Chapter 1: Introduction

Operating Systems

What is an operating system? | User and kernel mode | Extended machine | Resource manager | History | Processor and registers | Memory hierarchy | Multicore chips | Polling, interrupts, DMA | Kinds of OS | Policy and mechanism | From program to process

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_01_Introduction.html

Learning Outcomes

Definition and place

Explain where the OS sits between hardware and applications, and 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 batch processing, multiprogramming and time sharing solved.

Hardware basics

Explain how the processor and registers, the memory hierarchy and multicore chips shape the OS.

I/O methods

State the key difference between polling, interrupts and DMA; tell an interrupt from a trap.

Concepts

Explain, with examples, the kinds of OS, the policy–mechanism split and how a program becomes an executable.

What Is an Operating System?

- A modern computer has a multicore processor, gigabytes of memory, an SSD, a network card, a GPU and dozens of devices.
- No application developer can know **every detail** of this hardware and program it directly.
- **Operating system (OS)**: the software layer that manages the hardware and gives applications a **simple, safe and common** interface.
- **Kernel**: the privileged part that is always resident in memory; the "operating system" in the narrow sense.
- Users interact with the computer through a **shell** (command line) or a **GUI** (graphical user interface).

Desktop

Windows 11, macOS, Linux distributions (Ubuntu, Fedora), ChromeOS

Mobile

Android, iOS / iPadOS, HarmonyOS

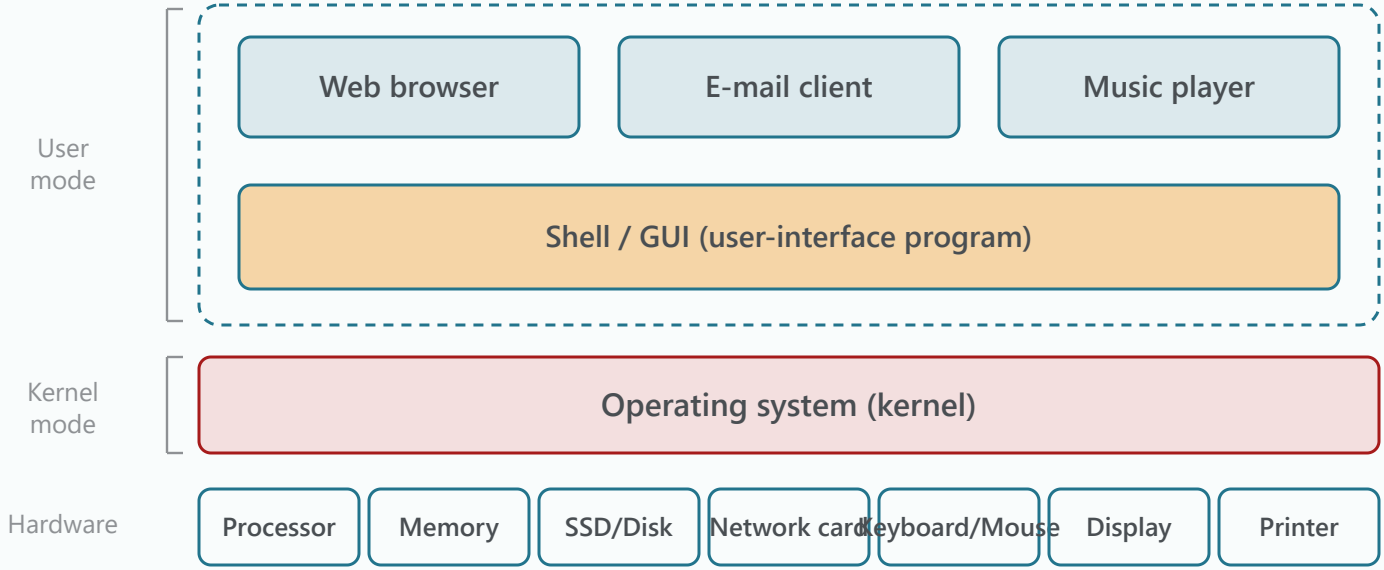
Server and cloud

Linux, Windows Server, FreeBSD

Embedded and IoT

Zephyr, FreeRTOS, Contiki-NG, VxWorks, QNX

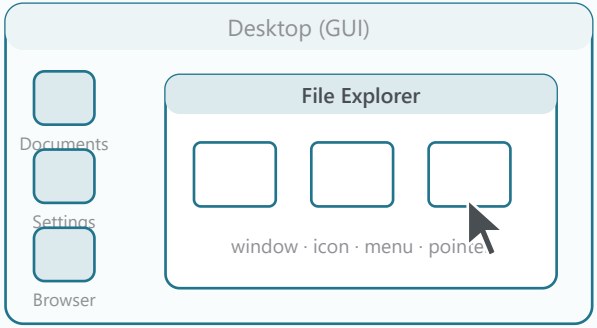
Components of a Computer System



Hardware: processor, memory, storage and I/O devices. **Software:** the OS running in kernel mode and everything that runs in user mode; even the shell/GUI is an ordinary user program.

Shell and GUI





- **Shell**: a command-line interface (CLI) · bash, zsh, fish, PowerShell, cmd.exe.
- **GUI** (graphical user interface): visual interaction with windows, icons, menus and a pointer (WIMP) · Windows Explorer, macOS Finder, GNOME, KDE Plasma.
- Neither is **part of the kernel**: both are user-mode programs that request kernel services through **system calls**. So they can be replaced and customized, and several can run at once.

INTRODUCTION

Shell or GUI?

Shell

- **Automation** with scripts: repeating a task a thousand times is easy.
- Pipelines: combine small tools, as in `ps aux | grep ssh | wc -l`.
- Light on resources; remote server administration over SSH.
- Does specific tasks quickly and precisely.

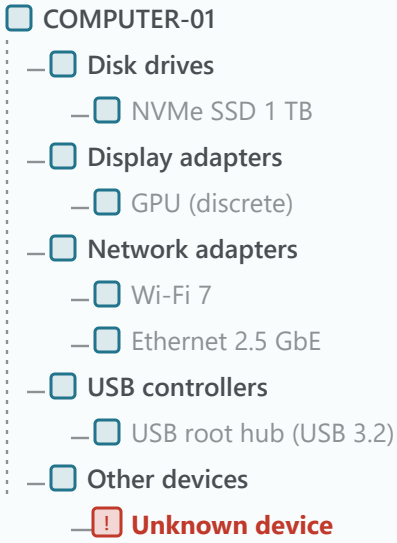
GUI

- **Intuitive** and discoverable: no commands to memorize.
- Natural for visual content (photos, maps, design).
- Reaches a wide audience; accessibility tools.
- Touch and voice interfaces belong to this family too.

They are not rivals but **complements**: the GUI for everyday work, the shell for repetitive and remote work. Both use the same kernel services; the kernel does not know which interface a request came from.

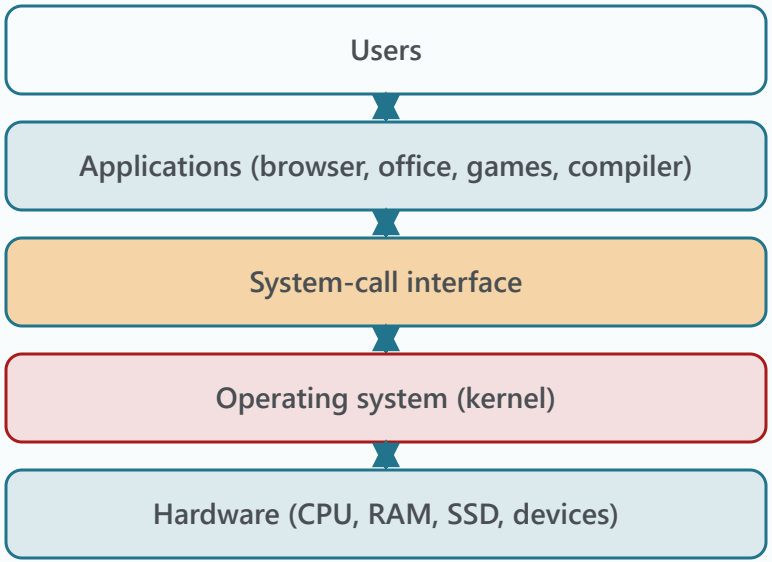
Device Manager

- A Windows administrative tool that lists the hardware devices attached to the computer and their **drivers** (`devmgmt.msc`, an MMC snap-in).
- Shows a device's type, vendor, driver version and the resources it uses (interrupt, memory range).
- Update or roll back a driver, disable a device; problem devices are flagged with a warning icon.
- The real work is done by the Plug and Play (PnP) manager and device drivers **in the kernel**; the tool is just a *view* of them.
- Linux equivalents: `lspci`, `lsusb`, `lsblk`, `lsmod`, `/sys` and `udev`.



Where the Operating System Sits

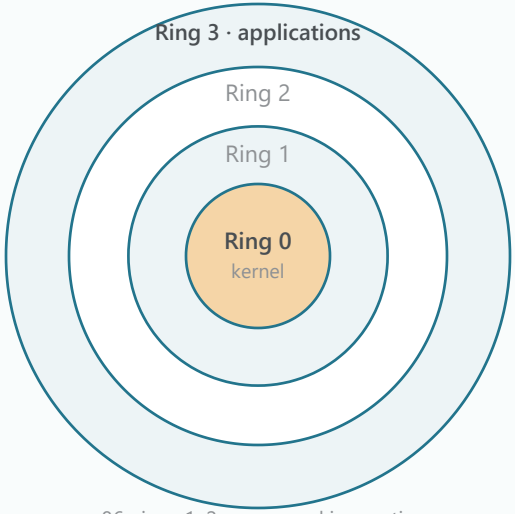
- The OS is the interface between **hardware** (the physical parts: CPU, RAM, disk) and **software** (code describing the work the computer should do).
- Applications reach the hardware not directly but through the **system calls** the OS provides.



- controls how software uses the hardware
- optimizes hardware utilization
- provides security and isolation

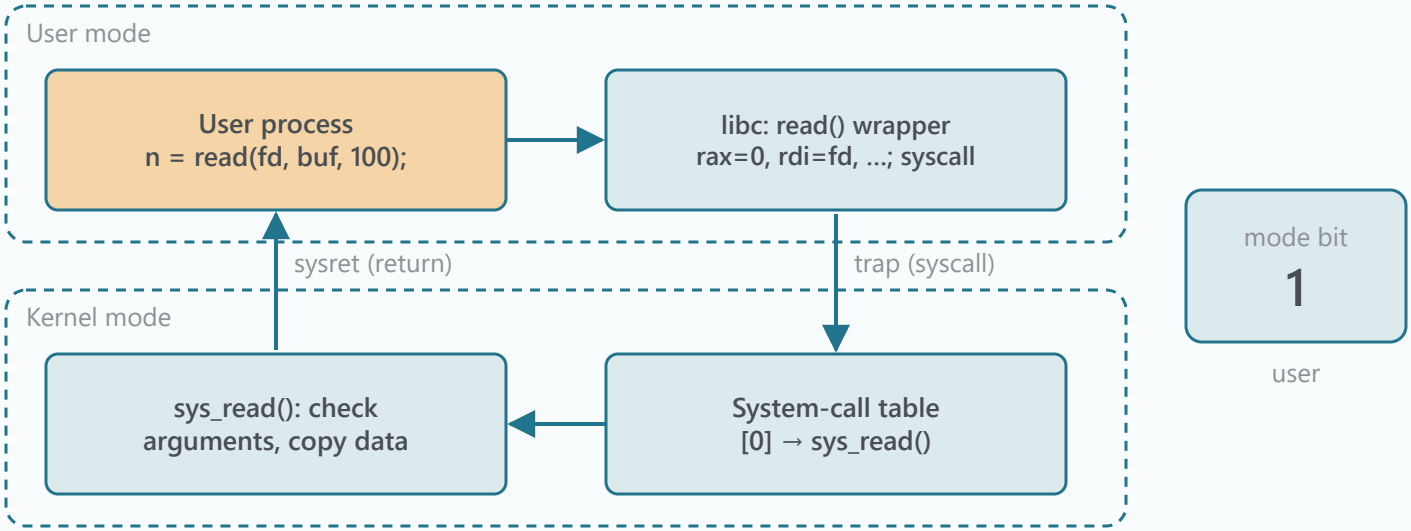
Kernel Mode and User Mode

- The processor has a **mode bit** / privilege level: x86 rings 0–3, ARMv8 EL0–EL3, RISC-V U/S/M.
- **Kernel mode**: every instruction and all of memory are available; *privileged* instructions such as I/O, disabling interrupts or changing the page table are allowed.
- **User mode**: privileged instructions are forbidden; trying one makes the processor raise an **exception (trap)**, and the kernel may kill the process.
- The shell and GUI are the lowest level of user-mode software.
- User code enters the kernel only through **controlled gates**: system call, interrupt, exception.



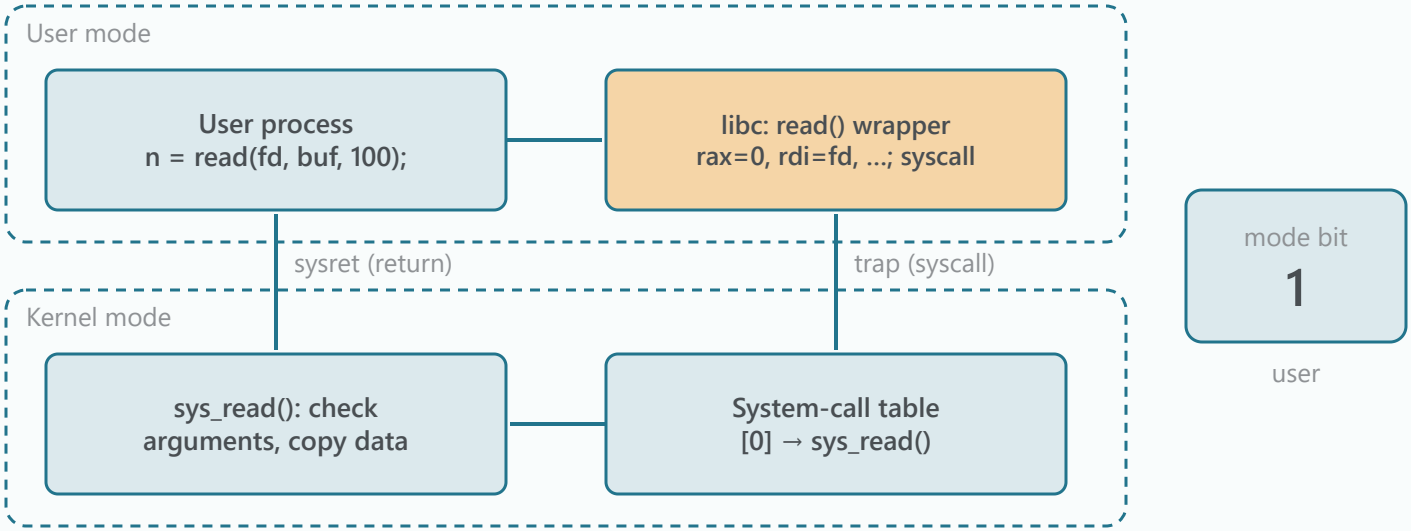
x86: rings 1–2 are unused in practice

Mode Switch Step by Step: the `read()` System Call



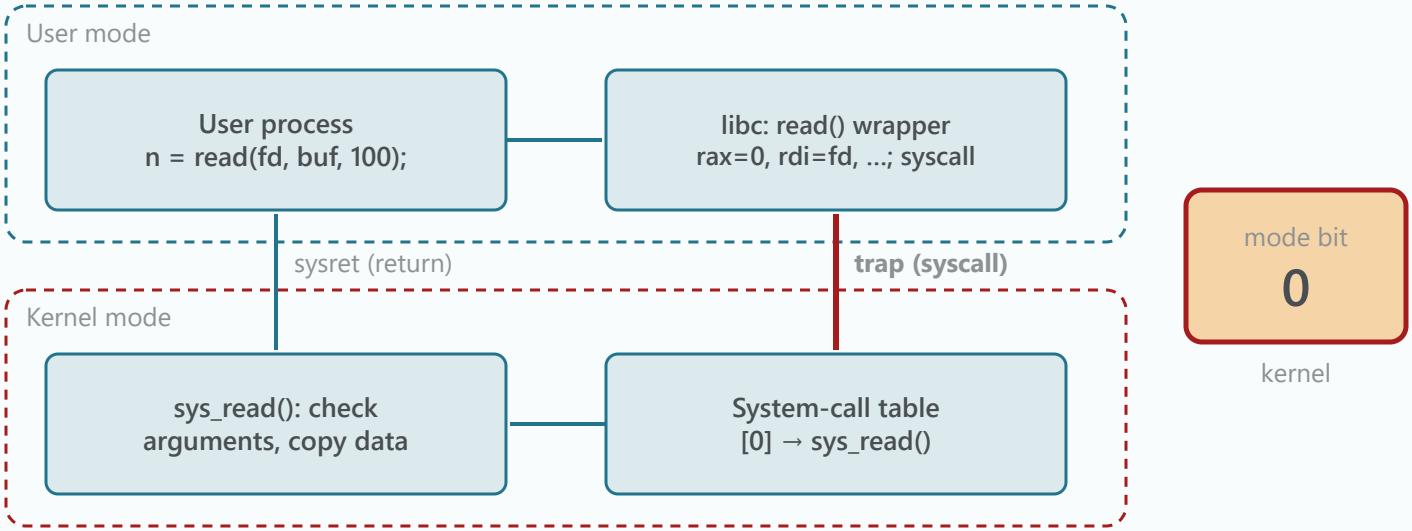
The process runs in **user mode** (mode bit = 1). It wants to read a file but cannot touch the disk itself.

