Chapter 2.1: Software and Operating Systems

Computer Fundamentals and Introduction to Programming

Software | Operating system | Resource management | Kinds of systems | Licensing

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/computer/slides/Chapter_02_01_Software_and_Operating_Systems.html

Learning Outcomes

Kinds of software

Tell system software from application software.

Operating system

Explain how the OS sits between the user, applications and hardware.

Resource management

Interpret process, memory, file and device management.

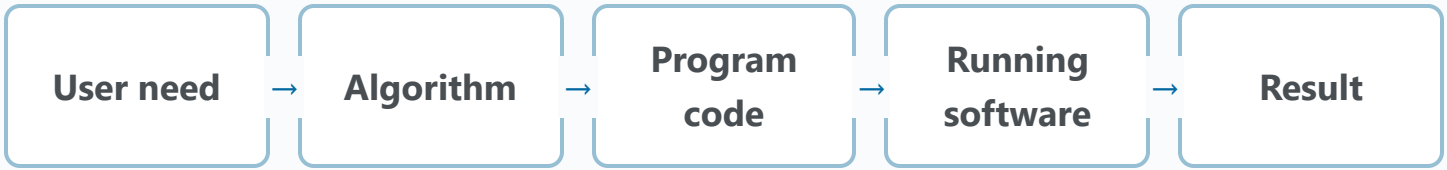
Choosing a system

Compare kinds of operating systems against requirements.

SOFTWARE

What Is Software?

Software is the set of programs, data and documentation that defines what a computer does.



Hardware physically carries out the work; software decides what is done, when, and how.

SOFTWARE

Software Quality Attributes

Functionality
Does the expected tasks correctly.

Usability
Easy to learn and effective to use.

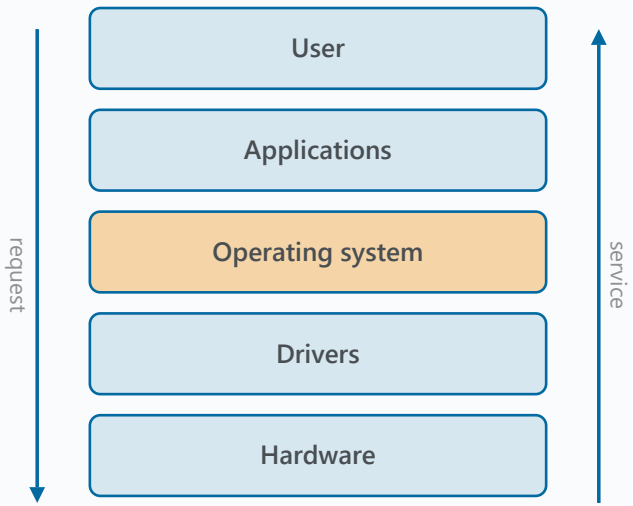
Performance
Uses time and resources efficiently.

Reliability
Behaves predictably when things go wrong.

Security
Resists unauthorized access and changes.

Maintainability
Fixes and improvements stay manageable.

Software and Hardware



- The user only interacts with the top layers.
- Each layer hides the details below it and offers simpler services to the layer above (abstraction).
- Thanks to drivers, the same application runs on different hardware.

Requests travel down; results (services) travel back up.

Kinds of Software

System software

Manages the hardware and provides a platform for other software.

Application software

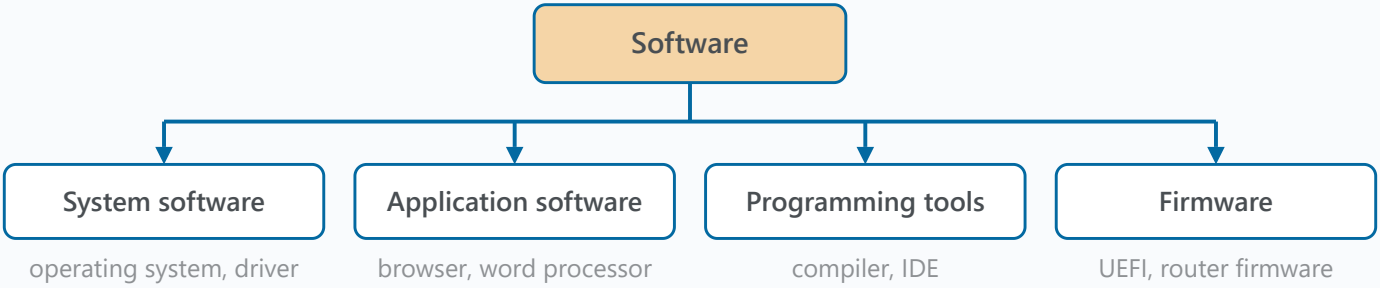
Does specific jobs for the user.

Programming tools

Support writing, translating, testing and debugging code.

Firmware

Embedded software that runs a device's basic functions.

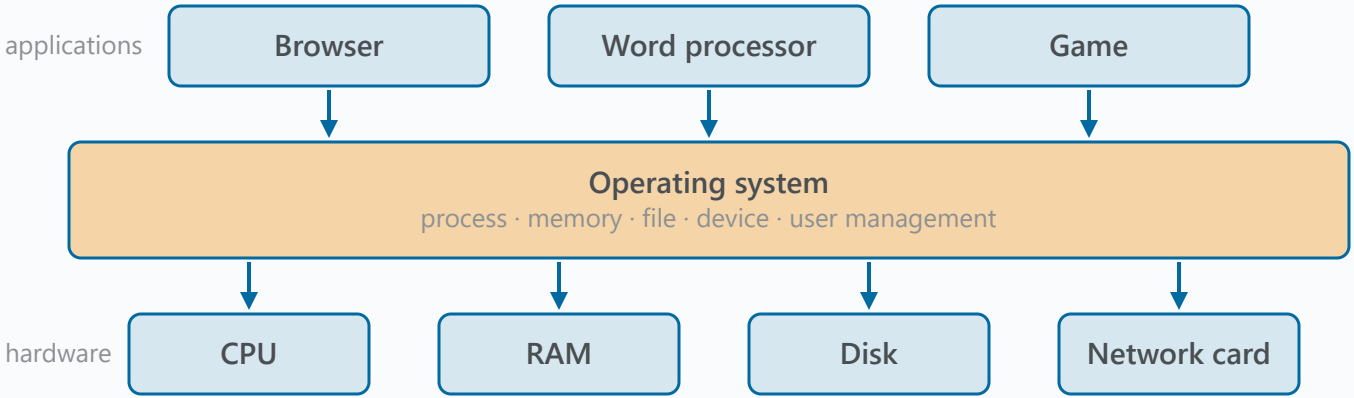


System and Application Software

Type	Examples	Main purpose
System	Operating system, driver, utility	Manage the computer and its resources
Application	Word processor, browser, game	Carry out the user's task
Middleware	Database access layer, message queue	Help applications talk to the system and to each other
Programming tools	Compiler, integrated development environment (IDE), debugger	Build new software

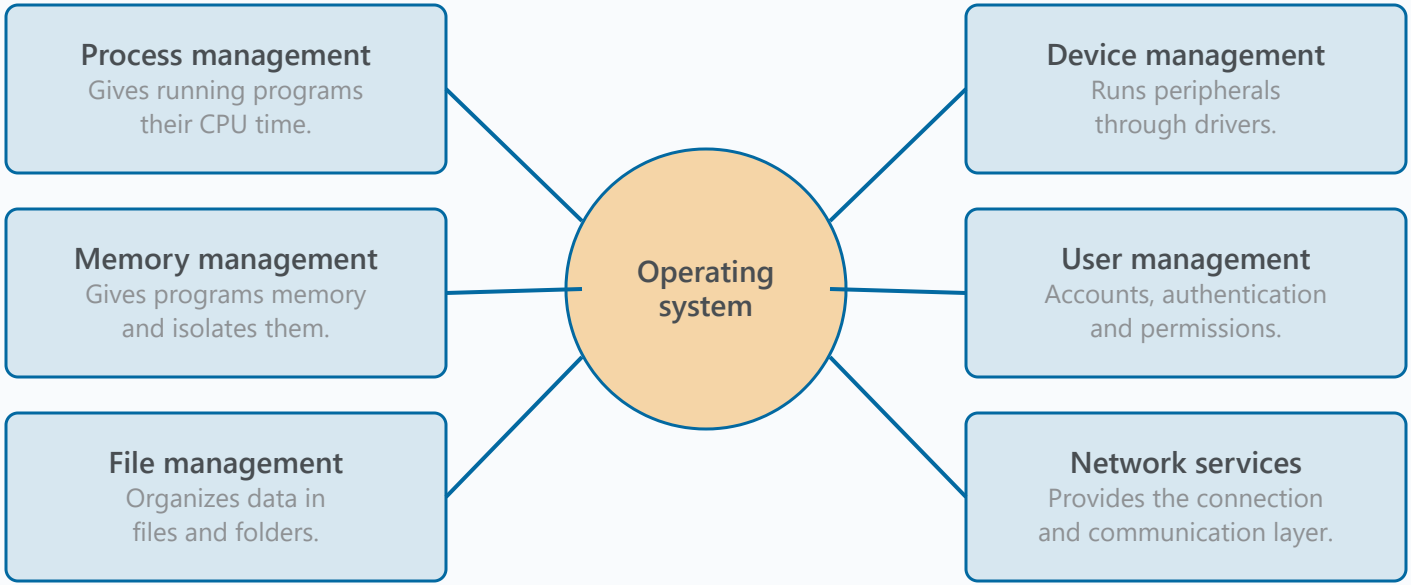
What Is an Operating System?

An operating system is the core system software that manages hardware resources and provides common services to applications.

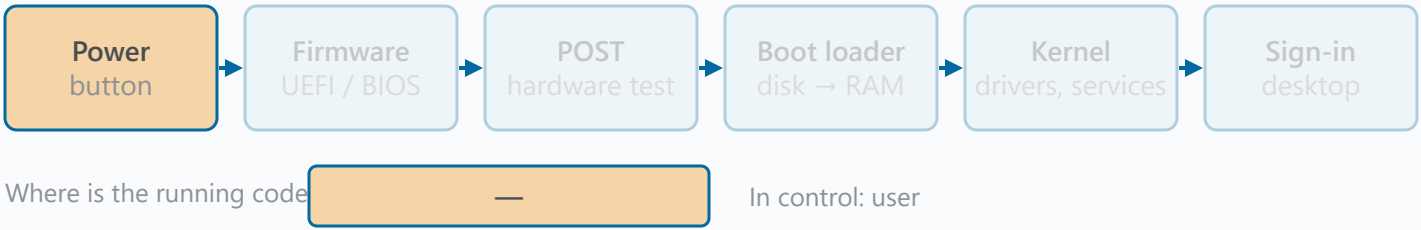


- Hides hardware details from applications.
- Lets several programs share resources safely.
- Gives the user a graphical or command-line interface.

What an Operating System Does

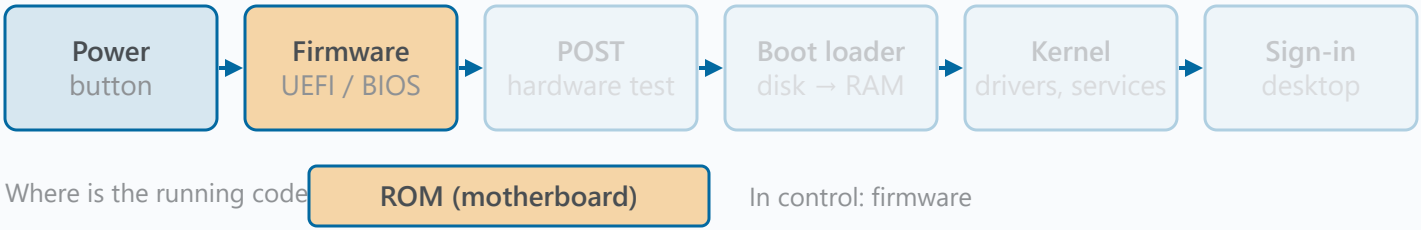


How Does a Computer Start? Booting Step by Step



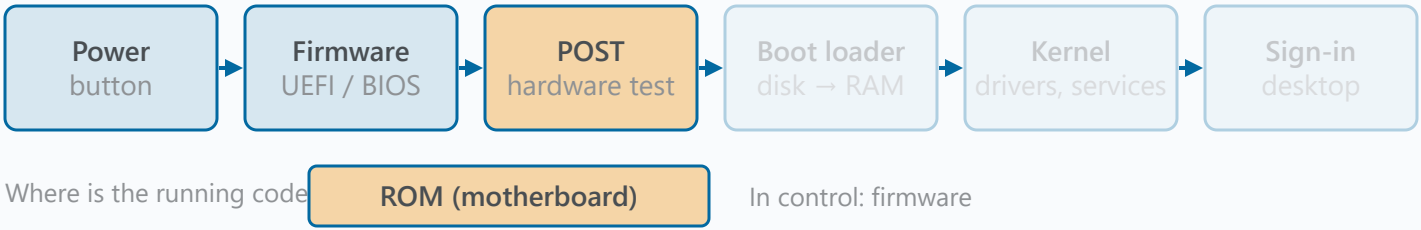
The power button is pressed; once voltages settle, the power supply sends the motherboard a “power good” signal. RAM is empty; the OS is still sitting on disk.

How Does a Computer Start? Booting Step by Step



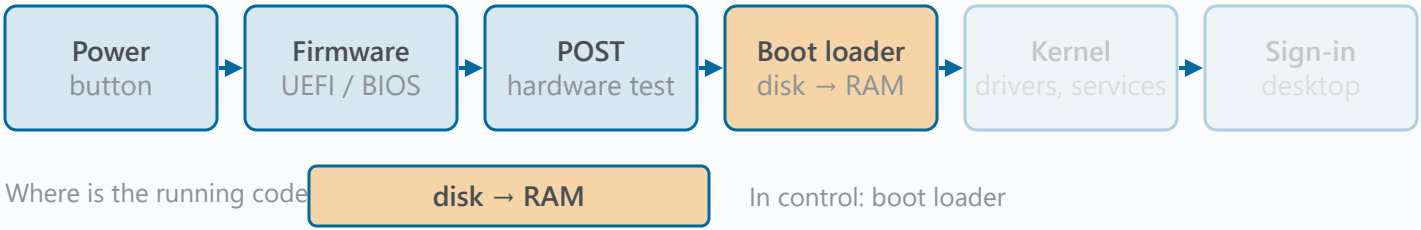
The CPU starts running the **firmware** (UEFI, or the older BIOS) stored in non-volatile memory (ROM/flash) on the motherboard.

How Does a Computer Start? Booting Step by Step



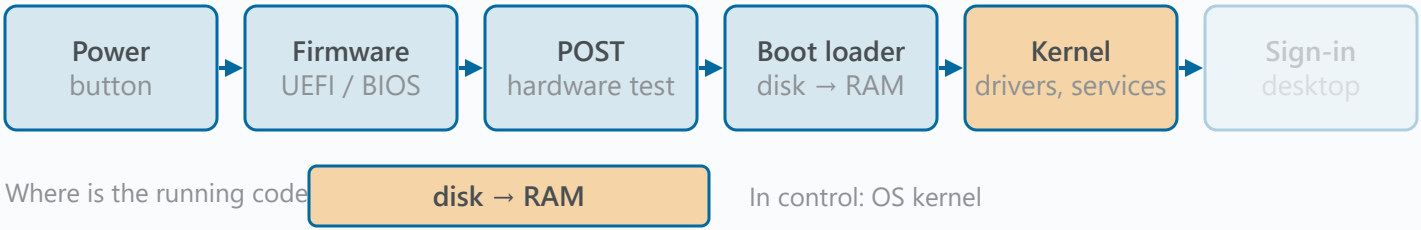
POST (power-on self-test): basic hardware such as memory, keyboard and disk is checked. Errors are reported with beeps or a message.

How Does a Computer Start? Booting Step by Step



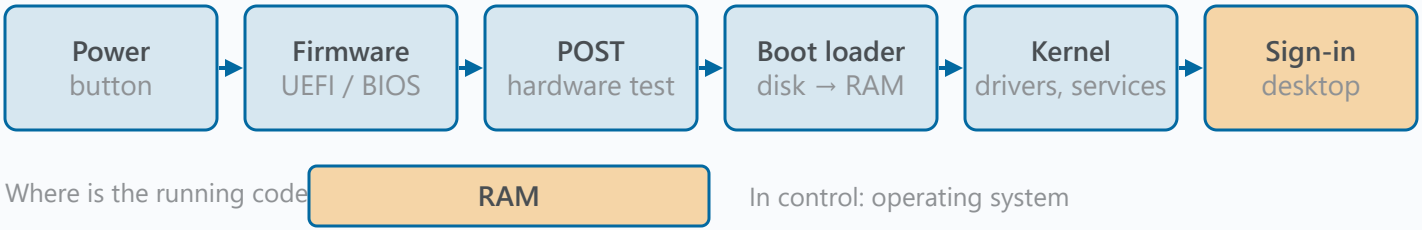
The boot order from the settings is tried (e.g. USB first, then SSD); missing devices are skipped. The **boot loader** (e.g. Windows Boot Manager: bootmgfw.efi, or GRUB) is loaded into RAM from the small **EFI system partition** (FAT32) on the disk.

How Does a Computer Start? Booting Step by Step



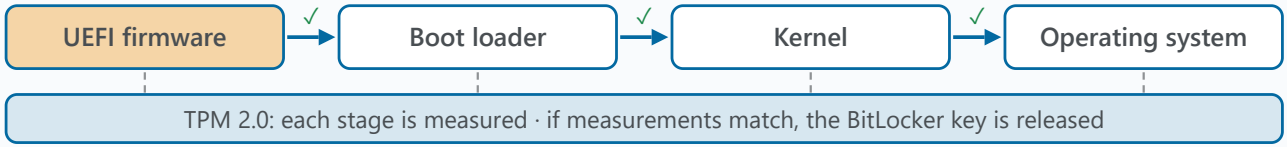
The boot loader loads the **kernel** into RAM (on Windows, ntoskrnl.exe via winload.efi) and hands over control. The kernel loads drivers and starts the first processes and system services.

How Does a Computer Start? Booting Step by Step



The **sign-in** screen appears; once the user is authenticated, their desktop and startup apps open.

Secure Boot and TPM



UEFI

Modern firmware that replaces BIOS: large disks, fast startup and signature checking.

Secure Boot

Only a boot loader and kernel with a trusted digital signature may run; forged or tampered boot code is blocked.

TPM 2.0

A security chip (on the motherboard, in the CPU or in firmware) that stores encryption keys and boot measurements; used by tools such as BitLocker.

Windows 11

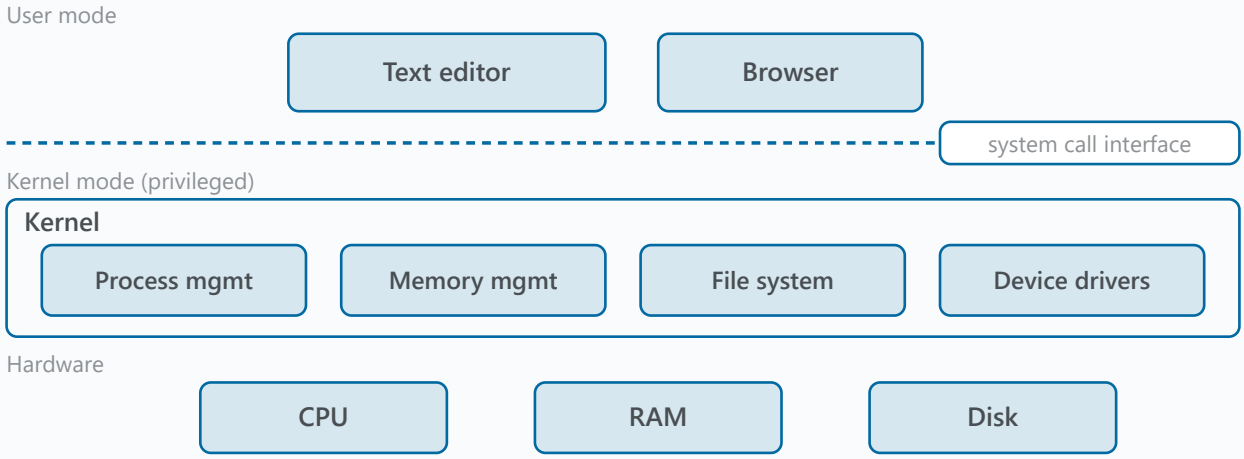
Requires UEFI with Secure Boot and TPM 2.0 to install; see Microsoft's current requirements page for details.

