📅 **Date:** 24-02-2025

- # Session-1: Introduction to Operating System

- ## What is an Operating System (OS)?

  An **Operating System** is **system software** that **manages hardware resources** and provides services for application software and users. It acts as an **interface between users and the hardware**.

- ## Main Functions of an OS:

  ✅ **Hardware Manager:** Manages all hardware resources (CPU, RAM, I/O devices).
  ✅ **Process Manager:** Supervises tasks/processes/jobs being executed by the CPU.
  ✅ **Memory Manager:** Allocates and deallocates memory dynamically.
  ✅ **Interface Between User & Hardware:** Provides a platform for application execution.

  📌 **Illustration (From Book Reference)**

Image

{:height 33, :width 62}

- ## How is OS Different from Other Application Software?

Feature	Operating System	Application Software
Installation	Installed on **hard drive**	Installed **under OS layer**
Execution	Runs **directly on hardware**	Runs **on OS**
Purpose	Manages system resources	Performs specific user tasks
Examples	Windows, Linux, macOS	MS Word, VLC, Photoshop

📌 **Key Point: An application software depends on OS, but OS does not depend on application software.**

- ## Booting Process

  Booting refers to **loading the OS from the hard drive into main memory (RAM).**

  📌 **Types of Booting:**

  1. **Cold Booting** → Starting the system from a **power-off** state (**Initial OS load**).
  2. **Hot Booting** → Restarting the system while it's already running (**OS reloads into RAM**).

- ## Why is an OS Hardware Dependent?

📌 **An OS is hardware-dependent because:**
✅ OS interacts with hardware via **device drivers** specific to each component.
✅ Different hardware architectures (**x86, ARM, RISC-V**) require **compatible OS versions**.
✅ Embedded systems require **customized OS versions** tailored for performance constraints.

- ## Different Components of OS

📌 **Major OS Components:**

- **Kernel** → Core system controlling hardware & processes.
- **Process Management** → Handles CPU scheduling & multitasking.
- **Memory Management** → Allocates RAM efficiently.
- **File System** → Manages file storage & retrieval.
- **Device Management** → Handles I/O devices via drivers.
- **Security Management** → Protects system data & access control.

📌 **Illustration:**

Image

- ## Basic Computer Organization Required for OS

📌 **System Components:**
✅ **CPU** → Executes instructions.
✅ **RAM** → Stores active processes.
✅ **Storage (HDD/SSD)** → Stores OS & programs.
✅ **I/O Devices** → Keyboard, Mouse, Monitor, Printers.

📌 **Illustration:**

Image

- ## Examples of Well-Known Operating Systems

📌 **Types of OS & Examples:**

Category	Examples	Description
Mobile OS	Android, iOS	Designed for **smartphones & tablets**
Embedded OS	FreeRTOS, VxWorks	Used in **IoT & real-time devices**
Real-Time OS (RTOS)	HRT, SRT	Used for **mission-critical systems**

Category	Examples	Description
Desktop OS	Windows, macOS, Chrome OS	**General-purpose computing**
Server OS	Ubuntu Server, CentOS, Windows Server	Manages **network services & databases**

📌 **To-Do: "How are these OS types different from each other?"**

- ## Functions of an OS

📌 **Core OS Functionalities:**

✅ **Process Management**

- **Handles CPU scheduling** (e.g., FCFS, SJF, Round Robin).
- Ensures smooth multitasking.
- **Illustration:**

Image

✅ **Memory Management**

- Allocates memory to processes dynamically.
- Uses **paging & segmentation**.

✅ **Device Management**

- Manages hardware devices via **device drivers**.

✅ **Disk Management**

- Organizes data using **file systems (NTFS, FAT32, ext4)**.

✅ **Security Management**

- Includes **firewalls, antivirus, authentication**.

- ## Topics for Tomorrow:

📌 **Upcoming Topics:**

- **User & Kernel Space**
- **Interrupts & System Calls**
- **Memory Hierarchy in Computers**
- **Types of OS (Batch, Time-Sharing, Real-Time, etc.)**

- # Session-2: Introduction to Linux

- ## What is Linux?

📌 **Linux is an open-source OS**, meaning its source code is available for modification.

- Developed by **Linus Torvalds in 1991**.

- Backed by an **open-source community** for continuous updates.

- ## Features of Linux

✅ **No Cost / Low Cost** → Free to use.
✅ **Multi-Tasking** → Supports **multiple processes** at once.
✅ **Security** → Built-in **permissions & encryption**.
✅ **Customizable** → Users can modify **kernel & utilities**.
✅ **Multi-User** → Supports **multiple users on the same system**.
✅ **Better File System** → Supports **ext4, XFS, Btrfs**.
✅ **CLI & GUI Support** → Terminal-based & graphical interfaces.

- ## Linux File System Hierarchy

📌 **Directory Structure in Linux:**

Directory	Description
`/`	**Root directory** (Base of the filesystem)
`/bin`	Contains **user binaries** (e.g., `ls` , `cp` , `mv`)
`/sbin`	Stores **system binaries** (e.g., `fdisk` , `fsck`)
`/etc`	Contains **configuration files** (e.g., `passwd` , `hosts`)
`/dev`	Stores **device files** (e.g., `/dev/sda` for disks)
`/proc`	Holds **process info** (e.g., `/proc/cpuinfo`)
`/var`	Stores **log & cache files**
`/tmp`	Temporary files (Deleted on reboot)
`/usr`	User-related files & programs
`/home`	**User home directories** (`/home/user1`)
`/log`	Stores **system logs** (May vary across distributions)

- ## Basic Linux Commands

📌 **File & Directory Commands:**

- `ls` → List files & directories
- `cd <dir>` → Change directory
- `mkdir <dir>` → Create a directory
- `rm <file>` → Remove a file
- `rmdir <dir>` → Remove a directory

  📌 **Operators in Linux:**

  1. **Redirection (`>`, `>>`)**
  - `echo "Hello" > file.txt` → Write to file
  - `echo "World" >> file.txt` → Append to file
  1. **Pipe (`|`)**
  - `cat file.txt | grep "error"` → Filters output

Here are your **OS Notes – Day 2 (Session 1)** without revision content:

- # OS Notes – Day 2

📅 **Date:** 26-02-2025

- Session 1

- ## OS Introduction and Basic Functions

- ### User and Kernel Space & Mode

📌 **Definition:**

- **User Space:** Runs user applications with **restricted access** to hardware.
- **Kernel Space:** Executes **OS services and device drivers** with **full system access**.

