OPERATING SYSTEM

Notes

Dr. Kailash Aseri
Assistant Professor (Computer Science)
D.A.V. College, Sri Ganganagar (Rajasthan)

Operating System

Unit I

Introduction to Operating System, layered Structure, Functions, Types; Process: Concept, Process States, PCB; Threads, System calls; Process Scheduling: types of schedulers, context switch, CPU Scheduling, Preemptive Scheduling, Scheduling Criteria- CPU Utilization, Throughput, Turnaround Time, Waiting Time, Response Time;

Unit II

Scheduling Algorithms FCFS, SJF, Priority Scheduling, Round Robin Scheduling, MLQ Scheduling. Synchronization: Critical Section Problem, Requirements for a solution to the critical section problem; Semaphores, simple solution to Readers-Writers Problem.

Unit III

Deadlock: Characterization, Prevention, Avoidance, Banker‟s Algorithm, Recovery from Deadlock. Memory Management: Physical and virtual address space, Paging, Overview of Segmentation; Virtual Memory Management: Concept, Page Replacement techniques- FIFO, LRU, Optimal.

Unit IV

Linux: features of Linux, steps of Installation, Shell and kernel, Directory structure, Users and groups, file permissions, commands- ls, cat, cd, pwd, chmod, mkdir, rm, rmdir, mv, cp, man, apt, cal, uname, history etc.; Installing packages

Unit IV

Shell scripts: writing and executing a shell script, shell variables, read and expr, decision making (if-else, case), for and while loops

UNIT –I

1. Introduction to Operating System (OS)

What is an Operating System?

An **Operating System (OS)** is a **system software** that controls the overall functioning of a computer.
It acts like a **manager or middleman** between the **user**, **application programs**, and **hardware** (like CPU, memory, disk, etc.).
Think of it as the **conductor of an orchestra** – it doesn't play an instrument itself but ensures all parts work together smoothly.

Real-Life Example:

- When you use your smartphone to **open YouTube**, the **OS (like Android or iOS)** handles:
  - Touch screen input
  - Loading the app
  - Playing video using hardware (CPU, GPU, sound card)
  - Connecting to the internet

2. Layered Structure of Operating System

Operating Systems are built in **layers**, each having a specific function. This structure makes it easier to **design**, **debug**, and **maintain** the system.

Layers (from bottom to top):

1. **Hardware**
   - All physical components like CPU, RAM, hard disk, keyboard, printer
   - *Example: Intel Processor, Kingston RAM*
2. **Kernel**
   - Core part of OS; directly interacts with hardware
   - Handles process scheduling, memory, and device control
   - *Example: Linux kernel, Windows NT kernel*
3. **System Calls / API Layer**
   - Interface for applications to request services from the OS
   - *Example: Open file, read file, allocate memory*
4. **Shell / Command Interpreter**
   - Interprets user commands
   - *Example: Bash shell in Linux, Command Prompt in Windows*
5. **Application Programs**
   - Software users use for specific tasks
   - *Example: Google Chrome, MS Word, WhatsApp*

3. Functions of Operating System

1. Process Management

- Manages processes (programs in execution)
- Handles creation, scheduling, and termination of processes
- *Example: Multitasking like listening to music while browsing*

2. Memory Management

- Allocates memory to programs when needed and frees it when done
- Ensures efficient use of RAM
- *Example: Keeping multiple tabs open in a browser without crashing*

3. File System Management

- Organizes files into directories
- Controls how data is stored and retrieved from storage devices
- *Example: Saving a Word document in "My Documents"*

4. Device Management

- Controls and manages input/output devices like keyboard, mouse, printer
- Uses device drivers to communicate
- *Example: Sending a print command to the printer*

5. Security and Protection

- Protects system from unauthorized access and threats
- *Example: User login passwords, antivirus support*

6. User Interface
- Provides ways for users to interact with the system (GUI or CLI)
- *Example: Windows desktop icons, Linux terminal commands*

7. Error Detection and Handling
- Detects hardware or software errors and takes corrective action
- *Example: Showing a message when a USB device is removed unsafely*

4. Types of Operating Systems

1. Batch Operating System
- No direct user interaction
- Jobs are collected and processed in batches
- *Example: Payroll systems in early computers*

2. Time-Sharing OS
- Multiple users share system resources at the same time
- CPU switches rapidly between tasks
- *Example: UNIX used in universities and companies*

3. Distributed Operating System
- Controls a group of networked computers as if they are one
- Helps share resources and data across systems
- *Example: Google's data center OS, LOCUS*

4. Real-Time Operating System (RTOS)
- Responds to inputs within strict time limits
- Used in mission-critical systems
- *Example: Airbag control system in cars, Mars rover software*

5. Multi-user OS
- Supports multiple users at the same time
- Each user has a separate account and settings
- *Example: Linux servers, mainframe systems*

6. Mobile Operating System
- Designed for mobile devices
- Lightweight, battery-efficient, touch-optimized
- *Example: Android (Google), iOS (Apple)*

Summary

Topic	Description	Example
OS	Interface between user and hardware	Windows, Linux, Android
Layered Structure	Organized in levels from hardware to apps	Kernel → Shell → Apps
Functions	Process, memory, file, device management, etc.	Task switching, saving files, printing
Types	Batch, Time-sharing, RTOS, Distributed, etc.	Android, UNIX, Windows Server

5. Concept of a Process

A **process** is a **program in execution**.
It is more than just the program code – it also includes the program's **current activity**, **variables**, **memory**, and **resources**.

Example:
When you open **MS Word**, a process is created. It includes:
- The code of MS Word
- Your current document

- The text you've typed
- The memory it uses
- The CPU time it's getting

Key Points:

- A program is **passive** (just code), but a process is **active** (running).
- The operating system uses **processes** to manage running programs.

6. Process States

A process changes its state as it executes. These are the **main states**:

State	Meaning
New	Process is being created
Ready	Process is waiting for CPU
Running	Process is currently being executed by the CPU
Waiting	Process is waiting for an event (like input or file access)
Terminated	Process has finished execution

Think of it like a person:

- **Ready** to work → gets a **chance** to work → **works** → **waits** for a file → **completes** the task.

7. Process Control Block (PCB)

A **PCB** is a **data structure** used by the OS to store all **information about a process**.

PCB Contains:

- **Process ID (PID)** – Unique number to identify the process
- **Process state** – Current state (ready, running, etc.)
- **Program counter** – Address of the next instruction
- **CPU registers** – Contents of processor registers
- **Memory info** – Memory used by the process
- **Accounting info** – CPU usage, time used
- **I/O status** – Files/devices used by the process

Example: Imagine PCB as a folder with all the records about a single employee (process).

8. Threads

A **thread** is the **smallest unit of execution** within a process.
Multiple threads can exist inside a single process, **sharing the same memory** but running independently.

Key Features:

- Threads within a process **share memory, files, and resources**.
- Each thread has its own **program counter**, **registers**, and **stack**.

Process vs Thread:

Feature	Process	Thread
Memory	Has its **own memory**	**Shares memory** with other threads
Communication	Slow, uses inter-process methods	Fast, since they share memory
Overhead	High	Low

Example: In a browser like Chrome:

- Each **tab** may be a thread
- All tabs (threads) share memory but run independently

9. System Calls

System Calls are the way a program requests a service from the operating system.

Think of them as **"asking permission"** from the OS to perform tasks like:

- Creating a file
- Reading from a file
- Allocating memory
- Creating or ending a process
- Communicating with devices

Common System Calls Categories:

Type	Example System Call
Process Control	fork(), exit(), wait()
File Management	open(), read(), write(), close()
Device Management	ioctl(), read(), write()
Information Maintenance	getpid(), alarm(), sleep()
Communication	pipe(), send(), recv()

Example:
When a program needs to open a file, it uses a system call like open(). The OS checks if the file exists and gives access if allowed.

Summary

Topic	Description
Process	A running program with its own memory and resources
States	New, Ready, Running, Waiting, Terminated
PCB	Data structure storing all process info (PID, memory, state, etc.)
Threads	Lightweight sub-processes; share memory but run independently
System Calls	Interface between user programs and OS to request services (e.g., file access)

10. Process Scheduling – Introduction

Process Scheduling is the method used by the operating system to **select which process will run next** on the CPU.
When multiple processes are in the **Ready Queue**, the OS needs to choose one – this is where scheduling comes in.
Goal:

- Maximize CPU usage
- Minimize waiting time
- Improve system performance

10.1. Types of Schedulers

Operating Systems use **three types of schedulers**:

a) Long-Term Scheduler (Job Scheduler)

- Decides **which jobs/processes should be admitted into the system** (from job queue to ready queue).
- Controls the **degree of multiprogramming** (how many processes are in memory).
- Used in batch systems.

Example: Picking 5 out of 10 jobs to load into memory.

b) Short-Term Scheduler (CPU Scheduler)

- Selects **one process from the ready queue** to execute on the CPU.
- Runs **very frequently** (every few milliseconds).
- It is the **most active scheduler**.

