OS – Complete Notes

📘 Operating System – Unit 1: Introduction

🔹 Topic 1: What is an Operating System?

🎓 *[College Level]*

✅ **Definition:**

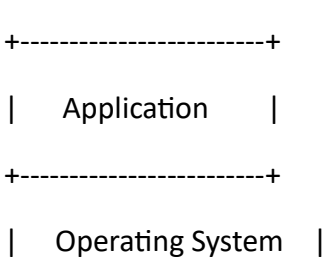
An **Operating System (OS)** is **system software** that acts as an **interface between user and computer hardware**.

It **manages** hardware and software resources and provides **services** to application programs.

✅ Functions of an Operating System

Function	Description
Process Management	Controls process creation, execution, termination
Memory Management	Allocates and deallocates memory
File System Management	Manages files and directories
Device Management	Handles input/output devices
Security & Protection	Controls access to system and data
User Interface	Provides CLI or GUI for interaction

✅ Diagram: OS as a Layer



+------------------------+

| Computer Hardware |

+------------------------+

You don’t talk to hardware directly. OS does it for you.

✅ **Examples of Operating Systems**

Type	OS Examples
Desktop	Windows, Linux, macOS
Mobile	Android, iOS
Server	RedHat, Ubuntu Server
Embedded	RTOS, VxWorks

✅ **Key Characteristics:**

- **Resource Manager**: Allocates CPU, memory, etc.
- **Concurrency**: Manages multiple tasks
- **Multitasking**: Runs many programs at once

✅ **Summary of Topic 1 (College):**

- OS = bridge between user and hardware
- Manages CPU, memory, I/O, files
- Examples: Windows, Linux, Android

📘 **Topic 2: Types of Operating Systems**

🎓 *[College Level]*

🔹 **Why Different Types of OS Exist?**

Different systems (like ATM, smartphones, servers) need **different behavior** from the OS — some need **speed**, others need **real-time control**, others need **multi-user handling**.

✅ 1. Batch Operating System

🔹 **Concept:**

- No interaction between user and computer during execution
- Jobs are collected (batched) and executed one after another

🔹 **Example:**

- Early mainframe computers
- Punch card era

🔹 **Diagram:**

User → Submit jobs → [Batch Queue] → Processed by OS → Output

---

✅ 2. Time Sharing OS

🔹 **Concept:**

- Multiple users share system resources **at the same time**
- CPU switches between users rapidly (called **time slicing**)

🔹 **Example:**

- UNIX, Multi-user Linux systems

🔹 **Advantage:**

- Interactive computing
- Fair CPU time for each user

✅ 3. Multiprogramming OS

🔹 **Concept:**

- **Multiple programs** are in memory at once
- OS switches between them so CPU is never idle

Main Goal: **Maximize CPU Utilization**

🔹 **Example:**

- Most modern OS like Windows/Linux internally use this

✅ **4. Multitasking OS**

🔹 **Concept:**

- One user can run **multiple tasks** simultaneously
- Tasks are managed using time slicing

Example: Playing music + coding in VS Code + downloading a file

---

✅ **5. Real-Time Operating System (RTOS)**

🔹 **Concept:**

- Responds to input/output **within strict time limits**
- Critical for applications like:
  - Flight systems 🛬
  - Medical devices 💉
  - Robots 🤖

🔹 **Two Types:**

Type	Description	Example
Hard RTOS	Missing deadline = failure	Pacemaker
Soft RTOS	Deadline miss = degraded result	Video streaming

✅ **6. Distributed OS**

🔹 **Concept:**

- Manages **a group of computers** so they work like one system
- **Resources shared** across machines

🔹 **Example:**

- Google Cloud, Amazon AWS (internally)
- Hadoop Distributed File System (HDFS)

✅ 7. Network OS

🔹 **Concept:**

- OS designed to manage **networked computers**
- Provides file sharing, printer sharing, remote login

🔹 **Example:**

- Novell NetWare, Windows Server OS

✅ 8. Mobile OS

🔹 **Concept:**

- Designed for smartphones, tablets
- Optimized for low power, touch interface

🔹 **Example:**

- Android (Linux-based), iOS

📊 Summary Table

OS Type	Key Idea	Examples
Batch	No user interaction	Early Mainframes
Time Sharing	Time slice for each user	UNIX
Multiprogramming	Multiple programs in memory	Linux, Windows
Multitasking	One user, many tasks	Windows, Android
RTOS	Instant response, time-critical	Pacemaker, Drones
Distributed	Multiple systems work as one	Google Cloud, HDFS
Network	Manages network resources	Windows Server

OS Type	Key Idea	Examples
Mobile	OS for mobile/touch UI	Android, iOS

📘 Topic 3: Functions of Operating System

🎓 *[College Level]*
(💼 Industry-level covered where relevant)

🔹 Why Functions Matter?

OS is not just one feature – it's a **manager** that:

- Allocates resources
- Executes programs
- Handles files/devices
- Secures system

✅ Core Functions of Operating System

Function	Description
1️⃣ Process Management	Creating, scheduling, and terminating processes
2️⃣ Memory Management	Managing primary memory (RAM)
3️⃣ File System Management	Storing, naming, and protecting files
4️⃣ Device Management	Managing hardware I/O devices
5️⃣ I/O Management	Handling input/output requests
6️⃣ Security & Protection	Restricting access to resources
7️⃣ User Interface	CLI (terminal) or GUI (desktop interface)

Function	Description
8️⃣ Networking	Managing communication between systems

Let's break these down:

🔸 1. Process Management

🎓 **What It Does:**

- Creates processes (using system calls like fork())
- Schedules which process gets CPU (we'll learn algorithms later)
- Tracks state via **Process Control Block (PCB)**

💼 **Industry Use:**

- Server apps must manage thousands of processes efficiently
- DevOps uses tools like htop, kill, top, pm2 to manage processes

---

🔸 2. Memory Management

🎓 **What It Does:**

- Allocates and deallocates memory
- Keeps track of **who is using which memory block**
- Manages **Virtual Memory** (we'll study this later)

💼 **Industry Use:**

- Backend/server optimization often needs memory tuning
- Cloud systems bill for RAM → efficiency is key

🔸 3. File System Management

🎓 **What It Does:**

- Handles file creation, deletion, access
- Provides naming, storage, access control
- Maintains file hierarchy (FAT, NTFS, ext4)

💼 **Industry Use:**

- Full-stack devs need to understand file I/O (e.g., reading configs, logs)
- DevOps manage file permissions, mounts, backups

🔸 **4. Device Management**

🎓 **What It Does:**

- OS uses **device drivers** to communicate with hardware
- Queues and controls I/O requests

💼 **Industry Use:**

- Used in **Embedded Systems**, printers, display drivers, etc.
- Linux device management (udev, lsblk, /dev) is essential in sysadmin roles

---

🔸 **5. I/O Management**

- Handles input/output requests (e.g., keyboard, mouse, printer)
- Buffers and interrupts

🔸 **6. Security & Protection**

🎓 **Role:**

- Protects files, memory, and CPU from unauthorized access
- User authentication (login), permissions (read/write)

💼 **Relevance:**

- Cloud & SaaS systems heavily depend on this
- Devs use access controls (ACLs, firewalls) daily

🔸 **7. User Interface**

- **CLI** (Command Line Interface): bash, PowerShell
- **GUI** (Graphical): Windows, GNOME, macOS

Some OS (e.g., embedded Linux) **only have CLI** to save resources.

🔸 **8. Networking (Modern Function)**

- Manages network stack (TCP/IP)
- Controls how devices connect and communicate

Often left out in books, but used in **web servers, FTP, file sharing, SSH**

📊 **Summary Table**

Function	Key Role	College Importance	Industry Use
Process Management	Runs programs	✅ ✅ ✅	✅ ✅ ✅
Memory Management	RAM allocation	✅ ✅ ✅	✅ ✅ ✅
File System Management	Store & access files	✅ ✅ ✅	✅ ✅ ✅
Device Management	I/O device control	✅ ✅	✅ ✅ ✅
I/O Management	Handle input/output	✅ ✅	✅ ✅
Security	Authentication & Protection	✅ ✅ ✅	✅ ✅ ✅
User Interface	GUI or CLI	✅ ✅	✅ ✅ ✅
Networking	Communicate with others	Optional	✅ ✅ ✅

🧠 **Easy Memory Trick:**

"**Please Make Food, Don’t Ignore Security, Use Net**"

Process, **M**emory, **F**ile, **D**evice, **I**/O, **S**ecurity, **U**I, **N**etwork

---

📘 Operating System – Unit 2: Process Management

🔹 **Topic 1: Program vs Process**

🎓 *[College Level]*

✅ **Definition:**

Term	Meaning
Program	A passive set of instructions (stored in a file). Not yet running.
Process	An active program in execution. It uses CPU, RAM, I/O, etc.

🔁 **Real-Life Analogy:**

A **program** is like a recipe book (just instructions).
A **process** is when someone is actually cooking using that recipe 🔥.

🧠 **Example:**

You double-click **Chrome**:

- The .exe file is the **program**
- The tab you opened is a **process**

✅ **Key Differences:**

Program	Process
Stored on disk (passive)	Loaded into memory (active)
No CPU usage	Requires CPU, memory, I/O
Single copy	Multiple processes can exist
Static	Dynamic – it changes as it runs

🔹 **Topic 2: Process Control Block (PCB)**

🎓 *[College Core]*

✅ **Definition:**

PCB is a **data structure maintained by the OS** to keep all information about a process.

It's like the **Aadhaar card** of a process.