📌 **Modes:**

- **User Mode:**
  - Runs normal applications (**text editors, browsers, media players**).
  - Cannot directly access hardware resources.
- **Kernel Mode:**
  - Runs **OS core functions, device drivers, and memory management**.
  - Has **full privileges** over CPU, memory, and hardware.

📌 **Illustration:**

Image

- ## Interrupts and System Calls

📌 **Interrupts:**
Interrupts **pause CPU execution** to handle critical events (e.g., keyboard input, disk I/O).

📌 **System Calls:**
System calls act as a **bridge between user applications and the OS kernel**.

📌 **Types of System Calls:**

Category	System Calls	Description
File Management	`open()`, `close()`, `read()`, `write()`, `delete()`	Operations on files
Process Control	`fork()`, `wait()`, `exec()`, `exit()`	Process creation and execution
Device Management	`ioctl()`, `read()`, `write()`	Communicate with hardware devices
Information Retrieval	`getpid()`, `sysinfo()`	Retrieve system data
IPC (Inter-Process Communication)	`wait()`, `notify()`	Process communication

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("Process ID: %d\n", getpid());  // Get process ID using system call
  return 0;
}
```

- # Types of Operating Systems

📌 **Major OS Types:**

Type	Description	Example
Batch OS	Executes jobs in batches, no user interaction	IBM OS/360
Multiprogramming OS	Runs multiple processes simultaneously	UNIX
Multitasking OS	Allows multiple applications to run at the same time	Windows, macOS

Type	Description	Example
Multiprocessing OS	Uses multiple CPUs for parallel execution	Linux, Unix
Clustered OS	Manages multiple computers as one system	Google Cloud OS
Distributed OS	Spreads processing tasks across networked computers	Amoeba OS
Embedded OS	Runs on **specialized devices** (low resource usage)	FreeRTOS, QNX

📌 **Illustration:**

Image

- ## Process Management

📌 **Definition:**
A **process** is a program **loaded into RAM for execution**.

📌 **Process Types:**

- **Preemptive Process:** Can be **interrupted** and resumed later.
- **Non-Preemptive Process:** Runs **without interruption** until completion.

📌 **Process Control Block (PCB):**
Each process has a **PCB** storing:
✅ **Process ID**
✅ **State (Ready, Running, Blocked, etc.)**
✅ **Program Counter**
✅ **CPU Registers**

📌 **Illustration:**

Image

- ### Process Life Cycle

📌 **Five States of a Process:**
1️⃣ **New** → Process is created.
2️⃣ **Ready** → Waiting for CPU allocation.
3️⃣ **Running** → Currently executing instructions.
4️⃣ **Blocked** → Waiting for I/O completion.
5️⃣ **Terminated** → Process execution finishes.

📌 **Illustration:**

Image

- **Schedulers & Scheduling Algorithms**

  📌 **Schedulers:**

- **Short-Term Scheduler** → Selects process for CPU execution.
- **Medium-Term Scheduler** → Swaps processes between RAM and disk.
- **Long-Term Scheduler** → Controls which processes **enter the system**.

  📌 **Scheduling Algorithms:**

  1️⃣ **FCFS (First Come First Serve):**

- Executes processes **in order of arrival**.
- **Non-preemptive** (Once started, it runs until completion).

  📌 **Illustration:**

Image

Schedulers & Scheduling Algorithms – Detailed Explanation

Schedulers and scheduling algorithms are **essential for process management** in an operating system. They determine **which process gets the CPU and for how long**, ensuring efficient execution of multiple processes.

1️⃣ What is a Scheduler?

A **scheduler** is a system component that manages **process execution** by selecting which process runs next. The OS contains **three types of schedulers**, each responsible for different stages of process execution.

📌 **Three Types of Schedulers:**

Scheduler	Function	Affects Which States?	Frequency of Execution
Long-Term Scheduler (Job Scheduler)	Decides which processes **enter the ready queue** from the new state	*New → Ready*	**Low (Seconds/Minutes)**
Short-Term Scheduler (CPU Scheduler)	Selects which process **runs on the CPU** next	*Ready → Running*	**Very High (Milliseconds)**

Scheduler	Function	Affects Which States?	Frequency of Execution
Medium-Term Scheduler (Swapper)	Swaps processes **in and out of RAM** to optimize memory usage	*Running/Blocked → Suspended*	**Medium (Seconds)**

- ### 📍 1. Long-Term Scheduler (Job Scheduler)

- **Function:** Controls **which processes are admitted** into the system for execution.
- **Key Role:** Regulates the **degree of multiprogramming** (number of processes in memory).
- **If Too Many Processes:** System may slow down due to **overloaded memory**.
- **If Too Few Processes:** CPU remains **idle**, leading to **poor resource utilization**.
- **Example:** A user starts **multiple programs** (browser, video player, text editor). The OS decides **which ones enter RAM** based on availability.

  📌 **Effect on Process Life Cycle:**

- **New** → **Ready** (Moves selected processes to memory).

- ### 📍 2. Short-Term Scheduler (CPU Scheduler)

- **Function:** Decides **which process gets CPU time** from the **ready queue**.
- **Key Role:** Ensures that processes **execute efficiently**.
- **Executes Frequently:** Runs **every few milliseconds** to switch processes rapidly.
- **Example:** If you're playing music while using a web browser, the CPU scheduler **switches tasks** between them rapidly, making it seem like both run simultaneously.

  📌 **Effect on Process Life Cycle:**

- **Ready** → **Running** (Selects process for CPU execution).
- **Running** → **Ready** (If preempted, moves back to ready queue).

- ### 📍 3. Medium-Term Scheduler (Swapper)

- **Function:** Swaps processes between **RAM and disk** to manage memory efficiently.
- **Key Role:** Helps **free up RAM** when memory is full.
- **Example:** If a **background process** (e.g., a minimized browser tab) is **inactive**, the OS moves it to the **swap area on disk**. It gets **restored** when needed.

  📌 **Effect on Process Life Cycle:**

- **Running/Blocked** → **Suspended** (Moves process to disk).
- **Suspended** → **Ready** (Brings it back when memory is available).

- ## 2️⃣ What is a Scheduling Algorithm?

  A **scheduling algorithm** determines **how the CPU is assigned to processes** in the ready queue.

  📌 **Objectives of CPU Scheduling:**
  ✅ **Maximize CPU Utilization** → Keep CPU **busy**.
  ✅ **Minimize Waiting Time** → Reduce time spent **waiting** in the ready queue.
  ✅ **Minimize Turnaround Time** → Shorten **total process execution time**.
  ✅ **Ensure Fairness** → Every process **gets CPU time**.

  📌 **Scheduling Algorithms are divided into:**

- **Non-Preemptive:** Once a process starts executing, it **cannot be interrupted** until it finishes.
- **Preemptive:** A process **can be interrupted** and moved back to the ready queue if a higher-priority process arrives.

- ## 3️⃣ CPU Scheduling Algorithms (With Examples & Diagrams)

- ### 🔹 1. First Come, First Serve (FCFS) – Non-Preemptive

- **Processes are scheduled based on arrival time (FIFO – First In, First Out).**
- **Disadvantage:** Causes **convoy effect** – a short job waits for a long job to finish.

  📌 **Example:**

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	0	8	8	8 - 0 = 8	0
P2	1	4	12	12 - 1 = 11	8 - 1 = 7
P3	2	9	21	21 - 2 = 19	12 - 2 = 10

  📌 **Gantt Chart:**