There are two partitioning schemes. **MBR**: goes with BIOS, disks up to 2 TB, at most 4 primary partitions. **GPT**: goes with UEFI, disks over 2 TB, many partitions.

OS COMPONENTS

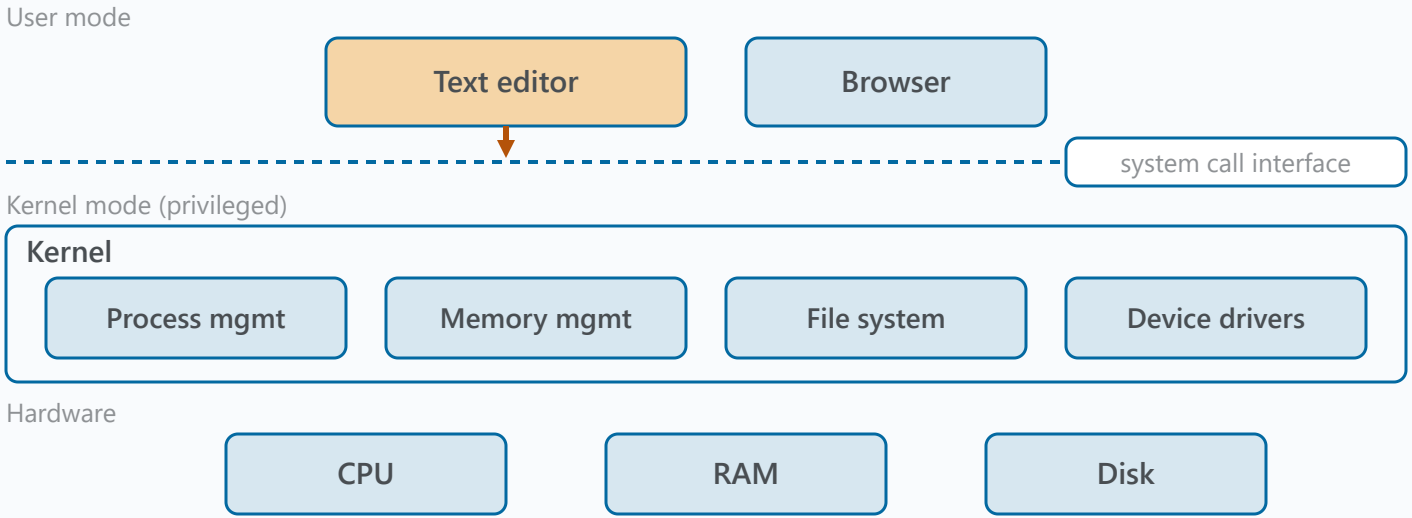
The Kernel

The kernel is the privileged core of the operating system. Applications don't touch the hardware directly; they ask the kernel for services through system calls.



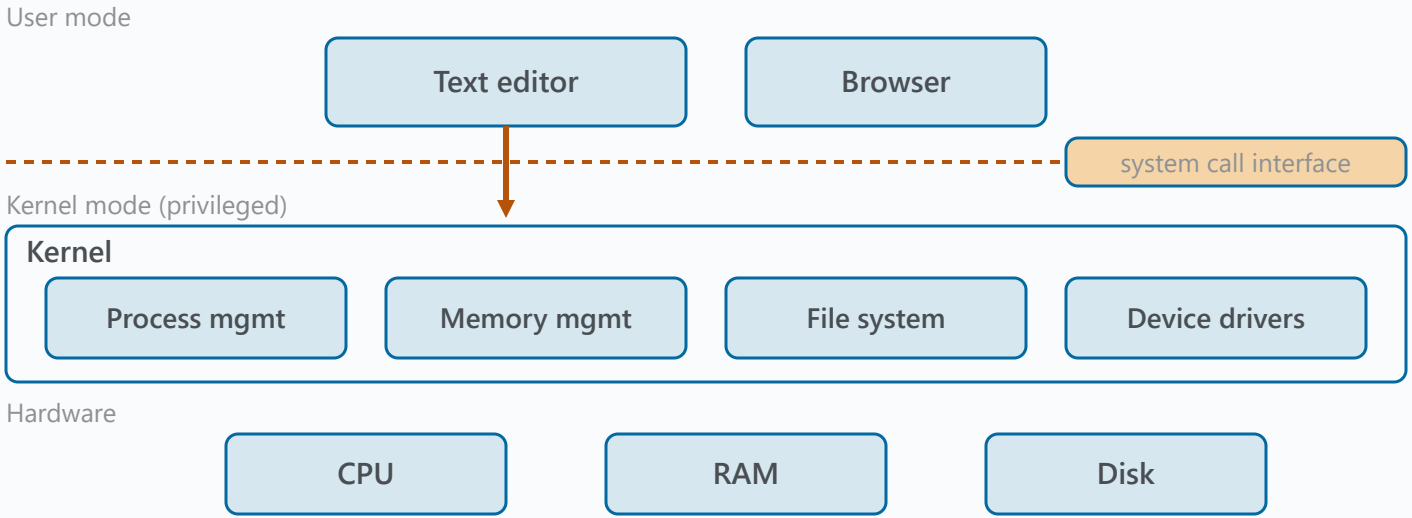
A bug in the kernel can bring down the whole system, which is why access control and isolation matter.

A System Call Step by Step: Reading a File



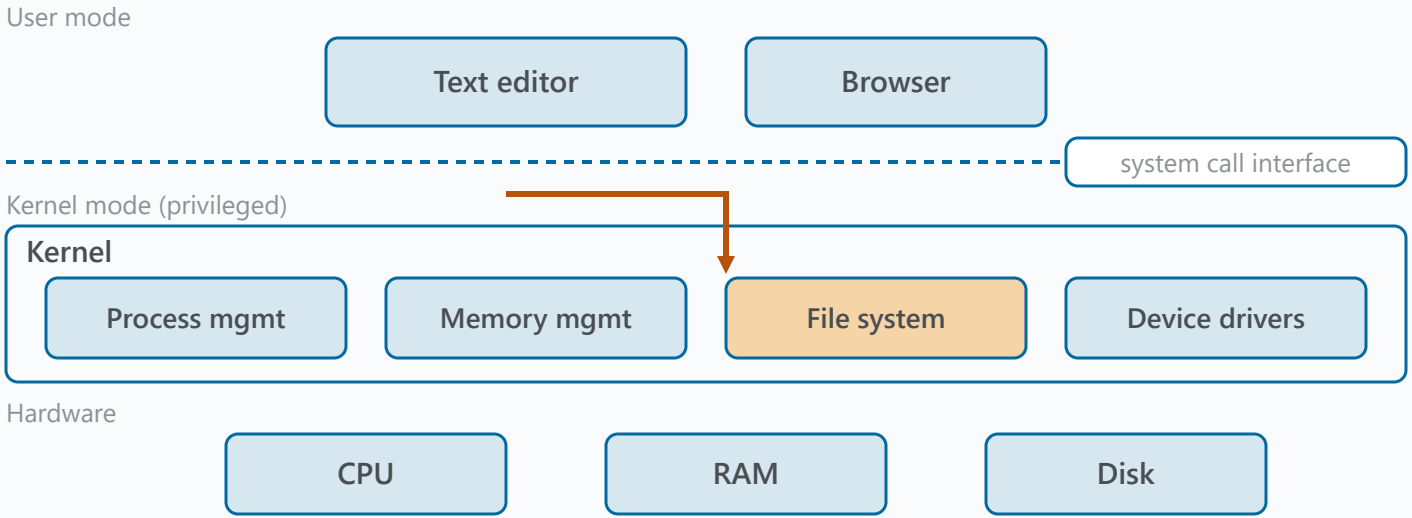
A text editor wants to read **report.txt**. It cannot access the disk itself, so it makes the **read** system call.

A System Call Step by Step: Reading a File



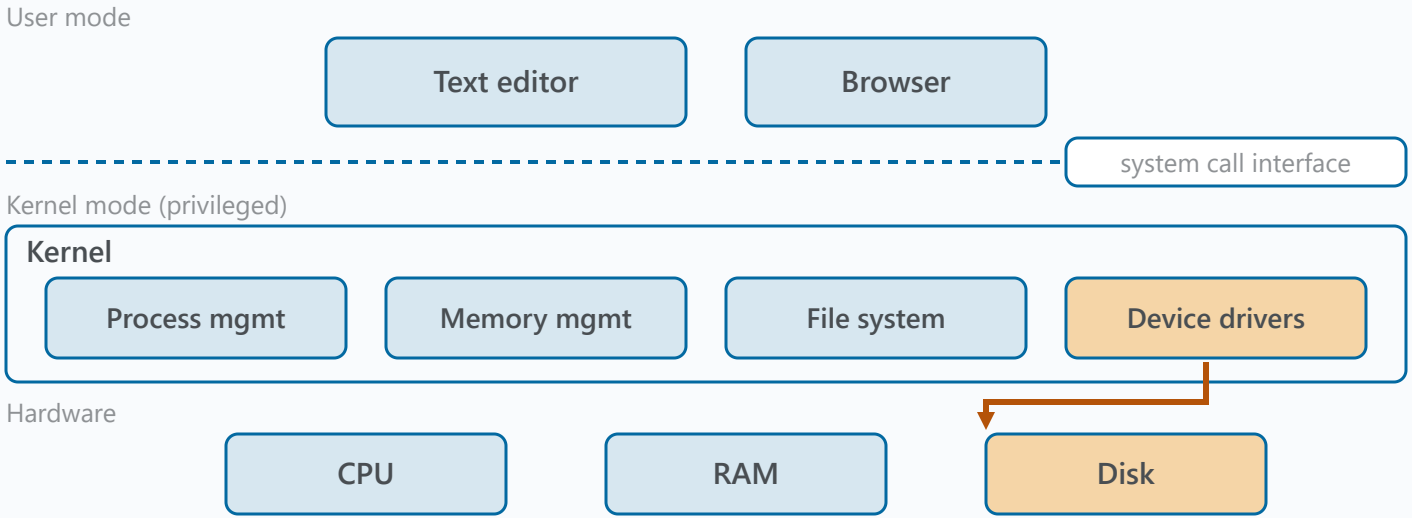
The CPU switches **from user mode to kernel mode**. Now only kernel code runs.

A System Call Step by Step: Reading a File



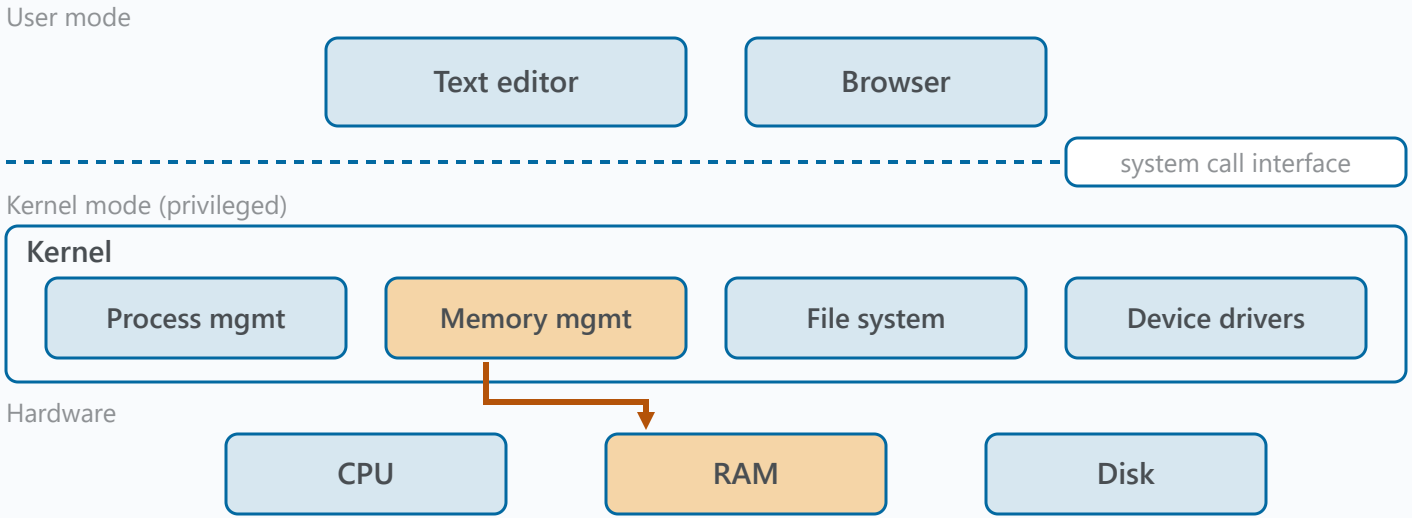
The **file system** finds where the file lives on disk and checks that the user has **read permission**.

A System Call Step by Step: Reading a File



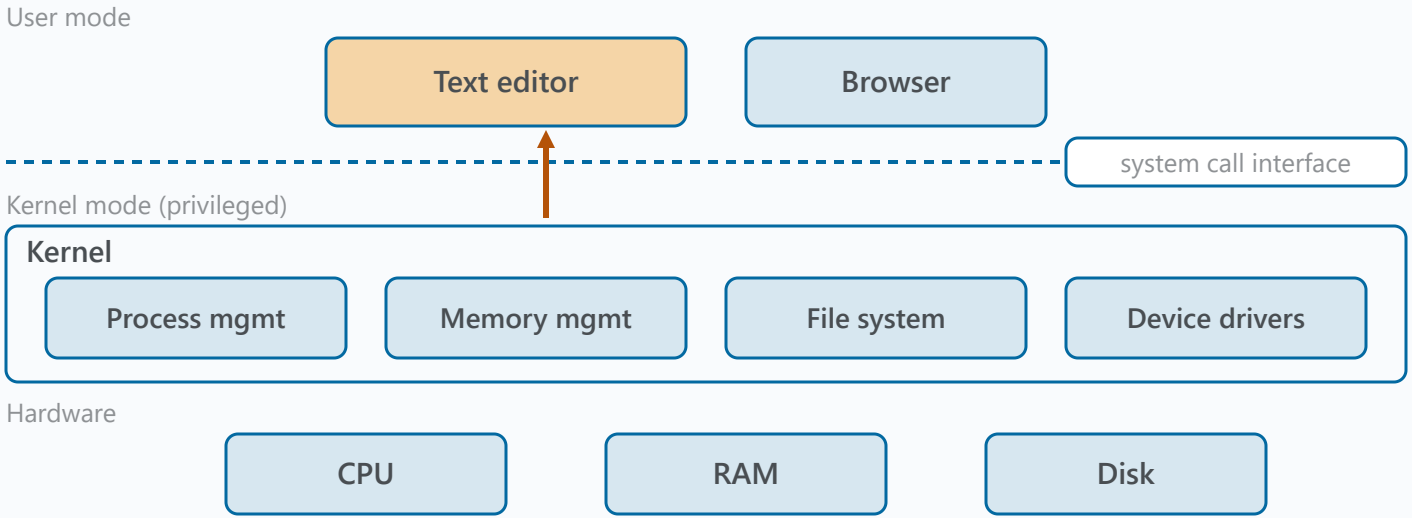
The **disk driver** sends a read command to the disk. When the disk is done, it notifies the CPU with an **interrupt**.

A System Call Step by Step: Reading a File



Memory management: the kernel copies the data into the editor's own memory area in RAM.

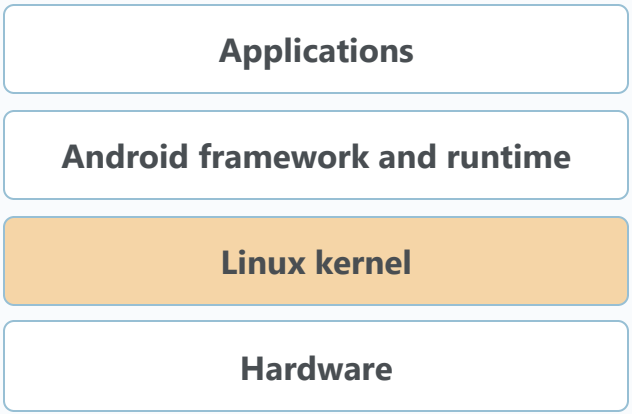
A System Call Step by Step: Reading a File



The call returns and the CPU switches back to **user mode**. The editor shows the data. The application never touched the hardware.

Kinds of Kernels

Type	Idea	Example
Monolithic kernel	Processes, memory, file system and drivers all in one privileged kernel; fast but large	Linux (loadable modules)
Microkernel	Only the essentials in the kernel; drivers and file system run in user mode; strong isolation	MINIX 3, QNX
Hybrid kernel	Starts from a microkernel design but keeps some services in the kernel for speed	Windows NT kernel, macOS/iOS (XNU)



The Android stack

Linux is just a kernel; a Linux distribution is the kernel plus tools plus a desktop. **Android** uses the Linux kernel, but the layers above it are Android's own.

Process Management

Program
Instructions and data stored on disk.

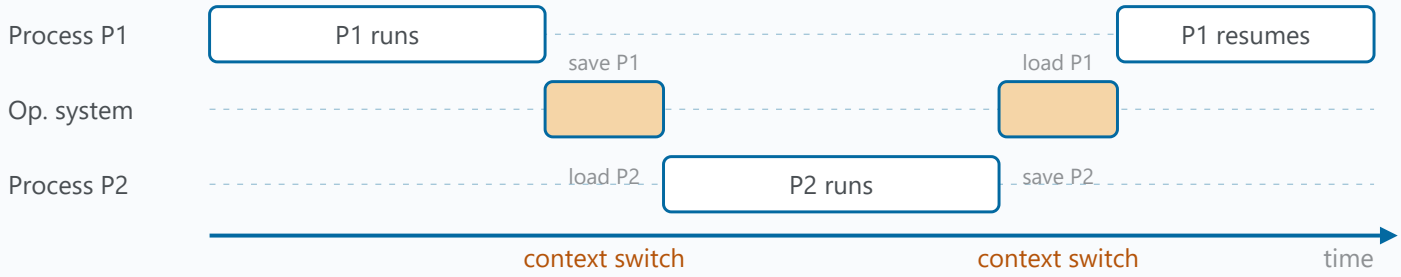
Process
A running instance of a program.

Thread
A flow of execution inside a process.

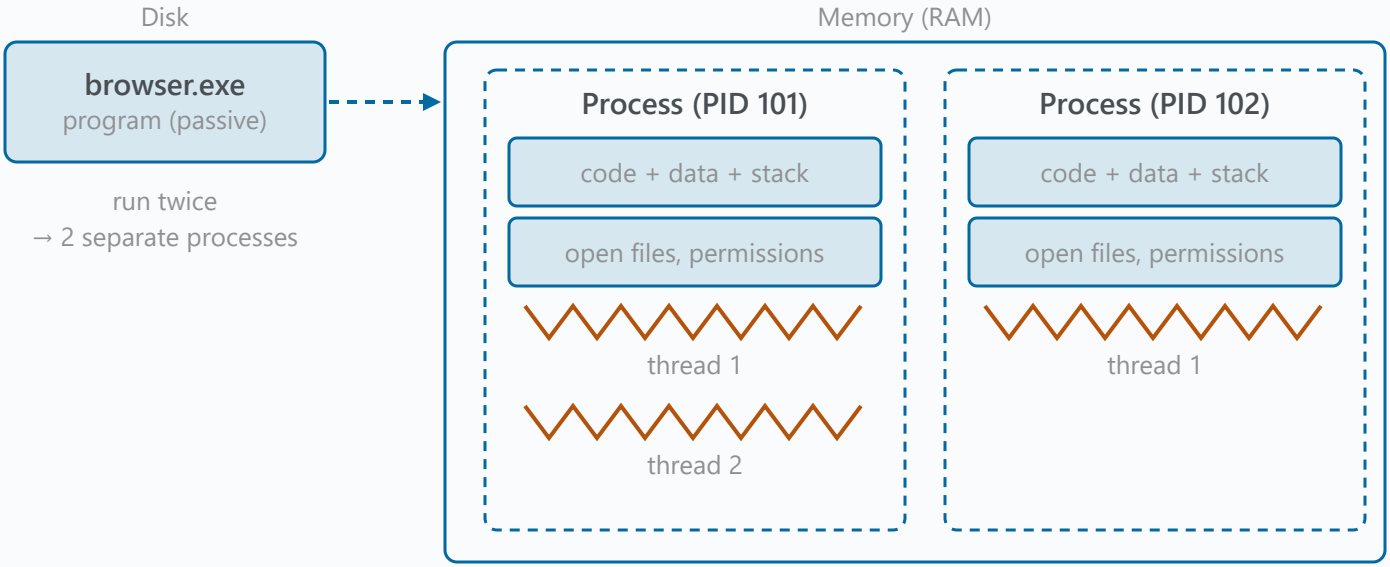
Scheduling
Decides which process gets the CPU and when.

