UNIT – 1 – OPERATING SYSTEM & LINUX

In the real world, **software does not run in isolation**. Every AI model, web application, research experiment, cloud service, or mobile app runs **on top of an operating system**. If an application is **slow**, **crashes**, **cannot scale**, **leaks data** or **fails under load**, the root cause is often **how the OS resources are used**, not just bad code.

For example:

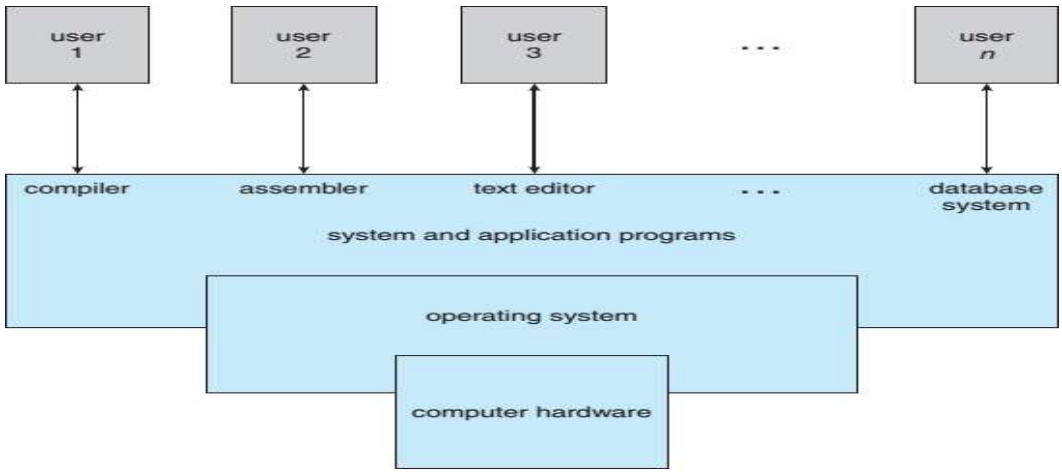
- An AI training job failing → memory management issue
- Server hanging → process scheduling or I/O issue
- Model outputs exposed → OS security & permissions issue

Operating System – Definition & Objectives

What is an Operating System?

An **Operating System (OS)** is system software that controls computer hardware and provides a **safe and efficient environment** for applications to run.

An OS sits between the **user and the hardware**. Its main role is to manage hardware resources and allow applications to run smoothly without dealing with low-level details.



The OS works like a **manager**. Decides who uses which resource and when, **how CPU time, memory, disk space, and devices are shared** among programs and users.

When you run an application such as a browser, an AI model, or a Python program, you are not directly using the hardware. The OS receives your request and handles all hardware interactions internally.

Why an Operating System is Needed?

Without an operating system, a computer cannot work in a stable or controlled way.

If there were no OS:

- Multiple programs would try to use the CPU at the same time, causing conflicts.
- One program could overwrite another program's memory.
- Files stored on disk could become corrupted or lost.
- A single faulty program could crash the entire system.

The operating system prevents these problems by **enforcing rules and managing resources carefully**.

Objectives of an Operating System

The operating system is designed with clear goals that explain what it tries to achieve in real systems.

1. Convenience: Running python train.py without worrying about CPU or memory.

2. Efficient Use of Resources: Multiple AI jobs running on a server without slowing each other down.

3. Resource Sharing and Management: CPU time and RAM are divided among multiple running applications.

4. Reliability and Stability: One application crash, but others continue running normally.

5. Security and Ethical Use: Only authorized users can access sensitive files or datasets.

Operating System Services

How Applications Depend on the Operating System?

Operating system services are layered and interconnected. Each service builds on the previous one to support safe and efficient application execution.

Why Operating System Services Are Needed?

Modern applications assume that the operating system will handle: Resource allocation, Hardware control, Safety and isolation, Security and permissions.

Applications focus on **what to do**, while the OS decides **how and when resources are used**. This separation makes systems reliable, scalable, and easier to develop.

1. Program Execution Service: Running python train.py where the OS loads the program, assigns CPU, and ends it safely.

2. Process Management Service: A server runs background services, user applications, and AI jobs at the same time.

3. Memory Management Service: Large AI models use virtual memory when RAM is not sufficient.

4. File System Service: Saving datasets, model files, and logs securely on disk.

5. Input/Output (I/O) Management Service: Reading data from disk or sending output to a GPU without knowing device details.

6. Communication Service: Client–server applications and distributed systems exchanging data.

7. Protection and Security Service: Only permitted users can access sensitive files or processes.

User View vs System View of an Operating System

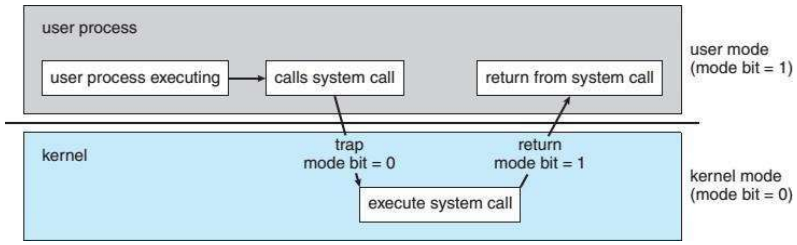
An operating system can be understood from two perspectives: **user view** and **system view**. These are safely separated using the **mode bit**.

User View: From the user view, the OS focuses on ease of use and responsiveness by hiding hardware and system internals.

Example: Opening a browser without worrying about CPU scheduling or memory allocation.

System View: From the system view, the OS acts as a resource manager that controls CPU, memory, and I/O devices to ensure stability and fairness.

Example: A cloud server sharing resources among many applications.



Mode Bit: The mode bit is a hardware flag that indicates the current execution mode.

- **User Mode:** Applications run with restricted access.
- **Kernel Mode:** The OS runs with full access to hardware.

Example: A file read request starts in user mode, switches to kernel mode for disk access, and returns to user mode.

Real-World Example: A file read request starts in user mode, switches to kernel mode for disk access, and returns to user mode.

Scenario: The user reads a file easily while the OS safely controls the disk operation.

Operating System Operations

The operating system continuously performs core operations while the system is running.. *Scenario:* Multiple applications run smoothly even though users do not see OS activities in the background.

Why OS Operations Are Important?

OS operations coordinate multiple applications, users, and background services to keep the system stable and usable. *Scenario:* Without OS operations, a system running many programs would become slow or crash.

Process Operations: Deciding which application gets CPU time.

Memory Operations: Using virtual memory when physical RAM is limited.

File Operations: Storing configuration files and logs.

Input/Output (I/O) Operations: Sending data to a GPU or reading from disk.

Protection and Security Operations: Preventing unauthorized access to sensitive files.

Error Detection and Handling: Handling hardware faults without crashing the system.

Real-World Perspective

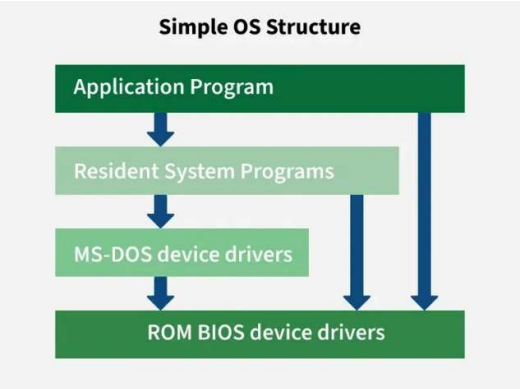
On a cloud server running web services, databases, and AI models, the OS continuously performs process, memory, file, and I/O operations to keep everything running smoothly. *Scenario:* A small failure in OS operations can slow down services or cause system crashes.

Operating System Structures

Operating system structures did not appear all at once. They **evolved over time** as systems became more complex and new problems emerged. Each structure was designed to solve the **limitations of the previous one**. Understanding this evolution helps explain **why modern operating systems look the way they do today**.

1. Simple Structure (Early Systems)

All OS components are placed together with no clear separation.



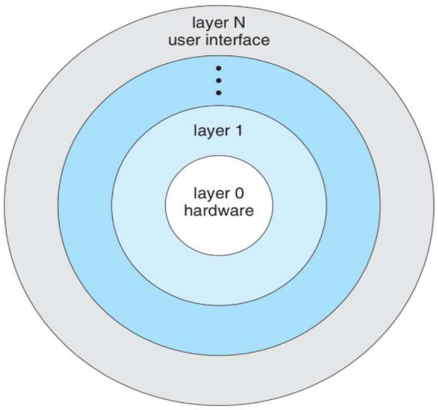
Example: MS-DOS.

Problem: If any part of the OS failed, the entire system crashed. There was no protection or isolation.

2. Layered Architecture (Need for Organization)

The OS is divided into layers, each performing a specific function. *Advantage:* Easier design and maintenance.

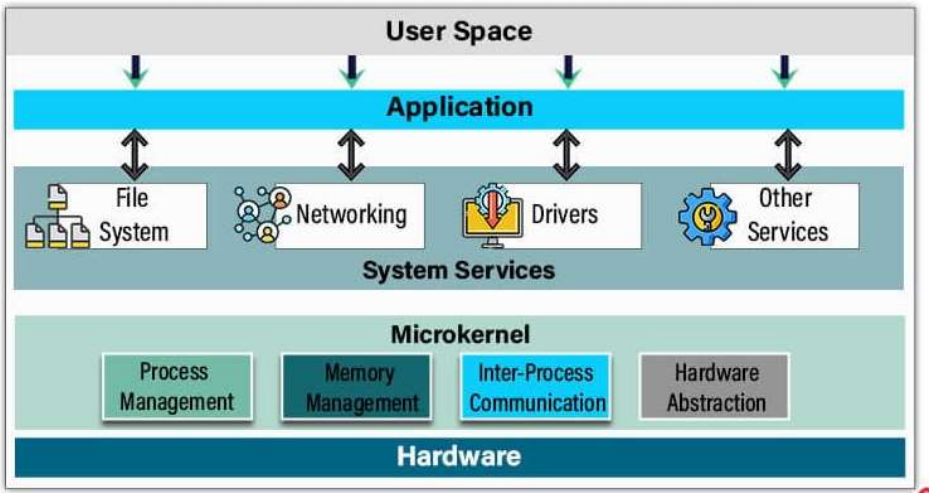
Limitation: Performance overhead.