Example: Switching between browser and music player quickly.

c) Medium-Term Scheduler

- **Temporarily removes** a process from memory (swapping) to reduce memory load.
- May bring back a suspended process later.

Example: Pausing a game when you minimize it, to free memory.

11. Context Switch

A **Context Switch** is the process of **saving the state of a running process and loading the state of another**.
The OS stores the **current process info in PCB**, and **loads another process's PCB** to resume it.

What is saved during context switch?

- Program counter (next instruction)
- CPU registers
- Memory info
- Process state

Example:
When you switch from a music player to a web browser, the OS saves the state of the music player and loads the browser's process.

12. CPU Scheduling

CPU Scheduling decides **which process gets the CPU next** when multiple are in the **Ready Queue**.
Done by the **short-term scheduler**, based on a scheduling algorithm.

12.1. Preemptive vs Non-Preemptive Scheduling

Preemptive Scheduling

- The OS **can take the CPU away** from a running process.
- Good for **responsive** systems like smartphones and computers.

Example: While watching a video, a phone call interrupts it — video process is paused, call process runs.

Non-Preemptive Scheduling

- Once a process gets the CPU, **it keeps it until it finishes** or waits.
- Simpler but less responsive.

Example: Printer job runs to completion without interruption.

Comparison:

Feature	Preemptive	Non-Preemptive
CPU can be taken?	Yes	No
Response time	Faster	Slower
Complexity	More complex	Simple
Used in	Real-time, interactive systems	Batch systems

Summary

Topic	Description
Process Scheduling	Choosing the next process to run on CPU
Schedulers	Long-term (admit jobs), Short-term (choose next process), Medium-term (pause/resume)
Context Switch	Saving and restoring process state during switching

Topic	Description
CPU Scheduling	Deciding which process in ready queue runs next
Preemptive	CPU can be taken from a process
Non-Preemptive	CPU runs a process till it finishes or waits

13. Scheduling Criteria

Scheduling criteria are **metrics** used to **measure and compare** the performance of different CPU scheduling algorithms.
Here are the most important ones:

13.1. CPU Utilization

- **Definition**: The percentage of time the CPU is **actually working** (not idle).
- **Goal**: Keep it as **high as possible** (ideally 100%, but usually 40–90%).

Example:
If the CPU is busy for 90 out of 100 seconds, CPU utilization is **90%**.

13.2. Throughput

- **Definition**: The **number of processes completed** per unit of time.
- **Goal**: Higher throughput means **more work is being done**.

Example:
If 10 processes finish in 5 seconds, throughput is **2 processes/second**.

13.3. Turnaround Time

- **Definition**: The **total time** taken from **submission of a process to its completion**.

Turnaround Time = Completion Time – Arrival Time

Example:
If a process is submitted at 2:00 PM and completes at 2:10 PM,
Turnaround Time = 10 minutes

13.4. Waiting Time

- **Definition**: The **total time a process spends in the Ready Queue** (not running).

Waiting Time = Turnaround Time – Burst Time (CPU execution time)

Example:
If a process needs 4 minutes to run and turnaround time is 10 minutes,
Waiting Time = 10 – 4 = 6 minutes

13.5. Response Time

- **Definition**: The time between **submission of a request** and the **first time the process gets the CPU**.

This is important in **interactive systems** (e.g., typing in a text editor).

Example:
If you click a button and the system responds after 1 second,
Response Time = 1 second

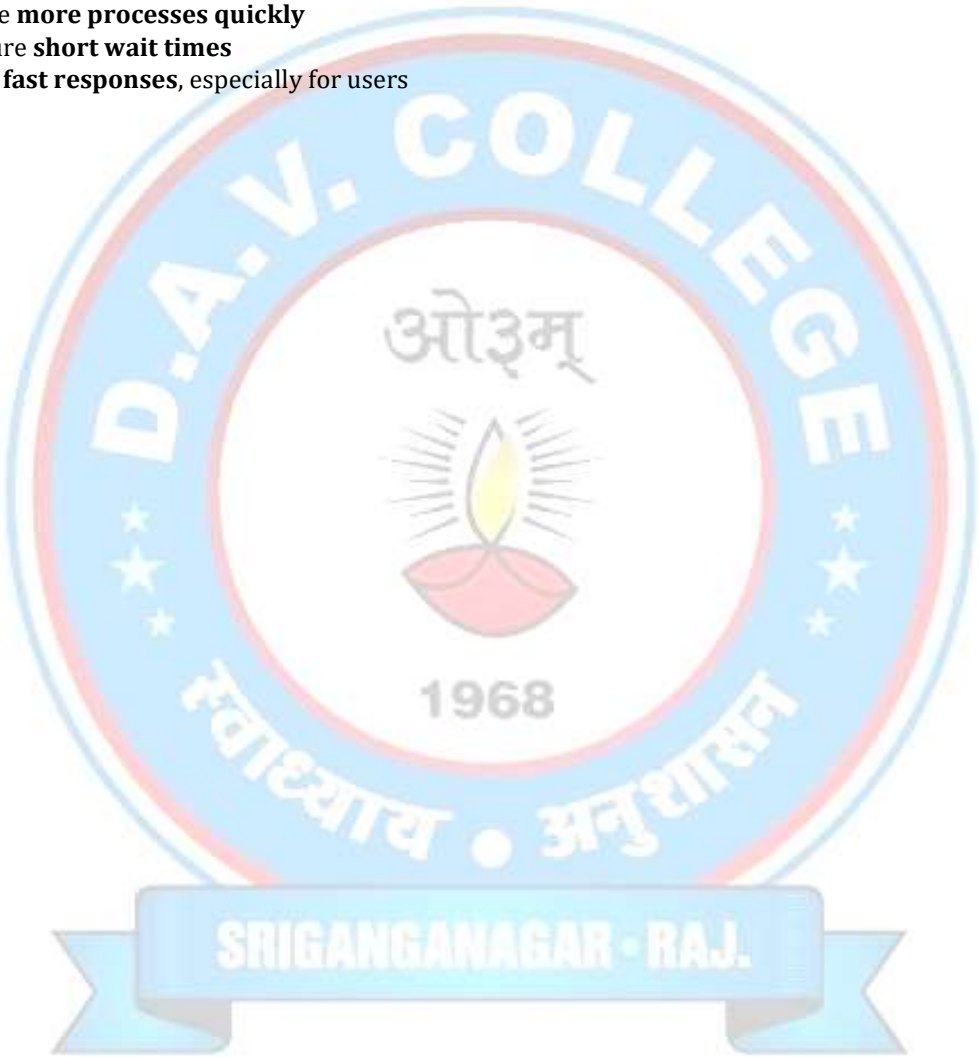
Summary Table:

Criteria	Definition	Goal
CPU Utilization	% of time CPU is active	Maximize (≈100%)
Throughput	# of processes completed per time unit	Maximize

Criteria	Definition	Goal
Turnaround Time	Time from submission to completion of a process	Minimize
Waiting Time	Time spent waiting in ready queue	Minimize
Response Time	Time between request and first CPU allocation	Minimize

Tip:
When designing or choosing a scheduling algorithm, we try to:

- Keep **CPU busy**
- Serve **more processes quickly**
- Ensure **short wait times**
- Give **fast responses**, especially for users



UNIT-II

2.1. FCFS (First-Come, First-Served)

Definition:

- The **first process to arrive** gets the CPU first.
- Simple, like waiting in a queue at a movie ticket counter.

Real-Life Example:

Imagine 3 print jobs come to a printer in this order:

Process	Arrival Time	Burst Time (Print Time)
P1	0 ms	4 ms
P2	1 ms	3 ms
P3	2 ms	2 ms

Execution Order:

- P1 → P2 → P3 (based on arrival time)

Gantt Chart:

CopyEdit

| P1 | P2 | P3 |

0 4 7 9

Calculation:

Process	Arrival	Burst	Start	Finish	Turnaround (FT - AT)	Waiting (TT - BT)
P1	0	4	0	4	4	0
P2	1	3	4	7	6	3
P3	2	2	7	9	7	5

Avg Turnaround Time = (4 + 6 + 7) / 3 = **5.67 ms**

Avg Waiting Time = (0 + 3 + 5) / 3 = **2.67 ms**

2.2. Round Robin (RR)

Definition:

- Each process gets a **fixed time slot** called a **time quantum** (e.g., 2 ms).
- After the time ends, the CPU moves to the **next process**, in a cycle.
- Used in **multitasking systems** (e.g., phones, PCs).

Real-Life Example:

4 apps running in your phone: music, chat, email, and camera.

