IN ALIGNMENT WITH NEP 2020

OPERATING SYSTEM

BCA | Semester II | Academic Session 2025–2028

Subject code: BCA202

Compiled by

Deepak Kumar Tiwari

Asst. Professor, Dept. of BCA RVSCET,
Jamshedpur

INDEX

Operating Systems — BCA Second Semester

By- Deepak Kumar Tiwari

UNIT 1

UNIT 2

UNIT 3

UNIT 4

UNIT 5

UNIT 6

UNIT 1

Introduction and System Structures

Contents

1. What is an Operating System?

An Operating System (OS) is a software program that acts as an intermediary between the user and the computer hardware. It is the most important software on any computer — without it, your laptop or phone would be just an expensive piece of plastic and silicon that nobody could use.

Easy Explanation

Think of the Operating System as the manager of a big company. The hardware is like the workers — the CPU does calculations, RAM stores the work in progress, the disk stores files, the printer prints, and so on. You, the user, are the customer giving orders. You do not talk to the workers directly. You give your order to the manager, and the manager decides which worker should do what, in what order, and gives you the result.

When you double-click on Chrome, you are giving an order to the OS. The OS finds the Chrome program on the disk, loads it into RAM, gives it CPU time to run, connects it to your screen, keyboard, and the network — and shows you a working browser.

Two Main Goals of an Operating System

- **Convenience —** make the computer easy to use. You should be able to use a computer without knowing how the CPU executes machine instructions or how the hard disk stores data. The OS hides all this complexity behind simple icons and commands.

- **Efficiency** — use hardware resources (CPU, memory, disk) in the best possible way so that nothing sits idle. Even when you are reading this on screen, dozens of background programs are running and the OS makes sure they all share the CPU and memory smoothly.

Real-World Examples

The most popular operating systems today are Windows 11 (made by Microsoft, used on most personal computers), macOS (made by Apple, used on MacBooks and iMacs), Ubuntu Linux (free and open-source, used on many servers and developer machines), Android (used on most smartphones), iOS (used on iPhones and iPads), and Chrome OS (used on Chromebooks).

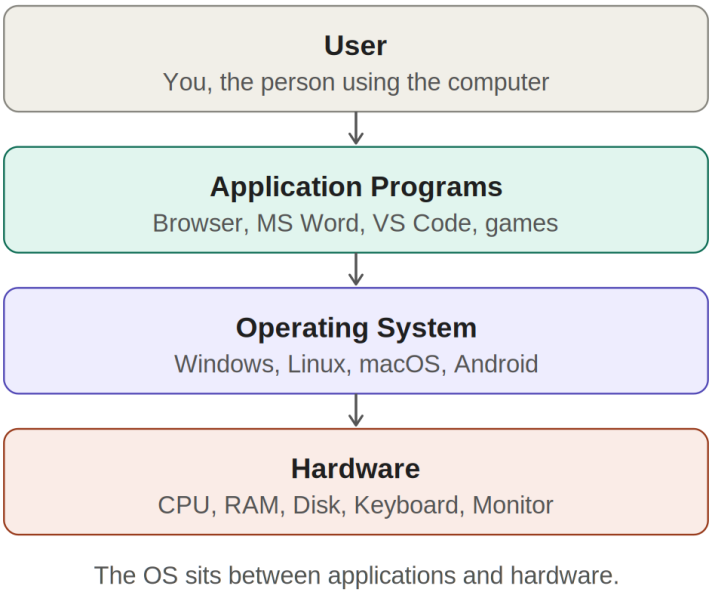


Figure 1.1 — The OS sits between applications and hardware. Every request from your apps reaches the hardware only through the OS.

2. Computer System Organization and Architecture

A modern computer system has one or more CPUs and a number of device controllers (small chips that manage specific devices), all connected through a common system bus that gives them access to shared main memory.

Key Components

- **CPU (Central Processing Unit)** — the brain of the computer. It executes the instructions of every program. Modern CPUs have multiple cores so they can execute several instructions in parallel.
- **Main memory (RAM)** — the temporary workspace of the computer. Whatever program is currently running, including the OS itself, lives here. RAM is fast but loses everything when power is cut.
- **Device controllers** — small chips on the motherboard that manage specific devices. The disk controller talks to the hard drive, the USB controller handles your keyboard and mouse, the graphics controller drives the monitor, and so on.

- **System bus** — the highway of the computer. It carries data between the CPU, the memory, and all the device controllers.

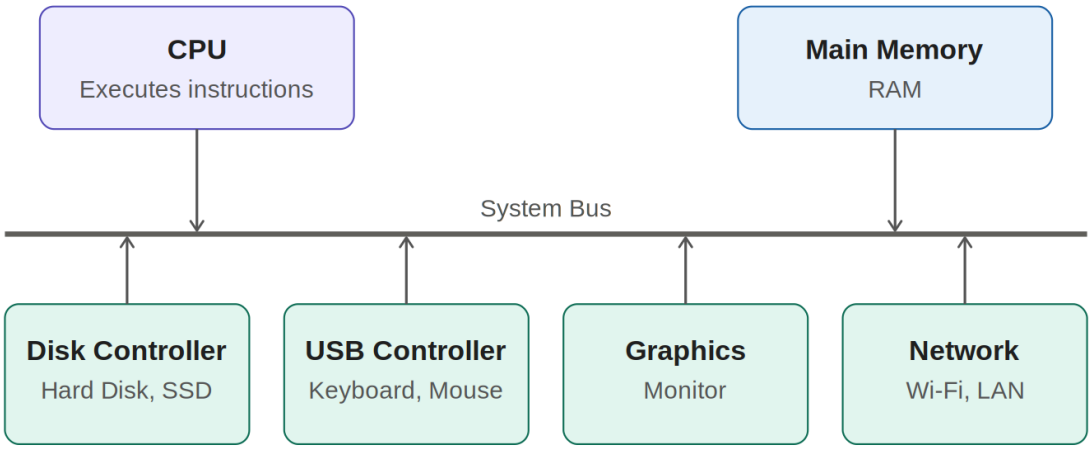


Figure 1.2 — Computer System Organization

How Devices Talk to the CPU — Interrupts

Imagine you are studying at your desk and the doorbell rings. You pause your studying, go answer the door, and then come back and resume where you left off. That is exactly how an interrupt works.

When a device — say, the keyboard when you press a key — needs the CPU's attention, it sends a signal called an interrupt. The CPU stops whatever it was doing, jumps to a special routine called the Interrupt Service Routine (ISR) to handle the device, and then resumes its previous work.

This is much more efficient than the alternative, called polling, where the CPU keeps asking each device "do you have anything for me? do you have anything?" all the time. Polling wastes a huge number of CPU cycles. With interrupts, the CPU does useful work and only stops when actually needed.

Storage Hierarchy

Not all storage is equal. Computers use several layers of storage, each faster, smaller, and more expensive than the one below it. The CPU first looks for data in the fastest place. If the data is not there, it looks at the next slower level, and so on.

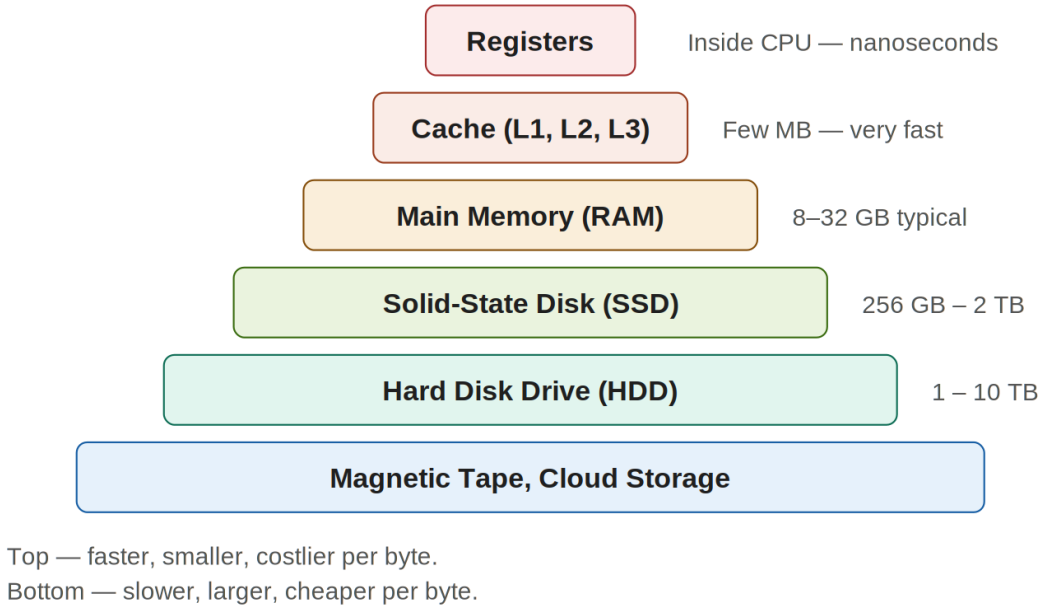


Figure 1.3 — The Storage Hierarchy

Important — Cache is the most important concept here. When you open the same file twice in a row, the second time is much faster because it is already in cache. This is why your computer feels faster the longer you use the same programs.

3. Operating System Structure and Operations

Multiprogramming and Multitasking

In the early days of computing, the CPU would sit idle whenever a program had to wait for I/O — for example, while reading a large file from disk. Engineers thought, "Why waste this time?" So they designed the OS to keep multiple programs in memory at the same time. When one program is waiting for I/O, the CPU switches to another program. This is called multiprogramming, and its main goal is to keep the CPU busy.

Multitasking, also called time-sharing, is an extension of this idea. The CPU switches between programs so rapidly — many times per second — that each user feels their program is running continuously. This is what lets you play music in Spotify, type a document in Word, browse the web in Chrome, and chat on WhatsApp Web all at the same time on the same machine.

Dual-Mode Operation (User Mode vs Kernel Mode)

This is one of the most important concepts in the entire course — pay close attention.

To protect the system from buggy or malicious programs, the CPU operates in two distinct modes. A small bit in the CPU, called the mode bit, keeps track of which mode the CPU is in at any moment.

- **User mode (mode bit = 1) —** normal application code runs here. In user mode, programs are restricted. They cannot directly touch hardware, change CPU settings, or access another program's memory.

- **Kernel mode (mode bit = 0)** — also called supervisor mode, system mode, or privileged mode. Only the OS code runs here. In kernel mode, the CPU can do anything: access hardware, modify any memory, change CPU settings.

Some CPU instructions, like "halt the CPU" or "disable all interrupts" or "directly read from a disk", are marked as privileged. They can only be executed in kernel mode. If a user program tries to execute such an instruction, the hardware traps it as an error and the OS terminates that program.

When a user program needs the OS to do something privileged — for example, to read a file from disk — it makes a system call. The system call switches the CPU to kernel mode, runs the OS code, and then switches back to user mode. This is the only legal way to enter kernel mode.

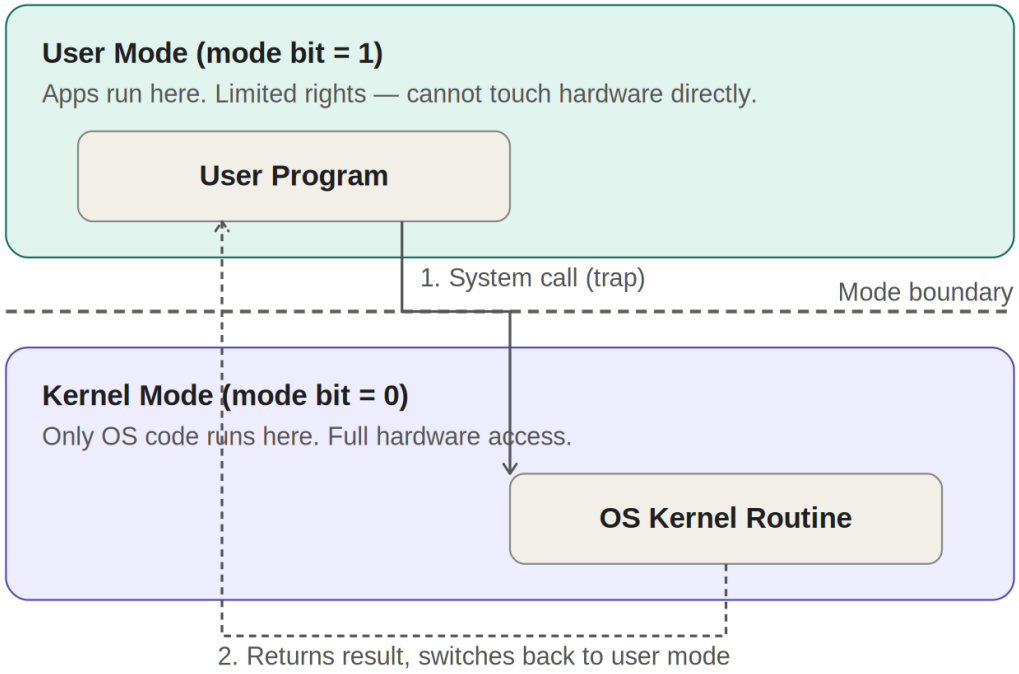


Figure 1.4 — User mode and Kernel mode. A system call is the only legal way to switch from one to the other.

The timer — There is also a hardware timer that interrupts the CPU at fixed intervals (say, every few milliseconds). This stops any user program from hogging the CPU forever. Once the timer fires, control returns to the OS, which decides what to run next.

4. Basics of Process, Memory, and Storage Management

The OS is essentially a giant resource manager. Its main jobs fall into a few buckets — process management, memory management, storage management, and I/O management.

Process Management

A process is a program in execution. There is a difference between a program and a process. A program is just a file on the disk (passive). A process is what happens when that file is loaded into memory and starts running (active).

Example: notepad.exe sitting in your Windows folder is a program. The moment you double-click and three Notepad windows open, you have three processes — all running the same program.

The OS handles creating processes, scheduling them on the CPU, suspending and resuming them, providing ways for them to communicate with each other, and deleting them when they finish. We will study this in detail in Unit 2.

Memory Management

RAM is limited and must be shared among many processes. The OS decides which processes get memory and how much, where in memory each process goes, and when to move processes in and out of memory.

Without good memory management, programs would constantly overwrite each other's data, and the system would crash within seconds.

Storage Management

Files on the disk need to be organized. The OS provides the file system — a logical view of files and folders that you see when you open File Explorer or Finder. Behind the scenes, the OS handles the actual reading and writing of bytes on the physical disk, manages free space, organizes directory structures, and enforces access permissions.

I/O Management

The OS hides the messy details of devices behind a clean interface. When you write text to a file, you do not care whether your storage is an SSD, an HDD, or a USB stick — the OS handles the differences. This is called device independence and it is one of the most valuable services the OS provides to programmers.

5. Protection and Security

These two terms sound similar but they mean different things in operating systems.

- **Protection** is about controlling access by processes and users inside the system. For example, Process A should not be allowed to read Process B's memory; an ordinary user should not be able to delete system files.
- **Security** is about defending the system from external threats — viruses, worms, hackers, denial-of-service attacks, identity theft, and so on.

The OS uses User IDs (UIDs) and Group IDs (GIDs) to identify who is doing what. Each file has permissions (read, write, execute) attached to it. In Linux, the chmod 755 file.sh command is used to set these permissions. The OS checks these permissions on every access and blocks unauthorized actions.

Protection mechanisms would be useless without the dual-mode setup we discussed in Section 3 — that is what physically prevents user code from bypassing the OS's checks.

6. Operating System Services

The OS provides a long list of services. These services split into two groups — services that help the user, and services that help the system run efficiently.

Services for the User

- **User interface** — CLI, GUI, or touch screen so you can interact with the system.
- **Program execution** — the OS loads programs into memory and runs them.
- **I/O operations** — read from the keyboard, write to the screen, talk to the printer.
- **File-system manipulation** — create, delete, read, write, search files and directories.
- **Communication** — let processes exchange information, on the same machine or across a network.
- **Error detection** — spot hardware errors (a failing disk), software errors (divide by zero), and respond appropriately.

Services for the System Itself

- **Resource allocation** — give CPU, memory, and devices to processes that need them.
- **Accounting** — keep track of which user used how much resources (useful in shared systems and cloud billing).
- **Protection and security** — as discussed in Section 5.

7. User Interface

The OS provides one or more ways for the user to interact with it.

- **Command-Line Interface (CLI)** — you type commands. Examples include Bash on Linux, PowerShell and CMD on Windows. CLIs are fast and powerful for experts, but require you to remember commands.
- **Graphical User Interface (GUI)** — you click icons, drag windows, and use menus. Examples are the Windows Desktop, GNOME on Linux, and macOS Finder. GUIs are easy for beginners.
- **Touch-screen interface** — you tap, swipe, and pinch. Examples are Android and iOS.
- **Batch interface** — old systems where you submitted a file of commands and the system processed it later. This is rare today, but is still seen in mainframe environments and in some scientific computing setups.

Most modern systems offer both CLI and GUI, and you can pick whichever one suits the task. For example, a developer might use the GUI to read email and the CLI to compile code.

8. System Calls

A system call is the programming interface to the services provided by the OS. It is how a user program asks the kernel to do something on its behalf.

When you write printf("Hello") in a C program, it looks simple — but internally, the C library calls the OS write() system call, which actually places characters on your screen. All this is hidden behind the friendly C function.

The Six Categories of System Calls

Silberschatz groups system calls into six families. Memorize these — they are a favourite exam question.

Category	Purpose	Example Calls
Process control	Create, run, wait for, end processes	fork(), exec(), wait(), exit(), abort()
File management	Work with files	open(), read(), write(), close(), create(), delete()
Device management	Request, release, and use devices	ioctl(), read(), write()
Information maintenance	Get/set system date, time, and IDs	getpid(), time(), getuid()
Communication	Let processes exchange messages	pipe(), socket(), send(), recv()
Protection	Control file and resource access	chmod(), chown(), umask()

API vs System Call

Programmers usually do not call system calls directly. They call functions in an API (Application Programming Interface) — like the Win32 API on Windows or the POSIX API on Unix and Linux. The API function takes care of preparing the arguments and triggering the actual system call. This makes programs portable and easier to write.

How Parameters Are Passed to a System Call

There are three common methods.

1. Through registers — a small number of parameters are loaded into CPU registers. Fast but limited.
2. Through a memory block — parameters are stored in a memory block, and the address of the block is passed in a register. Used when there are too many parameters for registers.
3. Through the stack — parameters are pushed onto the program's stack and popped by the OS.

9. System Programs

System programs (also called system utilities) are programs that come bundled with the OS to make life easier. They are not part of the kernel itself — they sit on top of it. Most of what you see when you log into a Linux system (the shell, editors, file utilities) are system programs, not the OS itself.

Categories of system programs:

- **File management —** cp, mv, rm, ls, mkdir. Copy, move, delete, list files.
- **Status information —** date, df, top, ps. Show date, disk free space, running processes.
- **File modification —** text editors like vi, nano, Notepad.
- **Programming language support —** compilers (gcc), interpreters (python), debuggers (gdb).
- **Program loading and execution —** loaders, linkers.
- **Communications —** ssh, web browsers, email clients.
- **Background services / daemons —** the print spooler, the scheduler, the login service, web servers.

10. Operating System Structure (Kernel Designs)

How is the OS itself organized internally? Different operating systems use different designs, each with its own trade-offs.

Simple / Monolithic Structure

All OS functionality is packed into a single large kernel. This is fast — there are no boundaries to cross internally — but it is hard to maintain. A bug anywhere can crash the entire system. Examples include the original UNIX, MS-DOS, and traditional Linux.

Layered Approach

The OS is divided into layers, each built on top of the layer below it. Layer 0 is the hardware; the topmost layer is the user interface. This is easier to debug because each layer only uses the layers below it. The downside is performance overhead from passing through multiple layers, and the difficulty of deciding what goes in which layer.

Microkernel

Move as much as possible out of the kernel and into user-space services. The file system, networking, and drivers all run as user processes. The kernel itself is tiny and only handles bare essentials like inter-process communication and basic memory management. Microkernels are more secure and reliable — a crashed driver does not bring down the OS — but they are slower because of constant message-passing. Examples include Mach, Minix, and QNX.

Modules (Loadable Kernel Modules)

A modern hybrid approach. The core kernel is small, and additional features (file systems, drivers) are loaded dynamically as modules when needed. Modern Linux works this way — the insmod, rmmod, and lsmod commands manage these modules.

Hybrid Systems

Most modern OSes mix approaches. Windows, macOS, and Linux all combine monolithic, layered, and modular ideas to balance performance and modularity.

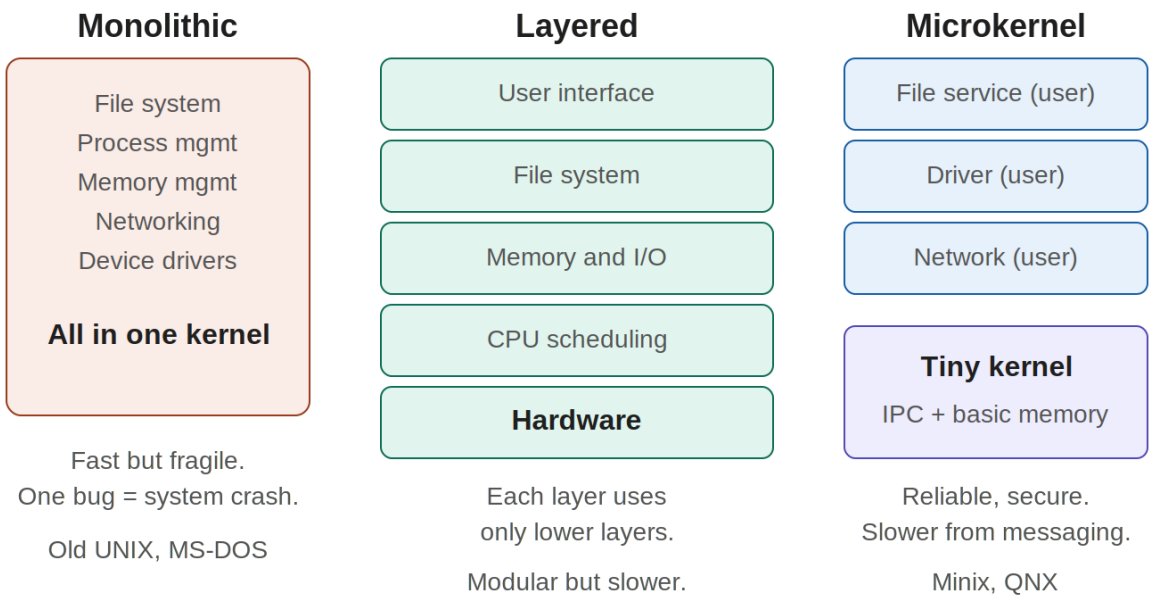


Figure 1.5 — The three classical OS structures compared side-by-side.

11. System Boot

When you press the power button, a sequence of carefully orchestrated steps brings the OS to life. This sequence is called booting.

Step-by-Step Boot Process

4. Power on. Electricity flows; the CPU starts in a known state and jumps to a fixed memory address in firmware.
5. BIOS or UEFI runs. The firmware stored in ROM/flash takes over. It performs POST (Power-On Self-Test) to check that essential hardware (RAM, CPU, keyboard, display) is working.
6. Bootloader is located and loaded. The BIOS/UEFI looks at boot devices in a configured order (SSD, USB, network) and loads the first stage of the bootloader — for example, GRUB on Linux, or Windows Boot Manager on Windows.
7. Bootloader loads the kernel. The bootloader finds the OS kernel image on disk, copies it into RAM, and jumps to it.
8. Kernel initializes. The kernel detects hardware, sets up memory management, starts essential services, and mounts the root file system.
9. First user process starts. On Linux this is init or systemd; on Windows it is smss.exe. This first process spawns all other user processes — login screens, system daemons, and so on.
10. Login screen appears. The OS is now ready for use.

Why is it called "booting"? — The very first program that runs is called the bootstrap program. It is small, hard-coded in firmware, and its only job is to load enough code to load the rest of the OS. The name comes from the phrase "pulling yourself up by your own bootstraps" — the OS literally lifts itself into memory.

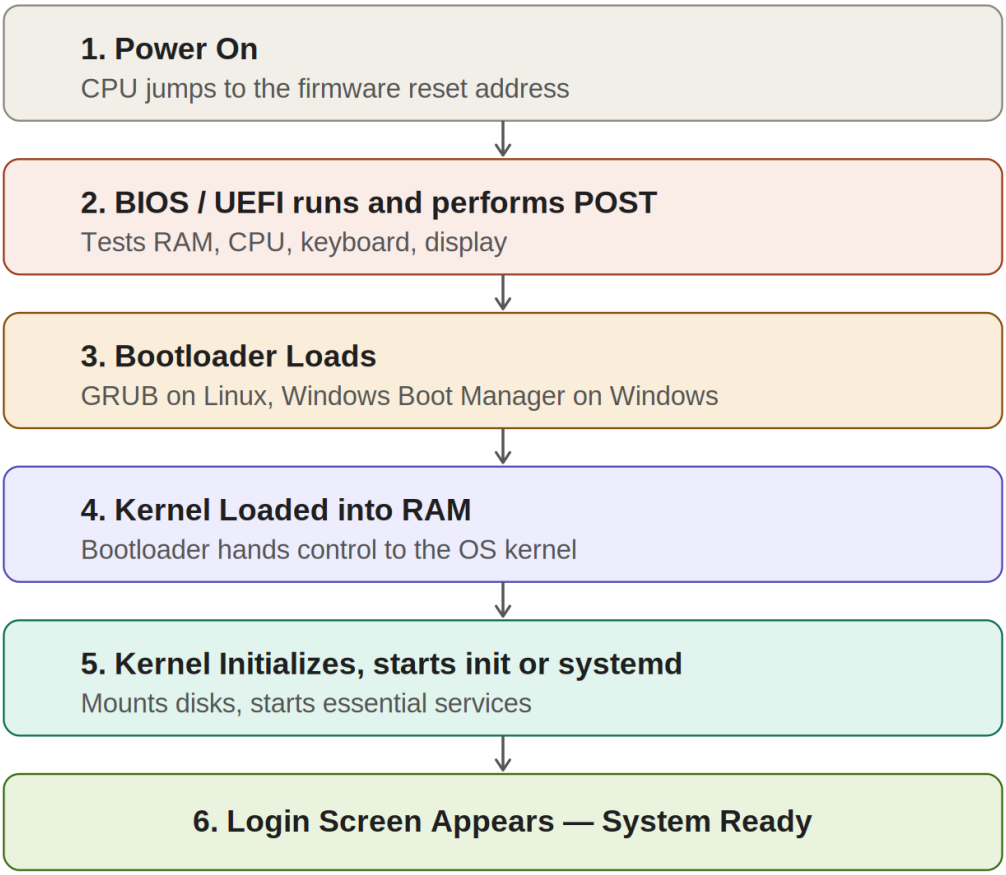


Figure 1.6 — The system boot sequence, from power-on to login.

12. Quick Revision — One-Page Summary

Here is everything in Unit 1 compressed for last-minute revision.

- **Definition:** OS = software that acts as an intermediary between user and hardware. Goals: convenience and efficiency.
- **Computer organization:** CPU + memory + I/O controllers connected by a system bus. Devices use interrupts (not polling) to get CPU attention.
- **Storage hierarchy:** Registers → Cache → RAM → SSD → HDD → Tape. Top is fast/small/expensive; bottom is slow/large/cheap.
- **Multiprogramming** keeps several programs in RAM to maximize CPU use. **Multitasking** switches between them rapidly to give each user the illusion of dedicated service.
- **Dual mode:** User mode (apps, restricted) vs Kernel mode (OS, full power). Mode bit in CPU. Privileged instructions only run in kernel mode. Switching happens via system calls or interrupts.
- **OS resource management:** Process management + Memory management + Storage/file management + I/O management.
- **Protection** = internal access control. **Security** = defence against external threats.

- **OS services:** for users (UI, program execution, I/O, files, communication, error detection) and for the system (resource allocation, accounting, protection).
- **System calls — six types:** Process control, File management, Device management, Information maintenance, Communication, Protection.
- **Parameter passing methods:** Registers, Block in memory, Stack.
- **Kernel structures:** Monolithic (one big lump) → Layered (stacked) → Microkernel (tiny core, rest in user space) → Modular/Hybrid (modern Linux, Windows, macOS).
- **Boot sequence:** Power on → BIOS/UEFI + POST → Bootloader → Kernel loaded → init/systemd → Login.

13. Practice Questions

Section A: Multiple Choice Questions (1 mark each)

Q1. The main role of an operating system is to:

a) Compile programs into machine code
b) Act as an intermediary between users and hardware
c) Provide internet access
d) Encrypt user data
Answer: (b)

Q2. Which of the following is NOT an operating system?

a) Ubuntu
b) macOS
c) Microsoft Excel
d) Android
Answer: (c) — Excel is an application program.

