Operating System

A Complete Textbook for BTEUP CSE, 3rd Semester (Subject 3.1)

KAMALDNP kamaldnp.com

About this book

This book teaches the Operating System subject of the Board of Technical Education, Uttar Pradesh (BTEUP) — Computer Science & Engineering, 3rd Semester, Subject 3.1 — from the first definition to the last file system, following the official syllabus unit by unit and topic by topic. Every chapter explains its ideas in simple language first, works the numerical problems fully (CPU scheduling, Banker's algorithm, memory fits, paging, page replacement, disk scheduling), and ends with practice. A companion Question Bank — objective, short-answer and long-answer questions with answers, worked numericals and two model papers on the board pattern — is published alongside this book.

Contents

- **Unit 1 — Overview of Operating System:** Chapter 1 Introduction and Types · Chapter 2 Functions and Services, the Kernel, System Calls and System Programs

- **Unit 2 — Process Management:** Chapter 3 Process Concepts and Operations · Chapter 4 CPU Scheduling · Chapter 5 IPC, Synchronisation and the Critical Section · Chapter 6 Deadlock and the Banker's Algorithm
- **Unit 3 — Memory Management:** Chapter 7 Memory Allocation: First, Best and Worst Fit · Chapter 8 Swapping, Paging and Segmentation · Chapter 9 Page Faults, Page Replacement, Fragmentation and Virtual Memory
- **Unit 4 — I/O System:** Chapter 10 Mass Storage and Disk Structure · Chapter 11 Disk Scheduling, Swap Space and RAID
- **Unit 5 — File Management:** Chapter 12 File Concept, Access Methods, Directory Structure and Mounting · Chapter 13 File Sharing and Protection, Implementation and Types
- **Appendix A** — The Operating System Lab: 12 experiments

How to use this book

Read the chapters in order; each unit builds on the one before. Do the small practice set inside every section before reading on, solve the chapter practice set on paper, and then test yourself with the Question Bank. The worked numericals are written to be copied as method: Gantt chart first, formulas next, answer last.

Published by KAMALDNP · kamaldnp.com · support@kamaldnp.com

Chapter 1: Introduction and Types of Operating Systems

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 1 — Overview of Operating System)

What you will learn

- What an operating system is — a complete definition you can understand in one reading
- Why a computer cannot work usefully without one
- The two main goals of an operating system
- How operating systems evolved, in brief
- The main types: batch, multiprogrammed, multitasking and time-sharing, multiprocessing, real-time, distributed and embedded — and how they differ

1.1 What is an operating system?

An **operating system (OS)** is the system software that acts as the layer between the user (and the application programs) and the computer hardware: it controls and coordinates the whole machine, manages all its resources — the processor (CPU), memory, storage disks and input/output devices — and provides the services and environment that application programs need in order to run. Every program, from a browser to a game, runs under the operating system; the user never deals with the hardware directly.

Think of a big restaurant. The customers never enter the kitchen, and the cooks do not take orders from tables. Between them stands the manager with the waiters: taking orders, deciding which cook prepares what, making sure two orders do not collide, and keeping track of ingredients. The operating system is that manager for the computer — the hardware is the kitchen, your programs are the customers' orders, and the OS keeps everything moving in an orderly way.

Where the Operating System Sits

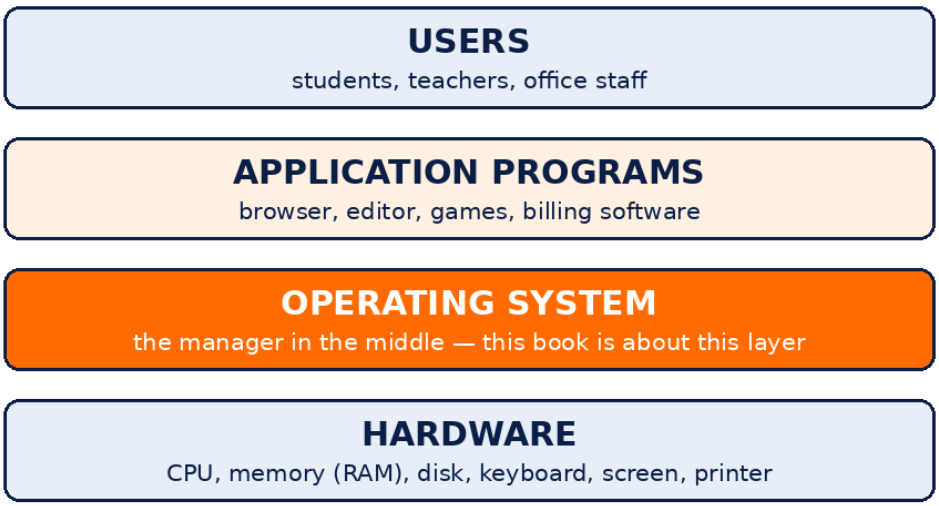


Figure 1.1 - The operating system is the layer between users, programs and hardware.

What the example teaches: Without the middle layer, every application would have to know how to drive every disk, printer and memory chip by itself — and two programs could fight over the same resource at the same moment. The OS exists so that hardware is used **conveniently** (easy for the user) and **efficiently** (nothing wasted). These are its two goals, and almost every topic in this book is one of these two goals in action:

- **Convenience** — the user should be able to run programs, save files and print without knowing anything about the machine's internals.

- **Efficiency** — expensive hardware (CPU, RAM) should rarely sit idle; if one program is waiting for the keyboard, the CPU should be doing someone else's work.

Practice Set 1.1

- Which layer sits between application programs and the hardware?
- A computer has hardware and application programs but no OS. Name two problems that would appear immediately.
- "An OS is a resource manager." Explain this line using the restaurant example.
- Which goal of the OS is served when the CPU starts another program while the first one waits for a printout — convenience or efficiency? Why?

1.2 How operating systems evolved — a short history

Operating systems were not designed in one day; each generation removed a pain of the previous one.

- **No OS (1945–mid 1950s).** Programs were fed by hand with switches and punch cards; the programmer was also the operator. The machine stood idle while the next program was being prepared.
- **Batch systems (mid 1950s–1960s).** Similar jobs were grouped into a batch and run one after another automatically. The idle time between jobs fell, but a programmer waited hours for output and could not interact with a running job.
- **Multiprogramming (1960s).** Several jobs were kept in memory at once; when one waited for input/output, the CPU switched to another. CPU idle time fell sharply — this is the efficiency goal winning.

- **Time-sharing / multitasking (1960s–70s onwards).** The CPU switches between jobs so quickly (in time slices) that many users feel the machine is theirs alone. Interaction becomes possible — this is the convenience goal winning.
- **Personal computer OS (1980s).** One user, one machine: first command-line systems (MS-DOS), then graphical ones (Windows, macOS) — ease of use became the main aim.
- **Modern systems (1990s–today).** Networking, security, multiple cores, mobile and cloud systems — today's OS (Linux, Windows, Android, iOS) is all the earlier ideas combined, with far more protection and polish.

Practice Set 1.2

- Put these in order of evolution: time-sharing, batch, no OS, multiprogramming.
- Which generation first made interactive computing possible?
- Multiprogramming mainly improved which of the two goals? Give the reason in one line.

1.3 Types of operating systems

The syllabus asks for the different types of operating systems. Each type below grew from one generation of the history above, and each is still in use somewhere today.

1.3.1 Batch operating system

Jobs of similar type are collected into a batch and executed one after another without any interaction with the user during execution; the user submits the job, waits, and collects the output later. Suitable where the

same processing is repeated on large data — old payroll and billing systems are the classic example. Weakness: no interaction, and one job's error or delay affects the queue.

1.3.2 Multiprogrammed operating system

Several programs are kept in main memory at the same time, and whenever the running program waits for input/output, the CPU is switched to another ready program, so the CPU is kept busy almost always. This is the foundation of the efficiency goal; most modern systems are descendants of this idea.

1.3.3 Multitasking and time-sharing operating system

A multitasking OS runs several programs "at the same time" from the user's point of view by switching the CPU among them very fast; a time-sharing system is its multi-user form, where many users on different terminals share one machine, each receiving small time slices in turn, so everyone gets quick responses. Your Windows/Linux desktop is multitasking; a college server used by a whole class through terminals is time-sharing.

1.3.4 Multiprocessing operating system

A system with two or more CPUs (or cores) sharing memory and working on programs truly in parallel — not switching, but genuinely simultaneous execution. Used where speed and reliability matter: servers, workstations, and every modern multi-core PC and phone.

1.3.5 Real-time operating system (RTOS)

An OS built to respond within a strictly fixed, guaranteed time limit; being late is treated as failure. **Hard real-time** systems miss no deadline ever (flight control, pacemaker, industrial safety). **Soft real-time** systems try

their best but an occasional delay is tolerated (video streaming, online games). Ordinary OS like Windows are not real-time systems — they never promise an exact response moment.

1.3.6 Distributed operating system

An OS that runs across several independent computers connected by a network and makes them behave, to the user, like one single system — sharing files, processing and devices across machines. Large web services and clusters work this way.

1.3.7 Embedded operating system

A small, specialised OS built into a device to do one dedicated job — washing machines, microwave ovens, ATM machines, smart TVs, car controllers. It is trimmed down: fast to start, low memory, often real-time in nature.

| Type | One-line identity | Typical example | |---|---|---| | Batch | Jobs in groups, no interaction | Old payroll/billing processing | | Multiprogrammed | Many jobs in memory, CPU never idle | Early mainframe systems | | Multitasking / Time-sharing | Fast switching; many users feel ownership | Windows desktop; college server | | Multiprocessing | Many CPUs, truly parallel | Servers, multi-core PCs | | Real-time | Guaranteed response time | Flight control, pacemaker | | Distributed | Many computers act as one | Clusters, large web services | | Embedded | Built into a device for one job | Washing machine, ATM |

Practice Set 1.3

- An ATM must respond predictably and do one fixed job. Which two types describe its OS, and why?
- "A time-sharing system is a logical extension of multiprogramming." Explain in two lines.

- A hospital monitor must never respond late. Batch or real-time — which, and what kind (hard/soft)?
- Match: (a) payroll of 1970
  (b) college lab server
  (c) smart TV — to batch, time-sharing, embedded.

Common mistakes

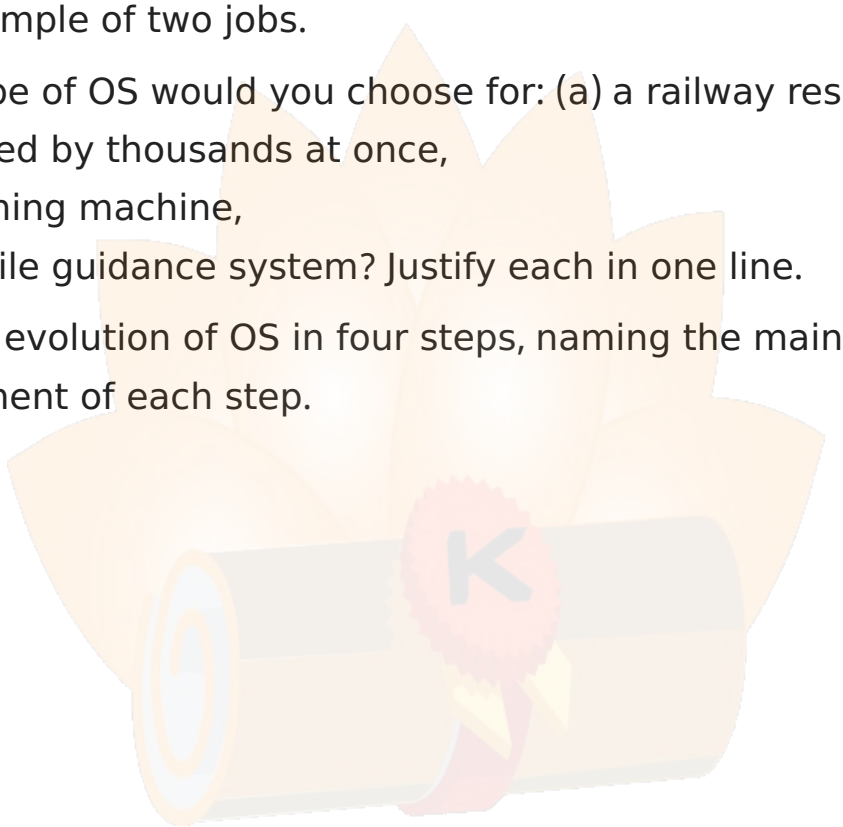
- Calling the OS an application program — it is system software, the layer applications run upon.
- Confusing multitasking with multiprocessing: switching quickly on one CPU is not the same as many CPUs working truly in parallel.
- Saying real-time means "very fast". It means guaranteed on time — a promise about deadlines, not raw speed.
- Listing Windows features when asked the functions of an OS — functions are jobs like process and memory management (Chapter 2), not product features.

Summary

- The OS is the manager layer between users/programs and hardware; its goals are convenience and efficiency.
- History moved: no OS → batch → multiprogramming → time-sharing → personal and modern systems, each step removing an idle-time or interaction pain.
- Types: batch, multiprogrammed, multitasking/time-sharing, multiprocessing, real-time (hard/soft), distributed, embedded — classified by how work is organised and what is guaranteed.

Chapter practice set

- Define an operating system in one complete sentence, naming what it manages and what it provides.
- State the two goals of an OS and give one example situation for each.
- Differentiate between multitasking and multiprocessing (two points).
- Differentiate between batch and time-sharing systems (two points).
- What is a hard real-time system? Give two examples.
- Why did multiprogramming reduce CPU idle time? Explain with a small example of two jobs.
- Which type of OS would you choose for: (a) a railway reservation server used by thousands at once,
  (b) a washing machine,
  (c) a missile guidance system? Justify each in one line.
- Write the evolution of OS in four steps, naming the main improvement of each step.



Chapter 2: Functions and Services, the Kernel, System Calls and System Programs

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 1 — Overview of Operating System)

What you will learn

- The main functions (jobs) of an operating system — the list this whole book expands
- The services an OS provides to programs and to users
- What the kernel is, and the two ways kernels are organised (monolithic and microkernel)
- What a system call is, how one travels, and its main types
- The UNIX/Linux architecture, layer by layer
- What system programs are, with everyday examples

2.1 Functions of an operating system

Chapter 1 defined the OS as a manager. Managing breaks into five main functions — learn these well, because Units 2 to 5 of this course are simply these five functions studied one by one:

- **Process management** — creating, scheduling and ending programs in execution (processes), and letting them share the CPU fairly. (Unit 2)

- **Memory management** — deciding which process gets which part of RAM, and keeping each process's memory separate and protected. (Unit 3)
- **File management** — organising data into files and directories on disks, and controlling who may read or change them. (Unit 5)
- **Device (I/O) management** — controlling printers, disks, keyboards and screens through device drivers, so programs need not know hardware details. (Unit 4)
- **Protection and security** — making sure users and programs can only do what they are allowed to do, and cannot damage the system or each other's work. (Runs through all units)

Around these sit supporting jobs: error detection (watching for hardware or program faults), accounting (recording who used how much CPU and memory), and the user interface (command line or GUI) through which people give instructions.

Practice Set 2.1

- A student opens two programs and both get RAM without disturbing each other. Which function is at work?
- Match: (a) deciding which program runs next
  (b) keeping a file safe from other users
  (c) talking to a printer — to process, protection, device management.
- Why is the user interface not counted among the five main resource functions?