Mode Switch Step by Step: the `read()` System Call



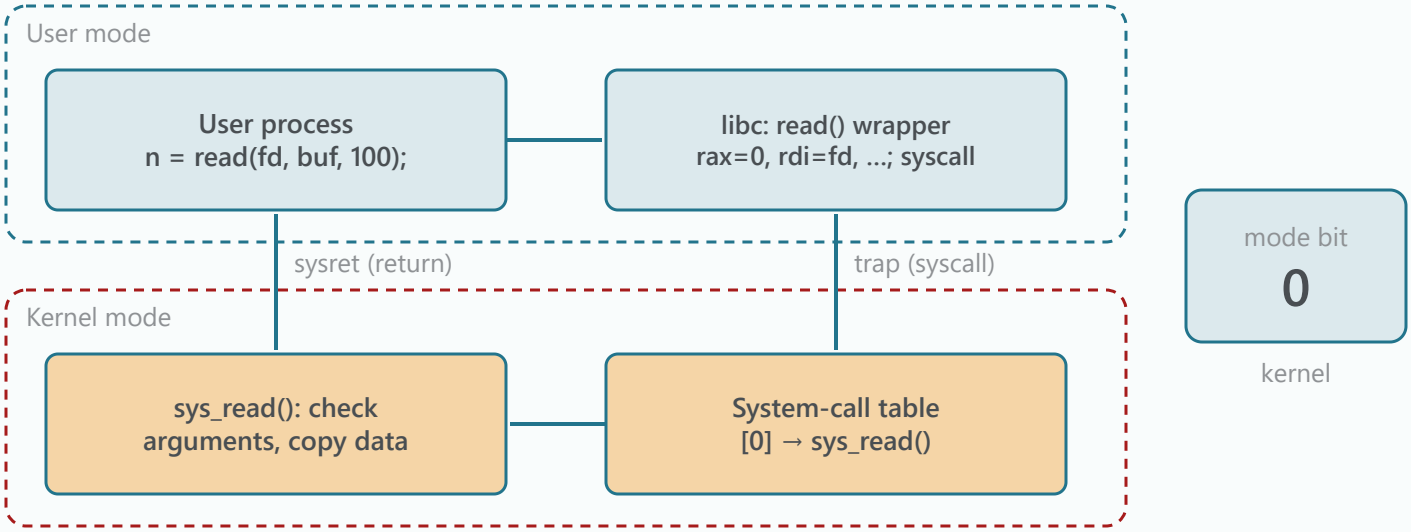
The `read()` wrapper in the C library puts the call number (x86-64 Linux: read = **0**, in `rax`) and the arguments in registers and executes `syscall`.

Mode Switch Step by Step: the `read()` System Call



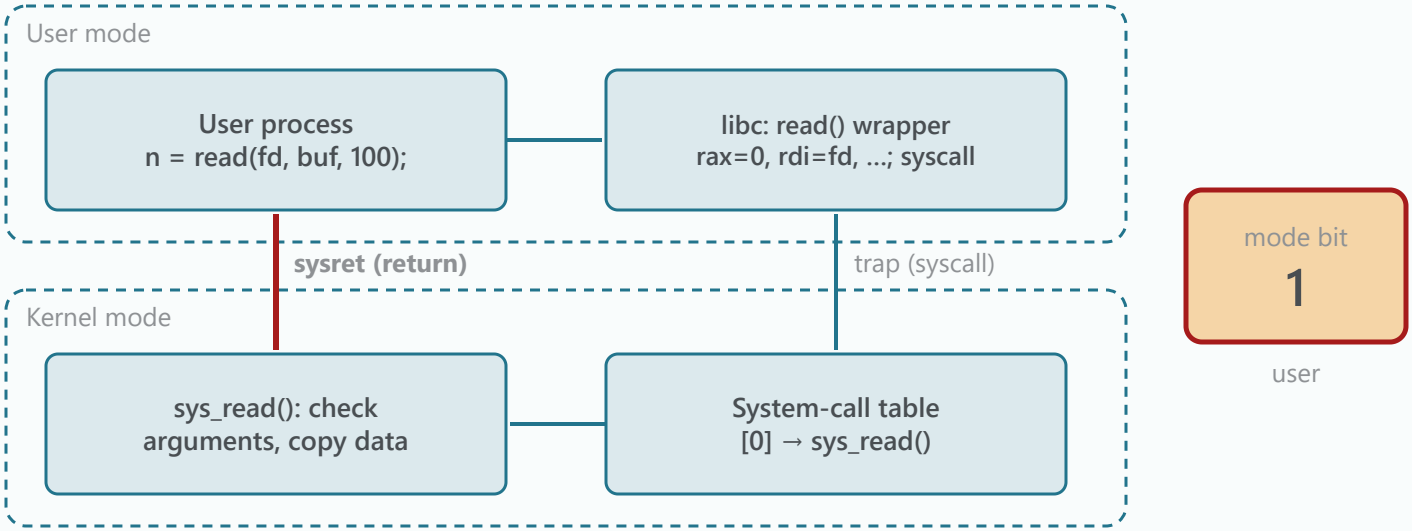
Trap: the hardware sets the mode bit to **0**, saves the return address and jumps to a **fixed** kernel entry point. The process cannot choose where it lands.

Mode Switch Step by Step: the `read()` System Call



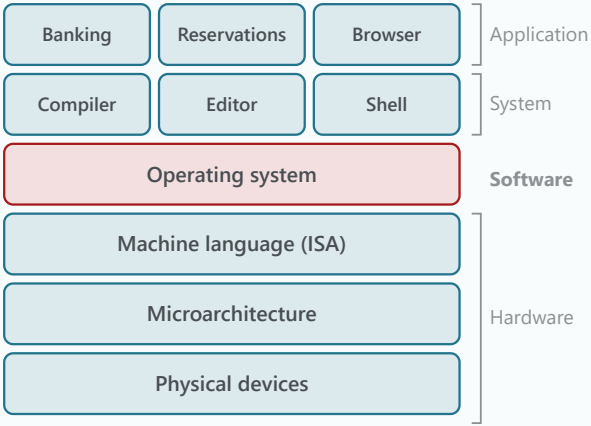
The kernel uses the number to find `sys_read()` in the **system-call table**, validates the arguments (does buf really belong to the process?) and does the work.

Mode Switch Step by Step: the `read()` System Call



The result (bytes read) goes into `rax`; `sysret` sets the mode bit back to **1** and the process continues after `read()`.

Physical Devices, Microarchitecture, Machine Language



Physical devices

Chips, memory modules, SSDs, I/O devices.

Microarchitecture

The processor's internal design: datapath, registers, ALU, caches, control unit.

Machine language (ISA)

Binary instructions the hardware executes directly: x86-64, ARM64, RISC-V.

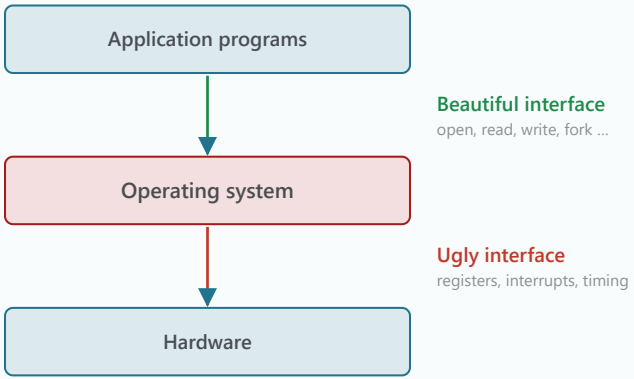
The OS's job

Drivers turn requests from above into low-level commands the device understands.

A *compiler* translates a program into machine language; the OS does not. The OS loads and runs the compiled program and mediates its access to the hardware.

The OS as an Extended Machine

- Software built on top of the hardware that **makes the hardware easier to use**.
- Makes the hardware's capabilities usable and **hides** its messy details.
- Keeps applications from touching the hardware **directly**, for both convenience and protection.
- Example: writing a block to an SSD involves NVMe queues, LBAs, error codes… The application just says `write(fd, buf, n)`.
- In Tanenbaum's words, the OS turns **ugly hardware into beautiful abstractions**.



Abstractions and Roles

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

Each abstraction makes a limited, shared hardware resource appear private to each program.

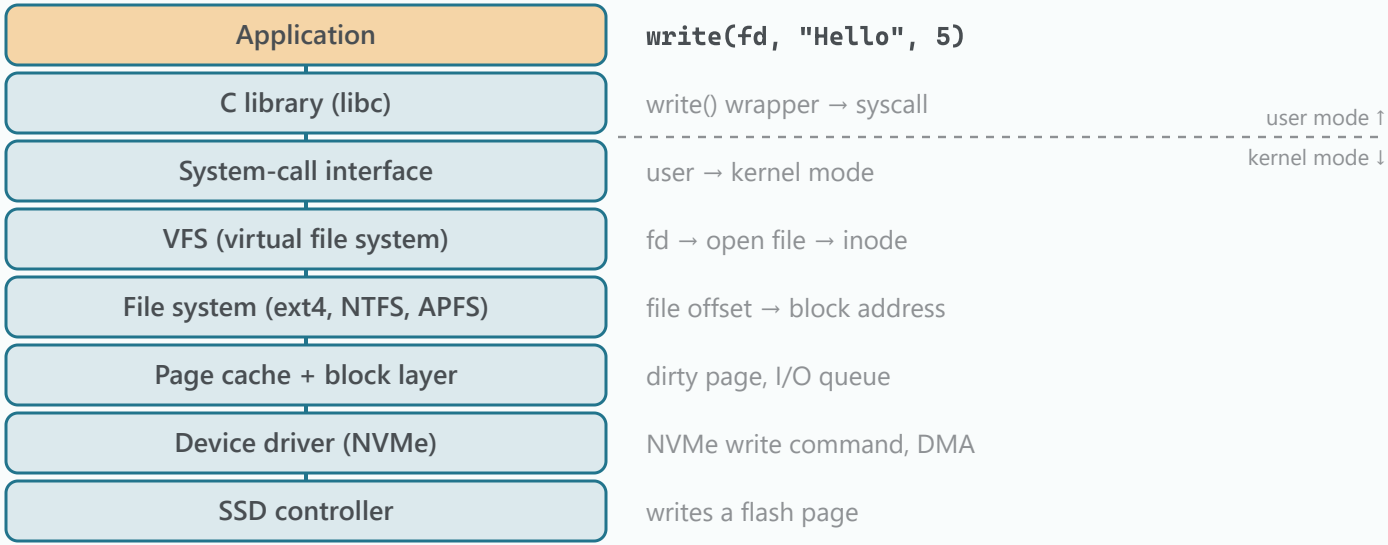
Hardware designer
Designs the ISA, the devices and their interfaces.

Kernel developer
Implements the abstractions on the hardware.

Application developer
Uses the abstractions through system calls and APIs.

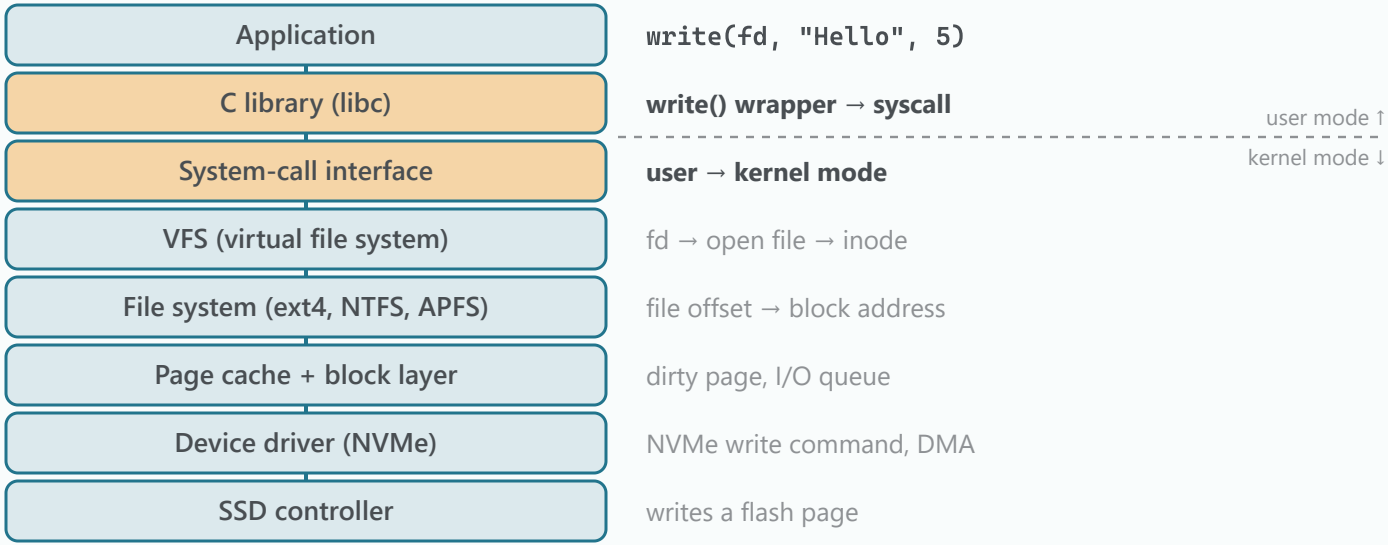
End user
Sees only the applications and the interface.