3. Microkernel Structure (Need for Security & Reliability)

Only essential services run in kernel mode; others run in user space.

Example: Minix, QNX.



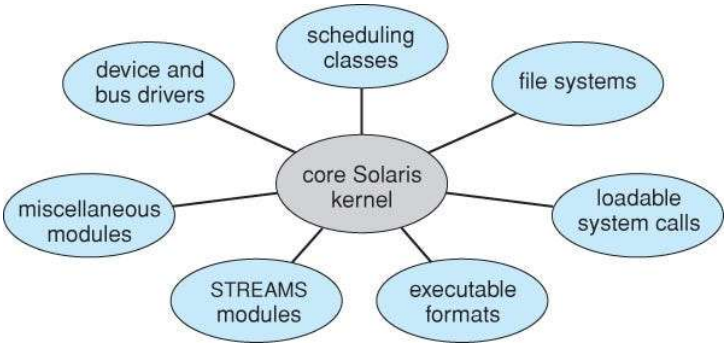
Why it evolved: Improved security, better fault isolation

Limitation: Increased communication overhead between user space and kernel.

4. Modular Structure (Need for Flexibility)

The OS has a core kernel with dynamically loadable modules.

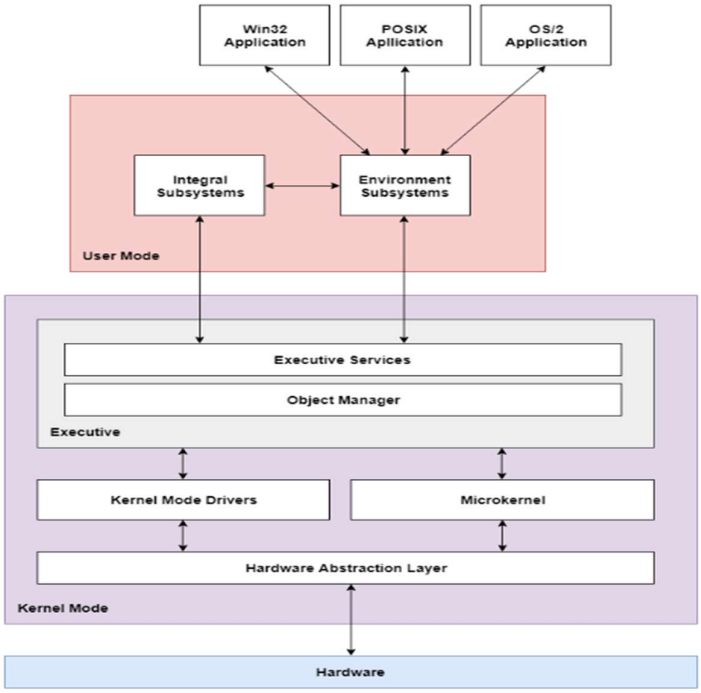
Example: Linux.



Why it evolved: Support for dynamic hardware, better performance than microkernels

5. Hybrid Structure (Modern Practical Solution)

- Combines performance and security features of different designs.
- *Example:* Windows, macOS.



Why it evolved: Real-world systems need speed, security, and scalability together

Linux Architecture

Linux architecture did not appear by accident. It evolved to support **multi-user systems, scalability, performance, and openness**. Understanding its architecture helps explain **why Linux dominates servers, cloud, and AI platforms** today.

Why Linux Architecture Was Needed?

Early operating systems were: Tightly coupled to hardware, Difficult to modify, Limited in scalability.

As systems grew larger and more complex, there was a need for: Clear separation of responsibilities, better resource management, support for multiple users and applications.

Linux was designed to meet these needs with a **layered yet flexible architecture**.

Linux Architecture – Layer by Layer Evolution

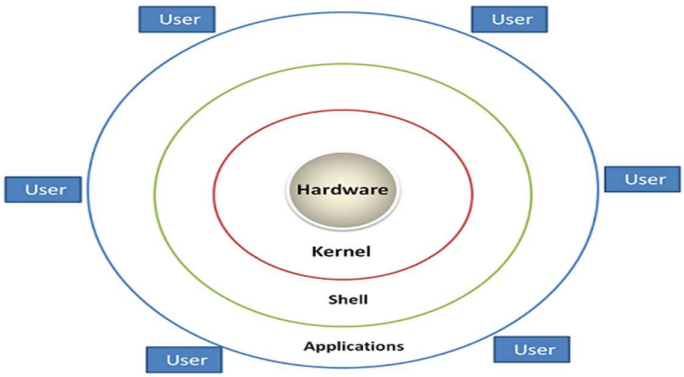
Hardware: CPU, memory, disk, network, and I/O devices.

Kernel: Manages processes, memory, devices, file systems, and security.

System Call Interface: Provides controlled access from applications to the kernel.

Shell: Accepts user commands and interacts with the kernel. *Examples:* bash, sh.

User Applications: Programs that run-in user mode and request OS services. *Example:* AI frameworks using CPU/GPU via the kernel.



Linux System Architecture

How Linux Architecture Supports Modern Systems?

Linux architecture enables: Multi-user access, Multi-tasking, High performance, Security and isolation, Scalability

This is why Linux is used in: Cloud servers, Supercomputers, AI research labs, Embedded systems

Linux architecture separates hardware control, system management, and user interaction into clear layers. This design provides security, flexibility, and scalability, making Linux suitable for modern computing environments.

Introduction to Linux Commands

Linux commands allow users to interact with the operating system through the terminal. These commands are used to navigate the file system, manage files and directories, control users, view system information, and manage compressed files. They are widely used in servers, cloud systems, DevOps, and AI environments.

Basic Linux Commands

PATH: PATH specifies the directories where the OS searches for executable commands. *Example:* Allows running python from any directory without giving the full path.

```
echo $PATH

export PATH=$PATH:/new/directory/path
```

man: Displays the manual page of a command. *Example:* Used to understand command usage and options.

```
man ls
```

echo: Prints text or variable values to the terminal. *Example:* Displaying environment variables.

```
echo "Hello"
echo $PATH
```

printf: Prints formatted output. *Example:* Used in shell scripts for formatted display.

```
printf "Value is %d\n" 10
```

script: Records a terminal session into a file. *Example:* Saving command execution during lab sessions.

```
script output.txt
```

passwd: Changes the password of the current user. *Example:* Used for user security.

```
passwd
```

uname: Displays system and kernel information. *Example:* Checking OS and kernel version.

```
uname -a
```

who: Shows users currently logged into the system. *Example:* Used in multi-user environments.

```
who
```

date: Displays the current system date and time. *Example:* Logging system time.

```
date
```

pwd: Displays the present working directory. *Example:* Knowing the current location in the file system.

```
pwd
```

cd: Changes the current directory. *Example:* Navigating between folders.

```
cd Documents
cd ..
```

ls: Lists files and directories. Common options: ls -l → detailed listing, ls -a → includes hidden files

```
ls
```

touch: Creates an empty file or updates the timestamp of a file. *Example:* Creating a file before editing.

```
touch file.txt
```

mv: Moves or renames files and directories. *Example:* Renaming a file.

```
mv old.txt new.txt
```

rm: Deletes files or directories permanently. *Example:* Removing unwanted files.

```
rm file.txt
rm -r folder
```

mkdir: Creates a new directory. *Example:* Creating folders for organizing files.

```
mkdir project
```

rmdir: Deletes an empty directory. *Example:* Removing unused empty folders.

```
rmdir oldfolder
```

cat: Displays the contents of a file. *Example:* Viewing text files.

```
cat file.txt
```

tar: Archives multiple files into a single file. *Example:* Packaging files before transfer.

```
tar -cvf files.tar folder/
```

gzip: Compresses files to reduce size. *Example:* Compressing archive files.

```
gzip files.tar
```

UNIT – 2 – PROCESS & CPU SCHEDULING

In laptops, servers, cloud platforms, and AI systems, **many tasks run at the same time** but **CPU resources are limited**. The operating system must:

- Decide **what runs now**, **what waits**, **how long each task runs**

Processes, threads, and CPU scheduling are the mechanisms that make **multitasking, fairness, and performance** possible.

Program vs Process

A **program** is a file stored on disk. A **process** is a program that is loaded into memory and executed by the CPU.

In real systems, the OS does not manage programs directly. It manages **processes**, because only processes consume CPU and memory.

What is a Process?

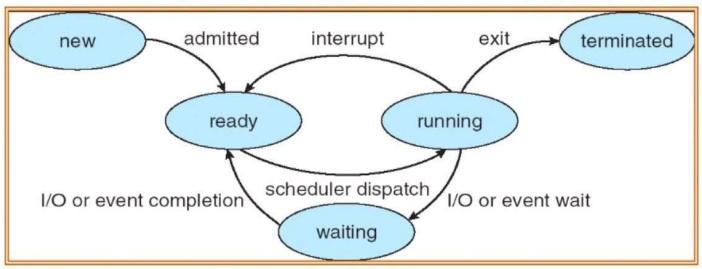
A **process** includes: Program code, Program counter, CPU registers, Stack & heap memory, Process ID (PID).

This abstraction allows the OS to **pause, resume, and schedule execution**, which is essential when many applications run together.

Process States

Processes move through states: **New**, **Ready**, **Running**, **Waiting**, **Terminated**

This explains why an application may wait even though the system is not frozen—the CPU is simply executing another process.



Operations on Processes

The OS performs: Process creation, CPU scheduling, Context switching, Process termination

These operations allow smooth multitasking without user intervention.

THREADS (WHY PROCESSES ALONE ARE NOT ENOUGH)

What is a Thread?