2.2 Services of an operating system

The OS provides services in two directions. **For programs:** program execution (load and run a program), input/output operations (read/write on their behalf, since programs may not touch devices directly), file-system manipulation (create, delete, read, write files), communication

between processes, and error reporting. **For users:** a user interface (CLI or GUI), and system programs (next section) for everyday tasks like copying files and seeing the disk's contents. A useful way to remember: a function is a responsibility of the OS; a service is that responsibility experienced by a program or a user.

Practice Set 2.2

- When your program calls print(), which two services does it use?
- "Error detection" — is it a function, a service, or both? Defend your answer in one line.

2.3 The kernel — the heart of the OS

The **kernel** is the core part of the operating system that stays loaded in memory all the time the computer is on and is the only part allowed to interact directly with the hardware: it performs the central duties — scheduling processes, managing memory, handling interrupts and executing system calls. Everything else in the OS is built around it.

Kernels are organised in two classic ways:

- **Monolithic kernel** — all OS services (process, memory, file, device management) run together inside the kernel in one large program. Fast, because parts communicate directly; but a fault in any part can bring down the whole system, and the code is large to maintain. Linux follows this design.
- **Microkernel** — only the bare minimum (basic scheduling, memory basics, message passing) stays in the kernel; other services run as ordinary programs outside it, talking by messages. More reliable (a service can crash without killing the kernel) and easier to extend, but slower because of the message passing. MINIX is the standard textbook example.

| Basis | Monolithic kernel | Microkernel | |---|---|---| | What runs inside | All OS services | Only minimum core services | | Speed | Faster (direct calls) | Slower (message passing) | | Reliability | One fault can crash all | A service crash need not crash the kernel | | Size / maintenance | Large, harder | Small core, easier to extend | | Example | Linux | MINIX |

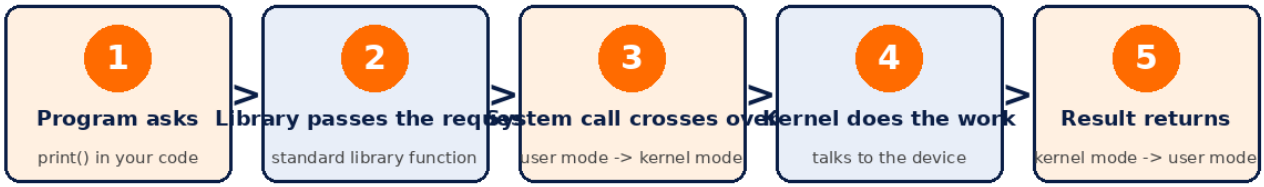
Practice Set 2.3

- Why must the kernel be present in memory all the time?
- A driver crashes and the whole system hangs with it. Which kernel design does that suggest, and why?
- Microkernels are more reliable yet Linux is monolithic — name the property Linux gains in exchange.

2.4 System calls — how programs ask the kernel

A **system call** is the controlled doorway through which a running program requests a service from the kernel — such as reading a file, writing to the screen, creating a process or allocating memory — because programs run in a restricted user mode and are not allowed to touch hardware or protected memory directly; the request switches the CPU to the privileged kernel mode, the kernel checks and performs the work, and returns the result. You met this pattern in the Python book every time a program printed or opened a file — the same doorway serves every language.

How a System Call Works



The kernel is the only part allowed to touch hardware directly.
That is why every request must pass through a system call.

Figure 2.1 - A system call travels from a program, through the library, into the kernel and back.

Main types of system calls (the standard five groups):

- **Process control** — create/terminate a process, load and execute a program, wait for a process (examples in UNIX: fork, exit, wait).
- **File management** — create, delete, open, close, read, write a file (open, read, write, close).
- **Device management** — request/release a device, read/write a device (ioctl, read/write on devices).
- **Information maintenance** — get/set time, date, process and system information (getpid, time).
- **Communication** — create/delete a connection, send/receive messages, shared memory (pipe, shmget).

A program rarely calls these directly; it calls a library function (like printf or print), and the library makes the actual system call. One library call can mean exactly one system call — or several, or none if the answer is already in the library's buffer.

Practice Set 2.4

- Why can't a program in user mode simply read the disk by itself?
- Classify into the five types: (a) creating a new process
  (b) reading a file
  (c) asking the current time
  (d) opening a pipe between two programs.
- A program prints 100 lines but the disk/screen work happens only a few times. Which idea from this section explains it?

2.5 UNIX/Linux architecture

UNIX and its descendant Linux organise the OS in clean layers — the syllabus names this architecture specifically, and Chapter 1's layered picture becomes concrete here:

- **Hardware** (bottom) — CPU, memory, disks, terminals.
- **Kernel** — the core (Section 2.3): process, memory, file and device management, reached only through system calls.
- **System call interface** — the controlled doorway of Section 2.4.
- **Libraries and utilities** — standard library functions, shells and commands, compilers: the layer programs and users actually touch.
- **Users and application programs** (top) — people and the software they run.

Linux follows this architecture with a monolithic kernel (Section 2.3), and adds the **shell** — the command interpreter that takes your typed commands (ls, cp, mkdir — the very commands of the OS Lab's first

experiment) and asks the kernel, through system programs, to carry them out. The lab's Linux commands and this architecture are the same story from two sides.

Practice Set 2.5

- In the UNIX layer picture, which layer may a user program never skip, and why?
- Where does the shell sit — inside the kernel or outside it? What does it do?

2.6 System programs

System programs are the ready-made utility programs supplied with the operating system that use its services to do everyday jobs for the user — file operations (copy, move, delete, search), status information (disk space, running processes), program development help (compilers, loaders, debuggers), text editing, and communication (mail, remote login). They are not the kernel and not user applications: they are the OS's own toolbox, sitting between the two. On Linux, commands like cp, mv, ls, grep and the shell itself are system programs; on Windows, File Explorer, Task Manager and the Command Prompt play the same role.

The difference that matters for the exam: the **kernel** is one always-running core; **system calls** are the doorway into it; **system programs** are ordinary (replaceable) programs standing outside, using that doorway like any other program.

Practice Set 2.6

- Classify: kernel, system call, system program — (a) cp command
  (b) read()
  (c) the scheduler
  (d) a text editor supplied with the OS.

- If a system program is deleted, does the OS stop working? What exactly is lost?

Common mistakes

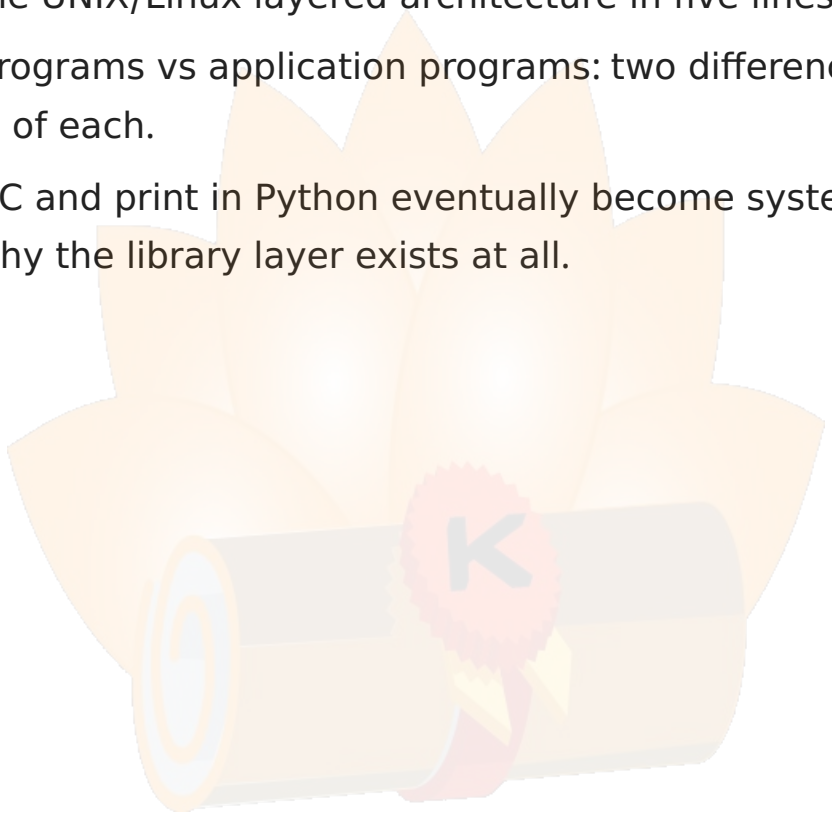
- Mixing the three K-words of this chapter: **kernel** (the core), **system call** (the doorway), **system program** (a utility outside). Keep them separate and half this chapter's marks are safe.
- Saying the user interface is the OS. The GUI/shell is a replaceable outer layer; the OS can run without it (servers do).
- Claiming microkernels are "better" in every way — they trade speed for reliability; exam answers must state the trade-off.
- Writing that programs "call the hardware through system calls". System calls reach the kernel; the kernel reaches the hardware.

Summary

- Five functions: process, memory, file, device management, and protection/security — the map of Units 2–5.
- Services are those functions as experienced by programs (execution, I/O, files, communication, errors) and users (interface, utilities).
- The kernel is the always-resident core; monolithic (all services inside, fast — Linux) vs microkernel (minimum inside, reliable — MINIX).
- A system call is the user-mode → kernel-mode doorway; five types: process control, file, device, information, communication.
- UNIX/Linux: hardware → kernel → system-call interface → libraries/shell/utilities → users; system programs are the OS's replaceable toolbox outside the kernel.

Chapter practice set

- List the five main functions of an OS and name the unit of this course that studies each of the first four.
- Define kernel. Compare monolithic and microkernel designs on four points.
- What is a system call? Draw/describe its five steps and list its five types with one example each.
- A user types `cp a.txt b.txt` on Linux. Tell the whole story: shell, system program, system call, kernel, hardware.
- Explain the UNIX/Linux layered architecture in five lines.
- System programs vs application programs: two differences, two examples of each.
- "printf in C and print in Python eventually become system calls." Explain why the library layer exists at all.



Chapter 3: Process Concepts and Operations on Processes

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 2 — Process Management)

What you will learn

- Program vs process — the difference the whole unit stands on
- The five process states and the diagram every examiner expects
- The Process Control Block (PCB) — what the OS records about each process
- Operations: creating and terminating processes, process trees, and the context switch
- The three schedulers and the queues processes wait in

3.1 What is a process?

A **process** is a program in execution — a program file loaded into memory together with everything it needs to run: its current instruction position, the values of its variables, its stack and the resources (files, memory) the OS has given it. The program on the disk is passive code; the process is that code alive and working.

The difference is easy to feel: one program can be many processes. Open the same browser program in three windows and you have one program but three processes — each with its own memory, its own pages, its own

state, none disturbing the others. In your Python work the same thing happened: the Python interpreter is one program; every script you ran was a separate process of it.

A process in memory has recognisable parts: the **text** (the code), the **data** (global variables), the **heap** (memory allotted while running) and the **stack** (function calls and local data) — plus the CPU registers and the program counter (which instruction is next), which live with the process logically even when the CPU is busy elsewhere.

Practice Set 3.1

- A program is opened five times. How many programs, how many processes?
- Which part of a process grows when functions keep calling functions?
- "The process is the program" — correct or not? Answer in two lines.

3.2 Process states

As a process lives, it moves through states. The five standard states:

- **New** — the process is being created (loaded, given its PCB and resources).
- **Ready** — it has everything it needs except the CPU; it waits in the ready queue for its turn.
- **Running** — its instructions are being executed on the CPU. On one CPU, only one process per core is running at any instant.
- **Waiting (blocked)** — it cannot continue until some event happens, typically I/O completing; even a free CPU cannot help it yet.
- **Terminated** — it has finished; the OS reclaims its resources.

The Process State Diagram

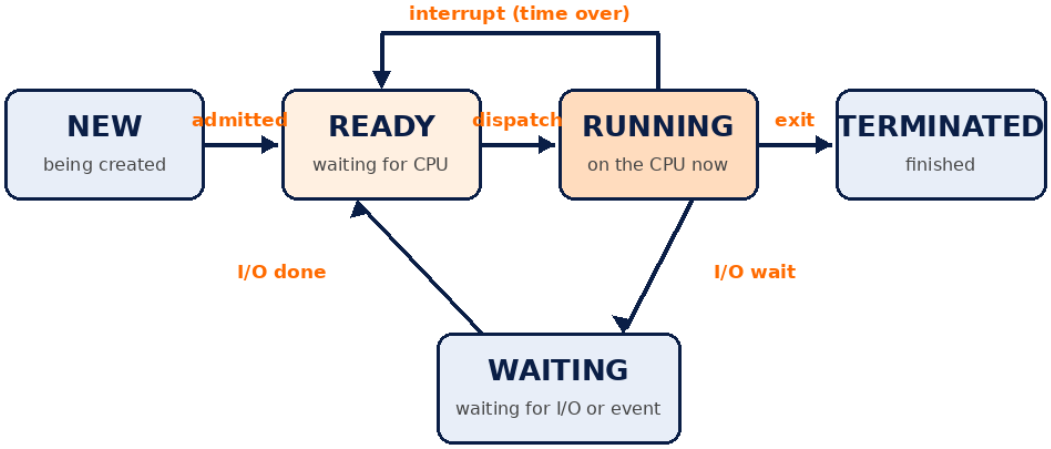


Figure 3.1 - The five process states and the moves between them.

Read the arrows carefully, they are the examinable part: a process is admitted from new to ready; the scheduler dispatches a ready process to running; a running process leaves by exit (terminated), by interrupt (its time slice ended — back to ready), or by an I/O wait (to waiting, returning to ready when the I/O completes). Note the honest detail: I/O completion takes a process to **ready, not directly to running** — it must wait for dispatch like everyone else.

Practice Set 3.2

- A process asked for data from the disk and the disk is busy. Its state, and its next two states in order?
- Can a process move from waiting straight to running? Why not?
- On a 4-core CPU, at most how many processes are in running state at one instant?

3.3 The Process Control Block (PCB)

The OS keeps the record of every process in a data structure called the **Process Control Block (PCB)** — one per process. Its typical contents:

- **Process ID (PID)** — the unique number of the process.

- **Process state** — new/ready/running/waiting/terminated.
- **Program counter** — the address of the next instruction to execute.
- **CPU registers** — the saved register values, needed to resume exactly where it stopped.
- **CPU scheduling information** — priority, queue position.
- **Memory management information** — the memory allotted to the process, page tables and limits.
- **Accounting information** — CPU time used, time limits.
- **I/O status information** — devices and files currently open for it.

Your OS Lab's Task Manager experiment is this section in visible form: the process list with IDs, states and CPU/memory figures is a window onto the system's PCBs.

Practice Set 3.3

- A process is paused mid-work and resumed an hour later at the exact same instruction. Which two PCB fields make that possible?
- Why does the PCB store open-file information at all?

3.4 Operations on processes

Creation. A process (the parent) creates new processes (its children) through a create-process system call — fork() in UNIX/Linux, which makes an almost-copy of the parent, usually followed by a call that loads a fresh program into the child. Children can create their own children, forming a **process tree**: on Linux the tree is rooted at the first process (init/systemd, PID 1), created by the kernel at boot; every process you start hangs somewhere under it. When you double-click an app, the shell/desktop process is the parent and the app is the child.

Termination. A process ends by an exit system call when its work is done (normal termination), or is ended by the OS or another authorised process when it errors, exceeds limits or is no longer needed (abnormal termination). On termination the OS reclaims its memory and resources and informs the parent; children may be adopted by the root process or terminated along with the parent, depending on the system.