```
| P1 | P2 | P3 |
0    8   12   21
```

📌 **Avg Waiting Time (AWT) = (0+7+10)/3 = 5.67 ms**

- ### 🔹 2. Shortest Job First (SJF) – Preemptive & Non-Preemptive

- **Selects the process with the shortest burst time.**
- **Preemptive SJF (Shortest Remaining Time First - SRTF)** allows process **preemption**.

  📌 **Example (Non-Preemptive SJF):**

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	8	8	0
P2	1	4	12	11	7
P3	2	9	21	19	10

  📌 **Gantt Chart:**

```
| P2 | P1 | P3 |
1    5    13   22
```

📌 **Avg Waiting Time (AWT) = 5.67 ms**

- ### ◇ 3. Priority Scheduling – Preemptive & Non-Preemptive

- **Assigns a priority to each process; the CPU selects the highest priority process.**
- **Preemptive Priority Scheduling:** If a higher-priority process arrives, it **interrupts the current process**.

  📌 **Example (Lower number = Higher priority):**

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
P1	0	8	2	8	8	0
P2	1	4	1	5	4	0
P3	2	9	3	21	19	10

  📌 **Gantt Chart:**

```
| P2 | P1 | P3 |
1    5    13   22
```

📌 **Avg Waiting Time (AWT) = 3.33 ms**

- ### ◇ 4. Round Robin (RR) – Preemptive

- **Each process gets a fixed time slice (Time Quantum).**

- **If a process doesn't finish within the time slice, it goes back to the queue.**

  📌 **Example (Time Quantum = 4 ms):**

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	16	16	8
P2	1	4	5	4	0
P3	2	9	21	19	10

  📌 **Gantt Chart:**

```
| P1 | P2 | P3 | P1 | P3 |
0    4    8    12   16   21
```

📌 **Avg Waiting Time (AWT) = 6 ms**

Here's a **detailed explanation** of your **Linux Notes – Day 2**, expanding on key concepts, commands, and shell scripting.

- Session 2

- # Linux and Useful Commands

- ## 📌 What is Linux?

  📌 **Linux is an open-source operating system**, meaning its **source code is freely available** for modification and distribution.

  - **Founder: Linus Torvalds (1991)**
  - **Community-driven:** Updated and maintained by an **open-source community**.

- ## 📌 Features of Linux

  ✔ **No Cost / Low Cost** → Available for free, reducing software costs.
  ✔ **Multi-Tasking** → Runs multiple applications **simultaneously**.
  ✔ **Security** → **User permissions, encryption, and firewalls** for secure computing.
  ✔ **Multi-User Support** → Supports multiple users at the same time.
  ✔ **Stable & Scalable** → Used in **servers, desktops, and embedded devices**.
  ✔ **Networking** → **Efficient networking capabilities**, making it ideal for **servers**.
  ✔ **CLI & GUI** → Supports **command-line (CLI)** and **graphical user interface (GUI)**.
  ✔ **Better File System** → Supports **ext4, XFS, Btrfs, ZFS** (better than FAT32/NTFS).

  📌 **Illustration – Linux System Architecture:**

Image

- ## 📌 Linux File System Basics

📌 **Linux follows a hierarchical directory structure starting from `/` (root).**

Directory	Description
`/`	**Root directory** (Base of the filesystem)
`/bin`	User **binary files** (e.g., `ls`, `cp`, `mv`)
`/sbin`	System **binary files** (for admin tasks)
`/etc`	Configuration files for **system settings**
`/dev`	Stores device files (e.g., `/dev/sda` for disks)
`/proc`	Virtual directory for **process information**
`/var`	Stores **logs, caches, and variable files**
`/tmp`	Temporary files (deleted on reboot)
`/usr`	User-related programs and libraries
`/home`	**User home directories** (`/home/user1`)
`/boot`	Bootloader files for starting Linux
`/opt`	Optional software packages
`/lib`	System libraries required for OS functionality

- ## 📌 Important Linux Commands

Linux is primarily controlled via the **command line (CLI)**.

📌 **File & Directory Management Commands:**
✅ `pwd` → Print the **current working directory**.
✅ `ls` → List **files and directories**.
✅ `nano <file>` → Open the **nano text editor**.
✅ `touch <file>` → Create a **new file**.
✅ `mkdir <dir>` → Create a **new directory**.
✅ `rm <file>` → Remove a **file**.
✅ `rmdir <dir>` → Remove an **empty directory**.
✅ `cd <dir>` → Change the **current directory**.
✅ `chmod 755 file` → Change **file permissions**.
✅ `chown user:group file` → Change **file ownership**.

📌 **Illustration:**

Image

- ## 📌 What is a Shell in Linux?

📌 **The Shell is an interface between the user and the Linux kernel.**

✔ **It takes user commands, interprets them, and sends them to the OS for execution.**
✔ **Users can interact with the shell via terminal commands or shell scripts.**

📌 **Types of Shells in Linux:**

Shell Type	Path	Description
Bourne Shell (sh)	`/bin/sh`	The **original Unix shell**
Bash (Bourne Again Shell)	`/bin/bash`	Most commonly used **default Linux shell**
Restricted Bash (rbash)	`/bin/rbash`	A limited version of Bash
Dash	`/bin/dash`	A **lightweight shell**, faster than Bash
Tmux	`/usr/bin/tmux`	A terminal multiplexer
Screen	`/usr/bin/screen`	Allows **multiple terminal sessions**

- ## 📌 Shell Variables

📌 **Shell variables store values for use in scripts.**

✔ Can store **strings, numbers, and command outputs**.
✔ No need to specify **variable types**.

📌 **Example:**