Let's take this data with **Time Quantum = 2 ms**:

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

Execution Order (time quantum = 2 ms):

- P1 runs for 2 ms → P2 runs for 2 ms → P3 runs for 1 ms (completes)
- Back to P1 (3 left) → P2 (1 left) → P1 (1 left)

Gantt Chart:

| P1 | P2 | P3 | P1 | P2 | P1 |

0 2 4 5 7 8 9

Completion Times:

- P1 finishes at 9
- P2 finishes at 8
- P3 finishes at 5

Process	Arrival	Burst	Completion	Turnaround	Waiting
P1	0	5	9	9 - 0 = 9	9 - 5 = 4
P2	1	3	8	8 - 1 = 7	7 - 3 = 4
P3	2	1	5	5 - 2 = 3	3 - 1 = 2

Avg Turnaround Time = (9 + 7 + 3) / 3 = **6.33 ms**
Avg Waiting Time = (4 + 4 + 2) / 3 = **3.33 ms**

FCFS vs Round Robin — Comparison Table

Feature	FCFS	Round Robin
Type	Non-preemptive	Preemptive
Fairness	Not fair (long jobs delay others)	Fair (equal time slices)
Used in	Batch systems	Interactive systems
Response Time	Can be high	Low (good for user interaction)
Implementation	Simple	Slightly complex (needs timer)

2.3. Shortest Job First (SJF)

Definition:

- The process with the **shortest CPU burst time** is scheduled **first**.
- Can be **non-preemptive** or **preemptive** (also called Shortest Remaining Time First – SRTF).

Real-Life Example:
In a photocopy shop, shorter documents are completed before longer ones to save waiting time for others.

Non-Preemptive SJF Example:

Process	Arrival Time	Burst Time
P1	0	6
P2	2	8
P3	4	7
P4	5	3

Execution Order:

- P1 comes first (runs first)
- At time 6, available: P2, P3, P4 → **P4** has shortest burst (3), so it goes next
- Then P3 (7), then P2 (8)

Gantt Chart:
| P1 | P4 | P3 | P2 |
0 6 9 16 24

Calculations:

Process	Arrival	Burst	Completion	Turnaround	Waiting
P1	0	6	6	6	0
P2	2	8	24	22	14
P3	4	7	16	12	5
P4	5	3	9	4	1

Average Turnaround Time = (6 + 22 + 12 + 4) / 4 = **11 ms**
Average Waiting Time = (0 + 14 + 5 + 1) / 4 = **5 ms**

2.4. Priority Scheduling

Definition:

- Each process is assigned a **priority**.
- The **process with the highest priority** (usually **lower number**) runs first.
- Can be **preemptive or non-preemptive**.

Example:

In hospitals, emergency patients are treated before others (higher priority).

Non-Preemptive Priority Scheduling Example:

Process	Arrival Time	Burst Time	Priority
P1	0	10	3
P2	2	1	1
P3	3	2	4
P4	5	1	2

Lower priority number = Higher priority

Execution Order:

- P1 starts at 0
- At time 2: P2 arrives (priority 1) → P1 is still running (non-preemptive)
- P1 completes, then P2 → P4 → P3

Gantt Chart:

| P1 | P2 | P4 | P3 |
0 10 11 12 14

Calculations:

Process	Arrival	Burst	Completion	Turnaround	Waiting
P1	0	10	10	10	0
P2	2	1	11	9	8
P3	3	2	14	11	9
P4	5	1	12	7	6

Average Turnaround Time = (10 + 9 + 11 + 7) / 4 = **9.25 ms**
Average Waiting Time = (0 + 8 + 9 + 6) / 4 = **5.75 ms**

SJF vs Priority Scheduling – Comparison

Feature	SJF	Priority Scheduling
Basis of choice	Shortest burst time	Highest priority
Fairness	Can starve long processes	Can starve low-priority processes
Preemptive version	Yes (SRTF)	Yes (Preemptive Priority Scheduling)
Used in	Batch systems	Real-time or critical systems

1. FCFS (First Come First Served)

Definition:

- Processes are scheduled **in the order they arrive**.
- Non-preemptive: Once a process starts, it runs until it finishes.

Real-Life Example:

Queue at a bank or cinema counter.

Characteristics:

- **Simple and easy to implement**
- Can cause **long waiting times** if a big process arrives early (convoy effect)

Example:

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

Execution Order: P1 → P2 → P3
Avg Waiting Time: (0 + 4 + 6) / 3 = **3.33 ms**

2. SJF (Shortest Job First)
Definition:

- The process with the **smallest burst time** is executed first.
- Can be **Non-Preemptive** or **Preemptive (SRTF – Shortest Remaining Time First)**

Real-Life Example:
Printing short documents before long ones.

Characteristics:

- **Minimizes average waiting time**
- Can cause **starvation** for long processes

Example:

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

Execution Order (Non-preemptive): P1 → P3 → P2
Avg Waiting Time: (0 + 6 + 8) / 3 = **4.67 ms**

3. Priority Scheduling
Definition:

- Each process is assigned a **priority**, and the **highest priority** process runs first.
- Can be **Preemptive or Non-Preemptive**

Lower number = higher priority (usually)

Real-Life Example:
Emergency room in a hospital (critical patients get treated first).

Characteristics:

- **Flexible** – can prioritize critical tasks
- **Starvation** possible for low-priority processes

Example:

Process	Burst Time	Priority
P1	5	2
P2	3	1
P3	4	3

Execution Order: P2 → P1 → P3
Avg Waiting Time: (0 + 3 + 8) / 3 = **3.67 ms**

4. Round Robin (RR) Scheduling
Definition:

- Each process gets a **fixed time slot** (time quantum).
- If not finished, it's put back in the queue.

Real-Life Example:
Time-sharing in multitasking systems (like tabs in a web browser).

Characteristics:
- **Fair** – all processes get equal CPU time
- Good for **interactive systems**
- Performance depends on **time quantum size**

Example:

Process	Burst Time
P1	5
P2	3
P3	1

Time Quantum = 2 ms
Execution: P1(2) → P2(2) → P3(1) → P1(3) → P2(1)
Avg Waiting Time: ~ 4.33 ms

2.5. Multilevel Queue (MLQ) Scheduling

Definition:
- Processes are **divided into groups (queues)** based on priority or type.
- Each queue has its **own scheduling algorithm**.
- **No process moves between queues**.

Real-Life Example:
College admissions:
- Management quota
- Merit-based quota
- Sports quota

Each has different rules and processing speed.

Characteristics:
- Good for separating **foreground vs background** jobs.
- Can cause **starvation** for lower-priority queues unless carefully designed.

Example:

Queue	Priority	Scheduling
System Tasks	1 (High)	FCFS
User Tasks	2	Round Robin

- CPU gives first preference to system tasks.
- If no system task, then user tasks are scheduled with RR.

Summary Table

Algorithm	Type	Preemptive	Fairness	Best For	Risk
FCFS	Simple Queue	No	Low (convoy effect)	Batch jobs	Long waits
SJF	Shortest Job	Optional	Medium	Min average waiting time	Starvation
Priority	Based on rank	Optional	Low (starvation)	Critical jobs first	Starvation

Algorithm	Type	Preemptive	Fairness	Best For	Risk
Round Robin	Time-sharing	Yes	High	Interactive systems	Overhead if time quantum is too short
MLQ	Multi-queue	Depends	Depends	Multi-user or OS-level jobs	Complex setup

2.6 Synchronization

What is Synchronization?
Synchronization in Operating Systems means **coordinating the execution of processes** so they **do not interfere** with each other when sharing resources like memory, files, or printers.

Why is it important?

- To avoid data inconsistency or errors
- To ensure correct program output in a multi-process system

2.6.1. The Critical Section Problem

What is a Critical Section?
A **critical section** is a part of the code **where a process accesses shared resources** (like variables, files, etc.).
Only one process should be allowed in its critical section at a time.

Example:
Two people editing the same Google Doc at the same time may overwrite each other's text. Synchronization is needed to allow only one person to type at a time.

Conditions to Solve Critical Section Problem

A solution must satisfy these **three conditions**:

Condition	Meaning
1. Mutual Exclusion	Only **one process** can be in the critical section at a time.
2. Progress	If no process is in the critical section, others **should not be blocked**.
3. Bounded Waiting	A process must get a turn to enter critical section **after a limited time**.

Example Scenario (Shared Counter):
Assume two processes P1 and P2 increase a shared variable count = 0.

Without synchronization:
c
// Both processes try to do this:
count = count + 1;

- If both read count as 0 at the same time,
- Both update it to 1, so final result is **1** instead of **2** ✕

With synchronization:
Only one process can read, modify, and write count at a time

How to Implement Synchronization?

There are different ways to solve the critical section problem:

1. Software-based solutions

- **Peterson's Algorithm**
- **Dekker's Algorithm**

Good for 2 processes, but complex and not practical for real systems.