Context switch. When the CPU moves from one process to another, the OS performs a **context switch**: save the current process's state (registers, program counter) into its PCB, and load the next process's saved state from its PCB. It takes time during which no useful work happens — pure overhead — which is why switches happen thousands of times a second only when they are very fast, and why the time slice (Chapter 4) must not be too small.

Practice Set 3.4

- Draw the process tree for: system start → login shell → an editor and a compiler started from the shell.
- A parent is terminated. State the two possible fates of its children mentioned in this section.
- Why is context-switch time called overhead? What balances making it rarer?

3.5 Schedulers and queues (a first look)

Processes wait in **queues**: the ready queue (ready processes, in memory), and device queues (processes waiting for a particular device). Three schedulers move processes between places:

- **Long-term scheduler (job scheduler)** — chooses which new jobs are admitted into memory (from the job pool on disk) and controls the degree of multiprogramming (how many processes are in memory). Runs rarely — seconds or minutes apart.

- **Short-term scheduler (CPU scheduler)** — chooses which ready process gets the CPU next. Runs very often — after almost every event; Chapter 4 is entirely about its algorithms.
- **Medium-term scheduler** — temporarily removes a process from memory to disk (swapping out) when memory is crowded, and brings it back later (this connects to swapping in Chapter 8).

Practice Set 3.5

- Which scheduler decides the degree of multiprogramming? Which must be the fastest, and why?
- A process is swapped out to disk to free memory. Which scheduler did that, and where does it wait?

Common mistakes

- Writing program and process as synonyms. The program is passive, on disk; the process is the program alive — one program, many processes.
- Drawing I/O completion as waiting → running. It is waiting → ready.
- Calling the context switch "free". It is pure overhead; its cost is one reason scheduling choices matter.
- Mixing the long-term and short-term schedulers: admission into memory (rare) vs CPU handover (constant).

Summary

- A process is a program in execution, with its own memory (text, data, heap, stack) and state.
- Five states: new, ready, running, waiting, terminated — moves happen by admission, dispatch, interrupt, I/O wait/completion and exit.

- The PCB records everything about a process: PID, state, program counter, registers, scheduling, memory, accounting, I/O.
- Parents create children (process tree rooted at PID 1); termination reclaims resources; a context switch saves one PCB and loads another, at overhead cost.
- Long-term (admission), short-term (CPU), medium-term (swapping) schedulers move processes between memory, ready queue and devices.

Chapter practice set

- Define a process. How is it different from a program? Give the three-window example in your own words.
- Draw the process state diagram and explain every arrow.
- List the contents of a PCB and state the role of the program counter within it.
- What is a context switch? Describe its two halves and why it is overhead.
- Explain process creation and the process tree. What is special about PID 1?
- Compare the three schedulers: what each decides, and how often each runs.
- In the OS Lab you will observe Task Manager — name four PCB fields you can see reflected there.

Chapter 4: CPU Scheduling

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 2 — Process Management)

What you will learn

- What CPU scheduling decides, and the measures (criteria) used to judge it
- Preemptive vs non-preemptive scheduling
- FCFS, SJF, SRTF, Round Robin and Priority — each with a fully worked Gantt-chart example
- The convoy effect, starvation, and aging

4.1 What scheduling is, and how it is judged

Whenever more than one process is ready, someone must choose which one gets the CPU next. That choosing — done by the short-term scheduler (Chapter 3) — is **CPU scheduling**. Different choices give very different experiences, so scheduling algorithms are judged by these criteria:

- **CPU utilisation** — percentage of time the CPU is busy doing useful work. Higher is better.
- **Throughput** — number of processes completed per unit time. Higher is better.
- **Turnaround time (TAT)** — from a process's arrival to its completion: TAT = Completion time – Arrival time. Lower is better.
- **Waiting time (WT)** — total time the process spent waiting in the ready queue: WT = TAT – Burst time. Lower is better.

- **Response time** — from arrival to the first time the process gets the CPU (matters for interactive use).

Two working styles: in **non-preemptive** scheduling, once a process gets the CPU it keeps it until it finishes or blocks for I/O — simple, but one long process can hold everyone up. In **preemptive** scheduling, the OS may take the CPU away (time slice over, or a more urgent process arrives) — fairer and more responsive, at the cost of more context switches. Every example below states which style it uses.

The method for all worked examples: draw the Gantt chart (a bar showing which process runs when), read off each completion time, then apply TAT = CT − AT and WT = TAT − BT. Burst time (BT) is the CPU time a process needs; arrival time (AT) is when it became ready.

Practice Set 4.1

- A process arrives at 2, completes at 11, and needs 6 units of CPU. Its TAT and WT?
- Which criterion does an interactive user feel most directly? Which does a server owner watch?

4.2 FCFS — First Come, First Served

Processes run in arrival order, non-preemptively — a fair queue, like a ticket counter.

Worked Example 1. All three arrive at time 0, in the order P1, P2, P3. Burst times: P1 = 24, P2 = 3, P3 = 3.

Gantt chart: | P1 (0–24) | P2 (24–27) | P3 (27–30) |

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

Average WT = (0 + 24 + 27) / 3 = **17**. Average TAT = (24 + 27 + 30) / 3 = **27**.

The convoy effect: P2 and P3 — tiny 3-unit jobs — waited 24 and 27 units because one long job arrived first, exactly like one slow truck holding up a line of cars. Had the order been P2, P3, P1, the average waiting time would drop to 3 (check it). FCFS is simple and starvation-free, but average waiting time depends heavily on arrival order and is usually poor. Your OS Lab simulates FCFS first for this reason: it is the baseline every other algorithm must beat.

Practice Set 4.2

- Verify the claim above: order P2, P3, P1 gives average WT = 3.
- Processes (AT, BT): A(0, 5), B(2, 2), C(3, 1). FCFS average WT? (Also mind the arrivals.)

4.3 SJF and SRTF — shortest job first

SJF (Shortest Job First): the ready process with the smallest burst time runs next. Non-preemptive version: the choice is made when the CPU becomes free.

Worked Example 1 (continued). Same three processes (24, 3, 3), SJF order is P2, P3, P1.

Gantt chart: | P2 (0–3) | P3 (3–6) | P1 (6–30) |

WT: P2 = 0, P3 = 3, P1 = 6. Average WT = **3** — against FCFS's 17 on identical work. SJF is provably the best possible average waiting time for a given set of processes, which is why it is the standard other algorithms are compared with. Its weakness is practical: the OS cannot know a process's exact future burst time; it can only predict it from past behaviour. A second weakness: long processes may keep losing to newly arriving short ones — starvation, treated in Section 4.5.

SRTF (Shortest Remaining Time First): the preemptive form of SJF — whenever a process arrives, if its burst is shorter than the remaining time of the running process, it takes over.

Worked Example 2. P1 (AT 0, BT 8), P2 (AT 1, BT 4), P3 (AT 2, BT 9), P4 (AT 3, BT 5), scheduled with SRTF.

Step by step: at 0 only P1 exists, it runs; at 1 P2 arrives with 4 < P1's remaining 7, so P2 takes over; P3 (9) and P4 (5) arrive during P2's run but neither beats P2's remaining time, so P2 finishes at 5. Remaining: P4 (5) is shortest, runs 5–10; then P1 (7), 10–17; then P3, 17–26.

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 WT = (9 + 0 + 15 + 2) / 4 = **6.5**. Average TAT = (17 + 4 + 24 + 7) / 4 = **13**.

For comparison, non-preemptive SJF on the same set gives average WT 7.75 (verify in the practice set) — preemption bought a small improvement here, at the price of extra context switches.

Practice Set 4.3

- Verify the SJF (non-preemptive) figures for Worked Example 2: order, completion times, average WT 7.75.
- Why can't a real OS implement perfect SJF? What does it do instead (one line)?

4.4 Round Robin — taking turns

Round Robin (RR): each ready process gets the CPU for one **time quantum** (time slice), then goes to the back of the queue if unfinished; preemptive by design. It is FCFS plus a timer — built for time-sharing, where response time matters more than finishing any one job early.

Worked Example 3. Same three processes as Example 1 (P1 = 24, P2 = 3, P3 = 3, all at 0), quantum = 4.

The queue story: P1 runs 0–4 (20 left, goes to the back); P2 runs 4–7 and finishes (it needed only 3); P3 runs 7–10 and finishes; P1 then runs alone to 30.

Gantt chart: | P1 (0–4) | P2 (4–7) | P3 (7–10) | P1 (10–30) |

WT: P1 = 30 − 24 = 6; P2 = 7 − 3 = 4; P3 = 10 − 3 = 7. Average WT = 17/3 ≈ **5.67**. Average TAT = (30 + 7 + 10)/3 ≈ **15.67**.

Read the trade-off in the numbers: RR's average waiting (5.67) is between SJF (3) and FCFS (17), but every process got the CPU within 4 units — response time is where RR wins, which is why interactive systems use it. The quantum is the tuning knob: too large and RR degenerates into FCFS; too small and context-switch overhead eats the machine (Chapter 3). A common rule of thumb: pick the quantum so that most processes finish within one slice.

Practice Set 4.4

- Redo Example 3 with quantum = 2 and compare the average waiting time and the number of context switches.
- With quantum = 30, what does RR become for this example? State the general rule.

4.5 Priority scheduling, starvation and aging

Each process carries a **priority** number; the CPU goes to the highest-priority ready process (in this book, a smaller number means higher priority — state your convention in the exam, both exist). It can be preemptive (a newly arrived higher-priority process takes over) or non-preemptive.

Worked Example 4 (non-preemptive). Same set as Example 2, with priorities: P1 (AT 0, BT 8, prio 3), P2 (AT 1, BT 4, prio 2), P3 (AT 2, BT 9, prio 4), P4 (AT 3, BT 5, prio 1).

At 0 only P1 is present, so it runs to completion at 8 (non-preemptive — priority cannot interrupt). Then the ready priorities are P4 (1), P2 (2), P3 (4): P4 runs 8–13, P2 13–17, P3 17–26.

Gantt chart: | P1 (0–8) | P4 (8–13) | P2 (13–17) | P3 (17–26) |

WT: P1 = 0, P2 = 17 − 1 − 4 = 12, P3 = 26 − 2 − 9 = 15, P4 = 13 − 3 − 5 = 5. Average WT = 32/4 = **8**.

Starvation: a low-priority process can wait indefinitely if higher-priority work keeps arriving — its wait grows without bound while the system stays busy and "correct". The standard cure is **aging**: a waiting process's priority gradually improves with time, so eventually it outranks newcomers and runs. SJF has the same disease in another form (long jobs starve) and aging is the cure there too. Priority scheduling is how real systems express "this work matters more" — system processes over background jobs, your taps over a background sync.

Practice Set 4.5

- Make Example 4 preemptive and recompute: who runs at time 1, and what are the new completion times?
- A server runs priority scheduling without aging and a backup job has not run in a week, though the CPU was never idle. Explain, and fix it in one line.

4.6 The five algorithms side by side

Algorithm	Rule	Preemptive?	Strength	Weakness
FCFS	Arrival order	No	Simple, no starvation	Convoy effect, poor average WT
SJF	Shortest burst next	No (SRTF is)	Best average WT (provable)	Needs future knowledge; long jobs starve
SRTF	Shortest remaining time	Yes	Best average WT with arrivals	Many context switches; starvation
Round Robin	Fixed time

quantum in turn | Yes | Best response time, fair | Overhead if quantum small; average WT mid | | Priority | Highest priority first | Either | Urgent work first | Starvation without aging |

Common mistakes

- Computing WT directly from the Gantt chart positions instead of WT = CT − AT − BT; with arrivals present, the chart method misleads — always use the formulas.
- In SRTF, comparing the newcomer's burst with the running process's original burst instead of its remaining time.
- In Round Robin, forgetting that a process finishing inside its quantum leaves immediately (P2 finished at 7, not 8, in Example 3), or forgetting to send an unfinished process to the back of the queue.
- Starting a priority example without stating whether a small or large number means high priority.

Summary

- Scheduling criteria: CPU utilisation, throughput, turnaround time, waiting time, response time; TAT = CT − AT, WT = TAT − BT.
- FCFS: arrival order; simple, starvation-free, suffers the convoy effect. Example: average WT 17.
- SJF: shortest burst first; provably best average WT (3 in the example) but needs predicted bursts; SRTF is its preemptive form (average WT 6.5 in Example 2).
- Round Robin: quantum-based turns; fairness and response time, quantum size is the trade-off (average WT ≈ 5.67 in Example 3).
- Priority: highest priority first; starvation cured by aging (Example 4 average WT 8).

Chapter practice set

- Processes (AT, BT): A(0, 6), B(1, 3), C(2, 2), D(4, 4). Compute average WT and TAT under (a) FCFS
  (b) SJF non-preemptive
  (c) SRTF.
- Redo the previous question with Round Robin, quantum 2.
- Draw the Gantt chart for Example 2 under non-preemptive priority with priorities P1=2, P2=4, P3=1, P4=3 and compute the average WT.
- Define the five scheduling criteria in one line each.
- Explain the convoy effect with the truck-and-cars picture, and name the algorithm that suffers it.
- "SJF is optimal yet unimplementable in perfect form." Resolve the contradiction in four lines.
- What is aging? Which two algorithms in this chapter need it, and why?

  (Answers to the numerical practice questions are worked in the Question Bank, Unit 2.)

Chapter 5: Inter-Process Communication, Synchronisation and the Critical Section

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 2 — Process Management)

What you will learn

- Why processes need to communicate (IPC) and the two models: shared memory and message passing
- The race condition — how correct programs produce wrong results together
- The critical section and the three requirements of any correct solution
- Semaphores: the standard tool for guarding critical sections
- The classic problems: producer–consumer, readers–writers, dining philosophers

5.1 Inter-process communication (IPC)

Processes are independent by default — separate memory, separate worlds (Chapter 3). But real work is often cooperating work: one process produces data another consumes, several processes share one printer queue, a browser's tabs share one download manager. **Inter-process communication (IPC)** is the set of mechanisms the OS provides for

processes to exchange data and coordinate their actions. Processes communicate for four classic reasons: **information sharing** (several processes need the same data), **computation speed-up** (a job split across CPUs or machines), **modularity** (a system built from cooperating parts), and **convenience** (working on several related tasks at once).

The OS offers two fundamental models:

- **Shared memory** — the OS maps one region of memory into several processes' address spaces; they communicate by reading and writing that region, like colleagues writing on one shared whiteboard. Fast (after setup, no kernel involvement per message), but the processes themselves must take care not to collide — which is exactly the problem of the next section.
- **Message passing** — processes send and receive messages through the OS (send / receive operations; a pipe is the everyday example from the Linux shell, where one command's output flows into the next). Slower per message (every exchange crosses the kernel), but safe by construction: no shared region exists to fight over.

| Basis | Shared memory | Message passing | |---|---|---| | How data moves | Read/write a common region | send() and receive() via the OS | | Speed | Faster | Slower (kernel involved each time) | | Safety | Processes must synchronise themselves | No shared data, fewer conflicts | | Best for | Large or frequent data exchange | Small messages, separate machines |

Practice Set 5.1

- A video editor's modules exchange whole frames many times a second; two office PCs exchange a daily report. Pick a model for each and justify in one line.
- Why is a pipe between two shell commands classed as message passing, not shared memory?

