MAA OMWATI DEGREE COLLEGE

V.P.O. Hassanpur, Teh. Hodal Distt. Palwal

(HR.)

BCA-3rd (NEP) OPERATING SYSTEMS

COURSE CODE - 24BCA403DS01

UNIT - 1

Introduction to Operating System (OS):-

An **Operating System (OS)** is a system software that manages computer hardware, software resources, and provides services for computer programs. It acts as an intermediary between users and the computer hardware, enabling efficient and effective use of the system.

Objectives of an Operating System

The main objectives of an Operating System are:

1. **Resource Management**
   Efficiently manage hardware resources like CPU, memory, disk drives, and I/O devices.
2. **Convenience**
   Provide a user-friendly interface that makes interaction with the computer easier.
3. **Efficiency**
   Optimize system performance by managing processes, memory, and tasks effectively.
4. **Security and Protection**
   Protect system resources and user data from unauthorized access or misuse.
5. **Multiprogramming**
   Enable multiple applications to run at the same time for better utilization of CPU.
6. **Job Scheduling**
   Decide the order in which processes will be executed, ensuring fairness and priority handling.
7. **Error Detection and Handling**
   Detect, report, and sometimes correct errors in hardware or software.

Characteristics of an Operating System

1. **Multitasking**
   Supports running multiple tasks (programs) simultaneously.

2. **Multi-user Support**
   Allows multiple users to use the system at the same time or at different times.
3. **Portability**
   Many OS are portable and can work across different hardware with minimal changes.
4. **Security**
   Provides user authentication, data encryption, and permissions to ensure secure access.
5. **Interactivity**
   Provides interaction between users and the system through GUI or CLI.
6. **Time-Sharing**
   Distributes CPU time among multiple users or tasks efficiently.
7. **Real-Time Operation (in RTOS)**
   Responds quickly and predictably to real-world inputs and events.
8. **File System Management**
   Manages data storage and organization through files and directories.

Classification of Operating Systems

1. Batch Operating System

- **Definition**: Executes batches of jobs without user interaction.
- **How it works**: Jobs with similar needs are grouped and processed together.
- **Features**:
  - No direct user interaction.
  - Jobs are executed in the order they arrive.
  - Suitable for large volume, repetitive tasks.
- **Example**: Early IBM systems.

---

2. Multiprogramming Operating System

- **Definition**: Allows multiple programs to reside in memory at the same time.
- **How it works**: The CPU switches to another job if the current one is waiting (e.g., for I/O).
- **Features**:
  - Increases CPU utilization.
  - Better resource management.
  - Requires memory management and job scheduling.
- **Example**: UNIX, older versions of Linux.

---

3. Multiprocessing Operating System

- **Definition**: Uses **two or more processors (CPUs)** to perform tasks simultaneously.
- **How it works**: Tasks are divided among processors, which work in parallel.

- **Features**:
  - Faster processing.
  - High reliability (if one CPU fails, others continue).
  - Used in high-performance systems.
- **Example**: Modern versions of Linux, Windows Server.

---

4. Multitasking Operating System

- **Definition**: Allows a single user to run **multiple tasks (processes)** at the same time.
- **How it works**: CPU switches between tasks quickly (time slicing).
- **Features**:
  - Provides smooth user experience.
  - Efficient use of CPU.
  - Tasks appear to run in parallel.
- **Example**: Windows, macOS, Linux.

---

5. Time-Sharing Operating System

- **Definition**: Allows **multiple users** to use the system **simultaneously** by sharing CPU time.
- **How it works**: Each user gets a time slice of the CPU in a round-robin or priority-based order.
- **Features**:
  - Multi-user environment.
  - Quick response time.
  - Requires scheduling and memory management.
- **Example**: UNIX, mainframe systems.

1. Distributed Network

A **Distributed Network** refers to a system where computing resources are spread across multiple physical locations. These systems work together and communicate over a network (often the internet or a local network) to achieve a common goal.

Key Features:

- Multiple nodes (computers/servers) work together.
- Resources (data, processing power) are shared.
- Fault tolerance (system can continue working even if one node fails).
- Scalability (easy to add more nodes).
- Examples: Peer-to-peer networks, cloud computing, blockchain.