```
X=100       # Assigns 100 to variable X
Y="Linux"   # Assigns "Linux" to variable Y

echo $X     # Prints 100
echo $Y     # Prints Linux
```

📌 **Reading Input in Shell Scripts:**

```
echo "Enter your name:"
read name
echo "Hello, $name!"
```

📌 **Printing Output:**

```
echo "This is a Linux script!"
```

- ## 📌 Operators in Linux

- ### 1️⃣ Redirection Operators (`>`, `>>`, `<`)

📌 **Used to redirect input/output.**

✔ `>` → Overwrites a file
✔ `>>` → Appends to a file
✔ `<` → Takes input from a file

📌 **Example:**

```
echo "Hello World" > file.txt  # Creates file.txt and writes "Hello World"
echo "Another Line" >> file.txt  # Appends "Another Line" to file.txt
cat < file.txt  # Reads contents of file.txt
```

- ### 2️⃣ Pipe Operator (`|`)

📌 **Used to pass output of one command as input to another.**

📌 **Example:**

```
cat file.txt | grep "error"  # Finds "error" in file.txt
ls -l | less  # Displays long list format in a paginated view
```

- ## 📌 File Permissions & Access Control

📌 **Linux assigns three types of permissions to each file:**

- **Read (`r`)** → View file contents.
- **Write (`w`)** → Modify file contents.
- **Execute (`x`)** → Run the file as a program.

📌 **File Permission Representation:**

```
-rwxr-xr--  1 user group  4096 Feb 25 10:00 file.txt
```

📌 **Breakdown of Permissions:**

Symbol	Owner	Group	Others

Symbol	Owner	Group	Others
rwx	Read, Write, Execute	Read, Execute	Read

📌 Changing File Permissions:

```
chmod 755 file.txt  # Owner: rwx, Group: r-x, Others: r-x
chmod 644 file.txt  # Owner: rw-, Group: r--, Others: r--
```

📌 Changing File Ownership:

```
chown user:group file.txt  # Assigns ownership to user and group
```

- ## 📌 Shell Programming Basics

  #### 📌 Shell scripts automate repetitive tasks in Linux.

- ### 1️⃣ Conditional Statements (`if-else`)

```
if [ $X -gt 10 ]
then
echo "X is greater than 10"
else
echo "X is 10 or less"
fi
```

- ### 2️⃣ Loops in Shell Scripts

- #### For Loop

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

- #### While Loop

```
X=1
while [ $X -le 5 ]
do
echo "Loop iteration: $X"
```

```
    X=$((X + 1))
done
```

- ## 📌 To Be Discussed Tomorrow Evening (27-02-2025)

  📌 **Advanced Linux Topics:**
  ☑ **Advanced Operators (Redirection, Pipe, etc.)**
  ☑ **File Permissions & Access Control Lists**
  ☑ **More Shell Programming – Wildcards, Regular Expressions**
  ☑ **Command Line Arguments in Shell Scripts**
  ☑ **Decision Loops (if-else, case, while, for, until)**
  ☑ **Arithmetic Expressions & Shell Scripting Examples**

- # OS Notes – Day 3

  📅 **Date:** 27-02-2025

  ---

  session 1

- # Memory Hierarchy – Detailed Explanation

- ## 📌 What is Memory Hierarchy?

  Memory hierarchy is the **structured arrangement of memory components** in a computer system, organized to **optimize speed, cost, and capacity**.

  📌 **Key Idea:**

- **Faster memories are expensive & small**, while **slower memories are cheaper & large**.
- **Frequently used data is stored in faster memory** (Registers, Cache).
- **Less frequently used data is stored in slower memory** (RAM, Disk).

  📌 **Illustration of Memory Hierarchy:**

```
        ⬆ Faster Access, Lower Capacity, Higher Cost
        ___________________________________
        |  Registers   |  (Inside CPU, Fastest, Smallest) |
        |  Cache (L1, L2, L3) |  (Fast, Stores Recent Data) |
        |  Main Memory (RAM)  |  (Larger but Slower) |
        |  Secondary Storage  |  (Hard Drive, SSD, Persistent) |
        |  Tertiary Storage   |  (Cloud, Magnetic Tape) |
```