5.2 Why synchronisation: the race condition

Cooperating processes that share data can corrupt it — not through bad code, but through bad timing. Consider a shared counter that two processes both increment once, starting at 4. "Increment" is really three steps: read the value, add 1, write it back. Now interleave:

Process A reads 4 → Process B reads 4 → A writes 5 → B writes 5. Two increments happened; the counter should be 6; it is 5. Run the interleaving differently and it can end at 4 or 5 or 6 — the result depends on the race.

A **race condition** is exactly this: a situation where several processes access and change the same data at the same time and the final result depends on the order in which the accesses happen — so it can be wrong. The joint bank account shows the stakes: two family members withdraw from two ATMs at the same moment; both machines read the same balance, both subtract, one withdrawal is never recorded. The cure is **process synchronisation**: a discipline that lets processes use shared data one at a time in the dangerous region, so outcomes stop depending on timing.

Practice Set 5.2

- Starting the counter at 10, construct an interleaving where three increments (one each by A, B, C) end with the counter at 11.
- "Each program was individually correct, so the system must be correct." Refute in two lines.

5.3 The critical section and its three requirements

The part of a process's code that accesses shared data — the counter, the account, the shared buffer — is its **critical section**. The whole problem reduces to one rule: at most one process may be inside its

critical section (for the same shared data) at any moment. A correct solution must satisfy three requirements, and exam answers should name all three:

- **Mutual exclusion** — if one process is in its critical section, no other may enter its own (for the same data).
- **Progress** — if the critical section is free and some processes want to enter, the choice of who enters next cannot be postponed forever, and processes outside their critical sections cannot block the choice.
- **Bounded waiting** — after a process asks to enter, there is a limit on how many times others may enter before it; nobody waits forever (no starvation).

Picture a single-lane bridge: mutual exclusion = one vehicle on the bridge at a time (same direction counting as one user here); progress = if the bridge is empty and cars wait, someone is waved through promptly; bounded waiting = no driver watches car after car pass forever while he waits.

Practice Set 5.3

- A solution guarantees mutual exclusion but lets one fast process re-enter endlessly while another waits. Which requirement fails?
- In the bridge picture, which requirement is violated if the attendant goes for tea while cars queue at both ends and the bridge stands empty?

5.4 Semaphores — the standard guard

A **semaphore** is an integer variable maintained by the OS that processes use to coordinate access to shared resources: it is accessed only through two indivisible (atomic) operations — **wait** (also called P or acquire), which decrements the value and blocks the caller if the value would go negative, and **signal** (also called V or release), which

increments the value and wakes a waiting process if any. Because wait and signal are atomic — no other process can slip into the middle of one — a semaphore can enforce the one-at-a-time rule that raw shared memory cannot.

Two kinds:

- **Binary semaphore (mutex)** — value only 0 or 1. Wrapped around a critical section, it is exactly the bridge gate: wait before entering, signal on leaving; a second process finds 0 and waits.
- **Counting semaphore** — value counts identical resources: initialised to 3, it lets three processes use the three printers/database connections at once and blocks the fourth until someone signals.

A note on waiting style: the simplest semaphore makes a blocked process busy-wait (spin, checking again and again) — acceptable for very short waits on multi-core systems, wasteful otherwise; real implementations put the process to sleep in a queue and wake it on signal, at the cost of a context switch.

Practice Set 5.4

- A counting semaphore starts at 2. Five processes call wait in turn. Draw the value after each call and say which processes are blocked.
- Why must wait be atomic — what race returns if wait itself can be interrupted half-done?

5.5 The classic synchronisation problems

Three textbook problems package every difficulty of this chapter; the OS Lab's Dining Philosophers experiment is the third one.

Producer–Consumer (bounded buffer). A producer process puts items into a fixed-size buffer; a consumer takes them out. Rules: the producer must wait when the buffer is full, the consumer must wait when it is empty, and both must never touch the buffer at the same instant. Solved

with three semaphores: mutex (1, guard the buffer), empty (starts at buffer size, counts free slots), full (starts at 0, counts items). Your printer queue and every download manager live inside this problem.

Readers–Writers. Many processes read a shared document freely together, but a writer needs the document alone. The tension: letting readers stream in can starve the writer; locking for the writer wastes readers' parallelism. Fairness between the two groups is the whole problem.

Dining Philosophers. Five philosophers sit round a table, each needing two chopsticks (one on each side, shared with neighbours) to eat. If everyone picks up the left stick first, everyone holds one stick and waits forever for the second — the picture of **deadlock**, which takes over Chapter 6. The lab asks you to implement the table and the fix: rules like "pick up both sticks together, or wait" and "let at most four sit at once" break the circle.

Practice Set 5.5

- In producer–consumer, what breaks if the full semaphore is forgotten — who runs when they shouldn't?
- In dining philosophers, which single rule change destroys the all-hold-one-stick situation in every case?

Common mistakes

- Calling the race condition a "coding bug". Each process's code is fine; the fault is unsynchronised timing on shared data.
- Listing only mutual exclusion as the requirement. Progress and bounded waiting carry equal marks.
- Writing wait/signal as ordinary increment/decrement. Their atomicity is the entire point — state it.

- Confusing a counting semaphore with a lock for one resource; its value counts how many identical resources remain.

Summary

- IPC lets cooperating processes exchange data: shared memory (fast, self-guarded) or message passing (kernel-mediated, safe).
- A race condition: shared data changed concurrently, result depends on order — wrong answers from correct code.
- The critical section is the code touching shared data; correct solutions give mutual exclusion, progress and bounded waiting.
- A semaphore is an OS integer used only via atomic wait/signal; binary = mutex for one-at-a-time, counting = pool of identical resources.
- Producer–consumer (mutex/empty/full), readers–writers (fairness), dining philosophers (deadlock preview) are the classic problems.

Chapter practice set

- Define IPC and give its two models with one advantage of each.
- A shared seat counter starts at 200; two booking processes each sell one seat concurrently and the counter ends at 199 instead of 198. Name the fault and find the exact interleaving.
- State the three requirements of a critical-section solution and explain each in the single-lane-bridge picture.
- What is a semaphore? Define wait and signal precisely, and distinguish binary from counting with a resource-pool example.
- Describe the producer–consumer problem and the role of its three semaphores.
- Explain the dining philosophers situation and one rule that prevents the deadlock.

- Shared memory is faster, so why does anyone use message passing? (Two reasons.)

Chapter 6: Deadlock — Conditions, Prevention, Avoidance and the Banker's Algorithm

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 2 — Process Management)

What you will learn

- What a deadlock is, precisely, with the traffic and chopsticks pictures
- The four conditions that must hold together for deadlock
- The four strategies: ignore, prevent, avoid, detect-and-recover
- Deadlock avoidance and the Banker's algorithm, with a complete worked example

6.1 What is deadlock?

A **deadlock** is a situation in which a set of processes is permanently blocked because each process in the set holds a resource that another process in the set is waiting for, so none of them can proceed, release anything, or finish — the blockage can only be broken from outside. Chapter 5's dinner ended here: every philosopher holds one chopstick and waits for a second that will never be free. A city shows the same shape: cars fill a square junction from all four sides, each waiting for the car ahead — which is waiting in the same circle.

The setting is a system model: the system has resource types (CPU cycles, memory space, files, printers, locks), each with one or more identical instances. A process must **request → use → release** each resource. Deadlock is a disease of that cycle when resources cannot be shared or taken back.

Practice Set 6.1

- Two processes, one printer: P1 holds the printer and waits for the scanner; P2 holds the scanner and waits for the printer. State, in the definition's own words, why this is a deadlock.
- Can a single process deadlock with itself on one single-instance resource? Think, then check with Section 6.2.

6.2 The four necessary conditions

A deadlock can occur only if **all four** conditions hold at the same time. Remove any one and deadlock becomes impossible — this single fact organises the whole chapter.

- **Mutual exclusion** — a resource can be held by only one process at a time (a printer, a lock).
- **Hold and wait** — a process holding some resources is allowed to wait for more.
- **No preemption** — a resource cannot be snatched away; only the holder can release it, voluntarily.
- **Circular wait** — a circle of processes exists, each waiting for a resource held by the next in the circle (P1 → P2 → P3 → P1).

Practice Set 6.2

- In the dining philosophers, point to each of the four conditions in the scene.

- A resource can be used by many processes at once (a read-only file). Which condition fails, and what follows for deadlock?

6.3 The four strategies for handling deadlock

1. **Ignore it (the ostrich approach).** Most general-purpose systems, including Windows and Linux, simply do not handle deadlock in general: it is rare, prevention is costly, and a hang can be cleared by killing a process or restarting. Honest engineering, chosen knowingly.

2. **Prevention — break one condition by design.** Guarantee that at least one of the four can never hold:

Condition broken	How	Cost
Mutual exclusion	Share the resource where possible (spool output to a queue instead of locking the printer)	Impossible for truly non-sharable resources
Hold and wait	A process must request all its resources at once, before starting	Resources sit idle-held; the full list must be known in advance
No preemption	If a waiting process holds resources, take them away (works for CPU and memory via saving state)	Impossible for printers and mid-write files
Circular wait	Number all resources; processes may request only in increasing order	The order must suit everyone; low-numbered resources over-demand

3. **Avoidance — decide safely at every request.** The OS knows each process's maximum possible demand in advance; before granting any request it checks that the resulting state is safe — the Banker's algorithm (Section 6.4). Nothing is forbidden by design; unsafe grants are simply refused/delayed.

4. **Detection and recovery.** Let deadlocks happen, periodically check for one (looking for cycles/waits that cannot resolve), and recover by terminating one or more processes in the circle or preempting their resources, choosing victims with the least lost work.

Practice Set 6.3

- A college lab prints through a single spool queue — which strategy and which condition is at work?
- "Request everything in advance" fails most clearly for which kind of program? Give one example.

6.4 Avoidance and the Banker's algorithm

A system state is **safe** if the OS can still sequence all processes so that each can obtain its maximum possible demand and finish — using currently free resources plus those released by processes that finish before it. Safe is not the same as deadlock-free-looking: a state can have no deadlock today and still be **unsafe**, one unlucky request away from it. Avoidance keeps the system always inside safe states.

The **Banker's algorithm** (Dijkstra) performs that check. Its name is the logic: a banker should never hand out cash in a way that makes it impossible to satisfy every customer's promised credit line. The OS keeps, for n processes and m resource types:

- **Available** — instances of each resource currently free.
- **Max** — each process's maximum total demand (declared in advance).
- **Allocation** — each process currently holds.
- **Need** = Max − Allocation — what each process may still ask for.

Safety check: Work = Available; Finish[i] = false for all. Repeatedly find a process with Finish = false and Need ≤ Work (it could be fully satisfied now); pretend it finishes: Work = Work + its Allocation, Finish = true. If

every process can finish this way, the state is **safe** and the order found is a safe sequence. If the loop stalls with someone unfinished, the state is **unsafe**.

Worked Example. 5 processes, 3 resource types (A = 10, B = 5, C = 7 instances in total).

Process	Allocation (A B C)	Max (A B C)	Need = Max − Alloc
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

Allocated totals: A = 7, B = 2, C = 5, so **Available = (10−7, 5−2, 7−5) = (3, 3, 2)**.

Run the safety check. Work = (3, 3, 2):

- P1's Need (1, 2, 2) ≤ Work → P1 can finish; Work = (3,3,2) + (2,0,0) = **(5, 3, 2)**.
- P3's Need (0, 1, 1) ≤ Work → finishes; Work = (5,3,2) + (2,1,1) = **(7, 4, 3)**.
- P4's Need (4, 3, 1) ≤ Work → finishes; Work = (7,4,3) + (0,0,2) = **(7, 4, 5)**.
- P0's Need (7, 4, 3) ≤ Work → finishes; Work = (7,4,5) + (0,1,0) = **(7, 5, 5)**.
- P2's Need (6, 0, 0) ≤ Work → finishes.

All five finish: **the state is safe**, one safe sequence being **P1, P3, P4, P0, P2**. (Other safe sequences exist; any one proves safety.)

Granting a request: when a process requests resources, the OS checks (i) request ≤ its Need (else it exceeded its claim — an error), (ii) request ≤ Available (else it must wait — resources simply are not free), then pretends to grant and runs the safety check on the imagined state: grant only if still safe. Example: P1 requests (1, 0, 2). Both checks pass (Need (1,2,2); Available (3,3,2)). Pretend: Available becomes (2, 3, 0), P1's Allocation (3, 0, 2), Need (0, 2, 0). Safety from Work (2, 3, 0): P1 (0,2,0) fits → Work (5, 3, 2); then P3, P4, P0, P2 as before → safe, so the request is **granted**. Your OS Lab implements exactly this: avoidance in one experiment, prevention in its companion.

Why real systems rarely use Banker's: Max demands are seldom known in advance, processes and requests change constantly, and the check runs at every request. It lives on as the clearest statement of the avoidance idea — and a board favourite for that reason.

Practice Set 6.4

- In the worked example, can P0 be the first to finish in any safe sequence? Show why or why not.
- P4 now requests (3, 3, 0) from the original state. Check (i) and (ii); then decide by safety whether it can be granted. (Hint: pretend carefully — Available would fall to (0, 0, 2).)

Common mistakes

- Listing fewer than four conditions, or presenting any one alone as "the cause". All four together are necessary.
- Calling an unsafe state "deadlocked". Unsafe means deadlock is possible later; the processes are still running now.
- In Banker's, comparing Need against total resources instead of current Work/Available.
- Forgetting the second half of the safety step: after a process "finishes", its Allocation returns to Work.

Summary

- Deadlock: a set of processes each holding what another in the set awaits — permanently blocked without outside help.
- Four necessary conditions, all together: mutual exclusion, hold and wait, no preemption, circular wait.
- Strategies: ignore (most real systems), prevent (break a condition by design), avoid (grant only into safe states), detect and recover (kill/preempt to break the circle).

- A safe state admits a sequence in which every process can reach its maximum and finish; Banker's checks safety with Available, Max, Allocation and Need = Max − Allocation.
- Worked example: the 5-process state is safe (sequence P1, P3, P4, P0, P2), and P1's request (1, 0, 2) is grantable.

Chapter practice set

- Define deadlock and state the four necessary conditions, one line each.
- For each condition, give one prevention method and its price.
- What is a safe state? Distinguish safe, unsafe, and deadlocked with a two-line scenario for the unsafe case.
- Using the chapter's worked example, verify whether the sequence P3, P1, P4, P2, P0 is also a safe sequence.
- From the same state, P0 requests (0, 2, 0). Decide whether the request can be granted, showing both checks and the safety run.
- Why do Windows and Linux not run Banker's algorithm in practice? (Three reasons.)
- Detection vs avoidance: which needs Max declared in advance, and which pays with lost work?

Chapter 7: Memory Management and Allocation — First Fit, Best Fit, Worst Fit

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 3 — Memory Management)

What you will learn

- What memory management must achieve, and the idea of logical vs physical addresses
- Contiguous allocation: fixed partitions and variable partitions
- The placement strategies — First Fit, Best Fit, Worst Fit — with one fully worked comparison
- Holes, and why allocation choice matters over time

7.1 The memory manager's job

RAM is fast, expensive and shared. The memory manager — the OS part studied in this unit — must: give each process the memory it needs, keep processes from touching each other's memory (protection), keep track of which parts are used and which are free, and make the best use of every byte, because a process that cannot get memory cannot run at all.