Context switch
Moves the CPU state from one process to another.

Concurrency
Lets several jobs make progress together.

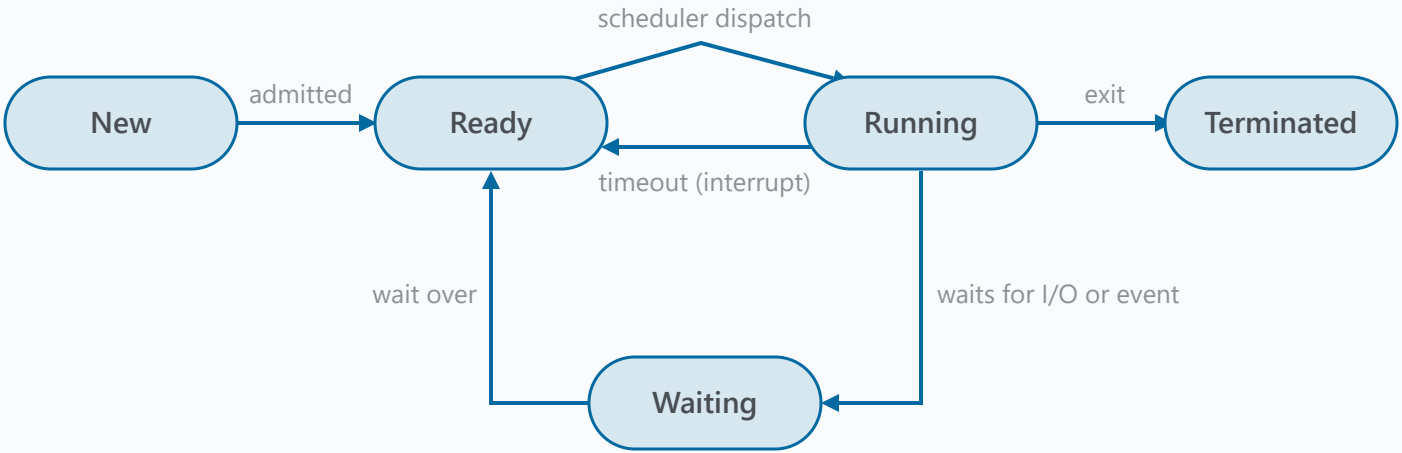


Program, Process and Thread



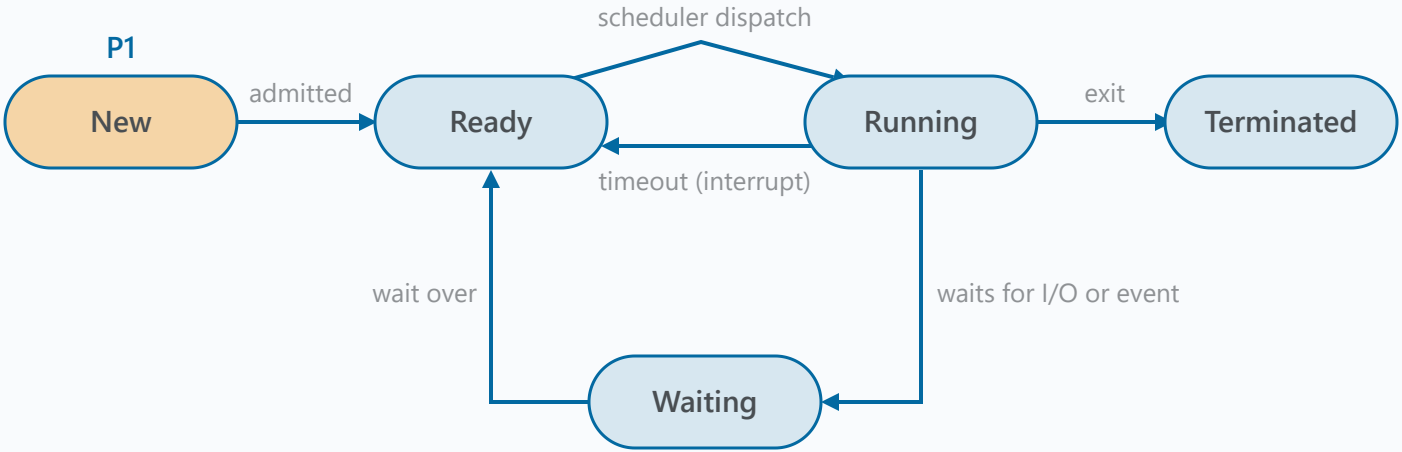
A program is a file sitting on disk; a process is a running instance of it, identified by a process ID (PID). Each process has its own memory space, while threads share the memory of their process.

Process States



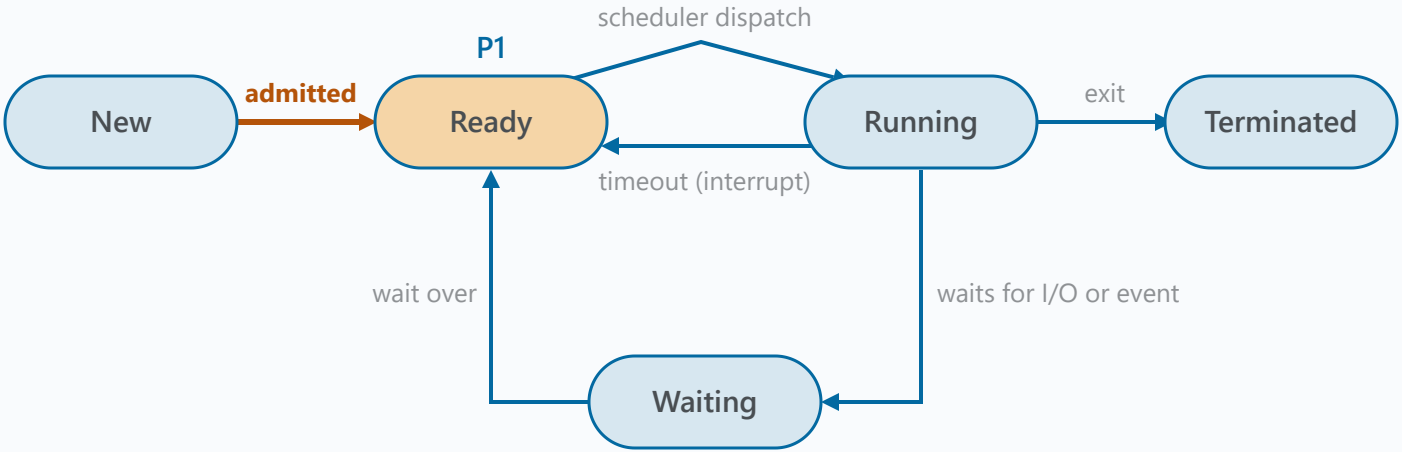
- On a single core, only **one** process is **Running** at a time; the other ready processes wait their turn.
- A **Waiting** process never gets the CPU; when the event it waits for (disk, keyboard, network) happens, it becomes **Ready** again.

Process State Transitions Step by Step



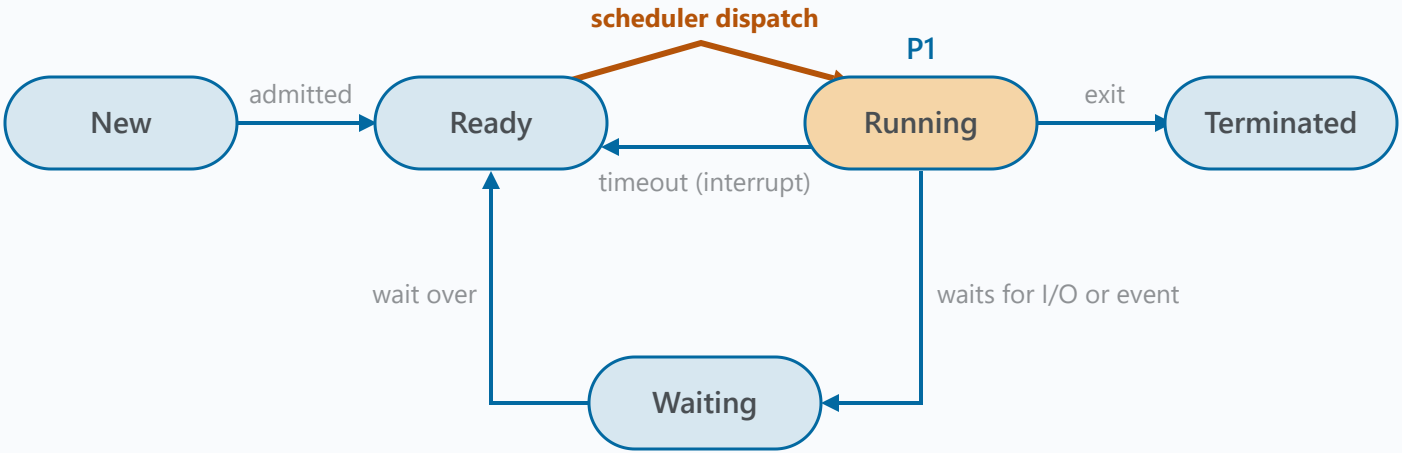
Process **P1** is being created (e.g. the user double-clicked a program): state **New**.

Process State Transitions Step by Step



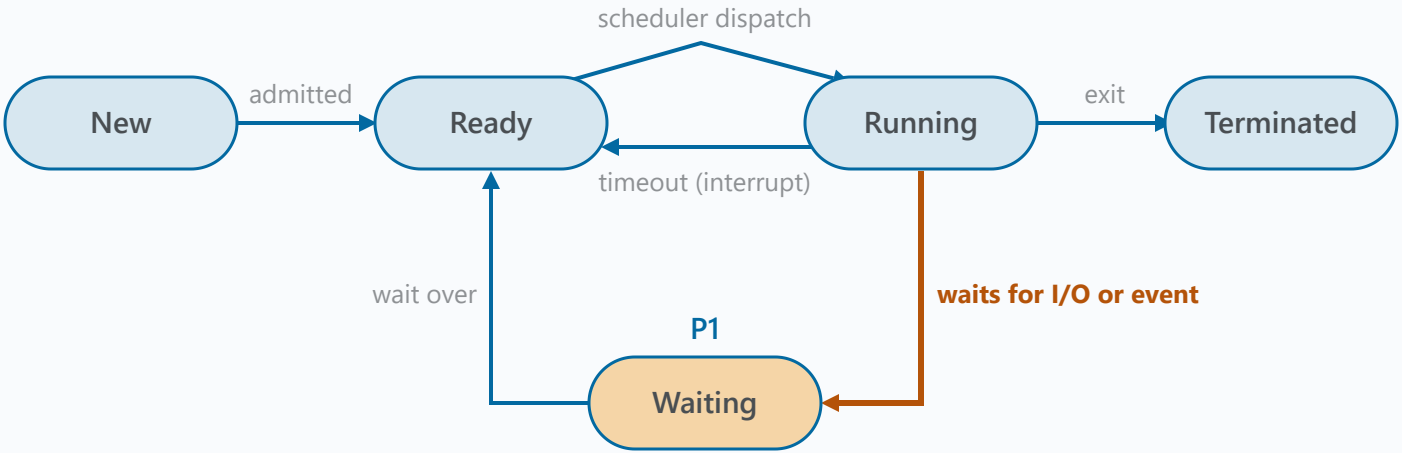
New → **Ready**: the OS **admits** the process; memory is allocated and it joins the ready queue.

Process State Transitions Step by Step



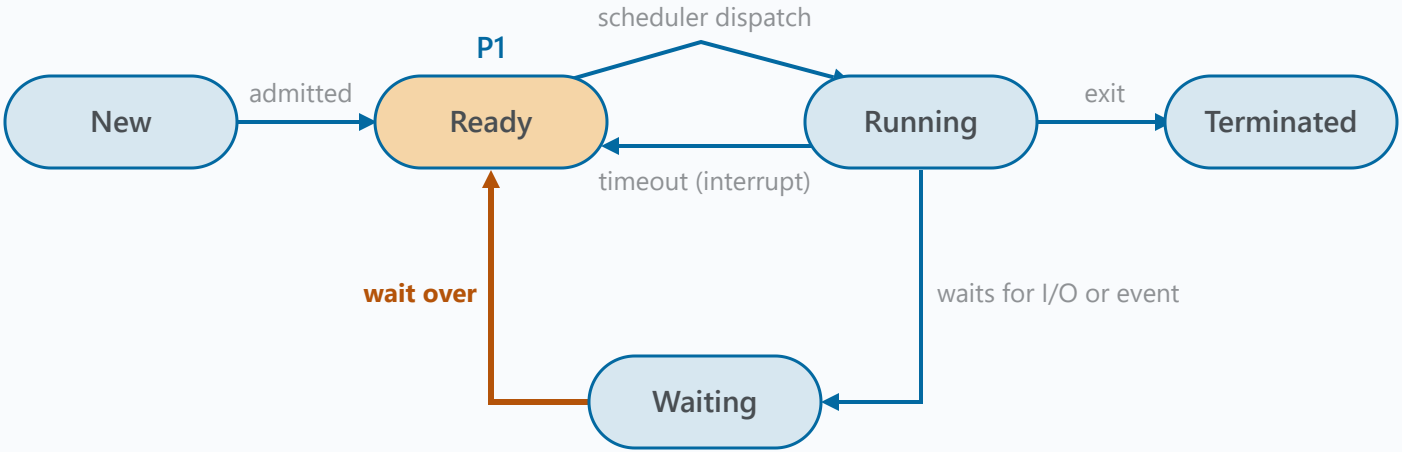
Ready → **Running**: the **scheduler** picks P1; its context is loaded and the CPU executes P1's instructions.

Process State Transitions Step by Step



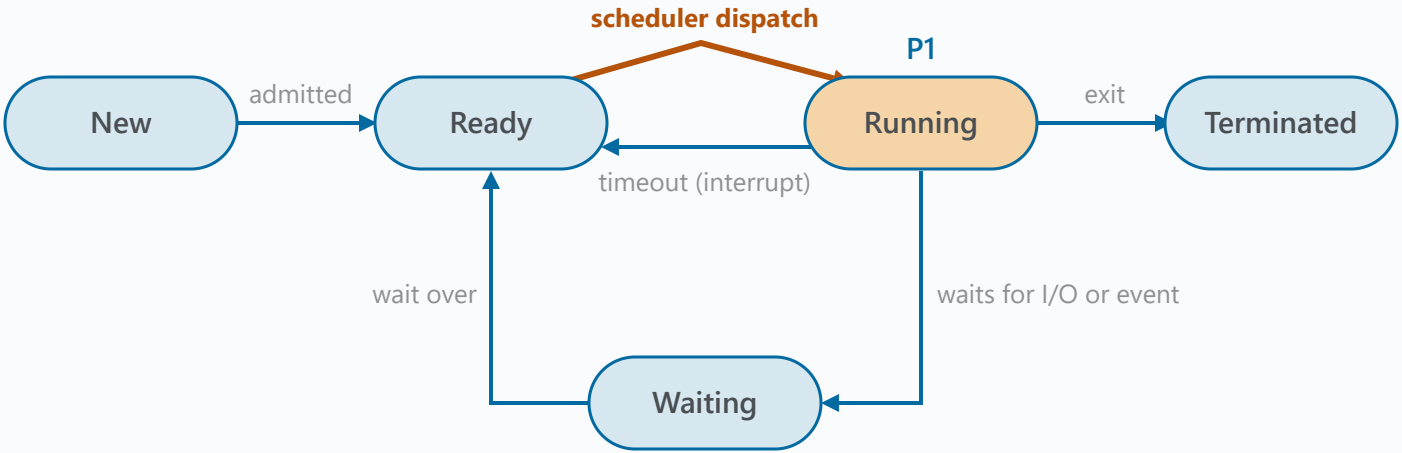
Running → **Waiting**: P1 asks for data from the disk. Rather than hold the CPU idle until it arrives, it moves to **Waiting**.

Process State Transitions Step by Step



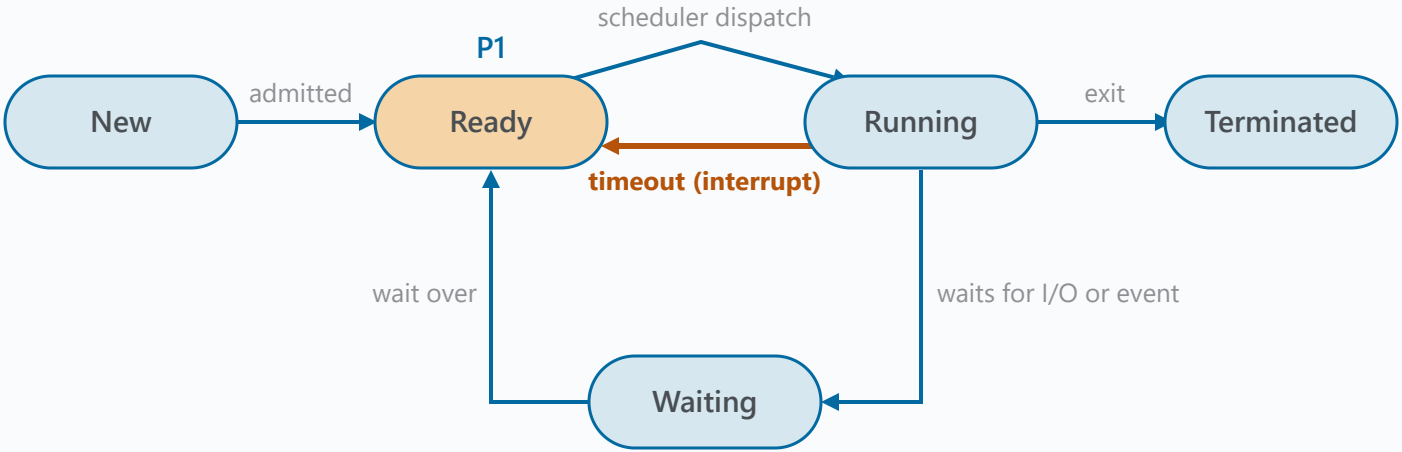
Waiting → **Ready**: the disk is done (an interrupt arrives). P1 doesn't run right away; it goes back to the **Ready** queue first.

Process State Transitions Step by Step



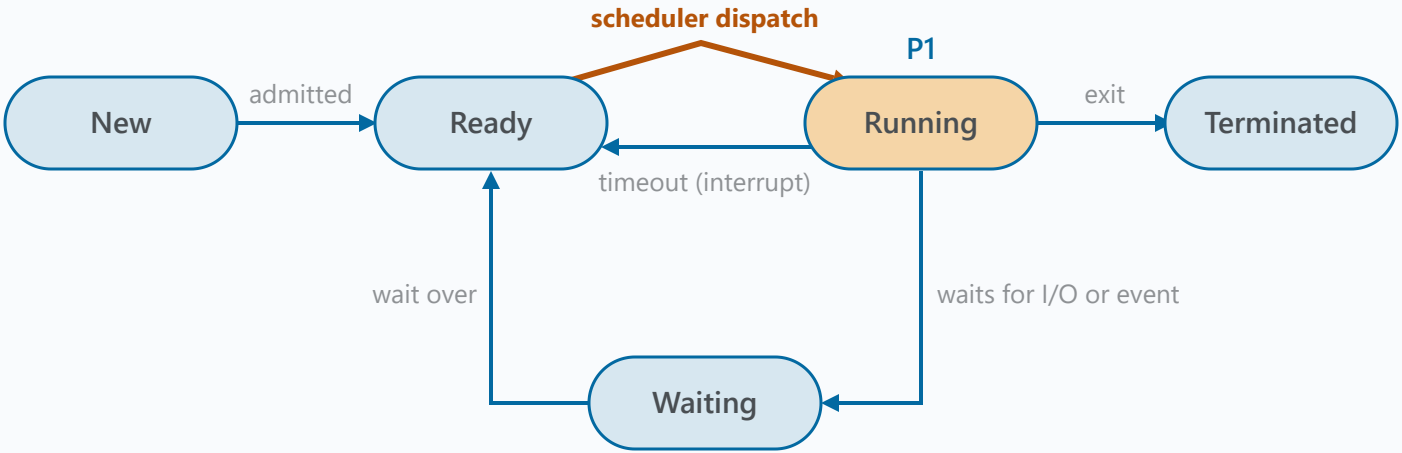
Ready → **Running**: when its turn comes, P1 is picked again and continues where it left off.