```
⬇ Slower Access, Higher Capacity, Lower Cost
```

- 📌 **Levels of Memory Hierarchy**
- 1️⃣ **Registers** (Fastest, Inside CPU)

✔ **Location:** Inside the CPU.
✔ **Characteristics:**

- Fastest memory, directly accessible by the CPU.
- Limited number (usually 32 or 64 registers per CPU).
- Stores temporary data for arithmetic/logical operations.
  ✔ **Use:** Holds the **operands and results** of CPU instructions.

📌 **Example:**

- When executing `A + B`, values of `A` and `B` are stored in registers for quick addition.

- 2️⃣ **Cache Memory** (High-Speed Buffer)

✔ **Purpose: Acts as a bridge between CPU and RAM** to reduce access time.
✔ **Types of Cache:**

- **L1 (Level 1) Cache** → Fastest but **smallest**, located inside the CPU core.
- **L2 (Level 2) Cache** → **Larger than L1**, but slightly slower.
- **L3 (Level 3) Cache** → **Shared among multiple CPU cores**, improves multitasking.
  ✔ **Characteristics:**
- **Stores frequently accessed data** to reduce memory access time.
- Works based on **locality principles**:
  - **Temporal Locality:** Recently used data is likely to be used again soon.
  - **Spatial Locality:** Data near recently used data is likely to be accessed soon.

📌 **Example:**

- If a program frequently accesses an array, cache stores **nearby elements** to speed up access.

- 3️⃣ **Main Memory (RAM - Random Access Memory)**

✔ **Purpose:** Stores **active processes and data** for quick CPU access.
✔ **Characteristics:**

- Larger capacity than Cache, but **slower**.
- **Volatile** (Data lost when power is off).
  ✔ **Use:** Holds **running programs, operating system**, and **frequently accessed data**.

  📌 **Example:**

- When opening an application (e.g., MS Word), it is **loaded from disk into RAM** for faster access.

- ### 4️⃣ Secondary Storage (Hard Drive & SSDs)

  ✔ **Purpose: Permanent storage** for files, programs, and the OS.
  ✔ **Examples: Hard Disk Drives (HDDs), Solid-State Drives (SSDs)**.
  ✔ **Characteristics:**

- **Non-volatile** (Retains data after shutdown).
- Much **larger capacity than RAM**.
- **Slower than RAM** but **cheaper per GB**.

  📌 **Example:**

- When you save a file, it is **written to the hard disk** instead of RAM for long-term storage.

  📌 **Comparison: HDD vs. SSD**

Feature	HDD (Hard Disk Drive)	SSD (Solid-State Drive)
Speed	Slower	Much Faster
Durability	Less Durable (Moving Parts)	More Durable (No Moving Parts)
Cost	Cheaper per GB	More Expensive per GB

- ### 5️⃣ Tertiary/External Storage

  ✔ **Purpose: Backup, Archival, and Rarely Accessed Data**.
  ✔ **Examples: Magnetic Tape, Optical Discs (CD/DVD), Cloud Storage**.
  ✔ **Characteristics:**

- **Very high capacity**, but **slowest access speed**.
- Used for **long-term storage** or **disaster recovery**.

  📌 **Example:**

- **Magnetic tapes** store archived data in large data centers.
- **Cloud storage (Google Drive, Dropbox)** allows **off-site backups**.

- ## 📌 Key Takeaways

☑ **Speed vs. Cost Trade-off:**

- **Faster memory = More expensive, Smaller size**.
- **Slower memory = Cheaper, Larger capacity**.

☑ **Why Use a Hierarchy?**

- **Registers are limited**, so we use Cache.
- **Cache is expensive**, so we use RAM.
- **RAM is volatile**, so we use HDD/SSD.
- **HDD/SSD is slow**, so we use Cache again.

☑ **Locality Principles:**

- **Temporal Locality:** If data is used once, it is likely to be used again soon.
- **Spatial Locality:** If data at a memory location is accessed, nearby memory locations are likely to be accessed next.

- ## 📌 Real-World Analogy: Memory Hierarchy as a Kitchen Setup

Imagine a **chef cooking in a kitchen**:

Memory Level	Kitchen Equivalent	Speed
Registers	Ingredients in chef's hands	🚀 Fastest
Cache Memory	Ingredients on the kitchen counter	🔥 Very Fast
RAM (Main Memory)	Ingredients in the fridge	⚡ Fast
Hard Drive (HDD/SSD)	Ingredients in a grocery store	⌛ Slow
Tertiary Storage (Backup)	Ingredients stored in a warehouse	🐢 Slowest

📌 **Key Idea:** The chef uses the fastest and closest memory (Registers & Cache) most often, while accessing the fridge (RAM) or store (HDD) only when necessary.

- ## 📌 Process Scheduling Algorithms

- # Process Scheduling Algorithms – Detailed Explanation

Process scheduling algorithms determine **which process the CPU executes next** from the ready queue. The goal is to optimize **CPU utilization, minimize waiting time, and improve system responsiveness**.

- ## 📌 1. Shortest Job First (SJF) Scheduling

📌 **Definition:**

- The CPU selects the **process with the smallest execution time (CPU burst)** first.
- **Goal:** Minimizes **average waiting time**, making it the **optimal algorithm** in ideal conditions.

📌 **Key Characteristics:**
☑ **Best average waiting time** if all processes arrive at the same time.
☑ **Works well for batch systems** where CPU burst times are known.
❌ **Starvation Issue:** Longer processes **may wait indefinitely** if shorter jobs keep arriving.

📌 **Two Types of SJF:**

- ### ◇ Non-Preemptive SJF

- **Once a process starts execution, it runs until completion** (No interruptions).
- **Use Case:** Best for batch systems with **predictable CPU bursts**.

📌 **Example – Non-Preemptive SJF:**

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT = CT - AT)	Waiting Time (WT = TAT - BT)
P1	0	8	8	8 - 0 = 8	0
P2	1	4	12	12 - 1 = 11	7
P3	2	9	21	21 - 2 = 19	10

📌 **Gantt Chart for Non-Preemptive SJF:**

```
| P2 | P1 | P3 |
0    4    12   21
```

📌 **Avg Waiting Time (AWT) = (0+7+10)/3 = 5.67 ms**
📌 **Illustration:**

Image

- ### ◇ Preemptive SJF (Shortest Remaining Time First - SRTF)

- **A new process can preempt the current running process if it has a shorter burst time.**
- **Use Case:** Best for **time-sharing** or interactive systems.

📌 **Example – Preemptive SJF (SRTF):**

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	13	13	5
P2	1	4	5	4	0
P3	2	9	21	19	10

📌 **Gantt Chart for Preemptive SJF (SRTF):**