One idea clears up everything else in this unit. Programs work with **logical addresses** — the addresses as the program sees them, starting from 0 in its own private world. The RAM chips hold data at **physical addresses**. Hardware (the MMU, memory management unit) translates logical → physical on every access, using base/limit values the OS sets

per process. That is how two processes can both use "address 1000" and mean different real places — and how a process physically cannot reach another's memory: its translations will never point there. In this chapter processes are placed in memory as single blocks; Chapters 8–9 divide them into pages and segments.

Practice Set 7.1

- A process's base register holds 5000 and it accesses logical address 300. What physical address is touched? What if it tries logical 99999 while its limit is 8000?
- Why does address translation make memory protection almost free?

7.2 Contiguous allocation: two partition styles

In **contiguous allocation** each process occupies one unbroken stretch of memory. Two classical styles:

- **Fixed partitions** — memory is cut in advance into partitions of set sizes (say 100 KB, 250 KB, 500 KB, 1 MB); each process takes one partition that fits it. Simple to manage, but a 60 KB process in the 250 KB partition wastes 190 KB inside its own partition — **internal fragmentation**, which Chapter 9 treats fully.
- **Variable (dynamic) partitions** — no fixed cuts: each process is given exactly the space it asks for, carved out of the free region. Precise, but as processes load and leave, the free memory breaks into scattered free blocks called **holes**, and the wasted space now lies between processes — external fragmentation (also Chapter 9). The rest of this chapter lives in this style: when a new process arrives and several holes could hold it, which hole should it get?

Practice Set 7.2

- A 90 KB process is placed in a fixed 128 KB partition. How much memory is wasted and what is that waste called?

- Why can variable partitions not suffer that same waste — and what waste appears instead?

7.3 The three placement strategies (worked example)

Given the list of holes, the OS picks one by a strategy:

- **First Fit** — take the first hole (searching from the start, or from where the last search stopped — that variant is Next Fit) that is big enough. Fast: stop at the first success.
- **Best Fit** — search all holes and take the smallest one that fits, leaving the least leftover.
- **Worst Fit** — take the largest hole, leaving the largest possible leftover, in the hope the leftover stays useful.

Worked Example. Free blocks (holes), in memory order: **100, 500, 200, 300, 600 KB**. Requests arrive in this order: **212, 417, 112, 426 KB**. Apply each strategy independently from the same starting picture.

First Fit:

- 212 → the 500 block (500 is the first that fits). Leftover hole 288.
- 417 → the 600 block. Leftover 183.
- 112 → scanning from the top: 100 too small; 288 fits → leftover 176.
- 426 → remaining holes are 100, 176, 200, 300, 183 — none ≥ 426. **Cannot be allocated**, despite 959 KB total free.

Best Fit:

- 212 → smallest fitting hole is 300. Leftover 88.
- 417 → smallest fitting is 500. Leftover 83.
- 112 → smallest fitting is 200. Leftover 88.
- 426 → smallest fitting is 600. Leftover 174.
- **All four allocated.**

Worst Fit:

- 212 → largest hole, 600. Leftover 388.
- 417 → largest now 500. Leftover 83.
- 112 → largest now 388. Leftover 276.
- 426 → holes are 100, 83, 200, 300, 276 — none fits. **Cannot be allocated.**

The lesson in the results. All three strategies faced identical work. Best Fit — by preserving the large 600 hole for the large 426 request — fitted everyone; First Fit and Worst Fit both stranded the biggest request amid plenty of total free memory. In practice, studies of real workloads find First Fit and Best Fit broadly comparable (First Fit usually faster, needing no full search; Best Fit's tiny leftovers become useless slivers), and Worst Fit generally the weakest — its "big leftover" promise fails because it destroys the large holes that large requests will need. This example is also the OS Lab's fit-strategies experiment, step for step.

Practice Set 7.3

- Holes: 150, 350, 80, 420; request 300. Which hole does each strategy choose, and what leftover remains in each case?
- A single request of 250 follows in the same setup (after the 300 was placed by Best Fit). Can it be placed? What if First Fit had placed the 300?

7.4 Living with holes

Allocation is half the job; the cycle completes when processes leave. A freed block adjacent to another hole merges with it (**coalescing**) so the free list stays in the largest usable pieces. When no single hole fits a request but the total free memory would, the OS may **compact** — slide all processes together to fuse the holes into one big block (Chapter 9) —

at the cost of copying and a pause. The strategies above are really answers to a long-game question: place this process so that tomorrow's requests still stand a chance.

Practice Set 7.4

- Memory order: [P1][hole 50][P2][hole 70][P3]. P2 now leaves. Describe the free blocks before and after coalescing.
- Why is compaction a last resort rather than an hourly routine?

Common mistakes

- Writing the three fits as definitions without the consequence: on the same data they allocate different holes and can succeed or fail differently — the worked example is the expected answer shape.
- Claiming Best Fit is always best. Its leftovers shrink to slivers; in long-run studies it does not beat First Fit overall.
- Forgetting that a failed placement (426 in First/Worst Fit) is not "memory full": total free space exceeded the request — the space was in the wrong pieces. That observation opens Chapter 9.

Summary

- Memory management: allot space, protect processes from each other, track free/used regions; logical addresses are translated to physical ones by the MMU using per-process base/limit.
- Fixed partitions waste space inside partitions; variable partitions carve exact sizes but leave holes between processes.
- First Fit takes the first adequate hole (fast); Best Fit the smallest adequate (least leftover); Worst Fit the largest (largest leftover, usually weakest).
- Worked example: on holes 100/500/200/300/600 with requests 212/417/112/426, only Best Fit placed all four.

- Freed adjacent holes coalesce; compaction can fuse scattered holes at a copying cost.

Chapter practice set

- Define logical and physical address. A process with base 12000 and limit 4000 accesses logical 2500 and then logical 4100 — what happens in each case?
- Compare fixed and variable partitioning on partition sizing, waste type, and management effort.
- Holes in order: 200, 600, 250, 450, 100. Requests: 380, 120, 290, 500. Show placement (or failure) under each of the three strategies.
- State the three fit strategies in one line each, naming the hole each selects.
- Why is Worst Fit usually the weakest strategy, despite its reasonable-sounding idea?
- What is coalescing, and what problem does it postpone but not solve?

Chapter 8: Swapping, Paging and Segmentation

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 3 — Memory Management)

What you will learn

- Swapping: moving whole processes between RAM and disk
- Paging: dividing memory into fixed-size pages and frames, and the page table
- Address translation in paging, with a fully worked numeric example
- Segmentation: dividing a program by logic, and how it differs from paging

8.1 Swapping

Swapping is the temporary removal of a whole process from main memory to a reserved area of disk (the swap space), and its later return to memory to continue execution: the medium-term scheduler (Chapter 3) swaps a process out when memory is overcrowded or the process will wait a long time, and swaps it in when memory frees and its turn approaches. Swapping lets the total memory of all resident processes exceed the physical RAM — more programs stay "in the system" than could ever fit at once. Its price is time: a process image may be megabytes, and disk transfers are slow beside RAM, so heavy swapping is a symptom of overload (Chapter 9 names its worst form). Modern

systems swap at the finer grain of pages (Section 8.2), but the whole-process idea is where the mechanism — and the swap space of Unit 4 — begins.

Practice Set 8.1

- A 200 MB process is swapped out to a disk moving 100 MB/s. Roughly how long does the transfer alone take, and what does that suggest about swap frequency?
- Which scheduler performs swapping, and what condition triggers it?

8.2 Paging — memory in fixed-size pieces

Chapter 7's processes needed one unbroken block each, and holes were the price. **Paging** removes the requirement: physical memory is divided into fixed-size blocks called **frames**, and each process's logical memory is divided into blocks of the same size called **pages**; a page may occupy any free frame, so a process's pages can lie scattered across RAM while the program still sees one continuous address space. Page sizes are powers of two (typically 4 KB) chosen by the hardware.

Every logical address splits into two parts: the **page number (p)** — which page of the process — and the **page offset (d)** — the position inside that page. The OS keeps one **page table** per process: page number → frame number (or a mark saying the page is not in memory, which Chapter 9 turns into page faults). Translation per access: look up p in the page table to get frame f; the physical address is f combined with the unchanged offset d.

Address Translation in Paging

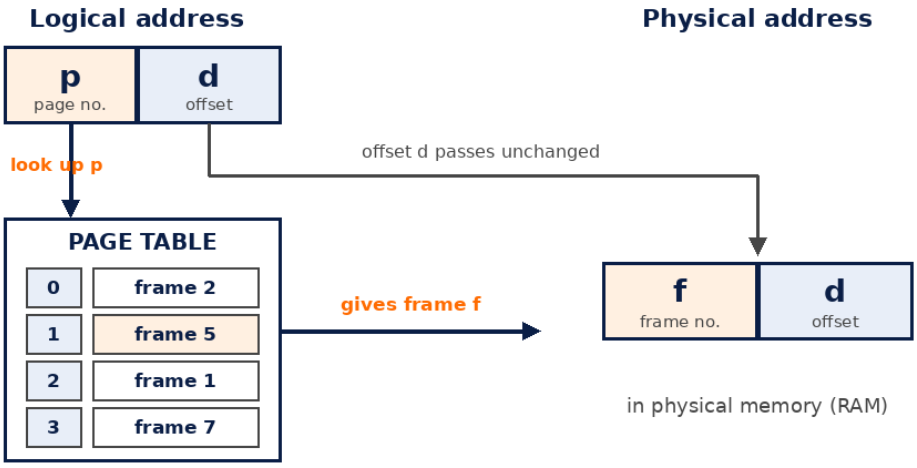


Figure 8.1 - Paging translation: the page number becomes a frame number; the offset passes unchanged.

Worked Example. Page size = 100 bytes (a teaching size; real systems use powers of two). A process has 16 pages. Its page table shows page 7 → frame 4. Translate logical address **753**.

- Page number p = 753 ÷ 100 = **7**; offset d = 753 mod 100 = **53**.
- Page table: page 7 lives in frame 4.
- Physical address = frame × page size + offset = 4 × 100 + 53 = **453**.

Two consequences examiners love. (1) There is **no external fragmentation** — any free frame serves any page, so holes cannot form; only the last page of a process may be partly empty, a small internal waste of at most one page per process. (2) The page table itself costs memory and time: a 32-bit address with 4 KB pages has 12 offset bits and a 20-bit page number — up to 2^{20} (about a million) entries per

process. Real systems answer with small fast caches of recent translations (the TLB) and multi-level page tables; the concept is unchanged.

Practice Set 8.2

- Page size 100, page table: page 2 → frame 9. Translate logical address 260, step by step.
- A machine has 64 frames of 100 bytes and a process of 550 bytes. How many pages does it need, how much internal waste is there, and can its pages sit in frames 3, 7, 9, 20, 21, 40?

8.3 Segmentation — memory divided by logic

A program is not one uniform block in the programmer's mind: it is code, global data, a stack, a heap, library parts — pieces of different sizes with different meanings and different protection needs. **Segmentation** is the memory scheme that matches that view: the logical address space is divided into **segments**, each a logical unit of its own length, named by a **segment number (s)** with an **offset (d)** inside it. The OS keeps a **segment table**: for each segment, its **base** (starting physical address) and its **limit** (length). Translation: the MMU checks d < limit (else a fault — the access ran past the segment's end) and computes physical = base + d.

Segments can be shared (two processes point a segment at the same code — shared libraries work this way) and protected individually (code read-only, data read-write), because the boundaries are the program's own. The cost: segments are variable-sized blocks in physical memory, so segmentation alone brings back Chapter 7's holes and external fragmentation — the disease paging cured. Modern designs therefore combine the two (paged segmentation: each segment is itself paged), keeping logical divisions without the scattering problem in physical memory.

| Basis | Paging | Segmentation | |---|---|---| | Division | Fixed-size pages, invisible to the programmer | Variable-size logical units (code, data, stack) | | Address parts | Page number + offset | Segment number + offset | | Table entry holds | Frame number | Base + limit | | External fragmentation | None | Yes (variable block sizes return) | | Internal fragmentation | Possible in the last page | None | | Sharing/protection | Awkward (page = arbitrary cut) | Natural (segment = meaningful unit) |

Practice Set 8.3

- Segment table: segment 1 has base 3000, limit 400. Translate (s=1, d=250), then (s=1, d=450) — what happens in the second case and why is that a feature?
- Which scheme makes sharing one library between two programs natural, and why?

Common mistakes

- Saying paging "has no fragmentation". It has no external fragmentation; the last page wastes space internally.
- Translating with the offset changed. The offset never changes in paging — only the page number becomes a frame number.
- In segmentation, forgetting the limit check: base + d without d < limit is both wrong and the loss of protection.
- Confusing swapping with paging: swapping moves whole processes (or behaves coarsely); paging works page by page and is the normal modern mechanism.

Summary

- Swapping moves whole processes between RAM and swap space under the medium-term scheduler, letting resident processes exceed RAM in total.

- Paging: fixed-size pages into any free frames; the page table maps page → frame; translation keeps the offset unchanged; worked example: logical 753 → physical 453.
- Paging kills external fragmentation, keeps small internal waste in the last page, and needs page tables (helped by the TLB).
- Segmentation divides by logic (segment number + offset; table holds base + limit with a limit check) — natural sharing/protection, but external fragmentation returns; combined designs page the segments.

Chapter practice set

- Define swapping. Where does the swapped process go, and who brings it back?
- With page size 200 and page 5 → frame 11, translate logical address 1130 fully.
- A 32-bit system uses 8 KB pages. How many bits is the offset, and how many page-table entries can one process have?
- Why does paging need no compaction, ever?
- Draw the segmentation translation for (s=2, d=100) with segment 2: base 5000, limit 800.
- Compare paging and segmentation on four points in table form.
- "Paging divides the program without asking it; segmentation asks the program how it is divided." Explain in three lines.

Chapter 9: Page Faults, Page Replacement, Fragmentation and Virtual Memory

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 3 — Memory Management)

What you will learn

- What a page fault is and the six steps that handle one
- The three replacement algorithms — FIFO, LRU, Optimal — on one worked reference string
- Belady's anomaly: when more memory gives more faults
- Internal vs external fragmentation, settled once and for all
- Virtual memory and thrashing

9.1 The page fault

With paging (Chapter 8), a process's pages need not all be in RAM: the page table marks absent pages invalid. When a process touches an invalid page, the hardware raises a **page fault** — not an error, but a request: "fetch this page now". The OS handles it in six steps:

- The reference traps to the OS; the process is suspended (it moves to waiting — disk I/O is slow).
- The OS checks the reference is legal (within the process's address space); if not, the process is terminated — this is the true error case.

- A free frame is found — or a victim page is chosen for replacement (Section 9.2) and, if the victim was modified, written back to disk first.
- The needed page is read from disk into the frame.
- The page table is updated: page → frame, marked valid.
- The instruction that faulted is restarted, and the process continues as if nothing happened — except that the whole detour took milliseconds, thousands of times longer than a normal access. That cost is why the number of faults matters and the next section exists.

Practice Set 9.1

- Why is the faulting process put into waiting state rather than kept running?
- If step 2 finds the reference illegal, what happens, and why is that not called a page fault success story?

9.2 Page replacement algorithms (worked example)

When no frame is free, someone must leave. The choice is the **page replacement algorithm**; it is judged by counting faults on a **reference string** (the sequence of page numbers a process touches). One string, three algorithms, 3 frames — compute all three with us.