✅ **Contents of PCB:**

Field	Description
PID	Process ID (unique)
Process State	Ready, Running, Waiting, etc.
Program Counter	Location of next instruction
CPU Registers	Values saved during context switch
Memory Info	Base, limit addresses
I/O Info	Files/devices used
Accounting Info	Time used, priority, etc.

📊 **Diagram: PCB**

+--------------------------+

| Process ID (PID) |

| Process State |

| Program Counter |

| CPU Registers |

| Memory Limits |

| Open File List |

| I/O Status |

+--------------------------+

💼 **Industry View:**

- Process debugging tools (like top, ps, or task manager) **read PCB-like structures**
- Context switching in OS is built using **PCB snapshots**
- Linux internally stores PCB in task_struct (kernel dev)

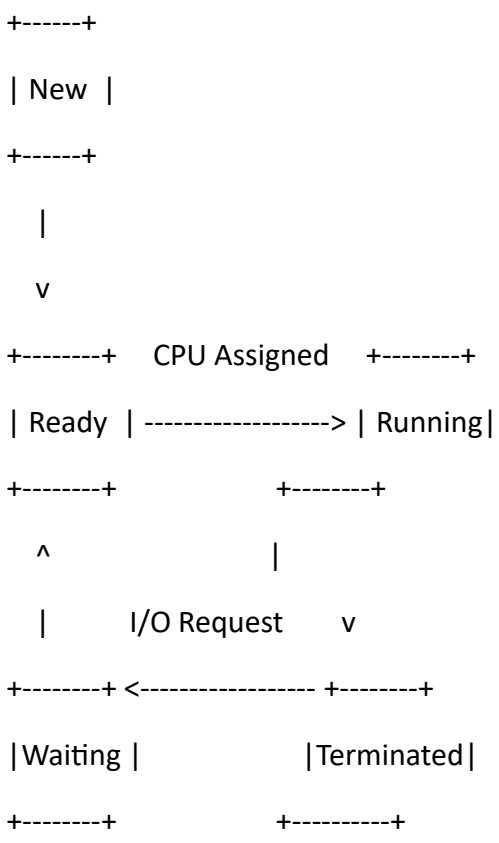
🔹 **Topic 3: Process States**

🎓 *[College Must-Know]*

✅ **Five Classic States:**

State	Meaning
New	Process is being created
Ready	Waiting for CPU
Running	Actively executing
Waiting	Waiting for I/O or event
Terminated	Finished execution

📊 **Diagram: Process Lifecycle**



💼 **Industry View:**

Understanding these states helps in:

- Analyzing performance (why processes are stuck in waiting)
- Managing concurrency in server apps
- Writing responsive apps (UI threads should not block)

✅ **Summary So Far:**

Topic	College Level	Industry Use
Program vs Process	✅ ✅ ✅	✅ ✅
PCB	✅ ✅ ✅	✅ ✅ ✅
Process States	✅ ✅ ✅	✅ ✅ ✅

🔹 **Topic 4: CPU Scheduling Algorithms**

🎓 *College Core* + 💼 *Industry Use (where needed)*

✅ **What is CPU Scheduling?**

When **multiple processes** are in the **Ready Queue**, the **OS must choose** which one to run next on the CPU.

This decision is made by the **CPU Scheduler**, using a **scheduling algorithm**.

📌 **Key Terms to Understand Before We Start**

Term	Meaning
Burst Time (BT)	Time process needs CPU
Arrival Time (AT)	Time when process enters ready queue
Completion Time (CT)	Time when process finishes
Turnaround Time (TAT)	CT − AT

Term	Meaning
Waiting Time (WT)	TAT − BT

We'll use these in all algorithms.

🔁 **Algorithm #1: FCFS – First Come First Serve**

🎓 **Concept:**

- Process that arrives **first** gets CPU **first**
- Like a queue at a railway counter 🚆

🧠 **Non-preemptive (Once running, it won't be stopped)**

🍼 **Example:**

Process	Arrival Time	Burst Time
P1	0	4
P2	1	3
P3	2	1

🟩 **Gantt Chart:**

Time → | 0 | 4 | 7 | 8 |

P1 P2 P3

🧮 **Calculation:**

Process	AT	BT	CT	TAT = CT−AT	WT = TAT−BT
P1	0	4	4	4	0
P2	1	3	7	6	3
P3	2	1	8	6	5

✅ Avg TAT = (4+6+6)/3 = **5.33**
✅ Avg WT = (0+3+5)/3 = **2.66**

📌 **Pros & Cons:**

Pros	Cons
Simple	Long processes block short ones
Fair	Not optimal for waiting time

🔁 **Algorithm #2: SJF – Shortest Job First**

🎓 **Concept:**

- CPU goes to the process with **smallest burst time**
- Think: Small tasks finish quickly and free up CPU

🔸 **Two Variants:**

- **Non-Preemptive** → once running, not interrupted
- **Preemptive SJF (aka SRTF)** → can interrupt if a shorter job arrives (we'll do this tomorrow)

🧪 **Example (SJF – Non-Preemptive):**

Same table as above:

Process	Arrival Time	Burst Time
P1	0	4
P2	1	3
P3	2	1

🟩 **Gantt Chart:**

Time → | 0 | 4 | 5 | 8 |

P1 P3 P2

P1 starts first since it arrived at 0
After P1 finishes, P3 (shortest remaining) runs, then P2

🧮 **Calculation:**

Process	AT	BT	CT	TAT = CT−AT	WT = TAT−BT
P1	0	4	4	4	0
P2	1	3	8	7	4
P3	2	1	5	3	2

✅ Avg TAT = (4+7+3)/3 = **4.66**
✅ Avg WT = (0+4+2)/3 = **2.00**

📌 **Pros & Cons:**

Pros	Cons
Minimum average waiting time	Starvation: long processes may never run

---

🔁 **Algorithm #3: Priority Scheduling**

🎓 **Concept:**

- Each process has a **priority number**
- Lower number = higher priority (unless stated otherwise)

🔸 **Two types:**

- **Preemptive** → Interrupt if a higher priority arrives
- **Non-preemptive** → Wait until current finishes

💉 **Example:**

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

Lower value = higher priority

🟩 **Gantt Chart (Non-preemptive):**

Time → | 0 | 4 | 7 | 8 |

P1 P2 P3

(P1 arrives first and gets CPU. No interruption)

Process	CT	TAT	WT
P1	4	4	0
P2	7	6	3
P3	8	6	5

✅ Avg TAT = **5.33**
✅ Avg WT = **2.66**

🔁 **Algorithm #4: Round Robin (RR)**

🎓 **Concept:**

- Each process gets a **fixed time slot (quantum)** in round-robin fashion
- After its time expires, next process gets CPU

Most commonly used in **time-sharing systems** (e.g., Linux)

⏱️ **Time Quantum = 2 units**

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

🟩 Gantt Chart:

0 2 4 6 7 8

P1 P2 P1 P3 P2

(We'll skip detailed calculation here unless you want it step-by-step)

✅ Avg WT: ~2.66
✅ Avg TAT: ~5.33

✅ **Summary: Algorithm Comparison**

Algo	Type	Preemption	Fair	Fastest Avg WT
FCFS	Queue-based	No	No ❌	
SJF	Shortest Job	No	No ✅	
Priority	Based on value	Both	No	Maybe
RR	Time slicing	Yes	✅ ❌ (but fair)	

💼 **Industry Use**

Scheduling Used In

Linux – Round Robin, Priority

Android – uses **CFS (Completely Fair Scheduler)**

Real-time OS – use **custom priority schedulers**

✅ **Topic 4 Complete!**

📘 **Unit 2 – Process Management**

🔹 **Topic 5: Advanced Scheduling Algorithms (Preemptive + Multi-Level)**

🔁 **Algorithm #1: SRTF – Shortest Remaining Time First**

🎓 *Preemptive version of SJF*
💼 Used in OS kernels (e.g., for foreground processes)

✅ **Concept:**

- Always pick the process with the **least remaining burst time**
- If a **shorter job arrives**, **preempt** the current one

🍼 **Example:**

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	2
P4	3	1

🟩 **Gantt Chart (Step-by-step):**

Time →

0 1 2 3 4 5 6 7 8 9 10 11 12

P1 P2 P3 P4 P3 P2 P2 P2 P1 P1 P1 P1 P1

P1 starts, but gets preempted multiple times as shorter jobs arrive.

🧮 **Calculation (Summary):**

Process	CT	TAT (CT−AT)	WT (TAT−BT)
P1	13	13	5
P2	8	7	3
P3	5	3	1
P4	4	1	0

✅ Avg TAT = (13+7+3+1)/4 = **6**
✅ Avg WT = (5+3+1+0)/4 = **2.25**

✅ **Advantage:**

- Minimum avg waiting time

❌ **Disadvantage:**

- High complexity
- Can cause **starvation** for long jobs

🔁 **Algorithm #2: Preemptive Priority Scheduling**

✅ **Concept:**

- Like normal priority, but **current process can be preempted** if a process with **higher priority** arrives (i.e., lower priority number)

💉 **Example:**

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

🟩 Gantt Chart:

0 1 4 5

P1 P2 P1 P3

P2 interrupts P1 due to higher priority.

Process	CT	TAT	WT
P1	5	5	0
P2	4	3	0
P3	6	4	3

✅ Avg WT ≈ **1**
✅ Avg TAT ≈ **4**

❌ **Starvation Risk:**

Low priority processes may **never get CPU** if high-priority ones keep coming.

✅ Solution: **Aging** (increase priority over time)

🔁 **Algorithm #3: Multilevel Queue Scheduling**

✅ **Concept:**

- **Multiple ready queues**, each for different priority/class of processes
- Example:
    - System processes (Queue 1)
    - Interactive (Queue 2)
    - Background (Queue 3)

🚦 **Fixed priority between queues:**

- Always run Queue 1 first, then Queue 2, etc.