2. Hardware-based solutions

- **Disable interrupts** (only one process runs at a time)
- **Test and Set / Compare and Swap** instructions

Fast but can be risky (no interrupts = less responsive system)

3. OS-level Synchronization Tools

- **Semaphores**
- **Mutex (Mutual Exclusion) Locks**
- **Monitors**

Commonly used in real applications and multithreaded programs.

Basic Example Using a Mutex Lock (in C-like syntax):

```
c
mutex_lock();       // enter critical section
count = count + 1;   // critical section code
mutex_unlock();     // exit critical section
```

Now, **only one process/thread can execute the critical section at a time**.

Summary

Term	Explanation
Synchronization	Controlling access to shared resources in multitasking systems
Critical Section	Code that accesses shared data and must not be executed by 2 processes at once
Mutual Exclusion	Ensures only one process in the critical section at any time
Progress	If no one is in the critical section, someone should get in
Bounded Waiting	A process should not wait forever to get its turn
Tools Used	Semaphores, Mutex, Monitors, Test-and-Set, Peterson's Algorithm

2.6.2. Requirements for a Solution to the Critical Section Problem

1. Mutual Exclusion

- **Definition**: Only **one process** can be inside the **critical section** at a time.
- Prevents **race conditions** (when two processes try to access and change shared data at the same time).

Example:

If Process P1 is updating a shared variable, P2 must wait until P1 finishes.

2. Progress

- **Definition**: If **no process is in the critical section**, and **some processes want to enter**, then one of them must be **allowed to enter without unnecessary delay**.

- The decision of who enters next should **not depend on processes that are not involved**.

Example:
If P1 is in the remainder section (not using the resource), and P2 wants to enter, it should be allowed — P1 shouldn't block P2.

3. Bounded Waiting

- **Definition**: There must be a **limit** on how many times other processes can enter their critical sections **before a waiting process is allowed**.
- Ensures that **no process waits forever** (prevents starvation).

Example:
If P3 has been waiting, it must eventually get its turn — we can't keep letting P1 and P2 go again and again.

Summary Table

Requirement	Purpose	Prevents
Mutual Exclusion	Only one process in the critical section at a time	Data inconsistency
Progress	Allow entry if no one is in the critical section	Unnecessary delays
Bounded Waiting	Limit wait time for every process	Starvation (infinite wait)

A good synchronization solution (like a semaphore or mutex lock) must satisfy all three of these conditions to be reliable and fair.

Peterson's Algorithm
What is it?

- A **software-based solution** to ensure **mutual exclusion** between **two processes**.
- Works without hardware support (like disabling interrupts).
- Uses only **two shared variables** to coordinate processes.

How it works?
Each process indicates:

- It **wants to enter** the critical section (flag[i] = true).
- Sets a shared variable turn to the **other process's id** to let the other process go first if it wants.
- Waits until either the other process is **not interested** (flag[j] = false) or it's **their turn**.

Variables Used:

- flag[2] — Boolean array to indicate if process wants to enter the critical section
- turn — Integer to indicate whose turn it is (0 or 1)

Pseudocode for Process i (where i = 0 or 1, and j = 1 - i):

```
c
flag[i] = true;        // Process i wants to enter critical section
turn = j;              // Give turn to other process

// Wait until either the other process doesn't want to enter OR it's our turn
while (flag[j] == true && turn == j) {
   // busy wait
}
```

```
// Critical Section starts here
// ... (process i's critical section code)

// Exit section
flag[i] = false;        // Process i leaves critical section
```

Intuition:

- If both processes want to enter, turn decides who waits.
- The one whose turn it is waits, while the other proceeds.
- Ensures **mutual exclusion**, **progress**, and **bounded waiting**.

Advantages:

- Simple software solution for two processes.
- No hardware instructions needed.
- Satisfies the 3 requirements of critical section problem.

Limitations:

- Only works for **two processes**.
- Not suitable for multiple processes.

2.7. Semaphores

What is a Semaphore?

- A **semaphore** is a synchronization tool used to control access to shared resources.
- It is basically an **integer variable** that is accessed through two atomic operations:

Operation	Description
wait() (also called P())	Decreases the semaphore value. If value < 0, the process waits (blocks).
signal() (also called V())	Increases the semaphore value. If value ≤ 0, a waiting process is released.

Why use semaphores?

- To **avoid race conditions**.
- To manage **process synchronization** and **mutual exclusion**.

Readers-Writers Problem

Problem statement:

- There is a shared data resource (like a database).
- Multiple **readers** can read the data at the same time.
- Only one **writer** can write at a time.
- Writers require **exclusive access** (no readers or writers during writing).
- Readers should not be kept waiting unnecessarily.

2.7.1. Simple Semaphore Solution for Readers-Writers (First Version)

Variables:

- mutex = 1 — to protect readcount variable
- wrt = 1 — to allow writers exclusive access

- readcount = 0 — number of readers currently reading

Reader Process:

```
wait(mutex);
readcount = readcount + 1;
if (readcount == 1) {
   wait(wrt);    // First reader locks the resource for writers
}
signal(mutex);

// Reading is performed here

wait(mutex);
readcount = readcount - 1;
if (readcount == 0) {
   signal(wrt);  // Last reader releases the resource for writers
}
signal(mutex);
```

Writer Process:

```
wait(wrt);   // Writer wants exclusive access
// Writing is performed here
signal(wrt); // Release exclusive access
```

How does this work?

- The first reader locks the resource for writers by waiting on wrt.
- Multiple readers can enter critical section simultaneously.
- Writers wait until there are no readers (wrt semaphore ensures this).
- When the last reader finishes, it signals wrt, allowing writers to proceed.

Summary

Process	Semaphore Usage
Readers	Use mutex to safely update readcount, and wrt to block writers when readers exist.
Writers	Use wrt to get exclusive access to the resource.

UNIT –III

3.1 Deadlock in Operating Systems

What is Deadlock?

A **deadlock** is a situation where **two or more processes are stuck forever**, each waiting for a resource held by the other(s), so none of them can proceed.

Characterization of Deadlock

Deadlock happens only if **all these four conditions** hold **simultaneously** (also called **Coffman's Conditions**):

Condition	Meaning
1. Mutual Exclusion	At least one resource must be held in a non-sharable mode (only one process at a time).
2. Hold and Wait	A process holds at least one resource and is waiting to acquire additional resources held by others.
3. No Preemption	Resources cannot be forcibly taken from a process; they must be released voluntarily.
4. Circular Wait	There exists a set of processes {P1, P2, ..., Pn} where P1 waits for a resource held by P2, P2 waits for a resource held by P3, ..., Pn waits for a resource held by P1, forming a cycle.

Deadlock Prevention

To **prevent deadlock**, ensure that **at least one of the four conditions is never true**. Examples:

Condition	How to Prevent It
Mutual Exclusion	Make resources sharable (not always possible, e.g., printers).
Hold and Wait	Require processes to request all needed resources at once, or release held resources before requesting new ones.
No Preemption	Allow the system to **preempt** resources from processes if needed.
Circular Wait	Impose a **total ordering** on resource types, and require processes to request resources in increasing order.

Deadlock Avoidance

- Deadlock avoidance **dynamically checks** resource allocation to avoid unsafe states.
- The system **grants resources only if it is safe** (i.e., guarantees no deadlock will occur).
- Uses algorithms like **Banker's Algorithm**.

3.2 Banker's Algorithm (Brief)

- Think of the system as a bank lending money (resources).
- Each process declares the **maximum resources** it may need.
- The system **grants resource requests only if safe**, meaning there is some sequence in which all processes can complete.
- If granting a request leads to an **unsafe state**, the request is delayed.

Summary Table

Aspect	Description

Aspect	Description
Deadlock	Processes stuck waiting forever for each other's resources
Characterization	Mutual exclusion, hold & wait, no preemption, circular wait
Prevention	Design system to break at least one deadlock condition
Avoidance	Dynamically allocate resources to avoid unsafe states (Banker's)

Banker's Algorithm — Detailed Explanation

What is Banker's Algorithm?

- It's a **deadlock avoidance algorithm**.
- Named after the banking system where the bank lends money carefully to customers, ensuring it never runs out.
- The algorithm **checks every resource request** from processes and only grants it if the system remains in a **safe state** after allocation.

Key Concepts:

- **Safe State:** The system can allocate resources in some order to all processes and still avoid deadlock.
- **Unsafe State:** No guaranteed safe sequence exists, so deadlock *might* occur.

Data Structures Used:

- **Available:** Vector of available resources of each type.
- **Max:** Matrix specifying the maximum demand of each process.
- **Allocation:** Matrix showing current allocated resources for each process.
- **Need:** Matrix showing remaining resource needs for each process, calculated as:
  Need[i][j] = Max[i][j] - Allocation[i][j]

Algorithm Steps:

1. **When a process requests resources:**
   - Check if request ≤ Need (request doesn't exceed max demand)
   - Check if request ≤ Available (resources are available)
2. **Pretend to allocate** the requested resources:
   - Available = Available - Request
   - Allocation = Allocation + Request
   - Need = Need - Request
3. **Check system state for safety:**
   - Try to find a sequence of processes that can finish with current available resources.
   - If such a sequence exists → **safe state**, grant request.
   - Otherwise → **unsafe state**, process must wait.