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3

FIFO — evict the page that has been in memory longest: 7(F), 0(F), 1(F) fill the frames. 2 evicts 7 (F). 0 hit. 3 evicts 0 (F). 0 evicts 1 (F). 4 evicts 2 (F). 2 evicts 3 (F). 3 evicts 0 (F). 0 evicts 4 (F). 3, 2, 3 all hit. **FIFO faults = 10.**

LRU (Least Recently Used) — evict the page untouched for the longest time: 7, 0, 1 fault in. 2 evicts 7 (least recent) (F). 0 hit. 3 evicts 1 (F). 0 hit. 4 evicts 2 (F). 2 evicts 3 (F). 3 evicts 0 (F). 0 evicts 4 (F). 3, 2, 3 hit. **LRU faults = 9.** LRU reads the past as a prediction: a page unused for long is likely cold. It needs hardware help (counters or a reference stack) and is the classic practical choice.

Optimal (OPT) — evict the page whose next use is farthest in the future (or never): 7, 0, 1 fault in. 2 evicts 7 (never used again) (F). 0 hit. 3 evicts 1 (never again) (F). 0 hit. 4 evicts 0 (its next use is the farthest of the three) (F). 2, 3 hit. 0 evicts 4 (never again) (F). 3, 2, 3 hit. **Optimal faults = 7.** Optimal is unbeatable — and unimplementable, because it needs the future. Its role is as a benchmark: run OPT on a recorded string and see how close FIFO/LRU came.

| Algorithm | Faults (3 frames) | Idea | Practical? | |---|---|---|---| | FIFO | 10 | Oldest page out | Simple; suffers Belady's anomaly | | LRU | 9 | Least recently used out | Yes, with hardware support | | Optimal | 7 | Farthest future use out | No — benchmark only |

Belady's anomaly. Intuition says more frames ⇒ fewer faults. FIFO can break it: on the string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, FIFO gives **9 faults with 3 frames but 10 faults with 4 frames** (verify both in the practice set — the counting is the learning). LRU and Optimal are immune: they belong to the stack algorithms, whose resident set with n frames is always a subset of that with n+1 frames. This is also your OS Lab's page-replacement experiment — FIFO, LRU and Optimal on exactly such strings.

Practice Set 9.2

- Verify Belady's anomaly: count FIFO faults for 1,2,3,4,1,2,5,1,2,3,4,5 with 3 frames, then with 4.
- On the chapter string, which single eviction decision first makes LRU better than FIFO? (Track both until they differ.)
- Why can Optimal never be the algorithm of a real OS, and why do textbooks keep it?

9.3 Fragmentation — the two wastes, settled

Fragmentation is memory wasted because free space exists in pieces the allocation scheme cannot use. It comes in exactly two forms, and this chapter closes the topic opened in Chapter 7:

- **Internal fragmentation** — waste inside an allocated block: the block is bigger than the request. Fixed partitions (a 60 KB process in a 250 KB partition) and paging's last page (a process filling 3.2 pages gets 4) both suffer it. The space is lost because it belongs to someone — free-looking, but unusable by others.
- **External fragmentation** — waste between allocated blocks: total free memory is enough, but it lies in scattered holes, no single one fitting the request — Chapter 7's 426 KB request dying amid 959 KB free. Variable partitions and segmentation suffer it; paging does not (any frame fits any page). Remedies: **compaction** (slide processes together to merge holes — costly copying, pauses, and impossible while I/O pins memory) or switching to paging, which removes the requirement that caused it.

Practice Set 9.3

- Classify, with the deciding feature: (a) 18 KB unused at the end of a process's last page
  (b) 400 KB free as five 80 KB holes when a 200 KB process arrives.
- A system reports 30% of memory free yet starts no new process. Give both possible fragmentation explanations.

9.4 Virtual memory — the concept that ties it together

Virtual memory is the memory scheme in which each process runs in its own virtual address space that may be larger than the physical RAM, because only the pages currently needed are kept in memory and the

rest live on disk, brought in automatically on page faults — the technique of Sections 9.1–9.2 used as a system design, called **demand paging**. To the process, memory looks whole and enormous; to the hardware, RAM holds a working subset.

Its benefits: programs larger than RAM can run; more processes fit at once (each holds only part of itself), raising multiprogramming and CPU use; less I/O is needed at start (load on demand, not all upfront). Its price: page faults are expensive, so VM works only because programs show locality — in any short window they touch a small, stable set of pages (loops, hot functions, nearby data).

Abuse the idea and the system **thrashes**: when the resident processes' combined working sets exceed RAM, every few instructions fault, the disk queue saturates, CPU utilisation collapses (the scheduler sees idle processes and — fatally — admits more, worsening it). The cures are to keep the working sets resident (enough frames per process) or reduce the degree of multiprogramming: thrashing is too much virtual memory ambition for the physical memory bought. The curve is worth remembering: CPU use rises with more processes, peaks, then falls off a cliff at thrashing.

Practice Set 9.4

- A laptop with 4 GB RAM runs programs whose total size is 11 GB. Which concept makes this unremarkable, and which assumption about programs does it rest on?
- CPU utilisation is 8% while the disk runs at 100%. Diagnosis, and two cures.

Common mistakes

- Calling a page fault an error or a crash. It is the normal engine of virtual memory; only illegal references are errors.

- In FIFO, evicting the least recently loaded is right — but in LRU evicting by load order instead of use order. Hits refresh LRU's judgement; FIFO ignores hits entirely.
- Mixing the fragmentations: inside an allocated block (internal) vs between blocks in the free space (external).
- Defining virtual memory as "using disk as RAM" without the mechanism: demand paging on faults, per-process virtual address space.

Summary

- A page fault fetches an absent page in six steps (trap → legality check → frame/victim → read in → update table → restart the instruction).
- On the worked string with 3 frames: FIFO 10 faults, LRU 9, Optimal 7; FIFO alone suffers Belady's anomaly (9 faults at 3 frames, 10 at 4).
- Internal fragmentation wastes space inside allocated blocks; external fragmentation wastes it in scattered holes between them — paging eliminates the external kind.
- Virtual memory = demand paging over a per-process virtual space, resting on locality; pushed too far it becomes thrashing.

Chapter practice set

- List the six steps of page-fault handling in order.
- Reference string 1, 2, 3, 2, 1, 4, 3 with 3 frames: count faults under FIFO, LRU and Optimal.
- Define Belady's anomaly and name one algorithm family that cannot suffer it.
- Distinguish internal and external fragmentation with one example each, and state which allocation schemes suffer which.

- What is virtual memory? State its benefits and the locality assumption underneath.
- Explain thrashing with the CPU-utilisation curve in words, and give two remedies.
- Why is a victim page that was modified more expensive to evict than a clean one?

Chapter 10: Mass Storage Structure, Disk Structure and Disk Attachment

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 4 — I/O System)

What you will learn

- The place of secondary (mass) storage in the system, and its main kinds
- The physical anatomy of a hard disk: platters, tracks, sectors, cylinders
- Logical blocks and how a disk's capacity is computed (worked example)
- The three parts of disk access time (worked example)
- How disks attach: host-attached storage, NAS and SAN

10.1 Mass storage — an overview

Main memory is fast but small, costly and volatile: everything in RAM dies with the power. The system therefore needs **mass storage** (secondary storage) — non-volatile, large and cheap-per-byte media that hold the OS itself, all programs and all files permanently, and also serve as the home of swapped-out pages (Chapters 8–9) and the destination of every save. The main kinds, in speed order:

- **Solid-state drives (SSD/NVMe)** — flash memory, no moving parts; the fastest secondary storage, now standard for system disks.

- **Hard disk drives (HDD)** — magnetic spinning platters; slower but cheapest per terabyte, still the bulk-storage workhorse, and the device whose mechanics drive this unit's scheduling topic (Chapter 11).
- **Magnetic tape** — very cheap per byte, sequential access only; survives in backups and archives where speed hardly matters.

Whatever the medium, the OS mostly sees one abstraction: an array of **logical blocks** of fixed size (commonly 512 bytes or 4 KB) numbered from 0 — the file system (Unit 5) is built on that flat array, and the device hides its real geometry behind it.

Practice Set 10.1

- Why can virtual memory not live in RAM alone — which property of mass storage does it borrow?
- A student's laptop boots from SSD but archives video on an HDD and keeps backups on tape. Justify each choice in one line.

10.2 Disk structure: platters, tracks, sectors, cylinders

A hard disk packs its data on spinning circular **platters** coated with magnetic material, each surface served by a read/write **head** on a moving arm; all arms move together. Each surface is divided into concentric rings called **tracks**; each track is divided into **sectors** — the smallest physical unit of storage (traditionally 512 bytes). The set of tracks at the same arm position across all surfaces forms a **cylinder**. A location on disk is thus identified by (cylinder, head/surface, sector) — though modern disks present the flat logical-block numbering of Section 10.1 and translate internally.

Worked Example (capacity). A disk has 4 platters, both sides used = 8 surfaces; 1024 tracks per surface; 64 sectors per track; 512 bytes per sector. Total sectors = 8 × 1024 × 64 = 524,288. Capacity = 524,288 × 512 bytes = 268,435,456 bytes = **256 MB** (in binary units).

Access time. Reaching data on an HDD costs three components:

- **Seek time** — moving the arm to the right track (the slowest, most variable part).
- **Rotational latency** — waiting for the platter to spin the wanted sector under the head; on average, half a rotation. At 6000 RPM one rotation takes 60/6000 s = 10 ms, so average latency = **5 ms**.
- **Transfer time** — actually reading/writing the sectors once they pass under the head; small for small transfers.

Disk scheduling (Chapter 11) attacks the first component: the order in which requests are served decides how far the arm travels.

Practice Set 10.2

- A disk: 6 surfaces, 2048 tracks/surface, 128 sectors/track, 512 B/sector. Compute its capacity in GB (binary).
- A 15000 RPM disk: what is the average rotational latency? Why "average" and not exact?

10.3 Disk attachment — how storage connects

- **Host-attached storage** — disks connected directly to a computer through its I/O buses (historically IDE/PATA, today SATA for HDD/SSD and NVMe over PCIe for fast SSDs), managed by a host **controller** that the OS drives through its device driver. Your lab PCs' internal disks are this. Small scale, simple, storage belongs to that one machine.

- **Network-attached storage (NAS)** — a storage appliance placed on the network that serves files over the network to many clients (the shared office drive); clients see folders, not raw blocks.
- **Storage-area network (SAN)** — a dedicated high-speed network of block storage that several servers attach as if the disks were local; servers see raw logical blocks and build their own file systems. Used in data centres at institutional scale.

The examining distinction: NAS exports files over the ordinary network; SAN exports blocks over a dedicated storage network; host-attached exports nothing — it is wired to one host.

Practice Set 10.3

- A data centre gives 40 servers one shared pool of disks that each server formats itself. NAS or SAN — and what clue decided it?
- Why does the OS need a different driver story for a SATA disk and a NAS box?

10.4 Formatting: from raw disk to usable storage

A raw disk becomes usable in three steps: **physical (low-level) formatting** divides the platter into sectors with their headers and error-correction codes (done by the manufacturer); **partitioning** cuts the disk into regions the OS treats as separate disks (C:, D:...); **logical formatting** writes a file system into a partition (Unit 5's subject) so files can exist. Only then does the flat logical-block array become the files and folders of everyday use.

Practice Set 10.4

- A brand-new disk shows as "unallocated" in the OS installer. Which formatting steps remain, and in what order?

Common mistakes

- Defining a sector as the logical block. A sector is the physical unit; the OS's logical block may span or match sectors — the device presents blocks, the platter holds sectors.
- Forgetting rotational latency scales with RPM only: seek time is about arm movement, not spin speed.
- Saying NAS and SAN are both "storage on the network" without the file-vs-block distinction — that distinction is the answer.

Summary

- Mass storage is non-volatile, large and cheap per byte: SSD (fastest), HDD (bulk), tape (backup); the OS sees an array of logical blocks.
- HDD anatomy: platters/surfaces, heads on one arm, tracks, sectors, cylinders. Capacity = surfaces × tracks × sectors × bytes (worked: 256 MB).
- Access time = seek + rotational latency (average = half a rotation; 5 ms at 6000 RPM) + transfer.
- Attachment: host-attached (SATA/NVMe to one machine), NAS (files over the LAN), SAN (blocks over a dedicated network).
- Usable storage needs physical formatting, partitioning, then logical formatting with a file system.

Chapter practice set

- Name the three main mass-storage kinds, one strength of each, and the property all share against RAM.
- Compute the capacity of a disk with 10 surfaces, 4096 tracks per surface, 256 sectors per track, 512 bytes per sector (answer in GB, binary).

- Define seek time, rotational latency and transfer time. Which does disk scheduling reduce, and how (one line, pointing to Chapter 11)?
- A sector is 512 B and a logical block is 4 KB. How many sectors in one block, and why would an OS prefer blocks larger than sectors?
- Distinguish host-attached storage, NAS and SAN with the export test (what crosses the wire).
- List the three formatting steps in order and state what each creates.

Chapter 11: Disk Scheduling, Swap Space Management and RAID

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 4 — I/O System)

What you will learn

- Why the order of disk requests decides performance
- FCFS, SSTF, SCAN and C-SCAN on one worked queue, with total head movement calculated
- Swap space: where it lives and how the OS manages it
- RAID: using several disks as one — levels 0, 1, 5, 6 and 10

11.1 Disk scheduling — the problem and the measure

Requests for disk blocks arrive from many processes at once and queue up. On a hard disk the dominant cost is the **seek**: moving the arm (Chapter 10). The order in which queued requests are served therefore decides the total arm travel — and with it, the disk's effective speed.

Disk scheduling is the OS's reordering of the pending request queue; it is judged by **total head movement** (in cylinders/tracks crossed) on a given queue, lower being better. Fairness matters too: an algorithm that minimises travel but can starve a far-out request indefinitely is not acceptable either. SSDs have no arm, so this topic is an HDD topic — one more reason HDD mechanics filled Chapter 10.

The worked queue for the whole section: requests for cylinders **98, 183, 37, 122, 14, 124, 65, 67**, in that arrival order; the head starts at cylinder **53**; the disk has cylinders **0–199**. (This is the OS Lab's disk-scheduling experiment, on paper.)

FCFS — serve in arrival order

Service order: 53 → 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67. Movements: 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = **640 cylinders**. Fair — nobody ever jumps the queue — but the arm zigzags across the disk; this is the baseline to beat.

SSTF — Shortest Seek Time First

Always serve the pending request nearest the current head position. From 53: 65 (12), 67 (2), then 37 (30), 14 (23), then across to 98 (84), 122 (24), 124 (2), 183 (59). Total = **236 cylinders**. Far better travel — but a steady stream of middle requests can starve requests at the extremes, exactly the disease of SJF (Chapter 4) in disk clothing.

SCAN — the elevator

The arm sweeps in one direction serving everything on its path, continues to the disk's end, reverses, and sweeps back — like a lift serving floors. Assume the head is moving toward larger cylinder numbers: Order: 65, 67, 98, 122, 124, 183, (end 199), then back: 37, 14. Movements: 12 + 2 + 31 + 24 + 2 + 59 + 16 (to 199) + 162 (down to 37) + 23 = **331 cylinders**. No starvation: every request is reached within one full sweep. Its relative **LOOK** turns around at the last request instead of travelling to the physical end.