Q3. The two primary goals of an operating system are:

a) Speed and color
b) Convenience and efficiency
c) Cost and design
d) Compilation and linking
Answer: (b)

Q4. Which storage is the fastest in the memory hierarchy?

a) Hard disk
b) Cache
c) Registers
d) Main memory
Answer: (c)

Q5. A signal sent by a device to get the CPU's attention is called:

a) Polling
b) Interrupt
c) Cache miss
d) System bus
Answer: (b)

Q6. The bootstrap program is stored in:

a) RAM
b) Hard disk
c) ROM or firmware
d) Cache
Answer: (c)

Q7. Multiprogramming was introduced primarily to:

a) Run programs in parallel on multiple CPUs
b) Increase CPU utilization
c) Provide graphical interfaces
d) Reduce program size
Answer: (b)

Q8. In dual-mode operation, the mode bit value when a user program is running is:

a) 0
b) 1
c) -1
d) Undefined
Answer: (b)

Q9. Which of the following can be executed only in kernel mode?

a) Add two numbers
b) Read a value from RAM
c) Disable interrupts

d) Print a character
Answer: (c)

Q10. The hardware feature that prevents a user program from monopolizing the CPU is:

a) DMA
b) Cache
c) Timer interrupt
d) Bus arbiter
Answer: (c)

Q11. fork(), exec(), and exit() belong to which category of system calls?

a) File management
b) Process control
c) Communication
d) Information maintenance
Answer: (b)

Q12. Which is NOT a method of passing parameters to a system call?

a) Through registers
b) Through a memory block whose address is in a register
c) Through the stack
d) Through email
Answer: (d)

Q13. Which OS structure has all services running inside one large kernel?

a) Microkernel
b) Layered
c) Monolithic
d) Virtualized
Answer: (c)

Q14. In a microkernel design, services like file system and device drivers run in:

a) Kernel space
b) User space
c) Hardware
d) BIOS
Answer: (b)

Q15. POST stands for:

a) Power-On Self Test
b) Process On System Time
c) Pre-Operation Service Tool
d) Primary Operating System Test
Answer: (a)

Section B: Short Answer Questions (2 marks each)

Q1. Define operating system. State its two main goals.

Ans: An operating system is a software program that acts as an intermediary between the user and the computer hardware. It manages all hardware and software resources of the computer system. Its two main goals are convenience (making the system easy to use) and efficiency (using hardware resources optimally).

Q2. What is an interrupt? Why is it preferred over polling?

Ans: An interrupt is a signal sent by a hardware device to the CPU when it needs attention. It is preferred over polling because, in polling, the CPU continuously checks devices and wastes cycles, whereas with interrupts the CPU does useful work and only responds when actually needed.

Q3. Differentiate between user mode and kernel mode.

Ans: User mode (mode bit = 1) is the restricted mode in which all application programs run; they cannot directly access hardware or execute privileged instructions. Kernel mode (mode bit = 0) is the unrestricted mode in which the OS runs; it can execute any instruction and access any memory.

Q4. What is a system call? Give two examples.

Ans: A system call is the programming interface through which a user-mode program requests a service from the OS kernel. Examples: fork() (create a new process) and open() (open a file).

Q5. Define multiprogramming and multitasking.

Ans: Multiprogramming keeps several programs in memory at the same time so the CPU can switch to another whenever the current one waits for I/O — its goal is high CPU utilization. Multitasking (time-sharing) extends this by switching between programs so quickly that each user feels their program is running continuously.

Q6. What are privileged instructions? Give two examples.

Ans: Privileged instructions are special CPU instructions that can be executed only in kernel mode because they could harm the system if misused. Examples: disabling all interrupts, halting the CPU, switching from user to kernel mode.

Q7. What is the function of the bootstrap program?

Ans: The bootstrap program is a small program stored in firmware (ROM/UEFI) that runs first when the computer is powered on. Its job is to initialize the hardware and load the OS kernel from disk into main memory and start its execution.

Q8. List any four services provided by an OS.

Ans: Program execution, I/O operations, file-system manipulation, communication between processes, error detection, resource allocation, accounting, and protection and security. (Any four are fine.)

Q9. Differentiate between protection and security.

Ans: Protection deals with controlling access inside the system — making sure one process or user cannot interfere with another. Security deals with defending the system from external threats such as viruses, hackers, and unauthorized access.

Q10. What is dual-mode operation? Why is it needed?

Ans: Dual-mode operation is a hardware feature in which the CPU operates in either user mode or kernel mode, tracked by a mode bit. It is needed to protect the OS and other programs from buggy or malicious user code by ensuring privileged instructions can only run in kernel mode.

Q11. List the six categories of system calls.

Ans: Process control, File management, Device management, Information maintenance, Communication, and Protection.

Q12. What is meant by booting?

Ans: Booting is the start-up sequence that brings the computer from power-off to a usable state — running the bootstrap program, doing POST, loading the bootloader, loading the OS kernel, and starting the first user process.

Section C: Long Answer Questions (5 marks each)

Q1. Explain the layered view of a computer system, showing where the OS fits.

Approach: *Cover the four-layer model — Hardware → Operating System → Application Programs → User. Explain how each layer is built on top of the one below. Mention that the OS hides hardware complexity from applications and provides a clean, consistent interface. Include Figure 1.1 in your answer.*

Q2. Describe the storage hierarchy in a computer system.

Approach: *Cover the pyramid: Registers → Cache (L1, L2, L3) → Main memory (RAM) → SSD → HDD → Magnetic tape / cloud. Explain the trade-off — top is faster, smaller, and costlier per byte; bottom is slower, larger, and cheaper. Mention that the CPU first looks for data in registers, then cache, then RAM, then disk (this is why caching matters).*

Q3. Discuss the six categories of system calls with examples.

Approach: *Give a brief description of each category (Process control, File management, Device management, Information maintenance, Communication, Protection) and provide one or two example system calls for each — for instance, fork(), open(), ioctl(), getpid(), pipe(), chmod().*

Q4. Compare monolithic, layered, and microkernel OS structures.

Approach: *Use a small comparison table covering: (a) where services live, (b) performance, (c) reliability/security, (d) ease of extension, (e) examples. Conclude that modern OSes (Linux, Windows, macOS) use a hybrid/modular approach combining the best of each. Include Figure 1.5 in your answer.*

Q5. Explain the system boot process step by step.

Approach: *Describe the six steps: (1) Power on, (2) BIOS/UEFI runs and performs POST, (3) Bootloader is loaded from boot device (GRUB / Windows Boot Manager), (4) Bootloader loads the OS kernel into RAM, (5) Kernel initializes hardware and starts init or systemd, (6) Login screen appears. Mention what bootstrap means. Include Figure 1.6 in your answer.*

Q6. Explain how parameters are passed from a program to the OS during a system call.

Approach: *There are three methods: (a) Through registers — parameters loaded into CPU registers (used when there are few parameters), (b) Through a memory block — parameters stored in a memory block, with the block's address passed in a register (used when there are many parameters), (c) Through the stack — parameters pushed onto the program's stack and popped by the OS. Briefly explain each with a small diagram.*

Q7. Explain the role of dual-mode operation and the timer interrupt in protecting the OS.

Approach: *Dual-mode operation isolates the OS from user programs by classifying instructions as privileged or non-privileged; only the OS in kernel mode can execute privileged ones. The timer interrupt prevents a user program from looping forever and never giving control back — when the timer fires, control passes to the OS, which can preempt the program. Together, they ensure the OS always remains in control.*

Q8. List and explain any five services provided by the OS.

Approach: *Pick five from: program execution, I/O operations, file-system manipulation, communication, error detection, resource allocation, accounting, protection and security. Give a one-line explanation of each with a real-world example. For instance, "file-system manipulation lets you create, delete, read, and write files — when you save a Word document, the OS writes the file to disk on your behalf".*

UNIT 2

Process Management

Contents

1. Process Concept

What is Process?

A process is simply a program in execution. But there is an important distinction here — a program is a passive entity (like a cooking recipe sitting in a book), while a process is an active entity (the actual cooking happening in your kitchen). The program is just a file stored on disk; the process is what happens when that file is loaded into memory and the CPU starts running its instructions.

A single program can create multiple processes. For example, opening three Chrome windows means there are three processes, all running the same Chrome program.

Program vs Process — Quick Comparison

Aspect	Program	Process
Nature	Passive entity	Active entity
Storage	On disk	In main memory (RAM)
Lifetime	Exists until deleted	Exists until it terminates
Resources	None	Has CPU time, memory, files, etc.
Example	notepad.exe file	A running Notepad window

Process in Memory

When a process is loaded into memory, it is divided into four sections. Each section stores a different kind of information.

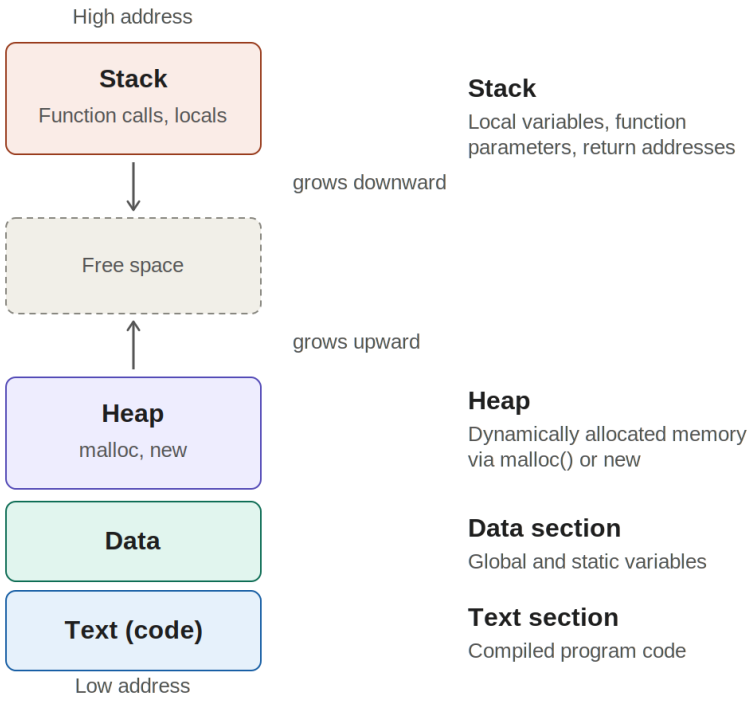


Figure 2.1 — Process memory layout: text, data, heap (grows up), stack (grows down).

> **Why this matters —** The heap and stack grow toward each other from opposite ends. If they collide, you get a stack overflow or out-of-memory error. This is why infinite recursion crashes a program.

2. Process States

A process does not just run from start to finish without interruption — as it goes through its life, it moves between different states depending on what it is doing. The classical five-state model is:

New — the process is being created.

Ready — the process is in memory and waiting to be assigned to the CPU.

Running — instructions are currently being executed on the CPU.

Waiting (Blocked) — the process is waiting for some event (I/O completion, a signal, etc.).

Terminated — the process has finished execution.

At any given moment on a single-CPU system, only one process can be in the Running state. Many processes can be in the Ready state (waiting their turn) and in the Waiting state (waiting for I/O).

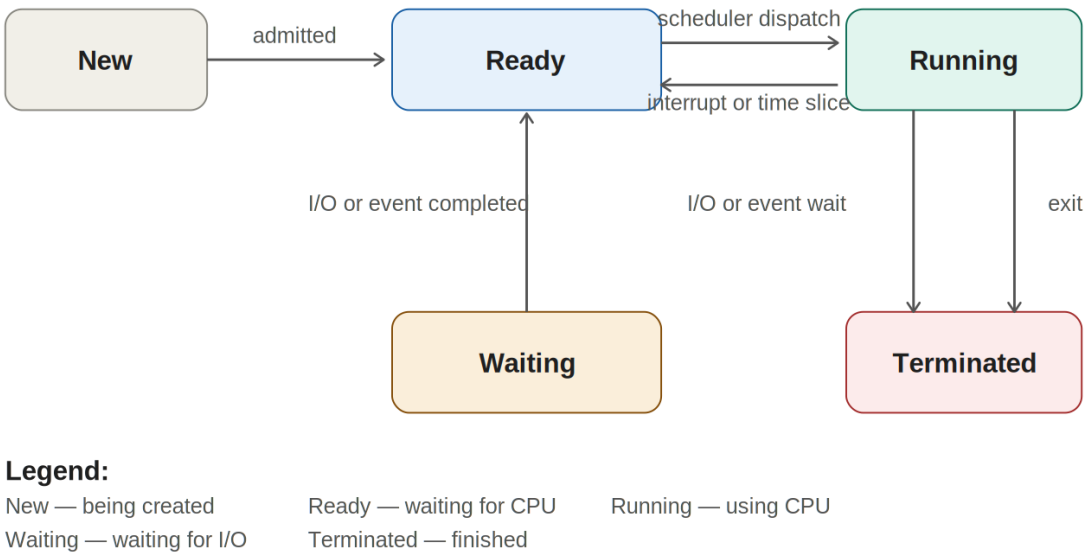


Figure 2.2 — Process state transition diagram.

> **Real-life analogy —** Think of a patient at a doctor's clinic. New = just walked in and registering. Ready = seated in the waiting room. Running = being examined. Waiting = sent to the lab for a blood test. Terminated = treatment done, walks out.

3. Process Control Block

The OS keeps a data structure called the Process Control Block (PCB) for every process. The PCB is like a file folder that contains everything the OS needs to know about that process. When a process is suspended and later resumed, the OS reads the PCB to restore it exactly where it left off.

Contents of a PCB

Process ID (PID) — unique number identifying the process.

Process state — new, ready, running, waiting, or terminated.

Program counter (PC) — address of the next instruction to execute.

CPU registers — values of all CPU registers (accumulator, index registers, stack pointer, etc.).

CPU scheduling information — priority, pointers to scheduling queues.

Memory management information — base register, limit register, page tables.

Accounting information — CPU time used, time limits, user ID.

I/O status information — list of open files, I/O devices allocated.

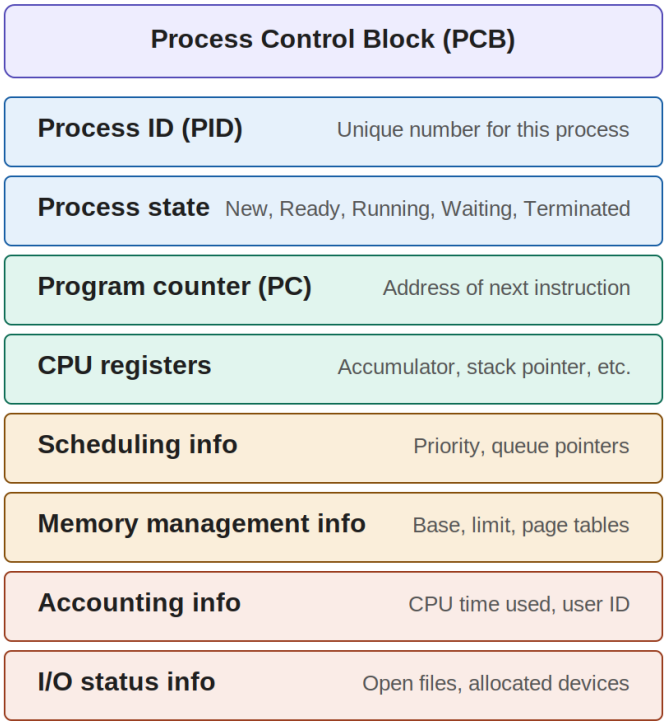


Figure 2.3 — Structure of a Process Control Block.

4. Process Scheduling

Since there are usually far more processes than CPUs, the OS must decide which process gets the CPU and when. This is called process scheduling. Its main goal is to maximize CPU utilization and give every process fair access.

Scheduling Queues

The OS maintains several queues to organize processes:

Job queue — contains all processes in the system (including new ones waiting to be admitted).

Ready queue — processes in main memory that are ready and waiting for the CPU.

Device queues — separate queue for each I/O device; processes waiting for that device line up here.

A process moves between these queues throughout its life. It enters the ready queue, gets picked for the CPU, maybe goes to a device queue for I/O, comes back to the ready queue, and eventually terminates.

Types of Schedulers

Long-term scheduler (Job scheduler) — decides which jobs from the job queue get admitted into memory. Runs infrequently (seconds or minutes). Controls the degree of multiprogramming (how many processes are in memory).

Short-term scheduler (CPU scheduler) — picks which ready process gets the CPU next. Runs very frequently (milliseconds). Must be fast.

Medium-term scheduler — temporarily removes (swaps out) processes from memory to disk when memory is tight, and brings them back (swaps in) later. Helps manage memory pressure.

Context Switch

When the CPU switches from one process to another, it must save the current process's state to its PCB and load the next process's state from its PCB. This is called a context switch.

Context switching is pure overhead — the CPU does no useful work for the user during a switch. The faster the hardware, the faster the switch, but it is never free. Typical context switch times range from a few microseconds to a few hundred microseconds.

5. Operations on Processes

Process Creation

In Unix and Linux, a new process is created by the fork() system call. The calling process is the parent, and the new process is the child. The child is an almost-identical copy of the parent — same program code, same open files, same variable values.

The key trick: fork() returns twice — once in the parent (returning the child's PID) and once in the child (returning 0). This lets a single piece of code behave differently in parent and child.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int pid = fork();
   if (pid < 0) {
      printf("Fork failed\n");
   } else if (pid == 0) {
      printf("I am the child. My PID is %d\n", getpid());
   } else {
      printf("I am the parent. My child's PID is %d\n", pid);
   }
   return 0;
}
```

After fork(), the child usually calls exec() to replace its memory image with a new program. The parent can call wait() to wait for the child to finish.

Process Termination

A process terminates when it:

Finishes executing and calls exit().

Is killed by another process using kill(pid, SIGKILL).

Encounters a fatal error (division by zero, invalid memory access).

Is terminated by the parent using abort().

Zombie and Orphan Processes

A zombie process is one that has finished executing but whose PCB is still in the system because its parent has not yet called wait() to read its exit status. Zombies consume a PID but no memory.

An orphan process is one whose parent has terminated before it. In Linux, orphans are automatically "adopted" by the init process (PID 1), which eventually calls wait() on them.

6. System Boot Inter-Process Communication(IPC)

Processes often need to share data or coordinate their work. Since processes have their own separate memory spaces, they cannot directly read each other's variables. The OS provides IPC mechanisms to make communication possible.

Two major models:

Shared memory — the OS sets up a region of memory that two or more processes can both access. Fast (once set up, processes just read and write memory), but you must be careful about synchronization (two processes writing the same location at once causes chaos).

Message passing — processes send and receive messages through the OS. Slower than shared memory (the OS has to copy the data), but no shared memory conflicts.

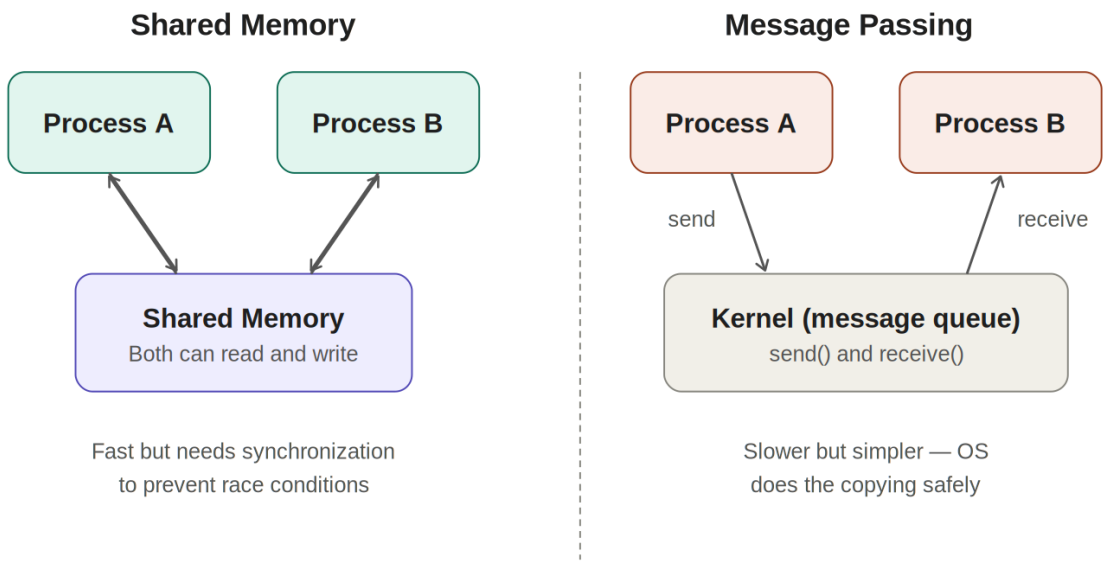


Figure 2.4 — IPC models: Shared memory (left) vs Message passing (right).

Common IPC Mechanisms

Pipes (|) — used between related processes (parent-child). One-way by default. Example: ls | grep .txt

Named pipes (FIFO) — like pipes, but have a name on the filesystem, so any two processes can use them.

Message queues — a queue of messages managed by the kernel.

Shared memory segments — explicitly created via shmget() in Unix.

Sockets — used for communication across networks, or between processes on the same machine.

Signals — small notifications sent between processes (like SIGKILL, SIGSTOP).

The Producer-Consumer Problem

A classic IPC example: a producer process generates data (say, the compiler reads source code) and a consumer process uses it (the assembler takes the compiler's output). They share a bounded buffer (a fixed-size queue). The producer fills the buffer; the consumer empties it. They must coordinate so the producer does not overwrite data the consumer has not read yet, and the consumer does not try to read from an empty buffer.

We will see this problem solved with semaphores in Unit 3.

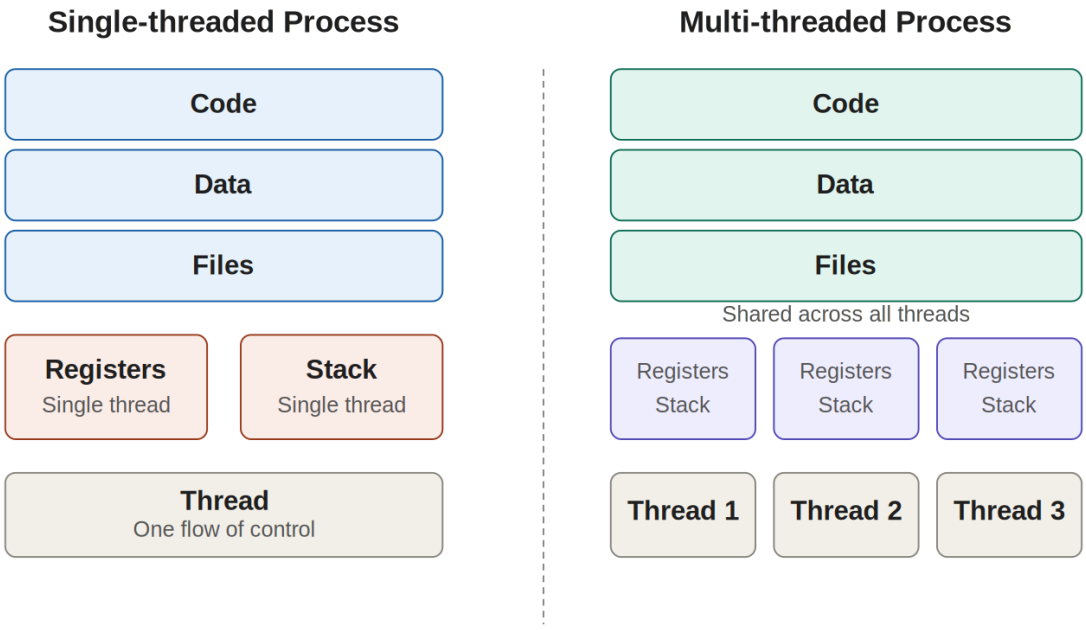
7. Threads

What is a Thread?

A thread is a lightweight process — it is the smallest unit of execution. Think of a process as a complete program, and threads as separate streams of execution within that program. All threads of a process share the same memory and resources, but each has its own program counter, registers, and stack.

Real-world analogy: Imagine a restaurant kitchen (the process). The kitchen has ovens, ingredients, and recipes (shared memory). Multiple chefs (threads) work in the same kitchen — they share the facilities but each chef prepares a different dish at the same time.

Process vs Thread — Memory Layout



Threads share code, data, and files — but each has its own registers and stack.

Figure 2.5 — Single-threaded process (left) vs Multi-threaded process (right).

Benefits of Multithreading

Responsiveness — in a GUI app, one thread can handle user clicks while another does background work (like saving a file), so the app never "freezes".

Resource sharing — threads automatically share the process's memory, so they can exchange data without IPC.

Economy — creating and context-switching threads is much cheaper than processes (no separate memory space).

Scalability — on a multi-core CPU, different threads can truly run on different cores in parallel.

User Threads vs Kernel Threads

User-level threads — managed by a user-space library; the kernel does not know about them. Fast to create and switch, but if one thread makes a blocking system call, the entire process blocks.

Kernel-level threads — managed directly by the OS. Slower to create, but the OS can schedule them independently on different CPU cores.

8. Multi-threading Models

How do user threads map to kernel threads? Three common models:

Many-to-One Model

Many user threads map to a single kernel thread. Thread management is done entirely in user space, so it is fast. But the whole process blocks if one thread blocks, and it cannot use multiple CPU cores. Example: Green threads in older Java.

One-to-One Model

Each user thread maps to its own kernel thread. True parallelism is possible, and a blocked thread does not block the process. But creating kernel threads has overhead, so the number of threads is limited. Example: Windows, Linux.

Many-to-Many Model

Many user threads multiplex onto a smaller or equal number of kernel threads. Combines the best of both — the user can create many threads, but the kernel only sees a manageable number. Example: Solaris, older Windows NT.

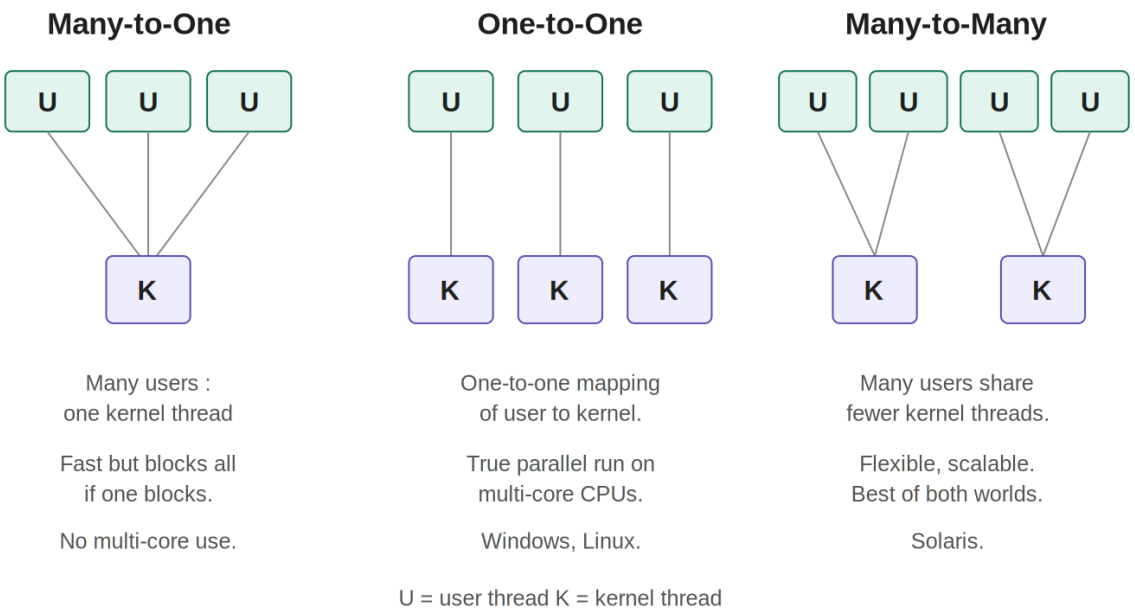


Figure 2.6 — The three multi-threading models compared.

9. Threading Issues

A few tricky problems arise when you use threads:

fork() and exec() Semantics

If a multithreaded process calls fork(), should the child have all the parent's threads, or just the one that called fork()? Some systems provide two versions. Usually, if exec() follows immediately, duplicating only the calling thread is sufficient.

Signal Handling

When a signal (like Ctrl+C) arrives at a multithreaded process, which thread should handle it? Options: deliver to the thread the signal applies to, deliver to every thread, or have a specific thread assigned to handle all signals.

Thread Cancellation

Terminating a thread before it completes its task. Two styles — asynchronous cancellation (terminate immediately) is fast but risky (shared data may be left inconsistent); deferred cancellation (the thread periodically checks if it should terminate) is safer.

Thread-Local Storage (TLS)

Sometimes each thread needs its own private copy of a variable, even though threads share memory. TLS lets you declare such variables.

Scheduler Activations

A communication scheme between user thread library and kernel (used in many-to-many model) so the kernel can tell the library "your thread is about to block" and let the library reschedule other user threads.

10. Numerical Questions with Solutions

Numerical 1 — fork() Descendants

Q: Consider the following C code. How many child processes are created?