Example

Given:

- 3 Processes: P0, P1, P2
- 3 Resource types: A, B, C

Process	Max Demand (A,B,C)	Allocation (A,B,C)	Need (Max - Allocation)
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)
P1	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)

Process	Max Demand (A,B,C)	Allocation (A,B,C)	Need (Max - Allocation)
P2	(9, 0, 2)	(3, 0, 2)	(6, 0, 0)

- Available resources = (3, 3, 2)

Step 1: Is system safe now?

- Work = Available = (3, 3, 2)
- Finish = [false, false, false]

Check if any process's Need ≤ Work:

- P1 Need = (1, 2, 2) ≤ Work (3, 3, 2) → Yes
  P1 can finish, so:
  Work = Work + Allocation[P1] = (3,3,2) + (2,0,0) = (5,3,2)
  Finish[P1] = true
- P0 Need = (7,4,3) ≤ Work (5,3,2) → No
- P2 Need = (6,0,0) ≤ Work (5,3,2) → No

No other processes can finish yet.

- Since no other can finish, the system is **not safe** with current work.

Step 2: Suppose P1 requests (1,0,2)

- Request ≤ Need[P1]: (1,0,2) ≤ (1,2,2) → Yes
- Request ≤ Available: (1,0,2) ≤ (3,3,2) → Yes

Pretend allocation:

- Available = (3,3,2) - (1,0,2) = (2,3,0)
- Allocation[P1] = (2,0,0) + (1,0,2) = (3,0,2)
- Need[P1] = (1,2,2) - (1,0,2) = (0,2,0)

Check safe state again:

- Work = Available = (2,3,0)
- Finish = [false, false, false]

Check processes:

- P1 Need = (0,2,0) ≤ Work (2,3,0) → Yes
  Work = Work + Allocation[P1] = (2,3,0) + (3,0,2) = (5,3,2)
  Finish[P1] = true
- P0 Need = (7,4,3) ≤ Work (5,3,2) → No
- P2 Need = (6,0,0) ≤ Work (5,3,2) → No

No other processes can finish.

Step 3: Conclusion

- P1 can finish, releasing resources.
- However, P0 and P2 cannot finish now, so **the system is unsafe** if we grant this request.

Therefore, P1's request must wait.

Summary

- Banker's Algorithm **checks resource requests carefully**.
- Only grants requests if the system **remains in a safe state**.
- Helps **avoid deadlocks** by **rejecting unsafe resource allocations**.

3.3. Recovery from Deadlock

When deadlock occurs, the system needs to **recover** to get processes running again. There are several methods to do this:

1. Process Termination

a) Abort all deadlocked processes

- The simplest approach.
- Terminate every process involved in the deadlock.
- Frees all resources, but can cause data loss and waste.

b) Abort processes one at a time

- Terminate one process at a time.
- After each termination, check if deadlock is resolved.
- Repeat until deadlock is cleared.
- More controlled but can still cause data inconsistency.

2. Resource Preemption

- Temporarily take resources away from some processes and give them to others to break the deadlock cycle.
- Requires:
    - **Selecting victim**: Choose which process to preempt resources from (based on priority, cost, or process state).
    - **Rollback**: The victim process might have to restart or roll back to a safe state.
- Risky because preempted processes lose progress and system complexity increases.

3. Process Rollback

- Roll back one or more processes to a previously saved safe state (using checkpoints).
- Release resources allocated after the checkpoint.
- Processes resume from that checkpoint, avoiding deadlock.
- Requires processes to support rollback and checkpoints.

4. Combined Approaches

- Use a mix of the above methods to recover efficiently.
- For example, preempt resources from low-priority processes and terminate them if rollback is not possible.

Challenges in Recovery

- **Selecting victim** fairly is hard.
- Rollbacks or termination can cause inconsistent data.
- Recovery may be costly in terms of performance and user experience.

Summary Table

Recovery Method	Description	Pros	Cons
Abort all processes	Kill all deadlocked processes	Simple	Data loss, wasted work
Abort one process	Kill one process at a time until fixed	More controlled	May still lose data
Resource Preemption	Take resources from some processes	Can resolve deadlock	Complex, may need rollback

Recovery Method	Description	Pros	Cons
Process Rollback	Rollback to safe state	Avoids full termination	Requires checkpointing

3.4. Memory Management

Memory management is a key job of an operating system (OS). It controls how memory is allocated, used, and freed, ensuring programs run efficiently and safely.

1. Physical and Virtual Address Space

Physical Address Space

- This refers to the actual **physical memory (RAM)** installed in the computer.
- When a program runs, it uses physical addresses to access memory locations.
- Limited by the size of RAM.

Virtual Address Space

- Virtual memory is an **abstraction** that gives each process its own independent address space.
- The OS uses **virtual addresses**, which the hardware and OS translate to physical addresses.
- Virtual memory allows a process to use **more memory than physically available** by using disk space (swap or page file).
- It helps protect processes from interfering with each other by keeping their memory separate.

2. Paging

Paging is a popular technique for implementing virtual memory.

What is Paging?

- Divides both virtual memory and physical memory into fixed-size blocks:
  - **Pages** (virtual memory blocks)
  - **Frames** (physical memory blocks)
- Size of each page/frame is the same (e.g., 4 KB).

How Paging Works

- When a process wants to access a memory address, the OS:
  1. Divides the virtual address into:
     - **Page number** (which page)
     - **Page offset** (location within the page)
  2. Uses a **page table** to find the corresponding frame in physical memory.
  3. Combines the frame number with the offset to get the **physical address**.
- This **mapping** is done by hardware with the help of the OS.

Benefits of Paging

- Eliminates **external fragmentation** (since pages/frames are fixed size).
- Supports **virtual memory** by allowing pages to be swapped between RAM and disk.
- Simplifies memory allocation.

Example:

- Virtual address = 32-bit number
- Page size = 4 KB = 2^{12} bytes
- So, lower 12 bits = offset, upper bits = page number

- If virtual address = 0x00003FA7,
    - Page number = upper bits (virtual address / page size)
    - Offset = lower 12 bits (address within page)

Summary Table

Term	Meaning
Physical Address	Actual address in RAM
Virtual Address	Address used by a process, translated to physical
Page	Fixed-size block of virtual memory
Frame	Fixed-size block of physical memory
Page Table	Maps pages to frames

Page Replacement Algorithms

Why do we need page replacement?

- When a process tries to access a page **not in physical memory** (a **page fault**), the OS must load that page.
- If physical memory is full, the OS must **replace (remove)** an existing page to make space.
- The **page replacement algorithm** decides **which page to remove**.

Common Page Replacement Algorithms

1. FIFO (First-In, First-Out)

- Replace the page that has been in memory the **longest** (the oldest page).
- Simple to implement with a queue.
- **Downside:** Can remove a heavily used page, leading to poor performance.

2. LRU (Least Recently Used)

- Replace the page that **hasn't been used for the longest time**.
- Assumes that pages used recently will be used again soon.
- More efficient than FIFO but **harder to implement** because it requires tracking page usage history.

3. Optimal (OPT) Algorithm

- Replace the page that **will not be used for the longest time in the future**.
- Provides the **best possible performance**.
- Not practical since it needs future knowledge of page references.
- Used as a benchmark to compare other algorithms.

4. Clock (Second Chance) Algorithm

- A practical approximation of LRU.
- Pages are arranged in a circular list (like a clock).
- Each page has a **reference bit**.
- When replacing, the algorithm checks the reference bit:
    - If 0 → replace this page.
    - If 1 → give it a second chance, clear the bit, and move on.
- Efficient and simple to implement.

5. LFU (Least Frequently Used)

- Replace the page that has been **used least frequently** over time.
- Can be unfair to pages used heavily in the past but not recently.

Summary Table

Algorithm	How it Works	Pros	Cons
FIFO	Replace oldest page	Simple to implement	Can remove frequently used pages
LRU	Replace least recently used page	Good performance	Complex to implement
Optimal	Replace page not needed for longest future time	Best possible performance	Requires future knowledge
Clock	Approximate LRU using reference bits	Efficient, easy to implement	Not always perfect LRU
LFU	Replace least frequently used page	Considers usage frequency	Can remove recently important pages

3. Overview of Segmentation

What is Segmentation?

- **Segmentation** is a memory management technique where a program is divided into **different logical segments** instead of fixed-size pages.
- Each segment represents a **meaningful unit** such as:
  - A function or procedure
  - A data array
  - A stack
  - A symbol table

How Segmentation Works

- Each segment has a **variable length** (unlike fixed-size pages).
- Programs use **two values to access memory**:
  - **Segment number** (which segment)
  - **Offset** within that segment
- The operating system keeps a **segment table** for each process:
  - The segment table stores the **base address** (starting physical address) and the **length** of each segment.