The Journey of a `write()` Call



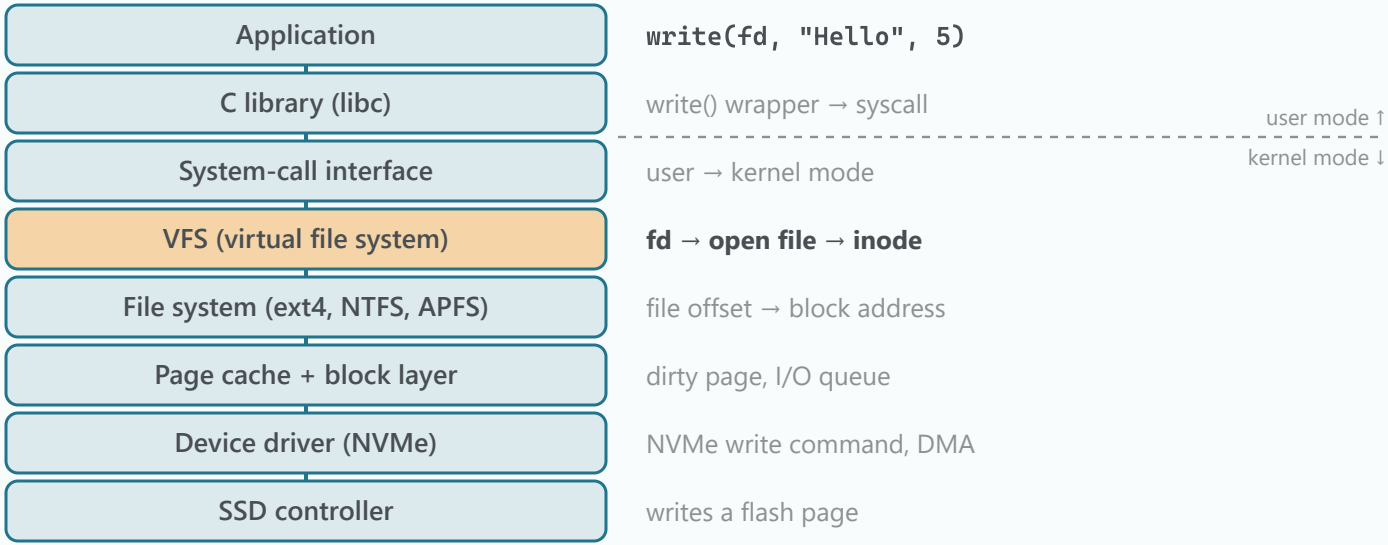
The application supplies only a **file descriptor** (fd), a buffer and a length; it does not even know what kind of disk it is.

The Journey of a `write()` Call



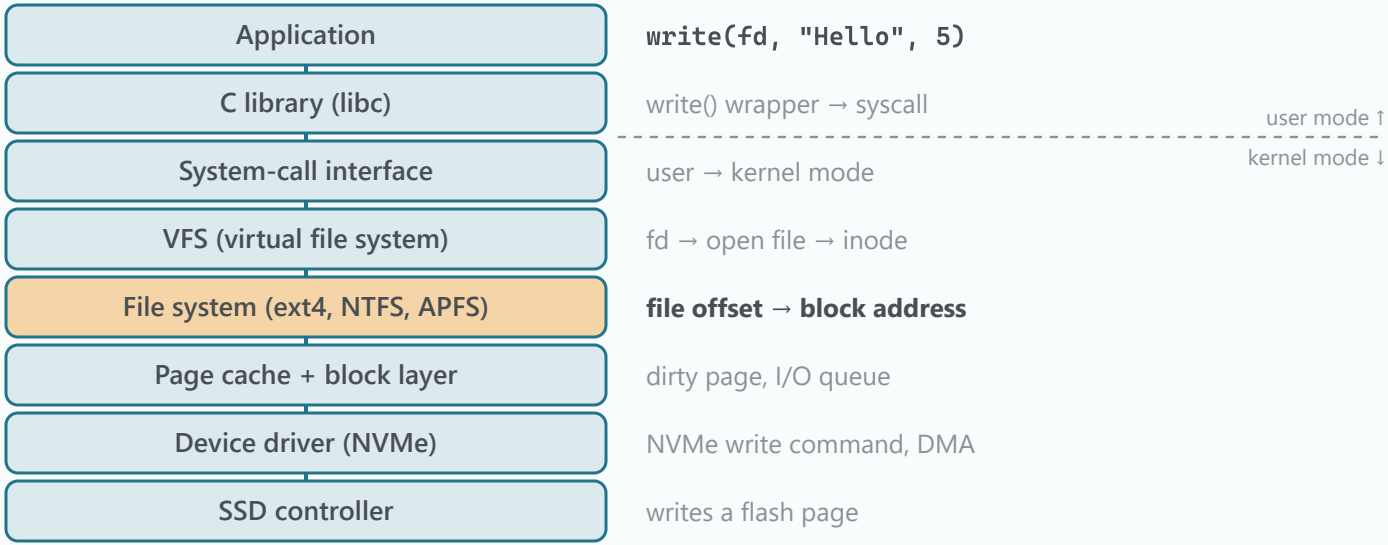
The libc wrapper puts the arguments in registers and enters the kernel with **syscall**.

The Journey of a `write()` Call



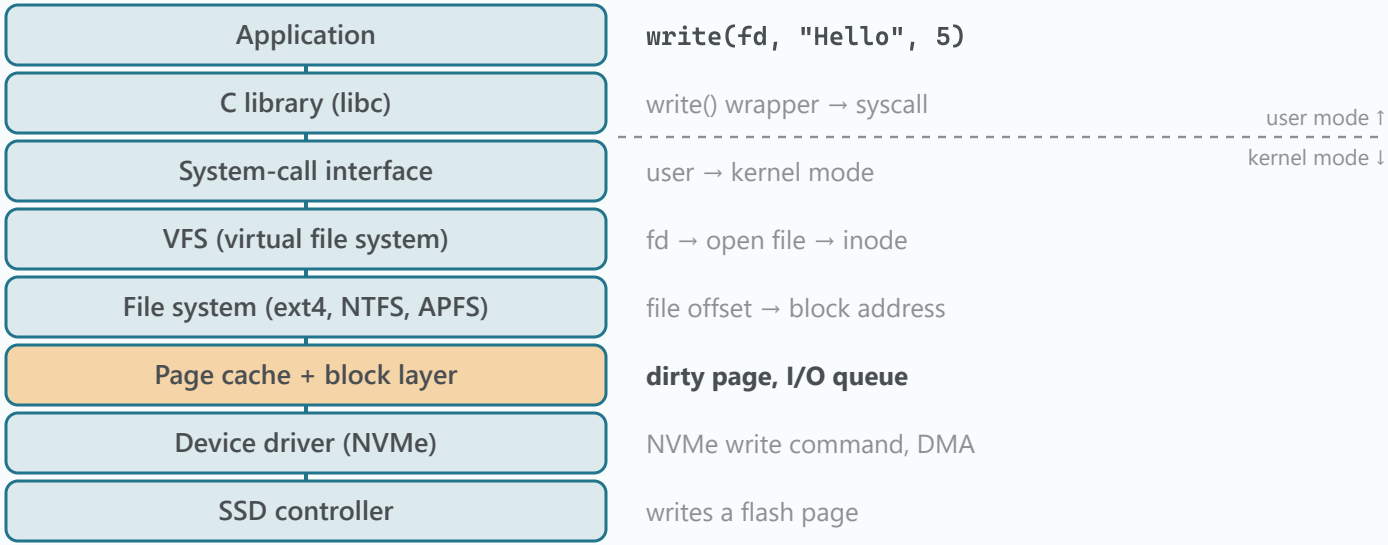
VFS: the fd is resolved through the process's open-file table to a file object and its **i-node**. One common layer for all file systems.

The Journey of a `write()` Call



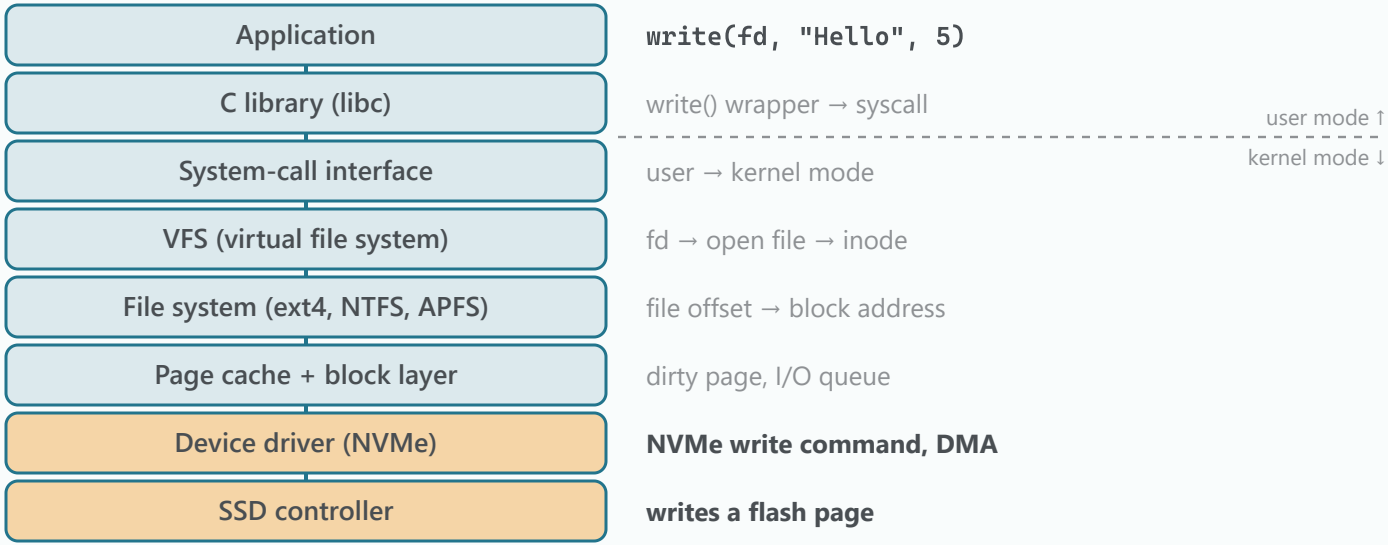
The file system maps the position in the file (the offset) to a **block address** on disk, allocating a new block if needed.

The Journey of a `write()` Call



Data usually goes into the **page cache** first and `write()` returns at once; the disk write happens later (`fsync()` for durability).

The Journey of a `write()` Call



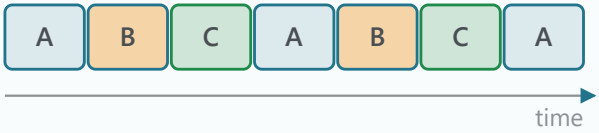
The driver builds an **NVMe command**; the SSD fetches the data from memory by DMA, writes it to flash and raises an interrupt when done.

The OS as a Resource Manager

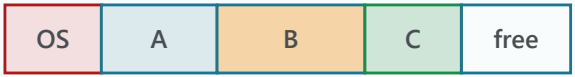
- Lets several programs run **at the same time**, and **allocates and protects** resources such as the CPU, memory and I/O devices.
- Goal: use resources **efficiently** and share them **fairly** among programs.
- **Top-down** view: it provides abstractions to applications. **Bottom-up** view: it manages the pieces of a complex system.
- Resources are multiplexed in two ways: in **time** (taking turns) and in **space** (dividing into parts).

Questions: what if several programs want to print at once? How is each process's resource usage measured? How do we keep fairness and efficiency as resources multiply?

Time multiplexing: one CPU

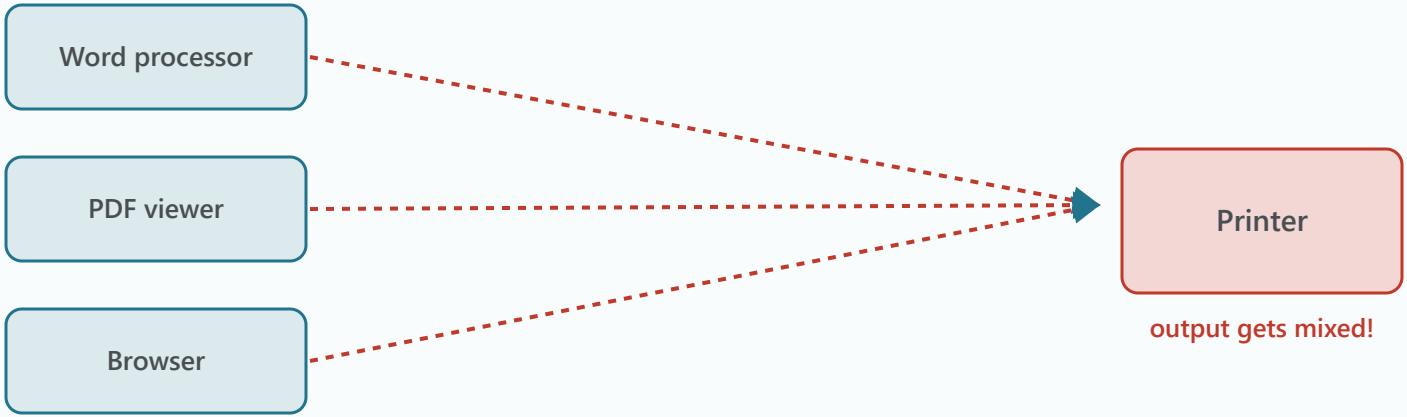


Space multiplexing: memory



Printer → time · disk blocks → space

When Several Programs Print at Once



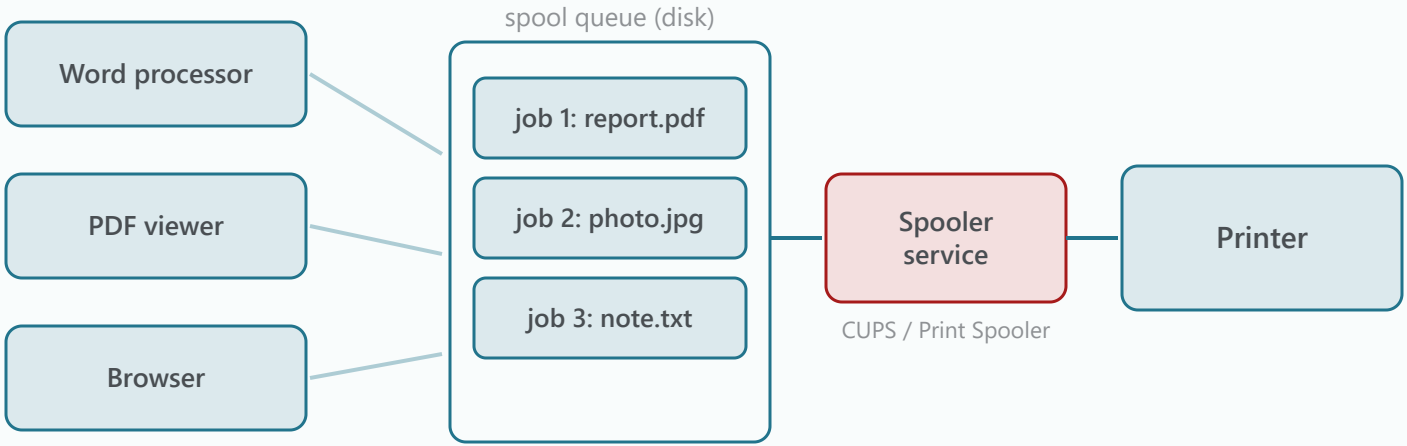
Three programs want to print almost at once, and there is only one printer. If all of them wrote to it **directly**, the pages would get interleaved.

When Several Programs Print at Once



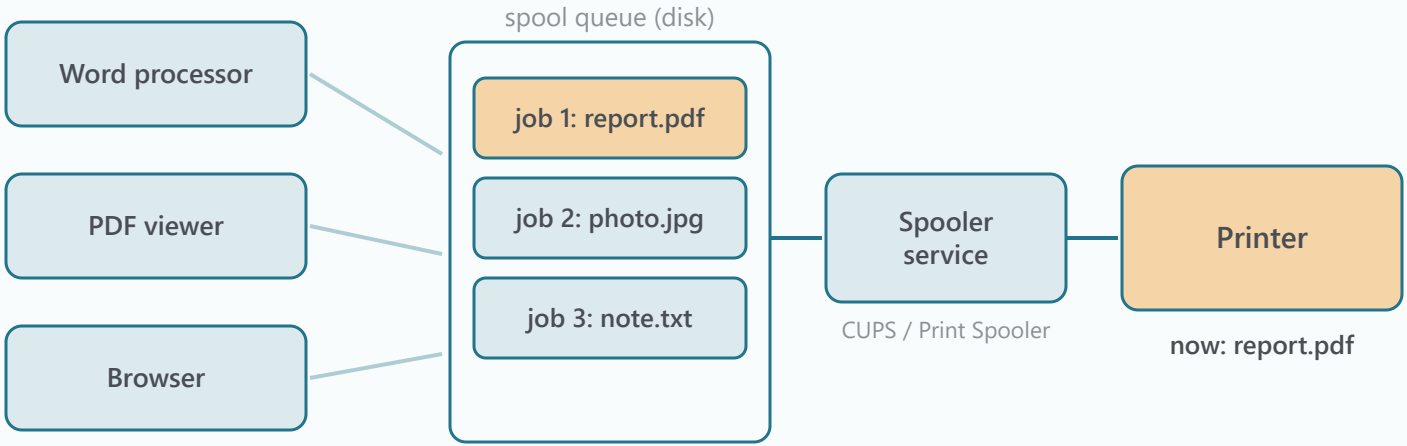
Programs write their jobs not to the printer but to the OS's **spool queue** on disk, and go straight back to work.