```
| P2 | P1 | P3 |
1    5    13   21
```

📌 **Avg Waiting Time (AWT) = 5.67 ms**
📌 **Illustration:**

Image

- ## 📌 2. Priority Scheduling

  📌 **Definition:**

- **Each process is assigned a priority**, and the CPU selects the **highest-priority process** first.
- **Priority can be static (fixed) or dynamic (changes over time).**

  📌 **Key Characteristics:**
  ☑ **Ensures important tasks run first** (e.g., real-time OS).
  ☑ **Used in scheduling system processes.**
  ✖ **Starvation Issue:** Low-priority processes **may never execute**.
  ☑ **Solution: Aging** (gradually increasing priority of waiting processes).

  📌 **Two Types of Priority Scheduling:**

- ### 🔹 Preemptive Priority Scheduling

- **A higher-priority process can interrupt a lower-priority running process.**
- **Use Case: Real-time systems** (e.g., medical monitoring, airline systems).

- ### 🔹 Non-Preemptive Priority Scheduling

- **A running process is not interrupted, even if a higher-priority process arrives.**
- **Use Case:** Suitable for batch systems where tasks must **finish once started**.

  📌 **Example – Priority Scheduling:**

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
P1	0	8	2	8	8	0
P2	1	4	1	5	4	0
P3	2	9	3	21	19	10

📌 **Gantt Chart for Priority Scheduling:**

```
| P2 | P1 | P3 |
1    5    13   22
```

📌 **Avg Waiting Time (AWT) = 3.33 ms**
📌 **Illustration:**

Image

- ## 📌 3. Round Robin (RR) Scheduling

📌 **Definition:**

- **Each process gets a fixed time slice (Time Quantum).**
- **If a process doesn't finish within its time slice, it is moved to the back of the queue.**
- **Used in multi-user and time-sharing systems.**

📌 **Key Characteristics:**
☑ **Fair Scheduling:** Every process **gets CPU time**.
☑ **Good for interactive systems.**
☑ **Ensures no process is starved.**
❌ **Too small a quantum = Too many context switches (overhead).**
❌ **Too large a quantum = Behaves like FCFS.**

📌 **Example – Round Robin (Time Quantum = 4 ms):**

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	16	16	8
P2	1	4	5	4	0
P3	2	9	21	19	10

📌 **Gantt Chart for Round Robin (Time Quantum = 4):**

```
| P1 | P2 | P3 | P1 | P3 |
0    4    8    12   16   21
```

📌 **Avg Waiting Time (AWT) = 6 ms**
📌 **Illustration:**

Image

- 🔹 **Impact of Reduced Quantum in Round Robin**

  📌 **If time quantum is too small, processes are switched too frequently, causing high context-switching overhead.**

  📌 **Example with Reduced Quantum:**

Image

- ## 📌 Summary of Scheduling Algorithms

Algorithm	Preemptive?	Optimal Waiting Time?	Starvation Risk?	Best For
FCFS	❌ No	❌ No	☑️ Yes (Long jobs delay short jobs)	Simple batch processing
SJF (Non-Preemptive)	❌ No	☑️ Yes	☑️ Yes (Starvation of long jobs)	Ideal if burst time is known
SJF (Preemptive - SRTF)	☑️ Yes	☑️ Yes	☑️ Yes	Best for multitasking
Priority (Non-Preemptive)	❌ No	❌ No	☑️ Yes	Used in critical systems
Priority (Preemptive)	☑️ Yes	❌ No	☑️ Yes	Real-time OS (e.g., medical systems)
Round Robin (RR)	☑️ Yes	❌ No	❌ No	Time-sharing systems

- ## 📌 Key Takeaways

✔️ **SJF minimizes average waiting time** but causes **starvation**.
✔️ **Priority Scheduling ensures important tasks run first** but can cause **starvation**.
✔️ **Round Robin guarantees fairness** but depends on **quantum size**.
✔️ **Choosing the best algorithm depends on system requirements.**

- ## 📌 Memory Hierarchy

  📌 **Definition:** A structured arrangement of different storage types in a computer system, balancing **speed, cost, and capacity**.

- ### Levels of Memory Hierarchy

Memory Level	Characteristics	Speed	Size
Registers	Inside the CPU, extremely fast, very limited in size	🚀 Fastest	🔹 Smallest
Cache Memory (L1, L2, L3)	Holds frequently accessed data to speed up processing	🚀 Very Fast	🔹 Small
Main Memory (RAM)	Stores actively used data & programs	⚡ Fast	🔹 Moderate
Secondary Storage (HDD/SSD)	Long-term storage (disk-based)	⏳ Slower	🔹 Large
Tertiary Storage (Tape, Optical Disks)	Used for backup & archives	🐢 Slowest	🔹 Very Large

📌 **Key Takeaways:**
✔ **Speed vs. Cost Trade-off** → Faster memory is **more expensive** per bit.
✔ **Locality Principle:**

- **Temporal Locality** → Recently accessed data is likely to be used again.
- **Spatial Locality** → Nearby data is likely to be accessed soon.

- ## 📌 Linux Useful Commands & Shell Scripting

- ### 🔹 Shell Scripting – Decision Loops

  📌 1️⃣ **If-Else Statement**

```
if [ condition ]
then
  statement
else
  statement
fi
```

📌 **Example:**

```
```

```
echo "Enter a number:"
read num
if [ $num -eq 5 ]
then
  echo "Number is 5"
else
  echo "Number is not 5"
fi
```

📌 2️⃣ Nested If-Else

```
if [ condition ]
then
  if [ condition ]
  then
      statement
  else
      statement
  fi
else
  if [ condition ]
  then
      statement
  fi
fi
```

📌 Example:

```
echo "Enter three numbers:"
read num1 num2 num3
if [ $num1 -gt $num2 ]
then
  if [ $num1 -gt $num3 ]
  then
      echo "$num1 is the largest"
  else
      echo "$num3 is the largest"
  fi
else
  if [ $num2 -gt $num3 ]
  then
      echo "$num2 is the largest"
  else
      echo "$num3 is the largest"
  fi
fi
```

- ◇ **Loops in Shell Scripting**

  📌 1️⃣ **For Loop** (Repeats code **n times**)

```
for variable in value1 value2 value3
do
  echo $variable
done
```

📌 **Example:**

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

📌 **Example with Sum Calculation:**