Address Translation in Segmentation

- A logical address consists of (segment number, offset).
- To get the physical address:
  - Check if offset < segment length (to avoid invalid access).
  - Physical address = base address of the segment + offset.

Advantages of Segmentation

- Supports **modularity** by keeping related data and code together.
- Easier to share and protect segments between processes.
- More natural for programmers because it reflects program structure.

Disadvantages of Segmentation

- Can lead to **external fragmentation** because segments vary in size.
- Memory allocation is more complex than paging.

Example

Segment No.	Base Address	Length (bytes)
0	1000	500
1	2000	1000
2	3500	300

- Logical address (1, 100) → physical address = 2000 + 100 = 2100
- If offset 1200 requested in segment 1 → invalid, since 1200 > length 1000

Summary

Term	Meaning
Segmentation	Dividing program into logical variable-sized segments
Segment Table	Stores base and length of each segment
Logical Address	(Segment number, offset)
Physical Address	Base address + offset

Segmentation with Paging

What is it?

- It's a **combination of segmentation and paging**.
- Each **segment is divided into fixed-size pages**.
- The system manages memory as **segments of pages**, instead of segments of variable length or just pages alone.

Why Combine Segmentation and Paging?

- **Segmentation** provides a logical view of memory (program structure).
- **Paging** solves external fragmentation by dividing memory into fixed-size blocks.
- Combining both means:
    - Programs still have logical segments.
    - But segments are split into pages to avoid external fragmentation.
    - Memory allocation is easier and more flexible.

How It Works

1. **Segment Table:**
    - Each process has a segment table.
    - Each segment entry points to a **page table** (instead of directly to a memory location).
2. **Page Table for Each Segment:**
    - The page table maps pages of that segment to physical frames.
3. **Address Translation:**
    - Logical address = (Segment Number, Page Number, Offset)
    - Step 1: Use segment number to find the segment table entry → get page table.

- Step 2: Use page number to find frame number from page table.
- Step 3: Physical address = (frame number × frame size) + offset.

Benefits

- Combines **logical organization** (segments) with **efficient memory use** (paging).
- Reduces external fragmentation (paging).
- Supports protection and sharing at segment level.
- Easier to grow segments dynamically by adding more pages.

Example

Suppose:

- Segment 1 has 3 pages.
- Segment 2 has 2 pages.

For a logical address (Segment 1, Page 2, Offset 100):

- Look up Segment 1 in segment table → points to Segment 1's page table.
- Page 2 in Segment 1's page table → mapped to physical frame 5.
- Physical address = (frame 5 × frame size) + 100.

Summary Table

Component	Role
Segment Table	Maps segment number to a page table
Page Table	Maps page number within segment to frame
Logical Address	(Segment number, Page number, Offset)
Physical Address	Calculated using frame number and offset

4. Virtual Memory Management

Concept

- **Virtual Memory** is a technique that allows a computer to use more memory than physically available by temporarily transferring data between RAM and disk storage.
- It creates an illusion for programs that they have **large, continuous memory**, even if physical RAM is limited.
- When a program accesses data not in RAM (a **page fault**), the OS loads the required page from disk.
- If RAM is full, the OS must **replace** some page in memory to load the new page — this is where **page replacement algorithms** come in.

Page Replacement Techniques

When memory is full, the OS decides which page to remove to make room for the new page. Here are common algorithms:

1. FIFO (First-In, First-Out)

- **Idea:** Remove the page that has been in memory the longest.
- Maintains a queue of pages.
- Simple but may remove frequently used pages, causing poor performance (known as **Belady's anomaly**).

2. LRU (Least Recently Used)

- **Idea:** Remove the page that hasn't been used for the longest time.
- Assumes pages used recently will be used again soon.
- More efficient than FIFO but harder to implement because it requires tracking usage history.

3. Optimal

- **Idea:** Remove the page that **won't be used for the longest time in the future**.
- The best possible algorithm for minimizing page faults.
- Not implementable in practice because it needs future knowledge.
- Used as a benchmark to compare other algorithms.

Example of Page Replacement (Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2)

Assuming 3 frames in RAM:

Step	Pages in Frame (FIFO)	Page Fault?	Pages in Frame (LRU)	Page Fault?	Pages in Frame (Optimal)	Page Fault?
1	7	Yes	7	Yes	7	Yes
2	7, 0	Yes	7, 0	Yes	7, 0	Yes
3	7, 0, 1	Yes	7, 0, 1	Yes	7, 0, 1	Yes
4	0, 1, 2	Yes (7 out)	0, 1, 2	Yes (7 out)	0, 1, 2	Yes (7 out)
5	0, 1, 2	No	0, 1, 2	No	0, 1, 2	No
6	1, 2, 3	Yes (0 out)	1, 2, 3	Yes (0 out)	1, 2, 3	Yes (0 out)
...	...	...	...	...	...	...

Summary Table

Algorithm	How it Works	Pros	Cons
FIFO	Removes oldest page	Simple to implement	Can remove frequently used pages (poor performance)
LRU	Removes least recently used page	Good performance	Requires tracking usage, complex to implement
Optimal	Removes page not used longest ahead	Best possible performance	Impossible to implement practically

UNIT –IV

4.1 Linux Overview

Linux is an open-source, Unix-like operating system based on the Linux kernel. It is widely used in servers, desktops, mobile devices (like Android), and embedded systems.

Features of Linux

1. **Open Source:**
   - Linux is free to use and its source code is publicly available.
2. **Multitasking:**
   - It can run multiple processes simultaneously without interference.
3. **Multi-user Capability:**
   - Multiple users can access system resources like memory and applications without affecting each other.
4. **Security:**
   - Built-in features like user authentication, file permissions, and firewalls provide strong security.
5. **Portability:**
   - Can run on different hardware platforms (x86, ARM, PowerPC, etc.).
6. **Stability and Performance:**
   - Rarely crashes; performs well under heavy loads.
7. **Customizability:**
   - You can modify the kernel and system components to suit your needs.
8. **Shell/Command Line Interface:**
   - Powerful command-line interface for advanced users and automation.
9. **Support for Networking:**
   - Strong networking features for server and network management.
10. **Wide Range of Distributions:**
    - Examples include Ubuntu, Fedora, Debian, CentOS, Arch Linux, etc.

4.2. Steps of Linux Installation (Example: Ubuntu)

Prerequisites:

- A computer (with minimum 2GB RAM, 25GB disk space)
- USB/DVD with Ubuntu ISO image
- Internet connection (optional but useful)

Step-by-Step Installation Guide

Step 1: Download Ubuntu ISO

- Go to the official site: https://ubuntu.com/download
- Choose the desired version (e.g., Ubuntu Desktop 22.04 LTS)

Step 2: Create a Bootable USB Drive

Use tools like:

- **Rufus (Windows)**
- **Etcher or UNetbootin (Linux/Mac)**

Example using **Rufus**:

1. Open Rufus
2. Select USB device
3. Select Ubuntu ISO

4. Click **Start**

Step 3: Boot from USB

- Restart your computer
- Enter BIOS/UEFI (usually by pressing F2, F12, or DEL)
- Set USB as the first boot device
- Save and exit BIOS

Step 4: Try or Install Ubuntu

- On booting, you'll see an option:
  - **Try Ubuntu** (live session)
  - **Install Ubuntu** (proceeds to installation)

Step 5: Select Keyboard Layout

- Choose the correct keyboard layout (e.g., English (US))

Step 6: Updates and Other Software

- Choose whether to download updates and install third-party software (e.g., Wi-Fi drivers)

Step 7: Installation Type

- Choose:
  - **Erase disk and install Ubuntu** (clean install)
  - **Install alongside existing OS** (dual boot)
  - **Manual partitioning** (advanced users)

Choose carefully—erasing disk deletes all data.

Step 8: Partition Setup (if manual)

- Create partitions:
  - / (root) - Minimum 20GB
  - swap - Usually 1-2x RAM
  - /home (optional) - For user files

Step 9: Set Timezone

- Select your region (e.g., Asia/Kolkata)

Step 10: Create User Account

- Enter your name, username, and password

Step 11: Installation Begins

- Files are copied and system configured
- Takes ~10–20 minutes

Step 12: Restart and Login

- Remove USB when prompted
- Login with your credentials

Conclusion

Linux offers a robust, secure, and flexible operating system environment. Installing it is straightforward, especially with user-friendly distributions like Ubuntu. Whether for personal use, servers, or development, Linux is a powerful choice.

4.3 What is the Kernel?

The **kernel** is the **core component of the Linux operating system**. It is responsible for managing the system's **hardware** and allowing **software** to communicate with it. The kernel runs at the lowest level of the OS, typically in a privileged mode known as **kernel space**, giving it full control over the hardware.