🧠 **Real-Life Analogy:**

Think of:

- VIP Passengers (Q1)

- Regular Ticket (Q2)
- Waitlist (Q3)

VIPs get served first always.

❌ **Issue:**

Lower queues **starve** without CPU

🔁 **Algorithm #4: Multilevel Feedback Queue**

✅ **Concept:**

- Similar to multilevel queue, **but flexible**
- A process can **move between queues** based on its behavior

⚙️ **Rules:**

- If a process uses **too much CPU**, move it to a lower-priority queue
- If a process waits too long, promote it to higher queue

OS tries to **balance fairness and efficiency**

💼 **Where It's Used:**

- Modern Linux/Windows schedulers use **modified multilevel feedback**
- Highly interactive systems (UIs, games) benefit from this

✅ **Summary of All Scheduling Algorithms**

Algorithm	Preemptive	Optimal WT	Starvation?	Fairness
FCFS	❌	❌	❌	✅
SJF (Non-Pre)	❌	✅	✅	❌
SRTF	✅	✅	✅ ✅	❌
Priority (Non-Pre)	❌	Medium	✅	❌

Algorithm	Preemptive	Optimal	WT	Starvation?	Fairness
Priority (Pre)	✅	Medium	✅✅	❌	
Round Robin	✅	❌	❌	✅✅	
Multilevel Queue	Optional	Medium	✅✅	❌	
Multilevel Feedback	✅	✅	❌	✅✅✅	

🎓 **Exam Tips:**

- Practice Gantt charts with different arrival times
- Formulas:
  - TAT = CT − AT
  - WT = TAT − BT
- Know when preemption happens

💼 **Industry Tip:**

Modern OS (Linux, Windows) use **modified multilevel feedback + priority**

- Real-time systems use **custom preemptive** schedulers
- For performance debugging → schedulers matter a lot (e.g., in game engines, OS kernels)

📘 **Unit 2 – Process Management**

🔹 **Topic 6: Context Switching, Threads, and Multithreading**

🎓 College + 💼 Industry Blend

🔁 **Part 1: Context Switching**

🎓 Required in exams
💼 Happens in real-time OS and multitasking apps

✅ What is Context Switching?

When the CPU switches from one process (or thread) to another, it **saves the current state** (context) and **loads the next one**.

That's called **Context Switch**.

🔧 What is Stored in "Context"?

- Program Counter (PC)
- CPU Registers
- Process State
- Stack Pointer
- Memory info

📊 Diagram: Context Switching

🔁 [Process A Running]

↓ (I/O Request or Time Over)

📦 Save A's Context → PCB_A

🔁 [Process B Starts]

📥 Load B's Context ← PCB_B

💼 Real-World Analogy:

Like a mobile game pausing when you answer a call.
When you return, the **exact screen** is restored.

⏱️ Is it Free?

No. Context switching takes **CPU time**, known as:

Context Switch Overhead

Too many switches = slow system

🧠 **Why It Happens:**

- Time slice ends (RR)
- I/O wait
- Higher priority process arrives

✅ **Summary: Context Switching**

Point	Value
Saves + Loads Process State	✅
Happens on every preemption	✅
Consumes CPU Time	✅
Enables multitasking	✅

🧵 **Part 2: Threads and Multithreading**

✅ **What is a Thread?**

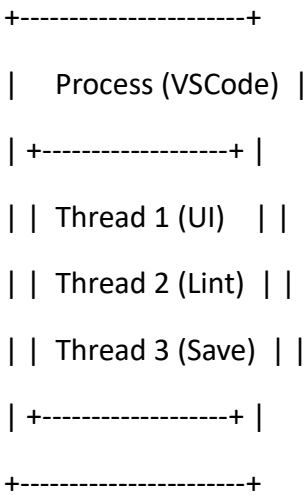
A **thread** is a lightweight **unit of CPU execution** inside a process.

One process can have **multiple threads** sharing the same memory.

🔁 **Process vs Thread**

Feature	Process	Thread
Memory	Has its own memory space	Shares memory with process
Communication	Slower (via IPC)	Fast (shared memory)
Creation Time	High	Low
Examples	Chrome, VS Code	Tabs, Extensions, UI, etc.

📊 Diagram: Threads in a Process



💼 Industry Relevance:

- Web servers (Node.js, Java, Python) use threads to handle **parallel requests**
- Game engines (Unity, Unreal) use threads for **physics**, **rendering**, **AI**
- Multithreaded systems = more responsive, faster apps

🔸 Single Thread vs Multithreading:

Type	Description
Single Thread	One task at a time
Multithreading	Multiple tasks in parallel
Multicore + Multithreading	Even faster; uses all CPU cores

🔸 Types of Threads:

Type	Description
User-Level Threads	Managed by user libraries (e.g., POSIX)
Kernel-Level Threads	Managed by OS (Linux, Windows kernel)

Modern systems often support both (hybrid threading).

✅ **Benefits of Multithreading:**

Benefit	Example
Faster execution	Parallel tasks
Better CPU use	Threads run on idle cores
Responsiveness	UI thread + worker thread
Cost-efficient	Cheaper than spawning new processes

⚠️ **Thread Problems (Asked in viva):**

Issue	Meaning
Race Condition	2 threads access/modify data at once
Deadlock	Threads wait forever for each other's resource
Starvation	A thread is always skipped
Context Switch Overhead	Too many switches = slow system

We'll study these in detail in **Deadlock & Synchronization (Unit 3)**.

✅ **Summary Table**

Concept	College Required	Industry Use
Context Switching	✅ ✅ ✅	✅ ✅ ✅
Threads	✅ ✅ ✅	✅ ✅ ✅
Multithreading	✅ ✅	✅ ✅ ✅
Issues	✅ ✅	✅ ✅ ✅

🎯 **Revision Tip (One Line):**

“A thread is the smallest unit of execution; context switching lets threads and processes **take turns** on the CPU.”

📘 Unit 3 – Process Synchronization & Deadlocks

🔹 **Topic 1: Critical Section Problem & Race Condition**

✅ Why Synchronization is Needed?

In a **multitasking or multithreaded environment**, **multiple processes or threads** can access **shared resources** at the same time.

If they don't coordinate, it causes bugs.

🧠 Real-World Analogy:

Two people editing the same Google Doc without coordination — one deletes while the other writes. Result = **corruption** 🧨

🔸 What is a Race Condition?

When **two or more processes access shared data** at the same time, and the **final result depends on who runs first**, it is a **race condition**.

📉 Example:

Assume a shared variable: count = 0

Two threads execute:

count = count + 1;

Each does:

1. Read count
2. Add 1
3. Write result back

If both read count = 0 at the same time, both write 1, and you lose an increment. ❌

Expected result: 2
Actual result: 1 🧨

🔸 What is the Critical Section?

A **critical section** is a part of a program where the process **accesses shared resources** (like memory, files, etc.)

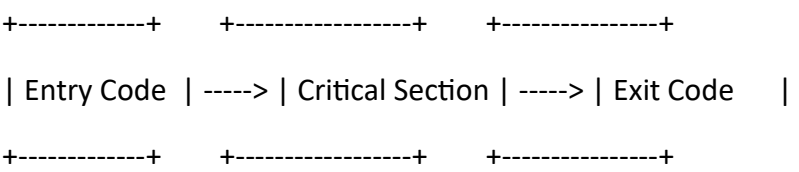
Only **one process** should execute its critical section at a time.

✅ Rules for Critical Section Problem (Very Important)

Every synchronization solution must satisfy these 3 conditions:

Rule	Meaning
Mutual Exclusion	Only one process in critical section at a time
Progress	If no one is in CS, some process must enter
Bounded Waiting	No process should wait forever

📊 Diagram: Critical Section



All solutions focus on building safe **entry** and **exit** codes.

🔸 Solution Approaches Overview

Type	Description
Software Algorithms	Peterson’s Algo, Bakery Algo
Hardware Support	Locks, Interrupt Disable
OS Support	Semaphores, Mutex, Monitors

✅ Summary:

Term	Meaning
Race Condition	Multiple processes interfere with each other
Critical Section	Code that accesses shared resource
Mutual Exclusion	Only 1 process allowed at a time
Progress	Some process must proceed
Bounded Waiting	No one should wait forever

🎓 **Viva/Exam Tip:**

Q: What is a Race Condition?

A race condition happens when the result of execution depends on the order of access to shared variables, and no synchronization is used.

Q: What are the 3 conditions for solving Critical Section Problem?

Mutual Exclusion, Progress, Bounded Waiting ✅

✅ **Topic 1 Complete!**

➡️ **Next Topics (Topic 2 onward):**

1. **Software Solutions**
    - 🧠 Peterson's Algorithm
    - 🛒 Bakery Algorithm
2. **Hardware Solutions** (Test-and-Set, Disable Interrupts)
3. **Semaphores and Mutexes** (🔑 Most used in industry)
4. **Classical Problems** (Producer-Consumer, Dining Philosopher)
5. **Deadlocks** (Conditions + Prevention + Banker's Algo)

Shall we continue with **Peterson's Algorithm** next?

Great! Let's now move into **Topic 2 of Unit 3** – a classic exam and viva question:

📘 **Unit 3 – Process Synchronization**

🔹 **Topic 2: Peterson's Algorithm**

🎓 *Software solution for Critical Section*
💼 *Conceptual understanding helps in thread-safe coding*

✅ **Goal:**

We want to allow **two processes** to access a **shared resource**, but **only one at a time** → i.e., solve the **Critical Section Problem**.

Peterson's Algorithm does this using **software only** (no hardware lock, no OS help).