When Several Programs Print at Once



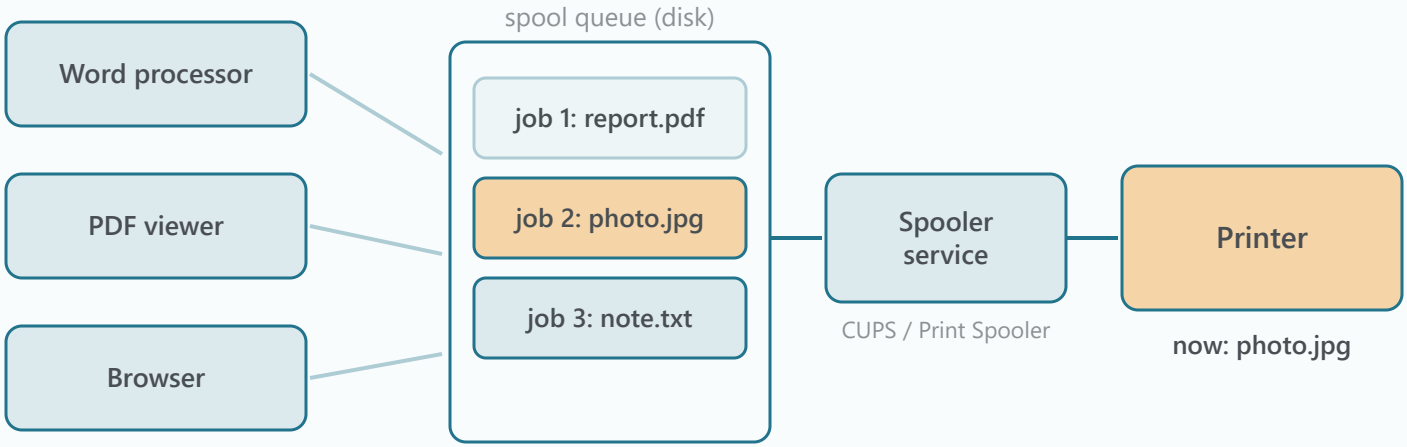
The **only process** that uses the printer is the spooler (Linux: CUPS, Windows: Print Spooler); this exclusive ownership prevents mixing.

When Several Programs Print at Once



The spooler usually sends jobs **in arrival order** (FCFS); an administrator can apply priorities or pause jobs. Job 1 is printing now.

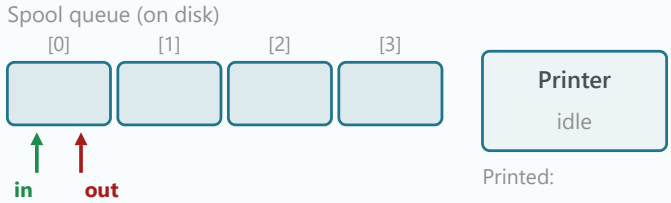
When Several Programs Print at Once



Job 1 is done; job 2 is next. While the slow printer worked, the programs and the CPU did other things: **I/O overlaps with computation**.

Code Trace: The Spool Queue

```
▶ char *queue[4];
▶ int in = 0, out = 0, count = 0;
3 void enqueue(char *file) {    /* programs */
4   queue[in] = file;
5   in = (in + 1) % 4;
6   count++;
7 }
8 void spooler(void) {
9   while (count > 0) {
10    send_to_printer(queue[out]);
11    queue[out] = NULL;    /* free slot */
12    out = (out + 1) % 4;
13    count--;
14  }
15 }
```



`count = 0`

The queue has 4 slots and is **circular**: `in` points to the next free slot, `out` to the first job to print. Empty queue: count = 0.

Step 1 / 11

VARIABLES

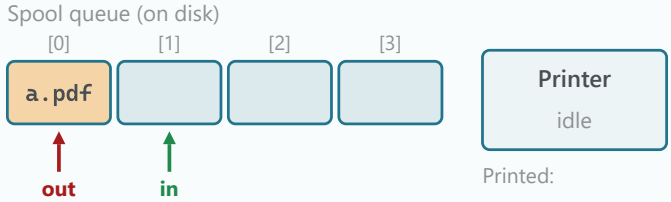
`in = 0` `out = 0` `count = 0`

CALL STACK

empty

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
▶    queue[in] = file;
▶    in = (in + 1) % 4;
▶    count++;
7  }
8  void spooler(void) {
9    while (count > 0) {
10     send_to_printer(queue[out]);
11     queue[out] = NULL;   /* free slot */
12     out = (out + 1) % 4;
13     count--;
14   }
15 }
```



`count = 1`

Program 1 calls `enqueue("a.pdf")`: the file goes into slot [0], in = 1, count = 1. The program returns to work without waiting for the printer.

Step 2 / 11

VARIABLES

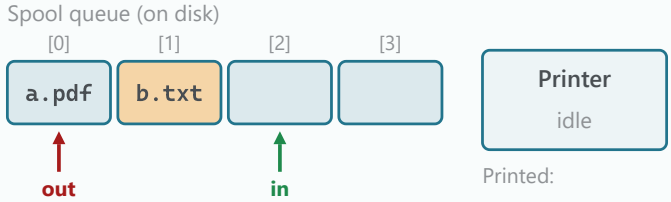
`in = 1` `out = 0` `count = 1`

CALL STACK

`enqueue("a.pdf")`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
▶    queue[in] = file;
▶    in = (in + 1) % 4;
▶    count++;
7  }
8  void spooler(void) {
9    while (count > 0) {
10     send_to_printer(queue[out]);
11     queue[out] = NULL;   /* free slot */
12     out = (out + 1) % 4;
13     count--;
14   }
15 }
```



`count = 2`

`enqueue("b.txt")` → [1], in = 2, count = 2.

Step 3 / 11

VARIABLES

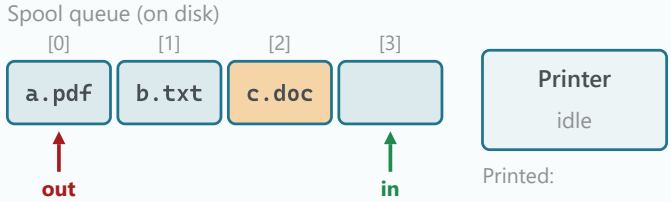
`in = 2` `out = 0` `count = 2`

CALL STACK

`enqueue("b.txt")`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
▶    queue[in] = file;
▶    in = (in + 1) % 4;
▶    count++;
7  }
8  void spooler(void) {
9    while (count > 0) {
10     send_to_printer(queue[out]);
11     queue[out] = NULL;   /* free slot */
12     out = (out + 1) % 4;
13     count--;
14   }
15 }
```



`count = 3`

`enqueue("c.doc")` → [2], in = 3, count = 3.

Step 4 / 11

VARIABLES

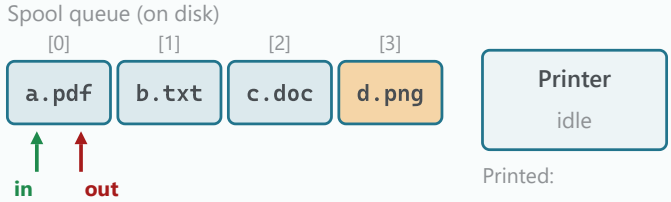
`in = 3` `out = 0` `count = 3`

CALL STACK

`enqueue("c.doc")`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
▶    queue[in] = file;
▶    in = (in + 1) % 4;
▶    count++;
7  }
8  void spooler(void) {
9    while (count > 0) {
10     send_to_printer(queue[out]);
11     queue[out] = NULL;   /* free slot */
12     out = (out + 1) % 4;
13     count--;
14   }
15 }
```



`count = 4`

`enqueue("d.png")` → [3]; in = (3 + 1) % 4 = **0**: it **wraps around** to the start. The queue is full: count = 4, in = out.

Step 5 / 11

VARIABLES

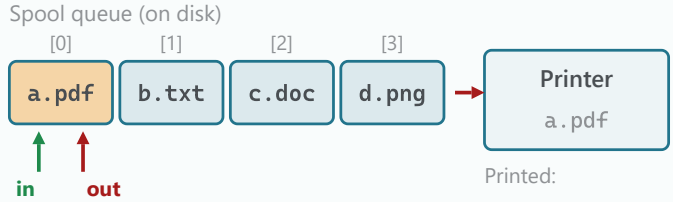
`in = 0` `out = 0` `count = 4`

CALL STACK

`enqueue("d.png")`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
4    queue[in] = file;
5    in = (in + 1) % 4;
6    count++;
7  }
8  void spooler(void) {
▶    while (count > 0) {
▶      send_to_printer(queue[out]);
11     queue[out] = NULL;   /* free slot */
12     out = (out + 1) % 4;
13     count--;
14   }
15 }
```



`count = 4`

The spooler runs: count = 4 > 0. The **oldest** job, queue[out] = a.pdf, goes to the printer (FCFS).

Step 6 / 11

VARIABLES

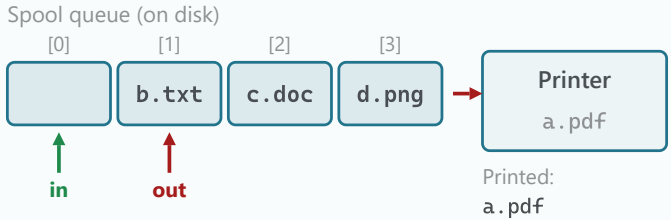
`in = 0` `out = 0` `count = 4`

CALL STACK

`spooler()`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
4    queue[in] = file;
5    in = (in + 1) % 4;
6    count++;
7  }
8  void spooler(void) {
9    while (count > 0) {
10     send_to_printer(queue[out]);
▶      queue[out] = NULL;   /* free slot */
▶      out = (out + 1) % 4;
▶      count--;
14   }
15 }
```



`count = 3`

The slot is freed (queue[0] = NULL): out = 1, count = 3. The printer is slow; meanwhile the programs do other work.

Step 7 / 11

VARIABLES

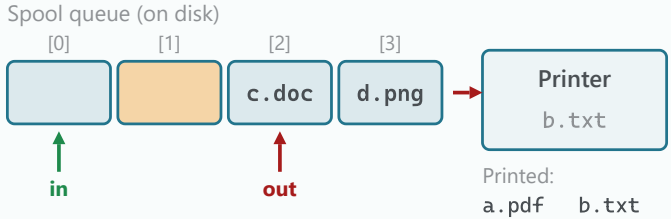
`in = 0` `out = 1` `count = 3`

CALL STACK

`spooler()`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
4    queue[in] = file;
5    in = (in + 1) % 4;
6    count++;
7  }
8  void spooler(void) {
▶    while (count > 0) {
▶      send_to_printer(queue[out]);
▶      queue[out] = NULL;   /* free slot */
▶      out = (out + 1) % 4;
▶      count--;
14   }
15 }
```



`count = 2`

count = 3 > 0: **b.txt** is sent; out = 2, count = 2.

Step 8 / 11

VARIABLES

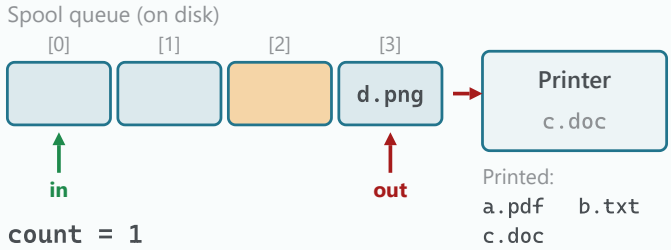
`in = 0` `out = 2` `count = 2`

CALL STACK

`spooler()`

Code Trace: The Spool Queue

```
 1 char *queue[4];
 2 int in = 0, out = 0, count = 0;
 3 void enqueue(char *file) {   /* programs */
 4   queue[in] = file;
 5   in = (in + 1) % 4;
 6   count++;
 7 }
 8 void spooler(void) {
▶  while (count > 0) {
▶    send_to_printer(queue[out]);
▶    queue[out] = NULL;   /* free slot */
▶    out = (out + 1) % 4;
▶    count--;
14   }
15 }
```



count = 2 > 0: **c.doc** is sent; out = 3, count = 1.

Step 9 / 11

VARIABLES

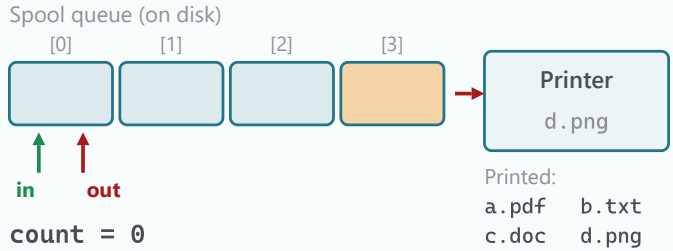
`in = 0` `out = 3` `count = 1`

CALL STACK

`spooler()`

Code Trace: The Spool Queue

```
 1 char *queue[4];
 2 int in = 0, out = 0, count = 0;
 3 void enqueue(char *file) {   /* programs */
 4   queue[in] = file;
 5   in = (in + 1) % 4;
 6   count++;
 7 }
 8 void spooler(void) {
 ▶   while (count > 0) {
 ▶     send_to_printer(queue[out]);
 ▶     queue[out] = NULL;   /* free slot */
 ▶     out = (out + 1) % 4;
 ▶     count--;
14   }
15 }
```



count = 1 > 0: **d.png** is sent; out = 0, count = 0.

Step 10 / 11

VARIABLES

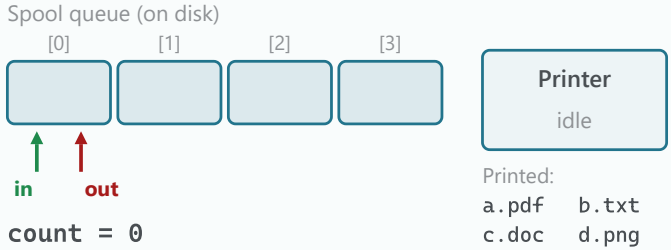
`in = 0` `out = 0` `count = 0`

CALL STACK

`spooler()`

Code Trace: The Spool Queue