```
fork();
fork();
fork();
```

Solution: Each fork() call doubles the number of processes.
Initially: 1 process (the parent).
After 1st fork(): 2 processes (parent + 1 child).
After 2nd fork(): 4 processes (each of the 2 calls fork again).
After 3rd fork(): 8 processes.
So total number of processes = $\mathbf{2^3 = 8}$. Out of these 8, one is the original parent, so the number of child processes created = $\mathbf{2^3 - 1 = 7}$.

General rule: *For n consecutive fork() calls, total processes = 2n, child processes = 2n − 1.*

Numerical 2 — Context Switch Overhead

Q: A CPU scheduler uses a time quantum of 20 ms. Each context switch takes 2 ms. Calculate the CPU overhead due to context switching.

Solution:

Time spent doing useful work (user process) = 20 ms

Time spent on context switch = 2 ms

Total time per cycle = 20 + 2 = 22 ms

Overhead percentage = (context switch time / total time) × 100 = (2 / 22) × 100 = 9.09%

Answer: About 9.09% of CPU time is wasted in context switching. Notice — if we reduce the quantum to 10 ms, overhead rises to 2/12 = 16.67%. Smaller quanta give smoother multitasking but waste more time switching.

Numerical 3 — CPU Utilization with Multiprogramming

Q: A process spends 80% of its time waiting for I/O. What is the CPU utilization when there are (a) 1 process, (b) 4 processes, (c) 10 processes in memory?

Solution: Using the standard formula:

CPU utilization = 1 − p^n

where p = fraction of time a process spends waiting for I/O (0.8 here), and n = number of processes in memory.

(a) n = 1: Utilization = $1 - 0.8^1 = 1 - 0.8 = 0.20$ (20%)
(b) n = 4: Utilization = $1 - 0.8^4 = 1 - 0.4096 = 0.59$ (≈ 59%)
(c) n = 10: Utilization = $1 - 0.8^{10} = 1 - 0.1074 = 0.89$ (≈ 89%)
This shows why multiprogramming is so powerful — with just 10 processes in memory, the CPU stays busy nearly 90% of the time even though each individual process is I/O-bound.

Numerical 4 — PCB Memory Usage

Q: A system has 200 processes active at any time, and each PCB takes 1 KB of memory. How much total memory is consumed by the PCBs?

Solution:

Memory per PCB = 1 KB = 1024 bytes
Number of processes = 200
Total memory = 200 × 1024 bytes = 204,800 bytes = 200 KB
This is the minimum memory overhead the OS incurs just to track all running processes — it does not include the processes' own code, data, or stack.

Numerical 5 — fork() Output Prediction

Q: What does the following code print? How many lines of output?

```
#include <stdio.h>
#include <unistd.h>
int main() {
   fork();
   fork();
   printf("Hello\n");
   return 0;
}
```

Solution: Each fork() doubles the processes. Two fork() calls produce $2^2 = 4$ processes. All four reach printf, so the output has 4 lines of "Hello".

Numerical 6 — Thread vs Process Creation Time

Q: Process creation takes 10 ms, thread creation takes 1 ms. A program needs 100 concurrent tasks. Compare the time to create all tasks using processes vs threads.

Solution:

Using processes: 100 × 10 ms = 1000 ms = 1 second

Using threads: 100 × 1 ms = 100 ms = 0.1 second

Threads are 10 times faster to create. This is why web servers (like Apache in threaded mode) and databases prefer threads for handling concurrent client requests.

Numerical 7 — Identifying fork() Results

Q: In the following code, which line does the child process execute?

```
int pid = fork();
if (pid == 0)
   printf("A\n");       // Line 1
else if (pid > 0)
   printf("B\n");       // Line 2
else
   printf("C\n");       // Line 3
```

Solution:

In the child process, fork() returns 0 → condition pid == 0 is true → child executes Line 1 ("A").

In the parent process, fork() returns child's PID (a positive number) → condition pid > 0 is true → parent executes Line 2 ("B").

Line 3 executes only if fork() fails (returns -1).

11. Quick Revision — One-Page Summary

Here is everything in Unit 2 compressed for last-minute revision.

Process = program in execution (active entity); **Program** = code on disk (passive entity).

Process memory sections: Text (code) | Data (globals) | Heap (malloc) | Stack (local vars).

Five process states: New → Ready → Running → (Waiting ↔ Ready) → Terminated.

PCB stores: PID, state, PC, registers, scheduling info, memory info, accounting info, I/O status.

Schedulers: Long-term (admits jobs) | Short-term (picks next CPU process) | Medium-term (swaps).

Context switch = save current PCB + load next PCB. Pure overhead.

Process operations: fork() creates child, exec() replaces program, wait() waits, exit() terminates.

fork() rule: n forks = 2n total processes, 2n − 1 children.

Zombie = dead but parent has not reaped it. **Orphan** = parent died, adopted by init.

IPC models: Shared memory (fast, needs sync) | Message passing (slow, safe).

IPC mechanisms: Pipes, FIFOs, message queues, shared memory, sockets, signals.

Thread = lightweight process. All threads of a process share code, data, files — but have their own registers and stack.

Benefits of threading: Responsiveness, resource sharing, economy, scalability on multi-core.

Multi-threading models: Many-to-One (old Java), One-to-One (Windows/Linux), Many-to-Many (Solaris).

Threading issues: fork/exec semantics, signal handling, cancellation (async vs deferred), thread-local storage.

Key formulas:

CPU utilization with multiprogramming: U = 1 − p^n

Number of processes after n forks: 2^n

Context switch overhead: CS time / (CS time + useful time) × 100

12. Practice Questions

Section A: Multiple Choice Questions (1 mark each)

Q1. A program in execution is called:

a) Code

b) Process

c) File

d) Compilation

Answer: (b) — A process is a program that is currently executing in memory.

Q2. Which of the following is a passive entity?

a) Process

b) Program

c) Thread

d) PCB

Answer: (b) — A program is just a file on disk; it has no execution context.

Q3. In the process memory layout, dynamically allocated memory (malloc) goes into:

a) Stack

b) Text

c) Heap

d) Data

Answer: (c)

Q4. In the process memory layout, the section that stores function calls and local variables is:

a) Heap

b) Stack

c) Text

d) Data

Answer: (b)

Q5. On a single-CPU system, how many processes can be in the Running state at one time?

a) 0

b) 1

c) 2

d) Many

Answer: (b)

Q6. Which data structure does the OS use to store information about each process?

a) Page Table

b) File Descriptor

c) Process Control Block

d) Inode

Answer: (c) — PCB.

Q7. Which scheduler decides which process should next get the CPU?

a) Long-term scheduler

b) Medium-term scheduler

c) Short-term scheduler

d) I/O scheduler

Answer: (c)

Q8. Saving the state of one process and loading the state of another is called:

a) Dispatching

b) Context switch

c) Preemption

d) Multiprogramming

Answer: (b)

Q9. In Linux, the system call used to create a new process is:

a) new()

b) create()

c) fork()

d) spawn()

Answer: (c)

Q10. After a fork() call, the value returned to the child process is:

a) The child's own PID

b) The parent's PID

c) 0

d) -1

Answer: (c)

Q11. A process whose parent has terminated before it is called:

a) Zombie

b) Orphan

c) Daemon

d) Idle

Answer: (b)

Q12. A process that has finished but whose entry remains in the process table is:

a) Zombie

b) Orphan

c) Daemon

d) Suspended

Answer: (a)

Q13. In Linux, an orphan process is adopted by:

a) The shell

b) The init process (PID 1)

c) The parent's parent

d) The OS scheduler

Answer: (b)

Q14. Which of the following is NOT an IPC mechanism?

a) Pipe

b) Shared memory

c) Page table

d) Message queue

Answer: (c)

Q15. Compared to processes, threads are:

a) Heavier

b) Lighter

c) Equally heavy

d) Slower to create

Answer: (b)

Q16. Threads of the same process share which of the following?

a) Stack

b) Registers

c) Code and data

d) Program counter

Answer: (c)

Q17. The Many-to-One model maps:

a) One user thread to one kernel thread

b) Many user threads to one kernel thread

c) Many user threads to many kernel threads

d) Each kernel thread to a separate process

Answer: (b)

Q18. Which OS uses the One-to-One thread model?

a) Older Java green threads

b) Solaris

c) Linux and Windows

d) MS-DOS

Answer: (c)

Q19. If a multithreaded process calls fork(), one common semantic is:

a) All threads are duplicated

b) Only the calling thread is duplicated

c) No threads exist in the child

d) The fork fails

Answer: (b) — Common when exec() will immediately follow.

Q20. After two consecutive fork() calls in a program, the total number of processes is:

a) 2

b) 3

c) 4

d) 5

Answer: (c) — $2^2 = 4$.

Section B: Short Answer Questions (2 marks each)

Q1. Differentiate between a program and a process.

Ans: A program is a passive entity — a file stored on disk containing instructions. A process is an active entity — a program in execution, loaded into memory, with its own PCB, registers, and resources. The same program can give rise to multiple processes.

Q2. Name the four sections of process memory.

Ans: Text section (compiled code), Data section (global and static variables), Heap (dynamically allocated memory via malloc/new), and Stack (function calls, local variables, return addresses).

Q3. List the five process states.

Ans: New, Ready, Running, Waiting (also called Blocked), and Terminated. A process moves between these states during its lifetime.

Q4. What is a Process Control Block (PCB)? List any four fields it contains.

Ans: A PCB is a data structure maintained by the OS for every process. It stores all information needed to manage the process. Fields include: PID, process state, program counter, CPU registers, scheduling info, memory info, accounting info, I/O status.

Q5. What is a context switch? Why is it considered overhead?

Ans: A context switch is the procedure of saving the state (PCB) of the currently running process and loading the state of the next process to be run. It is overhead because during the switch the CPU is not doing any useful work for the user — it is busy managing process state.

Q6. Differentiate between long-term, short-term, and medium-term schedulers.

Ans: Long-term scheduler decides which jobs are admitted into memory (controls degree of multiprogramming). Short-term scheduler picks which ready process gets the CPU next. Medium-term scheduler swaps processes between memory and disk to manage memory pressure.

Q7. What is the fork() system call? What does it return in the parent and child?

Ans: fork() creates a new process by duplicating the calling process. It returns twice — in the parent it returns the PID of the new child (a positive number); in the child it returns 0. On failure it returns -1 in the parent and no child is created.

Q8. Define zombie process and orphan process.

Ans: A zombie process is one that has terminated but whose entry remains in the process table because the parent has not yet called wait(). An orphan process is one whose parent has terminated before it; in Linux, orphans are adopted by the init process (PID 1).

Q9. What is Inter-Process Communication (IPC)? Name the two main models.

Ans: IPC is the mechanism that allows separate processes to exchange data and coordinate their actions. The two main models are shared memory (a region of memory that multiple processes can access) and message passing (processes communicate through send/receive operations managed by the kernel).

Q10. Differentiate between a process and a thread.

Ans: A process is an independent execution unit with its own memory space. A thread is a lightweight unit of execution inside a process; multiple threads of the same process share the code, data, and files but each has its own registers and stack. Thread creation and switching are far cheaper than process creation.

Q11. List the four benefits of multithreading.

Ans: Responsiveness (UI does not freeze during background work), Resource sharing (threads share memory naturally), Economy (cheaper than separate processes), and Scalability (true parallelism on multi-core CPUs).

Q12. Name the three multi-threading models with one example each.

Ans: Many-to-One (old Java green threads), One-to-One (Windows, Linux), and Many-to-Many (Solaris).

Q13. What are the two styles of thread cancellation?

Ans: Asynchronous cancellation — the thread is terminated immediately; fast but may leave shared data inconsistent. Deferred cancellation — the thread periodically checks if it should terminate and exits at a safe point; safer.

Section C: Long Answer Questions (5 marks each)

Q1. Explain the process concept and the layout of a process in memory.

Approach: *Define process as a program in execution and contrast with program. Describe the four memory sections (text, data, heap, stack) — what each holds and how heap and stack grow toward each other. Include Figure 2.1 in the answer.*

Q2. Draw and explain the process state transition diagram.

Approach: *Describe the five states (New, Ready, Running, Waiting, Terminated) and explain every transition: admitted, scheduler dispatch, interrupt/time slice, I/O wait, I/O completion, and exit. Mention that only one process can be in Running state on a single-CPU system. Include Figure 2.2.*

Q3. Explain the Process Control Block (PCB) with all the fields it stores.

Approach: *Define PCB and explain each field: PID, process state, program counter, CPU registers, scheduling info, memory management info, accounting info, I/O status info. Mention that the PCB is essential for resuming a process after a context switch. Include Figure 2.3.*

Q4. Compare the three types of schedulers (long-term, short-term, medium-term).

Approach: *Explain each scheduler's role and frequency of execution. Make a comparison table covering: which queue they operate on, how often they run, and what they decide. Mention that in modern UNIX, the long-term scheduler may be minimal/absent because new processes are immediately admitted.*

Q5. Discuss the operations on processes — creation and termination — with examples.

Approach: *Cover process creation via fork(): explain that the child is a copy of the parent, fork() returns twice, and exec() is often called next. Cover termination methods — exit(), kill, errors, abort. Discuss zombie and orphan processes. Provide a small C code example showing fork() with parent and child paths.*

Q6. Explain Inter-Process Communication (IPC) with the two main models and at least four mechanisms.

Approach: *Define IPC. Compare shared memory (fast, needs synchronization) and message passing (slow, safe). List mechanisms: pipes, named pipes (FIFO), message queues, shared memory segments, sockets, signals. Include Figure 2.4. Briefly mention the producer-consumer problem as a classic IPC example.*

Q7. Define a thread. Compare single-threaded and multi-threaded processes, and list the benefits of multithreading.

Approach: *Define thread as the smallest unit of execution within a process. Show the memory layout difference: single-threaded has one stack and one set of registers; multi-threaded has separate per-thread stacks but shares code/data/files. Include Figure 2.5. List the four benefits: responsiveness, resource sharing, economy, scalability.*

Q8. Compare the three multi-threading models with diagrams.

Approach: *Cover Many-to-One (single kernel thread, fast but no parallelism, blocks all on one block), One-to-One (one kernel thread per user thread, true parallelism, used in Windows and Linux), and Many-to-Many (many user threads multiplex over fewer kernel threads, flexible, used in Solaris). Include Figure 2.6 in the answer.*

Q9. Explain any five threading issues that arise in multithreaded programming.

Approach: *Cover: (1) fork() and exec() semantics — should fork duplicate all threads or just the calling thread; (2) signal handling — to which thread should a signal be delivered; (3) thread cancellation — asynchronous vs deferred; (4) thread-local storage — when each thread needs its own private variables; (5) scheduler activations — kernel-library communication in many-to-many model.*

UNIT 3

Process Synchronization

Contents

1. Why Do We Need Synchronization?

When multiple processes (or threads) share data and run concurrently, weird bugs appear that never occur in single-threaded programs. The reason: the OS can pause one process at any instruction and switch to another. If two processes are modifying the same variable without coordination, the final result depends on the exact order of operations — which is unpredictable.

The Classic Example — A Race Condition

Suppose two processes share a variable counter that starts at 5. Process A wants to increment it; Process B wants to decrement it.

```
// Process A                    // Process B
counter = counter + 1;          counter = counter - 1;
```

This looks innocent, but at the machine level, each statement actually translates to three CPU instructions:

```
// counter = counter + 1        // counter = counter - 1
register1 = counter             register2 = counter
register1 = register1 + 1       register2 = register2 - 1
counter = register1             counter = register2
```

If these instructions interleave unluckily, the final value of counter could be 4, 5, or 6 — even though logically (+1) and (-1) should cancel and leave it at 5. This unpredictable outcome is called a race condition because the result depends on who "races" to the finish line first.

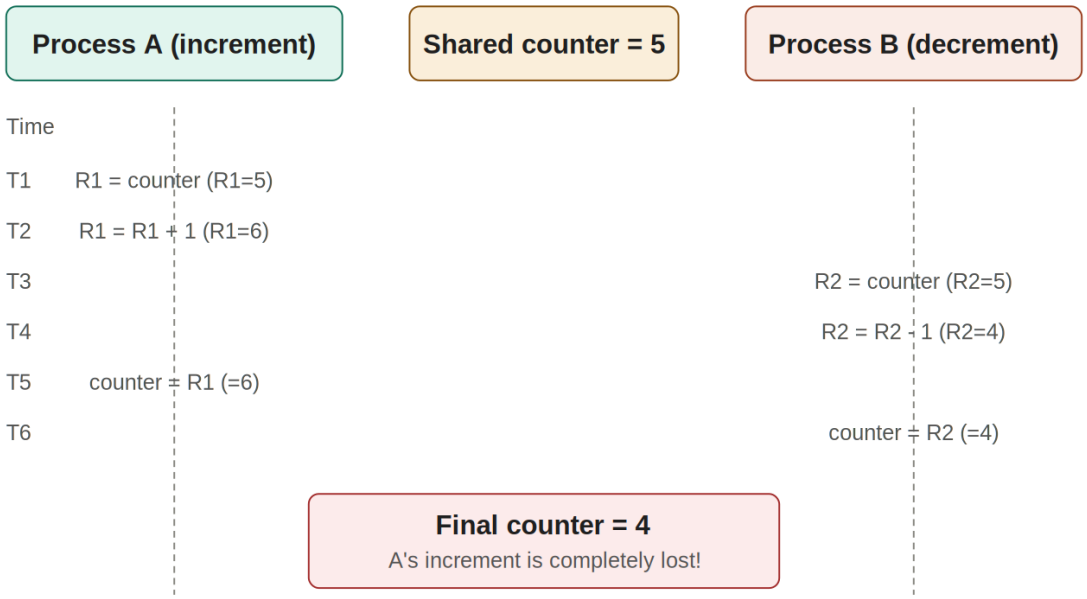


Figure 3.1 — A race condition trace. Process A's increment is completely lost.

Why Synchronization Matters

Synchronization means coordinating processes so that when they access shared data, they do not step on each other's toes. The goal: make sure that operations on shared data happen one at a time, even though the processes themselves run concurrently.

Real-world analogy — Two people editing the same Google Doc paragraph at once — without conflict resolution, one person's sentence vanishes when the other saves. Google Docs handles this with locks behind the scenes. That is exactly what synchronization primitives do in an OS.

2. The Critical Section Problem

The section of code in each process where the shared variable is accessed is called the critical section (CS). To prevent race conditions, we need a protocol that ensures only one process is in its critical section at any time.

Structure of a Typical Process

Every process that uses shared data has this structure:

```
do {
   entry_section      // Request permission to enter CS
      critical_section   // Access shared data
   exit_section       // Release permission
      remainder_section  // Other work (not touching shared data)
} while (true);
```

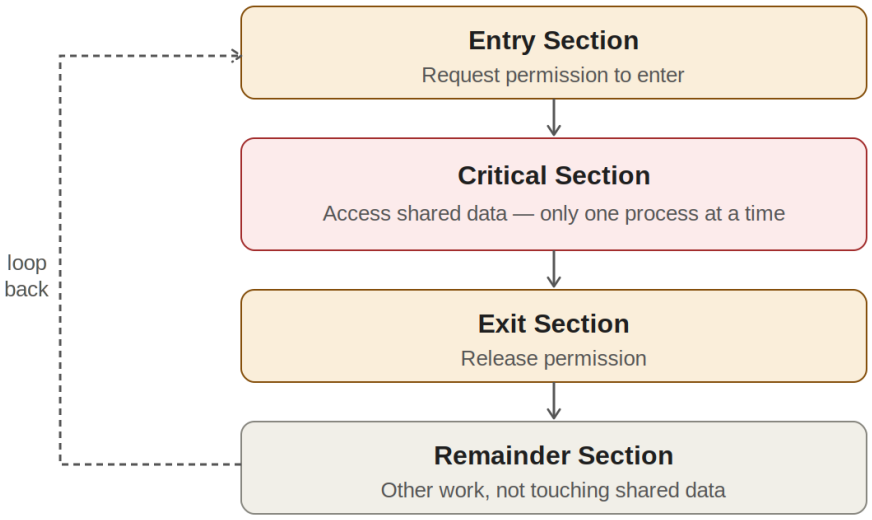


Figure 3.2 — The four sections of a process using shared data.

The Three Requirements for a Correct Solution

Any solution to the critical section problem must satisfy these three properties. Remember them — they are almost always asked in exams.

11. Mutual Exclusion. If one process is executing in its critical section, no other process can be in its critical section at the same time. This is the whole point of the exercise.
12. Progress. If no process is in the critical section and some processes want to enter, the decision of which one enters next cannot be postponed indefinitely. No false delays — if the CS is free and someone wants in, someone must get in soon.
13. Bounded Waiting. There must be a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its own. No starvation — you do not wait forever.

3. Peterson's Solution

Peterson's solution is a classic software-only algorithm that works for two processes (say, P0 and P1). It uses only two shared variables:

- **int turn;** — indicates whose turn it is to enter the CS.
- **boolean flag[2];** — flag[i] is true if process Pi wants to enter its CS.

The Algorithm

For process Pi (where j = 1 − i, the other process):

```
do {
   flag[i] = true;          // I want to enter
   turn = j;                // But let the other guy go first (politeness)

   while (flag[j] && turn == j)
      ;                     // Busy wait

   // ---- critical section ----

   flag[i] = false;         // I'm done; I don't want to enter anymore

   // ---- remainder section ----
} while (true);
```

Why It Works

- **Mutual exclusion:** Both P0 and P1 can set their flags, but only one value of turn exists at any time. Whoever sets turn last ends up being the one who waits — so only one of them passes the while loop.
- **Progress:** If the other process is not interested (flag[j] == false), you enter immediately. If it is interested but the turn is yours (turn == i), you also enter.
- **Bounded waiting:** After a process exits its CS, it sets its own flag to false. So the other process, on its next attempt, passes immediately. Waiting is bounded to exactly one turn.

The Catch

Peterson's solution is a beautiful teaching tool, but it has real-world limitations:

- It only works for two processes. Generalizing to n processes is much more complex.
- It relies on the assumption that memory reads and writes are atomic — which is not guaranteed on modern CPUs with multiple cores and caches.
- It uses busy waiting (spinning on while) — the waiting process keeps the CPU busy doing nothing useful. This is called a spinlock and wastes CPU cycles.

4. Synchronization Hardware

Software-only solutions like Peterson's are limited. Modern systems rely on special atomic hardware instructions provided by the CPU — instructions that complete in a single, uninterruptible step.

Test-and-Set Instruction

TestAndSet(target) atomically reads the value of target, sets it to true, and returns the old value. Since the whole thing happens atomically, no other process can squeeze in between the read and the write.

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;   // Read old value
   *target = true;         // Set to true
   return rv;              // Return old value
}
```

Usage:

```
do {
   while (TestAndSet(&lock))
      ;                 // Busy wait until lock is false

   // ---- critical section ----

   lock = false;

   // ---- remainder section ----
} while (true);
```

If lock is false (nobody is in CS), TestAndSet returns false and sets lock to true. You enter. The next process finds lock == true and spins. When you exit, you set lock = false; the waiter's TestAndSet returns false and it enters.

Compare-and-Swap (CAS) Instruction

A more flexible atomic instruction. CompareAndSwap(target, expected, new_value) atomically reads the current value of target; if it equals expected, it replaces it with new_value; and returns the original value. Used widely in lock-free data structures. The x86 CPU implements this as the CMPXCHG instruction.

5. Mutex Locks

Since busy-waiting is wasteful, OS designers built higher-level abstractions. The simplest is the mutex lock (short for mutual exclusion). A mutex has two operations:

- **acquire()** — if the lock is free, take it; otherwise wait (possibly by sleeping, not spinning).
- **release()** — free the lock so someone else can take it.

```
do {
   acquire();
```

```
    // ---- critical section ----

    release();

    // ---- remainder section ----
} while (true);
```

Real-world analogy — A mutex is like the key to a single-occupancy washroom. You pick up the key (acquire), use the washroom (critical section), then hang the key back (release). Anyone else who wants the washroom waits for the key.

In Linux, the pthread library provides pthread_mutex_lock() and pthread_mutex_unlock(). We will use these in Lab Program 16.

6. Semaphores

A semaphore is a more powerful synchronization tool invented by Edsger Dijkstra in 1965. It is an integer variable with two atomic operations: wait() and signal() (originally called P and V, from Dutch words proberen "to test" and verhogen "to increment").

```
wait(S) {                // Also called P(S)
    while (S <= 0)
        ;                // Wait
    S--;
}

signal(S) {              // Also called V(S)
    S++;
}
```

Two Types of Semaphores

- **Binary semaphore** (also called mutex semaphore). Value is either 0 or 1 — works like a mutex lock. 1 means available; 0 means taken.
- **Counting semaphore.** Value can range over any integer. Useful for controlling access to a resource with N instances. Initially set to N.

How Counting Semaphores Work

Suppose 3 identical printers are shared among many processes. Initialize semaphore S = 3.

- A process calls wait(S) before using a printer → S becomes 2, 1, 0.
- When S = 0, the next wait(S) blocks — the process sleeps until a printer is free.
- When a process finishes, it calls signal(S) → S increments, waking up one waiting process.

Counting Semaphore S = 3 (three printers available)

Step	Operation	Value of S
1	P1 calls wait(S)	S = 2
2	P2 calls wait(S)	S = 1
3	P3 calls wait(S)	S = 0
4	P4 calls wait(S) — blocked	S = 0 (P4 sleeps)
5	P5 calls wait(S) — blocked	S = 0 (P5 sleeps)
6	P1 calls signal(S)	S = 1 (wakes P4)
7	P4 resumes, S decrements	S = 0
8	P2 calls signal(S)	S = 1 (wakes P5)

S always reflects the number of free resources.

Figure 3.3 — Trace of wait() and signal() operations on a counting semaphore.

Semaphore Without Busy Waiting

The naive while (S <= 0); implementation still wastes CPU. Modern OS implementations use a waiting queue:

```
wait(S) {
   S.value--;
   if (S.value < 0) {
      add this process to S.list;
      block();            // Put to sleep — no CPU spinning
   }
}

signal(S) {
   S.value++;
   if (S.value <= 0) {
      remove a process P from S.list;
      wakeup(P);          // Make P ready
   }
}
```

With this version, a process waiting for the semaphore is put to sleep and does not consume CPU. When signal is called, one sleeping process is woken up.

7. Classical Synchronization Problems

Three textbook problems that every OS course covers. They are abstract but represent very real patterns in operating systems.

7.1 The Producer-Consumer Problem (Bounded Buffer)

A producer generates items and puts them in a shared buffer of size N. A consumer takes items out. We need to ensure: producer waits if the buffer is full; consumer waits if the buffer is empty; only one process accesses the buffer at a time.

Solution using three semaphores:

```
semaphore mutex = 1;        // For mutual exclusion on buffer
semaphore empty = N;        // Count of empty slots
semaphore full = 0;         // Count of filled slots

// Producer
do {
   produce_item();
   wait(empty);             // Wait if buffer is full
   wait(mutex);             // Lock the buffer
   add_item_to_buffer();
   signal(mutex);           // Unlock
   signal(full);            // One more item available
} while (true);

// Consumer
do {
   wait(full);              // Wait if buffer is empty
   wait(mutex);             // Lock the buffer
   remove_item_from_buffer();
   signal(mutex);           // Unlock
   signal(empty);           // One more slot empty
   consume_item();
} while (true);
```

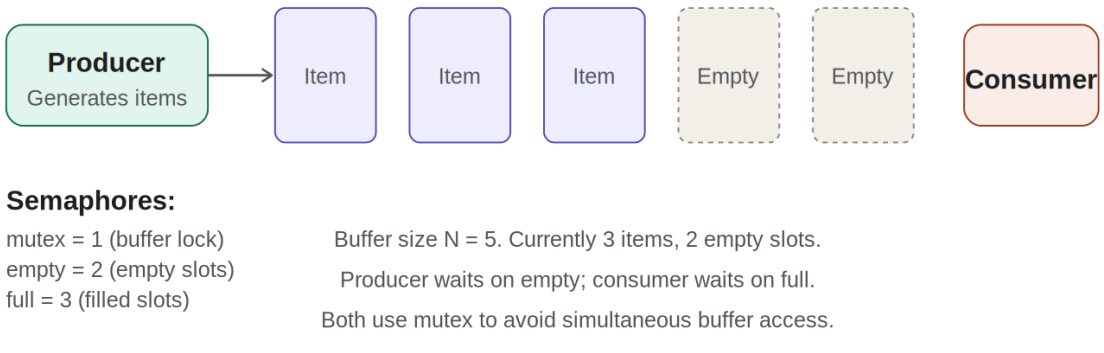


Figure 3.4 — Producer-consumer with a bounded buffer of size 5.

7.2 The Readers-Writers Problem

A database is shared by multiple readers and writers. Multiple readers can read at the same time (no harm), but a writer must have exclusive access (no other reader or writer should be active).