Process State Transitions Step by Step



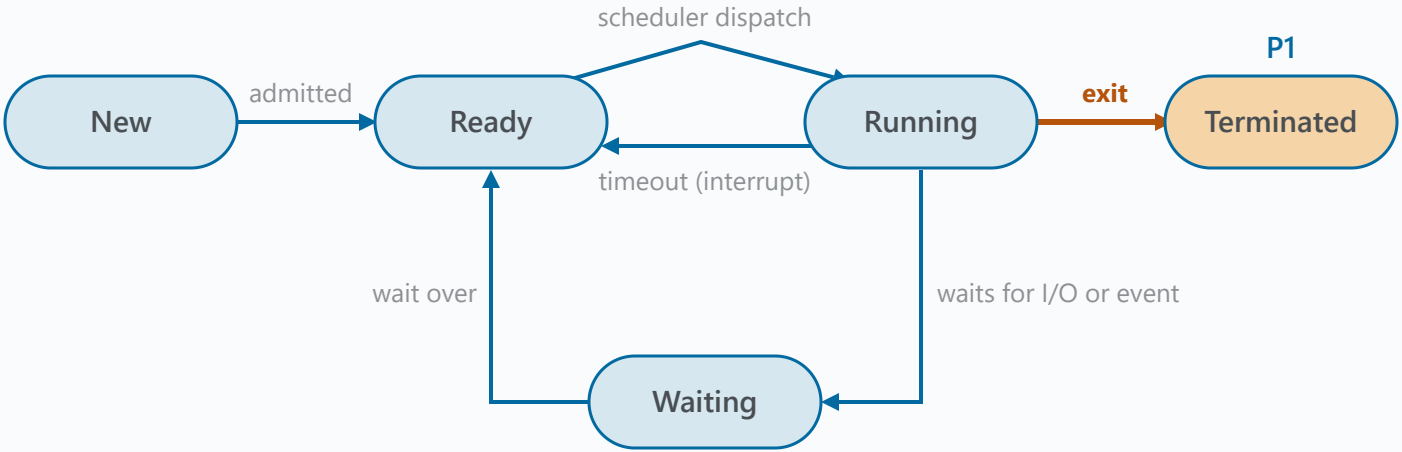
Running → **Ready**: P1's **time slice is up**; a timer interrupt gives the CPU to another process.

Process State Transitions Step by Step



Ready → **Running**: P1 is picked again and runs the rest of its work.

Process State Transitions Step by Step



Running → **Terminated**: P1 finishes and **exits**; the OS reclaims its memory and resources.

Two Processes, One CPU: Tracing States



Two processes, one core. P1 will compute for 3 ms, read from disk (3 ms), then compute 2 ms more. P2 just computes for 4 ms. Both are **Ready**; the scheduler picks P1.

Step 1 / 8

```
1 P1: compute 3 ms
2 P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
4 P2: compute 4 ms, exit
```

VARIABLES

`t = 0` `P1 = Ready` `P2 = Ready` `CPU = idle`

Two Processes, One CPU: Tracing States



t = 0–3: **P1 is Running**; P2 waits in the ready queue.

Step 2 / 8

```
▶ P1: compute 3 ms
2 P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
4 P2: compute 4 ms, exit
```

VARIABLES

`t = 3` `P1 = Running` `P2 = Ready` `CPU = P1`

Two Processes, One CPU: Tracing States



t = 3: P1 asks to read from disk (a system call). The disk is slow, so P1 goes to **Waiting** and gives up the CPU. Context switch.

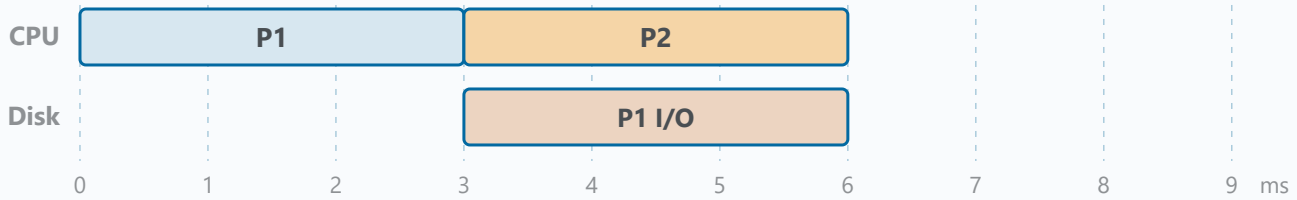
Step 3 / 8

```
1 P1: compute 3 ms
▶ P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
4 P2: compute 4 ms, exit
```

VARIABLES

`t = 3` `P1 = Waiting` `P2 = Ready` `CPU = idle`

Two Processes, One CPU: Tracing States



t = 3–6: **P2 is Running** while the disk reads P1's data. CPU and disk work **in parallel**.

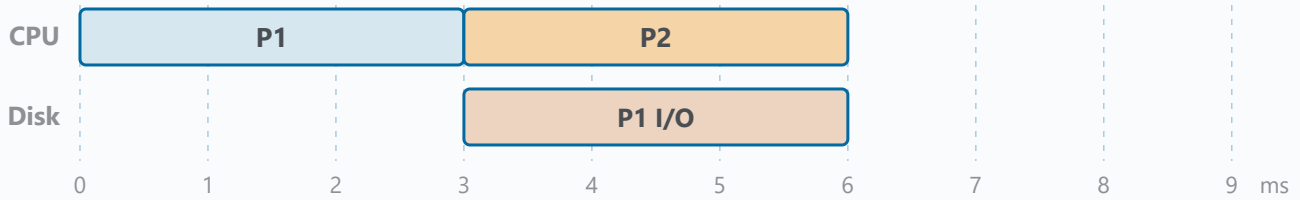
Step 4 / 8

```
1 P1: compute 3 ms
2 P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
▶ P2: compute 4 ms, exit
```

VARIABLES

`t = 6` `P1 = Waiting` `P2 = Running` `CPU = P2`

Two Processes, One CPU: Tracing States



t = 6: the disk finishes and signals with an **interrupt**. P1 becomes **Ready** but doesn't run yet: P2 has the CPU.

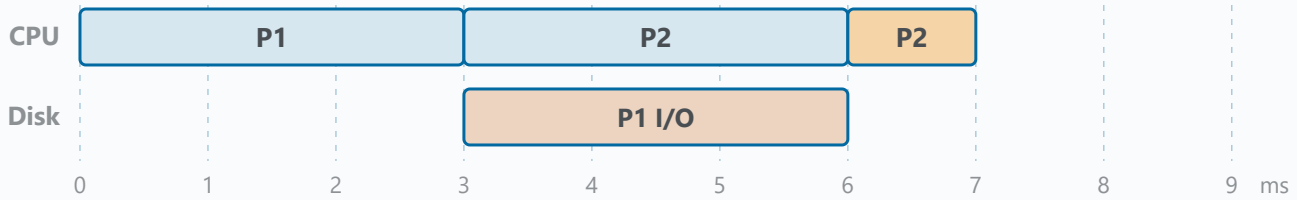
Step 5 / 8

```
1 P1: compute 3 ms
▶ P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
4 P2: compute 4 ms, exit
```

VARIABLES

`t = 6` `P1 = Ready` `P2 = Running` `CPU = P2`

Two Processes, One CPU: Tracing States



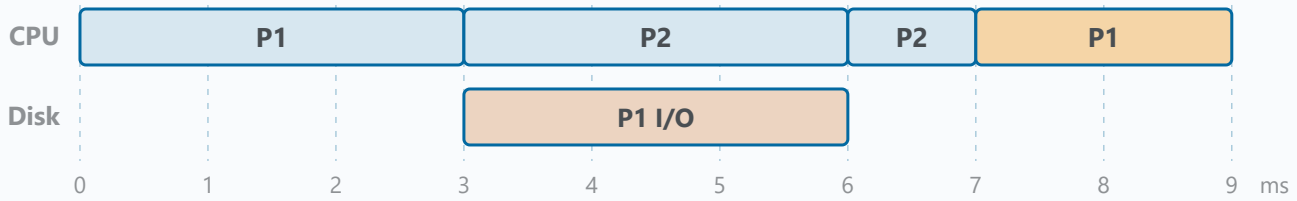
t = 7: P2 completes its 4 ms and is **Terminated**. Only P1 is left in the ready queue.


```
1 P1: compute 3 ms
2 P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
▶ P2: compute 4 ms, exit
```

VARIABLES

`t = 7` `P1 = Ready` `P2 = Terminated` `CPU = idle`

Two Processes, One CPU: Tracing States



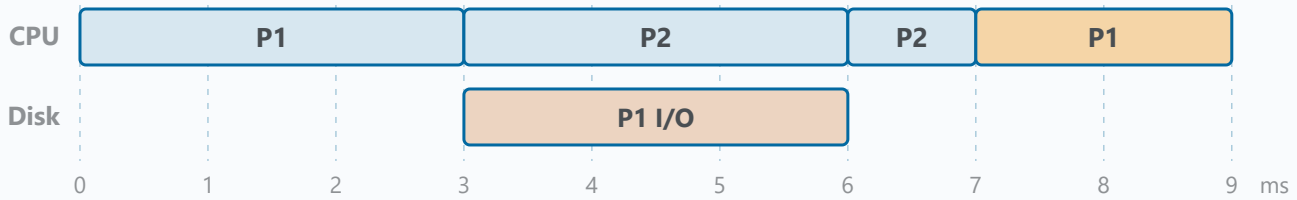
t = 7–9: **P1 is Running**; it finishes its last 2 ms and is **Terminated**.


```
1 P1: compute 3 ms
2 P1: read disk (I/O 3 ms)
▶ P1: compute 2 ms, exit
4 P2: compute 4 ms, exit
```

VARIABLES

`t = 9` `P1 = Terminated` `P2 = Terminated` `CPU = idle`

Two Processes, One CPU: Tracing States



Total: **9 ms**, and the CPU was never idle. Had the CPU sat idle while P1 waited for the disk, it would have taken 3 + 3 + 2 + 4 = **12 ms**. Running another process during I/O (multiprogramming) saves time.

Step 8 / 8

```
1 P1: compute 3 ms
2 P1: read disk (I/O 3 ms)
3 P1: compute 2 ms, exit
4 P2: compute 4 ms, exit
```

VARIABLES

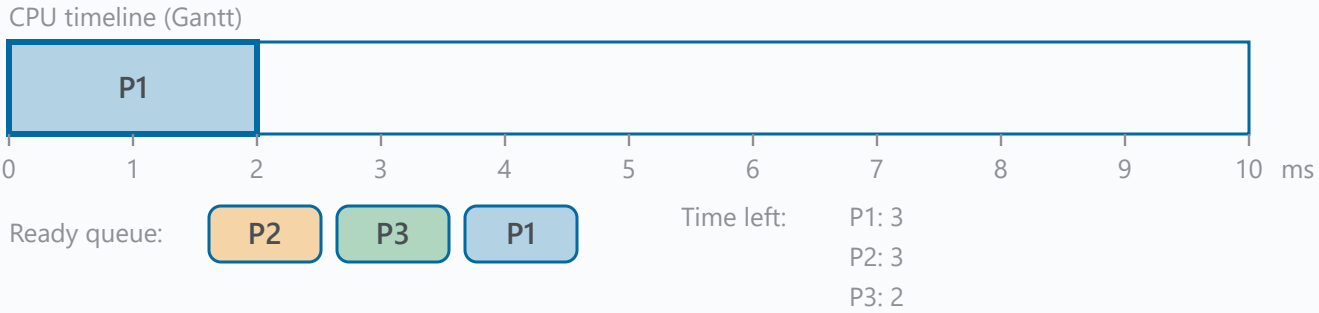
`t = 9` `P1 = Terminated` `P2 = Terminated` `CPU = idle`

Time Sharing: Round-Robin CPU Allocation



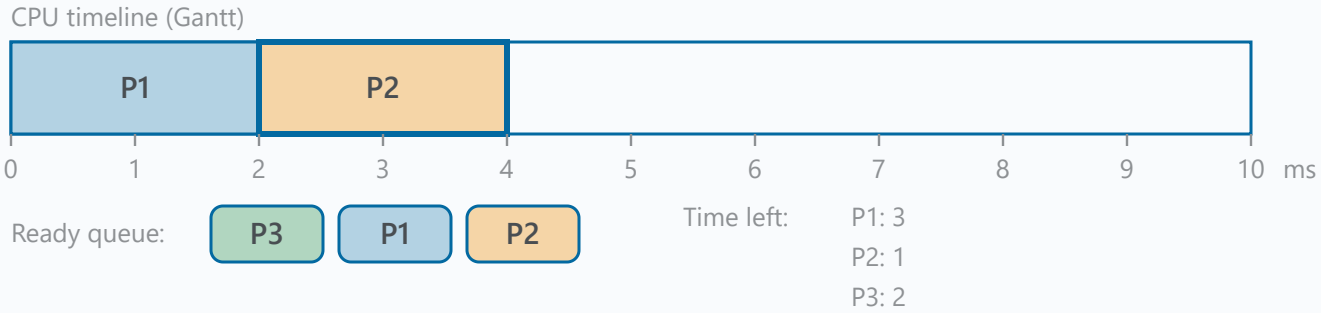
Three processes are ready: P1 needs 5 ms, P2 3 ms, P3 2 ms. **Round-robin** scheduling gives each one at most **2 ms** in turn.

Time Sharing: Round-Robin CPU Allocation



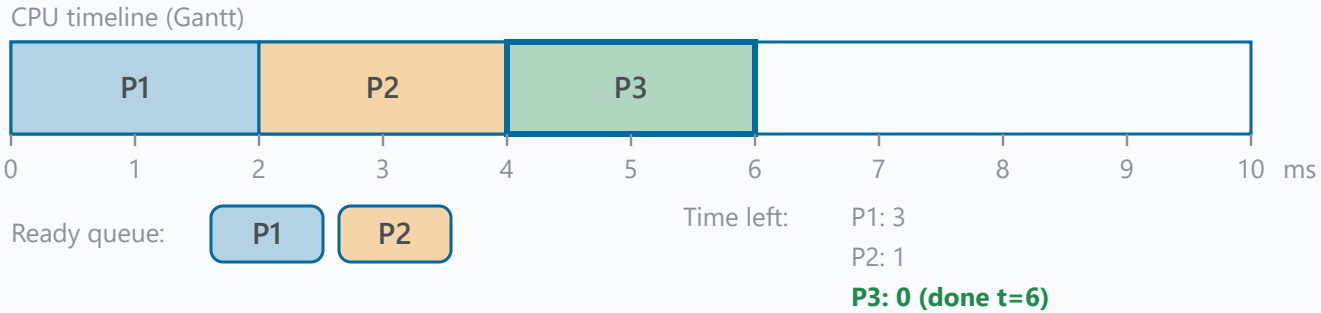
t = 0–2: the CPU runs **P1**. Its time slice expires; P1 goes to the back of the queue (3 ms left). Context switch.

Time Sharing: Round-Robin CPU Allocation



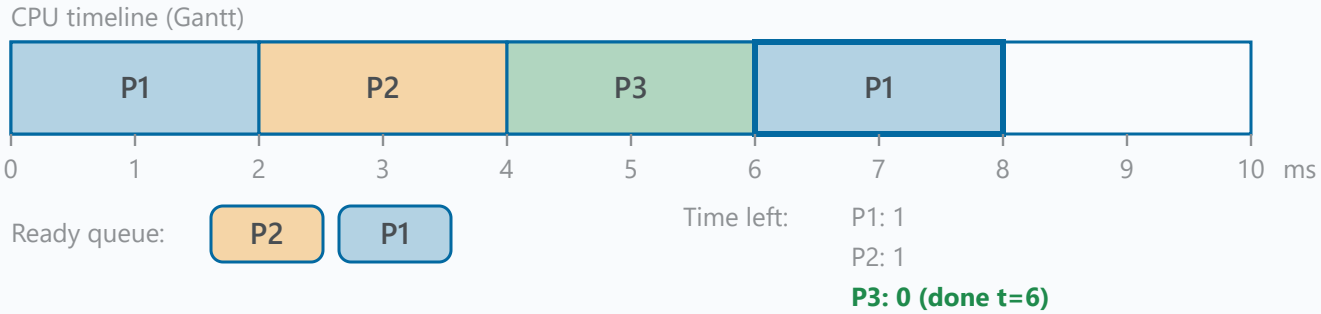
t = 2–4: the CPU runs **P2**. Its time slice expires; P2 goes to the back of the queue (1 ms left). Context switch.

Time Sharing: Round-Robin CPU Allocation



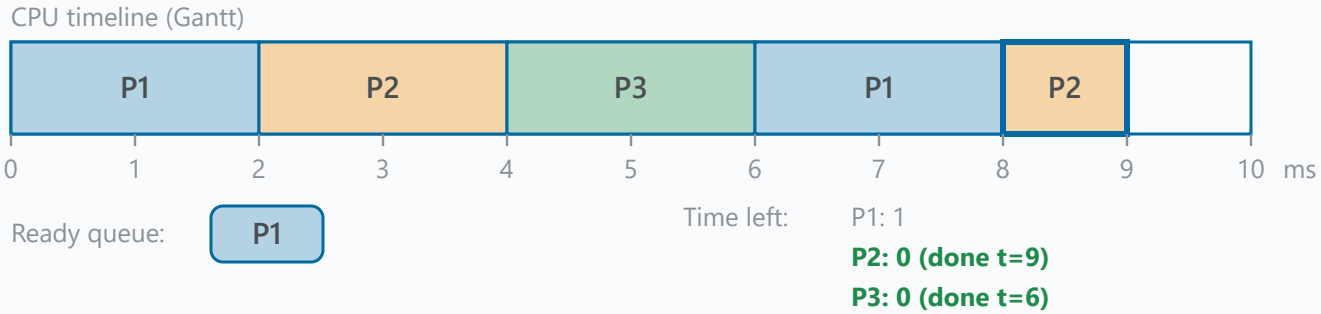
t = 4–6: the CPU runs **P3**. P3 finishes (t = 6) and leaves the system.

Time Sharing: Round-Robin CPU Allocation



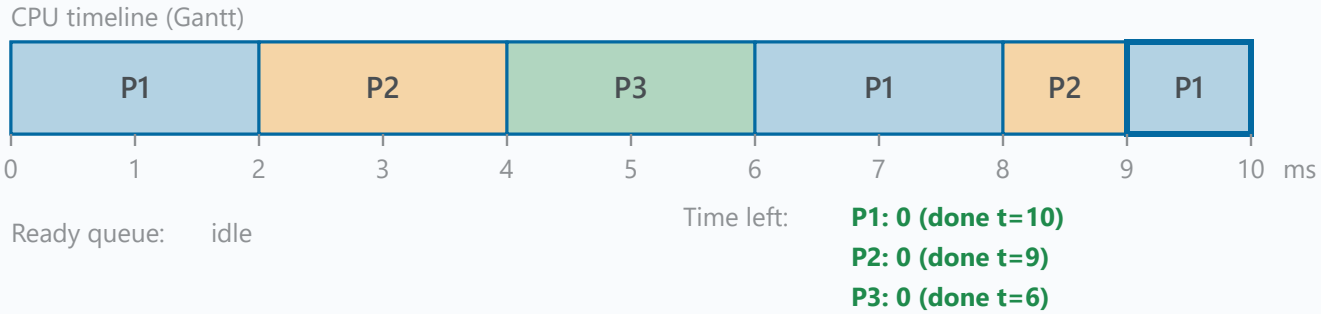
t = 6–8: the CPU runs **P1**. Its time slice expires; P1 goes to the back of the queue (1 ms left). Context switch.