A **thread** is the smallest unit of execution inside a process.

Threads share memory but execute independently. In real systems, threads are used to **increase performance and responsiveness**, especially in servers and AI workloads.

Process vs Thread (Real System Reason)

Processes are heavyweight and isolated. Threads are lightweight and share memory.

This is why: Web servers use threads to handle many users, AI pipelines use threads for parallel data loading

Process Control Block (PCB)

What Is a PCB?

A **Process Control Block (PCB)** is a **data structure** used by the operating system to **store all information about a process**.

The OS does not manage processes directly — it manages **PCBs**.

What Information Does PCB Store?

A PCB typically contains: Process ID (PID), Process state (new, ready, running, waiting, terminated), Program counter, CPU registers, Memory management information, Scheduling information, I/O status

Why PCB Is Needed

PCB allows the OS to: Pause a process, Resume a process, Switch CPU between processes, Track process status. Without PCB, **multitasking is impossible**.

PCB is like a **file for each employee** in a company:

- Contains role, status, work details
- Manager (OS) uses it to assign work

Context Switching

What Is Context Switching?

Context switching is the process of **saving the state of the currently running process** and **loading the state of another process** so the CPU can switch execution.

Why Context Switching Is Required

Only one process can use the CPU at a time (per core). To support multitasking, the OS rapidly switches between processes.

How Context Switching Happens

→ Running Process A

→ Save its state into PCB

→ Load Process B state from PCB

→ CPU starts executing Process B

What Is Saved During Context Switch?

- Program counter → CPU registers → Process state → Memory information

When you:

- Switch between apps
- Play music while coding
- Run background downloads

Context switching makes it possible.

Zombie Process

What Is a Zombie Process?

A **zombie process** is a process that: Has finished execution but still has an entry in the process table. It is **dead but not fully removed**.

Why Zombie Process Occurs?

Zombie process occurs when: Child process finishes and Parent process does NOT call wait()

The OS keeps the child's exit status until the parent collects it.

→ Parent creates child

→ Child finishes execution

→ Parent does not wait

→ Child becomes zombie

Why Zombie Processes Are a Problem?

- Consume process table entries and Too many zombies can exhaust system resources

Orphan Process

What Is an Orphan Process?

An **orphan process** is a process whose: Parent process has terminated and Child is still running

What Happens to Orphan Processes?

The OS assigns orphan processes to: init process (PID 1), init takes responsibility and cleans them up.

→ Parent exits

→ Child still running

→ Child becomes orphan

→ init adopts it

Why Orphan Processes Are NOT Dangerous

- OS handles them automatically, they do not cause resource leaks

Zombie Process	Orphan Process
Child finished	Child still running
Parent alive	Parent terminated
Needs wait ()	Handled by init
Harmful if many	Usually, safe

CPU SCHEDULING (RESOURCE FAIRNESS)

Why CPU Scheduling Exists

Only one process can use the CPU at a time (per core). Scheduling ensures:

- Fairness
- Responsiveness
- Efficient CPU usage

Scheduling Criteria

Arrival Time (AT)

Time when process enters ready queue

Burst Time (BT)

CPU execution time required

Finish Time (FT)

Time when process finishes execution

Turnaround Time (TAT)

TAT = FT − AT

Waiting Time (WT)

WT = TAT − BT

Response Time (RT)

RT = First CPU Start Time − AT

CPU SCHEDULING ALGORITHMS

FCFS (First Come First Serve)

Processes execute in arrival order. Simple but unfair—long jobs block short ones.

FCFS Example

Tasks are executed in the order they arrive, like a **single queue at a billing counter**.

Given Processes

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	2

Step 1: Order of Execution (FCFS)

Order by arrival time:

P1 → P2 → P3

Step 2: Gantt Chart

0 5 8 10
| P1 | P2 | P3 |

Step 3: Finish Time (FT)

Process	FT
P1	5
P2	8
P3	10

Step 4: Turnaround Time (TAT)

P1 = 5 − 0 = 5, P2 = 8 − 1 = 7, P3 = 10 − 2 = 8

Step 5: Waiting Time (WT)

P1 = 5 − 5 = 0, P2 = 7 − 3 = 4, P3 = 8 − 2 = 6

FCFS is simple but causes **long waiting time** for short jobs.

SJF (Shortest Job First – Non-Preemptive)

Shortest CPU burst executes first. Reduces average waiting time by finishing small tasks early.

Limitation

Long processes may starve.

Shortest task is done first to **reduce average waiting time** (like quick tasks handled first).

Given Processes

Process	Arrival	Burst
P1	0	6
P2	1	2
P3	2	4

Step 1: At Time 0

Only P1 has arrived → execute P1

Step 2: Remaining Processes

P2 (2), P3 (4) → choose **shortest**

Gantt Chart

0 6 8 12

| P1 | P2 | P3 |

Waiting Time

Process	WT
P1	0
P2	5
P3	6

SJF gives minimum average waiting time but may cause **starvation**.

SRTF (Shortest Remaining Time First)

Preemptive version of SJF. CPU switches to shorter jobs to improve responsiveness.

CPU switches to a **shorter task if it arrives**, improving responsiveness.

Given

Process	Arrival	Burst
P1	0	8
P2	1	4
P3	2	2

Execution Flow

- Time 0–1: P1
- Time 1: P2 arrives → shorter than P1 → switch
- Time 2: P3 arrives → shortest → switch

Gantt Chart

```
0 1 2 4 6 10
|P1|P2|P3|P2|P1|
```

SRTF improves response time but causes **many context switches**.

Priority Scheduling

CPU assigned based on priority.

- **Non-Preemptive**: Running process completes
- **Preemptive**: Higher priority can interrupt

OS and system services run at higher priority.

Higher-priority tasks run first (like emergency tickets).

Given

Process	Priority (1 = High)	Burst
P1	2	4
P2	1	3
P3	3	2

Execution Order

```
P2 → P1 → P3
```

Gantt Chart

```
0   3   7   9
|P2|P1|P3|
```

Priority scheduling may cause **starvation**, solved using **aging**.

Round Robin

Each process gets a fixed **time quantum**. **It Is Used**: Fairness, Good response time, Suitable for interactive systems.

Each task gets **fixed CPU time**, like time slots in interviews.

Given, Time Quantum = 2

Process	Burst
P1	5
P2	4
P3	2

Execution Flow

```
P1 → P2 → P3 → P1 → P2 → P1
```

Gantt Chart

```
0 2 4 6 8 10 11
|P1|P2|P3|P1|P2|P1|
```

Round Robin gives **fairness and good response time**, widely used.

SIMPLE C PROGRAMS (CONCEPTUAL UNDERSTANDING)

Program 1: Process Creation (fork)

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

Explanation

- fork() creates a child process
- Both parent and child execute printf
- Output appears **twice**

One program can create **multiple processes**, which is how shells and server's work.

Program 2: fork + wait (Parent waits)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
   if (fork() == 0) {
      printf("Child running\n");
   } else {
      wait(NULL);
      printf("Parent running\n");
   }
   return 0;
}
```

Explanation

- Parent waits for child to finish, Prevents zombie processes

Real Use: Shell waits for commands to complete.

Program 3: Thread Creation (Concept)

```
#include <stdio.h>
#include <pthread.h>
void* task() {
   printf("Thread running\n");
   return NULL;
}
int main() {
   pthread_t t;
   pthread_create(&t, NULL, task, NULL);
   pthread_join(t, NULL);
   return 0;
}
```

Explanation

- Creates a lightweight execution unit, Faster than creating a process

Real Meaning: Servers handle many clients using threads.

System Call Interface for Process Management

Every time you run a command in Linux, open an application, or a server handles a request, the operating system **creates processes, runs programs, and cleans them up**. This entire life cycle is controlled using **process management system calls**.

Understanding these system calls explains:

- How a shell executes commands
- How parent–child processes work
- How servers and background jobs are managed
- How the OS avoids zombie processes

Process Life Cycle (Big Picture Flow)

In real systems, process execution usually follows this flow:

Create → Execute → Wait → Exit

Linux provides system calls for each step:

- **fork / vfork** → create process
- **exec** → run a new program
- **wait / waitpid** → synchronize parent & child
- **exit** → terminate process

1. fork() – Creating a New Process

What fork() Does?

fork() creates a **new child process** by duplicating the parent process.

After fork():

- Parent and child run **independently**
- Both continue execution from the next line
- Only PID values differ

Simple fork() Program

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

Explanation

- fork() creates one child process
- Both parent and child execute printf
- Output appears **twice**

This shows how **one program can create multiple processes**, which is exactly how:

- Linux shell runs commands
- Servers handle multiple clients

2. fork() Return Value

pid = fork();

- pid == 0 → child process
- pid > 0 → parent process
- pid < 0 → fork failed

fork() with Parent–Child Identification

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int pid = fork();
   if (pid == 0)
      printf("Child process\n");
   else
      printf("Parent process\n");
   return 0;
}
```

Explanation

The OS uses return value to let the program know **who is parent and who is child**.

3. exec() – Replacing Process Image

What exec() Does?

exec() **replaces the current process code** with a new program. PID remains the same, but program changes.

This is how a child process runs a **different program** after fork.

fork() + exec() Program

```
#include <stdio.h>
#include <unistd.h>
int main() {
   if (fork() == 0) {
      execlp("ls", "ls", NULL);
   }
   return 0;
}
```

Explanation

- Child process replaces itself with ls
- Parent continues normally
- No new process is created by exec

When you type a command in Linux:

Shell → fork() → child → exec(command)

This is one of the **most asked OS interview flows**.

4. exit() – Terminating a Process

What exit() Does?

exit() terminates a process and returns a status to the parent.

Simple exit() Example

```
#include <stdlib.h>
int main() {
   exit(0);
}
```

Explanation

- 0 indicates successful termination
- Non-zero values indicate errors

Programs signal success or failure to the OS, which helps:

- Scripts detect errors
- Parent processes manage children

5. wait() – Parent Waiting for Child

Why wait() Is Needed?