---

2. Real-Time Operating System (RTOS)

An **RTOS** is a type of operating system designed to process data as it comes in, typically within a guaranteed time frame.

Characteristics:

- **Deterministic**: Predictable response times.
- **Low latency**: Fast response to events.
- **Multitasking** with priority scheduling.
- Used in embedded systems, robotics, medical devices, automotive systems.

Examples:

- FreeRTOS
- VxWorks
- RTLinux

---

3. System Calls in Operating Systems

System calls provide an interface between user programs and the operating system. They allow user-level applications to request services from the OS.

Categories of System Calls:

1. **Process Control** (e.g., `fork()`, `exec()`, `exit()`)
2. **File Management** (e.g., `open()`, `read()`, `write()`, `close()`)
3. **Device Management** (e.g., `ioctl()`, `read()`, `write()`)
4. **Information Maintenance** (e.g., `getpid()`, `alarm()`, `sleep()`)
5. **Communication** (e.g., `pipe()`, `shmget()`, `msgsnd()`)

---

4. Operating System Services

Operating system services are the functions that an OS provides to users and programs.

Common Services:

- **Program Execution**
- **I/O Operations**
- **File System Manipulation**

- **Communication Services** (inter-process communication)
- **Error Detection**
- **Resource Allocation**
- **Security and Protection**

Functions and Structure:-

- **Process Management**
- **Memory Management**
- **Secondary Storage Management**

---

1. Process Management

A **process** is a program in execution. The OS manages all processes in the system.

Functions:

- **Process Creation and Termination**: OS creates processes using system calls like `fork()` and terminates them using `exit()`.
- **Scheduling**: Decides which process runs at a given time (using schedulers like Round Robin, Priority Scheduling).
- **Process Synchronization**: Manages process cooperation, especially when sharing resources (e.g., semaphores, mutexes).
- **Process Communication**: Allows processes to exchange data (e.g., pipes, message queues, shared memory).
- **Deadlock Handling**: Detects and resolves situations where processes wait indefinitely for resources.

---

2. Memory Management

Memory management controls and coordinates the computer's main memory (RAM).

Functions:

- **Allocation and Deallocation**: Assigns memory to processes and reclaims it when done.
- **Tracking Memory Usage**: Keeps track of which parts of memory are in use and by whom.
- **Paging and Segmentation**: Uses virtual memory techniques to divide memory logically.
- **Swapping**: Moves processes between RAM and disk when memory is low.

- **Protection and Isolation**: Prevents one process from accessing memory allocated to another.

---

3. Secondary Storage Management

Secondary storage refers to non-volatile storage like hard drives and SSDs. It stores data and programs permanently.

Functions:

- **Space Management**: Allocates and frees up disk space.
- **Scheduling Disk Access**: Manages how and when data is read from or written to disk.
- **File System Management**: Organizes files in directories, manages metadata, access permissions, etc.
- **Backup and Recovery**: Ensures data can be restored in case of failure.

OPERATING SYSTEM FUNCTIONS (continued)

4. Input/Output (I/O) Management

I/O Management controls all the input and output devices like keyboard, mouse, printer, and disk.

Functions:

- **Device Drivers**: Translate OS commands into device-specific operations.
- **I/O Scheduling**: Determines the order in which I/O requests are serviced.
- **Buffering, Caching, Spooling**: Techniques to improve I/O performance.
    - **Buffering**: Temporary storage to manage speed mismatch.
    - **Caching**: Storing frequently used data.
    - **Spooling**: Queueing data for devices like printers.

---

5. File Management

The OS manages files on various storage devices and provides operations on them.

Functions:

- **File Creation and Deletion**
- **Directory Management**
- **File Access Methods** (Sequential, Direct, Indexed)

- **File Protection** (Access control)
- **Metadata Handling** (Name, type, size, permissions)

---

6. Protection and Security

This ensures that system resources are used properly and access is controlled.

Protection:

- Ensures **only authorized processes** can access specific resources (CPU, memory, files).
- Uses techniques like **Access Control Lists (ACLs)** and **user permissions**.

Security:

- Defends against external threats (e.g., viruses, hackers).
- Involves **authentication (passwords, biometrics)** and **encryption**.