Time Sharing: Round-Robin CPU Allocation



t = 8–9: the CPU runs **P2**. P2 finishes (t = 9) and leaves the system.

Time Sharing: Round-Robin CPU Allocation



t = 9–10: the CPU runs **P1**. P1 finishes (t = 10) and leaves the system. Completion: P3 = 6, P2 = 9, P1 = 10 ms. Because switches are fast, all three seem to run “at the same time”.

Deadlock



solid arrow: process holds the resource · dashed arrow: process waits for it

Process A has acquired the **printer** and is using it (solid arrow).

Step 1 / 5

Mutual exclusion	**Hold and wait**	**No preemption**	**Circular wait**
one owner	waits while holding	can't be taken away	forms a ring

All four conditions are needed. One fix: everyone requests resources in the same **order** (printer first, then file), so no circular wait can form.

Deadlock



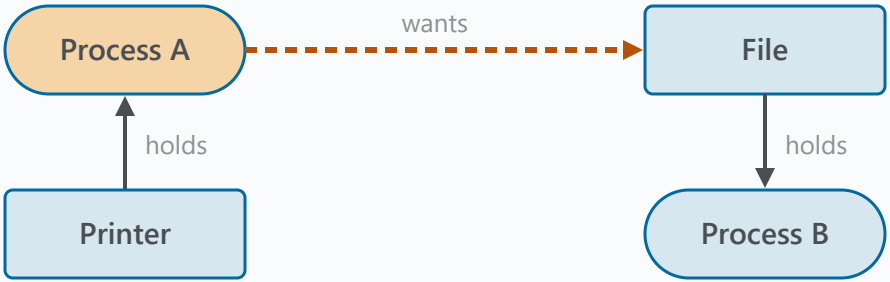
solid arrow: process holds the resource · dashed arrow: process waits for it

Process B has locked the **file** and is using it.

Mutual exclusion	**Hold and wait**	**No preemption**	**Circular wait**
one owner	waits while holding	can't be taken away	forms a ring

All four conditions are needed. One fix: everyone requests resources in the same **order** (printer first, then file), so no circular wait can form.

Deadlock



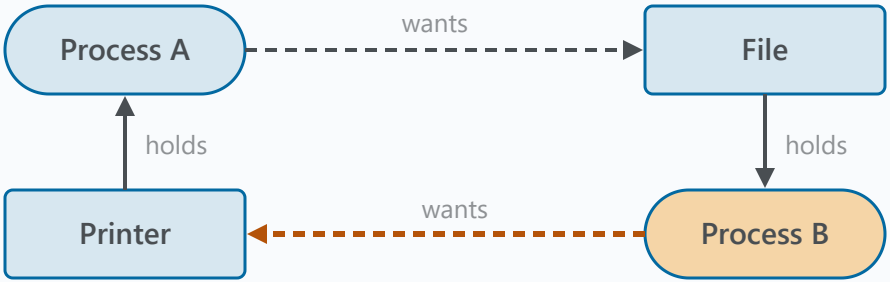
solid arrow: process holds the resource · dashed arrow: process waits for it

Process A wants the **file** without releasing the printer (dashed arrow); B has the file, so A waits.

Mutual exclusion	**Hold and wait**	**No preemption**	**Circular wait**
one owner	waits while holding	can't be taken away	forms a ring

All four conditions are needed. One fix: everyone requests resources in the same **order** (printer first, then file), so no circular wait can form.

Deadlock



solid arrow: process holds the resource · dashed arrow: process waits for it

Process B wants the **printer** without releasing the file; A has the printer, so B waits.

Mutual exclusion	**Hold and wait**	**No preemption**	**Circular wait**
one owner	waits while holding	can't be taken away	forms a ring

All four conditions are needed. One fix: everyone requests resources in the same **order** (printer first, then file), so no circular wait can form.

Deadlock



solid arrow: process holds the resource · dashed arrow: process waits for it

A → File → B → Printer → A: a **circular wait**. Neither lets go, so both wait forever: **deadlock**.

Step 5 / 5

Mutual exclusion	**Hold and wait**	**No preemption**	**Circular wait**
one owner	waits while holding	can't be taken away	forms a ring

All four conditions are needed. One fix: everyone requests resources in the same **order** (printer first, then file), so no circular wait can form.

RESOURCE MANAGEMENT

Memory Management

Allocation

Gives each program the memory it needs.

Protection

Stops one process from changing another's memory.

Virtual memory

Gives programs an address space independent of physical RAM.

Paging

Manages memory in fixed-size pieces.

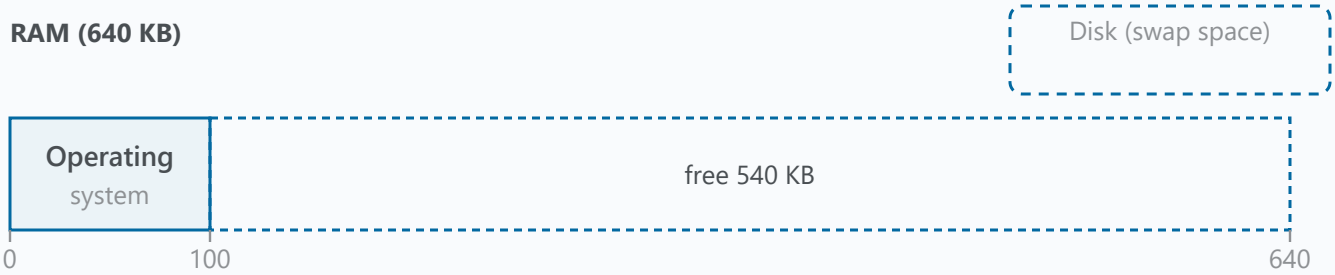
Swapping

Moves some data to disk temporarily when needed.

Sharing

Lets processes share code or data.

Memory Management Step by Step: Allocation, Protection, Swapping

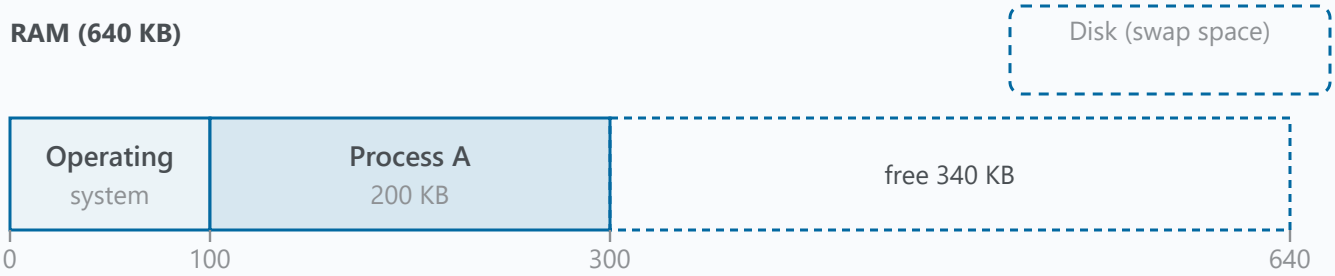


Base / limit registers: no user process yet

The start of RAM is reserved for the kernel; the rest is free.

Start: the **operating system (kernel)** sits at the start of RAM; the rest is free.

Memory Management Step by Step: Allocation, Protection, Swapping

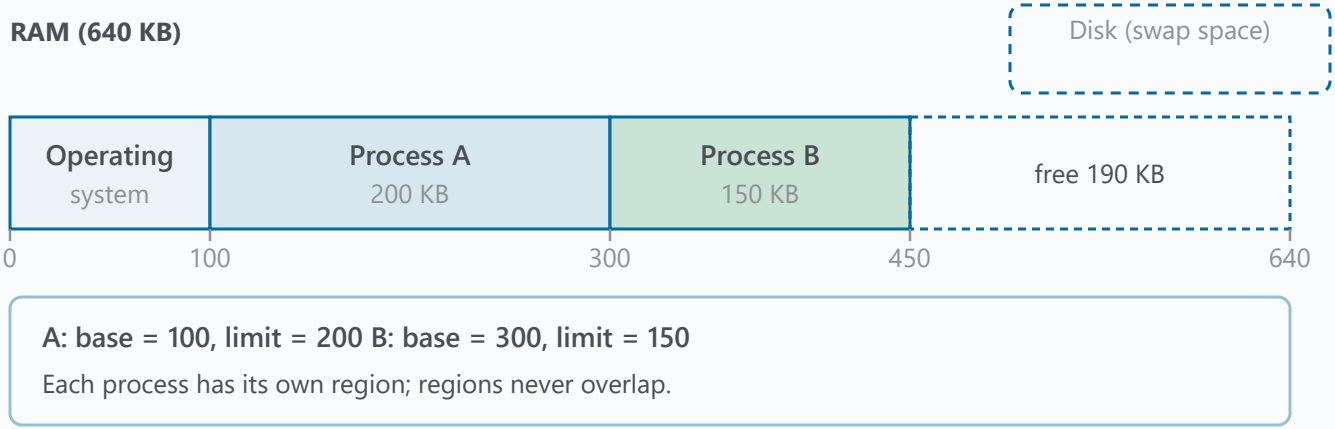


Process A → base = 100, limit = 200

Base: start of the region. Limit: its length (KB).

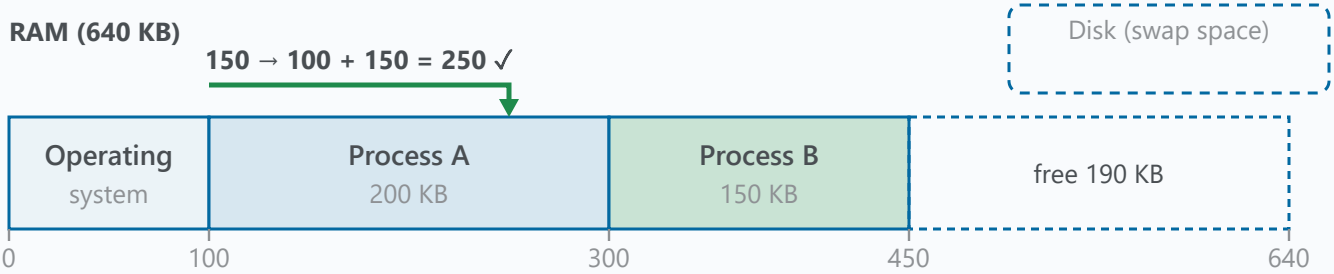
Allocation: 200 KB from 100 to 300 is set aside for A. In this simple model a region is described by **base** and **limit** registers (real systems use paging).

Memory Management Step by Step: Allocation, Protection, Swapping



B gets 150 KB from 300 to 450. Each process has its own register pair; the OS loads it on every switch.

Memory Management Step by Step: Allocation, Protection, Swapping

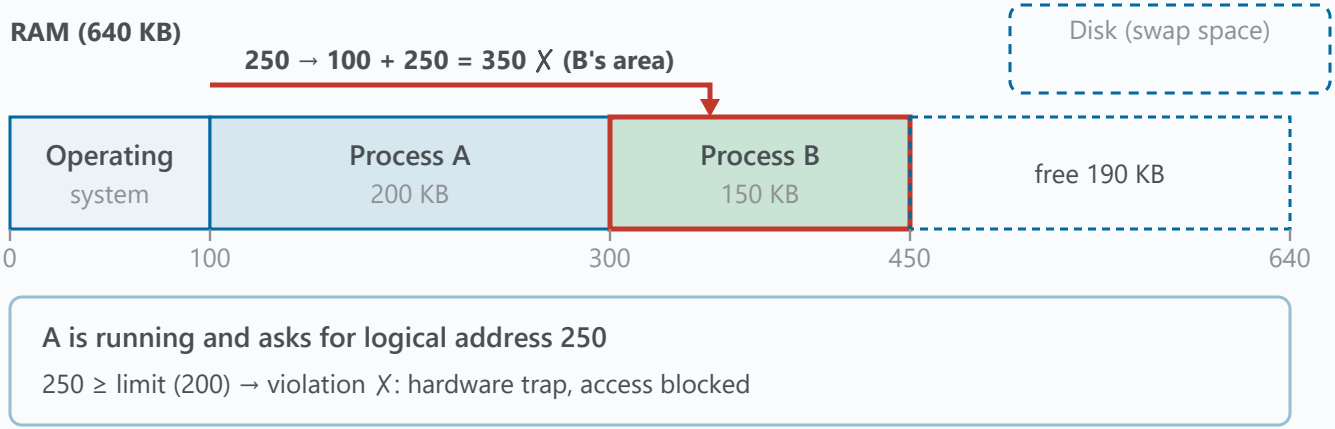


A is running and asks for logical address 150

150 < limit (200) → physical address = base + 150 = 250 ✓

Valid access: A's address 150 is below the limit (200), so the hardware goes to physical address **base + 150 = 250**.

Memory Management Step by Step: Allocation, Protection, Swapping

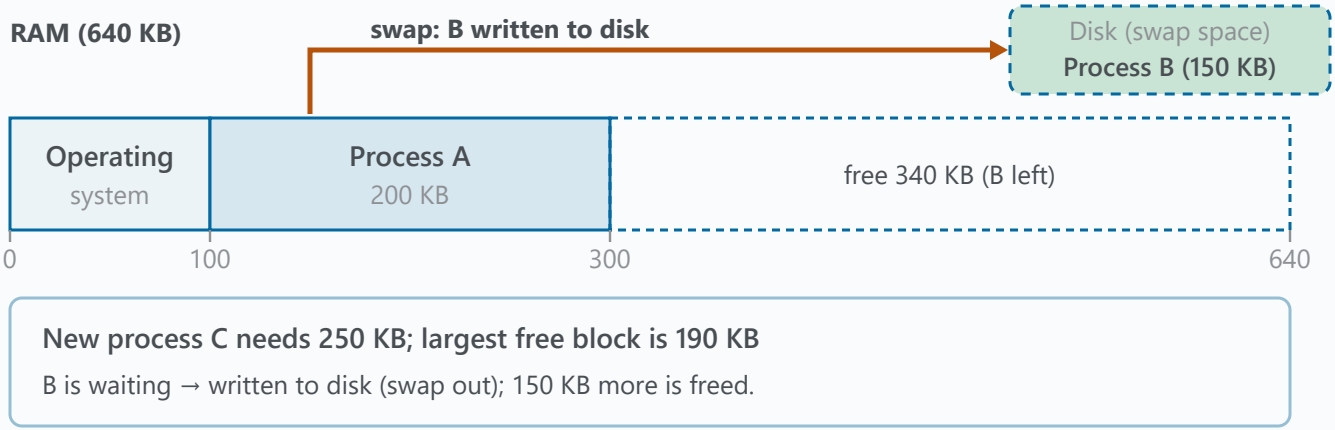


A is running and asks for logical address 250

250 ≥ limit (200) → violation ✗: hardware trap, access blocked

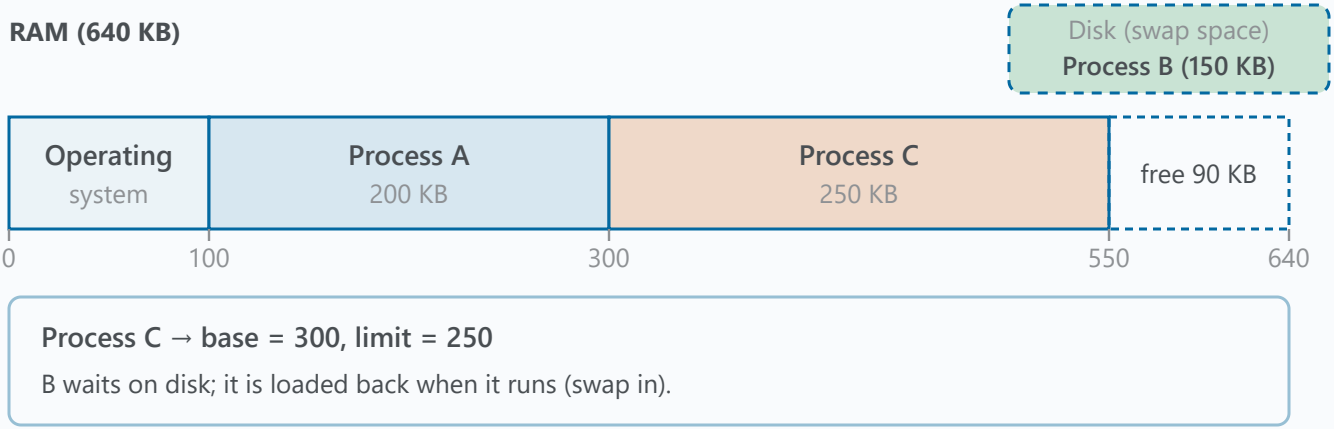
Protection fault: A asks for 250; since 250 ≥ 200, the hardware blocks the access and control passes to the kernel. A real system would kill A with an “access violation / segmentation fault”; in the example A's region stays put. B's memory is safe.

Memory Management Step by Step: Allocation, Protection, Swapping



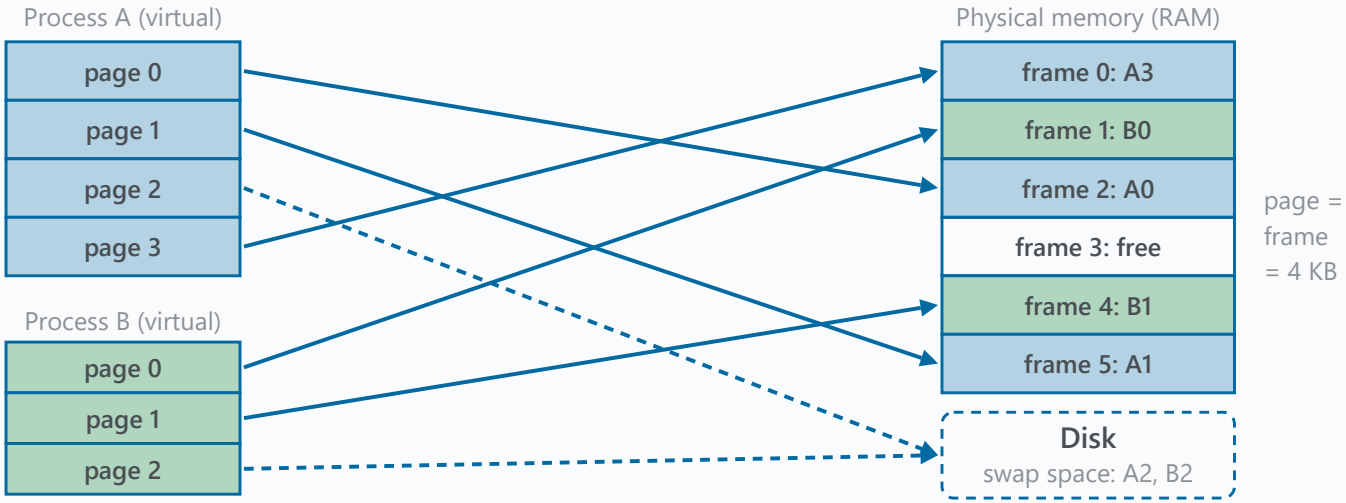
Swapping: there is no room for C (190 KB < 250 KB). The waiting process B is written to disk, leaving 340 KB of contiguous free RAM. Disk is far slower than RAM.