If a parent does not wait, the child may finish first and become a **zombie process**.

fork() + wait() Program

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
   if (fork() == 0) {
      printf("Child running\n");
   } else {
      wait(NULL);
      printf("Parent running\n");
   }
   return 0;
}
```

Explanation

- Parent waits until child completes
- Ensures proper cleanup
- Prevents zombie processes

Shell waits for commands to finish before showing prompt again.

6. waitpid() – Controlled Waiting

What waitpid() Does?

waitpid() allows the parent to:

- Wait for a **specific child**
- Perform **non-blocking wait**

waitpid() Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
   int pid = fork();
```

```
    if (pid == 0) {
        printf("Child running\n");
    } else {
        waitpid(pid, NULL, 0);
        printf("Parent resumed\n");
    }
    return 0;
}
```

Explanation

- Parent waits only for the given child PID
- Useful when multiple children exist

Servers managing multiple worker processes use waitpid().

7. vfork() – Optimized Process Creation

What vfork() Does?

vfork() is similar to fork() but:

- Child **shares address space**
- Parent is blocked until child calls exec() or exit()

Why vfork() Exists?

It reduces overhead when:

- Child immediately calls exec()

vfork() Example

```
#include <stdio.h>
#include <unistd.h>
int main() {
    if (vfork() == 0) {
        execlp("ls", "ls", NULL);
    }
    return 0;
}
```

- vfork() is rarely used today
- Mainly asked for **conceptual difference**

fork() vs vfork() (One-Line Interview Clarity)

- fork() → separate memory
- vfork() → shared memory until exec/exit

8. Complete Execution Flow

User types command

↓

Shell calls fork()

↓

Child calls exec()

↓

Parent calls wait()

↓

Child exits

↓

Parent resumes

Process Management and Synchronization

In real systems:

- Many programs run at the same time
- They often share common data (files, counters, records, memory)

Because the CPU switches very fast between programs, two programs may access the same data at the same time, leading to wrong results.

To avoid this, the OS needs synchronization mechanisms.

2. Race Condition (The First and Main Problem)

What is a Race Condition?

A race condition occurs when: Two or more programs access shared data, at least one program modifies the data and the final result depends on execution order

Execution order is decided by the OS, not by the programmer, so results become unpredictable.

Simple Example

Shared variable:

counter = 10

- Program A: counter = counter + 1
- Program B: counter = counter - 1

Expected result = 10

Actual result = may be 9, 10, or 11

This unpredictability is a **race condition.**

3. Critical Section (Where the Problem Occurs)

What is a Critical Section?

A critical section is the part of a program where shared data is accessed or modified. If more than one program enters the critical section at the same time, race conditions occur.

Why Critical Section Is Needed?

Critical section is needed because: Shared data must remain correct and only one program should modify shared data at a time.

Without controlling the critical section: Data corruption occurs, System becomes unreliable

Real-World Critical Sections

- Updating bank balance
- Writing to a shared file
- Updating database record
- Ticket booking systems

Standard Format of a Critical Section

Entry Section

Critical Section

Exit Section

Remainder Section

- Entry → request permission
- Critical → access shared data
- Exit → release permission

4. Critical Section Problem (Conditions)

Any correct solution must satisfy all three:

1. Mutual Exclusion: Only one program inside critical section
2. Progress: If no one is inside, someone must be allowed to enter
3. Bounded Waiting: No program waits forever

5. Why Normal Code Fails (Atomicity)

What is an Atomic Operation?

An atomic operation: Executes completely, cannot be interrupted and appears as a single step

Why Atomicity Is Required?

Statement:

```
counter = counter + 1
```

Internally:

1. Read counter
2. Add 1
3. Write back

Another program can interrupt between steps → race condition.

6. Synchronization Hardware (Why Hardware Support Is Needed)

Software alone cannot prevent interruptions. The CPU provides atomic instructions that execute without interruption. These instructions are the foundation of synchronization.

7. Test-and-Set (Atomic Instruction)

Test-and-Set is an atomic instruction that:

- Tests a lock
- Sets it in one step

Test-and-Set Algorithm

```
boolean TestAndSet(boolean *lock) {
    boolean old = *lock;
    *lock = true;
    return old;
}
```

Using Test-and-Set for Critical Section

```
while (TestAndSet(lock) == true)
    ;   // wait
// critical section
lock = false;
```

What This Achieves

- Only one program enters critical section, Mutual exclusion is guaranteed

Limitation of Test-and-Set: Busy waiting (CPU waste), Low-level, Hard to manage

Hence, better abstractions are needed.

8. Compare-and-Swap (CAS)

Compare-and-Swap is an atomic instruction that:

- Compares memory value with expected value
- Updates it only if they match

Compare-and-Swap Algorithm

```
CAS(address, expected, new_value):
   if (*address == expected)
      *address = new_value
      return true
   else
      return false
```

Using CAS for Lock

```
while (CAS(lock, 0, 1) == false)
   ;   // wait
// critical section
lock = 0;
```

Why CAS Is Better

- More flexible than Test-and-Set
- Used in modern processors
- Foundation for lock-free programming

9. Why Hardware Alone Is Not Enough?

Problems with hardware solutions: Busy waiting, Difficult to program, Error-prone, Not portable. So higher-level synchronization mechanisms are required.

10. Semaphores

A semaphore is an integer variable used to control access to shared resources using atomic operations.

Semaphore Operations

- wait(P) → request resource
- signal(V) → release resource

Semaphore Algorithm

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

Types of Semaphores

Binary Semaphore (Mutex)

- Values: 0 or 1
- Used for mutual exclusion

Counting Semaphore

- Values ≥ 0
- Used for multiple identical resources

Simple Semaphore Example

```
wait(mutex)
// critical section
signal(mutex)
```

Semaphore acts like a key:

- One key → one person enters room
- Others wait

Classical Problems of Synchronization

Why Classical Synchronization Problems Exist?

Before learning advanced synchronization tools, we must understand real problems that occur in real systems.

Classical synchronization problems are not puzzles. They represent common patterns of resource sharing that appear in: Operating systems, Databases, Servers, Producer–consumer pipelines, File systems

1. Producer–Consumer Problem

In many real systems:

- One part produces data
- Another part consumes data
- Data is stored in a shared buffer

This happens everywhere: Print queue, Logging system, Message queues, Data pipelines, OS I/O buffers.

Think of a printer:

- Producer → User sending print jobs
- Consumer → Printer printing jobs
- Buffer → Print queue (limited size)

Problems:

- Producer should not add when buffer is full
- Consumer should not remove when buffer is empty

Why Synchronization Is Needed?

Without control:

- Producer may overwrite data
- Consumer may read invalid data
- System crashes or behaves wrongly

Producer–Consumer Algorithm

We use three controls:

- mutex → protects buffer (critical section)
- empty → number of empty slots
- full → number of filled slots

```
//Producer
wait(empty)
wait(mutex)
   add item to buffer
signal(mutex)
signal(full)
//Consumer
wait(full)
wait(mutex)
```

```
    remove item from buffer
signal(mutex)
signal(empty)
```

2. Readers–Writers Problem

Many systems have:

- Many users reading data
- Few users writing data

Examples:

- Database queries
- File systems
- Shared configuration files

Think of a college database:

- Many students → read marks
- Admin → updates marks

Rules:

- Many readers can read together
- Writer must have exclusive access

Why Simple Lock Is Not Enough?

If we use one lock:

- Readers block each other (slow system)
- Performance drops

So, we need: Concurrent reading and Exclusive writing

Readers–Writers Solution

Controls used:

- readCount → number of readers
- mutex → protects readCount
- writeLock → blocks writers

```
//Reader
wait(mutex)
readCount++
if readCount == 1
    wait(writeLock)
signal(mutex)
```

```
        //read data
wait(mutex)
readCount--
if readCount == 0
   signal(writeLock)
signal(mutex)
//Writer
wait(writeLock)
write data
signal(writeLock)
```

3. Dining Philosophers Problem

This problem represents:

- Multiple processes
- Multiple shared resources
- Each process needs more than one resource

Examples:

- CPU + memory
- File + network
- Database locks

Think of 5 processes needing:

- Resource A
- Resource B

If each grab one and waits for another:

- Nobody progresses
- System freezes

Why This Problem Is Important?

It explains: Deadlock, Resource starvation, Poor resource allocation

If everyone: Picks left resource and waits for right resource

Then:

- No one finishes
- Deadlock occurs

Problem with Semaphores

Even though semaphores solve problems:

- Code is hard to read, Easy to make mistakes, Order of wait/signal matters, Bugs are hard to detect. So, developers needed safer and cleaner solutions.

4. Critical Regions (Step Towards Simplicity)

A critical region is a block of code: Only one program can execute inside and Entry and exit are managed automatically

Why Critical Regions Were Introduced

To: Reduce programmer mistakes, avoid forgetting signal calls, Improve readability

```
Conceptual View
enter critical region
   access shared data
exit critical region
```

5. Monitors (FINAL AND BEST SOLUTION)

Monitors were introduced because:

- Semaphores are powerful but dangerous
- Synchronization logic should be automatic
- Mutual exclusion should be guaranteed by design

What Is a Monitor?

A monitor is a high-level synchronization construct that:

- Automatically allows only one process inside
- Hides locking details
- Makes programs safer

Monitor Structure (Simple)

```
monitor SharedResource {
   shared data
   function update() {
      // only one process executes here
   }
}
```

Real-World Usage

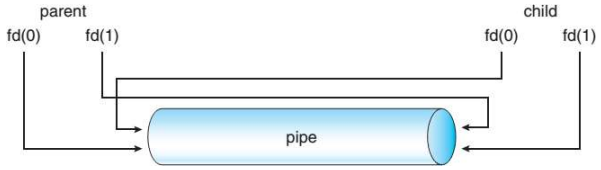
- Java synchronized, Database transactions, Thread-safe libraries, OS resource managers.