Solution (first readers-writers problem — readers priority):

```
semaphore rw_mutex = 1;      // Write lock
semaphore mutex = 1;         // Protect read_count
int read_count = 0;

// Writer
do {
   wait(rw_mutex);
   // ---- write ----
   signal(rw_mutex);
} while (true);

// Reader
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);      // First reader locks out writers
   signal(mutex);

   // ---- read ----

   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);    // Last reader releases write lock
   signal(mutex);
} while (true);
```

The trick: only the first reader needs to acquire rw_mutex to block writers; subsequent readers just bump read_count. Only the last reader releases it. The mutex semaphore protects read_count itself from race conditions. This is called "readers priority" because if readers keep coming, writers may starve.

7.3 The Dining Philosophers Problem

Five philosophers sit around a circular table. Between each pair of philosophers is one chopstick (so 5 chopsticks total). A philosopher alternates between thinking and eating. To eat, a philosopher needs both the chopstick on their left and the chopstick on their right.

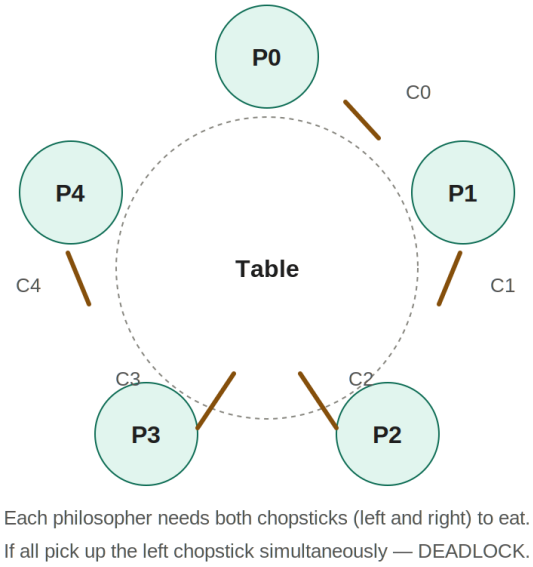


Figure 3.5 — Five philosophers with one chopstick between each pair.

Naive solution:

```
semaphore chopstick[5];     // All initialized to 1

// Philosopher i
do {
   wait(chopstick[i]);             // Pick up left
   wait(chopstick[(i+1) % 5]);     // Pick up right

   // ---- eat ----

   signal(chopstick[i]);           // Put down left
   signal(chopstick[(i+1) % 5]);   // Put down right

   // ---- think ----
} while (true);
```

The problem: if all five philosophers simultaneously pick up their left chopstick, all five wait(chopstick[(i+1) % 5]) calls block forever. This is deadlock — no one can proceed because each is holding one resource and waiting for another.

Solutions to prevent deadlock:

14. Allow at most 4 philosophers at the table at once (so at least one can eat).
15. Pick up both chopsticks atomically (all-or-nothing).
16. Asymmetric approach — odd philosophers pick up left first, even ones pick up right first.

8. Thread Synchronization Using Mutex and Semaphore

In Linux C programs, the pthread library gives you both tools. Here is a quick reference.

Mutex (pthread)

```
#include <pthread.h>

pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

// In each thread:
pthread_mutex_lock(&lock);
// ---- critical section ----
pthread_mutex_unlock(&lock);

pthread_mutex_destroy(&lock);
```

Semaphore (POSIX)

```
#include <semaphore.h>

sem_t sem;
sem_init(&sem, 0, 1);   // Third argument = initial value

// In each thread:
sem_wait(&sem);
// ---- critical section ----
sem_post(&sem);         // sem_post = signal

sem_destroy(&sem);
```

Mutex vs Semaphore — Key Differences

Aspect	Mutex	Semaphore
Value	Binary (0 or 1)	Any non-negative integer
Ownership	Only the locking thread can unlock	Any thread can signal
Use case	Mutual exclusion	Signalling, counting resources
Typical role	Lock	Counter

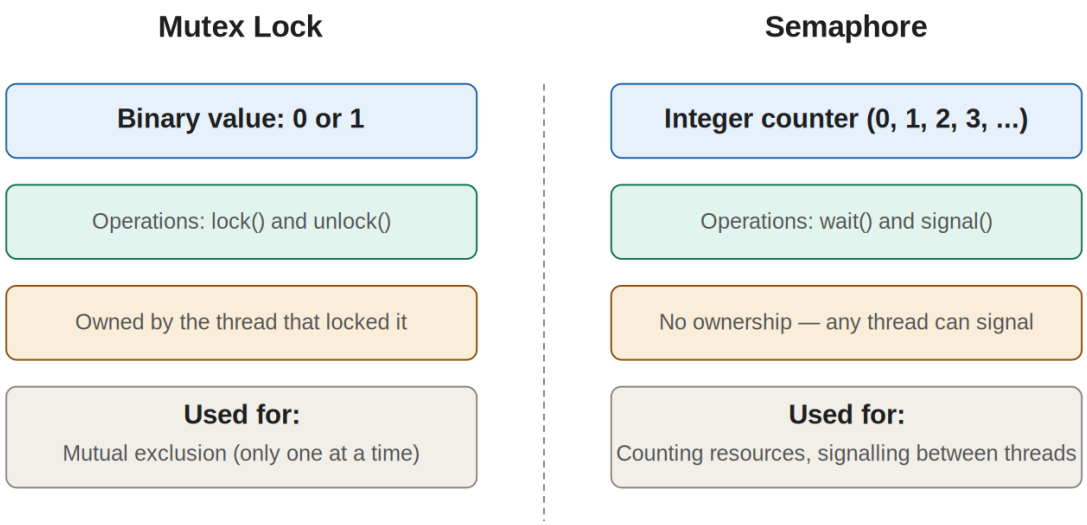


Figure 3.6 — Mutex vs Semaphore at a glance.

A mutex is essentially a specialized binary semaphore with ownership rules. Use a mutex when you want mutual exclusion; use a counting semaphore when you are tracking "how many of something" are available.

9. Numerical Questions with Solutions

Numerical 1 — Peterson's Algorithm Dry Run

Q: Using Peterson's algorithm, two processes P0 and P1 want to enter the critical section. Initially flag[0] = flag[1] = false. Trace the execution if P0 executes entry code fully, then P1 starts.

Solution: Initial: flag = {false, false}, turn = ?

- Step 1: P0 executes flag[0] = true; → flag = {true, false}
- Step 2: P0 executes turn = 1; → turn = 1
- Step 3: P0 evaluates flag[1] && turn == 1 → false && true = false → P0 enters CS

Now P1 starts:

- Step 4: P1 executes flag[1] = true; → flag = {true, true}
- Step 5: P1 executes turn = 0; → turn = 0
- Step 6: P1 evaluates flag[0] && turn == 0 → true && true = true → P1 waits

After P0 finishes CS and runs exit section:

- Step 7: P0 executes flag[0] = false; → flag = {false, true}
- Step 8: P1 re-evaluates flag[0] && turn == 0 → false && true = false → P1 enters CS

Conclusion: Mutual exclusion holds — only one process in CS at any time. ✓

Numerical 2 — Semaphore Value Trace

Q: A counting semaphore S is initialized to 5. The following operations are performed in order: three wait(), two signal(), four wait(), three signal(). What is the final value of S? How many processes are blocked?

Solution:

Step	Operation	S value
0	(initial)	5
1	wait()	4
2	wait()	3
3	wait()	2
4	signal()	3
5	signal()	4
6	wait()	3
7	wait()	2
8	wait()	1
9	wait()	0
10	signal()	1
11	signal()	2
12	signal()	3

Net change: started with 5, had 7 waits and 5 signals. Final S = 5 − 7 + 5 = 3. No process is blocked (S never went negative). Answer: Final S = 3, blocked = 0.

Numerical 3 — Detecting Block Through Semaphore

Q: A counting semaphore S is initialized to 7. After executing 20 wait() and 15 signal() operations, what is the value of S? How many processes are blocked?

Solution: Using the formula:

Final S = Initial S − (number of waits) + (number of signals)

Final S = 7 − 20 + 15 = 2

Tracking step by step: after 7 waits, S = 0. The next 13 waits cause 13 processes to block. But 15 signals arrive — each wakes a blocked process. So 13 signals wake the 13 blocked processes. Remaining 2 signals go to the counter.

Answer: Final S = 2, blocked processes = 0.

Numerical 4 — Buffer State in Producer-Consumer

Q: A bounded buffer has size N = 5. Initially it is empty. The producer produces 3 items, then the consumer consumes 1 item, then the producer produces 2 more items. What are the values of the three semaphores (mutex, empty, full)?

Solution: Initial: mutex = 1, empty = 5, full = 0, buffer has 0 items.

After producer produces 3 items:

- empty decremented 3 times: empty = 5 − 3 = 2
- full incremented 3 times: full = 0 + 3 = 3
- mutex stays at 1 (each iteration locks and unlocks)

After consumer consumes 1 item:

- full decremented 1 time: full = 3 − 1 = 2
- empty incremented 1 time: empty = 2 + 1 = 3

After producer produces 2 more items:

- empty decremented 2 times: empty = 3 − 2 = 1
- full incremented 2 times: full = 2 + 2 = 4

Final values: mutex = 1, empty = 1, full = 4. Buffer has 4 items. Sanity check: empty + full = 1 + 4 = 5 = N ✓

Numerical 5 — Dining Philosophers Deadlock

Q: In the naive dining philosophers solution with 5 philosophers, show one specific sequence of actions that leads to deadlock.

Solution: Label chopsticks C0, C1, C2, C3, C4 between philosophers P0–P4.

Step	Action	Result
1	P0 picks up C0	P0 holds C0
2	P1 picks up C1	P1 holds C1
3	P2 picks up C2	P2 holds C2
4	P3 picks up C3	P3 holds C3
5	P4 picks up C4	P4 holds C4
6	P0 tries to pick up C1	Blocked (P1 has it)
7	P1 tries to pick up C2	Blocked (P2 has it)
8	P2 tries to pick up C3	Blocked (P3 has it)
9	P3 tries to pick up C4	Blocked (P4 has it)
10	P4 tries to pick up C0	Blocked (P0 has it)

Every philosopher is holding one chopstick and waiting for another that is held by someone else. The circular dependency (P0 → P1 → P2 → P3 → P4 → P0) cannot be broken. DEADLOCK.

Numerical 6 — Counting Waiting Processes

Q: A binary semaphore mutex is initialized to 1. Ten processes P1, P2, ..., P10 all try to enter their critical section in that exact order. None of them has exited yet. How many processes are waiting?

Solution:

- P1 calls wait(mutex): mutex = 1 → 0. P1 enters CS.
- P2 calls wait(mutex): mutex = 0 → blocked. 1 waiting.
- P3 calls wait(mutex): blocked. 2 waiting.
- ...
- P10 calls wait(mutex): blocked. 9 waiting.

Answer: 1 process in CS (P1), 9 processes blocked/waiting.

Numerical 7 — Producer-Consumer with Unequal Rates

Q: In a bounded buffer of size 10, the producer produces items every 2 ms, and the consumer consumes every 5 ms. Both start at time 0. After 20 ms, how many items are in the buffer?

Solution: In 20 ms:

- Number of items produced = 20 / 2 = 10 items
- Number of items consumed = 20 / 5 = 4 items
- Items in buffer (if no blocking) = 10 − 4 = 6 items

Buffer can hold only 10 items. Maximum items ever in buffer = max over time of (produced so far − consumed so far). At t = 20 ms: produced = 10, consumed = 4 → 6 in buffer. Buffer never exceeds 10, so no blocking occurs.

Final answer: 6 items in buffer.

10. Quick Revision — One-Page Summary

Here is everything in Unit 3 compressed for last-minute revision.

- **Race condition** — outcome depends on order of execution. Occurs when multiple processes access shared data without coordination.
- **Critical section (CS)** — code accessing shared data. Must be executed by only one process at a time.

Three requirements for a solution:

- Mutual exclusion — only one process in CS at a time.
- Progress — free CS cannot stall indefinitely.
- Bounded waiting — limit on waits before entering CS.
- **Peterson's solution** — two-process software algorithm using flag[] and turn. Works but uses busy waiting and is limited to 2 processes.

Hardware support:

- TestAndSet(target) — atomic read-then-set.
- CompareAndSwap(target, expected, new) — atomic compare-and-update.
- **Mutex lock** — binary lock with acquire() and release(). Simple mutual exclusion.
- **Semaphore** — integer with atomic wait() (decrement or block) and signal() (increment or wake).
- Binary semaphore: 0 or 1 (like mutex).
- Counting semaphore: tracks N-instance resources.

Classical problems:

- Producer-Consumer — uses 3 semaphores (mutex, empty, full).
- Readers-Writers — readers can share; writers need exclusive access.
- Dining Philosophers — 5 philosophers, 5 chopsticks; naive solution deadlocks.

Key formulas:

- Final semaphore value = Initial − waits + signals (when no blocking)
- Producer-Consumer sanity: empty + full = N (buffer size)
- Number of processes that can be in CS simultaneously = initial value of semaphore
- **pthread API:** Mutex — pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock. Semaphore — sem_init, sem_wait, sem_post.
- **Deadlock** — circular wait where each process holds one resource and needs another held by someone else.

11. Practice Questions

Section A: Multiple Choice Questions (1 mark each)

Q1. A situation where the result depends on the order in which processes execute is called:

a) Deadlock
b) Race condition
c) Starvation
d) Mutual exclusion
Answer: (b)

Q2. The section of code where shared data is accessed is called:

a) Entry section
b) Exit section
c) Critical section
d) Remainder section
Answer: (c)

Q3. Which of the following is NOT a requirement for a critical section solution?

a) Mutual exclusion
b) Progress
c) Bounded waiting
d) Preemption
Answer: (d)

Q4. Peterson's solution uses how many shared variables?

a) 1
b) 2
c) 3
d) 4
Answer: (b) — flag[] (one boolean per process) and turn.

Q5. Peterson's solution works for how many processes?

a) 1
b) 2
c) Any number
d) Up to N
Answer: (b)

Q6. Busy waiting (spinning) is a problem because:

a) It uses too much memory
b) It keeps the CPU busy doing nothing useful
c) It causes deadlock
d) It is not portable
Answer: (b)

Q7. Which hardware instruction is used for atomic synchronization?

a) Add
b) Test-and-Set
c) Move
d) Branch
Answer: (b)

Q8. A mutex lock can have how many possible states?

a) 1
b) 2
c) 3
d) Many
Answer: (b) — locked or unlocked.

Q9. Who invented the semaphore?

a) Linus Torvalds
b) Edsger Dijkstra
c) Dennis Ritchie
d) Andrew Tanenbaum
Answer: (b)

Q10. The two atomic operations on a semaphore are:

a) lock and unlock
b) wait and signal
c) acquire and release
d) get and put
Answer: (b) — also called P and V.

Q11. A counting semaphore initialized to 5 allows:

a) 1 process at a time
b) Up to 5 processes simultaneously
c) 5 processes total
d) No processes
Answer: (b)

Q12. In the producer-consumer problem, the semaphore "empty" is initialized to:

a) 0
b) 1
c) N (buffer size)

d) Infinity
Answer: (c)

Q13. In the producer-consumer problem, the semaphore "full" is initialized to:

a) 0
b) 1
c) N (buffer size)
d) -1
Answer: (a)

Q14. In the readers-writers problem, multiple readers can access the data because:

a) Reading does not modify data
b) Reading is faster than writing
c) Writers do not exist
d) Readers always come first
Answer: (a)

Q15. In the dining philosophers problem, deadlock occurs when:

a) All philosophers eat at once
b) All philosophers pick up their left chopstick at the same time
c) One philosopher has all chopsticks
d) No philosopher is hungry
Answer: (b)

Q16. How many chopsticks are there in the dining philosophers problem with 5 philosophers?

a) 5
b) 10
c) 4
d) 6
Answer: (a) — one chopstick between each adjacent pair.

Q17. Which is NOT a difference between mutex and semaphore?

a) Mutex has ownership; semaphore does not
b) Semaphore can be a counter; mutex is binary
c) Mutex uses lock/unlock; semaphore uses wait/signal
d) Both can only be in two states
Answer: (d) — counting semaphore can hold many values.

Q18. In pthread, which function acquires a mutex?

a) pthread_lock()
b) pthread_mutex_lock()
c) pthread_acquire()
d) pthread_get()
Answer: (b)

Q19. In POSIX semaphores, sem_post() is equivalent to which classical operation?

a) wait
b) signal
c) lock
d) acquire
Answer: (b)

Q20. If a counting semaphore S = 4 and 6 wait() calls have been made (no signals), how many processes are blocked?

a) 0
b) 1
c) 2
d) 6
Answer: (c) — first 4 succeed, last 2 are blocked.

Section B: Short Answer Questions (2 marks each)

Q1. Define race condition with an example.

Ans: A race condition is a situation where the final outcome of concurrent execution depends on the precise timing of operations. Example: two processes incrementing a shared counter concurrently can produce the wrong final value because the operation "counter = counter + 1" actually happens in three machine steps that can be interleaved.

Q2. What is a critical section?

Ans: A critical section is the portion of a process's code where shared data is accessed. To avoid race conditions, only one process should be allowed inside its critical section at any given time.

Q3. List the three requirements for a critical section solution.

Ans: Mutual exclusion (only one process in CS at a time), Progress (a free CS must be granted to a willing process without indefinite delay), and Bounded waiting (a process must not wait forever to enter its CS).

Q4. What is busy waiting? Why is it bad?

Ans: Busy waiting (spinning) is when a process repeatedly checks a condition in a loop until it becomes true, instead of sleeping. It is bad because the CPU stays fully occupied doing no useful work — wasting CPU cycles and possibly preventing the process holding the lock from running.

Q5. What is a mutex lock? Mention its two operations.

Ans: A mutex (mutual exclusion) lock is a synchronization primitive that allows only one thread or process to hold it at a time. Its two operations are acquire() (lock — wait if not free) and release() (unlock — let the next waiter in).

Q6. What is a semaphore? Differentiate binary and counting semaphores.

Ans: A semaphore is an integer variable that supports two atomic operations: wait() and signal(). A binary semaphore takes only the values 0 or 1 (similar to a mutex). A counting semaphore can take any non-negative integer value, useful for tracking N instances of a resource.

Q7. Define wait() and signal() operations on a semaphore.

Ans: wait(S): if S > 0, decrement S; otherwise the calling process is blocked until S becomes positive. signal(S): increment S; if any process was blocked on this semaphore, one of them is woken up and allowed to proceed.

Q8. Briefly state the producer-consumer problem.

Ans: A producer process generates items and puts them in a shared bounded buffer. A consumer process takes items out. The producer must wait when the buffer is full; the consumer must wait when it is empty. Both must avoid simultaneous access to the buffer.

Q9. List the three semaphores used in the producer-consumer solution.

Ans: mutex (binary semaphore for mutual exclusion on the buffer), empty (counting semaphore initialized to N — the number of empty slots), and full (counting semaphore initialized to 0 — the number of filled slots).

Q10. State the readers-writers problem.

Ans: A shared data object (database) is accessed by multiple readers and writers. Several readers can read concurrently (no harm), but a writer must have exclusive access — no other reader or writer should be active while a writer is writing.

Q11. State the dining philosophers problem.

Ans: Five philosophers sit around a circular table with one chopstick between each pair. Each philosopher alternates between thinking and eating. To eat, a philosopher needs both the left and right chopstick. The challenge is to design a protocol that prevents deadlock and starvation.

Q12. Differentiate mutex and semaphore.

Ans: A mutex is a binary lock with strict ownership — only the thread that locked it can unlock it. A semaphore is an integer counter without ownership — any thread can call signal() to release it. Mutex is used for mutual exclusion; counting semaphores are used for resource counting and signalling.

Q13. Define deadlock.

Ans: Deadlock is a situation where two or more processes are blocked forever, each waiting for a resource that another holds. Example: in the naive dining philosophers solution, if all five pick up their left chopstick at the same time, all wait forever for their right chopstick.

Section C: Long Answer Questions (5 marks each)

Q1. Explain the race condition with an example. Show how interleaved execution causes incorrect output.

Approach: *Define race condition. Take the counter increment/decrement example. Break each statement into three machine instructions (read, modify, write). Trace an unlucky interleaving where one process's update is lost. State the conclusion that the final result depends on timing. Include Figure 3.1.*

Q2. Define the critical section problem and explain its three requirements.

Approach: *Define critical section. Show the standard process structure with entry section, critical section, exit section, and remainder section. Explain mutual exclusion, progress, and bounded waiting clearly with one-line examples. Include Figure 3.2.*

Q3. Explain Peterson's solution and prove that it satisfies all three requirements.

Approach: *Define the shared variables flag[] and turn. Write the algorithm for process Pi. Show how mutual exclusion is preserved (only one passes the while loop). Show progress (a willing process always enters when CS is free). Show bounded waiting (after exit, the other process gets in next). Mention the limitation that it is restricted to two processes and uses busy waiting.*

Q4. Discuss the hardware solution to the critical section problem using TestAndSet.

Approach: *Describe the TestAndSet instruction with pseudo-code. Show how it is used inside a do-while loop to enforce mutual exclusion. Explain the atomicity guarantee. Mention CompareAndSwap as a more flexible alternative. Conclude that hardware-supported atomic instructions are the foundation on which all higher-level synchronization is built.*

Q5. Explain semaphores. Differentiate between binary and counting semaphores with examples.

Approach: *Define semaphore as an integer with two atomic operations wait() and signal(). Give the naive busy-wait pseudo-code. Distinguish binary (0 or 1, behaves like mutex) and counting (any non-negative value, used for N-instance resources). Use a printer pool (N=3) example. Mention the queue-based implementation that avoids busy waiting. Include Figure 3.3.*

Q6. Describe the producer-consumer problem and provide its solution using semaphores.

Approach: *Describe the bounded buffer scenario. List the three semaphores (mutex, empty, full) and their initial values. Write the producer code (wait empty → wait mutex → add → signal mutex → signal full) and the consumer code (mirror). Explain why each step is needed. Include Figure 3.4.*

Q7. Explain the readers-writers problem with its solution using semaphores.

Approach: *Define the problem. State the rules: many readers can read together, only one writer at a time and no readers during write. Give the solution with rw_mutex, mutex, and read_count. Trace through the cases — first reader locks rw_mutex, last reader releases it. Mention the readers-priority issue (writer starvation) as a discussion point.*

Q8. Describe the dining philosophers problem. Show how the naive solution leads to deadlock and propose three remedies.

Approach: *Describe the setup. Write the naive solution where each philosopher does wait(left), wait(right), eat, signal(left), signal(right). Trace the deadlock scenario where all philosophers pick up their left chopstick simultaneously. List remedies: at most 4 philosophers at the table, atomic both-or-nothing pickup, and asymmetric (odd/even) approach. Include Figure 3.5.*

Q9. Compare mutex and semaphore. Explain how each is used in pthread programming.

Approach: *Make a clear comparison table covering value, ownership, operations, use cases. Provide pthread_mutex_init / lock / unlock / destroy snippets and sem_init / sem_wait / sem_post / sem_destroy snippets. Conclude with which one to choose in which situation. Include Figure 3.6.*

UNIT 4

CPU Scheduling

Contents

1. Basic Scheduling Concepts

The CPU-I/O Burst Cycle

Every process alternates between two activities: CPU bursts (when the process is actively computing) and I/O bursts (when the process is waiting for input or output). A process typically looks like: CPU burst → I/O burst → CPU burst → I/O burst → ... → final CPU burst → exit.

Some processes are CPU-bound (long CPU bursts, short I/O bursts) — think scientific simulations or video encoding. Others are I/O-bound (short CPU bursts, long I/O bursts) — think text editors or database servers. A good scheduler handles both types well.

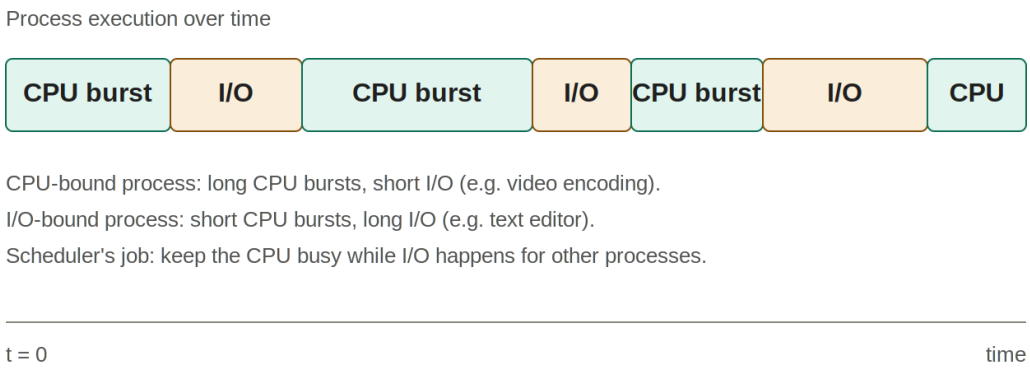


Figure 4.1 — The CPU-I/O burst cycle.

CPU Scheduler and Dispatcher

- **CPU Scheduler (short-term scheduler) —** the OS component that picks which ready process gets the CPU next, based on a scheduling algorithm.
- **Dispatcher —** the OS component that actually hands over the CPU to the chosen process. It performs the context switch, switches to user mode, and jumps to the right instruction in the user program.

The time taken by the dispatcher to do all this is called dispatch latency. It is pure overhead and we want to minimize it.

When Does Scheduling Happen?

Scheduling decisions may take place under four circumstances:

17. A process switches from running to waiting state (e.g., blocked for I/O).
18. A process switches from running to ready state (e.g., interrupted by a timer).
19. A process switches from waiting to ready (e.g., I/O completes).
20. A process terminates.

Preemptive vs Non-Preemptive Scheduling

- **Non-preemptive scheduling —** once the CPU is given to a process, it keeps it until it terminates or voluntarily moves to waiting state. Simple but unfair — a long process can hog the CPU. Scheduling happens only at cases 1 and 4 above.
- **Preemptive scheduling —** the OS can forcibly take the CPU away from a running process (usually when a higher-priority process becomes ready, or when a time slice expires). More complex but fairer. Scheduling happens in all four cases.

2. Scheduling Criteria

How do we judge whether a scheduling algorithm is good? Five standard criteria:

- **1. CPU Utilization.** Fraction of time the CPU is busy doing useful work. We want this maximized (ideally 100%, realistically 40–90%).
- **2. Throughput.** Number of processes completed per unit time. Higher is better.
- **3. Turnaround Time (TAT).** Time from submission to completion of a process. Formula: TAT = Completion Time − Arrival Time.
- **4. Waiting Time (WT).** Total time a process spends in the ready queue. Formula: WT = TAT − Burst Time.
- **5. Response Time.** Time from submission until the process first responds. Critical for interactive systems.

Goal — Maximize CPU utilization and throughput. Minimize turnaround time, waiting time, and response time.

Key Formulas — Memorize These

Quantity	Formula
Completion Time (CT)	Time when process finishes (read from Gantt chart)
Turnaround Time (TAT)	TAT = CT − AT
Waiting Time (WT)	WT = TAT − BT
Response Time (RT)	RT = First time on CPU − AT
Average TAT	Sum of all TATs / number of processes
Average WT	Sum of all WTs / number of processes

Where AT = Arrival Time, BT = Burst Time, CT = Completion Time.

3. Scheduling Algorithms

3.1 First Come First Served (FCFS)

Simplest possible algorithm. Whoever arrives in the ready queue first gets the CPU first. Non-preemptive. Think of it as a queue at a ticket counter.

Implementation: a FIFO queue. New arrivals go to the tail; scheduler picks from the head.

- **Pros:** Simple, easy to implement, fair in the order-of-arrival sense.
- **Cons:** A long process at the front makes everyone else wait. This is the convoy effect — short processes pile up behind a long one. Leads to high average waiting time.

Example

Three processes that all arrive at t = 0:

Process	Arrival Time	Burst Time
P1	0	24
P2	0	3
P3	0	3

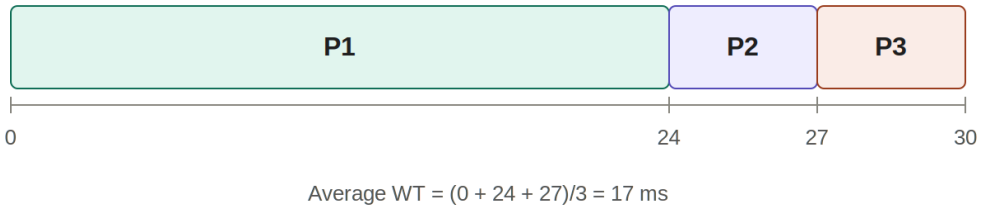


Figure 4.2 — FCFS Gantt chart.

Computation table:

Process	AT	BT	CT	TAT = CT − AT	WT = TAT − BT
P1	0	24	24	24	0
P2	0	3	27	27	24
P3	0	3	30	30	27

Average TAT = (24 + 27 + 30)/3 = 27 ms

Average WT = (0 + 24 + 27)/3 = 17 ms

Notice: if P2 and P3 had arrived first, the average WT would be much lower. FCFS depends heavily on arrival order.

3.2 Shortest Job First — Non-Preemptive (SJF)

When the CPU becomes free, pick the process in the ready queue with the shortest next CPU burst. If there is a tie, break it with FCFS.