Memory Management Step by Step: Allocation, Protection, Swapping



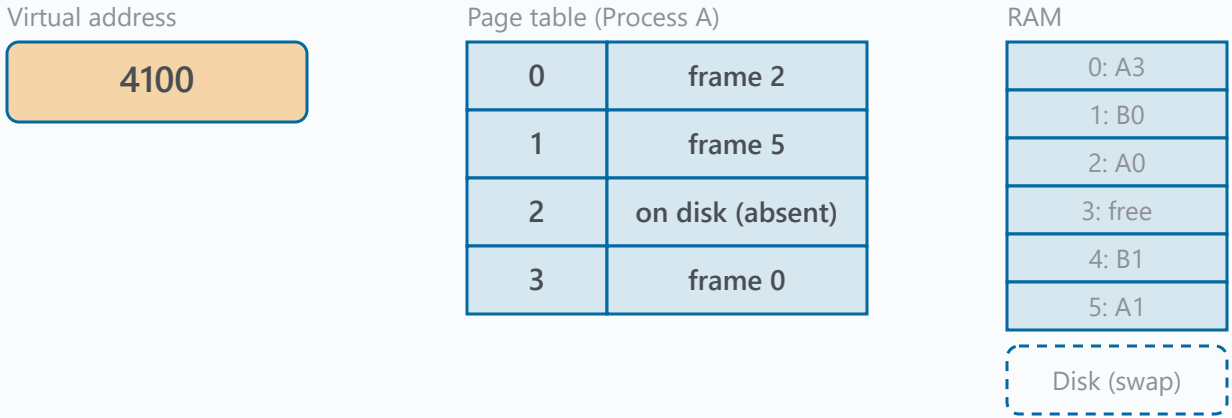
C is loaded at 300–550. B waits on disk and is brought back into RAM when it runs again; if there is no room, some other process is swapped out.

Virtual Memory and Paging



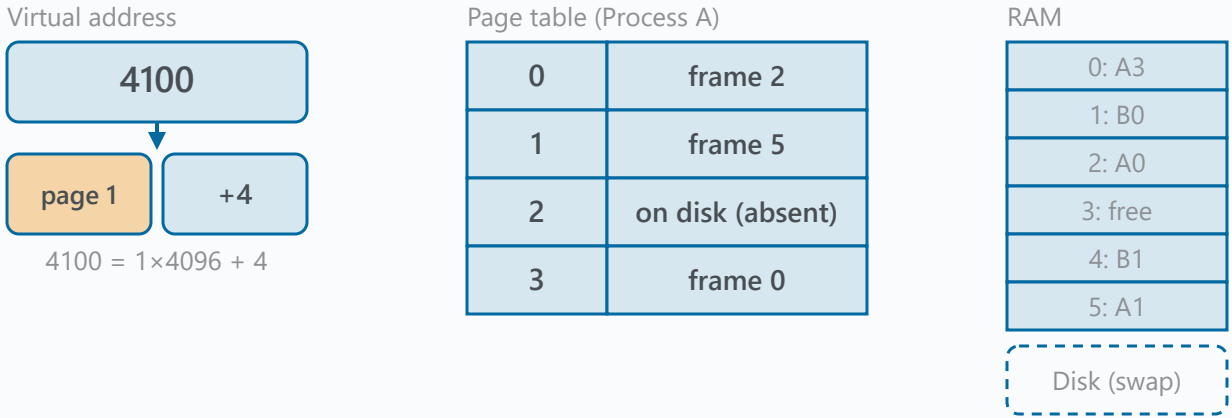
- Each process sees its own **virtual address** space starting at 0; memory is split into 4 KB **pages**, and a page table records which **frame** holds each page.
- Pages need not be contiguous in RAM; rarely used pages can be moved out to **disk** (swapped).
- The swap area is called the **page file** on Windows and **swap** on Linux and macOS. Reading from disk (SSD ≈ 0.1 ms) is orders of magnitude slower than RAM (50–100 ns); heavy swapping makes a computer noticeably slow.

Address Translation Step by Step: 4100 and 8200



Process A wants to read the data at virtual address **4100**. The CPU's memory management unit (MMU) has to translate it.

Address Translation Step by Step: 4100 and 8200



The address splits in two: **page = 4100 / 4096 = 1**, **offset = 4100 mod 4096 = 4**.

Address Translation Step by Step: 4100 and 8200

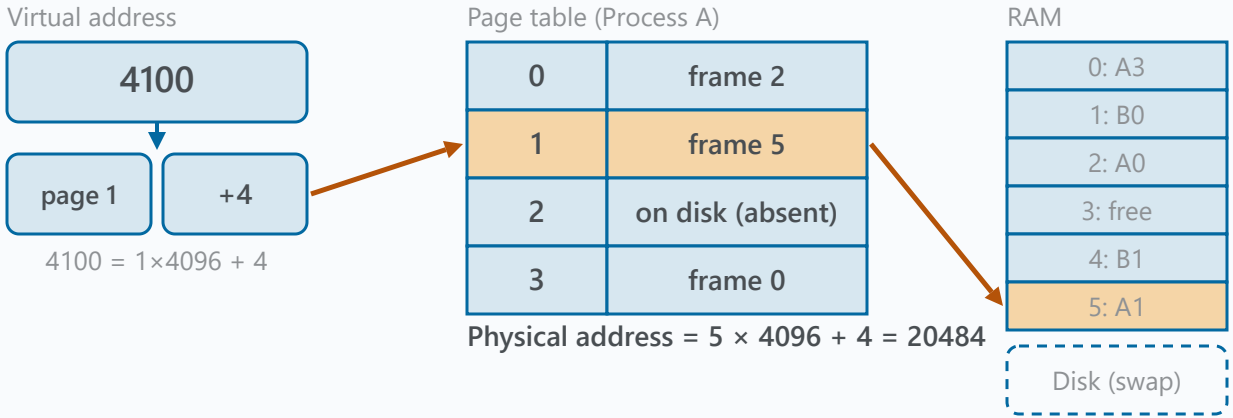
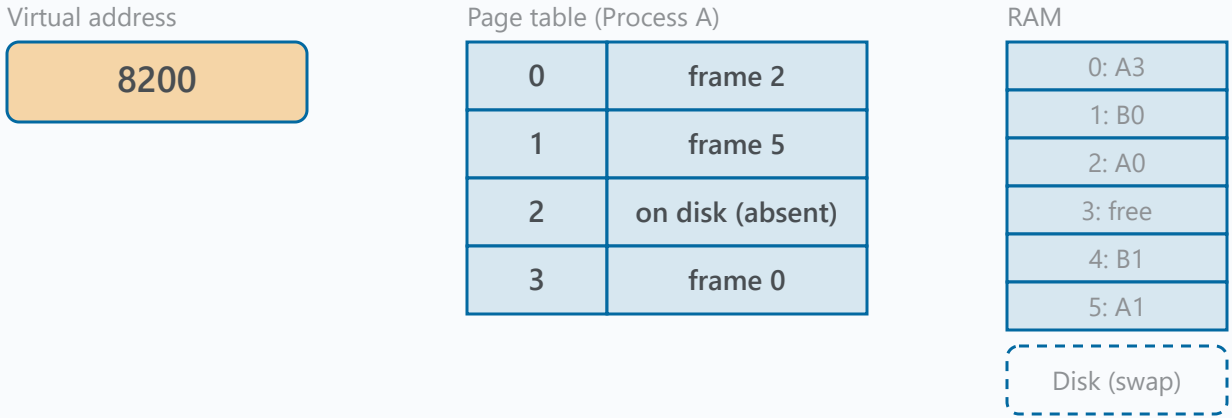


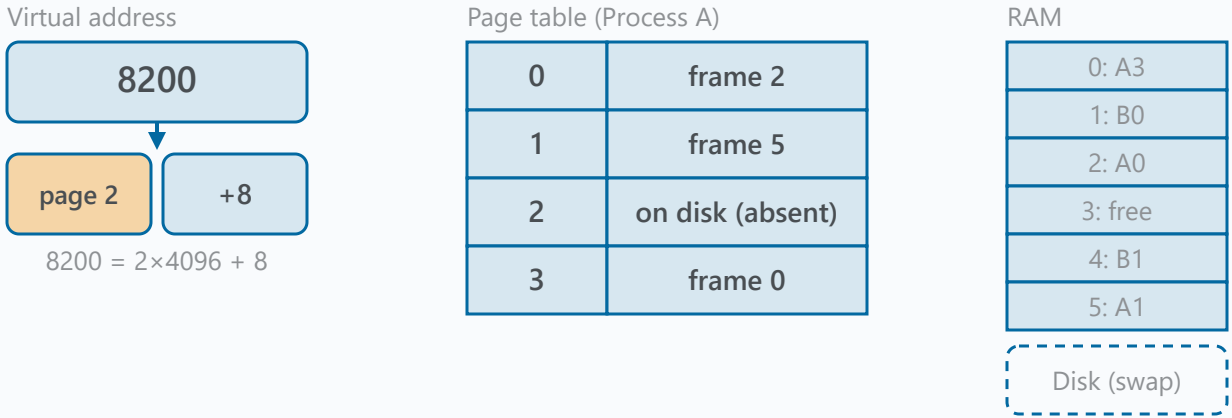
Table: page 1 → frame 5. **Physical address = 5 × 4096 + 4 = 20484**.

Address Translation Step by Step: 4100 and 8200



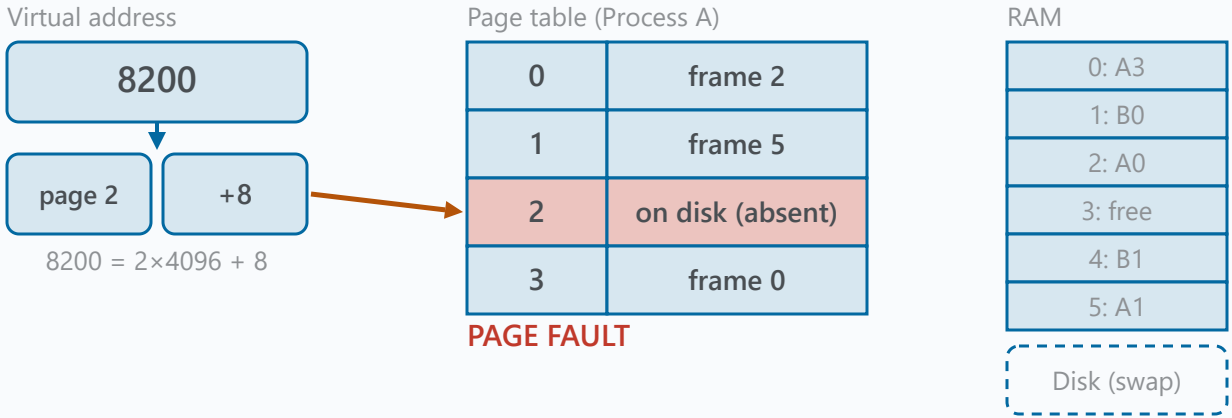
Process A wants to read the data at virtual address **8200**. The CPU's memory management unit (MMU) has to translate it.

Address Translation Step by Step: 4100 and 8200



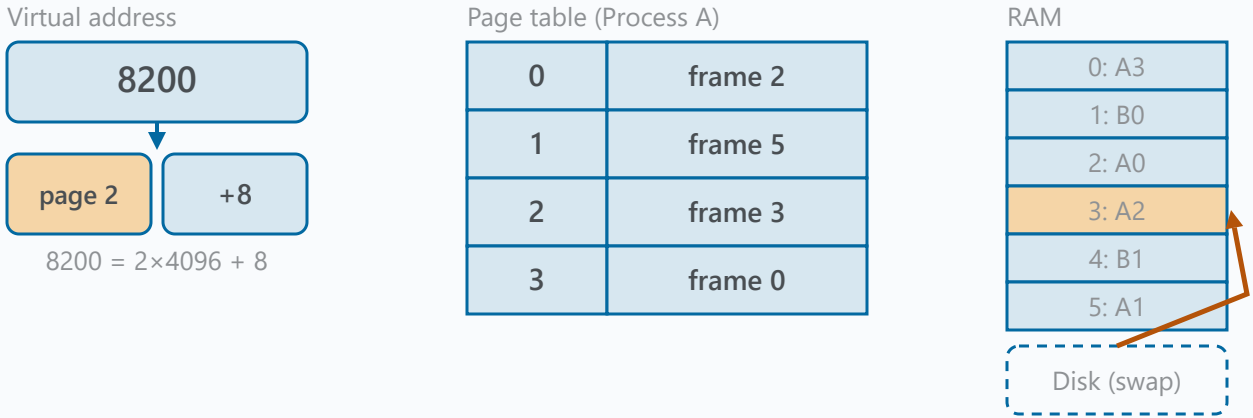
The address splits in two: **page = 8200 / 4096 = 2**, **offset = 8200 mod 4096 = 8**.

Address Translation Step by Step: 4100 and 8200



The table says page 2 is **not in RAM** → page fault. The program is paused and the OS steps in.

Address Translation Step by Step: 4100 and 8200



The OS loads the page from disk into free **frame 3** and updates the page table.

Address Translation Step by Step: 4100 and 8200

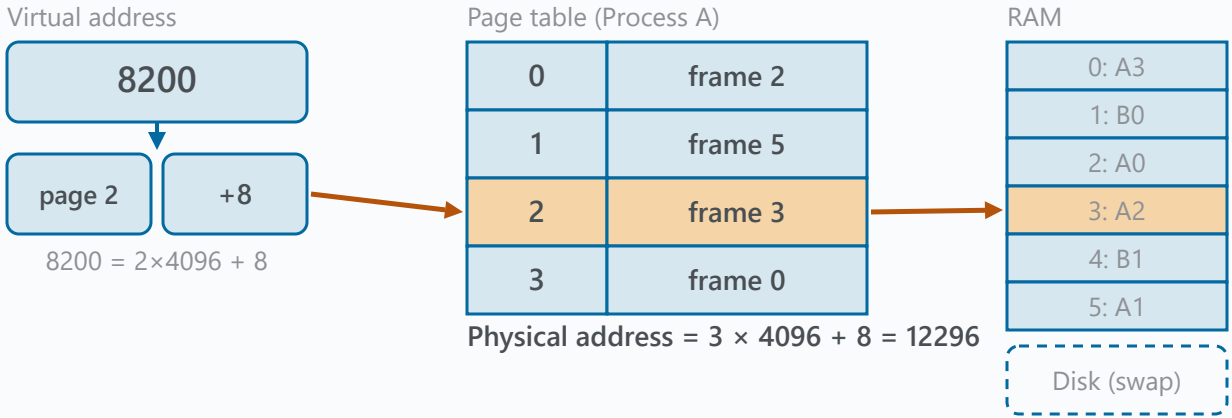
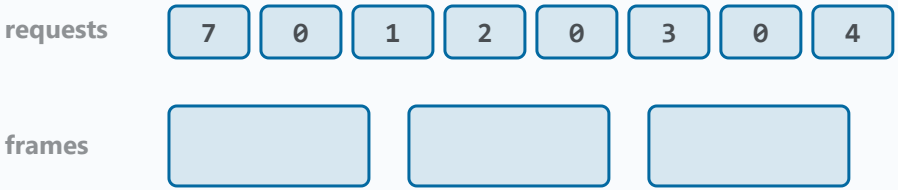


Table: page 2 → frame 3. **Physical address = 3 × 4096 + 8 = 12296**. The instruction is restarted and succeeds this time.

Page Replacement (LRU) Step by Step



The process has only **3 frames** in RAM and requests pages in the order 7, 0, 1, 2, 0, 3, 0, 4. When RAM is full, which page goes to disk? The **LRU** (least recently used) rule.

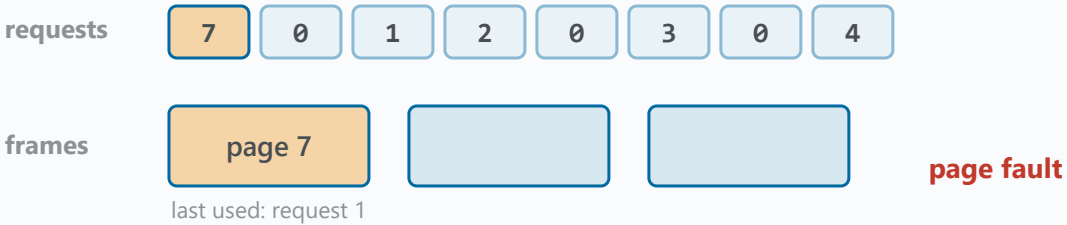
Step 1 / 10

```
page in RAM?  Yes → hit
No → page fault
  free frame?  Yes → load
  No → evict least recent, load
```

VARIABLES

`request = –` `frames = empty` `faults = 0`

Page Replacement (LRU) Step by Step



Page 7 is not in RAM → **page fault**; there is a free frame, so it is loaded there from disk.

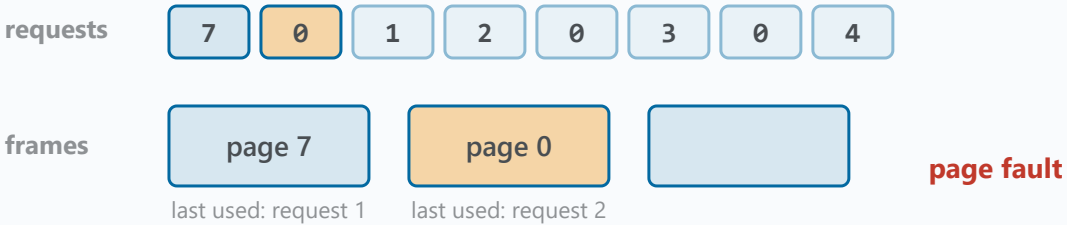
Step 2 / 10

```
1 page in RAM?  Yes → hit
2 No → page fault
▶   free frame?  Yes → load
4   No → evict least recent, load
```

VARIABLES

`request = 7` `frames = 7` `faults = 1`

Page Replacement (LRU) Step by Step



Page 0 is not in RAM → **page fault**; there is a free frame, so it is loaded there from disk.

Step 3 / 10

```
1 page in RAM?  Yes → hit
2 No → page fault
▶   free frame?  Yes → load
4   No → evict least recent, load
```

VARIABLES

`request = 0` `frames = 7, 0` `faults = 2`

Page Replacement (LRU) Step by Step



Page 1 is not in RAM → **page fault**; there is a free frame, so it is loaded there from disk.

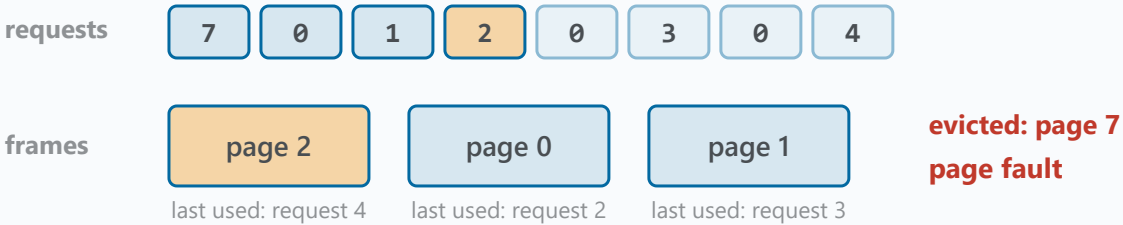
Step 4 / 10

```
1 page in RAM?  Yes → hit
2 No → page fault
▶   free frame?  Yes → load
4   No → evict least recent, load
```

VARIABLES

`request = 1` `frames = 7, 0, 1` `faults = 3`

Page Replacement (LRU) Step by Step



Page 2 is missing → **page fault**; no free frame. Page 7 was used longest ago → evict it, load 2.

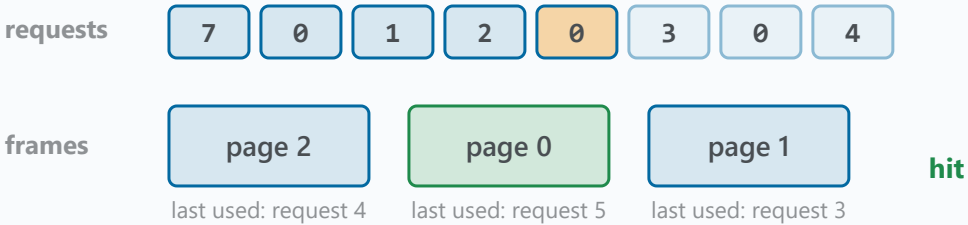
Step 5 / 10

```
1 page in RAM?  Yes → hit
2 No → page fault
3   free frame?  Yes → load
▶   No → evict least recent, load
```

VARIABLES

`request = 2` `frames = 2, 0, 1` `faults = 4`

Page Replacement (LRU) Step by Step



Page 0 is already in RAM → **hit**; only its last-use time is updated.