🔸 **Assumptions:**

- Works for **2 processes only** (P0 and P1)
- Shared memory model (both can read/write same variables)

📦 **Shared Variables:**

```
boolean flag[2];    // flag[i] = true means process i wants to enter CS
int turn;           // whose turn it is
```

🧠 **Peterson's Algorithm Code**

🔹 **For Process 0:**

```
flag[0] = true;        // I want to enter
turn = 1;              // Give turn to P1
while (flag[1] && turn == 1)
   ;                   // Wait (Busy waiting)

// === Critical Section ===

flag[0] = false;       // Done
```

🔹 **For Process 1:**

```
flag[1] = true;
turn = 0;
while (flag[0] && turn == 0)
   ;              // Wait

// === Critical Section ===

flag[1] = false;
```

🔁 **How It Works:**

- Both processes set their **flag** to true.
- They also set **turn** to the other process.
- If both want to enter at the same time, the **turn** variable ensures **only one proceeds**.

📊 **Timeline Example:**

1. **P0** wants to enter: sets flag[0]=true, turn=1
2. **P1** wants to enter: sets flag[1]=true, turn=0
3. Now:
    - P0 sees flag[1]=true and turn=1 → can't enter → **waits**
    - P1 sees flag[0]=true but turn=0 → can't enter → **waits**
      → But the **last one to set turn loses** (ensures fairness)

✅ **Satisfies All 3 Critical Section Conditions:**

Condition	Satisfied?	How?
Mutual Exclusion	✅	Only one can enter due to turn and flag logic
Progress	✅	No process is stuck forever unless other is in CS

Condition	Satisfied?	How?
Bounded Waiting	✅	The turn variable ensures alternate access

⚠️ **Limitations:**

Issue	Reason
Works only for 2 processes	Not scalable
Uses busy waiting	CPU cycles are wasted

We'll later see **Semaphores**, which fix this.

🧠 **Easy Memory Tip:**

"Set your **flag**, give **turn** to other, and **wait** till safe."

💼 **Relevance in Industry**

While real systems **don't use Peterson's Algorithm** directly, it helps understand how synchronization logic works:

- Thread-safe shared variables
- Spinlocks (used in OS, DB, drivers)
- Building mutex-like logic in embedded/low-level systems

✅ **Topic 2 Complete!**

Awesome, Ghanshyam! Let's now move to:

📘 **Unit 3 – Process Synchronization**

🔹 **Topic 3: Bakery Algorithm** 🥐

🎓 *Multi-process software solution to Critical Section problem*

✅ What is the Bakery Algorithm?

Developed by **Leslie Lamport**, it is a **generalized version of Peterson's Algorithm** that works for **multiple processes** (not just 2).

🧠 Analogy:

Like a **bakery token system** – each customer takes a number and waits for their turn.

The customer with the **smallest token number** enters first.

📦 Shared Variables:

```
boolean choosing[n];    // is process i choosing its number?
int number[n];          // ticket number for each process
```

Where n = number of processes

🔄 Steps for Process i:

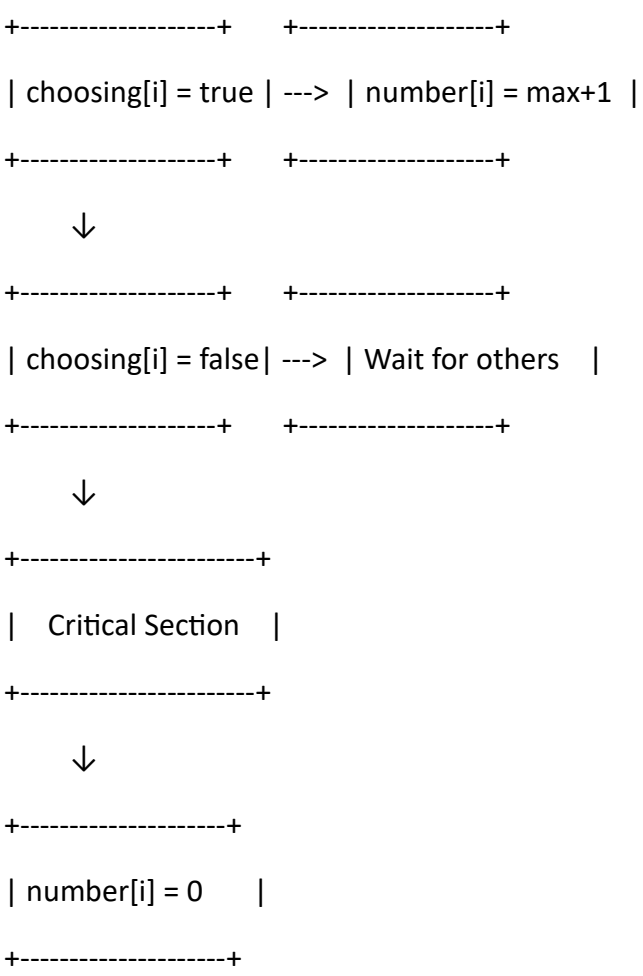
```
choosing[i] = true;
number[i] = 1 + max(number[0], number[1], ..., number[n-1]);
choosing[i] = false;

// Wait until it's my turn
for (j = 0; j < n; j++) {
   while (choosing[j]) ; // Wait if j is choosing
   while ((number[j] != 0) &&
        (number[j] < number[i] ||
        (number[j] == number[i] && j < i)))
      ; // Wait if j has higher priority
}

// === Critical Section ===
```

number[i] = 0; // Exit CS

🧠 **Breakdown of Logic:**

- Each process **chooses a number** (like taking a token).
- It waits for all other processes who:
  - Are choosing
  - Have **smaller** token
  - OR same token but **smaller process ID** (tie-breaker)

🔁 **Diagram (Visual Flow):**



✅ **Satisfies All Critical Section Conditions**

Condition	Satisfied?	Reason
Mutual Exclusion	✅	Token order enforces exclusive access
Progress	✅	No deadlock, eventually someone enters
Bounded Waiting	✅	Every process gets turn in order

⚠️ Limitations

Problem	Detail
Slow with many processes	Finding max number takes time
Uses busy-waiting	Like Peterson's, wastes CPU cycles
Rarely used today	Too slow for real systems

💼 Industry View

Bakery Algorithm is **not used directly**, but:

- It teaches **fairness, order, and symmetry**
- Inspired **ticket locks**, **distributed mutexes**, etc.
- Helps in **low-level algorithm design** for distributed systems

---

✅ Summary Table

Feature	Peterson's	Bakery
No. of Processes	2 only	Any number
Fairness	Fair with turn	Fully fair
Waiting	Busy waiting	Busy waiting
Usage	Teaching concept	Teaching concept

✅ Topic 3 Complete!

Great! Let’s continue with:

📘 **Unit 3 – Process Synchronization**

🔹 **Topic 4: Hardware-Based Synchronization**

🎓 College ✅ | 💼 Industry (OS Kernel, Device Drivers) ✅ ✅ ✅

✅ **Why Hardware Solutions?**

Software-only solutions (Peterson's, Bakery) are:

- Slow ⚠️
- Complex 😵
- Not scalable ❌

So, CPUs provide **atomic instructions** that OS can use for **fast and safe synchronization**.

🧲 **Atomic Operation**

Atomic = cannot be interrupted. It completes in **one step** without interference.

Examples: test-and-set, compare-and-swap

🔧 **1. Test-and-Set (TAS) Instruction**

✅ **Concept:**

An **atomic** instruction that:

- **Reads** a memory location (e.g., lock)
- Sets it to 1
- Returns the **old value**

📜 **Pseudo-code:**

```
boolean TestAndSet(boolean *target) {
   boolean old = *target;
```

```
    *target = true;
    return old;
}
```

🛠️ **How It's Used (Spinlock):**

```
while (TestAndSet(&lock)) {
    // Busy-wait
}
// === Critical Section ===
lock = false;
```

💼 **Real Example:**

Used in:

- OS kernel locks
- Device drivers
- Low-level libraries (glibc, pthread, etc.)

✅ **Pros:**

- Fast and **atomic**
- Simple to implement

❌ **Cons:**

- **Busy waiting** = waste CPU
- **Starvation** (some threads may never enter)

🔧 **2. Compare-and-Swap (CAS)**

✅ **Concept:**

Atomically compares a memory value to an expected value.
If matched → replaces it. Else → does nothing.

📜 **Pseudo-code:**

```
boolean CompareAndSwap(int *addr, int expected, int new_val) {
   if (*addr == expected) {
      *addr = new_val;
      return true;
   }
   return false;
}
```

🛠️ **Use Case:**

Lock-free data structures (queues, stacks, etc.)

🧠 **Real Example:**

```
// Lock example
while (!CompareAndSwap(&lock, 0, 1)) {
   // Retry (spin)
}
```

✅ **Pros:**

- **No starvation** (if used properly)
- Foundation of **lock-free programming**

❌ **Cons:**

- Still has busy-wait
- Complex logic

🔧 **3. Disabling Interrupts**

✅ **Concept:**

- Temporarily **disable CPU interrupts** so no other process preempts.
- Used only in **uniprocessor systems**

```
disableInterrupts();
// critical section
enableInterrupts();
```

✅ **Pros:**

- Very fast
- No busy waiting

❌ **Cons:**

- Unsafe in multiprocessors (other cores still run!)
- Can cause **system freeze** if not re-enabled

📊 **Summary Table**

Method	Atomic?	Waits?	Multiprocessor Safe?	Used Today?
Test-and-Set	✅	Yes (spin)	✅	✅ (Kernel)
Compare-and-Swap	✅	Yes (spin)	✅	✅ ✅ (Locks)
Disable Interrupts	❌	No	❌ (unsafe in SMP)	❌ (except embedded)

🧠 **Industry Use**

Used In	Method
Linux Kernel	Test-and-Set, spinlocks
Java, Go, Rust	CAS in Atomic Libraries
Databases (MySQL)	CAS-based memory allocators

Used In	Method
Embedded Systems	Interrupt disable (carefully)

🔹 **Topic 5: Semaphores & Mutexes**

🎓 Must-Know for Exams ✅
💼 Industry Core (Every backend, OS, multithreaded system) ✅ ✅ ✅

✅ **What is a Semaphore?**

A **Semaphore** is a **special integer variable** used to control access to a shared resource in a **multi-process or multi-threaded** system.