Main Responsibilities of the Kernel:

1. **Process Management**
   - Handles creation, scheduling, and termination of processes.
   - Ensures fair CPU time allocation using various **scheduling algorithms**.
2. **Memory Management**
   - Allocates and deallocates RAM for processes.
   - Manages **virtual memory** and **swap space**.
   - Prevents processes from interfering with each other's memory (protection).
3. **Device Management**
   - Controls all connected hardware devices (like hard disks, USB, network cards).
   - Communicates through **device drivers**, which are specific to each hardware type.
4. **File System Management**
   - Manages how data is stored and retrieved on storage devices.
   - Supports various file systems like **ext4**, **xfs**, **btrfs**, and more.
5. **Security and Access Control**
   - Manages user permissions and enforces security policies.
   - Provides system calls to allow or restrict access to resources.
6. **System Calls Interface**
   - Provides a set of **APIs (system calls)** through which user applications interact with hardware.
   - Examples: read(), write(), fork(), exec(), etc.

Types of Kernels:

- **Monolithic Kernel (used in Linux):** All essential OS services run in the kernel space.
- **Microkernel:** Only basic services run in the kernel; others run in user space.
- **Hybrid Kernel:** Combination of both (e.g., Windows NT).

4.4. What is the Shell?

The **shell** is a **command-line interpreter (CLI)** or a **graphical interface** that acts as an **interface between the user and the kernel**. While the kernel works in the background, the shell is what users interact with directly.

Role of the Shell:

- Accepts user input (commands)
- Interprets them
- Passes them to the kernel for execution
- Displays the output back to the user

Functions of the Shell:

1. **Command Execution**
   - Executes system commands like ls, cp, mkdir, echo, etc.
2. **Shell Scripting**
   - Allows users to automate tasks using shell scripts (e.g., .sh files).
   - Includes logic like loops, conditionals, variables, and functions.
3. **Job Control**
   - Lets users run tasks in the foreground or background using commands like bg, fg, jobs, and kill.
4. **Input/Output Redirection**

- Directs output to files or input from files.
- Example: ls > file.txt, cat < file.txt

5. **Pipelines**
   - Connects multiple commands together using | (pipe).
   - Example: ls -l | grep "test"
6. **Environment Management**
   - Manages environment variables like PATH, HOME, USER.

Types of Shells in Linux:

Shell Type	Description	Command to Start
Bash (Bourne Again Shell)	Most common, default on many Linux distros	bash
sh (Bourne Shell)	Original Unix shell, very portable	sh
zsh (Z Shell)	Advanced, customizable, better for interactive use	zsh
ksh (Korn Shell)	Combines features of sh and csh	ksh
fish (Friendly Interactive Shell)	User-friendly, rich features	fish

Shell vs Kernel: Key Differences

Feature	Shell	Kernel
Role	Interface between user and kernel	Core part of OS managing hardware
Function	Executes user commands	Manages hardware, processes, memory
Location	Runs in **user space**	Runs in **kernel space**
Types	bash, zsh, fish, ksh	monolithic, microkernel
User Interaction	Direct	Indirect (via shell or programs)

Interaction Between Shell and Kernel
Example:
Let's say the user runs this command:

$ cp file1.txt /home/user/Documents/

Step-by-step interaction:

1. **User** types the command in the **shell**.
2. The **shell** parses the command.
3. The shell makes a **system call** to the **kernel** requesting to copy a file.
4. The **kernel** accesses the file system and performs the operation.
5. The result (success or error) is returned to the shell.
6. The **shell** displays the result to the **user**.

User → Shell → Kernel → Hardware
↑ ↓
Output ← Result

Conclusion

- The **kernel** is the heart of the Linux system that manages hardware and system processes.

- The **shell** is the user interface that lets users communicate with the kernel.
- Both are essential: **the shell gives users control**, and **the kernel gives the system life**.

4.5 Linux Directory Structure

Linux uses a **hierarchical directory structure**, which starts from a single root directory: /.
This structure is called a **tree-like file system**, where everything is a file — including devices and processes.
Key Directories in Linux:

Directory	Description
/	Root directory; the starting point of the file system. All files and directories stem from here.
/bin	Contains essential binary executables (e.g., ls, cp, mkdir) needed for basic system operation.
/boot	Holds boot loader files and Linux kernel (e.g., vmlinuz).
/dev	Contains special device files (e.g., /dev/sda for disks, /dev/tty for terminals).
/etc	Configuration files for system-wide settings and installed software.
/home	User home directories (e.g., /home/alice, /home/bob).
/lib	Shared libraries needed by binaries in /bin and /sbin.
/media	Mount point for removable media (USB, CD-ROM).
/mnt	Temporary mount directory for external devices or file systems.
/opt	Optional software packages or third-party applications.
/proc	Virtual file system providing process and system information (e.g., /proc/cpuinfo).
/root	Home directory for the root user (not to be confused with /).
/run	Runtime data for processes since system startup.
/sbin	System binaries for administrative tasks (e.g., shutdown, reboot).
/srv	Data for services like FTP, HTTP (e.g., web server files).
/sys	Interface to the kernel for system devices and drivers.
/tmp	Temporary files; cleared on reboot.
/usr	User applications and files (includes /usr/bin, /usr/lib, /usr/share).
/var	Variable data files, logs, emails, spool files (e.g., /var/log).

4.6 Linux Users

A **user** in Linux is an account used to log in, execute processes, and access files.
Types of Users:

Type	Description
Root	The **superuser** with full system access. Can perform any operation.
System/User Accounts	Used by system services and daemons (e.g., nobody, www-data).
Regular Users	Human users with limited permissions (e.g., students, developers).

User Information is stored in:

- /etc/passwd → Basic user information.
- /etc/shadow → Encrypted passwords.
- /etc/group → Group membership.

- /home/username → Personal files and settings.

User Management Commands:

```
adduser alice       # Create user 'alice'
passwd alice        # Set password for 'alice'
deluser alice       # Delete user 'alice'
```

4.7. Linux Groups

A **group** is a collection of users with shared permissions. Groups simplify managing file access and security.

Purpose of Groups:

- Assign permissions to a set of users at once.
- Manage access to files, directories, and devices.

Group Types:

Type	Description
Primary Group	Each user has one primary group (usually same name as username).
Secondary Group	A user can be part of multiple secondary groups.

Group Information is stored in:

- /etc/group

Group Management Commands:

```
addgroup devs           # Create a group 'devs'
usermod -aG devs alice  # Add 'alice' to group 'devs'
delgroup devs           # Delete group 'devs'
```

4.8. Permissions: How Users and Groups Interact with Files

Each file/directory in Linux has three permission sets:

- **Owner (user)**
- **Group**
- **Others (everyone else)**

Permission Example:

```
-rwxr-xr-- 1 alice devs 1234 Jul 4 12:00 script.sh
```

Position	Meaning
alice	File owner (user)
devs	Group owner
rwx	Owner permissions: read, write, execute
r-x	Group permissions: read, execute
r--	Others: read only

Summary Table

Concept	Description
Kernel	Core of the OS; interacts directly with hardware.
Shell	Interface between user and kernel (e.g., Bash).
Directory Structure	Hierarchical layout starting from /.

Concept	Description
User	Account used to log into and use the system.
Group	Set of users with shared access rights.

4.8.1. File Permissions in Linux

In Linux, every file and directory has **permissions** that determine who can read, write, or execute them.

Permission Categories

Each file has **three types of users**:

- **User (u)** – Owner of the file
- **Group (g)** – Members of the group associated with the file
- **Others (o)** – Everyone else

And **three types of permissions**:

- r – **Read**: View file contents or list directory contents
- w – **Write**: Modify file or directory
- x – **Execute**: Run the file (for scripts or programs) or enter a directory

Permission Notation Example:

-rwxr-xr--

Symbol	Meaning
-	File type (- = file, d = directory)
rwx	Permissions for **user** (read, write, execute)
r-x	Permissions for **group** (read, execute)
r--	Permissions for **others** (read only)

4.9. Linux Commands with Examples

File and Directory Management

ls – List directory contents

```
ls          # List files in current directory
ls -l       # Long listing with permissions
ls -a       # Show hidden files
```

cd – Change directory

```
cd /home/user   # Move to specified directory
cd ..           # Move up one level
cd ~            # Go to home directory
```

pwd – Print working directory

```
pwd         # Shows current directory path
```

mkdir – Make directory

```
mkdir newfolder     # Create a new directory
mkdir -p a/b/c      # Create nested directories
```

rmdir – Remove empty directory

```
rmdir oldfolder     # Deletes directory if empty
```

rm – Remove files/directories

```
rm file.txt          # Delete a file
rm -r myfolder       # Delete directory and contents
rm -f file.txt       # Force delete without prompt
```

mv – Move or rename files

```
mv file.txt /home/user/     # Move file
mv oldname.txt newname.txt  # Rename file
```

cp – Copy files/directories

```
cp file.txt /home/user/     # Copy file
cp -r folder/ backup/       # Copy folder recursively
```