```
1  char *queue[4];
2  int in = 0, out = 0, count = 0;
3  void enqueue(char *file) {   /* programs */
4    queue[in] = file;
5    in = (in + 1) % 4;
6    count++;
7  }
8  void spooler(void) {
▶    while (count > 0) {
10     send_to_printer(queue[out]);
11     queue[out] = NULL;   /* free slot */
12     out = (out + 1) % 4;
13     count--;
14   }
15 }
```



`count = 0`

out has wrapped to 0 as well; count = 0 → the loop ends. Since only the spooler touched the printer, the output was **not mixed**. Concurrent `count++` / `count--` is a race, so real queues are protected by a lock.

Step 11 / 11

VARIABLES

`in = 0` `out = 0` `count = 0`

CALL STACK

`spooler()`

Measuring Resource Usage

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

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

The numbers come from the kernel: it updates per-process counters on every context switch, system call and page fault.

Fairness and Efficiency

More resources (many cores, large memory) bring performance and responsiveness, but make sharing them **fairly and efficiently** harder.

Scheduling
Which process runs when, and on which core? Linux 6.6 (2023) replaced CFS with the **EEVDF** scheduler.

Prioritization
`nice` values, Windows priority classes, real-time classes.

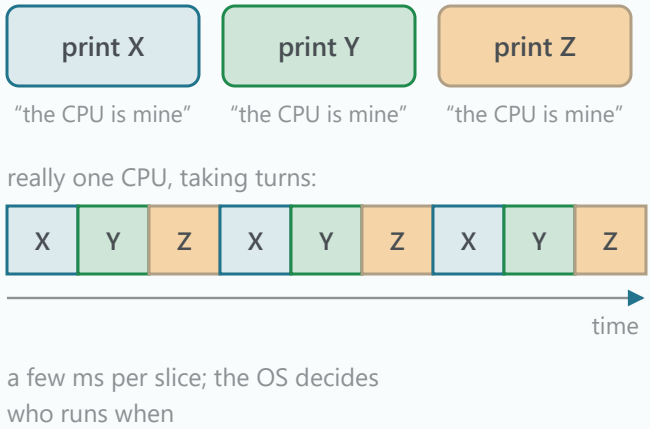
Quotas and limits
Linux **cgroups v2** (containers), `ulimit`, disk quotas, Windows Job objects.

Trade-off
Throughput ↔ response time ↔ fairness: improving one can hurt another.

The right approach depends on the **workload**: throughput on a server, response time on a desktop, meeting deadlines in a real-time system.

Three Big Ideas: Four Small Experiments

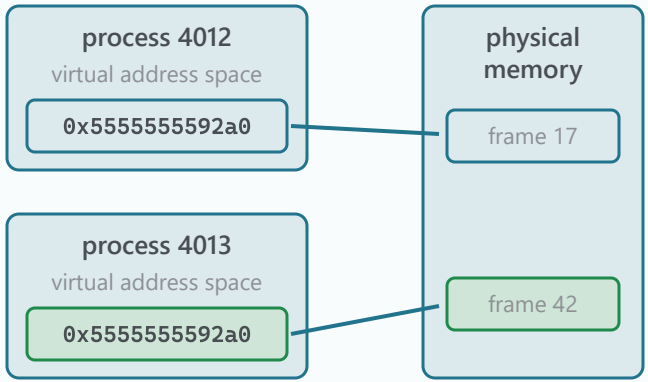
```
$ ./print X & ./print Y & ./print Z &
[1] 3101 [2] 3102 [3] 3103
X
Y
Z
Z
X
Y
…
```



Virtualizing the CPU: on one core, three programs seem to run at once. Through time sharing the OS gives each the illusion of its own CPU; who runs next is **policy** (scheduling), how the switch happens is **mechanism** (context switch).

Three Big Ideas: Four Small Experiments

```
$ setarch -R ./counter & setarch -R ./counter &
(4012) p = 0x5555555592a0
(4013) p = 0x5555555592a0
(4012) *p = 1
(4013) *p = 1
(4013) *p = 2
(4012) *p = 2
…
```

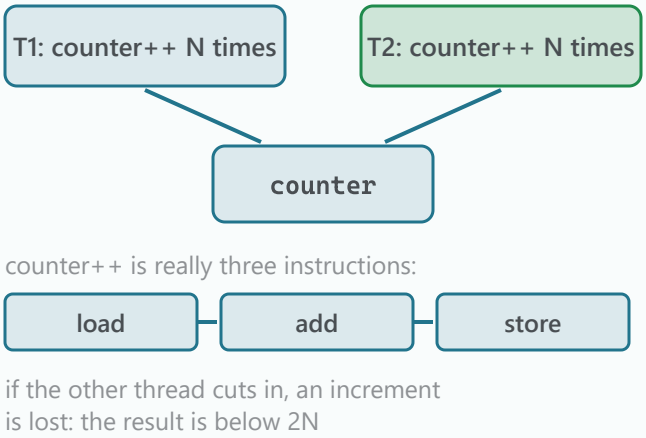


same virtual address, different physical frame

Virtualizing memory: two processes print the same address, yet their values grow independently. Each process sees its own address space; the OS maps them to different physical frames. ASLR is turned off (`setarch -R`) to see the same address; otherwise addresses change on every run.

Three Big Ideas: Four Small Experiments

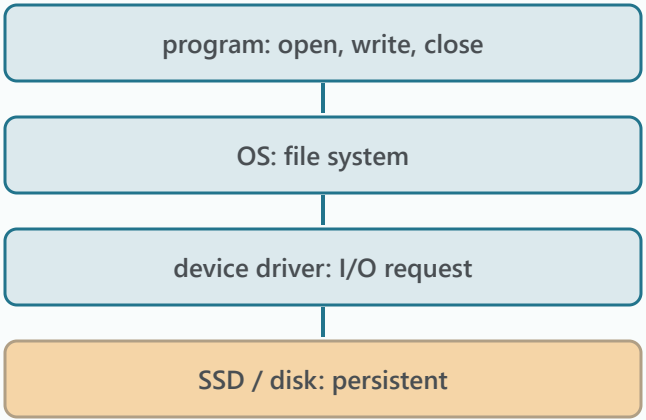
```
$ ./incr 1000
result: 2000
$ ./incr 1000000
result: 1318442
$ ./incr 1000000
result: 1407715
(expected: 2000000)
```



Concurrency: two threads each increment a shared counter N times. For small N the result looks right; for large N it is below 2N and differs on every run, because `counter++` is not atomic (sample output).

Three Big Ideas: Four Small Experiments