Invented by **Edsger Dijkstra** 🔧

🔁 **Semaphore Operations:**

There are **only 2 atomic operations**:

1. wait() (aka P(), down())

```
wait(S):
  while S <= 0:
    wait
  S = S - 1
```

2. signal() (aka V(), up())

```
signal(S):
  S = S + 1
```

🔸 **How it works:**

- **wait()** → blocks if no resource available
- **signal()** → releases resource after use

🧠 **Analogy: ATM Booth**

Only 1 person at a time:

- Before entering: wait(S)
- After leaving: signal(S)

🔁 Types of Semaphores

Type	Value Range	Use Case
Binary Semaphore	0 or 1	Like a mutex (lock/unlock)
Counting Semaphore	0 to n	Multiple identical resources (like printers, DB connections)

🔄 Example 1: Binary Semaphore (Mutual Exclusion)

Semaphore S = 1;

Process P1:

wait(S);

// Critical Section

signal(S);

Process P2:

wait(S);

// Critical Section

signal(S);

✅ Ensures **only one process** in critical section at a time

🔄 Example 2: Counting Semaphore

Semaphore printer = 3;

```
wait(printer);
// Use printer
signal(printer);
```

✅ Allows **3 processes** to use printers at a time

🔁 **Important Note:**

- If S < 0, the process **waits**
- Semaphores can be **busy-waiting** or **blocking** (depends on OS)
- OS manages **queue of blocked processes**

🔐 **Mutex vs Semaphore**

Feature	Semaphore	Mutex
Value Type	Integer (0 to N)	Binary (0 or 1 only)
Ownership	Not owned	Owned by thread
Type	Signaling mechanism	Lock mechanism
Usage	Sync multiple threads	Exclusive locking

Mutex is often implemented using **binary semaphore + ownership**

👷 **Used In:**

Use Case	Tool
Thread-safe databases	Semaphore
Linux/Unix systems	Semaphore
POSIX APIs (C/C++)	Mutex
Java, Python, Go, Rust etc.	Mutex/Semaphore classes
DB Connection Pools	Counting Semaphore

🎓 Classic Questions

Q: Why use semaphore?

To ensure **mutual exclusion** and **process synchronization**

Q: Difference between binary and counting semaphore?

Binary = 0/1, like a lock; Counting = allows n resources

Q: Semaphore vs Mutex?

Mutex has **ownership**, semaphore doesn't

🔹 Topic 6: Classical Synchronization Problems

🎓 Must for Exams + Viva ✅
💼 Asked in OS, Backend, System Design Interviews ✅ ✅ ✅

🔶 Problem 1: Producer–Consumer Problem

Also called: **Bounded Buffer Problem**

✅ Problem Statement

- One or more **Producers** → produce data and place in a **buffer**
- One or more **Consumers** → remove data from the buffer and use it
- **Shared resource** = buffer (with limited size)

⚠️ What We Must Ensure:

Rule	Why?
Buffer never overflows	Producer should **wait** if buffer full
Buffer never underflows	Consumer should **wait** if buffer empty
Mutual exclusion	Only one can access buffer at a time

🔧 Semaphores Used:

Semaphore mutex = 1; // for mutual exclusion

Semaphore full = 0; // count of filled slots

Semaphore empty = n; // count of empty slots

🔁 **Producer Code:**

wait(empty);

wait(mutex);

// Add item to buffer

signal(mutex);

signal(full);

🔁 **Consumer Code:**

wait(full);

wait(mutex);

// Remove item from buffer

signal(mutex);

signal(empty);

✅ **Industry Example:**

- Kafka, RabbitMQ → Producer-consumer queues
- Logging systems
- Upload/download managers

🔶 **Problem 2: Reader–Writer Problem**

✅ **Problem Statement**

- Multiple **readers** can read **simultaneously**
- **Writers** need **exclusive access**
- Shared resource = file or database

⚠️ **Conditions:**

Operation	Can do simultaneously?
Reader + Reader	✅
Writer + Writer	❌
Reader + Writer	❌

🔧 **Semaphores:**

```
Semaphore mutex = 1;     // protects readCount
Semaphore wrt = 1;       // controls write access
int readCount = 0;
```

🔁 **Reader Code:**

```
wait(mutex);
readCount++;
if (readCount == 1)
   wait(wrt);
signal(mutex);

// Reading...

wait(mutex);
readCount--;
if (readCount == 0)
   signal(wrt);
signal(mutex);
```

🔁 **Writer Code:**

```
wait(wrt);
// Writing...
signal(wrt);
```

🧠 **Problem Variants:**

- **Reader Priority** – readers don't wait
- **Writer Priority** – readers wait if writer is waiting
- **Fair Scheduling** – avoid starvation

✅ **Industry Example:**

- Database systems
- File systems (read logs while another thread writes to it)

🔶 **Problem 3: Dining Philosophers Problem**

✅ **Problem Statement**

- 5 Philosophers sit around a table
- Each has a plate & a fork on left and right
- To eat → needs **both forks**
- After eating → release forks

⚠️ **Risks:**

- **Deadlock**: All pick left fork, none get right
- **Starvation**: One philosopher never eats

🔧 **Semaphore Representation:**

Semaphore fork[5] = {1, 1, 1, 1, 1};

🔁 **Naive Code:**

wait(fork[i]);

wait(fork[(i+1)%5]);

// Eat

signal(fork[i]);

signal(fork[(i+1)%5]);

⚠️ This can **deadlock**!

✅ **Solutions:**

Method	**How it fixes**
Limit to 4 philosophers	At most 4 can try at once
Pick forks in different order	One philosopher picks right first
Use waiter (monitor)	Ask permission to eat

✅ **Industry Example:**

- Resource allocation in OS
- Thread pool or DB connection pool
- Avoiding circular wait (deadlock)

✅ **Topic 6 Summary:**

Problem	**Learn for…**	**Key Concept**
Producer–Consumer	OS, Threads, Queues	Buffer management
Reader–Writer	File/DB concurrency	Read-write fairness
Dining Philosophers	Deadlocks, resources	Circular wait

🔹 **Topic 7: Deadlocks**

🎓 Theory + Diagrams ✅
💼 Required for OS, DB, Backend, and Distributed Systems ✅ ✅ ✅

🔴 **What is a Deadlock?**

A **deadlock** is a situation where a set of processes are **blocked forever**, each waiting for a resource **held by the other**.

No process can proceed → system freezes.

🧠 **Real-Life Analogy:**

Imagine 2 people holding one shoe each, waiting for the other to give up theirs to walk.
Result: Both stuck. 🥸

🎯 **4 Necessary Conditions for Deadlock (💯 exam question)**

A deadlock **can happen only if all 4 exist**:

Condition	Meaning
1. Mutual Exclusion	At least one non-shareable resource
2. Hold and Wait	Holding one, waiting for another
3. No Preemption	Can't forcibly take resource
4. Circular Wait	Cycle of processes waiting

📊 **Diagram Example (Deadlock Cycle):**

P1 → waits for R1 held by P2

P2 → waits for R2 held by P1

⇒ Circular wait → 🔁 Deadlock

🧰 Deadlock Handling Methods

Method	Description
1. **Prevention**	Eliminate any of the 4 conditions
2. **Avoidance**	Use safe state checking (Banker's Algorithm)
3. **Detection**	Let it happen, then detect and recover
4. **Ignore**	Assume deadlocks are rare (used in real OS like Linux)

🔧 1. Deadlock Prevention

Remove one of the 4 conditions:

Condition	Prevention Strategy
Mutual Exclusion	Make all resources shareable (not always possible)
Hold and Wait	Request all at once (wastes resources)
No Preemption	Forcefully take resource if needed
Circular Wait	Order all resources → request in increasing order only

---

🔧 2. Deadlock Avoidance

Use **Banker's Algorithm** 🏦
Works like a bank — only gives loan if it can guarantee you'll return it safely.

📦 Data Structures:

Name	Description
Available[]	Resources currently free
Max[][]	Max demand of each process
Allocation[][]	Resources currently held

Name	Description
Need[][]	Max - Allocation

✅ **Safe State:**

A system is in a **safe state** if there is at least one **sequence** of processes that can finish **without deadlock**.

💉 **Steps of Banker's Algorithm:**

1. Find a process Pi such that:
   Need[i] <= Available
2. Pretend to allocate resources
3. Check if it leads to safe state
4. Repeat until all processes finish or stuck

🧠 **Example (Simple):**

Given:

- Available = [3 3 2]
- Max = [[7 5 3], [3 2 2], [9 0 2], [2 2 2], [4 3 3]]
- Allocation = [[0 1 0], [2 0 0], [3 0 2], [2 1 1], [0 0 2]]

→ Check if system is in safe state

(Skip here unless needed now — we'll do examples if you want later)

🔧 **3. Deadlock Detection & Recovery**

Let deadlock happen, then:

- Detect it using **wait-for graph**
- Kill a process or rollback

Used in:

- Databases (SQL Server, Oracle)

- Distributed systems

🔧 4. Ignore Deadlock (Ostrich Algorithm 🐓)

"Bury your head in the sand"
OS like Linux simply **ignores** deadlock risk to keep performance high.