□ OPERATING SYSTEM STRUCTURES

OS structure determines how components interact. Here are the main types:

1. Simple Structure

- Minimal structure, code grows without much modularization.
- Example: **MS-DOS**
- □ *Poor separation of concerns.*

---

2. Monolithic Structure

- Entire OS is a **single large program** in kernel space.
- All OS services run in kernel mode.
- Example: **Unix (early versions)**
- □ Fast execution, but □ hard to maintain and extend.

---

3. Layered Approach

- OS is divided into **layers**, each built on top of the lower one.

- Layer 0: Hardware → Layer N: User interface
- Easier to debug and update.
- Example: **THE OS**
- □ Slower due to layer-by-layer communication.

---

4. Microkernel

- Only **core functions** (e.g., communication, scheduling) are in the kernel.
- Other services (e.g., file systems, device drivers) run in **user space**.
- Example: **Minix, QNX**
- □ Better security and stability
- □ More complex and slower messaging.

---

5. Exokernel

- Bare-minimum kernel that gives **direct access to hardware**.
- Applications build their own abstractions.
- Example: **MIT Exokernel**
- □ Very efficient, customizable
- □ Hard to program and maintain.

---

6. Virtual Machine (VM)

- OS creates an illusion of multiple independent computers (VMs) on a single machine.
- Each VM can run its own OS.
- Example: **VMware, VirtualBox**
- □ Isolation, testing, resource sharing
- □ Overhead due to virtualization.

□ Summary Table:

OS Function / Structure	Description
I/O Management	Handles devices, drivers, buffering, and scheduling
File Management	Manages file creation, access, directories, permissions
Protection & Security	Safeguards resources & users through access control & authentication

OS Function / Structure	Description
Simple Structure	No clear modularity, direct system calls (e.g., MS-DOS)
Monolithic	Entire OS in one kernel program (e.g., UNIX)
Layered	OS divided into layers for clarity and modularity
Microkernel	Minimal kernel; services run in user space
Exokernel	Minimal abstraction; apps manage resources directly
Virtual Machine	Multiple OSes run on one machine via abstraction layer

UNIT – 2

Introduction to Process Management and Scheduling

▯ What is a Process?

A **process** is a program that is currently **being executed** by the CPU. It is more than just the program code — it includes:

- The program code (text section)
- Current activity (program counter, registers)
- Stack (function calls)
- Data (variables, heap)

When you open an app or run a command, the operating system creates a **process** to handle it.

▯ What is Process Management?

Process Management is one of the key responsibilities of an Operating System (OS). It involves **creating, executing, controlling, and terminating processes**.

Key Tasks of Process Management:

- **Process creation and deletion**
- **Scheduling** (deciding which process runs next)
- **Synchronization** (managing processes that share resources)
- **Communication** (between processes, using IPC)
- **Deadlock handling** (preventing or resolving resource conflicts)

The OS keeps information about each process in a data structure called the **Process Control Block (PCB)**.

□ What is CPU Scheduling?

The CPU can run **only one process at a time** (in a single-core system). So, when multiple processes are ready to run, the OS must **decide which one runs next**. This is called **CPU Scheduling**.

Why Scheduling is Important:

- Ensures fair use of CPU
- Improves system performance
- Reduces waiting time and response time
- Increases CPU efficiency

□ Objectives of Scheduling:

- **Maximize CPU utilization**
- **Minimize waiting time**
- **Minimize turnaround time**
- **Minimize response time**
- **Ensure fairness among processes**

□ Common Scheduling Algorithms:

Algorithm	Type	Description
FCFS	Non-preemptive	First Come First Served

Algorithm	Type	Description
SJF	Both	Shortest Job First
Priority	Both	Based on process priority
Round Robin	Preemptive	Equal time slice per process
Multilevel Queue	Preemptive	Different queues for different types of processes

---

□ Process Life Cycle (Simplified):

```
plaintext
CopyEdit
      New
       ↓
    Ready → Running → Terminated
       ↑       ↓
    Waiting ←--
```

PROCESS CONCEPT – Theory

A **process** is defined as a **program in execution**. It is not just the program code but includes a complete execution context. The operating system (OS) is responsible for creating, managing, and terminating processes.