```
$ ./save
$ cat note.txt
hello disk
$ sudo reboot
… (the system shuts down and boots again)
$ cat note.txt
hello disk
```

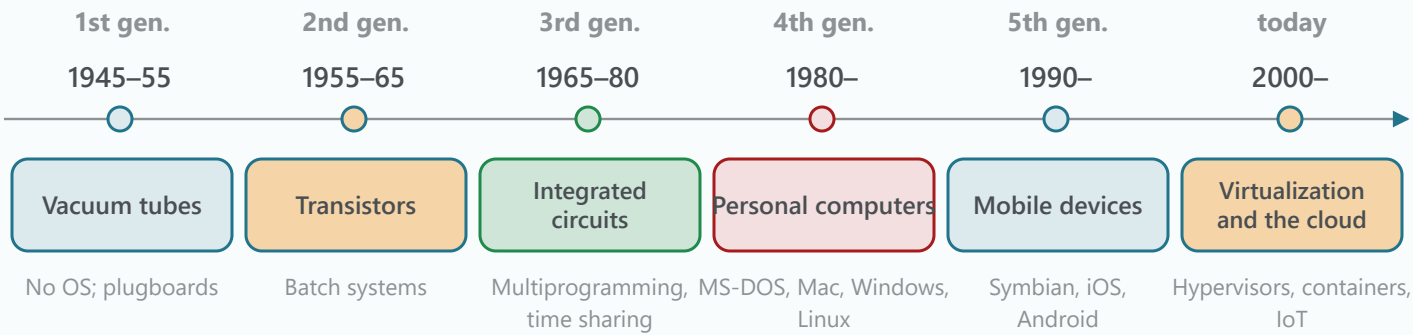


block choice, metadata, ordering: all in the OS

Persistence: DRAM is volatile; to keep data across a reboot we write it to disk through a file system. The program only calls `open`, `write`, `close`; choosing blocks, metadata and talking to the device are the OS's job.

HISTORY

A History of Operating Systems



- Each generation follows a leap in hardware technology (tube → transistor → integrated circuit → microprocessor).
- Each generation asks the same question again: **how do we make better use of the expensive resource (CPU, memory, human time)?**
- The last row is not one of Tanenbaum's generations; it is added to help explain today's systems.

HISTORY

The First and Second Generations

1st generation (1945–55): vacuum tubes

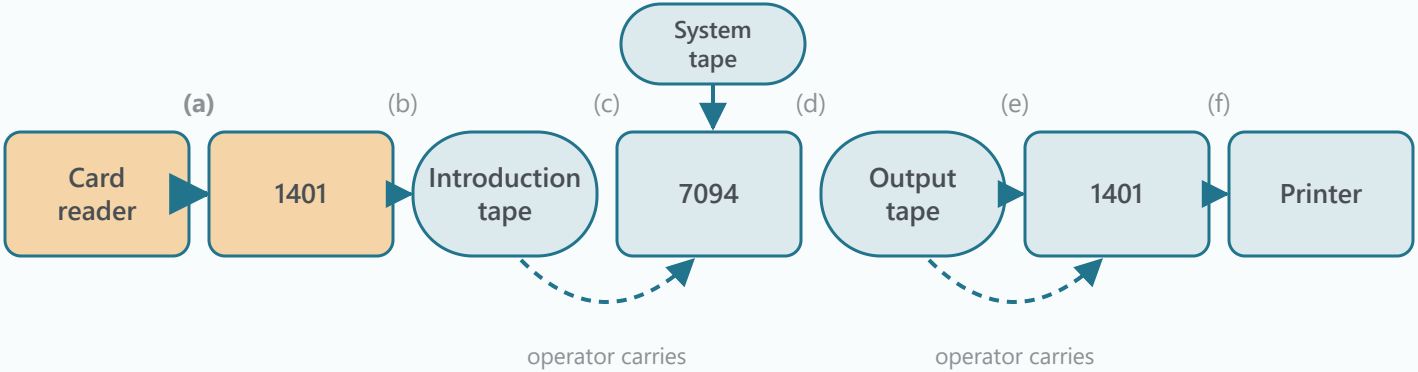
- Room-sized, slow, failure-prone machines (ENIAC, 1945).
- The same group of engineers designed, built, programmed and maintained them.
- Programming in machine language or with **plugboards**; **punched cards** in the early 1950s.
- **No programming languages and no operating system**; used mostly for numerical tables.

2nd generation (1955–65): transistors and batch systems

- Transistors replaced tubes: smaller, more reliable, more energy-efficient.
- Mainframes in air-conditioned rooms; an **operator** between programmer and machine.
- **Batch processing**: jobs are collected on tape and a small monitor program runs them **one after another, automatically**; the operator only carries tapes.
- One job in memory: the CPU idles during I/O; no interaction.

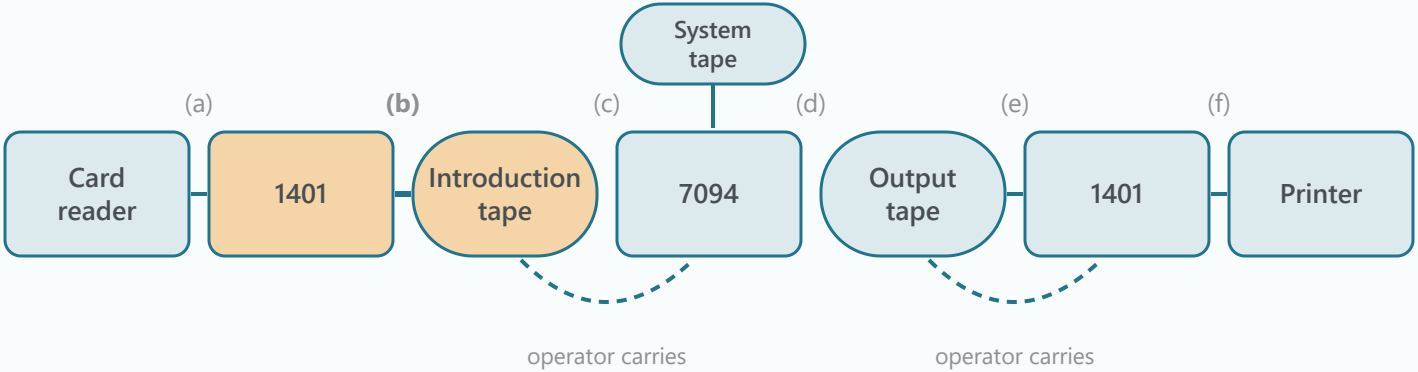
Batch processing aimed to eliminate the **manual changeover** time between jobs. It was unsuitable for interactive and real-time use.

An Early Batch System



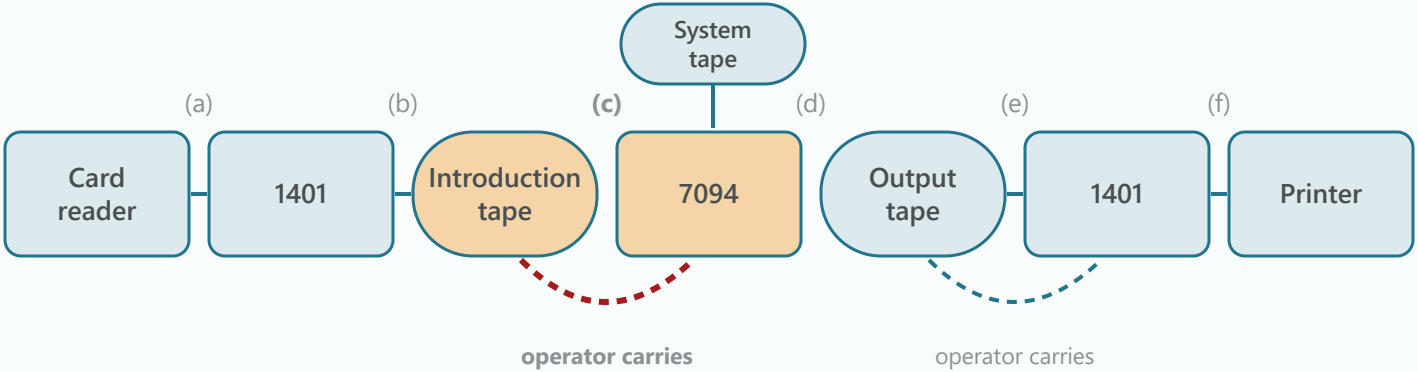
(a) Programmers bring their card decks to the 1401's card reader.

An Early Batch System



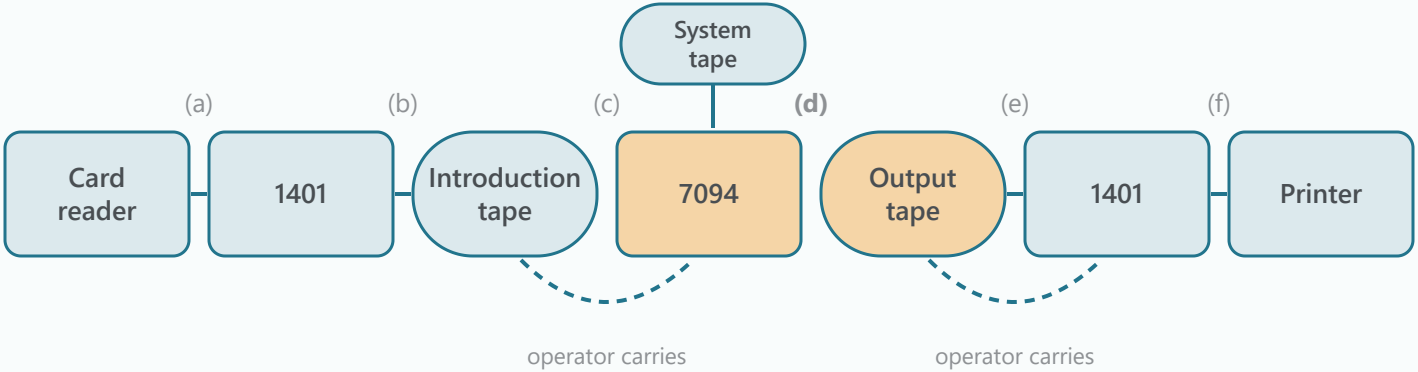
(b) The cheap 1401 reads a batch of jobs and writes them to the **input tape** (keeping slow card reading off the expensive machine).

An Early Batch System



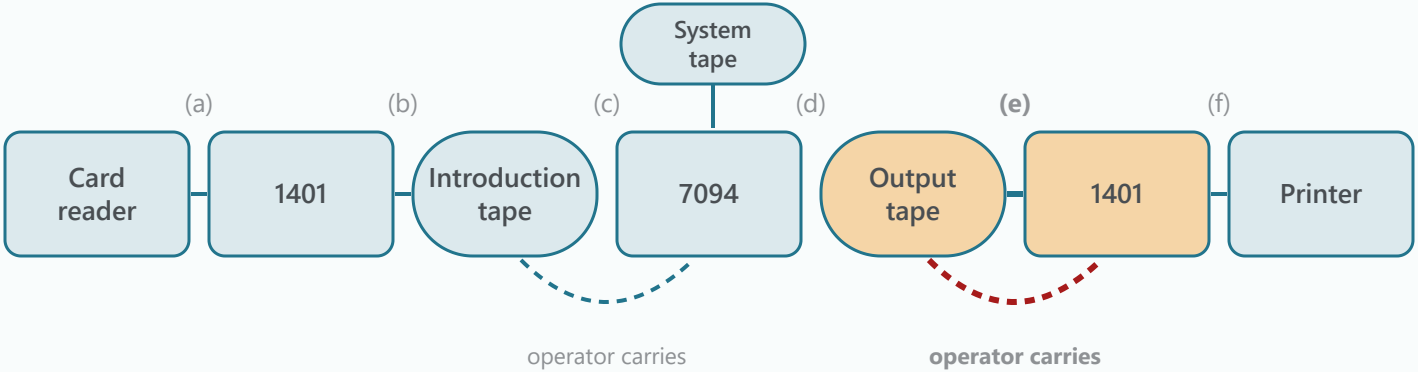
(c) The operator carries the input tape to the expensive **7094**.

An Early Batch System



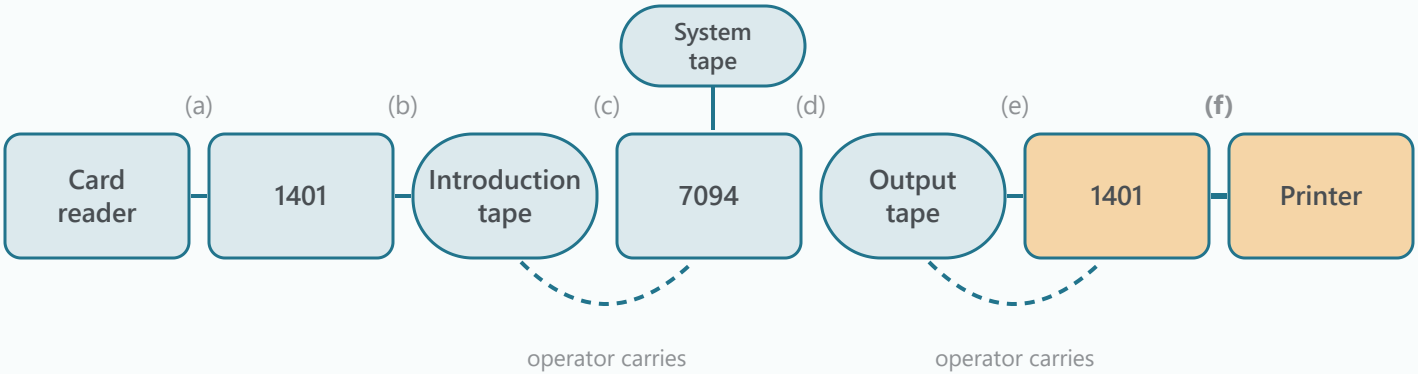
(d) The 7094 runs the jobs **one by one** under the monitor on the system tape and writes the output to the output tape.

An Early Batch System



(e) The operator carries the output tape back to the 1401.

An Early Batch System



(f) The 1401 prints the output (offline printing). The expensive 7094 never had to read cards or print.

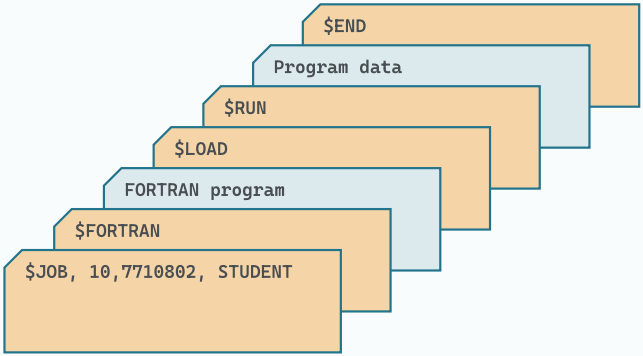
HISTORY

IBM 1401, 7094 and System/360

Machine	Year	Features
IBM 1401	1959	Business data processing; character-oriented and relatively cheap. Also used for I/O work such as reading cards, copying to tape and printing. One of the most common computers of the 1960s.
IBM 7094	1962	Scientific computing; 36-bit words, an architecture **entirely different from the 1401**. Ran batch jobs under IBSYS / FMS. Time sharing was tried on a modified 7094 at MIT with **CTSS**.
System/360	1964	A family sharing **one architecture** from small to large models. Its OS, **OS/360**, had multiprogramming and spooling; a huge, bug-ridden project of millions of lines (Brooks, *The Mythical Man-Month*).

The 1401 and 7094 did not run under OS/360, which was written for the System/360 family announced in 1964. Nor was the 7094 a bigger 1401; it was a separate machine that complemented it.

A Typical FMS Job

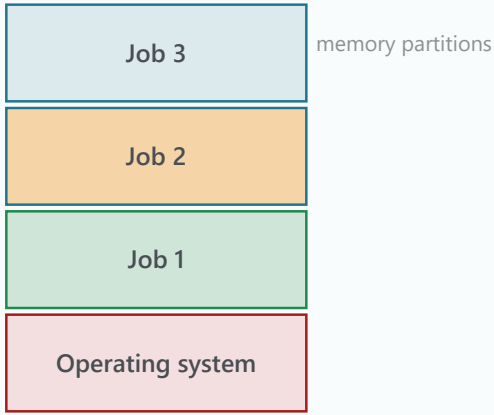


- **FMS = Fortran Monitor System**: a batch monitor program for the IBM 709/7094.
- **Control cards** starting with `$` tell the monitor what to do: start the job, load the compiler, link the program, run it, finish.
- These cards are the ancestors of today's **shell scripts** and IBM JCL.

Do not confuse FMS here with the "Flexible Manufacturing System" of factory automation; the two are unrelated.

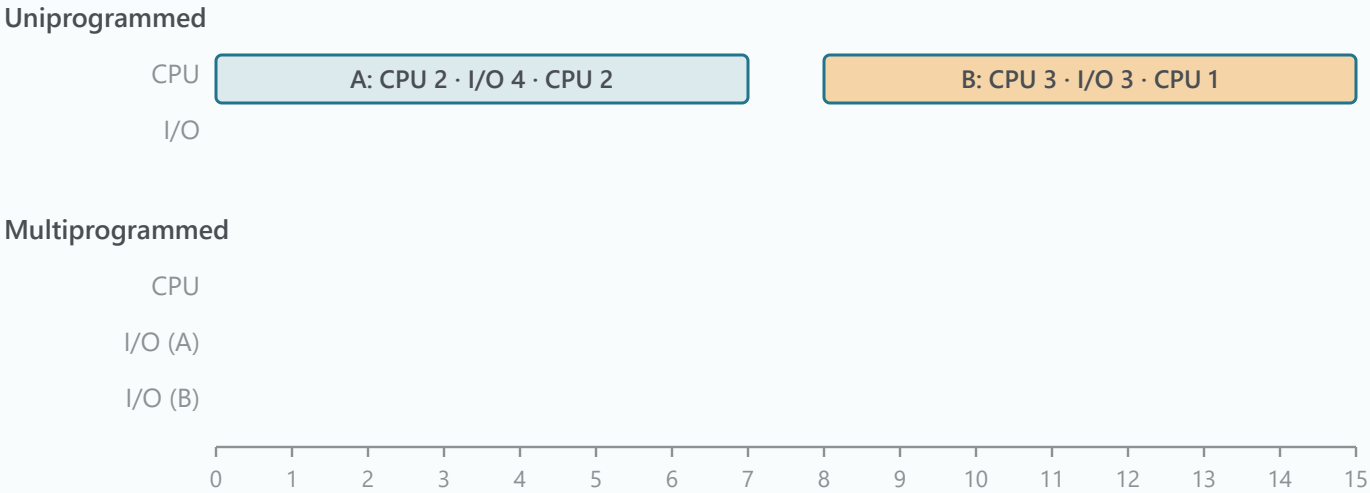
Third Generation: ICs and Multiprogramming

- **Integrated circuits** (1965–80): compatible families such as System/360; smaller, cheaper, more reliable.
- **Multiprogramming**: memory is split into partitions, one job per partition. When a job waits for I/O, the CPU **switches to another job**.
- On one CPU the jobs **interleave**: not true parallelism but **concurrency**.
- **Protection hardware** is needed so jobs cannot touch each other's memory.
- **Spooling**: cards go straight to disk; when a job finishes, the next one is loaded from disk.
- **Time sharing**: CTSS (MIT, 1961) → MULTICS → **UNIX** (Bell Labs, 1969) → BSD, POSIX, MINIX, Linux (1991).



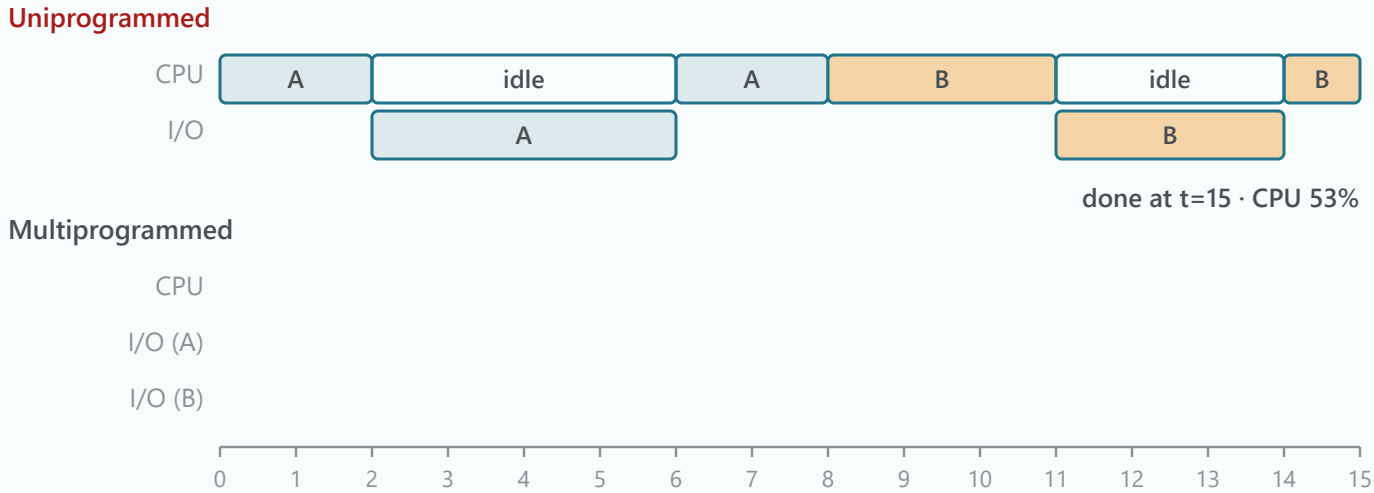
Three jobs in memory

Multiprogramming: Why the CPU Stays Busy



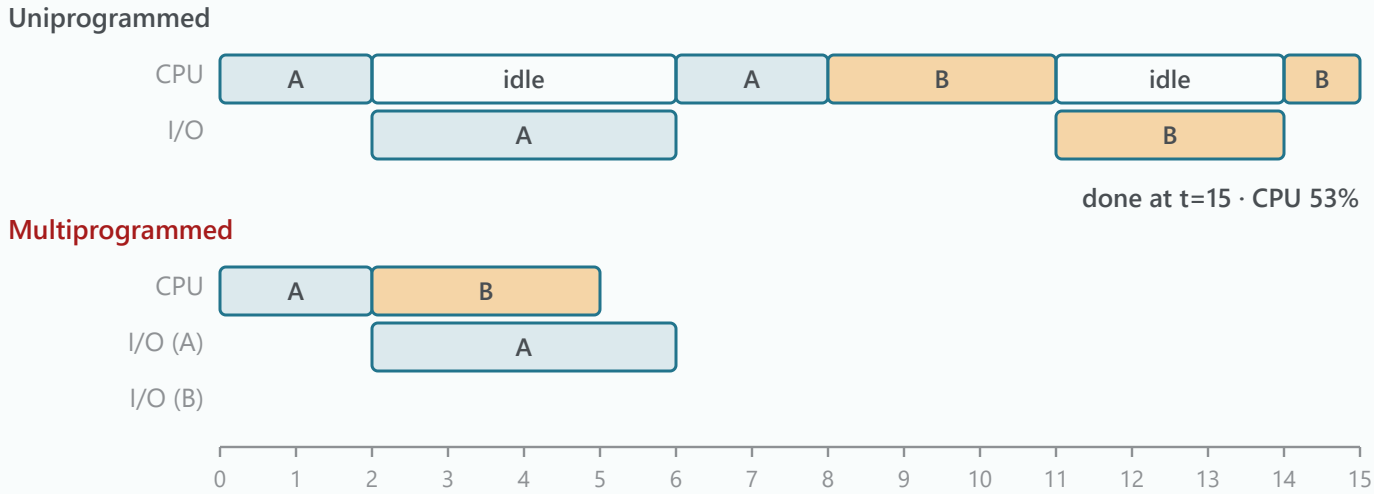
Two jobs (time units): **A** = CPU 2, I/O 4, CPU 2 · **B** = CPU 3, I/O 3, CPU 1. Total CPU demand: 8 units.