✅ Summary Table

Strategy	Safe?	Realistic?	Used In?
Prevention	✅	Sometimes	Embedded systems
Avoidance	✅ ✅	Only small systems	Banker's algo
Detection	✅	Yes	DBMS, dist. sys
Ignore	❌	Yes	Linux, Windows

✅ Deadlock Topic Complete! 🎉

You now understand:

- What is deadlock
- Conditions required
- How to prevent, avoid, detect
- Banker's Algorithm basics

📘 Unit 4 – Memory Management

🔹 Topic 1: Memory Management Basics & Allocation Techniques

✅ What is Memory Management?

The **Operating System's technique** for managing **RAM** (Main Memory), so that multiple processes can run efficiently without interfering with each other.

🎯 Main Goals:

Goal	Why it's needed
Process Isolation	Prevent one process from touching another
Efficient Allocation	Maximize memory usage
Fast Access	Reduce latency (page faults etc.)
Virtualization	Give each process its own view of memory

🔍 Memory Allocation Techniques

There are 2 major categories:

🔹 1. Contiguous Memory Allocation

Entire process gets one **continuous block** of memory

📦 Types of Contiguous Allocation:

Type	Description
Single Partition	Only one process in memory at a time (very old)
Fixed Partitioning	Memory divided into fixed-size blocks (some wasted)
Dynamic Partitioning	Memory allocated as per process size (causes fragmentation)

⚠️ Problems:

Problem	Meaning
Internal Fragmentation	Wasted space **inside** partition (if process is smaller)
External Fragmentation	Wasted space **between** partitions (not enough to fit next process)

📊 Example: External Fragmentation

+------------+---------+-------------+

| 100 KB | 300 KB | 50 KB free |

+------------+---------+-------------+

Now a 200 KB process can't be loaded, even though total free = 450 KB 😤

📌 Solution: Compaction

Shift memory blocks together to combine free space
(Not always feasible; CPU-heavy operation)

🔹 2. Non-Contiguous Memory Allocation

Process **can be split** across **multiple blocks**

👉 This solves fragmentation issues!

Two main techniques:

1. **Paging** (fixed-size blocks)
2. **Segmentation** (logical division like code/data stack)

We'll cover both in detail in next topics.

✅ Summary Table

Technique	Fixed or Dynamic	Fragmentation Type
Fixed Partition	Fixed	Internal
Dynamic Partition	Dynamic	External
Paging	Fixed-size pages	No External
Segmentation	Logical blocks	External possible

🧠 Industry Insight:

- OS like Linux & Windows use **paging + segmentation hybrid**
- Virtual memory enables non-contiguous allocation
- Compilers & DB engines care deeply about memory layout

Perfect! 🔥 Let's begin one of the most important and asked topics in OS — for both **theory + numericals + real systems**:

📘 Unit 4 – Memory Management

🔹 Topic 2: Paging

🎓 Exam Favorite ✅
💼 Core Concept in Linux, Windows, DBs ✅ ✅ ✅

🧠 Why Paging?

Contiguous memory causes:

- **Fragmentation**
- Wasted space
- Difficult to allocate large memory blocks

📦 **Paging solves this by breaking memory into fixed-size blocks.**

✅ Basic Idea of Paging

Divide memory into **equal-sized fixed blocks**:

- **Logical Memory (Process)** → divided into **Pages**
- **Physical Memory (RAM)** → divided into **Frames**

Page = Frame = Fixed size (e.g., 4KB)

📦 Terminology:

Term	Description
Page	A fixed-size block of **logical memory**
Frame	A fixed-size block of **physical memory**
Page Table	Maps **page number → frame number**

Term	Description
Page Number (p)	Index of page in logical memory
Offset (d)	Distance within a page

📊 **Diagram: Paging System**

```
+------------------+    +------------------+
| Logical Memory   | --> | Page Table       | --> Maps Pages to Frames
+------------------+    +------------------+

Logical Address: [ Page No | Offset ]
                 <---p---> <---d--->

Physical Address: [ Frame No | Offset ]
```

🔍 **Address Translation Example:**

🧪 **Given:**

- Logical Address = 13-bit
- Page size = 1 KB = 2^{10} bytes

→ So:

- **Offset** = 10 bits (because page size = 2^{10})
- **Page Number** = 3 bits (since 13 - 10 = 3)

🎯 **Logical Address = 1010110011010**

Split it:

- Page Number = 101 = Page 5
- Offset = 0110011010 = 410

→ Page Table says Page 5 is in **Frame 3**

👉 Physical Address = Frame 3 + Offset
= 000000000011 + 0110011010

📦 **Page Table**

A data structure that stores **mapping of page numbers to frame numbers**

Page No	Frame No
0	4
1	2
2	7
3	6
...	...

✅ **Advantages of Paging:**

Advantage	Why it's good
No External Fragmentation	Pages can go anywhere in RAM
Easy Swapping	Pages can be moved in/out easily
Simpler allocation	Fixed-size blocks = no compaction needed

❌ **Disadvantages of Paging:**

Disadvantage	What it causes
Internal Fragmentation	Last page might have unused space
Overhead	Page table takes memory
Slower Access	Requires **2 memory accesses** (one for page table, one for data)

→ Solved using **TLB** (Topic coming up next)

🧠 **Real-World Use**

System	Uses Paging
Linux	Uses paging + swap
Windows	Uses paging + virtual memory
JVM, DBs	Use page-based memory
MMU (Hardware)	Handles paging automatically

✅ **Quick Recap**

Concept	You Need to Know
Page & Frame	Fixed-size blocks of logical/physical memory
Page Table	Maps pages to frames
Address Split	[Page No
Internal Fragmentation	Last page may be half-used
No External Fragmentation ✅	

✅ **What is Segmentation?**

Segmentation is a memory management technique where **logical memory** is divided into **variable-sized segments** based on **logical divisions of a program**.

📂 **Real-Life Analogy:**

Your program is like a book:

- Chapter 1 → Code
- Chapter 2 → Data
- Chapter 3 → Stack

Each **chapter (segment)** is a meaningful unit.

🔍 **Why Segmentation?**

- Paging breaks memory into **equal-sized blocks** (not meaningful).
- Segmentation breaks memory into **logical parts**:
    - **Code**
    - **Data**
    - **Stack**
    - **Heap**

🔸 **Segment Table**

The OS maintains a **Segment Table**:

Segment No	Base (Start Addr)	Limit (Length)
0 (Code)	1400	400
1 (Data)	6300	600
2 (Stack)	4300	500

🔹 **Address Format:**

Logical address = <Segment No, Offset>

- **Segment No** → Which segment (Code/Data/Stack)
- **Offset** → Position within that segment

💉 **Address Translation:**

If: Logical Address = <2, 50>

→ Segment 2 has Base = 4300

→ Physical Address = 4300 + 50 = 4350

⚠️ But if offset > limit → **segmentation fault**

✅ **Advantages of Segmentation:**

Feature	Benefit
Logical grouping	Matches how programmers think
Easy protection	Prevent stack accessing code etc.
Flexible memory use	Variable sizes for different needs

❌ Disadvantages:

Problem	Explanation
External Fragmentation	Free space scattered across RAM
Complex Management	Variable sizes = harder to allocate

🧠 Paging vs Segmentation

Feature	Paging	Segmentation
Block Size	Fixed	Variable
Division Basis	Memory	Logical program parts
Fragmentation	Internal	External
Table Name	Page Table	Segment Table
Address Format	Page No + Offset	Segment No + Offset

📊 Real-World Usage:

System/Tool	Uses Segmentation?
Intel x86 CPUs	Yes (Segment Registers: CS, DS, SS)
Compilers	Yes (separates code/data/stack)
Linux	Only minimal segmentation (mostly paging)
C/C++ Stack Frames	Segments in function calls (stack/heap/code)

✅ **Summary:**

Concept	Meaning
Segmentation	Divide memory based on logical parts (code, data, etc.)
Segment Table	Stores base & limit for each segment
Translation	Add offset to base
Error	If offset > limit → segmentation fault

📘 **Unit 4 – Memory Management**

🔹 **Topic 4: Segmented Paging + Introduction to Virtual Memory**

🎓 Conceptual understanding ✅
💼 Used in real operating systems like Linux, x86 ✅ ✅

🧩 **Why Combine Paging & Segmentation?**

- **Paging** solves fragmentation but doesn't respect logical program structure
- **Segmentation** respects logic but causes external fragmentation

👉 So we **combine** both to get:
✅ Logical structure + ✅ Easy memory management

🔁 **Segmented Paging System**

Each **segment** is divided into **pages**, and each page is mapped to physical memory using a **page table**

📂 **How It Works:**

1. Process divided into **segments** (code, data, stack)
2. Each segment divided into **pages**
3. Each segment has its **own page table**
4. Final translation uses:

- Segment table → gives base of that segment's page table
- Page table → gives frame number
- Offset → added to frame to get physical address

🔹 **Logical Address Format:**

[Segment No | Page No | Offset]

- Segment No → which segment (e.g., 0 = code)
- Page No → which page within that segment
- Offset → byte within that page

📊 **Translation Diagram:**

Logical Address → [Seg#, Pg#, Offset]

↓

Segment Table → Get base of that segment's Page Table

↓

Page Table → Get frame number for page

↓

Physical Address = Frame Base + Offset

✅ **Advantages of Segmented Paging:**

Feature	Benefit
Logical Grouping	Code, stack, heap separated
No External Fragmentation	Pages = fixed size
Flexible	Can allocate just-in-time