Provable result: SJF gives the minimum average waiting time among all scheduling algorithms. It is optimal in that sense. Problem: the OS does not know the future burst, so it must estimate based on past bursts.

Example

Process	AT	BT
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Step-by-step execution:

- At t = 0: only P1 is ready. P1 runs (non-preemptive, runs to completion).
- At t = 7: P2, P3, P4 are all ready. Pick shortest burst → P3 (burst 1).
- At t = 8: P2 and P4 are ready with same burst (4). Tie broken by arrival — P2 came first.
- At t = 12: P4 runs.
- At t = 16: done.

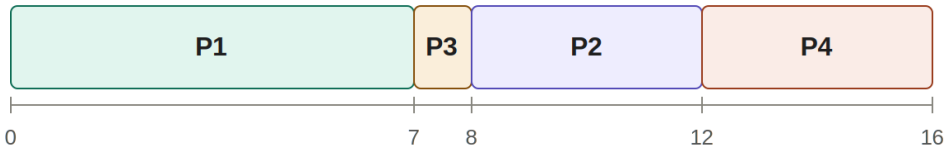


Figure 4.3 — SJF non-preemptive Gantt chart.

Process	AT	BT	CT	TAT	WT
P1	0	7	7	7	0
P2	2	4	12	10	6
P3	4	1	8	4	3
P4	5	4	16	11	7

Average TAT = (7 + 10 + 4 + 11)/4 = 8 ms

Average WT = (0 + 6 + 3 + 7)/4 = 4 ms

3.3 Shortest Remaining Time First — Preemptive SJF (SRTF)

The preemptive version of SJF. Whenever a new process arrives in the ready queue, check its burst against the remaining burst of the currently running process. If the new one is shorter, preempt the running process and switch.

Same example as before:

Process	AT	BT
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Step-by-step trace:

- t = 0: only P1. P1 runs. P1 remaining = 7.
- t = 2: P2 arrives (burst 4). P1 remaining = 5. P2's 4 < 5, so preempt P1. P2 runs.
- t = 4: P3 arrives (burst 1). P2 remaining = 2. P3's 1 < 2, so preempt P2. P3 runs.
- t = 5: P3 finishes. Choices: P1=5, P2=2, P4=4. Shortest = P2. P2 runs.
- t = 7: P2 finishes. Choices: P1=5, P4=4. P4 runs.
- t = 11: P4 finishes. P1 runs.
- t = 16: P1 finishes.

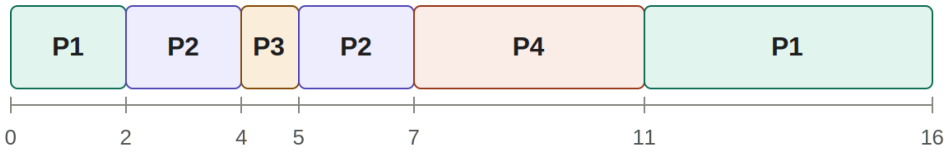


Figure 4.4 — SRTF (preemptive SJF) Gantt chart.

Process	AT	BT	CT	TAT	WT
P1	0	7	16	16	9
P2	2	4	7	5	1
P3	4	1	5	1	0
P4	5	4	11	6	2

Average TAT = (16 + 5 + 1 + 6)/4 = 7 ms

Average WT = (9 + 1 + 0 + 2)/4 = 3 ms

Compare to non-preemptive SJF (avg WT = 4 ms) — SRTF is better when arrival times differ.

3.4 Priority Scheduling

Each process has a priority number (say, 1 to 10). The CPU is given to the process with the highest priority. Tie broken by FCFS.

> **Convention alert —** In Silberschatz, lower numbers mean higher priority (priority 1 > priority 10). Some textbooks use the opposite. Always read the question carefully.

Priority can be assigned:

- Internally — based on time limits, memory requirements, I/O-to-CPU ratio.
- Externally — set by users or administrators.

Priority scheduling can be preemptive (higher-priority arrival preempts) or non-preemptive (higher-priority arrival just goes to the head of the queue).

Starvation problem: low-priority processes may wait forever if high-priority processes keep arriving. Solution: aging — gradually increase the priority of a process the longer it waits in the ready queue.

Example (non-preemptive, lower number = higher priority)

Process	AT	BT	Priority
P1	0	10	3
P2	0	1	1
P3	0	2	4
P4	0	1	5
P5	0	5	2

All arrive at t = 0. Order by priority: P2 (1) → P5 (2) → P1 (3) → P3 (4) → P4 (5).

Gantt chart: P2(0–1) | P5(1–6) | P1(6–16) | P3(16–18) | P4(18–19)

Process	CT	TAT	WT
P1	16	16	6
P2	1	1	0
P3	18	18	16
P4	19	19	18
P5	6	6	1

Average TAT = (16+1+18+19+6)/5 = 12 ms

Average WT = (6+0+16+18+1)/5 = 8.2 ms

3.5 Round Robin (RR)

Designed for time-sharing systems. Each process gets a small unit of CPU time called a time quantum (q) or time slice — usually 10–100 ms. After its quantum expires, the process is preempted and placed at the tail of the ready queue. The CPU picks the next process from the head.

Round Robin is inherently preemptive. Behaviour:

- If quantum is very large → behaves like FCFS.
- If quantum is very small → too many context switches, overhead dominates.
- Rule of thumb: 80% of CPU bursts should be shorter than the quantum.

Example (q = 4 ms)

Process	AT	BT
P1	0	5
P2	1	4
P3	2	2
P4	3	1

Step-by-step queue tracking:

- t = 0: [P1]. P1 runs 0–4 (1 ms remains).
- t = 4: P2, P3, P4 already in queue, P1 goes to back. Queue: [P2, P3, P4, P1]. P2 runs 4–8.
- t = 8: P2 done (had 4 = quantum exactly). Queue: [P3, P4, P1]. P3 runs 8–10 (done).
- t = 10: Queue: [P4, P1]. P4 runs 10–11 (done).
- t = 11: Queue: [P1]. P1 runs 11–12 (done).

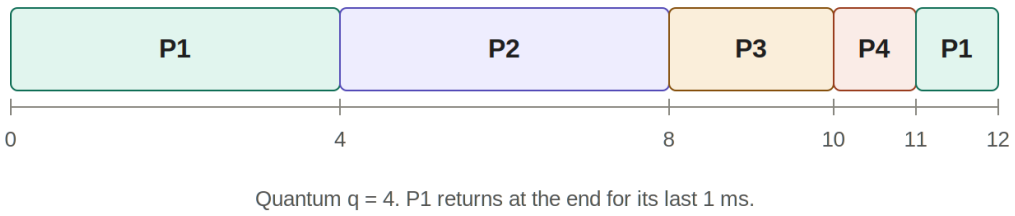


Figure 4.5 — Round Robin Gantt chart with quantum 4.

Process	AT	BT	CT	TAT	WT
P1	0	5	12	12	7
P2	1	4	8	7	3

P3	2	2	10	8	6
P4	3	1	11	8	7

Average TAT = (12 + 7 + 8 + 8)/4 = 8.75 ms

Average WT = (7 + 3 + 6 + 7)/4 = 5.75 ms

3.6 Multilevel Queue Scheduling

The ready queue is split into separate queues for different kinds of processes. For example:

- System processes (highest priority)
- Interactive processes
- Batch processes
- Student processes (lowest priority)

Each queue has its own scheduling algorithm. Between queues, fixed priority or time slicing applies. Downside: processes are permanently assigned to a queue and cannot move.

3.7 Multilevel Feedback Queue Scheduling

Like multilevel queue, but processes can move between queues based on their CPU-burst behaviour:

- New process enters the top (highest-priority) queue.
- If it uses up its full quantum, it is demoted to a lower queue (likely CPU-bound — punish it).
- If it releases the CPU before the quantum expires, it stays in the current queue (likely I/O-bound — reward it with fast response).

This is an adaptive scheme — the system learns which processes are interactive and which are CPU-bound, and adjusts automatically. It is the most general CPU scheduling algorithm.

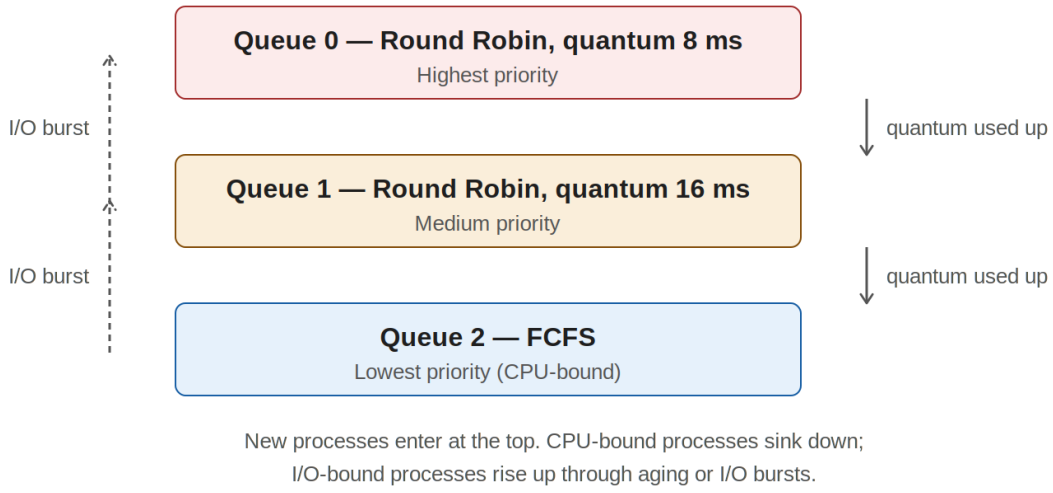


Figure 4.6 — Multilevel feedback queue with three priority levels.

4. Thread Scheduling

On systems with kernel-supported threads, it is the threads that are scheduled, not the processes. Two levels of scheduling:

- **Process-contention scope (PCS)** — for user-level threads. The thread library schedules threads within a process. User threads compete only with other threads of the same process.
- **System-contention scope (SCS)** — for kernel threads. All kernel threads in the system compete with each other for the CPU.

Linux and Windows use the one-to-one model, so they use SCS for all threads.

5. Multi-processor Scheduling

When there are multiple CPUs available, scheduling gets more complex. Two approaches:

- **Asymmetric multiprocessing** — one CPU (the master) handles all scheduling decisions, I/O, and system activities. The other CPUs just run user code. Simpler but the master becomes a bottleneck.
- **Symmetric multiprocessing (SMP)** — each CPU is self-scheduling. This is what modern systems (Linux, Windows, macOS) use.

Processor Affinity

When a process has been running on CPU 0, its data is loaded into CPU 0's cache. If the scheduler moves the process to CPU 1, all that cached data is wasted. To avoid this, most systems try to keep a process on the same CPU it ran on before. This is called processor affinity.

- **Soft affinity** — the OS tries to keep the process on the same CPU but may migrate it if needed.
- **Hard affinity** — the process specifies exactly which CPU(s) it can run on (set via system calls like sched_setaffinity() in Linux).

Load Balancing

On SMP with per-CPU queues, one CPU may be busy while others sit idle. Load balancing moves processes from busy CPUs to idle ones.

- **Push migration** — a separate process periodically checks each CPU's load and pushes processes from overloaded CPUs to underloaded ones.
- **Pull migration** — an idle CPU pulls a process from an overloaded CPU.

6. Numerical Questions with Solutions

Numerical 1 — FCFS with Different Arrival Times

Q: Calculate average TAT and WT using FCFS.

Process	AT	BT
P1	2	6
P2	5	2
P3	1	8
P4	0	3
P5	4	4

Solution: Order of arrival: P4 (0), P3 (1), P1 (2), P5 (4), P2 (5).

FCFS executes in arrival order. Gantt chart: | P4 (0–3) | P3 (3–11) | P1 (11–17) | P5 (17–21) | P2 (21–23) |

Process	AT	BT	CT	TAT	WT
P1	2	6	17	15	9
P2	5	2	23	18	16
P3	1	8	11	10	2
P4	0	3	3	3	0
P5	4	4	21	17	13

Average TAT = (15 + 18 + 10 + 3 + 17)/5 = 12.6 ms

Average WT = (9 + 16 + 2 + 0 + 13)/5 = 8.0 ms

Numerical 2 — SJF Non-Preemptive

Q: Given the following processes (all arrive at t = 0), calculate average WT using non-preemptive SJF.

Process	BT
P1	6
P2	8
P3	7
P4	3

Solution: All arrive at 0. Pick shortest first: P4 → P1 → P3 → P2.

Gantt chart: | P4 (0–3) | P1 (3–9) | P3 (9–16) | P2 (16–24) |

Process	AT	BT	CT	TAT	WT
P1	0	6	9	9	3
P2	0	8	24	24	16
P3	0	7	16	16	9
P4	0	3	3	3	0

Average TAT = (9 + 24 + 16 + 3)/4 = 13 ms

Average WT = (3 + 16 + 9 + 0)/4 = 7 ms

Numerical 3 — Preemptive SJF (SRTF)

Q: Using SRTF, calculate average WT.

Process	AT	BT
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Solution: Step-by-step trace:

- t = 0: only P1. P1 runs.
- t = 1: P2 arrives (4). P1 remaining = 7. 4 < 7, preempt. P2 runs.
- t = 2: P3 arrives (9). P2 remaining = 3. 3 < 9, continue P2.
- t = 3: P4 arrives (5). P2 remaining = 2. 2 < 5, continue P2.
- t = 5: P2 done. Remaining: P1=7, P3=9, P4=5. P4 runs.
- t = 10: P4 done. Remaining: P1=7, P3=9. P1 runs.
- t = 17: P1 done. P3 runs.
- t = 26: P3 done.

Gantt chart: | P1 (0–1) | P2 (1–5) | P4 (5–10) | P1 (10–17) | P3 (17–26) |

Process	AT	BT	CT	TAT	WT
P1	0	8	17	17	9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

Average TAT = (17 + 4 + 24 + 7)/4 = 13 ms

Average WT = (9 + 0 + 15 + 2)/4 = 6.5 ms

Numerical 4 — Priority Scheduling (Preemptive)

Q: Using preemptive priority (lower = higher), calculate average TAT and WT.

Process	AT	BT	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	1

Solution:

- t = 0–4: P1 runs (no higher-priority arrival until P5 at t=4).
- t = 4: P1 finishes. Pick P5 (pri 1, highest).
- t = 6: P5 finishes. Pick P2 (pri 3).
- t = 9: P2 finishes. Pick P3 (pri 4).
- t = 10: P3 finishes. Pick P4. t = 15: P4 finishes.

Gantt chart: P1(0–4) | P5(4–6) | P2(6–9) | P3(9–10) | P4(10–15)

Process	AT	BT	CT	TAT	WT
P1	0	4	4	4	0
P2	1	3	9	8	5
P3	2	1	10	8	7
P4	3	5	15	12	7
P5	4	2	6	2	0

Average TAT = (4+8+8+12+2)/5 = 6.8 ms

Average WT = (0+5+7+7+0)/5 = 3.8 ms

Numerical 5 — CPU Utilization

Q: In a 100 ms interval, the CPU is idle for 10 ms, busy on user processes for 80 ms, and busy on OS code for 10 ms. Calculate (a) total CPU utilization and (b) user CPU utilization.

Solution:

(a) CPU utilization (any busy time) = (100 − 10)/100 × 100% = 90%

(b) User CPU utilization = 80/100 × 100% = 80%

The 10 ms of OS time is overhead — system is busy but not doing user work.

Numerical 6 — Comparing All Algorithms

Q: Compare FCFS, SJF, SRTF, and RR (q = 2) for the following:

Process	AT	BT
P1	0	6
P2	1	4
P3	2	2

Solution:

FCFS: P1(0–6) | P2(6–10) | P3(10–12). WT: 0, 5, 8 → Avg = 4.33

SJF (NP): P1(0–6) | P3(6–8) | P2(8–12). WT: 0, 7, 4 → Avg = 3.67

SRTF: P1(0–1) | P2(1–2) | P3(2–4) | P2(4–7) | P1(7–12). WT: 6, 2, 0 → Avg = 2.67

RR (q=2): P1(0–2) | P2(2–4) | P3(4–6) | P1(6–8) | P2(8–10) | P1(10–12). WT: 6, 5, 2 → Avg = 4.33

Ranking: SRTF (2.67) < SJF (3.67) < FCFS = RR (4.33). Winner: SRTF — confirms SJF/SRTF give the minimum average WT.

Numerical 7 — Context Switch Overhead in RR

Q: RR with q = 10 ms. Each context switch takes 1 ms. 5 processes, each with BT = 20 ms. Calculate (a) total time and (b) overhead percentage.

Solution:

Each process needs 20 ms = 2 quantum slices. Total slices = 5 × 2 = 10. Switches between consecutive slices = 9.

(a) Total time = 10 × 10 ms (work) + 9 × 1 ms (switches) = 109 ms

(b) Overhead = 9/109 × 100% = 8.26%

7. Comparison Table

Algorithm	Preemptive?	Starvation?	Best For
FCFS	No	No	Batch systems
SJF	No	Yes (long jobs wait)	Batch with predictable bursts
SRTF	Yes	Yes	Batch; needs burst prediction
Priority	Both	Yes (low-priority starves)	Systems with clear priorities
Round Robin	Yes	No	Time-sharing, interactive
Multilevel Feedback	Yes	Controlled by aging	General-purpose OS

8. Quick Revision — One-Page Summary

Here is everything in Unit 4 compressed for last-minute revision.

- **CPU-I/O burst cycle** — process alternates between CPU bursts and I/O bursts.
- **Dispatcher** — performs the actual context switch. Dispatch latency = overhead.
- **Preemptive** — OS can take CPU back. **Non-preemptive** — process holds CPU until it releases.
- **Five criteria:** CPU utilization, throughput, TAT, WT, response time.

Core formulas:

- TAT = Completion Time − Arrival Time
- WT = TAT − Burst Time
- RT = First time on CPU − AT
- CPU utilization = (busy time / total time) × 100%

Six algorithms:

21. FCFS — arrival order; suffers convoy effect.
22. SJF (non-preemptive) — pick shortest next burst; optimal avg WT.
23. SRTF (preemptive SJF) — preempt if shorter job arrives.
24. Priority — pick highest priority; starvation fixed by aging.
25. Round Robin — time quantum q; ideal for interactive systems.
26. Multilevel feedback queue — processes move between priority queues based on behaviour.

- **Thread scheduling:** PCS (within process) vs SCS (system-wide).
- **Multi-processor:** SMP, processor affinity, push/pull migration.
- **Convoy effect** — short jobs stuck behind long one in FCFS.
- **Starvation fix:** Aging.

9. Practice Questions

Section A: Multiple Choice Questions (1 mark each)

Q1. Which scheduling algorithm is non-preemptive?

a) Round Robin
b) SRTF
c) FCFS
d) Preemptive Priority
Answer: (c)

Q2. In FCFS, the convoy effect refers to:

a) Short processes overtaking long ones
b) Short processes waiting behind a long process
c) Equal sharing of CPU
d) Priority inversion
Answer: (b)

Q3. Which algorithm gives the minimum average waiting time?

a) FCFS
b) SJF
c) Round Robin
d) Priority
Answer: (b) — SJF is provably optimal for average WT.

Q4. Turnaround time is calculated as:

a) BT − AT
b) CT − AT
c) WT + RT
d) AT − CT
Answer: (b)

Q5. Waiting time is calculated as:

a) TAT − BT
b) BT − TAT
c) CT − BT
d) AT + BT
Answer: (a)

Q6. Round Robin needs which essential parameter?

a) Priority
b) Burst time
c) Time quantum
d) Memory
Answer: (c)

Q7. If the time quantum in RR is very large, RR behaves like:

a) SJF
b) Priority
c) FCFS
d) SRTF
Answer: (c)

Q8. Starvation can occur in:

a) FCFS
b) Round Robin
c) Priority Scheduling
d) FIFO
Answer: (c)

Q9. Aging is used to solve:

a) Deadlock
b) Starvation
c) Convoy effect
d) Race condition
Answer: (b)

Q10. Which scheduler does the actual context switch?

a) Long-term scheduler
b) Short-term scheduler
c) Dispatcher
d) Loader
Answer: (c)

Q11. In preemptive SJF (SRTF), preemption happens when:

a) Time quantum expires
b) A process with shorter remaining burst arrives
c) A process completes
d) Priority changes
Answer: (b)

Q12. Multilevel feedback queue allows a process to:

a) Stay in one queue forever

b) Move between queues based on behaviour
c) Skip the OS
d) Run on multiple CPUs at once
Answer: (b)

Q13. In SMP:

a) Only one CPU schedules everything
b) Each CPU is self-scheduling
c) CPUs cannot communicate
d) Processes run only on CPU 0
Answer: (b)

Q14. Processor affinity means:

a) Killing a process
b) Keeping a process on the same CPU to use cache
c) Running a process on every CPU
d) Choosing process priority
Answer: (b)

Q15. Push migration:

a) Idle CPU pulls a process
b) Busy CPU sends process to idle CPU
c) Process forks itself
d) Process is killed
Answer: (b)

Q16. Three processes with BT = 5, 8, 12 (all arrive at t = 0). Average WT under SJF is:

a) 5 ms
b) 6 ms
c) 8.33 ms
d) 7 ms
Answer: (b) — Order: 5, 8, 12. WT = 0, 5, 13. Avg = 6 ms.

Q17. Response time matters most in:

a) Batch systems
b) Interactive systems
c) Sleep mode
d) Boot time
Answer: (b)

Q18. Which is the most general adaptive algorithm?

a) FCFS
b) SJF
c) Multilevel Feedback Queue
d) Round Robin
Answer: (c)

Q19. Convoy effect can be reduced by:

a) FCFS
b) Larger time quantum
c) SJF or SRTF
d) Priority queues only
Answer: (c)

Q20. Dispatch latency is:

a) Time from process arrival to first CPU use
b) Time taken by dispatcher to switch context
c) Total burst time
d) The quantum
Answer: (b)

Section B: Short Answer Questions (2 marks each)

Q1. Differentiate between preemptive and non-preemptive scheduling.

Ans: In non-preemptive scheduling, once a process gets the CPU it keeps it until it terminates or moves to waiting state. In preemptive scheduling, the OS can forcibly take the CPU away from a running process — usually when a higher-priority process arrives or a time slice expires.

Q2. List the five scheduling criteria.

Ans: CPU utilization (maximize), throughput (maximize), turnaround time (minimize), waiting time (minimize), response time (minimize).

Q3. Define turnaround time and waiting time.

Ans: Turnaround time (TAT) = Completion Time − Arrival Time. Waiting time (WT) = TAT − Burst Time, the total time a process spent in the ready queue.

Q4. What is the convoy effect in FCFS?

Ans: The convoy effect occurs when a long CPU-bound process holds the CPU and forces all shorter processes to wait behind it, dramatically increasing average waiting time.

Q5. Why is SJF considered optimal?

Ans: SJF gives the minimum possible average waiting time among all scheduling algorithms. Giving the CPU to the shortest job first removes that job from the queue quickly, reducing the wait for all remaining jobs.

Q6. What is the main drawback of priority scheduling and how is it solved?

Ans: The main drawback is starvation — low-priority processes may wait indefinitely. The solution is aging: gradually increasing the priority of a waiting process so it eventually gets scheduled.

Q7. What is the time quantum in Round Robin? What if it is too large or too small?

Ans: Time quantum is the small unit of CPU time given to each process before preemption. If it is too large, RR behaves like FCFS. If too small, the system spends most of its time on context switches.

Q8. Define context switch and dispatch latency.

Ans: A context switch is saving the state of the running process to its PCB and loading the state of the next process. Dispatch latency is the time the dispatcher takes to do this.

Q9. Differentiate between PCS and SCS for thread scheduling.

Ans: PCS (Process-Contention Scope) is for user threads — they compete only with other threads of the same process. SCS (System-Contention Scope) is for kernel threads — they compete with all other kernel threads system-wide.

Q10. What is processor affinity?

Ans: Processor affinity is the practice of keeping a process on the same CPU it ran on, so its data stays warm in that CPU's cache. Soft affinity is a preference; hard affinity is a hard constraint.

Q11. Differentiate between push migration and pull migration.

Ans: Push migration: a separate process pushes processes from overloaded CPUs to underloaded ones. Pull migration: an idle CPU pulls a process from an overloaded CPU on its own.

Q12. Define throughput.

Ans: Throughput is the number of processes completed per unit time. Higher throughput means the scheduler is doing more useful work per second.

Section C: Long Answer Questions (5 marks each)

Q1. Explain the CPU-I/O burst cycle and the role of the CPU scheduler and dispatcher.

Approach: *Define the burst cycle. Show how processes alternate between CPU and I/O bursts. Distinguish CPU-bound and I/O-bound. Explain that the scheduler picks the next process and the dispatcher actually performs the context switch. Mention dispatch latency. Include Figure 4.1.*

Q2. Explain the FCFS scheduling algorithm with an example. Discuss its advantages and the convoy effect.

Approach: *Define FCFS as non-preemptive FIFO. Take an example with 3 processes (one long, two short). Draw the Gantt chart and compute average WT. Show how reordering changes WT drastically. Explain the convoy effect with a real-world analogy. Include Figure 4.2.*

Q3. Compare SJF (non-preemptive) and SRTF (preemptive SJF) with a worked example.

Approach: *Define both. Use the same dataset. Trace SJF — once a process starts it runs to completion. Then trace SRTF — preempt whenever a shorter job arrives. Compute average WT for both. Show that SRTF gives lower WT when arrival times differ. Include Figures 4.3 and 4.4.*

Q4. Discuss priority scheduling. Explain the starvation problem and the concept of aging.

Approach: *Define priority scheduling. Explain internal/external priority assignment. Distinguish preemptive and non-preemptive variants. Describe starvation with an example. Explain aging — gradually increasing the priority of a process the longer it waits.*

Q5. Explain Round Robin scheduling. Discuss the effect of time quantum size with an example.

Approach: *Define RR. Explain that each process gets a time quantum, and if it does not finish, it goes to the back of the queue. Use the 4-process example with q = 4 ms. Compute average WT. Discuss what happens with very large q (becomes FCFS) and very small q (too much overhead). Include Figure 4.5.*

Q6. Compare multilevel queue scheduling and multilevel feedback queue scheduling.

Approach: *Describe multilevel queue with separate fixed queues for different process types. Mention its rigidity. Then describe multilevel feedback queue where processes can move between queues. Conclude that multilevel feedback queue is the most general algorithm. Include Figure 4.6.*

Q7. Explain multi-processor scheduling. Discuss SMP, processor affinity, and load balancing.

Approach: *Define asymmetric vs symmetric multiprocessing. Explain SMP — each CPU runs its own scheduler. Explain processor affinity (soft vs hard) and why cache warmth matters. Explain push and pull migration. Mention the trade-off between affinity and load balancing.*

Q8. Solve: Given (P1: AT=0, BT=5), (P2: AT=1, BT=3), (P3: AT=2, BT=8), (P4: AT=3, BT=6), compute average WT under FCFS, SJF, and RR (q=2).

Approach: *Trace each algorithm step by step. For FCFS, execution order = arrival order. For SJF non-preemptive, at each scheduling decision pick the shortest available burst. For RR, carefully track the queue. Make a comparison table at the end.*

Q9. Compare FCFS, SJF, SRTF, Priority, and Round Robin in terms of preemption, starvation, complexity, and best use case.

Approach: *Make a comparison table covering: (a) preemptive or not, (b) starvation possible, (c) complexity, (d) typical use case. Conclude with which scheduler is used in modern OSes (Linux uses CFS — a refined multilevel feedback queue style).*

UNIT 5

Memory Management

Contents

1. Overview of Memory Management

Why Memory Management Matters

The CPU can only execute instructions that are in main memory (RAM). But RAM is limited, expensive, and shared among many processes. The memory manager is the part of the OS that decides which processes get memory and how much, where in memory each process goes, when to move processes in and out of memory, and how to keep one process from overwriting another's data.

> **Real-world analogy —** Think of RAM as a big parking lot and processes as cars of different sizes. The parking attendant (memory manager) decides who parks where, manages empty slots, and makes sure two cars don't claim the same spot.

Basic Hardware — Base and Limit Registers

To protect processes from each other, the CPU has two special registers:

- **Base register —** holds the smallest legal physical address for the currently running process.
- **Limit register —** holds the size of the range.

Process memory is [base, base + limit − 1]. Every memory access by a user program is checked by hardware. If an address is outside this range, the CPU traps it as an illegal access. These registers can be loaded only by privileged instructions, so only the kernel can change them.