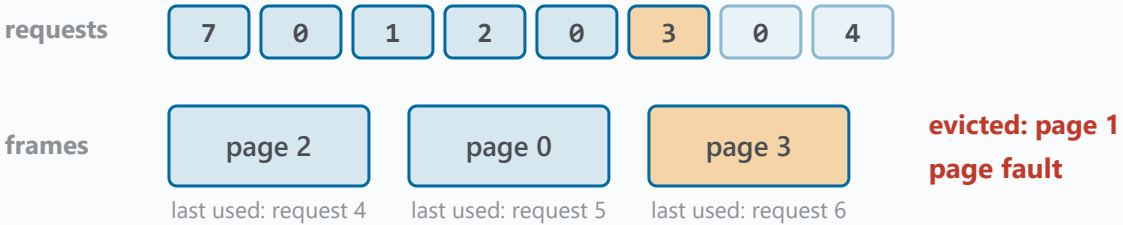
Step 6 / 10

```
▶ page in RAM?  Yes → hit
2 No → page fault
3   free frame?  Yes → load
4   No → evict least recent, load
```

VARIABLES

`request = 0` `frames = 2, 0, 1` `faults = 4`

Page Replacement (LRU) Step by Step



Page 3 is missing → **page fault**; no free frame. Page 1 was used longest ago → evict it, load 3.

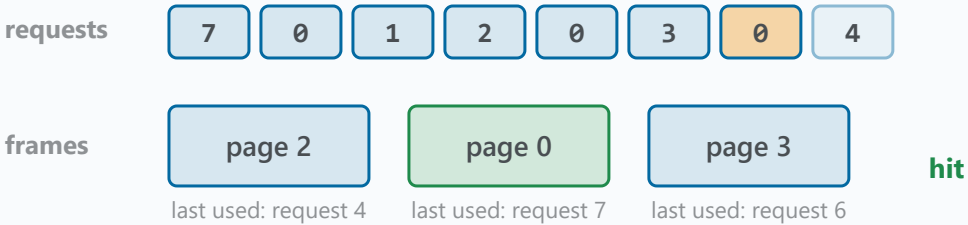
Step 7 / 10

```
1 page in RAM?  Yes → hit
2 No → page fault
3   free frame?  Yes → load
▶   No → evict least recent, load
```

VARIABLES

`request = 3` `frames = 2, 0, 3` `faults = 5`

Page Replacement (LRU) Step by Step



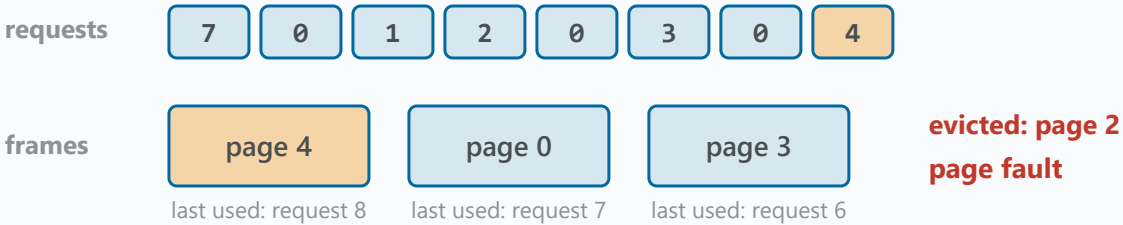
Page 0 is already in RAM → **hit**; only its last-use time is updated.


```
▶ page in RAM?  Yes → hit
2 No → page fault
3   free frame?  Yes → load
4   No → evict least recent, load
```

VARIABLES

`request = 0` `frames = 2, 0, 3` `faults = 5`

Page Replacement (LRU) Step by Step



Page 4 is missing → **page fault**; no free frame. Page 2 was used longest ago → evict it, load 4.

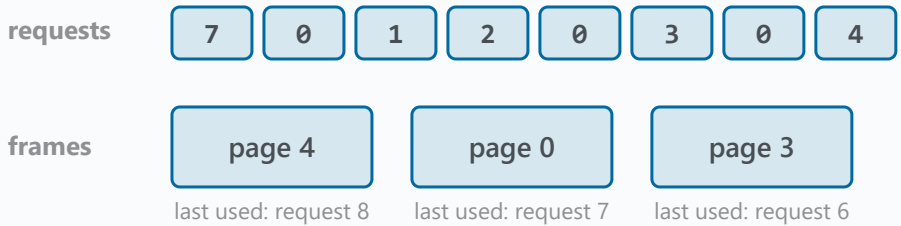
Step 9 / 10

```
1 page in RAM?  Yes → hit
2 No → page fault
3   free frame?  Yes → load
▶   No → evict least recent, load
```

VARIABLES

`request = 4` `frames = 4, 0, 3` `faults = 6`

Page Replacement (LRU) Step by Step



6 page faults in 8 requests. FIFO (first in, first out) gives 7 faults on the same string: it evicts the busy page 0 too when its turn comes. Every fault means a slow disk read.

Step 10 / 10

```
page in RAM?  Yes → hit
No → page fault
  free frame?  Yes → load
  No → evict least recent, load
```

VARIABLES

request = –

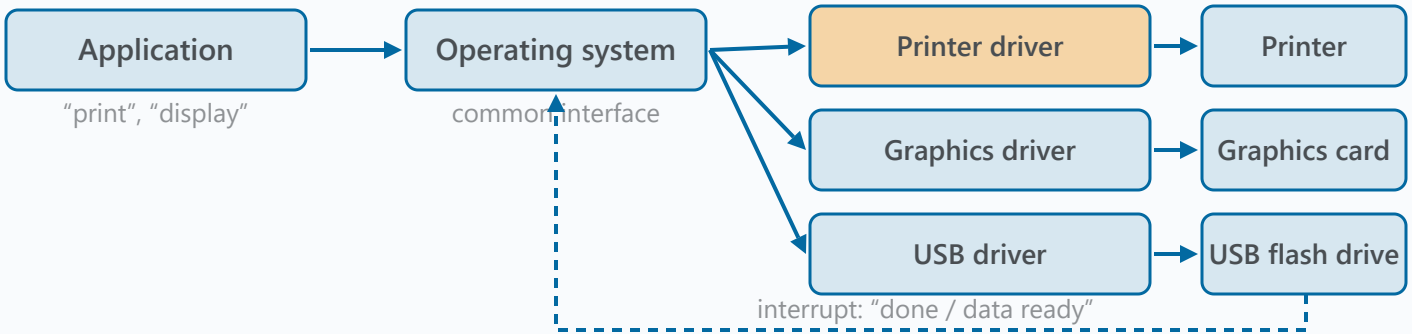
frames = 4, 0, 3

faults = 6

RESOURCE MANAGEMENT

Devices and Drivers

Hardware devices differ in their commands and features. A device driver translates the OS's generic requests into device-specific operations.



Plug and play

Detects a compatible device and sets up the right driver.

Interrupt

A device signals an event to the CPU.

Buffer

Holds data temporarily between parts running at different speeds.

RESOURCE MANAGEMENT

File System

A file system defines how data on a storage device is named, organized and found.

File

A unit of data with a name and contents.

Folder

Groups files and subfolders.

Path

Gives a file's location in the folder tree.

Metadata

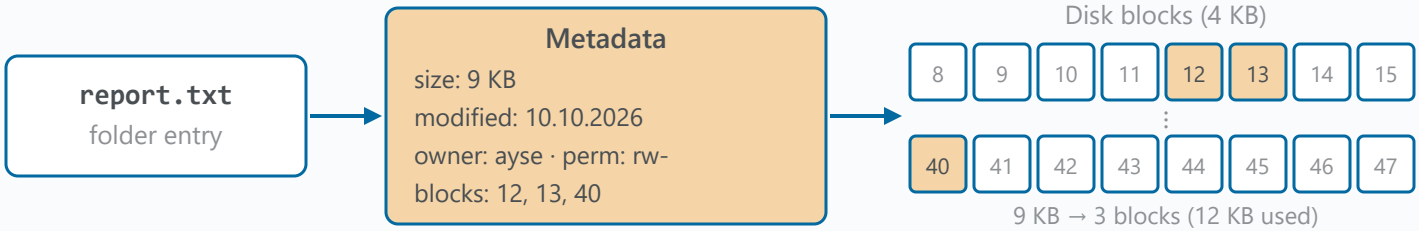
Properties such as size, date, owner and permissions.

Permissions

Read, write and execute rights.

Journaling

Helps keep things consistent after a crash or power loss.

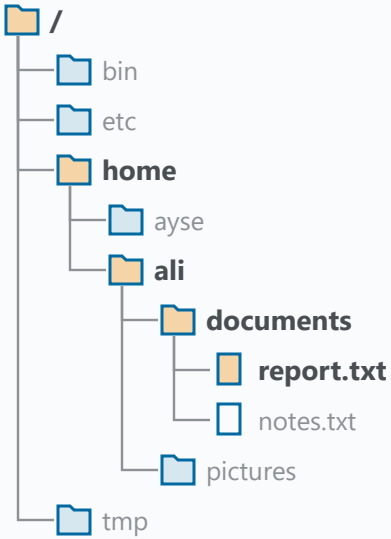


Kinds of File Systems

File system	Platform	Journaling	When to use it
NTFS	Windows (macOS read-only)	Yes	Windows system drive; when you need large files, permissions or encryption
FAT32	Almost every device and OS	No	Older devices, cameras, simple memory cards; **4 GB maximum file size**
exFAT	Windows, macOS and recent Linux	No	USB drives and SD cards with large files; moving data across platforms
ext4	Linux	Yes	Linux system and data disks
APFS	macOS, iOS, iPadOS	No classic journal; copy-on-write keeps it consistent	Default file system on Apple devices

Example: a 5 GB video cannot be copied to a FAT32 USB drive; the drive must be formatted as exFAT or NTFS (formatting erases its contents).

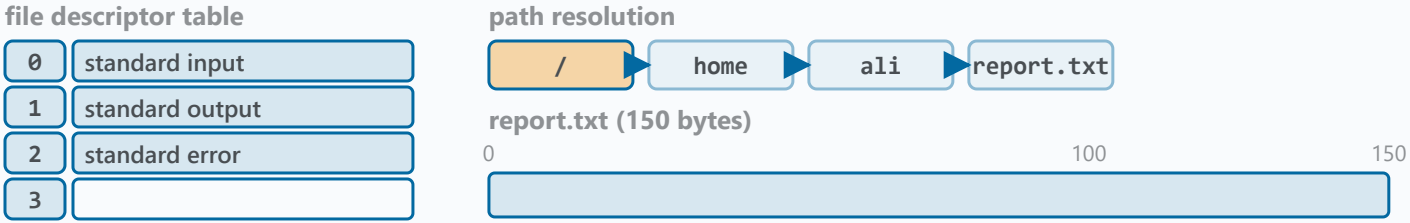
The File System Tree and Paths




```
/home/ali/documents/report.txt
```

- A file system is a tree: the **root** at the top, folders as branches, files as leaves.
- An **absolute path** starts at the root; a **relative path** starts at the current folder (from ali: `documents/report.txt`).
- Windows writes the same idea with a drive letter: `C:\Users\Ali\Documents\report.txt`.

Opening and Reading a File Step by Step



The editor makes the **open** system call. The kernel resolves the path one piece at a time, starting at the **root** folder /.

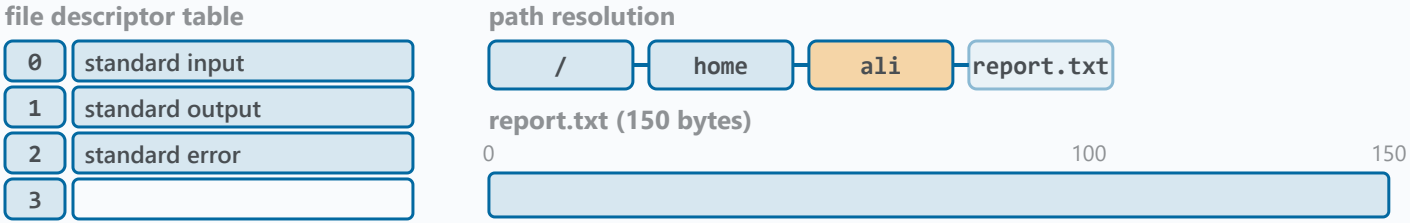
Step 1 / 8

```
▶ fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

fd = – offset = – n = –

Opening and Reading a File Step by Step



It looks for **home** in the root, then for **ali** inside it. Each folder is a list of the names it contains and where they live on disk.

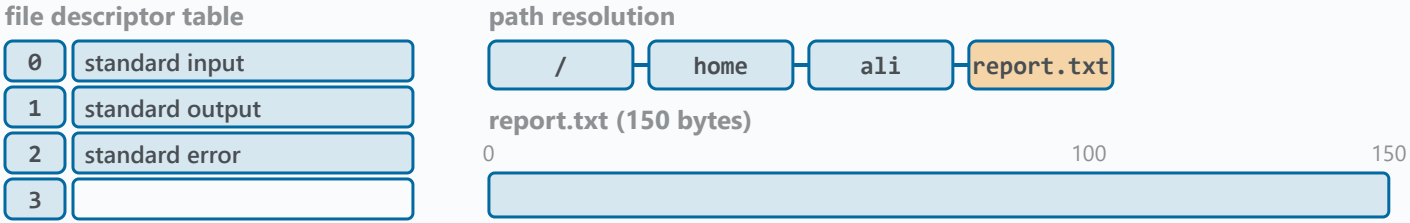
Step 2 / 8

```
▶ fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

`fd = –` `offset = –` `n = –`

Opening and Reading a File Step by Step



report.txt is found in ali. The metadata says: size 150 bytes; does the user have **read permission**? Yes. (Otherwise open returns an error and the file stays closed.)

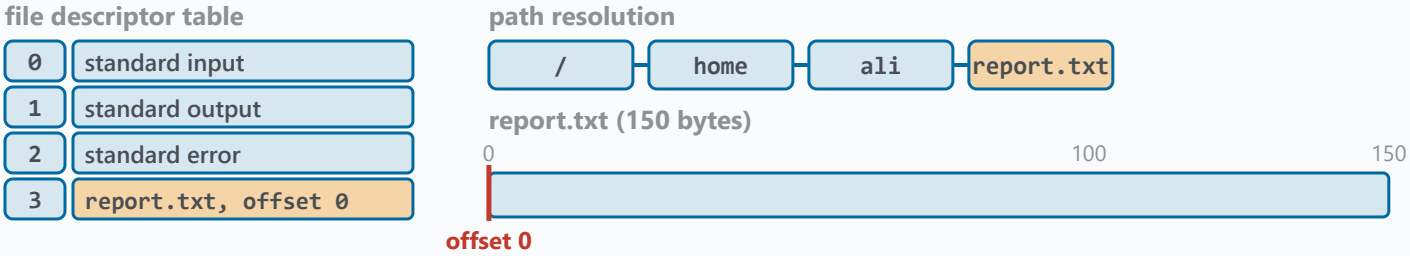
Step 3 / 8

```
▶ fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

`fd = –` `offset = –` `n = –`

Opening and Reading a File Step by Step



The kernel creates an open-file entry and gives the process the lowest free number: **fd = 3**. Numbers 0, 1 and 2 are always open in every process for standard input, output and error.

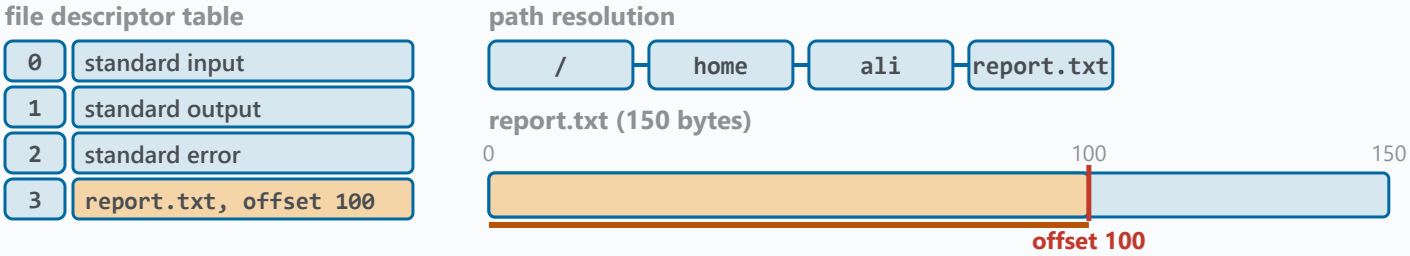
Step 4 / 8

```
▶ fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

`fd = 3` `offset = 0` `n = –`

Opening and Reading a File Step by Step



read asks for 100 bytes; bytes 0–99 are read, **n = 100**. The offset moves to 100 on its own; the application never sees addresses, only the fd.

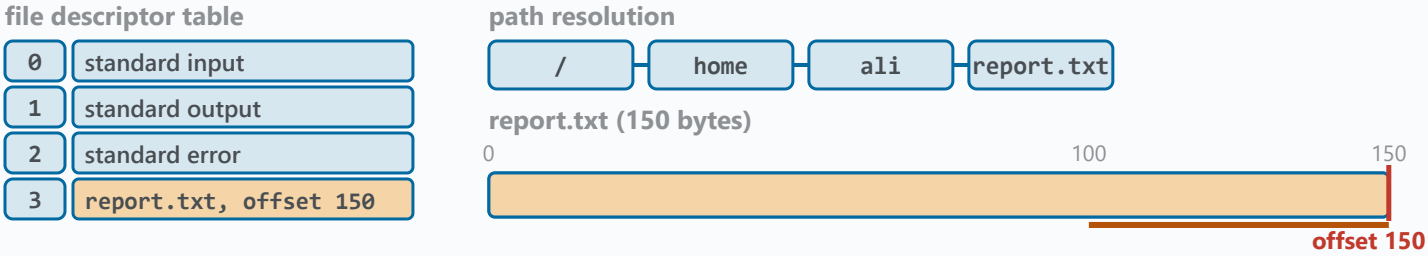
Step 5 / 8

```
1 fd = open("/home/ali/report.txt", O_RDONLY)
▶ n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

`fd = 3` `offset = 100` `n = 100`

Opening and Reading a File Step by Step



Again 100 bytes are requested, but only 50 are left: **n = 50**, offset 150.

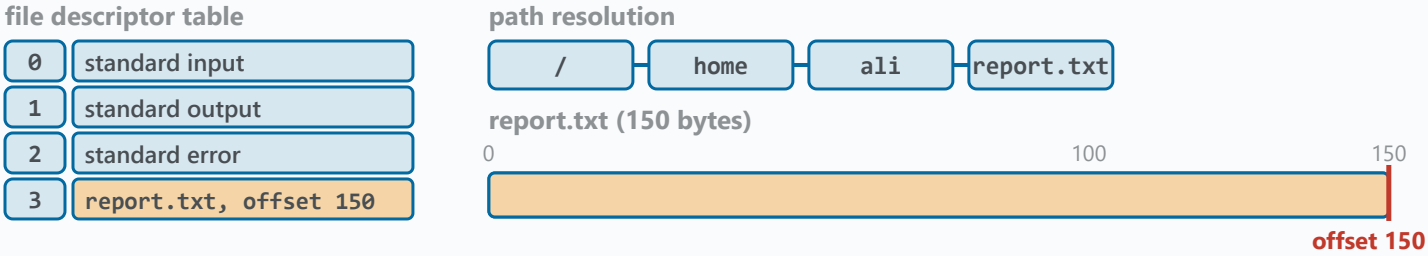
Step 6 / 8

```
1 fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
▶ n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

`fd = 3` `offset = 150` `n = 50`

Opening and Reading a File Step by Step



The third read returns **n = 0**: end of file. The program stops reading here.