🧠 **Real Use**

System	Implementation
Intel x86 CPUs	Segmented Paging
Linux, Windows	Flat segments, fully paged

🔹 **Bonus: What is Virtual Memory?**

Virtual Memory = illusion of large memory created using **disk + RAM**

✅ **Key Concepts:**

- You **don't need entire program** in RAM to run it
- Load pages **on demand** = **Demand Paging**
- If page is not in memory → **Page Fault** → fetch from disk

---

📦 **Terms:**

Term	Meaning
Page Table	Maps virtual → physical pages
Backing Store	Secondary storage (HDD/SSD)
Page Fault	Requested page is not in memory
Swap In/Out	Move pages between RAM and disk

📊 **Diagram: Virtual Memory Flow**

[CPU Request → Virtual Address]

↓

[Page Table]

↓

If Page Present → Access it

If Not → Page Fault → Bring it from Disk

💼 **Real Use:**

Feature	Used In
Virtual memory	Every modern OS (Linux, macOS, Windows)
Swapping	If RAM full, move least-used pages to disk
Fast boot	OS loads minimal pages, rest on demand

✅ **Summary**

Topic	Summary
Segmented Paging	Combines logical structure (segmentation) + fixed-size blocks (paging)
Virtual Memory	Lets processes run with more memory than physically available
Page Fault	Triggered when page is not in RAM

🔹 **Topic 5: Virtual Memory + Page Replacement Algorithms**

🎓 Most scoring + numerical topic ✅
💼 Real-world use in all modern OS ✅ ✅ ✅

✅ **What is Virtual Memory?**

A technique where **processes use more memory** than physically available, by using **disk as an extension of RAM**.

📦 **How it works:**

- **Only needed pages** are loaded into RAM
- If needed page not in RAM → **Page Fault**
- OS brings page from **disk (swap)** → may remove old page if RAM is full

---

🔁 **Steps in Page Fault Handling:**

1. CPU tries to access a page → not in memory
2. OS suspends process
3. Finds the page on disk
4. If RAM full → **replace** a page
5. Load new page into memory
6. Update page table
7. Resume process

📉 **Performance Issue: Thrashing**

Too many page faults = CPU spends more time **swapping** than executing → system slows to crawl

🧠 Prevented by:

- Good replacement algorithm
- Increasing RAM
- Locality of reference

🧮 **Page Replacement Algorithms**

When page needs to be replaced, OS uses one of these:

🔹 **1. FIFO (First-In First-Out)**

- Replace the **oldest loaded page**

📊 **Example:**

Frames = 3
Page stream: 7, 0, 1, 2, 0, 3, 0, 4

7 → miss

0 → miss

1 → miss

[7,0,1]

2 → replace 7

[2,0,1]

0 → hit

3 → replace 0

[2,3,1]

0 → replace 1

[2,3,0]

4 → replace 2

[4,3,0]

✅ Simple
❌ Replaces important pages blindly

🔹 **2. Optimal (OPT)**

- Replace page that will **not be used for the longest time** in future

→ Ideal (used for comparison only)

🔹 **3. LRU (Least Recently Used)**

- Replace page that was **least recently used**

🧠 Based on **past** use, not future

🧱 **LRU Example:**

Frames = 3
Stream: 7, 0, 1, 2, 0, 3, 0, 4

Use timestamps or stack to track usage

✅ Better than FIFO

❌ Harder to implement efficiently

🔹 **4. Second Chance (Clock Algorithm)**

- Improves FIFO using a **reference bit**

🕒 Like a clock hand — skip pages with ref = 1, give second chance

✅ Used in Linux

📊 **Comparison Table**

Algorithm	Strategy	Pros	Cons
FIFO	Oldest out	Simple	Bad decisions
Optimal	Furthest future	Best performance	Not possible
LRU	Least recent access	Practical & good	Needs tracking
Second Chance	FIFO + reference bit	Real-world usable	Moderate perf

---

💻 **Industry Usage**

OS/DBMS	Uses
Linux, Windows	Clock / LRU
PostgreSQL	LRU / Variants
JVM, Python	LRU cache for memory objects
Redis / Caches	LRU, LFU

✅ **Summary**

Concept	What You Learned
Virtual Memory	Uses disk when RAM full
Page Fault	When page not in memory

Concept	What You Learned
Page Replacement	FIFO, LRU, Optimal, etc
Thrashing	Too many page faults
Industry Use	All major OS + DBs

📘 **Unit 4 – Memory Management**

🔹 **Topic 6: Thrashing + Working Set Model + Belady's Anomaly**

🎓 Theory + Conceptual Questions ✅
💼 Real-world RAM management insight ✅ ✅

🚨 **What is Thrashing?**

A condition where the **system spends more time handling page faults** than executing actual processes.

🔁 **Cause of Thrashing:**

When:

- Too many processes
- Too few frames (RAM)
- Frequent **page swapping**

🧠 **Analogy:**

You try to read 4 books but your table only fits 2. You keep swapping books every few seconds = **thrashing** 📚🔄

⚠️ **Thrashing Symptoms:**

Symptom	Result
CPU usage drops	Always waiting for disk I/O

Symptom	Result
Page fault rate spikes	More than a threshold
Programs freeze	Slow or no progress

✅ **How to Handle Thrashing:**

Solution	Explanation
Local replacement	Don’t steal frames from others
Working Set Model	Give frames based on process's needs
Increase RAM	Add more physical memory
Suspend processes	Reduce multiprogramming temporarily

🔹 **Working Set Model**

Introduced by **Peter Denning** to avoid thrashing

🧠 **Idea:**

Each process needs a **working set** = set of pages it actively uses.

📦 **Working Set (W) = set of pages used in last Δ (time window)**

- OS tracks which pages are actively referenced
- Allocate at least that many frames
- If total demand > available → suspend processes

✅ **Benefits:**

Feature	Why it helps
Predicts usage	Uses history to allocate wisely
Thrashing control	Stops overload

Feature	Why it helps
Dynamic window (Δ)	Flexible for process types

❗ **Belady's Anomaly**

In **FIFO page replacement**, sometimes **more frames = more page faults!**

😱 **Example:**

Page Stream: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- With 3 frames → 9 faults
- With 4 frames → **10 faults** (more!)

🔍 **Happens only in:**

- **FIFO algorithm**
- Not in LRU or Optimal

📊 **Why Important?**

- Shows that **FIFO is not always good**
- Just increasing memory doesn't always help
- Need **smart replacement**, not just size

✅ **Final Summary – Topic 6**

Concept	Description
Thrashing	CPU stuck in page-fault-replace loop
Working Set Model	Predictive frame allocation strategy
Belady's Anomaly	More frames ≠ fewer page faults (in FIFO)

📘 Unit 5 – File Management

📂 What is a File?

A **file** is a named collection of **related data** stored on **secondary storage** (like SSD, HDD).

🧠 Why Files?

- Data persists even after shutdown (unlike RAM)
- Enables **sharing**, **security**, and **organization** of data
- Used by programs, OS, users

📄 File Attributes

Every file has **metadata** (stored in directory entry):

Attribute	Meaning
Name	Human-readable name (e.g., notes.txt)
Type	.txt, .exe, .jpg, etc.
Size	Bytes used
Location	Disk address / block info
Protection	Read/Write/Execute permissions
Timestamps	Created, modified, last accessed

📊 File Types (by extension or structure)

Type	Extension	Used For
Text File	.txt, .md	Human-readable text
Binary File	.exe, .jpg	Machine-readable only
Program File	.exe, .class	Executable code

Type	Extension	Used For
Multimedia	.mp4, .mp3	Audio/video content
Data File	.csv, .db	Structured data

🔹 **File Structure**

OS sees files in one of these structures:

Structure	Description
Byte sequence	Just raw bytes (most flexible)
Record-based	Fixed/variable-size records
Tree structure	Used in compiled code, object files

🧠 **Industry Note:**

- Linux treats **everything as a file** (even keyboard, printer, memory = files)
- File systems (like **ext4, NTFS, FAT32**) manage how files are stored and accessed

---

📘 **Quick Example:**

Let's say you create a file report.docx
The OS will store:

- File name: report.docx
- Location: Block 1240 → 1245
- Owner: Ghanshyam
- Size: 215 KB
- Created: 16 July 2025
- Type: Microsoft Word document

✅ **Summary**

Concept	Summary
File	Named data on disk
Attributes	Name, size, location, time, permissions
Types	Text, binary, program, media, data
Structure	Byte stream or records
Metadata	Managed by file system

Great, Ghanshyam! 🚀
Let's move to a super practical and important topic: how files are **accessed** in an OS.

📘 **Unit 5 – File Management**

🔹 **Topic 2: File Access Methods**

🎓 Theory + Diagrams ✅
💼 Used in OS, DBMS, file systems ✅ ✅

🔑 **Why Access Methods?**

Different applications need **different ways to read/write data**.
Example:

- Media player → jump to a timestamp (direct access)
- Notepad → read/write line by line (sequential access)

📂 **Main File Access Methods**

Method	Used When…
1. Sequential	Reading files line by line
2. Direct	Jumping to a particular record
3. Indexed	Fast search using an index (like DB)

🔹 **1. Sequential Access**

Read data **in order**, one record after another (like a tape recorder)

📊 **Diagram:**

[Start] → Rec1 → Rec2 → Rec3 → Rec4 → ... → [End]

✅ **Used In:**

- Text files (.txt, .log)
- Compilers reading source code
- Reading sensor data

🧠 **Pros & Cons:**

Pros	Cons
Simple to implement	Can't skip ahead easily
Efficient for full scan	Slow for random access

🔹 **2. Direct Access (a.k.a. Random Access)**

Jump directly to the desired block or record

📊 **Diagram:**

Go directly to → Rec4 → Read it → Back to Rec2

✅ **Used In:**

- Media players (seek to time)
- Database tables (primary key)

- Virtual memory paging

🧠 **Pros & Cons:**

Pros	Cons
Fast random access	Requires fixed-length records
Better performance for large files	Complex to implement

🔹 **3. Indexed Access**

Like a book index — OS maintains a **lookup table** of record locations

---

📊 **Diagram:**

Index:

ID=1001 → Block 20

ID=1002 → Block 45

ID=1003 → Block 32

Access using index!