C-SCAN — circular SCAN

Serve in one direction only; at the end, jump back to the beginning and sweep again (the return jump counts as movement here; some texts exclude it — state your convention). Order: 65, 67, 98, 122, 124, 183, (199),

jump to 0, then 14, 37. Movements: 146 (up to 199) + 199 (the jump) + 14 + 23 = **382 cylinders**. C-SCAN's gain is not total travel but uniformity: requests are treated as a circle, so waiting time is more evenly spread than SCAN's (requests just behind the arm do not wait a full double sweep). **C-LOOK** is the same idea with the jump taken from the last request on one side to the first on the other.

| Algorithm | Total head movement | Starvation? | Idea | |---|---|---|---| | FCFS | 640 | No | Arrival order | | SSTF | 236 | Yes (extremes) | Nearest request first | | SCAN | 331 | No | Elevator sweep both ways | | C-SCAN | 382 | No | One-way sweep, circular |

Practice Set 11.1

- Queue 82, 170, 43, 140, 24, 16, 190 with head at 50 (disk 0–199): compute FCFS total movement, then SSTF's service order.
- Why does SSTF's starvation worry a real OS more than its travel saving pleases it?

11.2 Swap space management

Chapters 8–9 kept sending pages and processes to disk: the **swap space** is where they live — a reserved region of disk used purely as an extension of memory. It is managed by the OS specially, not through the ordinary file system, for two reasons: speed (swap traffic must bypass file-system overhead like directories and allocation; the OS keeps a simple map of swap slots instead) and purpose (its contents are temporary by definition — nothing in swap needs naming, sharing or backup). Two placements exist: a **dedicated swap partition** (a raw disk region, the classic Linux way — fastest, fixed size) and a **swap file** inside a normal file system (flexible to create/resize, slightly slower). How much to reserve is a judgement: too little and heavy load runs out of memory

entirely; too much wastes disk — modern guidance scales with RAM size and workload, and systems using hibernation need swap at least as large as RAM because the whole memory image is written there.

Practice Set 11.2

- Why does the OS not simply store swapped pages as ordinary files?
- A system never hibernates but thrashes nightly. Is its problem swap size? Defend in two lines (recall Chapter 9).

11.3 RAID — many disks acting as one

RAID (Redundant Array of Independent/Inexpensive Disks) combines several physical disks into one logical unit to gain what a single disk cannot give: higher **performance** (splitting data across disks — striping — so parts are read in parallel), higher **reliability** (storing redundant information so a disk's failure loses nothing), or both. The standard levels:

| Level | Technique | Disks (min) | Survives a disk failure? | Space efficiency | Point of it | |---|---|---|---|---|---| | RAID 0 | Striping only | 2 | **No** — failure loses everything | 100% | Pure speed; scratch/video work | | RAID 1 | Mirroring (every disk copied) | 2 | Yes (a mirror remains) | 50% | Simple reliability; OS disks | | RAID 5 | Striping + one distributed parity | 3 | Yes (any one disk) | (n−1)/n | Balanced: servers' classic | | RAID 6 | Striping + two parities | 4 | Yes (any two disks) | (n−2)/n | Safety as disks grow huge | | RAID 10 | Mirrored pairs, striped (1+0) | 4 | Yes (one per pair) | 50% | Speed and safety; databases |

Parity is the arithmetic trick behind 5 and 6: for each stripe of data blocks across disks, one extra block stores their XOR (a checksum); if any one disk dies, its blocks are recomputed from the survivors plus parity — the array keeps working, degraded, and rebuilds onto a replacement disk. RAID 2, 3 and 4 are historical variants (bit-level striping, dedicated parity disk) that the syllabus's "types of RAID" expects you to name, while 0, 1,

5, 6, 10 are the levels in real use. One warning every administrator repeats: **RAID is not backup** — it survives disk failure, not deletion, corruption, theft or fire, which take all copies together.

Practice Set 11.3

- A 4-disk RAID 5 array of 2 TB disks: how much usable space, and how many disks may fail at once?
- Same disks in RAID 10: usable space, failure tolerance, and one reason to still choose it over RAID 5.

Common mistakes

- Counting head movement only between served requests in SCAN/C-SCAN and forgetting travel to the end (and C-SCAN's return) — or the reverse, when the question's convention is LOOK. Read which variant is asked.
- Claiming SSTF is "the best" — it is greedy and starves; SCAN family trades a little travel for a guarantee.
- Defining RAID 0 as reliable because it is RAID. Level 0 has zero redundancy — the R in RAID fails at its most famous level.
- Writing "RAID is a backup system". It protects against one failure kind only.

Summary

- Disk scheduling reorders the request queue to cut seek travel on HDDs; judged by total head movement and fairness.
- Worked queue (head 53): FCFS 640, SSTF 236 (starves), SCAN 331, C-SCAN 382 — sweeps trade travel for no starvation and uniform waits.
- Swap space is OS-managed disk reserved for memory overflow — partition or file, bypassing the file system for speed.

- RAID arrays stripe for speed and mirror/parity for survival: 0 speed only, 1 mirror, 5 one parity, 6 two parities, 10 mirrored stripes; RAID is not backup.

Chapter practice set

- Queue: 55, 120, 18, 88, 160, 42 with head at 40 (disk 0–199). Compute total head movement for FCFS, SSTF and SCAN (head moving up, to the end).
- For the same queue, give C-LOOK's service order (head moving up) without the totals.
- Explain SSTF starvation with a request pattern that demonstrates it.
- What is swap space? Compare swap partition and swap file on speed and flexibility.
- A 5-disk array, 4 TB each: usable space and simultaneous-failure tolerance under RAID 5 and RAID 6.
- Why is parity able to rebuild a dead disk's data? Answer with the XOR idea in two lines.
- "Our server has RAID 1, so we need no backups." Correct the statement fully.

Chapter 12: File Concept, Access Methods, Directory Structure and Mounting

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 5 — File Management)

What you will learn

- What a file is from the OS's point of view, and its attributes and operations
- The two access methods: sequential and direct
- Directory structures, from single-level to the tree you use daily
- Absolute and relative paths, and what mounting a file system means

12.1 The file concept and its attributes

Disks present raw blocks (Chapter 10); humans think in documents, photos and programs. The bridge is the **file**: a named collection of related information stored on secondary storage as one logical unit, with which users and programs work without knowing anything about blocks and sectors. A file is defined by its creator's content — text, data, source code, an executable, an image — and the OS supports it through a fixed set of bookkeeping, the **file attributes**:

- **Name** — the human-facing identifier (the only attribute people usually see).
- **Identifier** — a unique internal tag (a number) by which the OS knows the file.

- **Type** — text, executable, directory...; hinted by the extension (.txt, .exe, .jpg) or stored in the file.
- **Location** — where on the device its data lives (the pointer Chapter 13 expands).
- **Size** — current length in bytes (and sometimes the maximum allowed).
- **Protection** — who may read, write, execute (Chapter 13).
- **Time, date and owner** — created/modified/accessed stamps and the user who owns it.

The **operations** every OS offers on files: create, write, read, reposition (move the file pointer — the seek of file work), delete, and truncate (keep the file, empty its contents); open and close wrap them — opening loads the file's attributes into an open-file table so repeated operations stay fast, and the per-process pointer tracks how far each open session has read or written.

Practice Set 12.1

- Which attribute does the OS actually use to find a file, if not its name? Trace name → identifier → location in one line.
- Opening the same file in two programs: how many open-file entries, and how do their pointers behave?

12.2 Access methods

Once open, a file's bytes can be consumed in two classic patterns:

- **Sequential access** — information is processed in order, one record/byte after another, from the beginning (or wherever the pointer stands), like reading a book page by page or playing a tape. Reads and writes advance the pointer automatically. This is the natural pattern for logs, media playback, compiling source files and most everyday work — and the only pattern tape allows.

- **Direct (random) access** — any block of the file can be read or written in any order by giving its number, like jumping to a chapter or to a track on a disc. It requires the storage to support it (disks and SSDs do; tape does not) and is the pattern databases live on: record 5,000 first, then record 12, then record 5,000 again — a relative block number times the record size gives the position to seek to.

Real systems support both on the same files (an OS either provides direct access natively or simulates sequential files on top of it), and a third pattern, indexed access, layers an index over direct access — find the index entry, follow its pointer — which is how large databases locate records quickly.

Practice Set 12.2

- A CCTV system appends footage continuously and only ever plays it back from the start. Which access method, and what does the pointer do during a day's recording?
- A student-record database must fetch roll number 418's record instantly among 10,000. Which method, and what calculation finds the position (fixed-size records)?

12.3 Directory structure

Files multiply into thousands; the **directory** is the OS structure that organises them — itself a special file whose entries map names to files. The designs, in historical order:

- **Single-level directory** — one directory for all files on the disk (early systems, tiny devices). Simple, and two problems grow with it: every name must be unique across all users, and finding one file among hundreds is a chore.

- **Two-level directory** — a master directory holds one user directory per user; each user has a private flat space. Name collisions vanish between users, but users cannot organise projects inside their space or share naturally.
- **Tree-structured directory** — directories may contain subdirectories without limit, making the hierarchy you use every day: an absolute **path** from the root (/) names any file uniquely — `/home/student/notes.txt` — and each process also has a **current (working) directory** so shorter **relative paths** work (`../teacher` reaches sideways). Deleting a directory raises a policy question (refuse if non-empty, or delete recursively) that the tree design must answer.

A Tree-Structured Directory

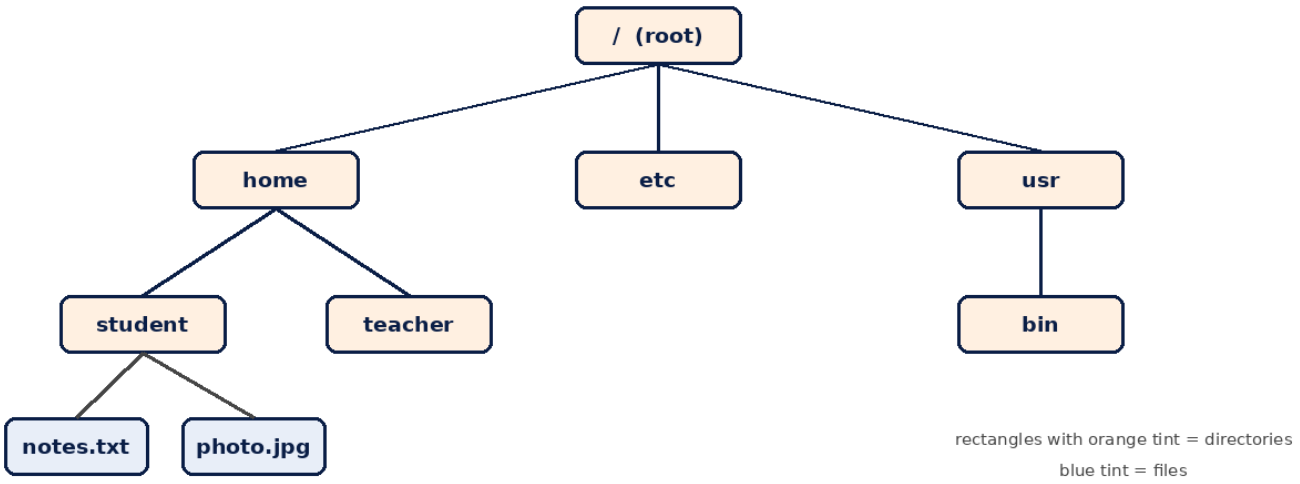


Figure 12.1 - A tree-structured directory: one root, nested directories, files at the leaves.

- **Acyclic-graph directory** — the tree relaxed to let one file/directory appear in two places (a link or shared folder): the same content under two names, changes visible in both. Cycles are still forbidden in the graph (hence "acyclic"), which keeps searches finite; true general graphs with links that can form cycles need care (reference counts and garbage collection when deleting) and are what real systems carefully implement.

Practice Set 12.3

- In a two-level system, can two users both own a file named `report`? In a single-level system? State the deciding feature.
- A file's absolute path is /usr/bin/python3 and the working directory is /usr. Give a relative path, and explain the role of `..` in it if the working directory were /usr/bin instead.

12.4 File system mounting

A file system on a separate disk or partition is invisible to users until it is **mounted**: mounting is the act of attaching a file system's root to a point (the **mount point**, an existing directory) inside the running system's directory tree, so that the attached system's files appear as if they were simply more folders of the one tree. Plugging in a pen drive does exactly this: its file system is mounted at a point like /media/usb or a drive letter on Windows, and unmounting cleanly detaches it (after flushing pending writes — the reason for "safely remove"). Mounting explains a puzzle: one tree can span many devices, because the tree is a logical construction of the OS, not a map of any single disk.

Practice Set 12.4

- A pen drive's root contains a folder `photos`. After mounting at /media/usb, what is that folder's absolute path?

- Why is pulling a mounted pen drive out without unmounting risky even if no copy dialog is visible?

Common mistakes

- Treating the extension as the file's type in the OS's eyes under all systems: some systems store type internally or ignore extensions; the attribute list is the general answer.
- Calling direct access "faster access". It is order-free access; on the same hardware a sequential read of the whole file can be quicker than scattered direct reads.
- Drawing directory trees that return to an ancestor via a link and still calling the structure a tree — that is a graph, with its own deletion problems.

Summary

- A file is a named logical unit on secondary storage, with attributes (name, identifier, type, location, size, protection, timestamps, owner) and operations (create, write, read, reposition, delete, truncate; wrapped by open/close).
- Access methods: sequential (in order, pointer advances — tape, logs, media) and direct (by block number — databases on disks), with indexed access layered on direct.
- Directories evolve: single-level → two-level (per user) → tree (paths, working directory) → acyclic graph (links/sharing).
- Mounting attaches a file system at a mount point in the live tree; one logical tree can span many physical devices.

Chapter practice set

- List the file attributes and mark the two a user normally never sees.

- A payroll file is read fully every month in order; an ATM balance file jumps between accounts all day. Assign access methods and justify the tape question: which of the two could live on tape at all?
- Compare single-level, two-level and tree directories on naming collisions and organisation.
- Absolute path of a file is /home/teacher/marks/cs.xlsx; working directory is /home/student. Write its relative path.
- What is mounting? Explain with a pen drive, including what the mount point is and why safe removal matters.
- What changes when a tree directory becomes an acyclic-graph directory? Name the new capability and the new problem.

Chapter 13: File Sharing and Protection, File System Implementation and Types

(BTEUP CSE 3rd Semester · Subject 3.1 · Unit 5 — File Management)

What you will learn

- How files are shared between users, and the consistency question sharing raises
- Protection: access types, access control lists, and the owner/group/others model
- The layers of a file system and the on-disk structures it keeps
- How directories are implemented, and the three file allocation methods
- Free-space management, efficiency notes, and the file systems you will actually meet

13.1 File sharing

Sharing lets several users (and one user's several processes) work on the same file: a class's shared notes, a team's project folder, the institute's marks file. The mechanisms you know from Chapter 12 — links in an acyclic-graph directory, network file systems mounting remote storage — all deliver it. Sharing raises one conceptual question the examiner may press: **consistency**. If two users edit simultaneously, whose changes survive? Systems answer with **session semantics** (changes become

visible when the file is closed — the model of most network file systems) or the stricter **immutable shared files** (a shared file can never change; updates create a new version). In multi-user systems on one machine the OS must also enforce protection (next section) so that sharing stays a choice of the owner, not an accident of the design.

Practice Set 13.1

- Two teachers edit one marks file at the same moment on a shared drive. Under session semantics, when does each see the other's work — and what can go wrong?
- Why does sharing make protection more necessary, not less?