Viewing and Editing Files

cat – Display file contents

```
cat file.txt         # Show contents of file
cat file1 file2      # Concatenate and display multiple files
```

chmod – Change file permissions

```
chmod +x script.sh       # Add execute permission
chmod 755 file.sh        # Set permissions using octal format
```

Octal Permissions Table:

Value	Permission
7	rwx
6	rw-
5	r-x
4	r--
0	---

Example:

```
chmod 755 myscript.sh
# User: rwx, Group: r-x, Others: r-x
```

4.10. System and Help Commands

man – Manual pages

```
man ls               # Show manual for ls command
```

apt – Package manager (for Debian/Ubuntu)

```
sudo apt update      # Update package list
sudo apt upgrade     # Upgrade all installed packages
sudo apt install vim # Install a package
```

cal – Calendar

```
cal             # Display current month calendar
cal 2025        # Show calendar for the year 2025
```

uname – System information

```
uname           # Show kernel name
uname -a        # All system info
```

history – Show command history

```
history         # Display previously run commands
!45             # Run command number 45 from history
```

Summary Table of Commands

Command	Description
ls	List directory contents
cd	Change directory
pwd	Print working directory
mkdir	Create directory
rmdir	Remove empty directory
rm	Remove file/directory
mv	Move or rename file
cp	Copy file/directory
cat	Display file contents
chmod	Change permissions
man	Show manual page
apt	Install/update packages
cal	Show calendar
uname	Show system info
history	Show command history

4.10.1 Installing Packages in Linux

A **package** is a collection of files needed to install and run a software application. Linux uses **package managers** to handle the installation, upgrade, configuration, and removal of software.

Types of Package Managers (by Distribution)

Distribution	Package Manager	File Type	Command
Ubuntu/Debian	APT (Advanced Packaging Tool)	.deb	apt or dpkg
Fedora/RHEL/CentOS	DNF or YUM	.rpm	dnf or yum
Arch Linux	Pacman	.pkg.tar.zst	pacman
SUSE Linux	Zypper	.rpm	zypper

Installing Packages on Ubuntu/Debian (APT-based)

Update the system package list:
sudo apt update

Upgrade all installed packages:
sudo apt upgrade

Install a package:
sudo apt install package-name

Example:
sudo apt install vlc

Remove a package:
sudo apt remove package-name

Search for a package:
apt search package-name

View details of a package:
apt show package-name

Using dpkg for .deb Files

Install a downloaded .deb package:
sudo dpkg -i package-name.deb

Fix broken dependencies:
sudo apt --fix-broken install

Installing Packages on Fedora/CentOS (DNF/YUM)

Using dnf (recommended for Fedora, RHEL 8+, CentOS 8+):
sudo dnf install package-name

Using yum (older systems):
sudo yum install package-name

Example:
sudo dnf install firefox

Installing Packages on Arch Linux (Pacman)

Update repository:
sudo pacman -Syu

Install a package:

sudo pacman -S package-name

Example:
sudo pacman -S neofetch

Installing from Source (Generic Linux Method)
For advanced users or software not in repositories:

Example flow:
./configure
make
sudo make install

Using Snap (Universal Linux Package Manager)

Install Snap (if not already installed):
sudo apt install snapd

Install a Snap package:

sudo snap install package-name

Example:
sudo snap install spotify

Using Flatpak (Another Universal Manager)

Install Flatpak:
sudo apt install flatpak

Add Flathub repository:
flatpak remote-add --if-not-exists flathub https://flathub.org/repo/flathub.flatpakrepo

Install an app:
flatpak install flathub org.gimp.GIMP

Summary Commands Table

Task	APT	DNF	Pacman
Install package	apt install	dnf install	pacman -S
Remove package	apt remove	dnf remove	pacman -R
Update repo	apt update	dnf check-update	pacman -Sy
Upgrade system	apt upgrade	dnf upgrade	pacman -Syu

UNIT-V

5.1 Shell Scripting in Linux

A **shell script** is a text file that contains a series of **commands**. It is used to automate tasks such as file operations, backups, installing software, or running programs.

Basic Structure of a Shell Script

```
#!/bin/bash
# This is a comment
echo "Hello, World!"
```

Explanation:

- #!/bin/bash → Shebang line: tells the system to use **Bash shell** to run the script.
- # → Lines starting with # are comments.
- echo → Displays output.

5.1.1. Steps to Write and Execute a Shell Script

Step 1: Create the Script File

Use a text editor like nano, vi, or gedit:

nano myscript.sh

Paste the following simple script:

```
#!/bin/bash
echo "Welcome to Linux Shell Scripting"
date
```

Press Ctrl + O to save and Ctrl + X to exit if using nano.

Step 2: Make the Script Executable

chmod +x myscript.sh

This gives the script execute permission.

Step 3: Run the Script

Option 1: Using ./

./myscript.sh

Option 2: Using bash

bash myscript.sh

Example Shell Script: Greeting Script

```
#!/bin/bash
echo "Enter your name:"
read name
echo "Hello, $name! Today is $(date +%A)."
```

Save it as greet.sh and run it:

chmod +x greet.sh

./greet.sh

5.1.2. Common Shell Script Features

1. Variables

```
name="Alice"
echo "Hello, $name"
```

2. User Input

```
read -p "Enter a number: " num
echo "You entered $num"
```

3. If Statements

```
if [ $num -gt 10 ]; then
  echo "Greater than 10"
else
  echo "10 or less"
fi
```

4. Loops

```
for i in 1 2 3
do
  echo "Number: $i"
done
```

Tips for Shell Scripting

- Always start scripts with #!/bin/bash
- Use comments to describe steps
- Test scripts with sample input
- Use set -x to debug (prints commands before execution)

Where to Store Shell Scripts

You can keep scripts in a custom folder like ~/scripts/, and add it to your PATH for easy execution:

```
mkdir ~/scripts
mv myscript.sh ~/scripts
echo 'export PATH="$HOME/scripts:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

Now you can run your script from anywhere by just typing:

```
myscript.sh
```

shell scripting concepts:

- **Shell Variables**
- **read and expr**
- **Decision Making (if-else, case)**
- **Loops (for, while)**

5.1.2.1. Shell Variables

What Are Shell Variables?
Shell variables are used to **store values** like strings, numbers, or paths for use in shell scripts.

Defining and Using Variables
bash
CopyEdit
name="Alice" # No spaces around '='
echo "Hello, $name" # Access variable with $

Types of Variables

- **User-defined:** Created by the user (e.g., name="John")
- **Environment variables:** System-wide (e.g., HOME, PATH)

5.1.2.2. read and expr

read – Accept User Input
echo "Enter your age:"
read age
echo "You are $age years old"

expr – Arithmetic Operations
a=5
b=3
sum=$(expr $a + $b)
echo "Sum is: $sum"

Common Operators:
+, -, *, /, %, =, !=, -lt, -gt, -eq, -ne, -ge, -le
Note: Use * to escape * in multiplication:

mul=$(expr $a * $b)

5.1.2.3. Decision Making in Shell Scripts

A. if-else Statement

#!/bin/bash
echo "Enter a number:"
read num

if [$num -gt 10]; then
  echo "Greater than 10"
elif [$num -eq 10]; then
  echo "Exactly 10"
else
  echo "Less than 10"

fi

Important Notes:

- **[ ]** is a synonym for the test command.
- Always leave **spaces** inside [ ].

B. case Statement

Used for multiple conditions:

```
#!/bin/bash
echo "Enter a letter:"
read letter

case $letter in
  a) echo "You chose A";;
  b) echo "You chose B";;
  c) echo "You chose C";;
  *) echo "Unknown choice";;
esac
```

5.1.2.4. Loops in Shell Scripts

A. for Loop

Example 1: Iterating over values

```
for i in 1 2 3 4 5
do
  echo "Number: $i"
done
```

Example 2: With range

```
for i in {1..5}
do
  echo "Count: $i"
done
```

B. while Loop

```
count=1
while [ $count -le 5 ]
do
  echo "Count is $count"
  count=$(expr $count + 1)
done
```

C. until Loop (Optional)

Runs until condition is **true**:

```
count=1
until [ $count -gt 5 ]
do
  echo "Count: $count"
  count=$(expr $count + 1)
```

done

Mini Project Example

```
#!/bin/bash
# Check if a number is even or odd

echo "Enter a number:"
read num

rem=$(expr $num % 2)

if [ $rem -eq 0 ]; then
  echo "$num is even"
else
  echo "$num is odd"
fi
```

Summary Table

Concept	Syntax/Usage
Variable	name="John" / echo $name
User input	read varname
Arithmetic	expr $a + $b
if-else	if [condition]; then ... fi
case	case $var in ...)
for loop	for var in list; do ... done
while loop	while [condition]; do ... done