✅ **Used In:**

- Database management systems
- File systems (ext4, NTFS)
- Indexed video/audio file formats

🧠 **Pros & Cons:**

Pros	Cons
Very fast lookups	Index itself takes space
Efficient for large datasets	Overhead of maintaining index

✅ **Comparison Table**

Feature	Sequential	Direct	Indexed
Access Pattern	Ordered	Any	Based on index
Flexibility	Low	Medium	High
Speed	Slow	Fast	Fastest (if indexed)
Used In	Logs, Text	Media, Memory	DBMS, File Systems

🧠 **Real-World Example:**

Task	Method Used
Playing song from timestamp	Direct
Browsing large DB table	Indexed
Reading config file	Sequential

✅ **Summary**

Concept	Summary
Sequential Access	Record-by-record (slow but simple)
Direct Access	Jump to any record instantly
Indexed Access	Use index for fast access

✅ **Topic 2 Complete!**

🔹 **Topic 3: File Allocation Methods**

🎓 Theory + Diagrams ✅
💼 Core for OS, DBMS, and File System Design ✅ ✅ ✅

📂 **What is File Allocation?**

It's the way the **OS stores blocks of a file** on disk — how the file's content is laid out in memory or storage.

⚠️ **Challenge:**

- Disk has blocks (e.g., 512B each)
- File has many blocks
- **Need to map file name → disk blocks**

🔢 **Three Main Allocation Methods:**

Method	Description
1. Contiguous	Blocks stored one after the other
2. Linked	Each block points to the next
3. Indexed	Index block contains all pointers

Let's explore each 👇

🔹 **1. Contiguous Allocation**

Entire file stored in **continuous blocks**

📊 **Diagram:**

File A → [100][101][102][103][104]

✅ **Pros:**

- Fast access (1 seek)
- Simple to implement

❌ **Cons:**

- Wastes space (hard to find large free blocks)
- File size must be known in advance
- External fragmentation

📘 **Real Use:**

Older file systems (e.g., early FAT), DVDs, CD-ROMs

🔹 **2. Linked Allocation**

Each block stores a **pointer to the next block**

📊 **Diagram:**

File B → [109] → [245] → [378] → NULL

🧠 Each block = data + pointer

✅ **Pros:**

- No fragmentation
- File can grow dynamically

❌ **Cons:**

- Slow for random access
- More disk I/O (read block + pointer)
- Pointer takes space inside block

📘 **Real Use:**

Used in FAT (File Allocation Table) system
Also used in memory (linked lists)

🔹 3. Indexed Allocation

A special **index block** stores all addresses of file's blocks

---

📊 Diagram:

Index Block → [103, 505, 210, 331]

→ Each entry points to actual data block

✅ Pros:

- Fast random access
- No external fragmentation
- Easy to grow file

❌ Cons:

- Index block size is limited
- Small files waste index space

📘 Real Use:

- UNIX inode system
- ext2/ext3/ext4
- NTFS uses Master File Table (MFT)

✅ Comparison Table

Feature	Contiguous	Linked	Indexed
Access Time	Fast	Slow	Medium–Fast
File Growth	Hard	Easy	Easy

Feature	Contiguous	Linked	Indexed
Random Access	Good	Bad	Good
Fragmentation	External	None	None
Complexity	Low	Medium	High

✅ **Summary**

Allocation Method	Use Case
Contiguous	Fast read, fixed size (e.g., ISO files)
Linked	Sequential files (e.g., logs, archive)
Indexed	Large, flexible files (e.g., OS, DB)

📘 **Unit 5 – File Management**

🔹 **Topic 4: Directory Structures & File Protection**

🎓 Theory + Diagrams ✅
💼 Used in Linux, Windows, AWS, DBMS ✅ ✅ ✅

📂 **What is a Directory?**

A **directory** is like a folder that stores **file names + metadata**, allowing users and OS to **organize and access files efficiently**.

📁 **Directory Structure Types**

Let's go from **simple to complex** 👇

🔹 **1. Single-Level Directory**

- All files in **one directory**
- **No subfolders**

📊 **Example:**

root/

├— file1.txt

├— file2.c

├— data.csv

✅ Easy to manage
❌ Name conflicts, no grouping

🔹 **2. Two-Level Directory**

- One directory **per user**

📊 **Example:**

root/

├— user1/

| ├— fileA.txt

├— user2/

| ├— fileB.c

✅ Avoids name conflicts
❌ No grouping within a user

🔹 **3. Tree-Structured Directory**

- Allows **hierarchical folders** (like Windows, Linux)

📊 **Example:**

root/

├— user1/

| ├— docs/

| ├— img/

├— bin/

├— etc/

✅ Real-world use
✅ Easy navigation, search
✅ Subdirectories

🔹 **4. Acyclic Graph Directory**

- Allows **file sharing** between directories via **links**

📊 **Example:**

user1/report → shared by user2 via symbolic link

✅ File sharing without duplication
❌ Complex (cyclic reference must be avoided)

🔹 **5. General Graph Directory**

- Allows **cycles**, handled via garbage collection or reference counting

❌ Not commonly used due to complexity

🛡️ **File Protection**

Operating systems protect files using **access control mechanisms**.

🔐 **Common Access Types:**

Access Type	What It Means
Read	View file contents
Write	Modify file
Execute	Run program/script
Delete	Remove file

🔹 **Access Control Methods**

---

1. Access Control Matrix (ACM)

A 2D matrix with **users vs. files** storing access rights

	fileA	**fileB**
user1	R, W	R
user2	R	R, W, X

✅ Complete control
❌ Space inefficient for large systems

---

2. Access Control List (ACL)

Each file has a list of **users and their permissions**

fileA →

user1: read, write

user2: read

✅ Used in Linux, cloud systems (AWS S3, GCP)
✅ Easier to manage per-file

---

3. User Classes (UNIX-style)

Permissions are defined for:

- **Owner**
- **Group**
- **Others**

Example:

chmod 755 script.sh

rwxr-xr-x

✅ Simple
✅ Efficient for multi-user environments

🧠 **Real-World Use:**

System	Protection Used
Linux	ACL + chmod
Windows	NTFS permissions + ACL
AWS/GCP	IAM roles, bucket policies
DBMS	Role-based Access Control

✅ **Summary Table**

Feature	Description
Directory Structures	Single, Two-Level, Tree, Acyclic
File Protection	Prevents unauthorized access
ACL	Per-file user access list
ACM	Matrix of users × files
UNIX Permissions	Owner, Group, Others

📘 **Unit 5 – File Management**

🔹 **Topic 5: File System Implementation Concepts**

🎓 Theory + Technical insight ✅
💼 Used in ext4, NTFS, FAT, cloud storage ✅ ✅ ✅

🧱 **Components of a File System**

To manage files, a file system must:

1. Store the file data (blocks)
2. Track **free space**
3. Keep **metadata** for each file
4. Translate file → physical disk locations

📂 Key Data Structures

Structure	Purpose
File Control Block (FCB)	Stores file metadata
Directory Entry	Maps name → FCB
Allocation Table	Tracks block usage (FAT, ext)
Free Space Structure	Shows which blocks are free

🔸 File Control Block (FCB)

Contains all info needed to manage a file

Attribute	Example
File name	data.txt
Size	124 KB
Location	Blocks: 340, 341, 342
Owner	ghanshyam
Permissions	rw-r--r--
Timestamps	Created, Modified

🗂️ Directory Implementation

Each directory stores:

- File names
- Link to FCB (inode in Linux)

📊 Directory Entry Table:

File Name	Inode No / FCB
hello.c	011

File Name Inode No / FCB

image.jpg 158

🔍 Free Space Management

OS must know: which disk blocks are available for new files?

🔹 1. Bitmap (Bit Vector)

Each block is represented by **1 bit**

- 1 = Used, 0 = Free

Example:

Blocks: [0][1][2][3][4][5]

Bitmap: 1 0 0 1 0 1

✅ Fast lookup
❌ Needs RAM space for large disks

🔹 2. Free Block List

Store addresses of all free blocks in a **linked list**

✅ Easy to implement
❌ Slow allocation (must scan)

🔹 3. Grouping or Counting

Store **number of free blocks in a row**

Example:

Block 40 → (start = 100, count = 10) → free blocks from 100 to 109

✅ Efficient for large filesystems
✅ Used in modern file systems

📂 Mounting a File System

Attaching a file system (like a USB or another partition) to the system's directory tree

mount /dev/sdb1 /mnt/usb

🧷 **Need Mount Table:**

Stores:

- Mounted device
- Mount point
- File system type

💡 **Example from Real Systems:**

File System	Uses
ext4	Bitmaps + Inodes + Journaling
NTFS	MFT (Master File Table), ACL
FAT32	Linked allocation + FAT
AWS S3	Key-Value + Metadata index

✅ **Summary**

Concept	Summary
FCB (Inode)	Stores file metadata
Free Space Management	Bitmap, List, Counting
Directory Structure	Maps names to inodes
Mounting	Attaching FS to tree
Allocation Table	Tracks file block layout

✅ **Topic 5 Complete!** 🎉