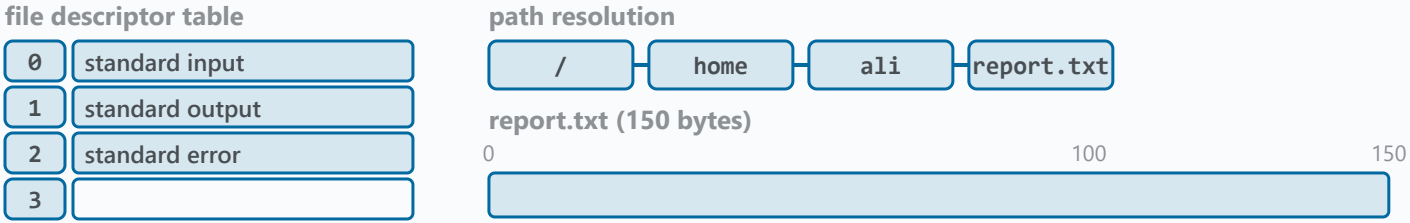
Step 7 / 8

```
1 fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
▶ n = read(fd, buf, 100)
5 close(fd)
```

VARIABLES

`fd = 3` `offset = 150` `n = 0`

Opening and Reading a File Step by Step



close: the entry is removed and number 3 is free again. The OS closes any files left open when the process ends, but a long-running program can run out of descriptors. Windows calls the same idea a handle.

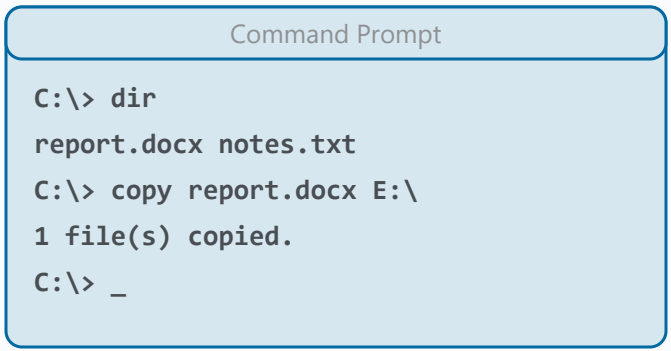
Step 8 / 8

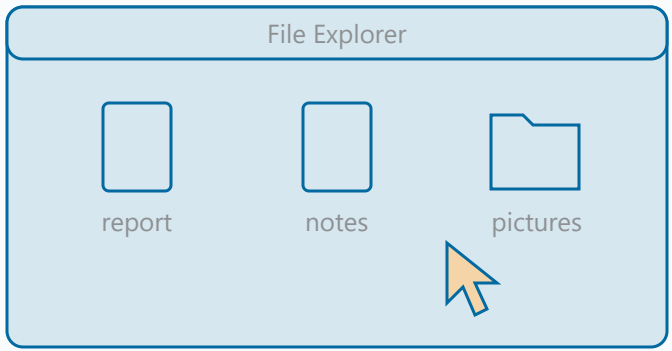
```
1 fd = open("/home/ali/report.txt", O_RDONLY)
2 n = read(fd, buf, 100)
3 n = read(fd, buf, 100)
4 n = read(fd, buf, 100)
▶ close(fd)
```

VARIABLES

`fd = –` `offset = –` `n = 0`

User Interfaces





GUI (graphical user interface)

Windows, icons, menus and a mouse.

CLI

Text commands: fast, repeatable, easy to automate.

Touch interface

Taps, swipes and gestures.

Accessibility

Screen reader, magnifier, high contrast.

Command-Line Basics

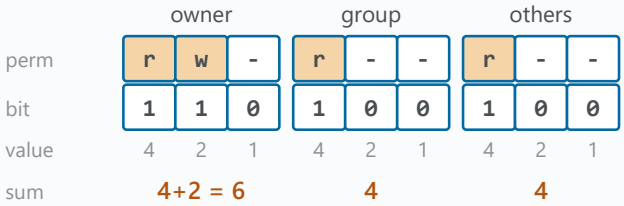
Task	Linux / macOS	Windows (cmd)
Show current folder	`pwd`	`cd` (no arguments)
List folder contents	`ls`	`dir`
Change folder	`cd folder`	`cd folder`
Create folder	`mkdir new`	`mkdir new`
Copy file	`cp a.txt b.txt`	`copy a.txt b.txt`
Delete file	`rm a.txt`	`del a.txt`

chmod permission format

For each of three groups, add **r**=4 (read), **w**=2 (write), **x**=1 (execute): owner · group · others.

`chmod 644 file`

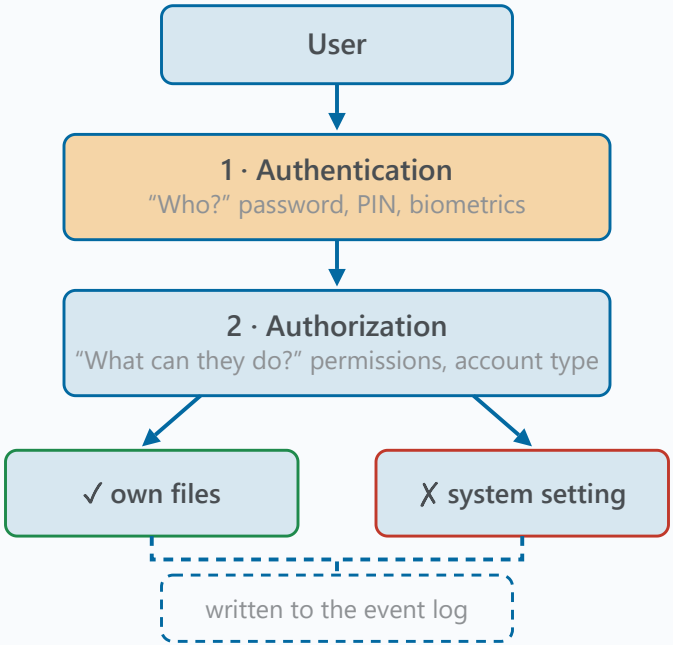
6 = rw- (owner) · 4 = r-- (group) · 4 = r-- (others)



`rm` and `del` don't send files to the Recycle Bin; check the path and name before you run them.

SECURITY

Users and Permissions



- A **standard account** limits system changes in everyday use; an **administrator account** can make system-wide changes.
- **Least privilege:** give users only the rights their job needs.
- **Logging** lets important events be reviewed later.
- In the example, a standard user can open their own file but cannot change a system setting.

Personal Computer Systems

Windows

Huge hardware and software ecosystem.

macOS

Desktop system integrated with Apple hardware.

Linux distributions

Systems tailored to different desktops and uses.

ChromeOS

Built around web and cloud use.

Mobile Operating Systems

Android

A broad mobile ecosystem running on many makers' devices.

iOS / iPadOS

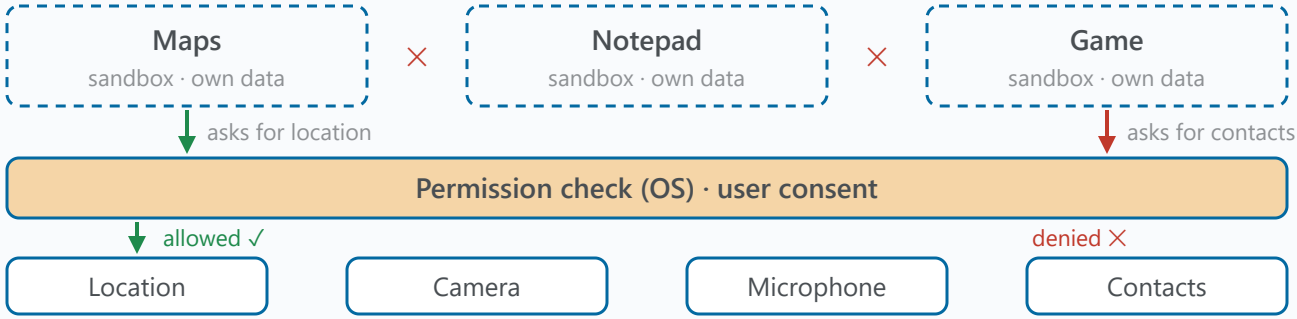
Integrated system for Apple mobile devices.

Mobile priorities

Touch interface, power management, sensors and wireless connectivity.

App isolation

Limits what apps can reach with permissions and sandboxes.



Server Operating Systems

Multi-user operation

Serves many users and services at once.

Network services

Provides web, database, file and identity services.

Availability

Built to run for long periods without interruption and be managed remotely.

Security

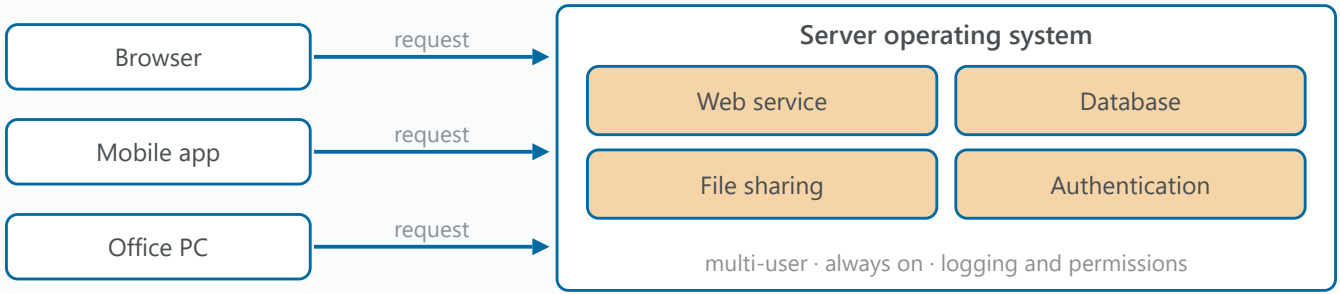
Permissions, logging, updates and network control come first.

Examples

Linux distributions, Windows Server and Unix variants.

Automation

Repeatable administration with the command line and configuration tools.



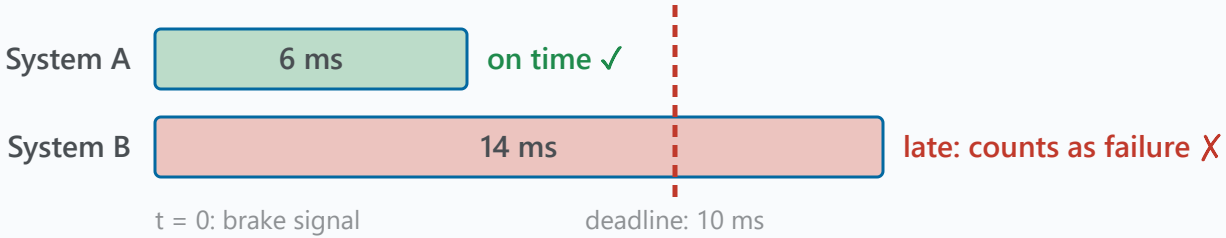
Embedded and Real-Time Systems

Embedded operating system

Runs a specific device task with limited CPU, memory and power.
Examples: router, car control unit, medical device.

Real-time operating system

Must deliver the right result within a set deadline.
Examples: industrial control and safety systems.



Choosing an Operating System

Requirement	What to evaluate
Application	Whether the required software and versions are available
Hardware	Driver and architecture support
Security	Update lifetime, permission model and management tools
Usability	Interface, accessibility and learning curve
Cost	Licensing, support, training and maintenance combined

LICENSING

Open-Source Software

Source code

Can be studied under the license terms.

Modification

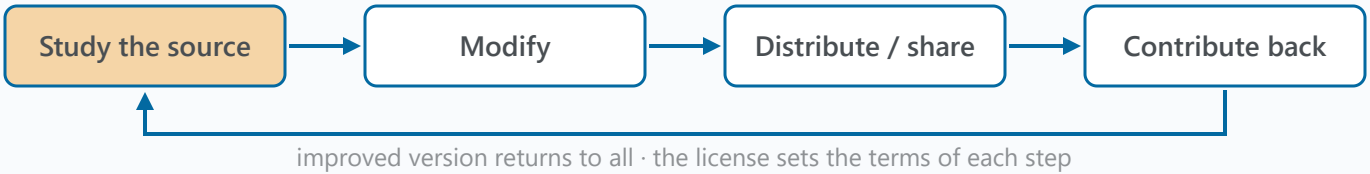
Can be adapted to your needs if the license allows.

Distribution

Sharing original or modified versions depends on the license.

Community

Development and support can come from individuals, institutions and companies.



Open source doesn't automatically mean free of charge or unsupported. The license defines the rights and obligations.

Commercial and Closed-Source Software

Right of use

You don't buy the software itself; usually you get a license to use it under certain terms.

Source code

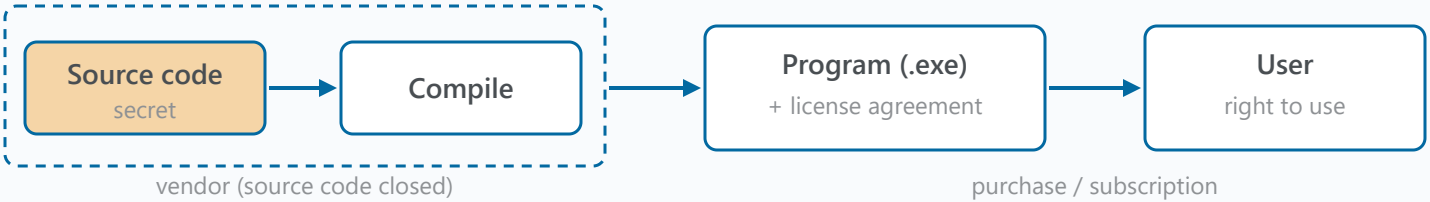
Usually kept closed by the developer.

Support

Official support and service-level agreements may be offered.

Cost

There may be license, subscription, maintenance and upgrade fees.



LICENSING

Comparing License Models

Criterion	Open source	Proprietary (closed source)
Source access	Open under the license	Usually closed
Customization	Possible within the license	Limited to the vendor's options
Support	Community or paid provider	Vendor or authorized partner
Cost	License may be free; running costs remain	License or subscription may be required
Example license	GPL, MIT, Apache	End-user license agreement (EULA)
Example product	Linux, Firefox, LibreOffice	Windows, Microsoft Office, Adobe Photoshop

Not every free program is open source; the license terms are what count. Commercial and open source are separate dimensions: commercial products such as Red Hat Enterprise Linux can be open source too.

MAINTENANCE

Updates and Support

Security update

Closes known vulnerabilities.

Feature update

Brings new functions and interface changes.

Driver update

Can improve hardware compatibility or stability.

Support lifecycle

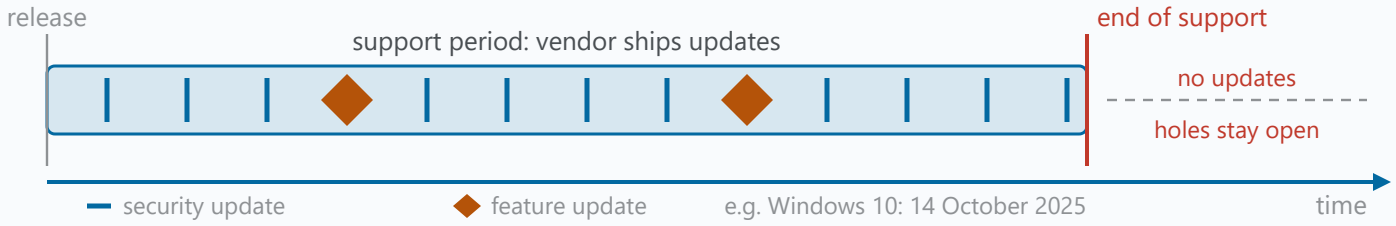
The period during which the vendor ships updates.

Backup

Makes major changes reversible.

Testing

In organizations, updates are tested before wide rollout.



Virtualization and Containers

Virtual machine

Runs isolated operating system instances on one physical computer.

Hypervisor

Gives virtual machines CPU, memory and device resources.

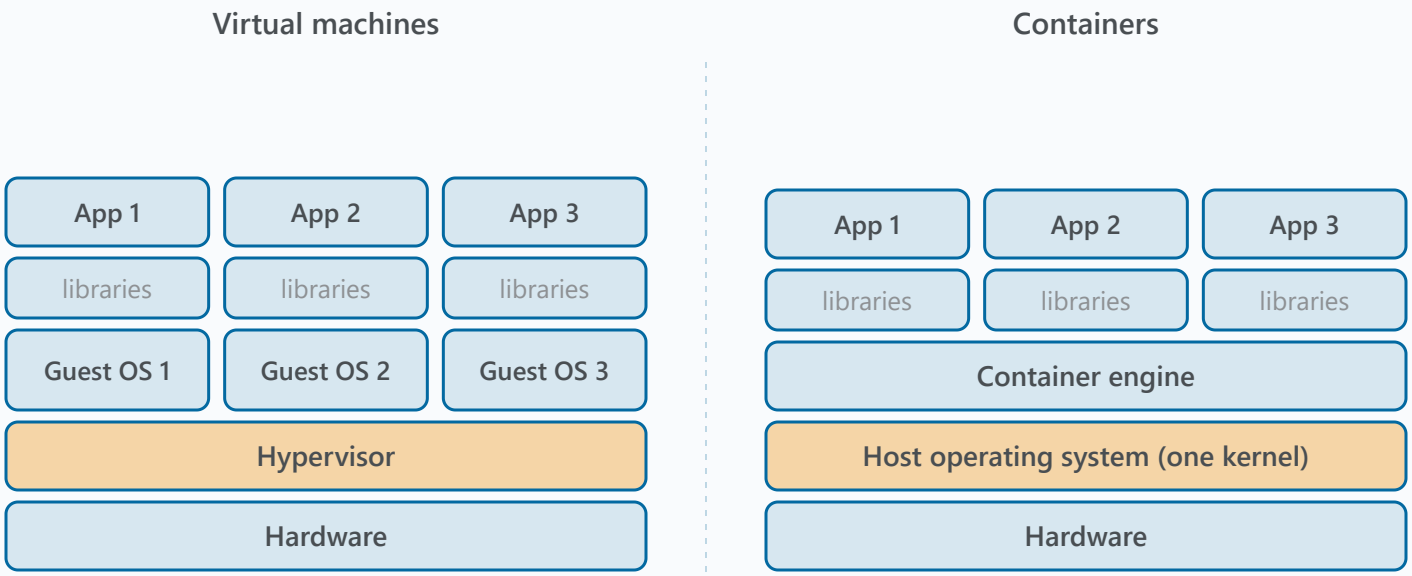
Container

Isolates applications and their dependencies at the process level; shares the kernel.

Uses

Testing, server consolidation, cloud and portable app deployment.

Virtual Machines vs Containers



Virtual machine

Carries a full guest OS: strong isolation, but more memory and slower startup.

Container

Just the app and its dependencies on a shared kernel: light and fast, weaker isolation.

SUMMARY

Chapter Summary

- Software defines the work the hardware carries out.
- The operating system is a management and abstraction layer between applications and hardware.
- The kernel manages process, memory, device and file resources.
- Personal, mobile, server, embedded and real-time systems have different priorities.
- Choosing an OS means weighing applications, hardware, security, support and total cost together.
- The license sets the rights for open and closed source.

Review Questions

1. What is the difference between system software and application software?
2. Why does the kernel run in privileged mode?
3. How do a program and a process differ?
4. What problem does virtual memory solve?
5. How do the priorities of server and mobile operating systems differ?
6. Why isn't open-source software always free of charge?

Exercises

1. Round-robin scheduling, time slice 2 ms: **P1 = 4, P2 = 2, P3 = 3 ms**. Draw the Gantt chart and find each process's completion time.
2. Page size is 4096 bytes. Which page and offset does virtual address **12300** fall in? If page 3 is in frame 1, what is the physical address?
3. What state is a process in while it waits for keyboard input? Which state does it move to when a key is pressed, and then which? Show the transitions on the diagram.
4. Why won't a modified boot loader run when Secure Boot is on? Explain using the signature checks in the chain of trust.
5. For a test setup that needs both Windows and Linux, would you choose virtual machines or containers? Explain in terms of kernel sharing.
6. Walk through an application writing to a file, step by step: system call, kernel mode, file system, disk driver.

References

- Silberschatz, Galvin and Gagne, *Operating System Concepts*.
- Tanenbaum and Bos, *Modern Operating Systems*.
- License resources from The Linux Foundation and the Free Software Foundation.

The End

Chapter 2.1: Software and Operating Systems

Previous: Chapter 1.5: Computation and Computational Thinking · Next: Chapter 2.2: The Windows Operating System