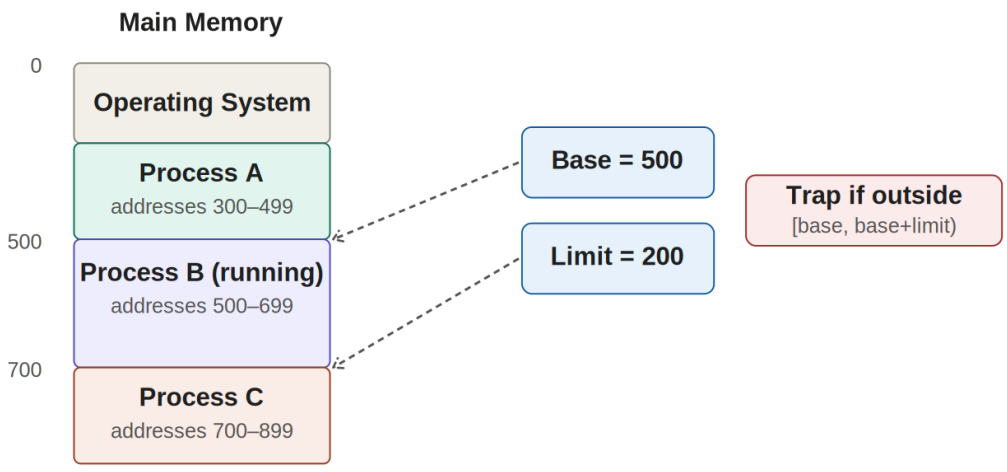


Figure 5.1 — Memory protection using base and limit registers.

Logical vs Physical Addresses

This distinction is central to the entire unit — pay close attention.

- **Logical address (virtual address) —** the address generated by the CPU while a program executes. The program "thinks" its data lives at this address.
- **Physical address —** the actual address that appears on the memory bus and reaches the RAM chip.

The user program never sees physical addresses. It only works with logical ones. The translation from logical to physical happens transparently in hardware, by a unit called the Memory Management Unit (MMU). The set of all logical addresses a program can generate is its logical address space; the corresponding physical addresses form the physical address space.

Address Binding

A program on disk has symbolic references (like variable names). Before it can run, these must be translated to actual memory addresses. This translation, called address binding, can happen at three times:

- **Compile time —** if you know in advance where the program will load, the compiler generates absolute addresses directly. If that location changes later, the program must be recompiled. (Old MS-DOS .COM programs worked this way.)
- **Load time —** the compiler generates relocatable code; the loader fixes addresses when loading the program into memory.
- **Execution time —** binding is delayed until runtime. The program can move around in memory even while running. Used by almost all modern OSes — and requires hardware support (the MMU).

The Memory Management Unit (MMU)

The MMU performs logical-to-physical translation at runtime. In its simplest form (with just a relocation register): every logical address has the relocation value added to it to produce the physical address.

Example: relocation register = 14000. Program references logical address 346. Physical address = 14000 + 346 = 14346.

2. Swapping

Sometimes we have more processes than RAM can hold at once. Swapping is the OS technique of temporarily moving a process out of memory to a backing store (typically a disk partition called the swap space) and bringing it back later when needed.

The Basic Idea

- **Swap out —** take a process in main memory that isn't currently running, copy its entire memory image to disk, and free up the RAM it occupied.
- **Swap in —** copy the process's memory image back from disk into RAM.

The medium-term scheduler (from Unit 2) manages swapping decisions. Swapping lets the OS support more processes than can fit in RAM at once, at the cost of slower context switches for swapped-out processes.

Example: A system has 128 MB of RAM. Eight processes of 20 MB each need to run. Without swapping, only 6 processes can fit (6 × 20 = 120 MB). With swapping, all 8 can exist in the system — at any instant only a subset is in RAM while others wait on disk.

Cost of Swapping

Swapping is expensive — disk I/O is much slower than RAM. Consider a 100 MB process and a disk transfer rate of 50 MB/s:

- Time to swap out = 100 / 50 = 2 seconds
- Time to swap in = another 2 seconds
- Total cost = 4 seconds just to switch to this process

Modern systems therefore swap only parts of processes (through paging, which we'll see later) rather than whole processes. Traditional full-process swapping is used rarely today — typically only when memory pressure gets severe.

3. Memory Allocation — Contiguous Allocation

In contiguous memory allocation, each process is assigned a single contiguous block of memory. RAM is divided into two regions: one for the OS and the rest for user processes.

Fixed Partitioning (Multiple Partitions)

The user area is divided into fixed-size partitions at system boot. Each partition holds exactly one process. The OS maintains a table showing which partitions are free and which are in use.

- **Pros:** Simple to implement.
- **Cons:** Internal fragmentation — if a process needs 7 KB but is assigned a 10 KB partition, 3 KB is wasted inside the partition.

Variable (Dynamic) Partitioning

Partitions are created on demand. When a process arrives, the OS finds a large enough hole and carves out a partition of exactly the process's size. The OS tracks all holes using a free list. When a process terminates, its memory becomes a new hole, possibly merging with adjacent holes.

The Three Classic Allocation Strategies

When a process of size N needs memory and there are several free holes, which hole should we pick?

- **First-fit** — allocate the first hole that is big enough. Scan from the top of the list and pick the first one that fits. Fast.
- **Best-fit** — allocate the smallest hole that is big enough. Must search the entire list. Leaves the smallest leftover hole.
- **Worst-fit** — allocate the largest hole. Must search entire list. The leftover hole is large, possibly useful for another allocation.

Studies have shown that first-fit and best-fit generally perform better than worst-fit in both storage utilization and speed. First-fit is faster; best-fit gives slightly better memory utilization.

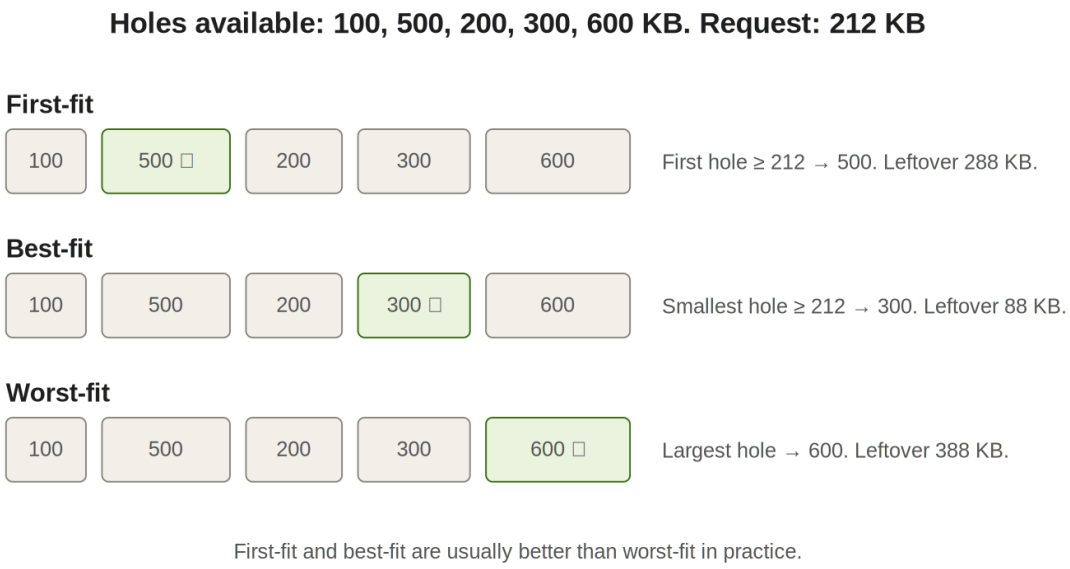


Figure 5.2 — First-fit, Best-fit, and Worst-fit strategies compared.

Example of the Three Strategies

Holes in memory (in order): 100 KB, 500 KB, 200 KB, 300 KB, 600 KB.

Process request: 212 KB.

- **First-fit:** Scans 100 (too small), 500 (fits!) → allocate from 500 KB. Leftover = 288 KB.
- **Best-fit:** Smallest hole ≥ 212 KB is 300 KB → allocate from 300 KB. Leftover = 88 KB.
- **Worst-fit:** Largest hole is 600 KB → allocate from 600 KB. Leftover = 388 KB.

Fragmentation

Memory allocation suffers from two types of waste:

- **Internal fragmentation** — memory allocated to a process but not used by it. Happens with fixed partitions when a process is smaller than its partition.
- **External fragmentation** — enough free memory exists in total to satisfy a request, but it's scattered in small pieces; no single hole is big enough.

Example: three free holes of 30 KB, 50 KB, and 40 KB (total 120 KB), and a process needs 100 KB — but no single hole is big enough.

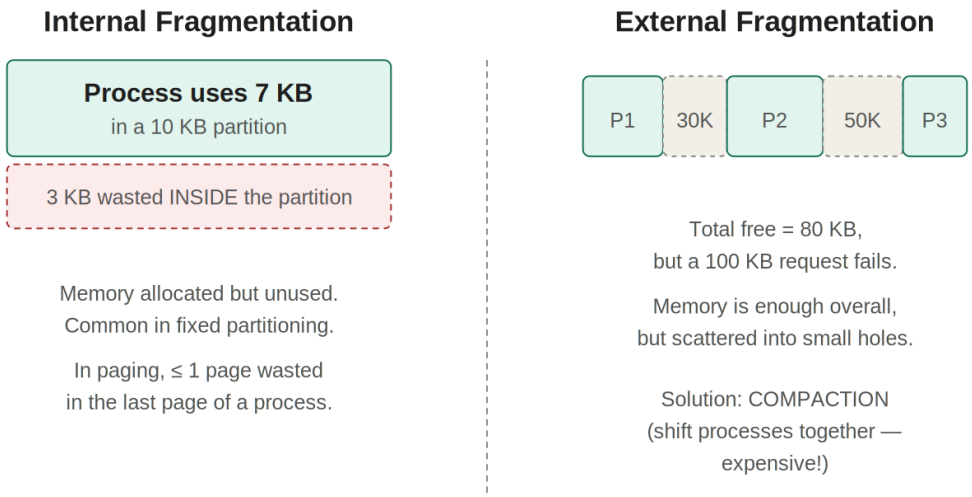


Figure 5.3 — Internal vs External fragmentation.

Solution — Compaction. Shift all processes toward one end of memory, pushing all free holes into one big hole at the other end. Compaction is expensive (must move entire processes in memory, requires relocation support), so modern systems avoid it by using paging instead.

4. Segmentation

The Idea

Users think of programs as a collection of logical segments — not as a flat byte array. A typical program has:

- A main code segment
- Segments for each function or module
- A stack segment
- A data segment (globals, heap)
- A segment for each object or table

Segmentation is a memory management scheme that respects this user view. Each segment can have a different size and lives somewhere in physical memory.

Address Translation

A logical address in segmentation is a pair: (segment number, offset) or <s, d>.

The OS maintains a segment table for each process. Each entry has:

- **Base** — starting physical address of the segment.
- **Limit** — length of the segment.

To translate <s, d> to a physical address:

27. Look up entry s in the segment table.
28. Check that d < limit — if not, this is an illegal address (trap).
29. Physical address = base + d.

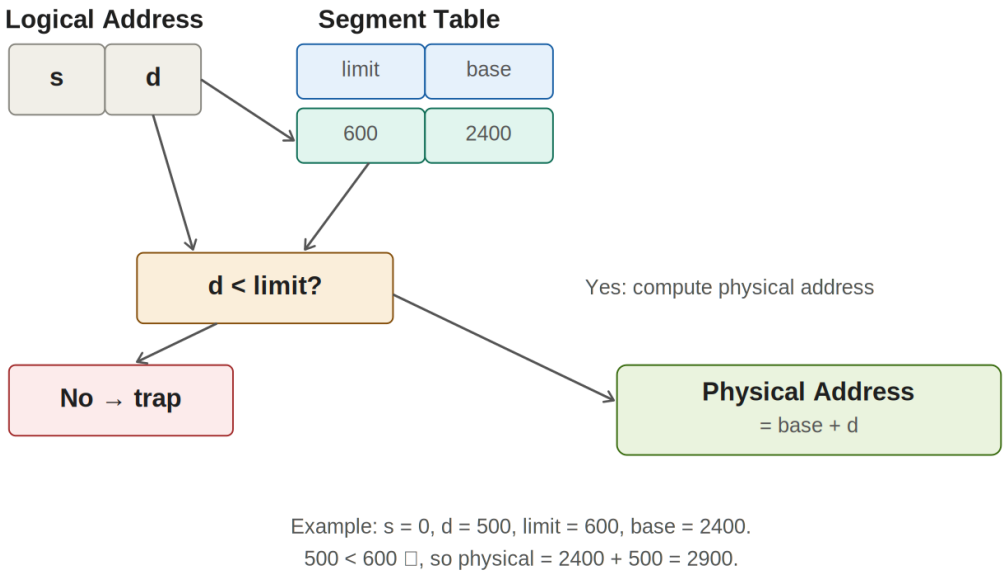


Figure 5.4 — Segmentation address translation.

Worked Example

Process segment table:

Segment	Base	Limit
0 (code)	1400	1000
1 (data)	6300	400
2 (stack)	4300	1100
3 (heap)	3200	1000

Logical address <2, 500>: segment 2 has base 4300, limit 1100. Offset 500 < 1100 ✓. Physical address = 4300 + 500 = 4800.

Logical address <1, 500>: segment 1 has limit 400. Offset 500 > 400 ✗ → addressing error (trap).

Pros and Cons of Segmentation

- **Pros:** Matches user's view of the program. Protection is natural — each segment can have its own access rights (read, write, execute). Sharing is easy — two processes can share a segment by having entries with the same base.
- **Cons:** External fragmentation — segments have varying sizes, so finding a hole big enough becomes the same problem as in dynamic partitioning. Memory allocation is complex.

5. Paging

Paging solves the external fragmentation problem by allowing a process's memory to be non-contiguous in physical memory. This is the most important memory management scheme in modern OSes.

The Core Idea

- Physical memory is divided into fixed-size blocks called frames.
- Logical memory is divided into blocks of the same size called pages.
- Any page can be mapped to any frame.

When a process of size N pages is to be executed, its pages are loaded into any N available frames — they don't have to be contiguous.

Typical page size: 4 KB (4096 bytes). It is always a power of 2 so that address splitting becomes easy (just use bits).

Address Translation in Paging

Every logical address generated by the CPU is split into two parts:

- **Page number (p)** — the upper bits of the address. Used as an index into the page table.
- **Page offset (d)** — the lower bits. Used as the offset within the page.

The OS maintains a page table for each process. Entry p in the page table gives the frame number f where page p is stored in physical memory.

> **Formula —** Physical address = (frame number × page size) + offset.

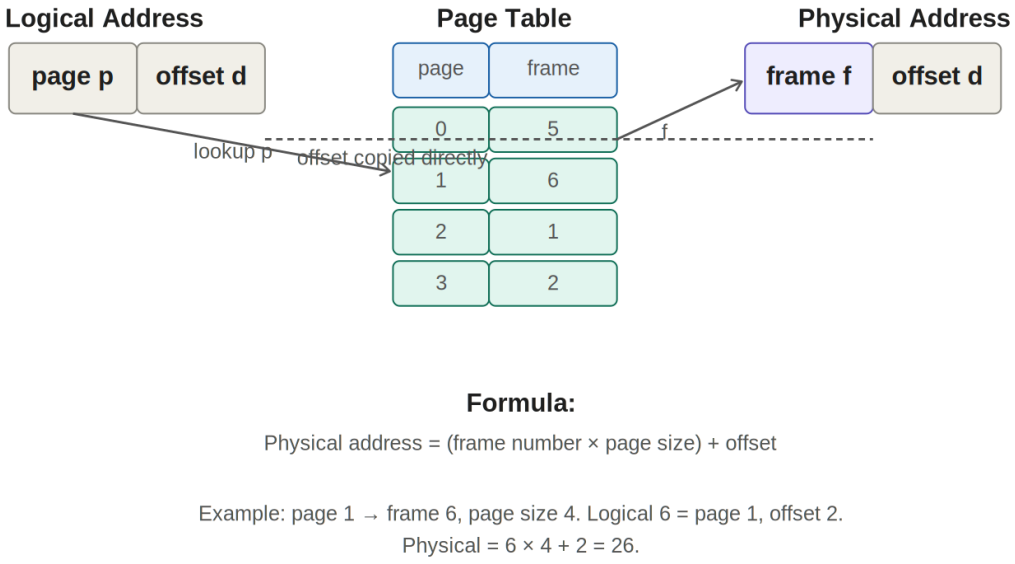


Figure 5.5 — Paging: pages map to frames through the page table.

Worked Example

Suppose page size = 4 bytes, physical memory = 32 bytes (8 frames), logical memory = 16 bytes (4 pages).

Page table for a process:

Page	Frame
0	5
1	6
2	1
3	2

Convert logical address 6 (binary 0110) to physical:

- Split: page number = 01 (binary) = 1, offset = 10 (binary) = 2.
- Page 1 → frame 6.
- Physical address = (6 × 4) + 2 = 26.

Convert logical address 11 (binary 1011):

- Page = 10 = 2, offset = 11 = 3.
- Page 2 → frame 1.
- Physical address = (1 × 4) + 3 = 7.

Why Paging Eliminates External Fragmentation

Since every frame is exactly the same size, any free frame can hold any page. No holes are wasted. The only fragmentation left is internal fragmentation — and only in the last page of each process. On average, half a page is wasted per process, which is negligible compared to the loss under variable partitioning.

Hardware Support — TLB

Every memory reference now requires two memory accesses: one to read the page table, one to read the actual data. This would halve CPU speed. The fix is a small, fast hardware cache called the Translation Look-aside Buffer (TLB).

The TLB stores recently used page-to-frame mappings. When a page number arrives:

- **TLB hit —** mapping is in the TLB → frame number is known immediately. No extra memory access.
- **TLB miss —** mapping not in TLB → go to the full page table in memory (extra memory access), then add this entry to the TLB for next time.

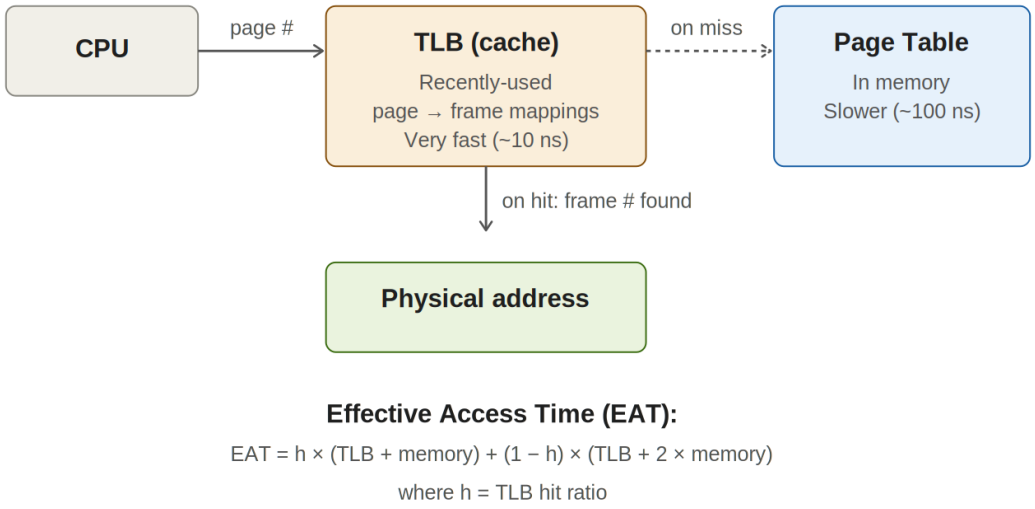


Figure 5.6 — Address translation with TLB. Hit ratio determines effective access time.

EAT formula (extremely common in exams) — EAT = h × (TLB_time + memory_time) + (1 − h) × (TLB_time + 2 × memory_time), where h = hit ratio.

6. Structure of the Page Table

For small processes, a single flat page table works. For large address spaces (like 32-bit or 64-bit systems), page tables themselves become huge. Several structures are used to handle this:

Hierarchical (Multi-Level) Page Tables

Split the page table into several levels. Only the parts actually used are kept in memory.

A common scheme is two-level paging (used in 32-bit x86). A 32-bit address is split into: outer page number | inner page number | offset. The outer page number indexes into a page directory that points to a second-level page table. The inner page number indexes into that second-level table to find the frame.

Benefit: if a process uses only a small portion of its address space (very common), only the relevant second-level tables need to exist. The rest are null pointers. Modern 64-bit systems use three- or four-level hierarchical page tables.

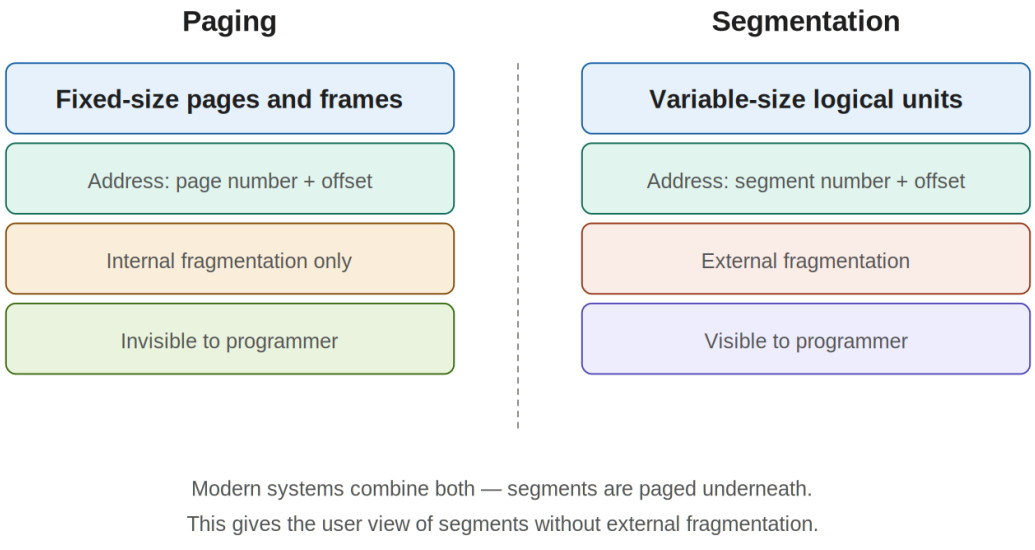


Figure 5.7 — Two-level (hierarchical) page table.

Hashed Page Tables

Common for address spaces larger than 32 bits. The virtual page number is hashed into a table. Each entry is a linked list of pages that hash to the same slot. To translate: hash the page number, search the linked list for a match, retrieve the frame number.

Inverted Page Tables

Instead of one page table per process, the system keeps a single inverted page table — one entry per physical frame. Each entry holds <process_id, page_number>. To translate a virtual address for process P: search the inverted table for entry matching <P, page_number>. If found, its index is the frame number.

Benefit: saves memory dramatically (one table, total size proportional to physical memory, not virtual). Drawback: searching the table is slower than direct indexing — solved with hashing.

7. Numerical Questions with Solutions

Numerical 1 — Paging Basics

Q: A system has a logical address space of 16 pages, each of size 1024 bytes, mapped to 32 frames of physical memory. How many bits are used for (a) the page number, (b) the offset, (c) the frame number, (d) the physical address?

Solution:

(a) Number of pages = 16 = 2^4 → page number uses 4 bits.

(b) Page size = 1024 bytes = 2^{10} → offset uses 10 bits.

(c) Number of frames = 32 = 2^5 → frame number uses 5 bits.

(d) Physical address = frame bits + offset bits = 5 + 10 = 15 bits.

Logical address = 4 + 10 = 14 bits (2^{14} = 16 KB of logical memory). Physical = 15 bits (2^{15} = 32 KB).

Numerical 2 — Address Translation

Q: Page size = 2 KB. A process has 4 pages. The page table is shown below. Convert logical addresses 4500, 8200, and 1000 to physical addresses.

Page	Frame
0	8
1	3
2	5
3	7

Solution: Page size = 2 KB = 2048 bytes.

Address 4500:

- Page number = 4500 / 2048 = 2 (integer division).
- Offset = 4500 mod 2048 = 4500 − 4096 = 404.
- Page 2 maps to frame 5.
- Physical address = (5 × 2048) + 404 = 10240 + 404 = 10644.

Address 8200:

- Page number = 8200 / 2048 = 4.
- But the process only has pages 0–3. Invalid reference — trap.

Address 1000:

- Page number = 1000 / 2048 = 0.
- Offset = 1000 mod 2048 = 1000.
- Page 0 maps to frame 8.
- Physical address = (8 × 2048) + 1000 = 16384 + 1000 = 17384.

Numerical 3 — Effective Access Time with TLB

Q: A paging system uses a TLB with access time 20 ns and main memory with access time 100 ns. The TLB hit ratio is 80%. Calculate the effective access time (EAT).

Solution: Using the EAT formula:

EAT = h × (TLB + memory) + (1 − h) × (TLB + 2 × memory)

- Hit term: 0.80 × (20 + 100) = 0.80 × 120 = 96 ns
- Miss term: 0.20 × (20 + 2 × 100) = 0.20 × 220 = 44 ns

EAT = 96 + 44 = 140 ns

Without paging, each access would take 100 ns. With paging and no TLB, it would take 200 ns. With an 80% TLB hit, we get 140 ns — a 30% slowdown compared to raw memory, but much better than 100% slowdown without the TLB.

Numerical 4 — TLB Hit Ratio Required

Q: Memory access time = 150 ns. TLB access time = 15 ns. What minimum TLB hit ratio is needed to keep the effective access time within 20% of the raw memory access time?

Solution:

20% of raw memory = 30 ns. Allowed EAT ≤ 150 + 30 = 180 ns.

Let h = hit ratio.

```
EAT = h × (15 + 150) + (1 − h) × (15 + 300)
    = 165h + (1 − h) × 315
    = 165h + 315 − 315h
    = 315 − 150h
```

Set 315 − 150h = 180:

150h = 135 → h = 0.9 (90%)

Answer: TLB hit ratio must be at least 90%.

Numerical 5 — Page Table Size

Q: A system has a 32-bit logical address and a page size of 4 KB. Each page table entry takes 4 bytes. Calculate the size of the page table.

Solution:

- Page size = 4 KB = 2^{12} bytes → offset = 12 bits.
- Logical address = 32 bits → page number = 32 − 12 = 20 bits.
- Number of pages = 2^{20} = 1,048,576.
- Each entry = 4 bytes.
- Page table size = 1,048,576 × 4 = 4,194,304 bytes = 4 MB.

Why hierarchical paging matters: A 4 MB contiguous page table for every process is huge. With two-level paging, only the parts of the table actually used need to be in memory.

Numerical 6 — Allocation Strategies

Q: Free holes (in order): 100 KB, 500 KB, 200 KB, 300 KB, 600 KB. Process requests in sequence: 212 KB, 417 KB, 112 KB, 426 KB. Show allocation under (a) first-fit, (b) best-fit, (c) worst-fit.

Solution:

First-fit (pick first hole big enough):

Request	Hole chosen	Holes after
212 KB	500 KB	100, 288, 200, 300, 600
417 KB	600 KB	100, 288, 200, 300, 183
112 KB	288 KB	100, 176, 200, 300, 183
426 KB	None ≥ 426 KB	Rejected

Best-fit (smallest hole that fits):

Request	Hole chosen	Holes after
212 KB	300 KB	100, 500, 200, 88, 600
417 KB	500 KB	100, 83, 200, 88, 600
112 KB	200 KB	100, 83, 88, 88, 600
426 KB	600 KB	100, 83, 88, 88, 174

All four processes fit.

Worst-fit (largest hole):

Request	Hole chosen	Holes after
212 KB	600 KB	100, 500, 200, 300, 388
417 KB	500 KB	100, 83, 200, 300, 388
112 KB	388 KB	100, 83, 200, 300, 276
426 KB	None ≥ 426 KB	Rejected

Conclusion: Best-fit wins — all four processes fit. Worst-fit's habit of breaking up large holes into medium holes backfires when a big request arrives later.

Numerical 7 — Internal Fragmentation

Q: Process size = 73 KB. Page size = 4 KB. How many pages does the process need? How much internal fragmentation occurs?

Solution:

- Number of pages needed = ceil(73 / 4) = ceil(18.25) = 19 pages.
- Memory allocated = 19 × 4 = 76 KB.
- Internal fragmentation = 76 − 73 = 3 KB (wasted in the last page).

General rule: Internal fragmentation ≤ (page size − 1) bytes per process. On average, half a page is wasted per process.

Numerical 8 — Segmentation Address Translation

Q: Given the segment table for a process, find the physical address for (a) <0, 350>, (b) <2, 150>, (c) <3, 900>.

Segment	Base	Limit
0	1200	400
1	2000	600
2	5000	100
3	8000	1000

Solution:

(a) <0, 350>: segment 0, limit 400. Offset 350 < 400 ✓. Physical = 1200 + 350 = 1550.

(b) <2, 150>: segment 2, limit 100. Offset 150 > 100 ✗. Addressing error (trap).

(c) <3, 900>: segment 3, limit 1000. Offset 900 < 1000 ✓. Physical = 8000 + 900 = 8900.

Numerical 9 — Multi-level Paging

Q: A 32-bit logical address uses two-level paging with 10 bits for outer page, 10 bits for inner page, 12 bits for offset. What is the page size? How many outer page table entries? How many inner page table entries per table?