UNIT – 3 – INTER PROCESS COMMUNICATION (IPC)

Each process has its **own memory**, so processes **cannot share data directly**. IPC mechanisms allow processes to **exchange data safely** in real systems like shells, servers, databases, and pipelines.

1. Pipes

A pipe allows **one process to write data** and **another to read it** (one direction). Used mainly between **parent and child processes**.



Important Syntax

```
int fd[2];
pipe(fd);
```

- fd[0] → read end, fd[1] → write end

Simple Pipe Program

```
int fd[2];
pipe(fd);
if (fork() == 0) {
   write(fd[1], "Hi", 2);
} else {
   char buf[10];
   read(fd[0], buf, 2);
   printf("%s", buf);
}
```

Explanation

- pipe(fd) creates communication channel
- fork() creates child
- Child writes, parent reads
- Shows **basic IPC using pipe.** Pipes are simple IPC used between related processes.

2. FIFOs (Named Pipes)

Pipes work only for related processes. **FIFOs allow unrelated processes to communicate.**

FIFO Creation: Creates a special file used for IPC.

```
mkfifo myfifo
```

FIFO Write Program

```
int fd = open("myfifo", O_WRONLY);
write(fd, "Hello", 5);
close(fd);
```

FIFO Read Program

```
int fd = open("myfifo", O_RDONLY);
read(fd, buf, 5);
printf("%s", buf);
close(fd);
```

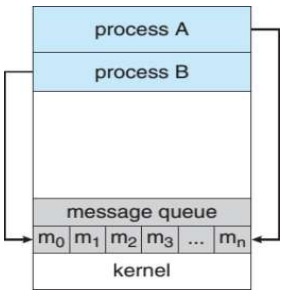
Explanation

- FIFO behaves like a file
- One process writes, another reads
- Communication happens through filesystem

FIFO is a named pipe used between unrelated processes.

3. Message Queues

Pipes/FIFOs send raw data. Message queues send **structured messages**.



Important System Calls

Call	Purpose
msgget	Create/access queue
msgsnd	Send message
msgrcv	Receive message
msgctl	Control/remove queue

Message Structure

```
struct msg {
    long type;
    char text[100];
};
```

- type helps select messages
- text stores data

Sender Program

```
int id = msgget(1234, 0666 | IPC_CREAT);
struct msg m = {1, "Hello"};
msgsnd(id, &m, sizeof(m.text), 0);
```

Receiver Program

```
int id = msgget(1234, 0666);
struct msg m;
msgrcv(id, &m, sizeof(m.text), 1, 0);
printf("%s", m.text);
```

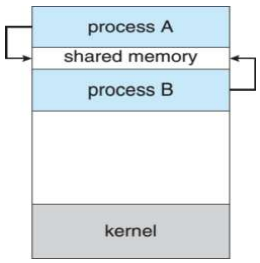
Explanation

- msgget creates or opens queue
- msgsnd sends message
- msgrcv receives by type

Message queues allow asynchronous and structured IPC.

4. Shared Memory

Other IPC methods **copy data**. Shared memory allows **direct memory access** → fastest IPC.



Important System Calls

Call	Purpose
shmget	Create shared memory
shmat	Attach memory
shmdt	Detach
shmctl	Control/remove

Writer Program

```
int id = shmget(1234, 1024, 0666 | IPC_CREAT);
char *data = shmat(id, NULL, 0);
strcpy(data, "Hello");
shmdt(data);
```

Reader Program

```
int id = shmget(1234, 1024, 0666);
char *data = shmat(id, NULL, 0);
printf("%s", data);
shmdt(data);
```

Explanation

- shmget creates memory
- shmat attaches memory to process
- Both processes access same memory

Shared memory provides **no synchronization**.

5. Semaphores (With IPC)

When multiple processes access shared memory, **race conditions occur**. Semaphores control access.

Important Semaphore Calls

Call	Purpose
semget	Create semaphore
semop	wait / signal
semctl	Control/remove

Semaphore Usage Concept

```
wait(semaphore)
    access shared memory
signal(semaphore)
```

Shared memory + semaphore = fast and safe IPC.

6. IPC Status Commands

IPC objects remain in kernel until removed.

Commands

```
ipcs    # view IPC objects
ipcrm   # remove IPC objects
```

IPC resources must be explicitly cleaned.

7. IPC Comparison

IPC	Used For
Pipe	Parent–child
FIFO	Unrelated processes
Message Queue	Structured messages
Shared Memory	Fast data sharing
Semaphore	Synchronization

DEADLOCKS

In real systems: Many processes run at the same time, they share **limited resources** (files, memory, database locks, printers)

If resources are not managed carefully:

- Processes wait forever; system stops making progress and applications appear "hung"

This situation is called **deadlock**.

Real-World Deadlock

Two people, two resources:

- Person A holds **Pen**, waits for **Paper**
- Person B holds **Paper**, waits for **Pen**

No one can finish. This is **deadlock**, even outside computers.

System Model (How OS Sees Deadlock)

The OS assumes: Processes **request**, **use**, and **release** resources, this is called **Resource Cycle**

```
Request → Use → Release
```

Deadlock occurs when **Request never completes**.

What Is a Deadlock?

A **deadlock** is a situation where: Each process holds a resource, Each process waits for another resource, no process can proceed

Deadlock Characterization (WHY Deadlock Happens)

Deadlock can occur **only if ALL four conditions are true**.

1. Mutual Exclusion

Some resources cannot be shared. **Example:** Printer, file lock

2. Hold and Wait

Process holds one resource and waits for another. **Example:** Holds file, waits for memory

3. No Preemption

Resources cannot be forcibly taken away. **Example:** OS cannot snatch printer from a process

4. Circular Wait

P1 → waits for P2

P2 → waits for P3

P3 → waits for P1

Deadlock occurs **only when all four conditions are satisfied simultaneously**.

Simple Deadlock Program

```
lock(A);
lock(B);
/* critical section */
unlock(B);
unlock(A);
Another process runs:
lock(B);
lock(A);
```

Explanation

- Process 1 holds A, waits for B
- Process 2 holds B, waits for A
- Circular wait is created
- Both processes wait forever

This is a **real deadlock caused by wrong lock order**.

This program shows:

- **Circular wait, Hold and wait**
- How deadlock happens because of **wrong lock order**

This is the **most common deadlock interview example**.

C Program: Deadlock Example (Using Mutex Locks)

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t lock1;
pthread_mutex_t lock2;
void* process1(void* arg) {
    pthread_mutex_lock(&lock1);
    printf("Process 1 acquired Lock 1\n");
    pthread_mutex_lock(&lock2);
    printf("Process 1 acquired Lock 2\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
void* process2(void* arg) {
    pthread_mutex_lock(&lock2);
    printf("Process 2 acquired Lock 2\n");
    pthread_mutex_lock(&lock1);
    printf("Process 2 acquired Lock 1\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    return NULL;
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);
    pthread_create(&t1, NULL, process1, NULL);
    pthread_create(&t2, NULL, process2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Explanation

- process1 locks **lock1** first, then waits for **lock2**
- process2 locks **lock2** first, then waits for **lock1**
- Each process holds one resource and waits for the other
- **Circular wait is created**
- Neither process can proceed → **DEADLOCK**

Which Deadlock Conditions Are Satisfied?

Condition	Present?	Why
Mutual Exclusion	YES	Locks are exclusive
Hold and Wait	YES	Each holds one lock
No Preemption	YES	Locks not forcibly taken
Circular Wait	YES	lock1 → lock2 → lock1

All four conditions satisfied → Deadlock

Why OS Needs Deadlock Handling Methods?

Deadlocks:

- Waste CPU and memory, freeze applications, reduce system reliability

So, OS must choose **how to deal with deadlocks**.

Methods for Handling Deadlocks

Ignore

Prevention

Avoidance

Detection & Recovery

Deadlock Prevention (Stop Deadlock Early)

Prevent deadlock by **breaking at least one of the four conditions**.

Example: Prevent Circular Wait

Rule:

Always acquire resources in the same order

Correct Program (No Deadlock)

```
lock(A);
lock(B);
/* critical section */
```

```
unlock(B);
unlock(A);
```

Now **all processes follow same order** → no circular wait. Prevention is safe but may reduce performance.

This program shows: How **deadlock is prevented**, by following **consistent resource ordering**

C Program: Deadlock-Free Version

```
void* process1(void* arg) {
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
    printf("Process 1 inside critical section\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
void* process2(void* arg) {
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
    printf("Process 2 inside critical section\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
```

Explanation

- Both processes acquire locks in **same order**
- No circular wait is possible
- **Deadlock is prevented**

Deadlock can be prevented by enforcing a strict resource ordering.

Deadlock Avoidance (Smarter Approach)

Prevention wastes resources. Avoidance checks **whether granting a request is safe**.

Banker's Algorithm (Conceptual)

How OS decides whether to grant a resource request?

Important Terms

- **Available** – free resources

- **Max** – maximum required
- **Allocation** – currently allocated
- **Need = Max − Allocation**

Banker's Logic (Simple)

```
If request ≤ need AND request ≤ available
   Pretend to allocate
   If system is safe → grant
   Else → deny
```

Real-World Analogy

Bank gives loan only if it can still serve all customers.

Banker's Algorithm Logic (Pseudo-C)

```
if (request <= need && request <= available) {
   allocate resources temporarily;
   if (system is in safe state)
      grant request;
   else
      rollback allocation;
}
```

Explanation

- OS checks **before granting**
- Ensures system remains in **safe state**
- Deadlock is avoided, not prevented

Banker's algorithm avoids deadlock by checking safe state before allocation.