```
sum=0
for num in 1 2 3 4 5
do
  sum=$((sum + num))
done
echo "Sum is: $sum"
```

📌 OS Notes – Day 4

📅 **Date:** 28-02-2025

1️⃣ Memory Management

Memory management ensures **efficient allocation, tracking, and usage of RAM**. It includes **partitioning, paging, segmentation, and virtual memory**.

📌 1.1 Static Partitioning

📌 **Definition:**

- Early systems used **fixed-size memory partitions**, where each process occupied **one contiguous block**.

📌 **Drawbacks:**
❌ **External fragmentation** → Free memory is scattered, making it hard to allocate new processes.
❌ **Wastage of memory** if a process does not fully utilize a partition.

📌 **Illustration:**
Image

📌 1.2 Memory Allocation Strategies

Different methods are used to allocate memory blocks to processes.

🔹 First Fit

- **Allocates the first free block** that fits the process size.
- ✅ **Fast allocation**, ❌ **Leads to small unusable memory gaps**.

📌 **Example:**

```
Free Blocks: 50, 200, 100
Process Request: 75
Allocation: 200-block (if 50 is too small)
```

📌 **Illustration:**
Image

🔹 Best Fit

- **Chooses the smallest block** that fits the process.
- ✅ **Minimizes wasted space**, ❌ **Can create many small fragments**.

📌 **Example:**

```
Free Blocks: 50, 200, 100
Process Request: 75
Allocation: 100-block (smallest fit)
```

🔹 Worst Fit

- **Allocates the largest block**, assuming that **splitting a large block leaves useful space**.
- ✅ **Larger leftover blocks**, ❌ **Can quickly reduce the size of free blocks**.

📌 **Example:**

```
Free Blocks: 50, 200, 100
Process Request: 75
Allocation: 200-block (largest)
```

📌 1.3 Internal & External Fragmentation

📌 **Internal Fragmentation:**

- **Unused memory inside allocated blocks** due to **fixed-sized partitions**.
- **Example:** Process needs **70 bytes**, but **100 bytes are allocated**, wasting **30 bytes**.

📌 **External Fragmentation:**

- **Free memory is scattered** across RAM, preventing allocation of large processes.
- **Example:** 90 bytes are free in total, but scattered as **30, 20, 40 bytes**, preventing a **50-byte process** from allocation.

📌 1.4 Compaction (Defragmentation)

- **Rearranges memory to merge free spaces into one large block.**
- ✅ **Solves external fragmentation**, ❌ **Time-consuming as it requires process relocation**.

📌 **Steps:**

1. **Shift processes to form contiguous allocated memory.**
2. **Merge free spaces into one large block.**
3. **Update memory addresses (page tables, pointers, etc.).**

2️⃣ Paging

📌 **Definition:**

- **Divides process memory into fixed-size "pages".**
- **Divides RAM into fixed-size "frames".**
- **Pages are mapped to frames using a Page Table.**

📌 **Why Paging?**
✅ **Allows non-contiguous memory allocation** (avoiding external fragmentation).
✅ **Supports demand paging and virtual memory**.

📌 **Illustration:**
Image

📌 2.1 Page Table

- **Maps logical addresses (pages) to physical addresses (frames).**

- **Stored in RAM or MMU (Memory Management Unit).**

📌 **Example Page Table:**

Page Number	Frame Number
0	3
1	7
2	1
3	4

📌 2.2 Demand Paging & Page Faults

📌 **Demand Paging:**

- **Pages are loaded into RAM only when needed** (on demand).
- **Reduces memory usage** by **loading only required pages**.

📌 **Page Fault:**

- **Occurs when the requested page is not in RAM**.
- **OS loads the page from disk to RAM** and updates the page table.

📌 **Page Replacement Algorithms:**
1️⃣ **FIFO (First-In-First-Out):** Removes the **oldest page first**.
✅ **Simple**, ❌ **Causes Belady's Anomaly**.
Image

2️⃣ **LRU (Least Recently Used):** Removes the **least recently accessed page**.
✅ **Efficient**, ❌ **Complex tracking required**.

3️⃣ **MRU (Most Recently Used - LIFO):** Removes the **most recently used page**.
✅ **Works well for some workloads**, ❌ **Not optimal in general cases**.
Image

3️⃣ Virtual Memory

📌 **Definition:**

- **Uses disk space to extend RAM (illusion of larger memory).**
- **Process pages are stored in Virtual Memory (disk) and swapped into RAM when needed.**

📌 **Key Concepts:**
✔ **Swap-in:** Load a page from disk to RAM.
✔ **Swap-out:** Move a page from RAM back to disk.

📌 **Illustration:**
Image

📌 **Translation Lookaside Buffer (TLB)**

- **Caches page table entries for faster memory access.**
- **Avoids repeated RAM lookups, improving performance.**

4️⃣ Segmentation

📌 **Definition:**

- **Divides a process into logical segments** (Code, Data, Stack).
- **Each segment has a variable size.**

📌 **Benefits:**
☑ **Logical separation of memory** (e.g., code cannot overwrite stack).
☑ **Supports shared memory (multiple processes share segments).**

📌 **Segmentation Table Example:**

Segment Number	Base Address	Limit	Permissions
Code	0x1000	0x0FFF	Execute
Data	0x2000	0x0FFF	Read/Write
Stack	0x3000	0x0FFF	Read/Write

📌 **Illustration:**
Image

📌 Detailed Explanation of Paging and Segmentation

Paging and Segmentation are **two memory management techniques** used to handle **process execution efficiently** in an operating system. Below is a **detailed breakdown** of these concepts, including **diagrams, examples, and real-world comparisons**.

1️⃣ Paging

📌 What is Paging?

Paging is a memory management technique where the OS **divides processes into fixed-size pages** and loads them into **available memory frames**.

📌 **Key Features of Paging:**
☑ **Eliminates External Fragmentation** → Pages can be placed anywhere in RAM.

☑️ **Supports Virtual Memory** → Pages are loaded **on-demand**, reducing RAM usage.
☑️ **Fixed Page Size** → Typically **4KB, 8KB, or 16KB** per page.

📌 **Illustration of Paging:**
Image

- **Logical Memory (Process Address Space) is divided into fixed-size "Pages"**.
- **Physical Memory (RAM) is divided into fixed-size "Frames"**.
- **Page Table maps Pages to Frames** for address translation.

📌 Why is Paging Needed?

1️⃣ **Memory Allocation is more flexible** → No need for **contiguous** allocation.
2️⃣ **Processes can be larger than available RAM** → Pages are loaded **on demand**.
3️⃣ **Avoids External Fragmentation** → Unlike **fixed partitions**, pages can be placed anywhere.

📌 **Example:**