□ Components of a Process:

A process consists of the following elements:

1. **Program Code (Text Section)** – The actual code to be executed.
2. **Program Counter** – Points to the next instruction to execute.
3. **Stack** – Contains temporary data such as function parameters and return addresses.
4. **Heap** – Used for dynamic memory allocation.
5. **Data Section** – Contains global and static variables.
6. **CPU Registers and Process Status**

□ Process Control Block (PCB):

Every process is represented in the OS by a **Process Control Block**, which stores:

- Process ID (PID)
- Current process state
- CPU registers
- Scheduling information

- Memory management info
- I/O status

The PCB allows the OS to **manage and switch between processes** efficiently.

□ PROCESS STATE MODEL – Theory

A process goes through several **states** from creation to termination. These states represent the **lifecycle** of a process.

□ Main Process States:

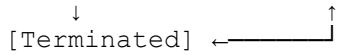
State	Description
New	Process is being created.
Ready	Process is loaded into memory and waiting for CPU time.
Running	The process is currently being executed on the CPU.
Waiting (Blocked)	The process is waiting for an event or I/O operation to complete.
Terminated	The process has finished execution or has been killed.

□ State Transitions:

- **New → Ready**: Process is ready to run after being created.
- **Ready → Running**: Scheduler selects it to use the CPU.
- **Running → Waiting**: The process waits for I/O.
- **Running → Ready**: Preemption (CPU is taken away).
- **Waiting → Ready**: I/O or event is completed.
- **Running → Terminated**: Process finishes or is stopped.

□ Process State Transition Diagram:

```
css
CopyEdit
    [New]
      ↓
    [Ready] ←──────────┐
      ↓                ↑
    [Running] ──→ [Waiting]
```



\PROCESS CONTROL BLOCK (PCB) – Theory

The **Process Control Block (PCB)** is a **data structure** maintained by the operating system for every process. It contains **all the information** about a particular process. The OS uses the PCB to **track process execution and manage scheduling**.

Contents of a PCB:

Field	Description
Process ID (PID)	Unique identifier for the process
Process State	Current status (New, Ready, Running, Waiting, Terminated)
Program Counter	Address of the next instruction to execute
CPU Registers	Contents of CPU registers when the process is not running
Memory Management Info	Page tables, segment tables, base and limit registers
Accounting Info	CPU time used, process priority, etc.
I/O Status Info	List of I/O devices assigned and I/O requests
Parent/Child Process Info	Relationships for process trees

Purpose of PCB:

- Helps in **context switching**
- Maintains the **state of the process**
- Supports **scheduling, resource allocation, and protection**

THREADS – Theory

A **thread** is the **smallest unit of CPU execution** within a process. A process can contain **one or more threads**.

🔹 Definition:

A **thread** is a **lightweight process** that shares the **same resources** (memory, files, code) with other threads in the same process, but has its **own execution context** (stack, program counter, registers).

🔹 Parts Shared Between Threads (in the same process):

- Code segment
- Data segment
- Open files and I/O resources

🔹 Parts Unique to Each Thread:

- Program Counter
- Registers
- Stack

🔹 Types of Threads:

Type	Description
User-Level Threads (ULT)	Managed by user-level libraries, not visible to OS
Kernel-Level Threads (KLT)	Managed directly by the OS
Hybrid Threads	Combine both user and kernel thread management

---

🔹 Advantages of Threads:

- Faster context switching than processes
- Efficient CPU utilization in multiprocessor systems
- Easy inter-thread communication (shared memory)
- Good for responsive applications (e.g., UI + background work)

🔹 Difference: Process vs. Thread

Feature	Process	Thread
Memory	Separate memory	Shared memory

Feature	Process	Thread
Creation	Slow	Fast
Communication	Through IPC	Shared variables
Overhead	High	Low
Crash effect	One process crash doesn't affect others	One thread crash can affect the whole process

PROCESS SCHEDULING – Theory

? What is Process Scheduling?

Process Scheduling is the **activity of the operating system** that decides which process will run **next** on the **CPU** when there are **multiple processes** in the **Ready queue**.

The main goal is to:

- Maximize CPU utilization
- Provide fairness
- Minimize waiting time and turnaround time

? Why is Scheduling Needed?