Deadlock Detection (Allow, Then Find)

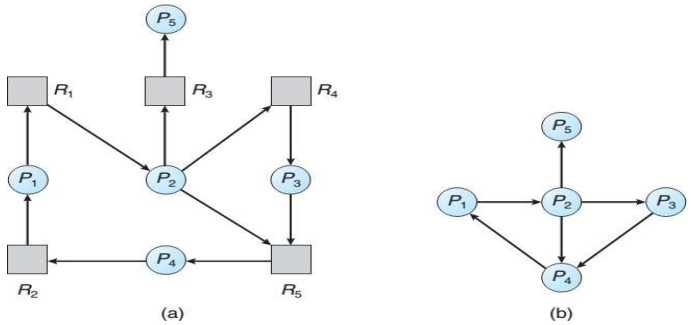
When Detection Is Used

- Deadlock is rare, performance is more important, OS checks periodically

Wait-For Graph

- Node → process
- Edge Pi → Pj means Pi waits for Pj

Detection Rule



Cycle exists → Deadlock exists

Example

P1 → P2 → P3 → P1

Cycle → deadlock detected.

Recovery From Deadlock

Once deadlock is detected, OS must recover.

1. Process Termination

- Kill one or more processes, Free resources

- kill(process_id);

Used when: Process is not critical.

2. Resource Preemption

- Take resource from process, Rollback and restart

Used in: Databases (transaction rollback)

Program Showing Prevention Idea

Wrong Order (Deadlock Risk)

```
lock(B);
lock(A);
```

Correct Order (Deadlock Free)

```
lock(A);
lock(B);
```

Explanation

- Same resources
- Different order causes deadlock
- Consistent order prevents deadlock

UNIT – 4 – MEMORY MANAGEMENT & VIRTUAL MEMORY

In real systems:

- RAM is **limited, m**any programs run at the same time, programs are large and dynamic

If memory is not managed properly:

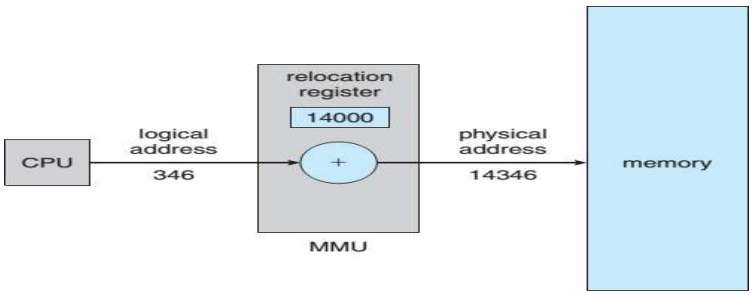
- Programs overwrite each other, System crashes, Performance becomes poor

The OS must **control memory strictly**.

Logical vs Physical Address Space

Programs think they have their **own memory**, but physically: All programs share the same RAM. OS must create an **illusion**.

Logical address is what program uses, physical address is where data actually exists in RAM.



Logical Address (Virtual Address)

- Generated by the **CPU** → Used by programs → Program thinks this is real memory

Physical Address

- Actual location in **RAM** → Used by memory hardware

Address Translation (Key Idea)

Logical Address → MMU → Physical Address

MMU = Memory Management Unit

Swapping

RAM is limited, but: Many programs want to run, OS must make space

What Is Swapping?

Swapping means: Moving a process from RAM to disk, bringing it back later when needed

Real-World Analogy

Think of: Desk = RAM, Cupboard = Disk

Keep active files on desk, others in cupboard.

Swapping increases multiprogramming but slows performance.

What Is Fragmentation?

Fragmentation means: Memory is available, but it is **not usable efficiently**.

Fragmentation leads to: Wasted memory, Failed allocations, Poor performance

There are **two types**:

1. External Fragmentation
2. Internal Fragmentation

External Fragmentation

External fragmentation occurs when: Total free memory is sufficient, but it is broken into **small non-contiguous blocks**, no single block is large enough for a process

How External Fragmentation Occurs

1. Process A allocated memory
2. Process B allocated memory
3. Process A finishes and frees memory
4. Process C needs a large block

Memory looks like this:

[Free][Used][Free][Used][Free]

Total free memory exists, but **not in one block**.

Problems Caused by External Fragmentation

- Memory allocation fails even when memory exists
- System performance degrades
- Leads to unnecessary swapping

How External Fragmentation Is Solved: Paging

Memory is allocated in fixed-size blocks → no need for contiguous memory. **So,**

External fragmentation occurs due to contiguous allocation and is solved by paging.

Internal Fragmentation

Internal fragmentation occurs when: Allocated memory block is **larger than required** and Unused space exists **inside** the allocated block.

Why Internal Fragmentation Happens

Paging uses **fixed-size frames**.

Example:

- Page size = 4 KB
- Process needs = 6 KB

Allocated: 2 pages = 8 KB

Unused = 2 KB

Problems Caused by Internal Fragmentation

- Wasted RAM
- Reduced memory efficiency

How Internal Fragmentation Is Reduced

- Smaller page size
- Better memory utilization techniques

Paging removes external fragmentation but may cause internal fragmentation.

Why Paging Was Introduced?

Problem Before Paging

- Contiguous allocation
- External fragmentation
- Rigid memory placement

Paging divides:

- Logical memory → fixed-size **pages**
- Physical memory → fixed-size **frames**

Pages can be placed **anywhere in RAM**.

Real-World Use

Modern OS: Windows, Linux, macOS all use paging.

Paging allows non-contiguous allocation and improves memory utilization.

Why Segmentation Is Needed When Paging Exists

Problem with Paging Alone

Paging: Is good for memory management and does **not represent program structure**

Programs have logical parts: Code, Data, Stack

Paging ignores this.

Segmentation (Definition)

Segmentation divides a program into:

- Logical units called segments
- Variable-sized blocks

Address format:

Segment number + Offset

Real-World Use

- Separate protection for code and data
- Stack growth handling
- Logical program organization

Paging is memory-oriented, segmentation is logic-oriented.

Virtual Memory (WHY IT EXISTS)

Programs can be:

- Larger than physical RAM
- But only parts are needed at a time

Virtual Memory (Definition)

Virtual memory allows execution of large programs by:

- Keeping only required pages in RAM
- Using disk as backup memory

Real-World Example

Browser:

- Entire browser not loaded
- Only active tabs/pages loaded

Virtual memory provides an illusion of large memory using disk.

Demand Paging & Page Faults

Demand Paging (Definition)

Pages are loaded into RAM: **Only when they are accessed**

Page Fault (Definition)

A **page fault** occurs when: A process accesses a page that page is **not in RAM**

Page Fault Handling (Step-by-Step)

Page fault occurs → OS finds page on disk → Free frame selected → Page loaded → Page table updated → Instruction restarted

Real-World Impact

- Too many page faults → slow system
- Proper replacement → good performance

Page Replacement (WHY REQUIRED)?

Real Problem: RAM is full.
New page must be loaded.

Now, OS must decide **which page to remove**.

This is **page replacement.**

PAGE REPLACEMENT ALGORITHMS & PROBLEMS

Problem 1: FIFO (First In First Out)

- **Remove the oldest page**
- **Simple**
- **May remove frequently used pages**

Reference string:

1 2 3 4 1 2 5

Frames = 3

FIFO Execution

Step	Pages in Memory
1	1
2	1 2
3	1 2 3
4	2 3 4
1	3 4 1
2	4 1 2
5	1 2 5

FIFO Page Faults = 7

FIFO is simple but may remove frequently used pages.

Problem 2: LRU (Least Recently Used)

- **Remove page not used for longest time**
- **Better performance**
- **Harder to implement**

Reference string:

1 2 3 4 1 2 5

Frames = 3

LRU Execution (Concept)

- Replace page **not used for longest time**

Step	Pages in Memory
1	1
2	1 2
3	1 2 3
4	2 3 4
1	3 4 1
2	4 1 2
5	1 2 5

LRU Page Faults = 6

LRU performs better than FIFO but is harder to implement.

Optimal Algorithm (Concept Only)

Definition: Replace the page that will not be used for the **longest future time**.

Why It Is Not Implemented?

- Future references are unknown, Used only as benchmark

Optimal algorithm gives minimum page faults but is not practical.

Fragmentation Problems & Solutions

Problem	Occurs In	Solution
External Fragmentation	Contiguous allocation	Paging
Internal Fragmentation	Paging	Smaller page size
Large programs	Limited RAM	Virtual memory
Full RAM	Demand paging	Page replacement

OPERATING SYSTEM PROTECTION

Why Protection Is Needed

Modern systems are: Multi-user, Multi-process, highly connected

Many programs and users **share the same system resources**: Files, Memory, CPU, Devices

If protection is not enforced:

- One program can read or modify another program's data

- Sensitive user data can be leaked
- A faulty or malicious program can crash the system

Protection ensures controlled and safe access to system resources.

Protection is about **controlling "who can do what" inside the system**.

System Protection

System protection refers to mechanisms used by the operating system to: Control access to resources so that each process or user can only perform permitted operations.

What Needs Protection in Real Systems

Resource	Real-World Example
Files	Student cannot read admin files
Memory	One process cannot access another's memory
CPU	No process can monopolize CPU
Devices	Printer used by one job at a time

Inside the System OS checks **permissions before every access,** Unauthorized access is **blocked automatically,** Protection works continuously, not occasionally.

System protection prevents unauthorized access and ensures system stability.

Goals of Protection (WHY OS DOES THIS)

Goal 1: Prevent Unauthorized Access

Only allowed users/processes should access resources. Example: A normal user cannot modify system configuration files.

Goal 2: Ensure System Stability

A faulty program should not affect other programs. Example: If one application crashes, others continue running.

Goal 3: Enable Controlled Sharing

Resources must be shared safely. Example: Multiple users can read the same file, but only one can write.

Goal 4: Support Ethical and Secure Usage

Sensitive data must not be misused. Example: AI systems must protect training data and user information.

The goal of protection is to ensure safe, controlled, and reliable use of system resources.

Principles of Protection

Principle of Least Privilege definition

A user or process should be given: Only the minimum access required to perform its task.

Why This Principle Matters in Real Systems

- Reduces damage from bugs, Limits impact of malware, Improves system security

Real-World Examples Inside Systems

Scenario	Least Privilege Applied
Normal user	Cannot install software
Application	Cannot access system files
Database user	Read-only access
API token	Limited permissions

Least privilege minimizes risk by limiting access to only what is necessary.

