock algorithm, developed at Stanford University, at SOSP in 1981. The algorithm combines the working-set idea with the circular frame list of the clock algorithm: the hand moves over the frames in turn and stops at the first suitable victim without scanning the whole table.

WHY IT MATTERS FOR ENGINEERING

WSClock is a practical form of the working-set model; it uses locality information in page replacement at low cost.

WSClock **Working set**

Page replacement

Why it matters: WSClock is a practical form of the working-set model; it uses locality information in page replacement at low cost.

1981

the three requirements of the critical-section problem (mutual exclusion, progress, bounded waiting); the fact that it breaks on modern processors without a memory barrier also shows the importance of memory ordering.

1981

Andrew D. Birrell and Bruce J. Nelson

Remote procedure call (RPC) · 1984

They described the RPC package they developed for the Cedar environment at Xerox PARC in the paper "Implementing Remote Procedure Calls," published in *ACM Transactions on Computer Systems* in 1984. A procedure on another machine can be called with the syntax of an ordinary local call, thanks to automatically generated client and server stubs.

WHY IT MATTERS FOR ENGINEERING

RPC is the basic distributed-systems abstraction that hides the details of message passing from the programmer; distributed file systems such as NFS and today's gRPC-like frameworks are built on this model.

RPC **Stubs** **Distributed systems**

Why it matters: RPC is the basic distributed-systems abstraction that hides the details of message passing from the programmer; distributed file

John Gustafson

Gustafson's law · 1988

Based on his work on a 1024-processor system at Sandia National Laboratories, he published the note "Reevaluating Amdahl's Law" in *Communications of the ACM* in 1988. He showed that as the number of cores grows, the problem usually grows too, and in that case the scaled speedup increases almost linearly with the number of cores.

WHY IT MATTERS FOR ENGINEERING

Amdahl answers "how fast will this job finish?", while Gustafson answers "how large a job can be done in the same time?"; together they are the two sides of evaluating how parallel systems scale.

Scaled speedup **Parallelism**

Scalability

Why it matters: Amdahl answers "how fast will this job finish?", while Gustafson answers "how large a job can be done in the same time?"; together they are the two sides of evaluating how parallel systems scale.

systems such as NFS and today's gRPC-like frameworks are built on this model.

1984

David Patterson, Garth Gibson and Randy Katz

RAID · 1988

In the paper "A Case for Redundant Arrays of Inexpensive Disks (RAID)," presented at SIGMOD at UC Berkeley in 1988, they proposed building arrays of small, cheap disks protected by redundancy instead of one large, expensive disk. The paper defined five RAID levels and compared their cost and performance. The "I" in the acronym was later redefined as "independent."

WHY IT MATTERS FOR ENGINEERING

Striping, mirroring and parity are the basic reliability and performance methods used today in the software RAID layers of operating systems and in storage controllers.

RAID **Redundancy** **Storage**

Why it matters: Striping, mirroring and parity are the basic reliability and performance methods used today in the software RAID layers of operating systems and in storage controllers.

1988

Mendel Rosenblum and John Ousterhout

The log-structured file system (LFS) · 1991

They presented the log-structured file system (LFS) they developed for the Sprite operating system at SOSP in 1991. The design assumes that most reads will be served from the cache and that the traffic to disk will be mostly writes; so all changes are written to disk as a single long sequential log. The log is divided into segments, and a cleaner compacts the live data in scattered segments to free space.

WHY IT MATTERS FOR ENGINEERING

By making writes sequential, LFS gained a large performance improvement on small-file workloads and sped up crash recovery; its ideas live on in journaling and copy-on-write file systems and in the translation layers of SSDs.

LFS **Log-structured** **Crash recovery**

Why it matters: By making writes sequential, LFS gained a large performance improvement on small-file workloads and sped up crash recovery; its ideas live on in journaling and copy-on-write file systems and in the translation layers of SSDs.

1988

John Mellor-Crummey and Michael L. Scott

The MCS queue lock · 1991

In their paper published in *ACM Transactions on Computer Systems* in 1991, they proposed the MCS queue lock, named after their initials. The waiters form a linked list; each waiter spins on its own flag in a separate cache line, and releasing the lock takes a single remote write. This work received the 2006 Dijkstra Prize.

WHY IT MATTERS FOR ENGINEERING

Locks that spin on a single shared variable drown in cache-coherence traffic as the number of cores grows; MCS is the fundamental example of scalable synchronization that solves this problem, and Linux's qspinlock is based on this idea.