Solution:

- Offset = 12 bits → page size = 2^{12} = 4096 bytes = 4 KB.
- Outer page number = 10 bits → 1024 outer page table entries.
- Inner page number = 10 bits → 1024 entries per second-level table.

Total potential pages = 1024 × 1024 = 2^{20} = 1M pages. Each page = 4 KB → total logical space = 4 GB ✓ (matches the 2^{32} address space).

Numerical 10 — EAT Without TLB

Q: A system without a TLB uses paging. Memory access time = 100 ns. What is the effective access time for a memory reference?

Solution:

Without TLB, every access needs two memory lookups:

- One to read the page table entry (100 ns).
- One to fetch the actual data (100 ns).

EAT = 100 + 100 = 200 ns.

This is a 100% overhead — exactly why modern systems cannot function without a TLB. A good TLB with ~98% hit rate brings EAT back close to 100 ns.

8. Comparison — Paging vs Segmentation

Aspect	Paging	Segmentation
Unit size	Fixed (page = frame)	Variable (one per logical unit)
Divided by	System (invisible to user)	User/compiler (visible)
Fragmentation	Internal only	External
Address	Page number + offset	Segment number + offset
Translation	Page table	Segment table
Sharing	Possible at page level	Easy at segment level
Protection	Per page	Per segment (more natural)
Compaction needed	No	Yes (sometimes)

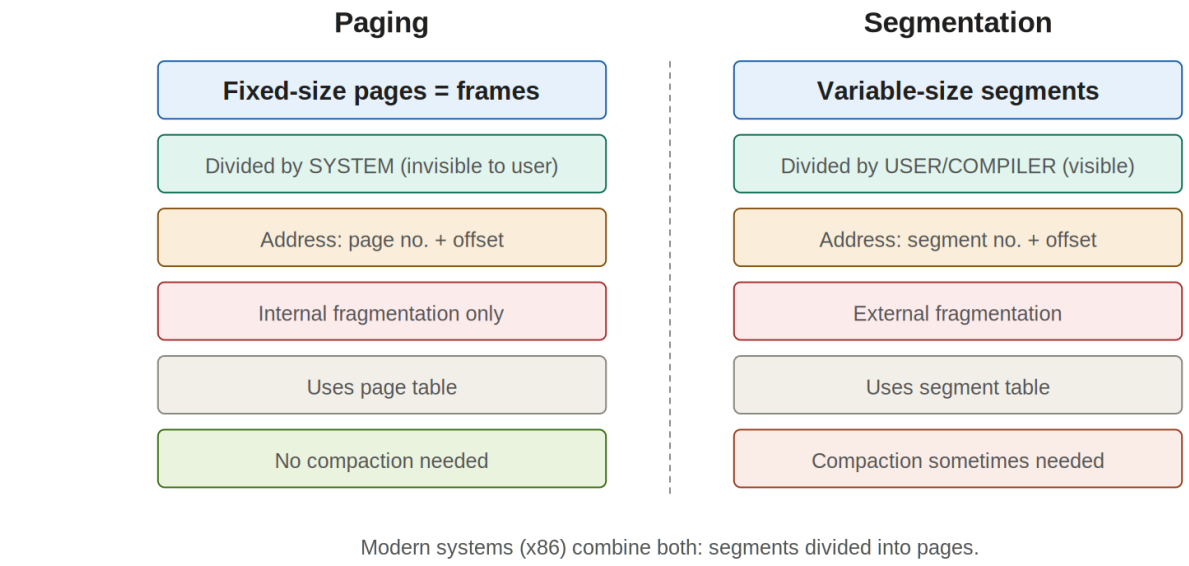


Figure 5.8 — Paging vs Segmentation at a glance.

Combined approach — Segmentation with paging: Modern systems like Intel x86 support both. Logical memory is divided into segments, and each segment is divided into pages. This gives the user view of segments but eliminates external fragmentation through paging underneath.

9. Quick Revision — One-Page Summary

Here is everything in Unit 5 compressed for last-minute revision.

- **Memory manager:** decides who gets memory, where, and when to swap.
- **Base + limit registers:** hardware protection, checked on every user-mode access.
- **Logical address** = what program generates. **Physical address** = what reaches RAM. **MMU** translates between them.
- **Address binding:** compile-time (absolute), load-time (relocatable), execution-time (runtime, needs MMU).
- **Swapping** — move a whole process from RAM to disk and back. Expensive; used sparingly today.
- **Contiguous allocation** — each process gets one block. Suffers external fragmentation.
- **Three allocation strategies:** First-fit (fast), Best-fit (lowest waste), Worst-fit (biggest leftover). First-fit is generally preferred.
- **Internal fragmentation** — wasted space inside a partition. **External fragmentation** — total free is enough but scattered. **Compaction** fixes external at high cost.
- **Segmentation** — logical divisions (code, data, stack). Address = <s, d>. Uses segment table with base + limit per segment.
- **Paging** — split memory into fixed-size pages and frames. Any page → any frame. Address = page # + offset. Uses page table. Eliminates external fragmentation.
- **TLB** — hardware cache for page table entries. Vital for performance.
- **Page table structures:** flat, hierarchical (multi-level), hashed, inverted.

Key formulas:

- Page number bits = $\log_2$(number of pages)
- Offset bits = $\log_2$(page size in bytes)
- Page table size = number of pages × size of each entry
- Physical address (paging) = frame × page_size + offset
- Physical address (segmentation) = base + offset
- EAT = h × (TLB + memory) + (1 − h) × (TLB + 2 × memory)
- Internal fragmentation per process ≤ page size − 1

10. Practice Questions

Section A: Multiple Choice Questions (1 mark each)

Q1. The address generated by the CPU is called:

a) Physical address
b) Logical address
c) Frame address
d) Page address
Answer: (b)

Q2. The hardware that translates logical to physical addresses is:

a) ALU
b) MMU
c) DMA
d) PCB
Answer: (b) — Memory Management Unit.

Q3. Address binding done at compile time produces:

a) Relocatable code
b) Absolute code
c) Position-independent code
d) Hashed code
Answer: (b)

Q4. Base and limit registers are used for:

a) Caching
b) Memory protection
c) Swapping
d) File access
Answer: (b)

Q5. Swapping moves processes between:

a) Two RAM areas
b) Main memory and disk
c) Two disks
d) CPU and RAM
Answer: (b)

Q6. In contiguous memory allocation, internal fragmentation is caused by:

a) Too many processes
b) Process being smaller than its partition
c) Holes between partitions
d) Lack of compaction
Answer: (b)

Q7. Which allocation strategy picks the smallest hole that is big enough?

a) First-fit
b) Best-fit
c) Worst-fit
d) Random-fit
Answer: (b)

Q8. External fragmentation is solved by:

a) Larger RAM
b) Compaction or paging
c) FCFS scheduling
d) Disabling swapping
Answer: (b)

Q9. In paging, logical memory is divided into:

a) Segments
b) Pages
c) Frames
d) Holes
Answer: (b)

Q10. In paging, physical memory is divided into:

a) Segments
b) Pages
c) Frames
d) Sectors
Answer: (c)

Q11. If page size is 1 KB, the offset uses:

a) 10 bits
b) 12 bits
c) 16 bits
d) 8 bits
Answer: (a) — 1 KB = 2^{10}.

Q12. The TLB is:

a) A page replacement algorithm
b) A cache of recent page-to-frame mappings
c) A type of segment table
d) Disk storage
Answer: (b)

Q13. If TLB hit ratio = 100%, EAT equals:

a) 2 × memory time
b) Memory time + TLB time
c) Memory time only
d) Zero
Answer: (b)

Q14. In segmentation, a logical address consists of:

a) Page number and offset
b) Segment number and offset
c) Frame number and offset
d) Base and limit
Answer: (b)

Q15. Segmentation suffers from:

a) Internal fragmentation only
b) External fragmentation
c) No fragmentation
d) Cache misses
Answer: (b)

Q16. Paging eliminates:

a) Internal fragmentation
b) External fragmentation
c) Both
d) Neither
Answer: (b)

Q17. Two-level paging is used to:

a) Make pages bigger
b) Reduce the size of the page table in memory
c) Speed up TLB
d) Avoid swapping
Answer: (b)

Q18. In an inverted page table, each entry corresponds to:

a) A virtual page
b) A physical frame
c) A segment
d) A process
Answer: (b)

Q19. The maximum internal fragmentation per process in paging is:

a) 1 byte
b) Half page
c) Page size − 1
d) Page size
Answer: (c)

Q20. Process size = 9 KB, page size = 4 KB. Pages needed and internal fragmentation:

a) 2 pages, 1 KB
b) 3 pages, 3 KB
c) 3 pages, 1 KB
d) 2 pages, 3 KB
Answer: (b) — ceil(9/4) = 3 pages = 12 KB allocated. 12 − 9 = 3 KB.

Section B: Short Answer Questions (2 marks each)

Q1. Differentiate between logical address and physical address.

Ans: Logical address (also called virtual address) is the address generated by the CPU during execution; it is what the program sees. Physical address is the actual address in main memory where the data is stored. The MMU translates logical to physical at runtime.

Q2. What is the role of the MMU?

Ans: The Memory Management Unit (MMU) is hardware that translates logical addresses into physical addresses at runtime. In simple schemes it does this by adding a relocation value; in paging it uses the page table; in segmentation it uses the segment table.

Q3. Differentiate between internal and external fragmentation.

Ans: Internal fragmentation is wasted space inside an allocated partition (process is smaller than the partition). External fragmentation is wasted space outside any partition — total free memory is enough, but it's scattered into small holes that no single process can use.

Q4. What is compaction? Why is it expensive?

Ans: Compaction shifts all allocated memory toward one end of RAM, merging all free holes into a single large hole. It is expensive because every process being moved must have its memory contents physically copied, requires relocation support in hardware, and stalls the system during the move.

Q5. Compare first-fit, best-fit, and worst-fit allocation strategies.

Ans: First-fit picks the first hole big enough — fast. Best-fit picks the smallest hole big enough — leaves the smallest leftover, but searches the entire list. Worst-fit picks the largest hole — leaves big leftovers but breaks up large holes. First-fit and best-fit generally outperform worst-fit.

Q6. Define paging.

Ans: Paging is a memory management scheme in which physical memory is divided into fixed-size blocks called frames and logical memory is divided into blocks of the same size called pages. Any page can be mapped to any frame, eliminating external fragmentation.

Q7. What is a page table? What is its purpose?

Ans: A page table is a per-process data structure maintained by the OS. It maps each logical page number to the physical frame number where that page currently resides. The MMU uses the page table during every memory access to translate logical addresses to physical.

Q8. What is a TLB and why is it needed?

Ans: Translation Lookaside Buffer (TLB) is a small, fast hardware cache that stores recently used page-to-frame mappings. Without it, every memory access would need two trips to memory (one for the page table, one for the data). The TLB lets most accesses skip the page table lookup, making paging fast enough for practical use.

Q9. State the EAT formula for paging with TLB.

Ans: EAT = h × (TLB time + memory time) + (1 − h) × (TLB time + 2 × memory time), where h is the TLB hit ratio.

Q10. Define segmentation.

Ans: Segmentation is a memory management scheme that divides a program into logical units of varying sizes called segments — typically code, data, stack, heap. Each segment lives somewhere in physical memory; the OS maintains a segment table with the base and limit of each segment.

Q11. Differentiate between paging and segmentation.

Ans: Paging uses fixed-size blocks (pages and frames) determined by the system; segmentation uses variable-size segments determined by the user/compiler. Paging causes only internal fragmentation (≤ 1 page); segmentation causes external fragmentation. Paging hides the structure from the user; segmentation respects the user's logical view of the program.

Q12. What is hierarchical paging?

Ans: Hierarchical paging splits the page table into multiple levels. Instead of one huge table, the OS uses an outer page table that points to smaller inner page tables. Only the inner tables actually used are kept in memory, saving a lot of space.

Q13. Define inverted page table.

Ans: An inverted page table has one entry per physical frame (instead of per virtual page). Each entry stores <process_id, page_number>. Translation requires searching the inverted table; this is slower but uses far less memory than per-process page tables.

Section C: Long Answer Questions (5 marks each)

Q1. Explain the role of base and limit registers in memory protection.

Approach: *Define base and limit registers. Explain that the CPU checks every user-mode address against base ≤ address < base + limit. If the check fails, the hardware traps and the OS terminates the offending*

process. Mention that these registers are loaded only by privileged instructions, so user code cannot bypass the check. Include Figure 5.1.

Q2. Discuss the three address binding times — compile time, load time, and execution time.

Approach: *Define each binding time. Compile-time binding gives absolute addresses (must recompile if location changes). Load-time binding produces relocatable code with addresses fixed by the loader. Execution-time binding allows the program to move during execution and needs the MMU. Conclude that modern OSes use execution-time binding.*

Q3. Compare first-fit, best-fit, and worst-fit allocation strategies with a worked example.

Approach: *Define each strategy. Use the example holes 100, 500, 200, 300, 600 KB and a request of 212 KB to show how each picks differently. Discuss complexity (first-fit O(n) average, best/worst-fit need full scan). Mention that first-fit and best-fit generally beat worst-fit. Include Figure 5.2.*

Q4. Explain internal and external fragmentation. How does compaction help?

Approach: *Define both. Internal — process smaller than partition. External — total free is enough, but scattered. Give an example for each. Explain compaction — shifting processes together to make one big free hole — and discuss its cost. Mention paging as a better long-term solution. Include Figure 5.3.*

Q5. Explain segmentation with diagram. Show address translation with an example.

Approach: *Define segmentation. Describe the segment table (base, limit per entry). Explain translation: lookup s in segment table, check d < limit, compute physical = base + d. Use the example with segment table {(1200, 400), (2000, 600), (5000, 100), (8000, 1000)} and addresses <0, 350>, <2, 150>, <3, 900>. Include Figure 5.4.*

Q6. Describe paging in detail. How does it eliminate external fragmentation?

Approach: *Define pages and frames. Explain that any page can map to any frame. Describe the logical address as (page #, offset). Use a worked example with page size 4 bytes and verify physical = frame × page_size + offset. Explain why fixed-size frames eliminate external fragmentation but cause minor internal fragmentation in the last page only. Include Figure 5.5.*

Q7. Explain the role of the TLB and derive the Effective Access Time (EAT) formula.

Approach: *Describe the problem: paging needs two memory accesses without TLB. Define TLB as a fast hardware cache. Explain hit and miss cases. Derive EAT = h × (TLB + memory) + (1 − h) × (TLB + 2 × memory). Work through an example with h = 80%, TLB = 20 ns, memory = 100 ns to get EAT = 140 ns. Include Figure 5.6.*

Q8. Explain hierarchical (multi-level) page tables.

Approach: *Show why a flat page table is huge for 32-bit (4 MB per process) and impossible for 64-bit. Describe two-level paging — outer table holds pointers to inner tables. Most inner tables don't exist for unused address regions, saving memory. Include Figure 5.7. Mention three- and four-level extensions for 64-bit systems.*

Q9. Compare paging and segmentation in detail. Why are they often combined?

Approach: *Make the comparison table covering unit size, fragmentation, sharing, protection, translation. Discuss strengths of each. Conclude that x86 uses segmentation with paging — segments give the user view, paging eliminates external fragmentation underneath. Include Figure 5.8.*

UNIT 6

Virtual Memory

Contents

1. What is Virtual Memory?

In Unit 5, we saw paging — which allows a process's pages to be scattered across physical memory. Virtual memory takes this idea one step further: a process doesn't need to have all its pages in physical memory to run. Only the pages currently being used need to be in RAM; the rest can stay on disk and be brought in only when required.

The Basic Idea

Your program thinks it has a huge, contiguous memory space — its virtual address space. In reality, only a small working portion is in RAM at any moment. The rest sits on disk in a special area called the swap space or backing store.

> **Real-world analogy —** Think of a student's study table (RAM) and a cupboard full of books (disk). You can't fit every book on your table. You keep only the books you're currently reading on the table; when you need a different book, you swap one off the table and fetch a new one from the cupboard.

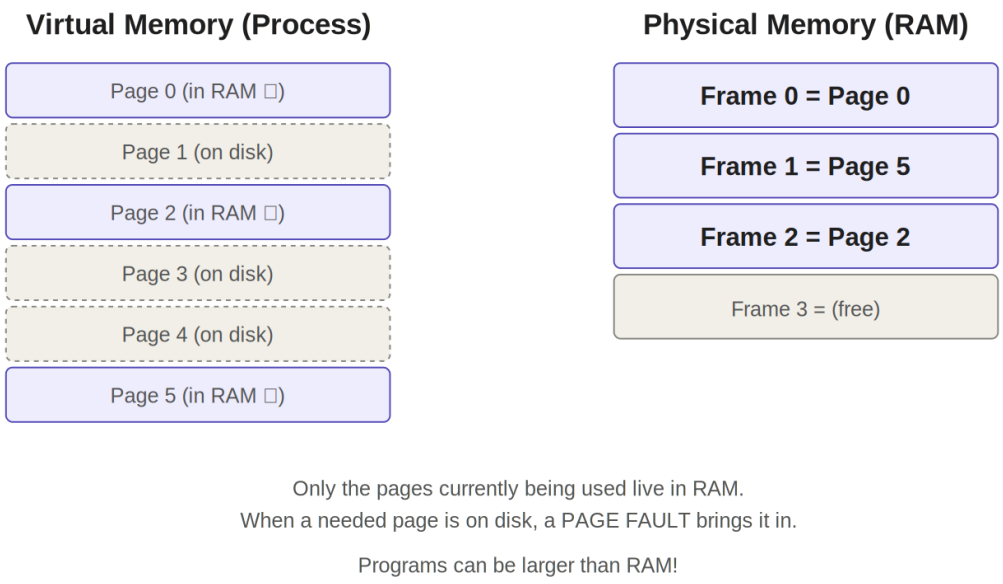


Figure 6.1 — Virtual memory: only some pages live in RAM at any time.

Why Virtual Memory is a Big Deal

30. Programs can be larger than physical memory. A 2 GB program can run on a 1 GB machine.
31. More programs can run simultaneously. Since each program only needs a fraction of its memory in RAM, more programs fit.
32. Less I/O needed to load programs. Only the parts actually used are loaded.
33. Easier programming. Programmers don't have to worry about fitting their program into physical memory.

Virtual Address Space — The Layout

The virtual address space is typically laid out like this (same as process memory layout from Unit 2):

- Text (code) at the bottom
- Data (globals, initialized)
- Heap — grows upward with malloc()
- A large unused region (the "hole")
- Stack — grows downward from the top

This hole between heap and stack is the key insight. With virtual memory, it doesn't actually consume RAM — it's just unused virtual address space. Only when the heap or stack grows into that region do new physical frames get allocated.

2. Demand Paging

The Mechanism

Demand paging is the most common implementation of virtual memory. The rule: bring a page into memory only when it is needed, not before. This is called a lazy swapper — a swapper that never brings a page in until required.

When the program starts, almost nothing is loaded. As the program tries to access various addresses, the OS brings in pages on demand.

Valid/Invalid Bit in Page Table

Each page table entry has a valid-invalid bit (also called a present bit):

- **Valid (v)** — the page is in memory; access is legal.
- **Invalid (i)** — either the page is not in memory (but exists on disk), or the page is not part of the process's address space.

When a program accesses a page marked invalid, the hardware generates a page fault trap. The OS catches this trap and handles it.

Page Fault Handling — Step by Step

A page fault is not an error — it's a normal, expected event in a demand-paged system. Here's what happens:

34. CPU tries to access a page. The page table says the page is invalid.
35. Hardware traps to the OS (this is the page fault).
36. OS checks an internal table — is this a valid reference? If not, kill the process.
37. If valid, OS finds a free frame in RAM.
38. OS schedules a disk read to bring the page from the backing store into the chosen frame.
39. While waiting for the disk read (several milliseconds!), the OS may context-switch to another process.
40. When the disk read completes, OS updates the page table — the page is now valid, and its frame is noted.
41. The instruction that caused the page fault is restarted. This time it succeeds.

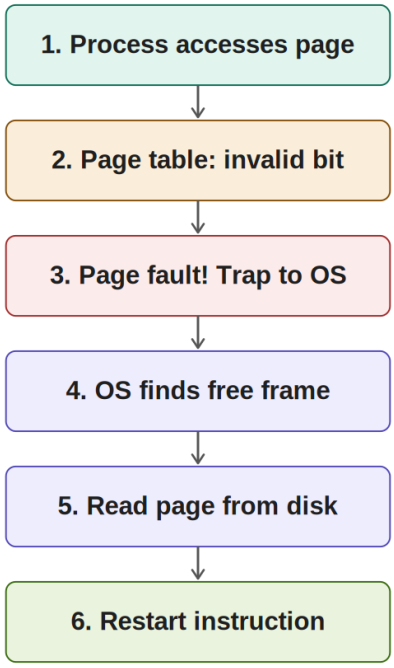


Figure 6.2 — The eight steps of page fault handling.

Performance of Demand Paging

Let:

- p = probability of a page fault ($0 \le p \le 1$)
- ma = memory access time
- page_fault_time = time to service a fault

> **EAT formula —** Effective Access Time (EAT) = $(1 - p) \times ma + p \times page\_fault\_time$

The page_fault_time is huge compared to ma. A typical disk access is 8 ms = 8,000,000 ns; RAM access is 100 ns. So a tiny page fault rate can destroy performance.

Worked example: ma = 100 ns, page_fault_time = 8 ms.

If p = 0.001 (one fault in 1000 accesses):

```
EAT = 0.999 × 100 + 0.001 × 8,000,000
    = 99.9 + 8,000
    = 8,099.9 ns ≈ 8.1 µs
```

That's 81× slower than pure RAM access! Even a 0.1% fault rate makes the system feel sluggish. Target: p < 0.00001 for acceptable performance.

3. Copy-on-Write (COW)

When a process calls fork(), the child is supposed to get a copy of the parent's memory. Traditionally, the OS would copy every page. But here's the thing — most fork() calls are followed by exec(), which replaces the child's memory anyway. So copying all pages upfront is wasteful.

Copy-on-Write is an optimization. Instead of duplicating pages, parent and child initially share all pages, with each marked read-only. If either one tries to write to a shared page, only then does the OS make a copy — and only for that one page.

Step-by-Step COW

42. Parent calls fork(). Instead of copying pages, OS marks all pages as COW (shared + read-only).
43. Child and parent both use the same physical pages. They can read freely.
44. Suppose child writes to page P: the write attempt triggers a page fault. OS recognizes it as a COW fault, allocates a new frame, copies page P into it, and updates the child's page table to point to the new frame, marked read-write. Parent's page table still points to the original frame.
45. If child immediately calls exec(), most pages are never written, so they are never copied. Huge savings.

Benefits

- Much faster fork() — no copying upfront.
- Memory savings — shared pages stay shared as long as neither writes.
- Transparent — the program doesn't know or care that COW is happening.

Linux, Windows, and macOS all use COW for fork().

4. Page Replacement Algorithms

What if, during a page fault, there's no free frame in RAM? We must evict some existing page to make room. The policy for choosing which page to evict is called a page replacement algorithm. The goal is to minimize the number of page faults.

Modified Page Fault Handling

With page replacement, the fault handling adds these steps:

- If there's a free frame, use it.
- If no free frame exists, apply the page replacement algorithm to pick a "victim" page.
- Write the victim back to disk (if it was modified).
- Use its frame for the new page.

Optimization — Dirty bit. Each page has a dirty bit (also called modified bit). If the page was never written (dirty bit = 0), we don't need to write it back to disk before evicting — the disk copy is already up to date. This cuts page replacement cost roughly in half.

Reference String

To compare algorithms, we use a reference string — the sequence of page numbers a process accesses over time. For example: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.

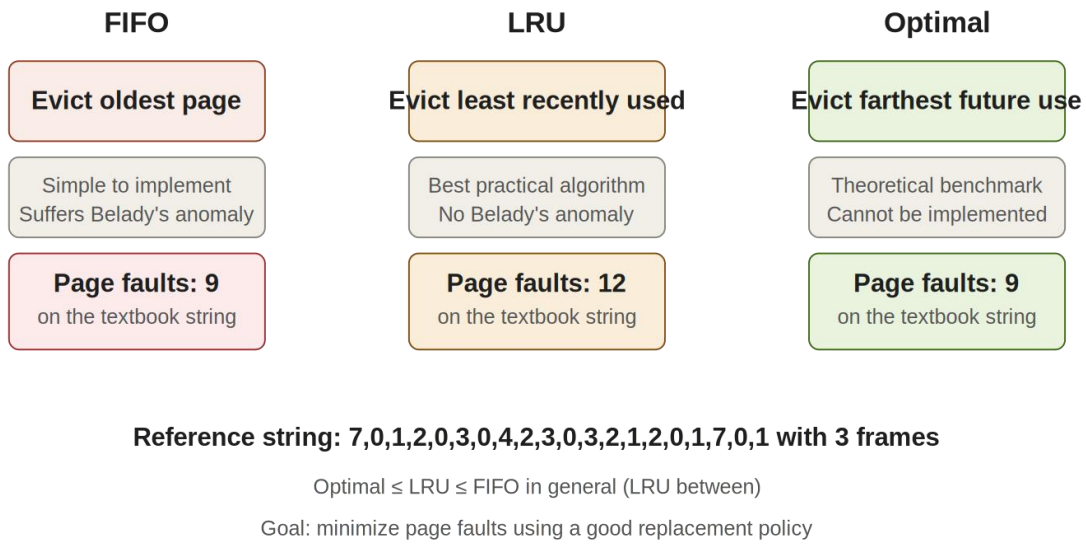


Figure 6.3 — FIFO, LRU, and Optimal page replacement compared.

4.1 FIFO Page Replacement

The simplest algorithm. Evict the page that has been in memory the longest (the oldest one). Implemented with a FIFO queue of frames.

Example: Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1. With 3 frames:

Ref	F1	F2	F3	Hit/Miss
7	7	–	–	Miss
0	7	0	–	Miss
1	7	0	1	Miss
2	2	0	1	Miss (evict 7)
0	2	0	1	Hit
3	2	3	1	Miss (evict 0)
0	2	3	0	Miss (evict 1)
4	4	3	0	Miss (evict 2)
2	4	2	0	Miss (evict 3)
3	4	2	3	Miss (evict 0)
0	0	2	3	Miss (evict 4)
3	0	2	3	Hit
2	0	2	3	Hit
1	0	1	3	Miss (evict 2)
2	0	1	2	Miss (evict 3)
0	0	1	2	Hit
1	0	1	2	Hit
7	7	1	2	Miss (evict 0)
0	7	0	2	Miss (evict 1)
1	7	0	1	Miss (evict 2)

Total page faults = 15.

- **Pros:** Simple to understand and implement.
- **Cons:** Ignores how often or how recently a page was used. Can evict a very active page just because it was loaded first.

Belady's Anomaly

With FIFO, intuition says: "more frames → fewer page faults." Surprisingly, this isn't always true. Classic example with reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5:

- With 3 frames: 9 page faults.
- With 4 frames: 10 page faults.

Adding a frame actually increased faults. This counter-intuitive behaviour is called Belady's anomaly. Only FIFO (and a few similar algorithms) suffer from it. Algorithms like LRU and Optimal do not.

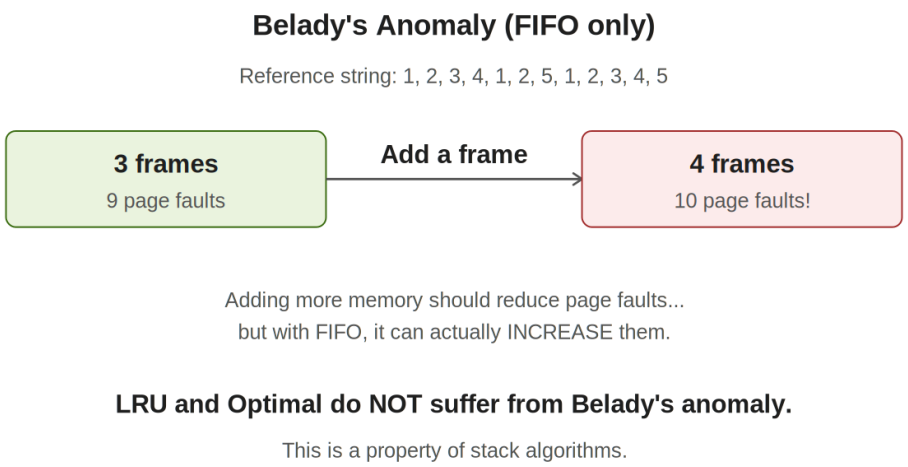


Figure 6.4 — Belady's anomaly: more frames sometimes means more faults.

4.2 Optimal (OPT) Page Replacement

The theoretical best. When a page must be evicted, pick the page that will not be used for the longest time in the future.