Multiprogramming: Why the CPU Stays Busy



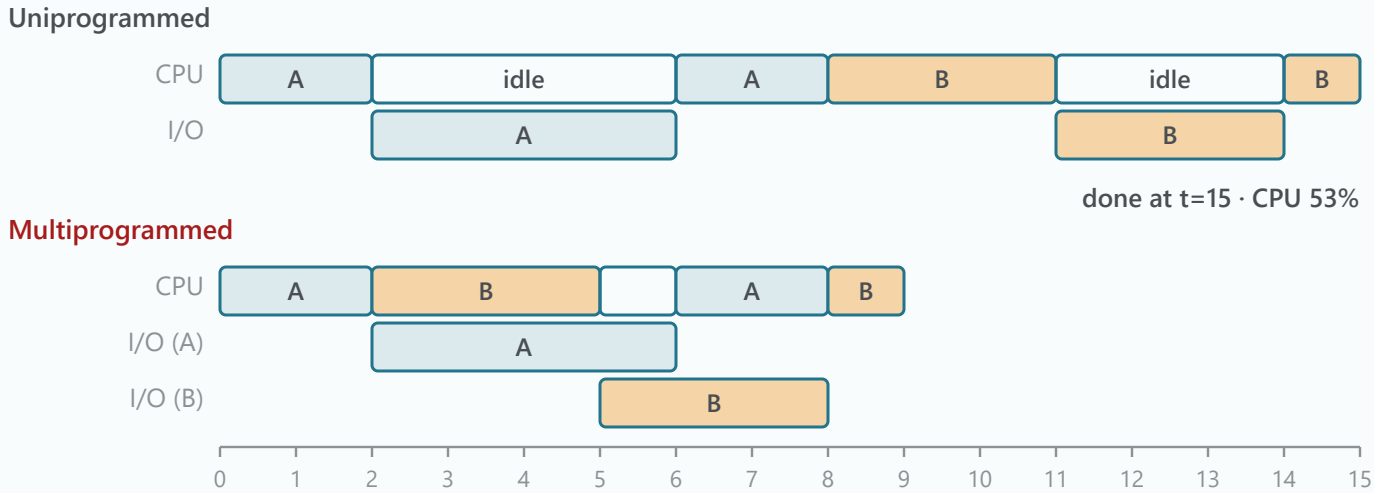
Uniprogrammed: first A (t = 0–8), then B (t = 8–15). The CPU is **idle** during I/O: utilization 8 / 15 ≈ **53%**.

Multiprogramming: Why the CPU Stays Busy



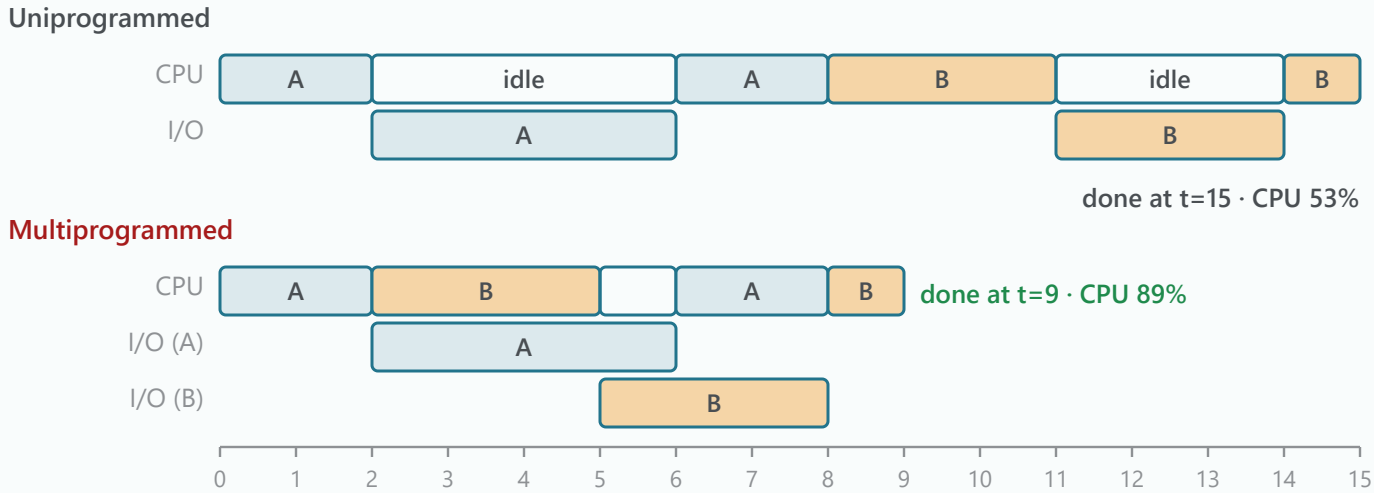
Multiprogrammed: when A starts I/O at t = 2, the CPU switches to **B** at once (t = 2–5). A's I/O proceeds in the background.

Multiprogramming: Why the CPU Stays Busy



B starts I/O at t = 5 (5–8). A's I/O ends at t = 6: the CPU idles only from 5 to 6; A runs 6–8, B runs 8–9.

Multiprogramming: Why the CPU Stays Busy



The same two jobs finish in **9 units instead of 15**; CPU utilization is 8 / 9 ≈ **89%**. The gain comes from overlapping CPU and I/O.

Fourth Generation: Personal Computers

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

Rapidly falling prices put computers in everyone's hands; the OS's goal shifted from "use the machine efficiently" to "make life easy for the user".

Fifth Generation: Mobile and Today

Mobile devices (1990–)

- 1990s: PDAs, Palm OS; 2000s: Symbian, BlackBerry OS.
- 2007 **iOS** (iPhone), 2008 **Android** (on the Linux kernel).
- Windows Phone launched in 2010; development stopped in 2017.
- Today Android and iOS share the market; touch interfaces, power management and permission models stand out.

Virtualization and the cloud

Hypervisors (KVM, Hyper-V, Xen); many virtual machines on one server.

Containers

Docker, Kubernetes: isolated environments sharing one kernel (namespaces, cgroups).

Heterogeneous hardware

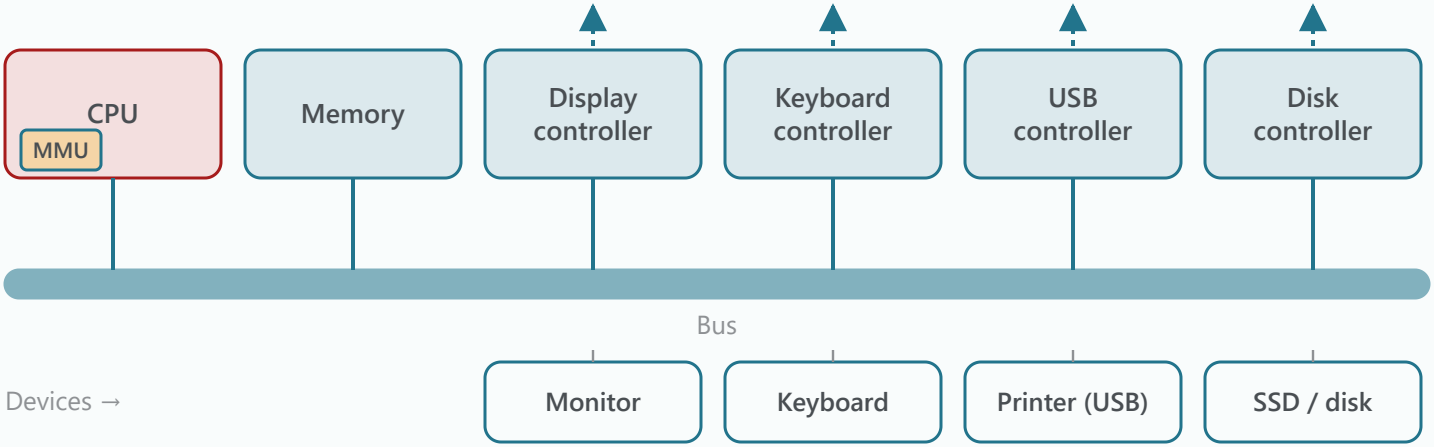
CPU + GPU + NPU; hybrid processors with big and little cores.

Security

Memory-safe languages: Rust support in the kernel since Linux 6.1.

HARDWARE

Components of a Personal Computer

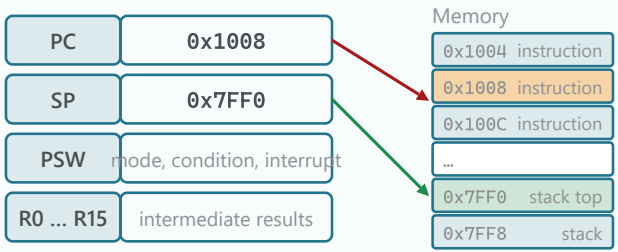


- The CPU, memory and I/O devices communicate over a **bus**; each device has a **controller**.
- The **MMU** (memory-management unit) sits inside the CPU and translates virtual addresses to physical ones.
- Real systems use point-to-point links (PCIe) instead of one shared bus, but the conceptual model is the same.

Processor and Registers

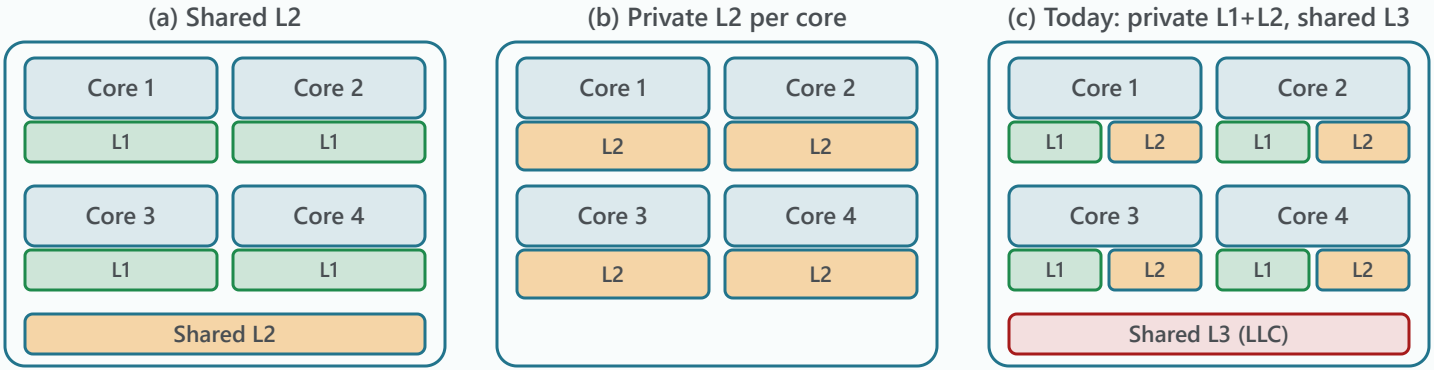
- The computer's "brain": the processor (CPU) executes every instruction.
- The basic cycle: **fetch**, **decode**, **execute**.
- Memory is slow, so variables and intermediate results are kept in **registers**: *load* (memory → register), *store* (register → memory).
- Each processor family has its own **instruction set**: an x86-64 program does not run directly on ARM.
- Performance: clock rate, core count, cache size, pipeline depth, memory bandwidth.
- On a context switch (changing processes) the OS **saves and restores** all registers.

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



HARDWARE

Multicore Chips



Multicore
Several independent CPUs on one chip; the OS sees and schedules each as a separate processor.

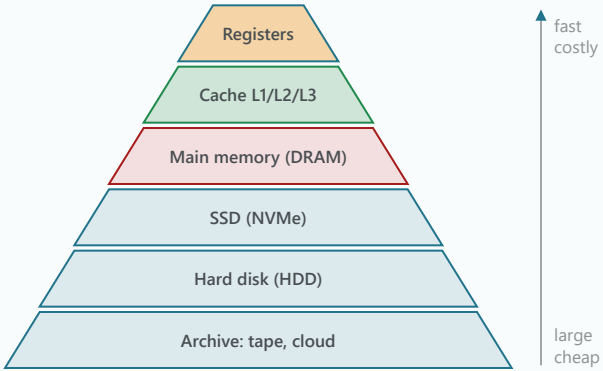
SMT / Hyper-Threading
One core runs two hardware threads; the OS sees two "logical CPUs", but they share resources.

Hybrid cores
Performance (P) and efficiency (E) cores (ARM big.LITTLE, Intel 2021+): the scheduler must know which to pick.

For the OS this means problems of parallelism, synchronization and cache coherence.

HARDWARE

Memory Hierarchy



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

DRAM and above are **volatile** (lost on power-off); SSDs and HDDs are persistent. Figures are rough 2026 magnitudes. Each level acts as a cache for the one below: thanks to **locality**, most accesses are served by the fast upper levels.

Three I/O Methods

Every I/O device has a **controller**; the kernel's **device driver** writes commands to the controller's registers and reads its status. There are three ways to learn that the work is done:

Method	Key difference
Polling	The CPU reads the status register in a loop until the device is ready; it can do nothing else 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 steps in only at the start (programming it) and the end (one interrupt).

An **interrupt** from a device and a **trap** caused by the executing instruction itself (divide by zero, page fault, system call) enter the kernel the same way: the processor switches to kernel mode and jumps to the handler in the table.

Kinds of Operating Systems (1)

Mainframe

Very high I/O capacity, reliability and availability; batch, transaction processing and time sharing. Banks, airlines. *IBM z/OS, z/VM, Linux on Z, Unisys MCP, Fujitsu BS2000.*

Server

Web, database, file and mail services for many users over a network. Data centers, cloud, virtualization (KVM, Hyper-V), containers. *Linux, Windows Server, FreeBSD.*

Multiprocessor

Several cores/sockets; runs work in parallel and deals with synchronization and NUMA. Today **every general-purpose OS** is in this class. *Linux, Windows, macOS.*

Personal

Single user, a good GUI, multitasking; browsing, office work, games, file management. *Windows 11, macOS, Linux distributions, ChromeOS.*