Spinlock **Scalable synchronization**

Cache coherence

Why it matters: Locks that spin on a single shared variable drown in cache-coherence traffic as the number of cores grows; MCS is the fundamental example of scalable synchronization that solves this problem, and Linux's qspinlock is based on this idea.

1991

1991

Carl Waldspurger and William Weihl

Lottery and stride scheduling · 1994–1995

In lottery scheduling, which they presented at OSDI in 1994 at MIT, every process is given tickets and in each time slice the owner of a randomly drawn ticket runs; more tickets mean a larger share of the processor. In 1995 Waldspurger and Weihl proposed stride scheduling, which provides the same shares without randomness.

WHY IT MATTERS FOR ENGINEERING

These methods implemented the idea of proportional share in a simple and understandable way; the weighted fair sharing in Linux's CFS and EEVDF schedulers comes from the same lineage.

Proportional share

Lottery scheduling **Stride scheduling**

Why it matters: These methods implemented the idea of proportional share in a simple and understandable way; the weighted fair sharing in Linux's CFS and EEVDF schedulers comes from the same lineage.

1994–1995

Jeff Bonwick

The slab allocator · 1994

He published the slab allocator he designed at Sun Microsystems for the SunOS 5.4 (Solaris) kernel at the USENIX Summer conference in 1994. The allocator keeps frequently used kernel objects in prepared caches; objects are exactly sized and packed side by side within a page.

WHY IT MATTERS FOR ENGINEERING

The slab allocator reduces internal fragmentation and initialization cost for kernel objects; Linux's SLAB and SLUB allocators derive from this design.

Slab allocator **Object cache**

Kernel memory

Why it matters: The slab allocator reduces internal fragmentation and initialization cost for kernel objects; Linux's SLAB and SLUB allocators derive from this design.

1994

Dawson Engler and Frans Kaashoek

Exokernel · 1995

In the exokernel architecture they presented at SOSP in 1995 at MIT, together with James O'Toole, the kernel only allocates hardware resources in a protected way and tracks ownership. Abstractions such as virtual memory and files are provided by library operating systems running in user mode. In the prototype the kernel is called Aegis and the library operating system is called ExOS.

WHY IT MATTERS FOR ENGINEERING

The slogan “separate management from protection” is the extreme end of the debate over how much functionality belongs in the kernel; the idea lives on today in unikernels and in I/O libraries that bypass the kernel, such as DPDK and SPDK.

Exokernel **Library OS**

Kernel architecture

Why it matters: The slogan “separate management from protection” is the extreme end of the debate over how much functionality belongs in the kernel; the idea lives on today in unikernels and in I/O libraries that bypass the kernel, such as DPDK and SPDK.

1995

Ion Stoica and Hussein Abdel-Wahab

EEVDF · 1995

The "Earliest Eligible Virtual Deadline First" (EEVDF) algorithm they proposed in 1995 bases proportional-share scheduling on two clear criteria: a task is eligible only when it has fallen behind its share, and among the eligible tasks the one with the earliest virtual deadline is chosen. Peter Zijlstra adapted the algorithm for the Linux kernel; EEVDF replaced CFS starting with version 6.6.

WHY IT MATTERS FOR ENGINEERING

EEVDF can give latency-sensitive tasks shorter time slices while preserving fairness; it is now the default scheduler for ordinary tasks in Linux.

EEVDF **Proportional share**

Linux scheduler

Why it matters: EEVDF can give latency-sensitive tasks shorter time slices while preserving fairness; it is now the default scheduler for ordinary tasks in Linux.

1995

Ingo Molnár

The O(1) scheduler and CFS · 2000s

He wrote the O(1) scheduler that arrived in the Linux kernel with version 2.6.0. In 2007 he developed the Completely Fair Scheduler (CFS), which replaced the O(1) scheduler in version 2.6.23. CFS emulates an ideal multitasking processor: it keeps each task's virtual runtime (vruntime) and runs the task that has fallen furthest behind in a red-black tree.

WHY IT MATTERS FOR ENGINEERING

CFS was the default scheduler for ordinary tasks in Linux for more than fifteen years and made weighted fair sharing practical at large scale.

CFS **vruntime** **Linux scheduler**

Why it matters: CFS was the default scheduler for ordinary tasks in Linux for more than fifteen years and made weighted fair sharing practical at large scale.

2002–2007