Domain of Protection (HOW OS APPLIES PERMISSIONS)

A **domain of protection** is: A set of access rights that a process or user has at a given time.

Think of a domain as:

- Current **permission context,** What the process is allowed to do **right now**

Inside the System

- When a process runs, it runs **inside a domain**
- Domain defines:
    - Which files it can access
    - Which operations it can perform

Real Examples

Situation	Domain Change
User login	User domain
Admin command	Elevated domain
System call	Kernel domain

A domain defines the access rights active for a process at a given time.

Access Matrix

An **access matrix** is a conceptual table that shows:

- Subjects (users/processes)
- Objects (files/resources)
- Allowed operations (read/write/execute)

Simple Representation

	File A	File B
User	Read	—
Admin	Read, Write	Read

Why This Matters

- Helps explain **who can access what**
- Forms the basis of real systems like:
    - File permissions, Role-based access control

Access matrix is a conceptual model to represent access rights in a system.

UNIT – 5 – FILE SYSTEM INTERFACE & OPERATIONS

Why File Systems Matter in Real Systems?

In real computers and servers: Data must be **stored permanently**, it must **survive system shutdown, m**ultiple users and programs access data **at the same time.**

Without a file system:

- Data would be lost after power off, Programs could overwrite each other's data, no security or organization would exist

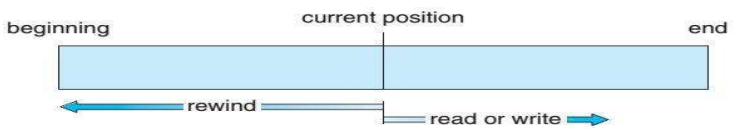
A **File System** is the OS component that: Organizes data on storage devices and controls how files are stored, accessed, and protected File systems are about **data organization, performance, and safety**.

File Access Methods (HOW DATA IS READ)

Different applications access data in different ways. The OS supports multiple **file access methods** to optimize performance.

1. **Sequential Access**

In sequential access, data is read or written in order, from beginning to end.



Real-World Use Inside Systems

- Log files, Media playback, Text files, Backup files

Why It Exists

- Simple, Efficient for large continuous data

Sequential access reads data in order and is efficient for linear processing.

2. **Direct Access (Random Access)**

In **direct access**, a file can be read or written at **any position directly**.

Real-World Use Inside Systems

- Databases, Binary files, Disk-based applications, File editing tools

Why It Exists

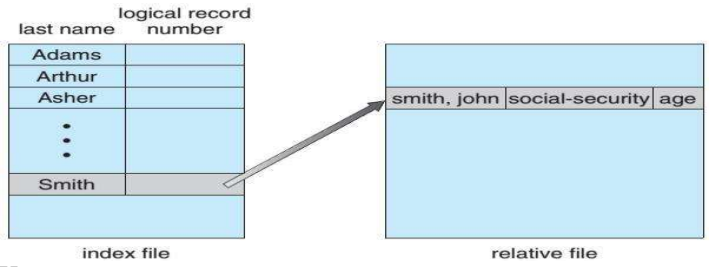
- Fast access to required data, no need to read entire file

Direct access allows reading or writing data at any file location.

3. **Indexed Sequential Access**

Indexed sequential access combines:

- Sequential access, Direct access using an index



Real-World Use

- Database systems, File systems with indexing

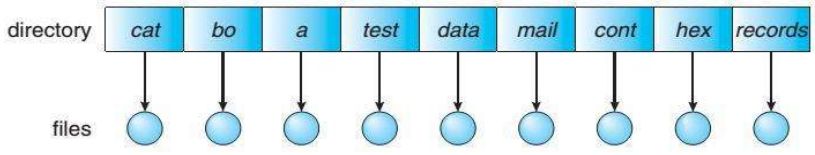
Directory Structure (HOW FILES ARE ORGANIZED)

A **directory** is a structure that: Stores file names, maintains file metadata, helps locate files on disk.

Common Directory Structures

Single-Level

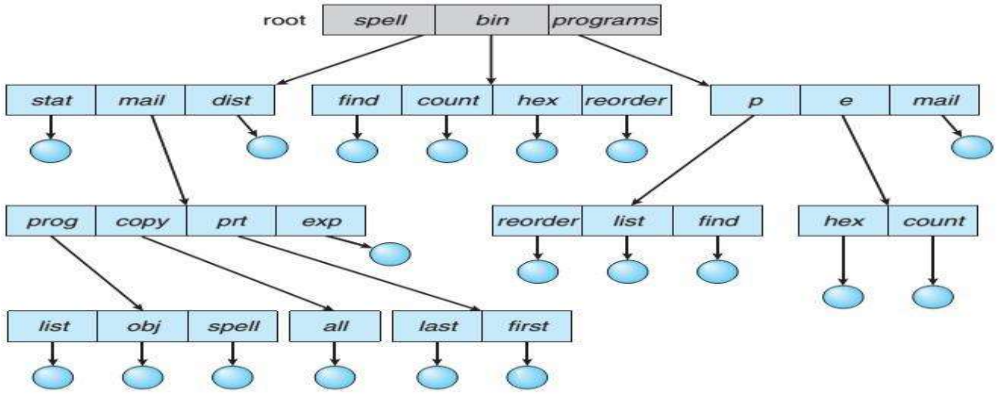
- All files in one directory, not practical for real systems



Tree-Structured Directory

Real-World Example: Linux, Windows, macOS:

/home/user/docs/file.txt



Why Tree Structure Is Used

- Logical organization, Easy navigation, Supports multi-user systems

Tree-structured directories are widely used because they support hierarchical organization.

File Protection (LINK TO OS PROTECTION)

File Protection Is Needed as we have Multiple users, Sensitive data, Shared storage

File protection controls: Who can read, write, or execute a file.

Real-World System Example (Linux)

Permissions:

r w x

- Read → view file
- Write → modify file
- Execute → run file

File System Structure (HIGH-LEVEL VIEW)

Logical Layers (Simple View)

User Programs → File System Interface → Logical File System → Physical File System → Disk

What This Means in Real Systems: Applications don't access disk directly, OS abstracts disk complexity, Improves portability and safety

File systems use layered architecture to separate logic from physical storage.

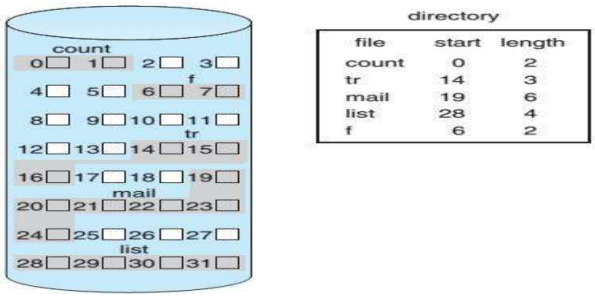
File Allocation Methods

Allocation methods decide: **How file blocks are stored on disk**

This directly affects: Performance, Fragmentation, Disk usage

1. **Contiguous Allocation**

File is stored in **continuous disk blocks**. Contiguous allocation is fast but suffers from external fragmentation.

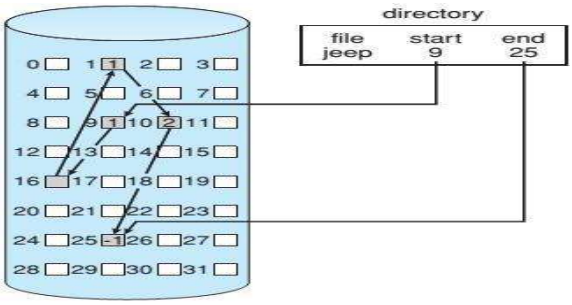


Advantages: Fast sequential access, Simple implementation

Problems: External fragmentation, Difficult to grow files

2. **Linked Allocation**

Each file block contains: Data, Pointer to next block. Linked allocation avoids external fragmentation but has poor random access.

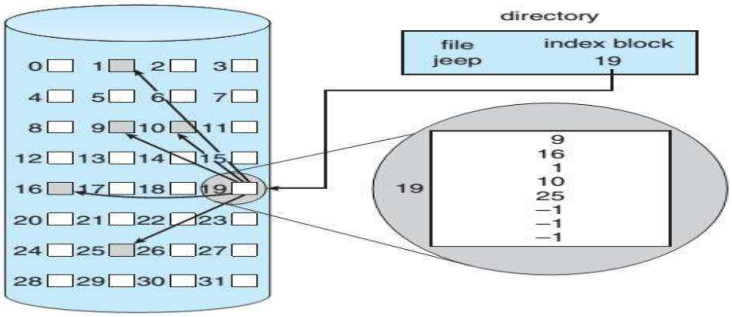


Advantages: No external fragmentation, Files can grow easily

Problems: Slow random access, Pointer overhead

3. **Indexed Allocation (MOST USED)**

All block addresses of a file are stored in a **separate index block**. Indexed allocation provides efficient direct access and is widely used.



Real-World Use: Modern file systems, UNIX inode structure

Advantages: Fast direct access, No external fragmentation

Problem: Index block overhead for small files

Free-Space Management (HOW OS TRACKS EMPTY SPACE)

OS must know: Which disk blocks are free → Which are allocated

Common Methods

Bit Map: Each bit represents a block, Easy to find free space

Free List: Linked list of free blocks

Free-space management helps the OS track unused disk blocks efficiently.

SYSTEM CALLS FOR FILE & DIRECTORY MANAGEMENT

In real systems:

- Applications do **not directly access disks, all** file operations go through the **Operating System,** OS enforces **protection, consistency, and performance.**

System calls are the controlled interface between:

Application → Operating System → Disk

File Management System Calls

How does a program create, access, modify, and manage files safely?

create(), open(), close() – File Lifecycle

1. **create() / open()**

create() and open() are used to: Create a new file, Or open an existing file

In practice, **open() is most commonly used**. When you open a file in a text editor: OS checks permissions → OS allocates file descriptor → OS prepares file for access

Syntax int fd = open("data.txt", O_CREAT | O_WRONLY, 0644);

- fd → file descriptor (integer)
- O_CREAT → create file if it does not exist
- O_WRONLY → write only
- 0644 → permissions

open() returns a file descriptor used for further file operations.