Example: same reference string. With 3 frames:

Ref	F1	F2	F3	Hit/Miss	Reasoning
7	7	–	–	Miss	
0	7	0	–	Miss	
1	7	0	1	Miss	
2	2	0	1	Miss	Evict 7 (next use farthest)
0	2	0	1	Hit	
3	2	0	3	Miss	Evict 1 (next use farthest)
0	2	0	3	Hit	
4	2	4	3	Miss	Evict 0
2	2	4	3	Hit	
3	2	4	3	Hit	
0	2	0	3	Miss	Evict 4
3	2	0	3	Hit	
2	2	0	3	Hit	
1	2	0	1	Miss	Evict 3
2	2	0	1	Hit	
0	2	0	1	Hit	
1	2	0	1	Hit	
7	7	0	1	Miss	Evict 2
0	7	0	1	Hit	
1	7	0	1	Hit	

Total page faults = 9.

- **Pros:** Guaranteed minimum number of page faults.
- **Cons:** Impossible to implement in practice — the OS cannot predict the future. Optimal is used as a benchmark against which other algorithms are compared.

4.3 LRU (Least Recently Used) Page Replacement

The OS can't see the future, but it can look at the past. LRU assumes the recent past is a good predictor of the near future — so evict the page that has not been used for the longest time.

Example: same reference string. With 3 frames:

Ref	F1	F2	F3	Hit/Miss	LRU evicted
7	7	–	–	Miss	
0	7	0	–	Miss	
1	7	0	1	Miss	
2	2	0	1	Miss	Evict 7
0	2	0	1	Hit	
3	2	0	3	Miss	Evict 1
0	2	0	3	Hit	
4	4	0	3	Miss	Evict 2
2	4	0	2	Miss	Evict 3
3	4	3	2	Miss	Evict 0
0	0	3	2	Miss	Evict 4
3	0	3	2	Hit	
2	0	3	2	Hit	
1	1	3	2	Miss	Evict 0
2	1	3	2	Hit	
0	1	0	2	Miss	Evict 3
1	1	0	2	Hit	
7	1	0	7	Miss	Evict 2
0	1	0	7	Hit	
1	1	0	7	Hit	

Total page faults = 12.

LRU (12) is between Optimal (9) and FIFO (15) — as expected. LRU is very good in practice and doesn't suffer from Belady's anomaly.

Implementing LRU

Requires hardware support. Two common approaches:

- **Counters** — each page has a timestamp updated on every reference. To evict, find page with oldest timestamp. High overhead.
- **Stack** — maintain a stack of page numbers. On each reference, move the page to the top. To evict, pick the page at the bottom.

True LRU is expensive, so real systems use LRU approximations (e.g., the reference bit algorithm, second-chance algorithm, clock algorithm).

Comparison Summary

For the textbook reference string with 3 frames:

Algorithm	Page Faults	Comments
Optimal (OPT)	9	Theoretical best
LRU	12	Good approximation
FIFO	15	Simple but weak

5. Frame Allocation

When there are multiple processes, each needs some frames. How many frames should each process get? This is the frame allocation problem.

Minimum Number of Frames

Each process needs a minimum number of frames, determined by the CPU architecture — specifically, by the maximum number of memory references a single instruction can make. If an instruction can reference 3 memory locations, we need at least 3 frames to run the instruction; otherwise it could page-fault endlessly.

Allocation Algorithms

- **Equal allocation.** If we have m frames and n processes, each process gets m/n frames. Simple, but unfair — a small process gets the same share as a big one.
- **Proportional allocation.** Give each process frames proportional to its size. If total size is S and process P_i has size s_i, then process P_i gets a_i = (s_i / S) × m frames.

Example: 62 free frames. Two processes: P1 = 10 KB, P2 = 127 KB.

- s_1/S = 10/137, a_1 = (10/137) × 62 ≈ 4 frames.
- s_2/S = 127/137, a_2 = (127/137) × 62 ≈ 57 frames.
- P2 gets far more frames because it's much larger.

Priority can also factor in — high-priority processes get more frames even if they aren't larger.

Global vs Local Replacement

- **Local replacement —** process can only take from its own allocated frames. Total frames per process stays fixed.
- **Global replacement —** process can take a frame from any other process in the system. More flexible; can give a struggling process more frames. But one process's behaviour affects others.

Global replacement generally gives better system-wide throughput and is used in most modern OSes, though a single misbehaving process can hurt others.

6. Thrashing

What is Thrashing?

Suppose a process doesn't have enough frames to hold its working set (the pages it's actively using). Every page it needs triggers a fault, evicting another page it just needed. That evicted page soon page-faults too, kicking out another active page. The process spends almost all its time page-faulting, doing very little actual work.

This is called thrashing. The CPU stays busy — but only servicing page faults, not running user code.

How Thrashing Starts

46. OS sees low CPU utilization, increases degree of multiprogramming (adds more processes).

47. New processes need frames — stolen from existing processes via global replacement.
48. Each process now has too few frames; all start page-faulting heavily.
49. CPU utilization drops further (everyone is waiting on disk).
50. OS thinks "CPU is underused" and adds more processes.
51. Things get dramatically worse.

This positive feedback loop is why thrashing is so destructive.

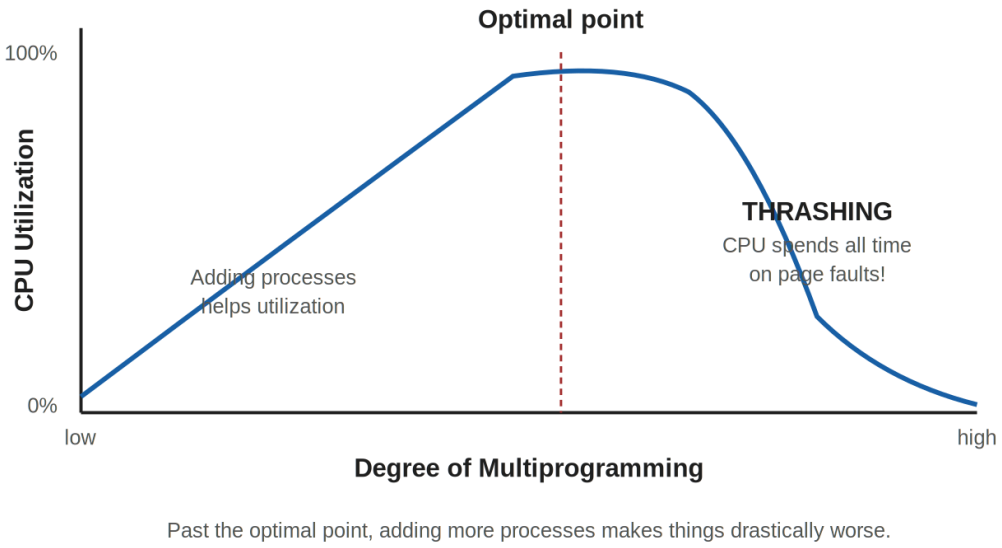


Figure 6.5 — CPU utilization vs degree of multiprogramming. After the peak, thrashing causes a sharp drop.

Preventing Thrashing

Two main techniques:

- **1. Working Set Model.** For each process, track the set of pages it has referenced in the last Δ time units. This is its working set W(Δ). Roughly, these are the pages it actually needs. If the OS can give every process enough frames to hold its working set, thrashing is avoided. If total working sets exceed available frames, the OS suspends some processes.
- **2. Page-Fault Frequency (PFF).** Monitor each process's page fault rate. Set upper and lower bounds: if rate exceeds upper bound, give the process more frames; if rate falls below lower bound, take frames away. If no frames are available and a process is still thrashing, suspend one process.

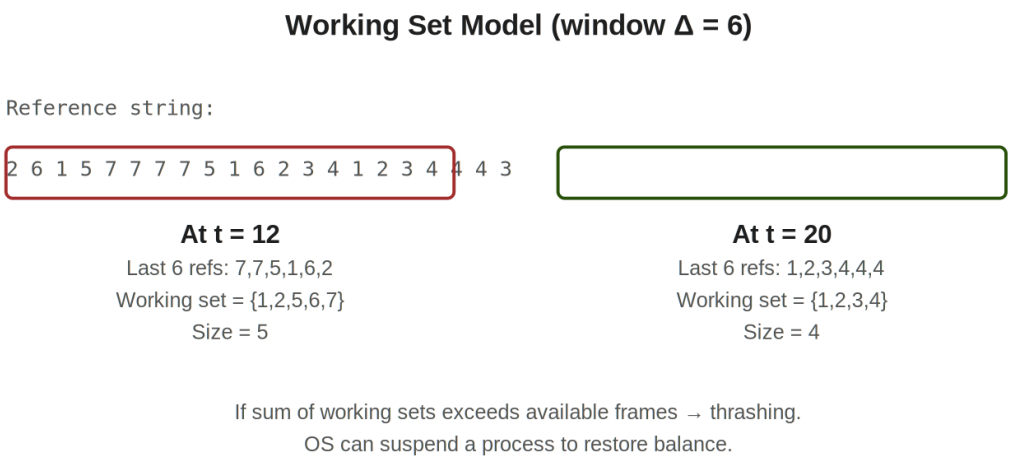


Figure 6.6 — Working set model: track recently used pages within a window Δ.

Working Set Example

Reference string: 1, 2, 3, 4, 5, 2, 1, 3, 2, 3, 4, 4, 4, 3, 4, 3, 4, 4, 3, 3, 1, 2, 3, 4. With Δ = 10:

- At t = 10 (after first 10 refs): Working set = {1, 2, 3, 4, 5}.
- At t = 20: refs 11–20 = 4, 4, 3, 4, 3, 4, 4, 3, 3, 1. Working set = {1, 3, 4}.

The working set shrinks from 5 pages to 3 pages as the program moves from a busy phase into a narrower phase.

7. Numerical Questions with Solutions

Numerical 1 — EAT with Page Fault Rate

Q: Memory access time = 200 ns. Average page fault service time = 8 ms. Page fault rate = 0.002. Calculate EAT.

Solution:

$$\begin{aligned}\text{EAT} &= (1 - p) \times ma + p \times \text{page\_fault\_time} \\ &= (1 - 0.002) \times 200 + 0.002 \times 8{,}000{,}000 \\ &= 0.998 \times 200 + 0.002 \times 8{,}000{,}000 \\ &= 199.6 + 16{,}000 \\ &= 16{,}199.6 \text{ ns} \approx 16.2 \text{ µs}\end{aligned}$$

The single term 0.002 × 8,000,000 = 16,000 ns completely dominates. A 0.2% fault rate makes the system 80× slower. This is why modern systems aim for fault rates below 0.00001.

Numerical 2 — Acceptable Fault Rate

Q: Memory access = 100 ns. Page fault service = 10 ms. We want EAT to be no more than 110% of the raw memory access time. What is the maximum acceptable page fault rate?

Solution:

Target EAT $\leq 1.10 \times 100 = 110$ ns.

$$(1 - p) \times 100 + p \times 10{,}000{,}000 \leq 110$$
$$100 - 100p + 10{,}000{,}000p \leq 110$$
$$100 + 9{,}999{,}900p \leq 110$$
$$9{,}999{,}900p \leq 10$$
$$p \leq 10 / 9{,}999{,}900 \approx 0.000001 = 1 \times 10^{-6}$$

Answer: no more than 1 page fault per million memory accesses is acceptable — remarkably strict.

Numerical 3 — FIFO Page Replacement

Q: Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. 3 frames. Compute page faults using FIFO.

Solution:

Ref	F1	F2	F3	Hit/Miss	Notes
1	1	–	–	Miss	
2	1	2	–	Miss	
3	1	2	3	Miss	
4	4	2	3	Miss	Evict oldest = 1
1	4	1	3	Miss	Evict 2
2	4	1	2	Miss	Evict 3
5	5	1	2	Miss	Evict 4
1	5	1	2	Hit	
2	5	1	2	Hit	
3	5	3	2	Miss	Evict 1
4	5	3	4	Miss	Evict 2
5	5	3	4	Hit	

Total page faults with 3 frames = 9.

With 4 frames (to demonstrate Belady's anomaly):

Ref	F1	F2	F3	F4	Hit/Miss
1	1	–	–	–	Miss
2	1	2	–	–	Miss
3	1	2	3	–	Miss
4	1	2	3	4	Miss
1	1	2	3	4	Hit
2	1	2	3	4	Hit
5	5	2	3	4	Miss (evict 1)
1	5	1	3	4	Miss (evict 2)
2	5	1	2	4	Miss (evict 3)
3	5	1	2	3	Miss (evict 4)
4	4	1	2	3	Miss (evict 5)
5	4	5	2	3	Miss (evict 1)

Total page faults with 4 frames = 10.

Adding a frame increased faults from 9 to 10 — classic Belady's anomaly.

Numerical 4 — Optimal Page Replacement

Q: Reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. 3 frames. Apply Optimal.

Solution:

Step	Ref	F1	F2	F3	Hit/Miss	Reasoning
1	1	1	–	–	Miss	
2	2	1	2	–	Miss	
3	3	1	2	3	Miss	
4	4	1	2	4	Miss	Evict 3 (next use farthest)
5	1	1	2	4	Hit	
6	2	1	2	4	Hit	
7	5	1	2	5	Miss	Evict 4
8	1	1	2	5	Hit	
9	2	1	2	5	Hit	
10	3	3	2	5	Miss	Evict 1 (not used again)
11	4	3	4	5	Miss	Evict 2 (not used again)
12	5	3	4	5	Hit	

Total page faults with Optimal = 7.

Numerical 5 — LRU Page Replacement

Q: Reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. 3 frames. Apply LRU.

Solution:

Step	Ref	After (MRU → LRU)	Hit/Miss	Action
1	1	1	Miss	Load
2	2	2, 1	Miss	Load
3	3	3, 2, 1	Miss	Load
4	4	4, 3, 2	Miss	Evict 1 (LRU)
5	1	1, 4, 3	Miss	Evict 2
6	2	2, 1, 4	Miss	Evict 3
7	5	5, 2, 1	Miss	Evict 4
8	1	1, 5, 2	Hit	Reorder
9	2	2, 1, 5	Hit	Reorder
10	3	3, 2, 1	Miss	Evict 5
11	4	4, 3, 2	Miss	Evict 1
12	5	5, 4, 3	Miss	Evict 2

Total page faults with LRU = 10.

Summary for this string with 3 frames:

Algorithm	Page Faults
Optimal	7
FIFO	9
LRU	10

Interesting: on this particular string, FIFO (9) happens to beat LRU (10). That's not common but it can happen — LRU is just an approximation of Optimal.

Numerical 6 — Hit Ratio Calculation

Q: Given a reference string of 20 page references and 5 page faults using a particular algorithm, compute (a) the page fault rate and (b) the hit ratio.

Solution:

(a) Page fault rate = faults / total references = 5 / 20 = 0.25 = 25%.

(b) Hit ratio = 1 − 0.25 = 0.75 = 75%.

Or directly: hits = 20 − 5 = 15. Hit ratio = 15 / 20 = 75%.

Numerical 7 — Frame Allocation (Proportional)

Q: 60 free frames. Four processes: P1 = 10 KB, P2 = 30 KB, P3 = 50 KB, P4 = 60 KB. How many frames does each get under proportional allocation?

Solution:

Total size S = 10 + 30 + 50 + 60 = 150 KB. Formula: a_i = (s_i / S) × m where m = 60.

- P1: (10/150) × 60 = 4 frames
- P2: (30/150) × 60 = 12 frames
- P3: (50/150) × 60 = 20 frames
- P4: (60/150) × 60 = 24 frames

Check: 4 + 12 + 20 + 24 = 60 ✓

Numerical 8 — Working Set

Q: Reference string: 2, 6, 1, 5, 7, 7, 7, 7, 5, 1, 6, 2, 3, 4, 1, 2, 3, 4, 4, 4 ... with window Δ = 6. Find the working set at positions t = 12 and t = 20.

Solution:

At t = 12: last 6 references (positions 7–12) = 7, 7, 5, 1, 6, 2. Distinct pages = {1, 2, 5, 6, 7}. Working set size = 5.

At t = 20: positions 15–20 = 1, 2, 3, 4, 4, 4. Distinct pages = {1, 2, 3, 4}. Working set size = 4.

As the program progresses, the working set shifts from early pages into later pages — phase behavior. Programs often exhibit distinct phases, each with its own working set.

Numerical 9 — Maximum Processes Before Thrashing

Q: A machine has 32 frames. Each process needs a working set of 6 frames. What is the maximum number of processes the system can run without thrashing?

Solution:

Max processes = 32 / 6 = 5.33 → 5 processes.

Frames used = 5 × 6 = 30. Remaining = 2 frames (held in reserve).

If we try 6 processes, frames needed = 36 > 32. Result: thrashing.

Numerical 10 — Three-Algorithm Comparison

Q: Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3. 3 frames. Compare faults under FIFO, Optimal, LRU.

Solution:

FIFO trace:

Ref	Frames	Fault?
7	[7]	Miss
0	[7, 0]	Miss
1	[7, 0, 1]	Miss
2	[0, 1, 2]	Miss (evict 7)
0	[0, 1, 2]	Hit
3	[1, 2, 3]	Miss (evict 0)
0	[2, 3, 0]	Miss (evict 1)
4	[3, 0, 4]	Miss (evict 2)
2	[0, 4, 2]	Miss (evict 3)
3	[4, 2, 3]	Miss (evict 0)

FIFO faults = 9.

LRU faults = 7. Optimal faults = 6.

Final comparison:

Algorithm	Faults
Optimal	6
LRU	7
FIFO	9

As expected: Optimal ≤ LRU ≤ FIFO on this string.

8. Quick Revision — One-Page Summary

Here is everything in Unit 6 compressed for last-minute revision.

- **Virtual memory** — run programs larger than RAM. Only actively-used pages stay in RAM.
- **Demand paging** — bring pages into memory only when accessed. Lazy swapper.
- **Page fault** — reference to a page not in memory. OS finds frame → schedules I/O → updates page table → restarts instruction.
- **Copy-on-Write (COW)** — after fork(), parent and child share pages read-only. Copy made only on first write. Fast fork() + memory savings.

Three page replacement algorithms:

- **FIFO** — evict oldest. Simple. Suffers Belady's anomaly.
- **Optimal (OPT)** — evict page with farthest next use. Benchmark. Cannot be implemented.
- **LRU** — evict page not used for longest. Best practical algorithm; no Belady's anomaly.
- **Belady's anomaly** — in FIFO, more frames can sometimes mean more faults.
- **Dirty bit** — tracks if page was written. Clean pages don't need disk write-back on eviction.

Frame allocation:

- Equal — m/n frames each.
- Proportional — $a_i = (s_i / S) \times m$.
- **Global vs local replacement:** global takes from any process; local only from own.

- **Thrashing** — process spends more time page-faulting than executing. CPU utilization drops sharply.

Prevention techniques:

- Working set model — give each process enough frames for its W(Δ).
- Page-fault frequency (PFF) — adjust frames based on fault rate bounds.

Key formulas:

- EAT = (1 − p) × ma + p × page_fault_time
- Page fault rate = faults / references
- Hit ratio = 1 − fault rate
- Proportional allocation: a_i = (s_i / S) × m
- Max processes before thrash ≈ total_frames / working_set_size

9. Practice Questions

Section A: Multiple Choice Questions (1 mark each)

Q1. Virtual memory allows programs to:

a) Run faster than RAM
b) Be larger than physical memory
c) Skip the OS
d) Use only registers
Answer: (b)

Q2. Demand paging brings pages in:

a) All at once at program start
b) Only when needed
c) Never
d) Periodically
Answer: (b)

Q3. A page fault occurs when:

a) The CPU is busy
b) A process accesses a page not in memory
c) Memory is full
d) The OS shuts down
Answer: (b)

Q4. The valid-invalid bit indicates:

a) Whether the page is dirty
b) Whether the page is in memory
c) Whether the process is running
d) Whether the disk is connected
Answer: (b)

Q5. If page fault rate is 0.001, memory access = 100 ns, page fault service = 8 ms, EAT is approximately:

a) 100 ns
b) 200 ns
c) 8 µs
d) 8 ms
Answer: (c) — about 8.1 µs.

Q6. Copy-on-Write is used to:

a) Speed up file copying
b) Avoid copying memory unnecessarily after fork()
c) Backup the OS
d) Prevent page faults
Answer: (b)

Q7. Which algorithm is theoretically optimal for page replacement?

a) FIFO
b) LRU
c) OPT
d) Random
Answer: (c)

Q8. The Optimal algorithm cannot be implemented because:

a) It uses too much memory

b) It needs to know the future
c) It is too slow
d) It is patented
Answer: (b)

Q9. Which algorithm suffers from Belady's anomaly?

a) LRU
b) FIFO
c) OPT
d) None
Answer: (b)

Q10. Belady's anomaly means:

a) More frames always means fewer faults
b) Adding frames can increase faults
c) FIFO is optimal
d) LRU is wrong
Answer: (b)

Q11. The dirty bit is used to:

a) Detect viruses
b) Avoid writing unmodified pages back to disk
c) Identify the running process
d) Mark stale TLB entries
Answer: (b)

Q12. The working set of a process is:

a) Pages used in the most recent Δ time units
b) All pages of the process
c) Pages on disk
d) Pages in the TLB
Answer: (a)

Q13. Thrashing occurs when:

a) The CPU runs too fast
b) Processes spend more time page-faulting than executing
c) The disk fails
d) Cache is full
Answer: (b)

Q14. During thrashing, CPU utilization:

a) Increases
b) Stays the same
c) Drops sharply
d) Becomes 100%
Answer: (c)

Q15. Proportional frame allocation gives each process frames in proportion to its:

a) Priority
b) Size
c) Age
d) Burst time
Answer: (b)

Q16. Reference string 1,2,3,4,1,2,5,1,2,3,4,5 with 3 frames using FIFO causes how many faults?

a) 7
b) 9
c) 10
d) 12
Answer: (b)

Q17. In LRU, the page evicted is the one:

a) Loaded earliest
b) Used least often
c) Not used for the longest time
d) Loaded last
Answer: (c)

Q18. Local frame replacement means:

a) Process can take any frame
b) Process can only take from its own allocated frames
c) The OS replaces all frames at once
d) Frames are local to a CPU
Answer: (b)

Q19. 32 frames, working set per process = 4 frames. Max processes without thrashing:

a) 4
b) 6
c) 8
d) 16
Answer: (c) — 32/4 = 8.

Q20. Page Fault Frequency (PFF) prevents thrashing by:

a) Reducing CPU clock speed
b) Adjusting frames based on fault rate bounds
c) Killing all processes
d) Disabling paging

Answer: (b)

Section B: Short Answer Questions (2 marks each)

Q1. Define virtual memory.

Ans: Virtual memory is a memory management technique that lets a process run with only a subset of its pages in physical memory; the rest reside on disk and are brought in only when accessed. This allows programs to be larger than RAM, more programs to run simultaneously, and reduces I/O for loading.

Q2. What is demand paging?

Ans: Demand paging is the implementation of virtual memory where a page is brought into memory only when it is actually referenced — not at program start. The mechanism is sometimes called a lazy swapper.

Q3. Define page fault and explain how it is handled.

Ans: A page fault is a trap raised by the hardware when a process accesses a page whose valid-invalid bit says it is not in memory. The OS handles it by finding a free frame, loading the page from disk, updating the page table, and restarting the faulting instruction. The process can context-switch out during the disk wait.

Q4. State the EAT formula for demand paging.

Ans: EAT = (1 − p) × ma + p × page_fault_time, where p is the page fault probability, ma is memory access time, and page_fault_time is the cost of servicing a fault. Even tiny p values can dominate because page_fault_time is much larger than ma.

Q5. What is Copy-on-Write (COW)?

Ans: COW is an optimization for fork(). Instead of copying every page of the parent to the child, the OS marks pages as read-only and shared. The child and parent share the physical pages until one tries to write, at which point only that page is copied. Saves time and memory, especially when fork() is followed by exec().

Q6. What is page replacement?

Ans: When a page fault occurs and no free frame is available, the OS must evict (replace) some page in RAM to make room for the incoming page. The algorithm that chooses which page to evict is called a page replacement algorithm.

Q7. What is the role of the dirty bit in page replacement?

Ans: The dirty bit (modified bit) records whether a page has been written to since it was loaded. If a page is clean (dirty bit = 0), the OS does not need to write it back to disk before evicting; the disk copy is already up to date. This roughly halves the average page replacement cost.

Q8. Differentiate between FIFO, Optimal, and LRU page replacement.

Ans: FIFO evicts the oldest page in memory — simple but can evict active pages and suffers Belady's anomaly. Optimal evicts the page that will not be used for the longest time in the future — gives the minimum faults but cannot be implemented (needs to know the future). LRU evicts the page not used for the longest time in the past — good approximation of Optimal, no Belady's anomaly.

Q9. What is Belady's anomaly?

Ans: Belady's anomaly is the counter-intuitive phenomenon where adding more frames to a process can actually increase the number of page faults under FIFO. LRU and Optimal do not suffer from this anomaly.

Q10. What is thrashing?

Ans: Thrashing is a state where a process (or the system) spends more time on page faults than on actual execution. It happens when processes don't have enough frames for their working sets. CPU utilization drops sharply because everyone is waiting on disk I/O.

Q11. What is the working set model?

Ans: The working set model defines W(Δ) as the set of pages a process has referenced in the last Δ time units. These are roughly the pages the process actually needs. If the OS gives each process enough frames for its working set, thrashing is avoided.

Q12. Differentiate between equal and proportional frame allocation.

Ans: Equal allocation gives each of n processes m/n frames regardless of size. Proportional allocation gives each process frames proportional to its size: a_i = (s_i / S) × m. Proportional is fairer when processes vary widely in size.

Q13. Differentiate between local and global page replacement.

Ans: Local replacement: a process can only evict from its own allocated frames. Global replacement: a process can evict from any process's frames. Global gives better overall throughput but lets a misbehaving process hurt others.

Section C: Long Answer Questions (5 marks each)

Q1. Explain virtual memory and demand paging. How does the OS handle a page fault?

Approach: *Define virtual memory and its benefits. Explain demand paging as a lazy swapper. Describe the valid-invalid bit. Walk through the eight steps of page fault handling. Conclude with the importance of low page fault rates. Include Figures 6.1 and 6.2.*

Q2. Derive the EAT formula for demand paging and work through an example.

Approach: *Define p, ma, page_fault_time. Derive EAT = (1 − p) × ma + p × page_fault_time. Use the example ma = 100 ns, page_fault_time = 8 ms, p = 0.001 to get EAT ≈ 8.1 µs. Show that p dominates EAT and explain why systems aim for very low fault rates.*

Q3. Explain Copy-on-Write with the steps the OS performs when a fork() is followed by a write.

Approach: *Describe the inefficiency of copying all pages on fork(). Explain how COW shares pages as read-only initially. Walk through the four steps: mark COW, share, write triggers fault, OS copies only that page. Explain why fork() + exec() benefits hugely from COW.*

Q4. Compare FIFO, Optimal, and LRU page replacement with a worked example. Demonstrate Belady's anomaly.

Approach: *Define each algorithm. Apply all three to the reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with 3 frames. Show fault counts: Optimal 9, LRU 12, FIFO 15. Then demonstrate Belady's anomaly with the string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 — FIFO gives 9 faults with 3 frames but 10 with 4 frames. Include Figures 6.3 and 6.4.*

Q5. Explain frame allocation. Compare equal and proportional allocation. Discuss global vs local replacement.

Approach: *Define the frame allocation problem. Explain equal allocation (m/n) and proportional allocation ($a_i = (s_i / S) \times m$) with the 62-frame example. Discuss minimum frame count constraint. Then explain global vs local replacement, comparing flexibility and isolation.*

Q6. Define thrashing. Explain how it starts and discuss two techniques to prevent it.

Approach: *Define thrashing. Walk through the feedback loop: low CPU → add more processes → starved frames → more faults → CPU utilization drops further → OS adds even more processes. Show the CPU utilization curve. Then explain working set model and page-fault frequency (PFF) as prevention techniques. Include Figures 6.5 and 6.6.*

Q7. Solve: A reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3 is given. With 3 frames, compute the number of page faults under FIFO, LRU, and Optimal.

Approach: *For each algorithm, build the trace table step by step showing the contents of each frame after every reference and marking hit/miss. Provide the eviction reasoning for Optimal. Final answer: FIFO 9, LRU 7, Optimal 6. Conclude that Optimal ≤ LRU ≤ FIFO.*

Q8. Discuss the working set model in detail with an example.

Approach: *Define $W(\Delta)$ as the set of pages referenced in the last Δ time units. Use the textbook example reference string with $\Delta = 10$ to compute the working set at $t = 10$ and $t = 20$. Show how the working set changes as the program moves through phases. Explain how the OS uses working set sizes to allocate frames and detect when a process needs to be suspended.*

Q9. Discuss the relationship between page fault rate and EAT. Find the maximum acceptable page fault rate so EAT is within 110% of memory access time, given ma = 100 ns and page_fault_time = 10 ms.

Approach: *Set up the inequality $(1 - p) \times 100 + p \times 10{,}000{,}000 \leq 110$. Solve to find $p \leq 1 \times 10^{-6}$. Discuss how strict this constraint is — only 1 fault per million accesses. Explain why modern systems work hard to keep TLB hits high and working sets fitted in memory.*