```
Process Size = 10 KB
Page Size = 4 KB
Number of Pages = 10 KB / 4 KB = 3 Pages
Each Page is placed in a Frame in RAM.
```

📌 2. Paging Table (Address Mapping)

📌 **Definition:**
A **Page Table** is a **data structure** maintained by the **Memory Management Unit (MMU)** to map **logical addresses (pages) to physical addresses (frames)**.

📌 **Example Page Table:**

Page Number	Frame Number
0	3
1	7
2	1
3	4

📌 **Logical to Physical Address Translation:**

```
Physical Address = (Frame Number × Frame Size) + Offset
```

- If **Page 2 is mapped to Frame 1**, and the **offset** is 200 bytes, then:

```
Physical Address = (1 × 4KB) + 200 = 4200 bytes
```

📌 3. Demand Paging

📌 **Definition:**

- **Pages are loaded into memory only when the CPU needs them.**
- **If a required page is not in RAM, a "Page Fault" occurs, and the OS fetches it from Disk.**

📌 **Steps in Demand Paging:** 1️⃣ CPU requests a page.
2️⃣ OS checks if it is in RAM.
3️⃣ If **not found**, a **Page Fault** occurs.
4️⃣ OS loads the page from Disk into RAM.
5️⃣ Page Table is updated.

📌 **Example:**

```
A program has 10 pages but RAM has space for only 4.
Initially, only 4 pages are loaded.
If another page is required, a Page Fault occurs.
OS loads the required page from Virtual Memory (Disk).
```

📌 4. Page Faults

📌 **Definition:**

- A **Page Fault occurs when a requested page is not found in RAM**.
- The OS **retrieves it from Virtual Memory (Disk)**.

📌 **Too Many Page Faults = Performance Decrease**

- **Thrashing** → When the system spends more time **swapping pages than executing processes**.

📌 5. Page Replacement Algorithms

📌 **Used when RAM is full, and a new page needs to be loaded.**

🔹 FIFO (First In, First Out)

✅ **Removes the oldest page first**.
❌ **Belady's Anomaly**: More frames **may increase** page faults.

📌 **Example:**

```
Frames: 3
Page Requests: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
FIFO Replacement: Oldest pages are removed first.
```

📌 **Illustration:**
Image

🔹 LRU (Least Recently Used)

✅ **Removes the page that has not been used for the longest time**.
❌ **More complex to track usage history.**

🔹 MRU (Most Recently Used - LIFO)

✅ **Removes the most recently used page first**.
❌ **Not optimal in general cases**.
📌 **Illustration:**
Image

📌 6. Virtual Memory

📌 **Definition:**

- **Uses disk space to extend RAM (illusion of larger memory).**
- **Process pages are stored in Virtual Memory (disk) and swapped into RAM when needed.**

📌 **Key Features:** ✔ **Swap-in:** Load a page from disk to RAM.
✔ **Swap-out:** Move a page from RAM back to disk.

📌 **Illustration:**
Image

📌 **Translation Lookaside Buffer (TLB)**

- **Caches page table entries for faster memory access.**
- **Avoids repeated RAM lookups, improving performance.**

📌 7. Dirty Bit (Modified Bit)

📌 **Definition:**

- **Indicates whether a page in RAM has been modified.**
- If **dirty bit = 1**, the page **must be written to disk before replacement**.

📌 **Example:**

- **Process modifies data in a page.**

- **Before replacing it, OS checks the Dirty Bit.**
- **If it's set to 1, the page is saved to disk.**

2️⃣ Segmentation

📌 What is Segmentation?

Segmentation divides a program into logical segments (Code, Data, Stack, etc.).

📌 **Key Features:**
✅ **Logical separation of memory (code, data, stack).**
✅ **Segments are of variable size.**
❌ **Leads to External Fragmentation** (requires compaction).

📌 **Illustration:**
Image

📌 1. Segmentation Table

Each segment has a **Segment Table Entry** (Base Address, Limit, Access Rights).

📌 **Example Table:**

Segment Number	Base Address	Limit	Permissions
Code	0x1000	0x0FFF	Execute
Data	0x2000	0x0FFF	Read/Write
Stack	0x3000	0x0FFF	Read/Write

📌 **Logical Addressing in Segmentation:**

```
Physical Address = Base Address + Offset
```

📌 **Example:**

```
Segment = Code
Base Address = 0x1000
Offset = 0x0200
Physical Address = 0x1000 + 0x0200 = 0x1200
```

📌 Summary

✔ **Paging divides memory into fixed-size pages; Segmentation divides memory logically.**
✔ **Paging uses a Page Table; Segmentation uses a Segment Table.**
✔ **Paging eliminates external fragmentation; Segmentation suffers from it.**
✔ **Virtual Memory uses Paging to load only needed pages into RAM.**
✔ **Page Replacement Algorithms (FIFO, LRU, MRU) manage memory efficiently.**

Would you like **example problems, detailed comparisons, or hands-on exercises** for these topics? 😊

5️⃣ Linux Shell Programming

📌 5.1 Loops in Shell Scripting

🔹 While Loop

```
a=0
while [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

🔹 Until Loop

```
a=0
until [ $a -gt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

📌 5.2 Wildcard Symbols & Meta Characters

✔ `*` → Matches all files.
✔ `?` → Matches **one** character.
✔ `[abc]` → Matches **a, b, or c**.
✔ `|` → **Pipe operator** (filters output).
✔ `>` → **Redirects output** to a file.
✔ `&&` → Runs **multiple commands if the first succeeds**.