2. **close()**

close() releases the file descriptor and frees OS resources.

Syntax

```
close(fd);
```

Real-World Impact: Prevents resource leaks, Flushes buffers to disk

Closing a file ensures data is safely written and resources are freed.

3. **read() and write() – File Data Access**

read() reads data from a file into memory, write() writes data from memory to a file

Syntax

```
read(fd, buffer, size);

write(fd, buffer, size);
```

Simple C Program (Very Important)

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
   int fd = open("test.txt", O_CREAT | O_WRONLY, 0644);
   write(fd, "Hello OS", 8);
   close(fd);
   return 0;
}
```

Explanation

- File is created → Data is written → File is safely closed

This is **exactly how applications write files internally**. read() and write() perform actual data transfer between file and memory.

4. **lseek() – File Pointer Movement:** lseek() changes the current position of the file pointer.

Why It Is Needed: To: Read/write from specific location, Support random access

Syntax

```
lseek(fd, offset, SEEK_SET);
```

Real-World Example: Editing a file at a specific position without reading the entire file.

lseek() enables random access in files.

5. **link(), symlink(), unlink() – File Linking**

Why Links Exist: Files may need: Multiple names, Shared references

link() – Hard Link: Creates another name for the **same file**.

```
link("a.txt", "b.txt");
```

Both names point to same data.

symlink() – Symbolic Link: Creates a **shortcut** to a file.

```
symlink("a.txt", "c.txt");
```

unlink(): Removes a file name.

```
unlink("a.txt");
```

File data is removed **only when no links exist**.

Hard Link	Symbolic Link
Same inode	Separate file
Cannot cross file systems	Can cross file systems
Faster	More flexible

6. **stat(), fstat(), lstat() – File Information:** These calls retrieve **metadata** about a file. Metadata includes: File size, Permissions, Ownership, Timestamps

Syntax

```
struct stat st;
stat("file.txt", &st);
```

Call	Works On
stat	File
fstat	File descriptor
lstat	Symbolic link

stat() family calls are used to retrieve file metadata.

7. **chmod() and chown() – File Protection: chmod() – Change Permissions**

```
chmod("file.txt", 0644);
```

Controls: Read, Write, Execute permissions

chown() – Change Owner

```
chown("file.txt", uid, gid);
```

Real-World Example: Admin assigning file ownership or permissions. chmod() controls access rights, while chown() changes file ownership.

Directory System Calls

Directories are special files that store file names and locations.

1. **opendir(), readdir(), closedir()**

Used to: Open a directory, Read directory contents, Close directory

Simple C Program

```
#include <dirent.h>
#include <stdio.h>
int main() {
    DIR *d = opendir(".");
```

```
    struct dirent *entry;
    while ((entry = readdir(d)) != NULL) {
        printf("%s\n", entry->d_name);
    }
    closedir(d);
    return 0;
}
```

Explanation

- opendir() opens directory
- readdir() reads each file name
- closedir() releases resources

Directory system calls allow programs to list and manage directories.

2. **mkdir() and rmdir()**

mkdir() – Create Directory

```
mkdir("testdir", 0755);
```

rmdir() – Remove Empty Directory

```
rmdir("testdir");
```

3. **umask() – Default Permission Control**

umask() sets default permission mask for new files.

Real-World Meaning: Controls which permissions are **disabled by default**.

umask() defines default file permission behavior.

SAMPLE INTERVIEW QUESTIONS

UNIT–1

1. *Why can't applications directly talk to hardware? Why do we even need an OS in between?*
2. *If I remove the operating system, what exactly will break first? CPU? Memory? Files?*
3. *From an OS point of view, what is more important — user convenience or system efficiency? Why?*
4. *What is the difference between user mode and kernel mode in very simple terms?*
5. *What problem does the mode bit solve in real systems?*
6. *Can a user program ever execute privileged instructions? How?*
7. *When you run a command in Linux, what does the OS actually do behind the scenes?*

8. *Why is Linux preferred for servers but not always for end users?*
9. *What makes Linux stable for long-running systems?*
10. *What happens internally when you type ls in the terminal?*
11. *Why does the OS provide services instead of applications handling everything themselves?*
12. *What would happen if OS services like memory management didn't exist?*
13. *How does PATH help the shell find commands?*
14. *Why do we need commands like man even when documentation exists online?*
15. *Difference between deleting a file using GUI and rm — from OS perspective?*

UNIT–2

1. *Is a program and a process the same thing? If not, explain with a real example.*
2. *What exactly does the OS store about a process?*
3. *Why does the OS need a PCB? Can't it just run processes directly?*
4. *What happens during a context switch? Why is it costly?*
5. *Why can't the CPU execute multiple processes at the same time?*
6. *Why are threads considered "lightweight"?*
7. *If threads are faster, why do we still need processes?*
8. *What problems arise when multiple threads access shared data?*
9. *Why does CPU scheduling exist? Why not let processes run freely?*
10. *Which scheduling algorithm would you choose for a real-time system and why?*
11. *Why does FCFS perform badly in some cases?*
12. *SJF is optimal, so why don't OSes always use it?*
13. *What happens if time quantum in Round Robin is too small or too large?*
14. *Why does priority scheduling cause starvation?*
15. *How do modern operating systems balance fairness and performance in scheduling?*

UNIT–3

1. *What is a race condition? Can you give a real system example?*
2. *Why does concurrent execution create problems?*
3. *What exactly is a critical section?*
4. *Why can't we allow multiple processes inside a critical section?*
5. *What are atomic operations and why are they important?*

6. *Why do we need semaphores when locks already exist?*
7. *What is the real difference between a mutex and a semaphore?*
8. *Where do classical synchronization problems appear in real systems today?*
9. *Why are Producer–Consumer and Readers–Writers still relevant?*
10. *What problem do monitors solve that semaphores do not?*
11. *What exactly is a deadlock? Explain without using textbook language.*
12. *Why must all four deadlock conditions be present?*
13. *Which deadlock condition is easiest to break in practice?*
14. *How does Banker's Algorithm prevent deadlock?*
15. *Why is Banker's Algorithm rarely used in real systems?*
16. *How does an OS detect deadlock after it has already occurred?*
17. *What is a wait-for graph and why is it useful?*
18. *Once deadlock is detected, how does the OS recover?*
19. *Why do some operating systems ignore deadlocks completely?*

UNIT–4

1. *Why can't programs directly use physical memory?*
2. *What is the practical difference between logical and physical address?*
3. *What real problem does swapping solve? What new problem does it introduce?*
4. *What exactly is fragmentation? Why is it bad?*
5. *Explain external fragmentation using a real-world example.*
6. *Why does paging eliminate external fragmentation?*
7. *If paging exists, why does internal fragmentation still occur?*
8. *Why do we need segmentation when paging already exists?*
9. *Paging is OS-friendly, segmentation is programmer-friendly — explain.*
10. *What is virtual memory and why is it absolutely necessary today?*
11. *How does virtual memory help in running large applications?*
12. *What is demand paging and why is it better than loading full program?*
13. *What exactly happens when a page fault occurs?*
14. *Why are too many page faults dangerous?*
15. *What is page replacement and when is it triggered?*
16. *Why does FIFO sometimes give poor performance?*
17. *What is Belady's anomaly?*

18. *Why does LRU usually perform better than FIFO?*
19. *Why is the Optimal algorithm only theoretical?*
20. *How does the OS decide which page to replace in real systems?*

PART - B

1. *Why can't processes communicate directly using memory?*
2. *Why does the OS restrict direct memory sharing between processes?*
3. *What IPC mechanism would you use for parent–child communication and why?*
4. *Why are pipes limited in functionality?*
5. *Why were FIFOs introduced when pipes already existed?*
6. *What problem do message queues solve that pipes cannot?*
7. *Why are message queues useful in distributed systems?*
8. *Why is shared memory the fastest IPC mechanism?*
9. *If shared memory is fast, why don't we always use it?*
10. *What happens if shared memory is used without synchronization?*
11. *Why are semaphores tightly coupled with shared memory?*
12. *How does IPC differ from synchronization?*
13. *Why must IPC resources be cleaned manually?*
14. *What happens if IPC objects are not removed?*

PART - C

1. *Why can't data be stored directly in memory permanently?*
2. *Why do operating systems need file systems?*
3. *Why are different file access methods required?*
4. *When would sequential access be better than direct access?*
5. *Why are directories needed instead of flat file storage?*
6. *Why is tree-structured directory most common?*
7. *How does file protection differ from memory protection?*
8. *Why does contiguous allocation cause external fragmentation?*
9. *Why can't contiguous allocation support file growth easily?*
10. *How does linked allocation solve fragmentation problems?*
11. *Why is indexed allocation preferred in modern file systems?*
12. *How does file allocation affect performance?*
13. *How does the OS keep track of free disk space?*
14. *Why is free-space management important?*

UNIT–5

1. *Why can't applications directly read or write to disk?*
2. *What exactly is a file descriptor? Why not use file names directly?*
3. *What happens internally when a file is opened?*
4. *Why is closing a file important? What happens if we don't close it?*
5. *What is the difference between read/write and lseek?*
6. *How does lseek enable random access?*
7. *What is the difference between hard link and symbolic link in practice?*
8. *What happens to a file when unlink is called?*
9. *Why do we need stat when file name already exists?*
10. *Difference between stat, fstat and lstat — where are they used?*
11. *How does chmod enforce file protection?*
12. *Why does chown require special privileges?*
13. *How does a program list files inside a directory?*
14. *Why does rmdir fail if directory is not empty?*
15. *What problem does umask solve in real systems?*