Kinds of Operating Systems (2)

Mobile (handheld)

Touch interface, battery and thermal management, sensors, app store and permission model. *Android, iOS/iPadOS, HarmonyOS.* (Windows Phone ended in 2017.)

Embedded

Cars, appliances, network gear, aircraft; users usually do not install software, and the system is fixed in flash. *Embedded Linux (Yocto), Zephyr, FreeRTOS, QNX, VxWorks.*

Sensor node

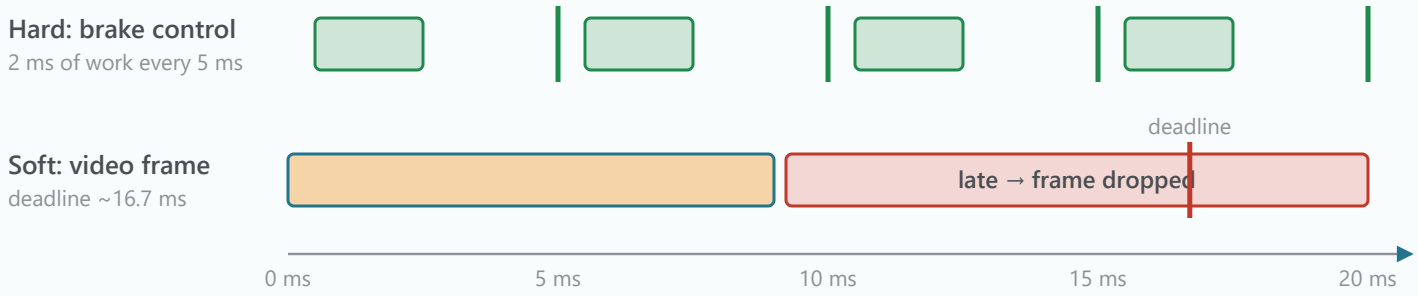
A few tens of KB of RAM, years on a battery, event-driven design, low-power radio (802.15.4, BLE). IoT and M2M networks. *Contiki-NG, RIOT, Zephyr; TinyOS is now historical.*

Smart card

Very limited CPU and memory; authentication, payment, cryptography, secure storage. Bank cards, SIM/eSIM, ID cards. *Java Card, MULTOS.*



Real-Time Operating Systems



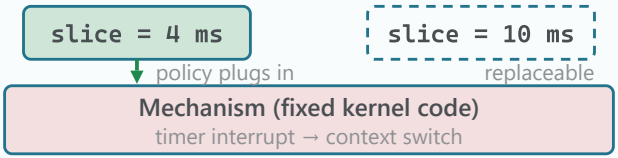
- What matters is **predictability, not speed**: a job must finish before its **deadline**.
- **Hard**: missing a deadline is a disaster (brakes, airbags, flight control, industrial robots).
- **Soft**: an occasional miss is tolerated; quality drops (audio/video streaming, games).
- Examples: **VxWorks** (Mars rovers), **QNX** (in-car systems), FreeRTOS, Zephyr.
- Linux's **PREEMPT_RT** patches were merged into mainline in 2024 (6.12); the old *RTLinux* project has ended.
- Other uses: audio, video, motion and sensor processing, train control, defense.

Policy and Mechanism

- **Policy**: *what* will be done?
- **Mechanism**: *how* will it be done?
- **Separating** the two is a key design principle: when policy decisions change later, the mechanism need not be rewritten → maximum flexibility.

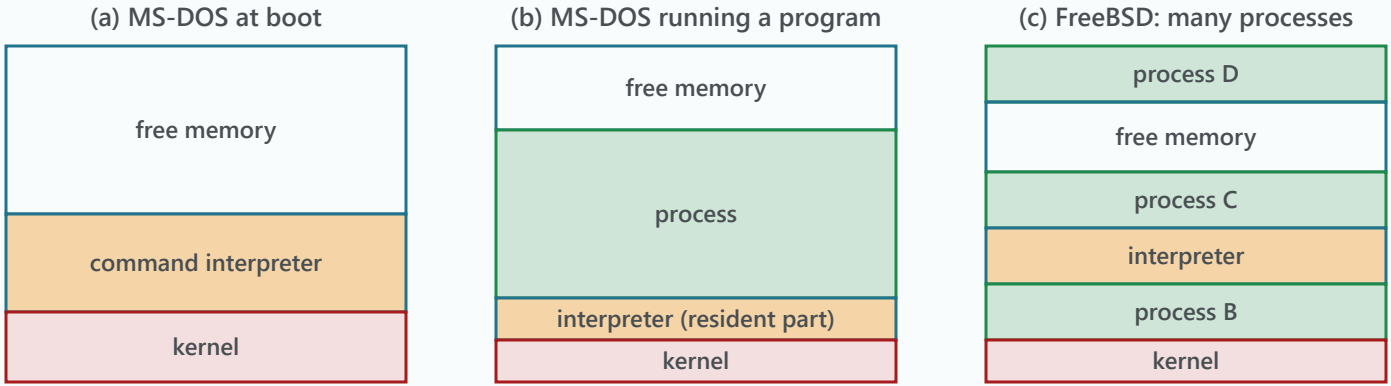
Example: on Linux the scheduling class can be changed with `sched_setscheduler()` and the I/O scheduler through `/sys` at run time; the kernel code stays the same.

Mechanism	Policy
Timer interrupt	How many ms should a time slice be?
Priority scheduler	Which process gets high priority?
File permission bits (rwx)	Who gets which permission?
Page-replacement machinery	Which page gets evicted?
cgroups	How much CPU/memory for which container?



Striking these balances takes creativity as much as engineering.

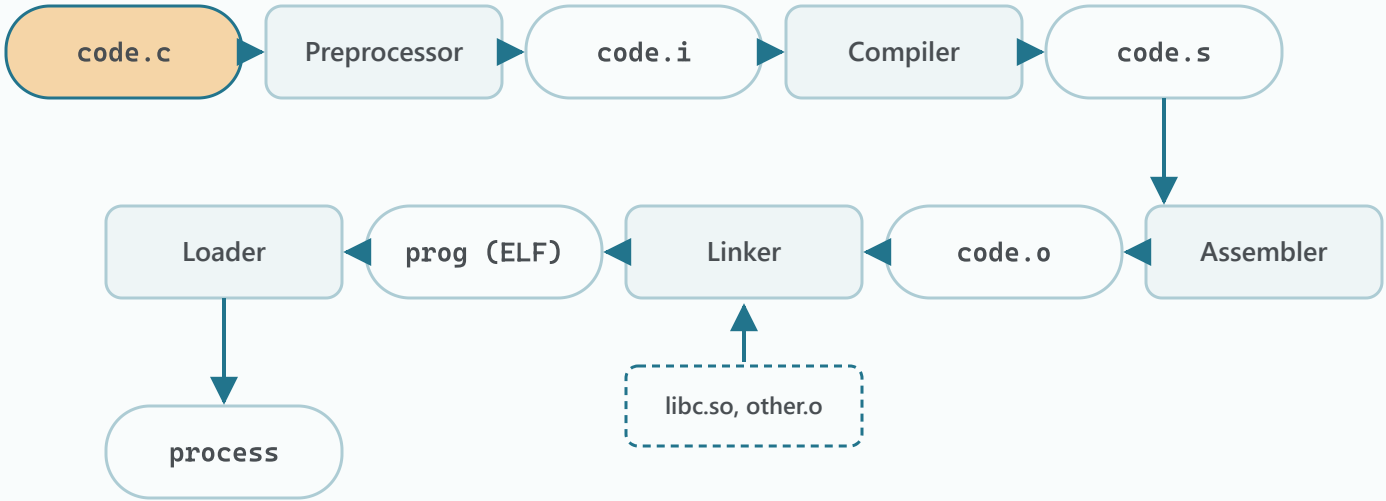
MS-DOS and FreeBSD



MS-DOS: single-tasking, one memory area. The shell (COMMAND.COM) is loaded at boot; a program is loaded **over** its transient part, and when the program ends the shell reloads itself from disk. No multitasking, no protection.

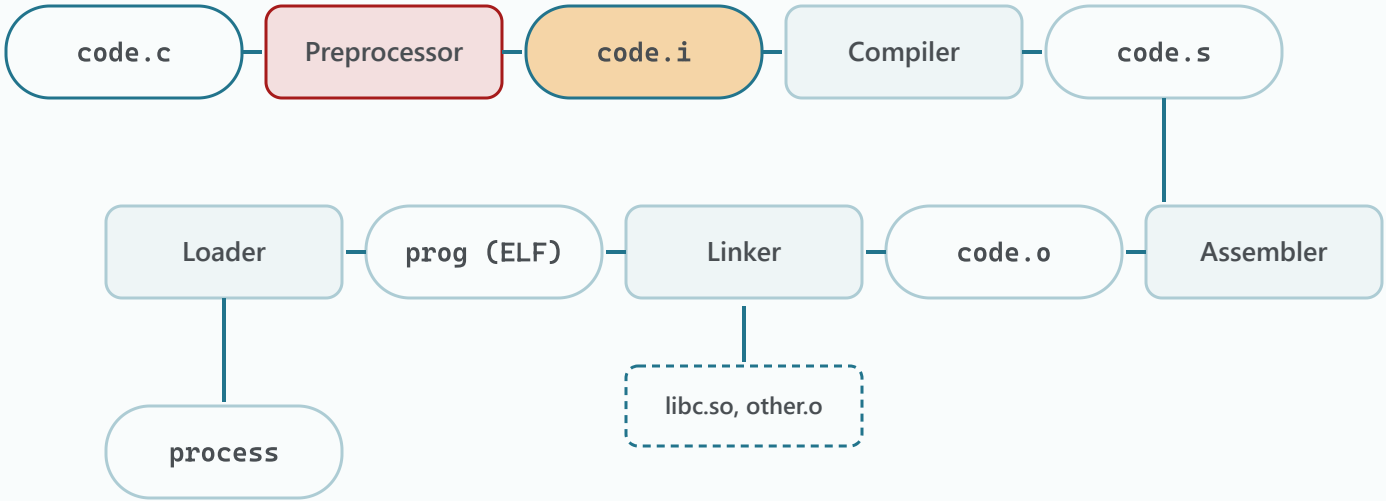
FreeBSD (a UNIX derivative): multitasking. When a user logs in, a shell starts; it creates a new process with `fork()` and loads the program into it with `exec()`, then waits for it or (with `&`) runs it in the background and keeps taking commands.

Building an Executable



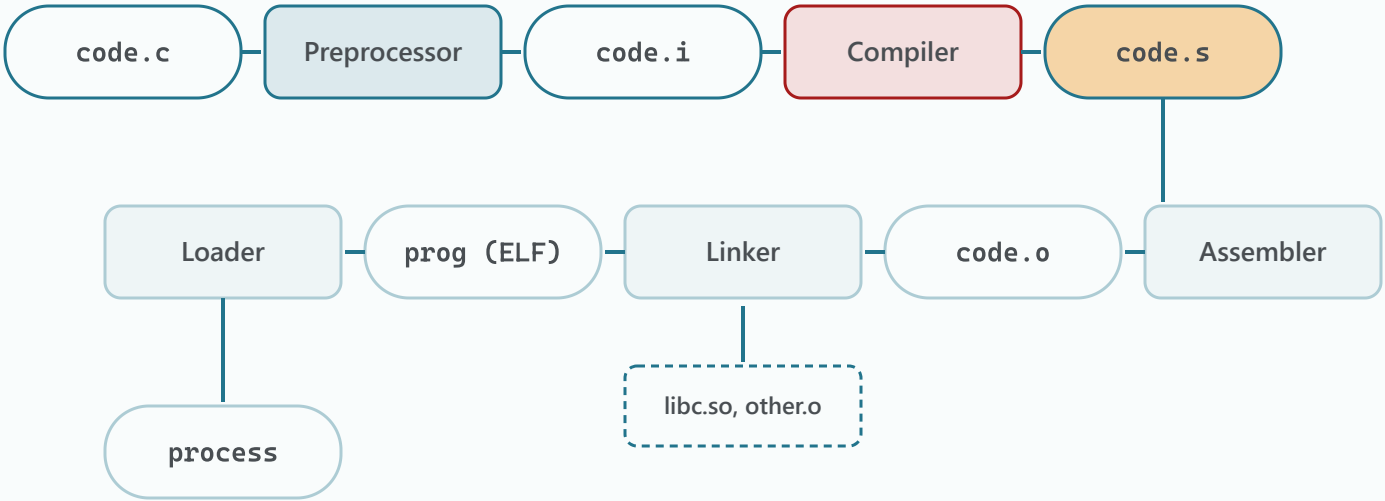
The source file `code.c` : human-readable C code.

Building an Executable



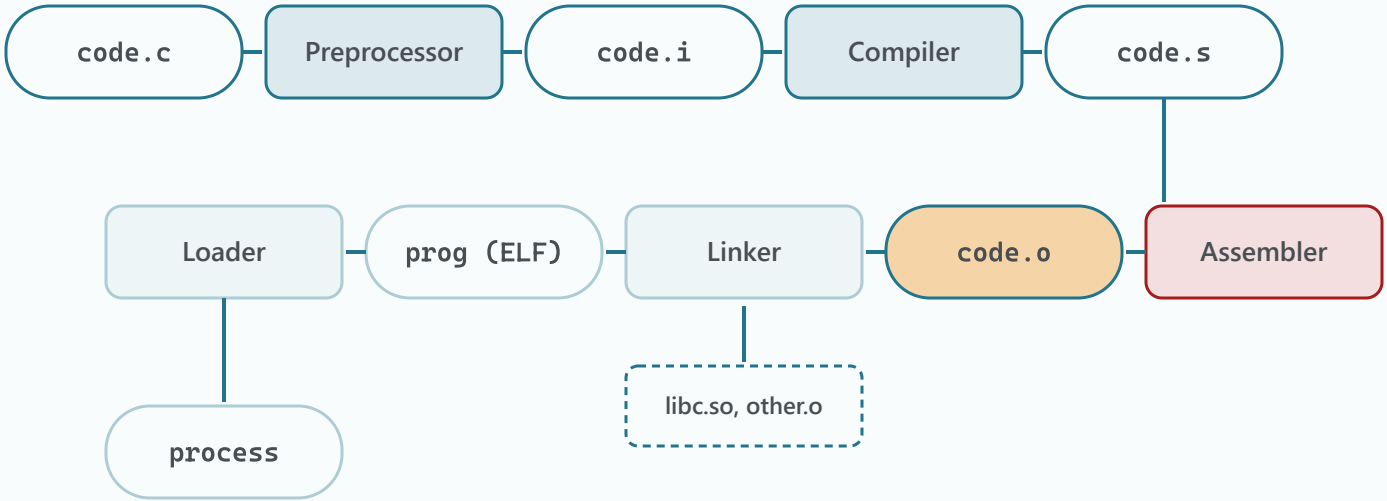
Preprocessor (`gcc -E`): pulls in headers with `#include` , expands macros, resolves conditional compilation with `#ifdef` → `code.i` .

Building an Executable



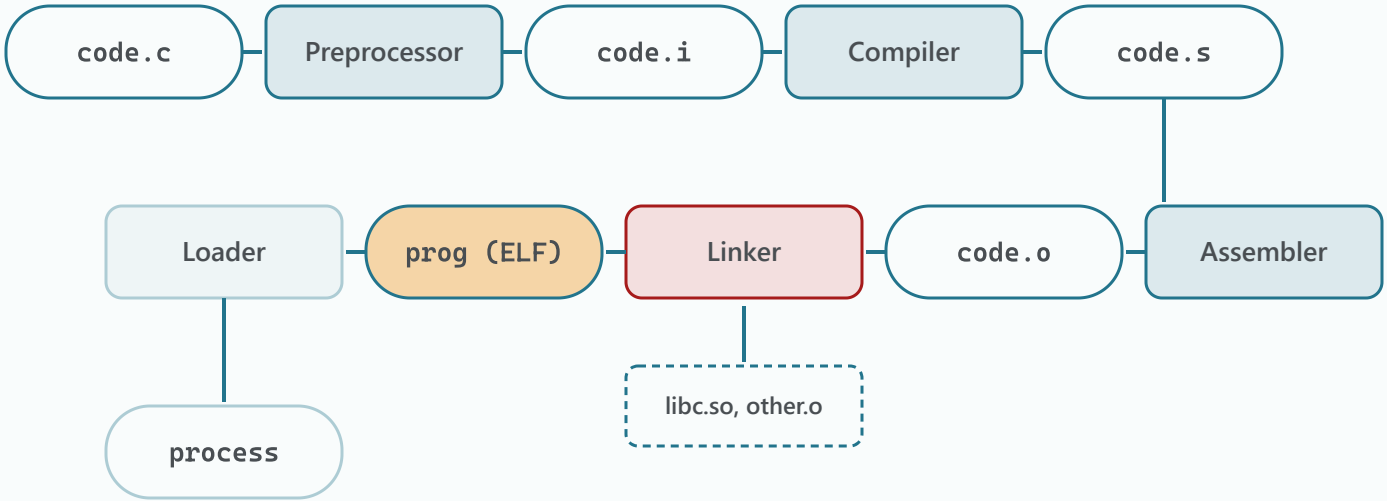
Compiler (`gcc -S`): translates C into the target processor's **assembly** language and optimizes → `code.s` .

Building an Executable



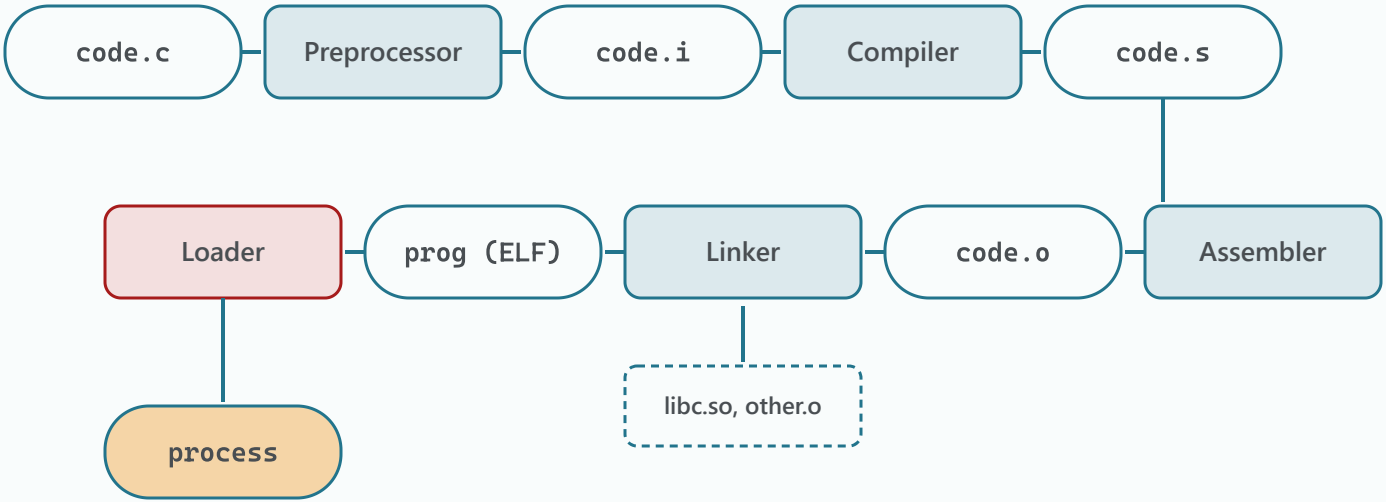
Assembler (`gcc -c`): turns assembly into machine code → the **object file** `code.o` .
Addresses of external functions (printf) are not yet known.

Building an Executable



Linker (`gcc code.o other.o -o prog`): combines object files and libraries and resolves addresses → an **executable** (Linux: ELF, Windows: PE/.exe).

Building an Executable



`./prog` : the shell does fork + exec; the kernel's **loader** maps the ELF into memory, the dynamic linker binds libc.so, and the program runs as a **process**.

Metric Units

Exp.	Name	Symbol	Exp.	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

Binary (IEC)	Value
1 KiB (kibibyte)	2^{10} = 1024 B
1 MiB	2^{20} B
1 GiB	2^{30} B
1 TiB	2^{40} B

Memory is usually measured in binary units (16 GiB RAM), disks and networks in decimal: 1 TB = 10^{12} B ≈ 0.91 TiB; 1 Gb/s = 10^{9} bit/s. Byte **B**, bit **b**.



SUMMARY

Review Questions

1. What happens if a user program executes a privileged instruction?
2. Why does multiprogramming raise CPU utilization? How does it differ from true parallelism?
3. Which of polling, interrupts and DMA suits bringing a 4 KB disk block into memory? Why?
4. Why are the upper levels of the memory hierarchy small and the lower ones large? Why does the hierarchy work?
5. Why must the OS save all registers on a context switch?
6. Distinguish policy from mechanism with an example.

▶ **Show answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014; 5th ed., 2023.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), version 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
- Linux man pages: `syscall(2)`, `fork(2)`, `execve(2)`, `proc(5)`.

The End

Chapter 1: Introduction

Next: Chapter 2: Operating-System Structures