13.2 Protection

When a file is created, its owner decides who may do what. The OS enforces it on every operation. **Types of access** that can be controlled: read, write, execute, append, delete, and list (see a directory's contents). Three implementation approaches:

- **Access Control List (ACL)** — each file carries a list: which users/groups may perform which operations. Most precise; the list can grow long, and checking it costs time.
- **Owner / Group / Others classes** — the compact UNIX scheme: every file has an owner, a group, and the rest of the world; each class gets a read/write/execute triple (the rwx you saw in the OS Lab's Linux commands — `ls -l` shows them, `chmod` changes them). Three bits per class — nine bits total — fit in the file's record, trading fine detail for speed and simplicity.
- **Passwords per file** — simple but clumsy: one password cannot express "she may read, he may write", and passwords multiply beyond memory.

Protection extends beyond files to the directory containing them: without execute (search) permission on a directory, even a readable file inside it cannot be reached — the path itself is guarded.

Practice Set 13.2

- `ls -l` shows `-rwxr-x---` for a file. Who can do what? Answer for all three classes.
- An ACL and the rwx scheme can both express "only the owner and her group may read this". Give one rule the ACL can express that rwx cannot.

13.3 File system structure and implementation

The file system is itself layered — each layer does one job and hides the one below:

- **Application programs** call file operations.
- **Logical file system** — handles names, directories, protection and the file record (FCB): it resolves a name to a file's identifier and checks rights.
- **File-organisation module** — maps logical blocks of a file to free physical blocks (allocation, Section 13.5) and manages free space.
- **Basic file system** — translates "read physical block N" into device commands through the driver.
- **I/O control (device drivers + interrupt handlers)** — actually moves the data (Unit 4's world).

On disk, a partition's file system keeps: a **boot block** (code to start the OS, if this partition can boot), a **volume control block** (the partition's totals: block size, counts, free-space pointer), the **directory structure**, and one **File Control Block (FCB)** per file — the attributes of Chapter 12 plus the block pointers of Section 13.5; in UNIX the FCB is the famous

inode. In memory it keeps mount tables, open-file tables and caches of recently used blocks — the reason a second open of a file is nearly instant.

Practice Set 13.3

- Match the task to the layer: (a) checking a user may write this file (b) choosing which free blocks a growing file gets (c) ordering the disk heads.
- Where does the file's size live on disk, and where while the file is open?

13.4 Directory implementation

A directory is a list of (name → FCB/inode) mappings; how the list is organised decides lookup speed:

- **Linear list** — entries one after another; creating a file appends, searching scans from the top — simple, and slow as directories grow to thousands of entries.
- **Hash table** — the name is hashed to a position, giving near-instant average lookup; collisions must be chained, and a fixed-size table can overflow — real systems often combine a hash with short lists.

Either way, the design goal is the same: name resolution (Chapter 12's paths) must feel instant even in directories with a city's worth of files.

Practice Set 13.4

- A directory holds 20,000 files. Estimate the linear-list comparisons for an unlucky search vs a hashed one, in words.
- Why is the directory stored on disk at all, when it is searched so often?

13.5 File allocation methods — where the blocks go

A file's data must be placed in disk blocks; **how** they are placed decides speed, growth and fragmentation. The three classical methods (the OS Lab's file-allocation experiment covers all three):

Contiguous allocation. Each file occupies a run of consecutive blocks; its record stores only start block + length. Reading is superb (the arm barely moves; direct access is one addition). The costs mirror Chapter 7's contiguous memory: external fragmentation of the disk, and a file cannot grow beyond its run — choosing the run's size is the same fit problem with the same strategies.

Linked allocation. Each block ends with a pointer to the file's next block; the record stores only the first block. No external fragmentation, files grow by linking any free block, sequential reading is natural. Weaknesses: direct access is a walk through the chain (block 500 costs 500 reads), pointer space is spent in every block, and one lost pointer loses the rest of the file. Its famous repair is the **FAT** (File Allocation Table): all pointers move into one table at the disk's start, which the OS can cache in memory — the design of FAT file systems below.

Indexed allocation. All of a file's block pointers are collected into one **index block** (pointed to by the file's record). Direct access is now one index lookup plus the data read; no external fragmentation; files can grow until the index fills — larger files chain index blocks or use multi-level indexes (the UNIX inode mixes direct pointers with single/double indirect ones for exactly this scaling). Cost: the index block itself, paid even by tiny files.

| Basis | Contiguous | Linked | Indexed | |---|---|---|---| | Record stores | Start + length | First block | Index block number | | Sequential access | Excellent | Good | Good | | Direct access | Excellent | Very poor | Good |

| External fragmentation | Yes | No | No | | File growth | Hard | Easy | Up to index size | | Extra space cost | None | Pointer per block | One index block per file |

Practice Set 13.5

- A video file grows daily and is watched start-to-finish. A database file is read by record number all day. Assign allocation methods, one reason each.
- In FAT, where are the pointers, and why does moving them change linked allocation's worst weakness?

13.6 Free-space management

The file-organisation module must know which blocks are free. Four schemes:

- **Bit vector (bitmap)** — one bit per block (1 = free or 0 = free, by convention): compact (a 1 TB disk with 4 KB blocks needs 32 MB of bitmap) and makes finding runs of free blocks easy — ideal partner for contiguous allocation.
- **Linked list** — free blocks chained by pointers stored in the blocks themselves: no extra space, but traversing to count or find many blocks is slow.
- **Grouping** — the first free block stores the addresses of many other free blocks, the last of which stores the next group: long lists fetched in chunks.

- **Counting** — because free space tends to occur in runs, store (first free block, count of consecutive free blocks) pairs: few entries, runs described exactly.

Practice Set 13.6

- A nearly full disk's free space is a scattering of single blocks. Which scheme's entry count explodes least, and which describes it best?
- Why does the bitmap size not grow when the disk fills up?

13.7 Efficiency, performance, and the file systems you will meet

Three everyday performance ideas: **block caching** — recently used directory, FAT/index and data blocks are kept in RAM, turning most reads into memory hits; **read-ahead and write buffering** — sequential patterns are prefetched and writes are batched; and **block/cluster size choice** — large blocks speed big files but waste space on small ones (the file-world echo of internal fragmentation). Journaling file systems add reliability: changes are first written to a log, so a crash mid-write can be repaired instead of corrupting the volume.

| File system | Where you meet it | Character in one line | |---|---|---| | FAT32 | Pen drives, SD cards | Simple, universal; 4 GB maximum file size, no protection | | exFAT | Large pen drives, cameras | FAT's successor without the 4 GB limit | | NTFS | Windows system disks | Journaling, ACL protection, encryption support | | ext4 | Linux system disks | Journaling, indexed allocation via inodes, robust | | APFS | Apple devices | Snapshots, encryption, copy-on-write design |

Common mistakes

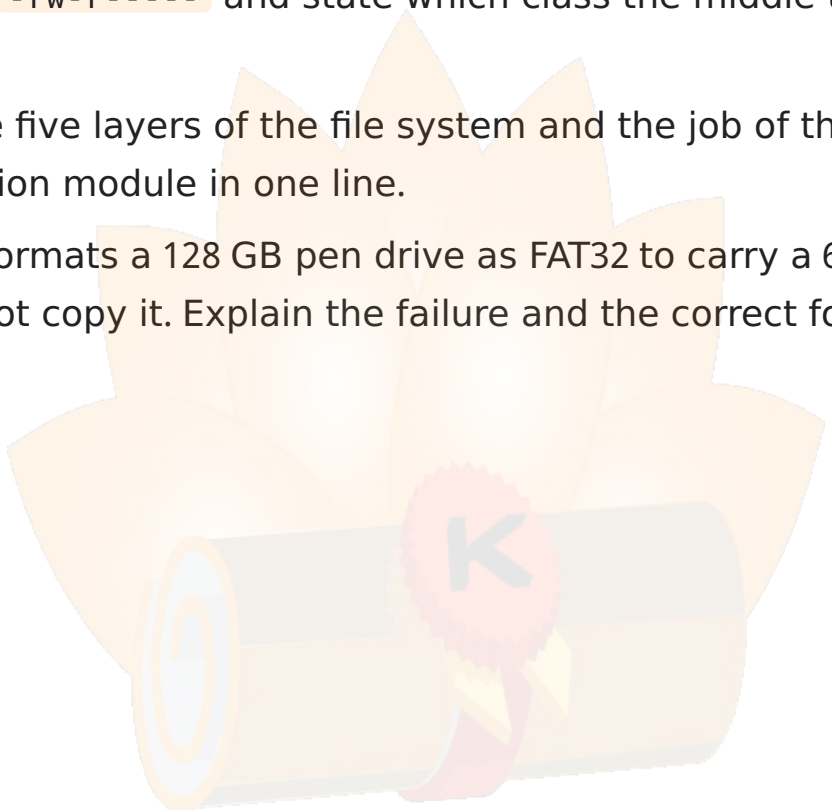
- Mixing allocation methods' weaknesses: contiguous → external fragmentation and growth; linked → direct access; indexed → index-block cost. The table is the marks.
- Defining protection as passwords only. The exam's core is access types plus ACL vs owner/group/others.
- Forgetting the FAT repair to linked allocation — it is the most asked follow-up of the unit.
- Saying the bitmap is wasteful: at 1 bit per block it is among the smallest structures on the disk.

Summary

- Sharing is delivered by links and network mounts; session semantics decide when edits become visible; protection keeps sharing the owner's choice.
- Protection controls read/write/execute/append/delete/list via ACLs (precise) or owner/group/others rwx classes (compact — 9 bits).
- The file system layers: logical FS (names, FCB, protection) → file-organisation module (logical→physical, free space) → basic FS → I/O control; on disk it keeps boot and volume control blocks, directories and FCBs/inodes.
- Directories are linear lists (simple, slow) or hash tables (fast), possibly combined.
- Allocation: contiguous (fast, fragments, no growth), linked (grows, poor direct access; FAT repairs it), indexed (balanced; index-block cost). Free space is tracked by bitmap, linked list, grouping or counting.
- Performance rests on caching, read-ahead and block size; FAT32/exFAT, NTFS, ext4 and APFS are the file systems of real life.

Chapter practice set

- Define the FCB and list any six of its contents.
- Compare the three allocation methods for a growing log file and a fixed-size database — choose and defend each time.
- Explain the FAT design: what problem of linked allocation does it attack, and how?
- Free blocks on a disk are {2, 3, 4, 9, 15, 16, 17, 30}. Show the free-space record under (a) linked list
  (b) counting.
- Interpret `-rw-r-----` and state which class the middle triple belongs to.
- Name the five layers of the file system and the job of the file-organisation module in one line.
- A friend formats a 128 GB pen drive as FAT32 to carry a 6 GB video and cannot copy it. Explain the failure and the correct format choice.



Appendix A: The Operating System Lab — 12 Experiments

(BTEUP CSE 3rd Semester · Subject 3.5 — Operating System Lab. Each experiment names its aim, the essence of the work, and the book chapter it brings alive. Write your programs and observations in your lab record in this order.)

Experiment 1 — Linux commands. Aim: become fluent with the shell. Practise file/directory commands (ls, cd, mkdir, cp, mv, rm, cat), viewing permissions with ls -l, and changing them with chmod. Brings alive: Chapters 2 (shell as system program), 12 and 13 (paths, rwx).

Experiment 2 — Installing Windows. Aim: install an operating system from media. Boot the installer, partition the disk, complete setup and first login. Note where partitioning and formatting appear. Brings alive: Chapter 10 §10.4 (the three formatting steps).

Experiment 3 — Task Manager study. Aim: observe live processes. List running processes, their PIDs, states and CPU/memory figures; end one process and watch the list. Brings alive: Chapter 3 (the PCB made visible).

Experiment 4 — FCFS and SJF scheduling. Aim: simulate both algorithms on given process sets and compare average waiting times. Brings alive: Chapter 4 §§4.2–4.3.

Experiment 5 — Round Robin and Priority scheduling. Aim: simulate both; vary the quantum in RR and report its effect on waiting time and context switches. Brings alive: Chapter 4 §§4.4–4.5.

Experiment 6 — Dining Philosophers. Aim: implement the table so deadlock can occur, observe it, then apply a fix (both-sticks-together or at most four seated) and show the deadlock gone. Brings alive: Chapters 5 §5.5 and 6.

Experiment 7 — Banker's algorithm: avoidance. Aim: implement the safety check and the request-granting test; run the book's worked example and one unsafe-state case of your own. Brings alive: Chapter 6 §6.4.

Experiment 8 — Deadlock prevention. Aim: demonstrate one prevention design (for example, resource ordering or request-all-at-once) on a two-resource scenario that deadlocks without it. Brings alive: Chapter 6 §6.3.

Experiment 9 — Memory fit strategies. Aim: implement First Fit, Best Fit and Worst Fit on given hole/request lists; reproduce the chapter's worked example and its lesson. Brings alive: Chapter 7 §7.3.

Experiment 10 — Page replacement algorithms. Aim: implement FIFO, LRU and Optimal on given reference strings; include one string showing Belady's anomaly under FIFO. Brings alive: Chapter 9 §9.2.

Experiment 11 — Disk scheduling. Aim: implement FCFS, SCAN and C-SCAN on given request queues and compare total head movement. Brings alive: Chapter 11 §11.1.

Experiment 12 — Directory structures and file allocation. Aim: model a tree directory with paths, and simulate contiguous, linked and indexed allocation for sample files, reporting each method's block map. Brings alive: Chapters 12 §12.3 and 13 §13.5.

A closing word: the lab is not separate from this book. Every experiment is one chapter with its sleeves rolled up — read the chapter first, and the experiment becomes a demonstration instead of a puzzle.

Glossary — Key Terms of This Book

- **Process** — a program in execution, with its own memory and state.
- **Kernel** — the always-resident core of the OS, the only part that touches hardware directly.
- **System call** — the controlled doorway by which programs request kernel services.
- **PCB** — the Process Control Block: the OS's record of one process.
- **Context switch** — saving one process's state and loading another's so the CPU can change hands.
- **Critical section** — the part of a process's code that touches shared data.
- **Semaphore** — an OS integer used only through the atomic wait and signal operations.
- **Race condition** — shared data changed concurrently so the result depends on timing.
- **Deadlock** — a set of processes each holding what another in the set waits for.
- **Safe state** — a state from which every process can still obtain its maximum and finish.
- **Logical / physical address** — the address as the program sees it / the real location in RAM.
- **Page / frame** — equal fixed-size blocks of logical / physical memory.

- **Page fault** — the event of touching a page that is not in memory, so it must be fetched.
- **Fragmentation** — memory wasted in unusable pieces: inside blocks (internal) or between them (external).
- **Virtual memory** — a per-process address space larger than RAM, served by demand paging.
- **Thrashing** — collapse caused when combined working sets exceed memory and faults dominate.
- **Seek time** — the time to move a disk arm to the wanted track.
- **RAID** — several disks combined for speed (striping) and/or survival (mirroring, parity).
- **File** — a named logical unit of stored information.
- **inode / FCB** — the per-file record holding attributes and block pointers.
- **Mounting** — attaching a file system at a point in the live directory tree.

About KAMALDNP

KAMALDNP is an IT training institute in Varanasi, Uttar Pradesh, offering computer courses and summer training, and publishing study material for diploma and engineering students through KAMALDNP.com.

KAMALDNP · kamaldnp.com · support@kamaldnp.com