- The CPU can run only **one process at a time** (in single-core systems).
- When multiple processes are **ready**, the OS must decide **which one to run next**.
- This decision is made by the **CPU Scheduler (short-term scheduler)**.

? Objectives of CPU Scheduling:

- **Efficiency** (keep the CPU busy)
- **Fairness** (equal chance for all processes)
- **Low waiting time**
- **High throughput** (more processes completed per unit time)
- **Low response time** (especially in interactive systems)

? SCHEDULING QUEUES – Theory

The Operating System maintains several **queues** to manage processes at different stages:

✅ Types of Scheduling Queues:

Queue	Description
Job Queue	Contains all processes in the system (new, ready, waiting, etc.)
Ready Queue	Contains processes that are loaded in memory and ready to run
Waiting (I/O) Queue	Contains processes waiting for I/O operations or events
Device Queues	One queue for each I/O device; holds processes waiting for that device

---

✅ Process Movement Between Queues:

Example flow:

1. A new process enters the **job queue**.
2. Once admitted to memory, it moves to the **ready queue**.
3. When selected by the **CPU scheduler**, it enters the **running state**.
4. If it needs I/O, it moves to the **waiting (I/O) queue**.
5. Once I/O is done, it returns to the **ready queue**.
6. After completing execution, it is **terminated**.

🖼 Diagram: Scheduling Queues (Text View)

```
sql
CopyEdit
        +-------------+
        |  Job Queue  |
        +-------------+
               ↓
        +-------------+
        | Ready Queue | ←──────────────┐
        +-------------+                │
               ↓                       │
        +-------------+                │
        |  Running    |                │
        +-------------+                │
          ↓           ↓                │
  [Waiting Queue] [Terminated]         │
          ↓                            │
          └──────→ Ready Queue ←───────┘
```

SCHEDULERS – Theory

A **Scheduler** is a part of the Operating System responsible for **selecting which process** should be executed **next** on the CPU.

There are **three main types** of schedulers:

1. Long-Term Scheduler (Job Scheduler)

Feature	Description
Role	Selects which **new processes** are admitted into the **system for execution**.
Frequency	Infrequent (runs occasionally)
Goal	Controls the **degree of multiprogramming** (number of processes in memory)
Example	Deciding which jobs in the job queue move to the ready queue

---

2. Short-Term Scheduler (CPU Scheduler)

Feature	Description
Role	Selects from the **ready queue** which process will run on the CPU next
Frequency	Runs **frequently** (milliseconds)
Goal	Fast and efficient process selection
Example	Choosing between two processes in ready state

---

3. Medium-Term Scheduler (Swapper)

Feature	Description
Role	Temporarily **removes processes from memory** (swapping) to reduce load
Frequency	As needed (depends on system load)
Goal	Improves performance and frees up memory
Example	Swapping out background processes when memory is low

□ CONTEXT SWITCH – Theory

A **Context Switch** occurs when the CPU changes from executing one process to another.

□ What Happens During a Context Switch?

The OS:

1. Saves the state of the **current process** (in its PCB)
2. Loads the state of the **next scheduled process**
3. Updates **CPU registers, program counter, and stack**

□ Information Saved in a Context Switch:

- Program Counter
- CPU Registers
- Stack Pointer
- Memory Management Info
- Process State

□ Is Context Switching Good?

Advantage	Disadvantage
Allows multitasking	Adds overhead (time when CPU does no useful work)

□ How to Minimize Context Switch Overhead:

- Use efficient scheduling algorithms
- Minimize switching frequency unless necessary
- Prefer lightweight threads where possible

OPERATIONS ON PROCESSES

The Operating System (OS) provides several operations for managing processes.

✅ Common Process Operations:

Operation	Description
Process Creation	A new process is created using system calls like `fork()` (in UNIX). A parent creates one or more child processes.
Process Termination	A process finishes execution using `exit()`, or is terminated by another process.
Process Suspension	A process is paused (e.g., due to memory shortage).
Process Resumption	A suspended process is brought back into the ready queue.
Process Wait/Signal	Used for synchronization between processes.
Orphan and Zombie Processes	Orphan: Parent terminated before child. Zombie: Child terminated but parent hasn't read its exit status.

🔄 COOPERATING PROCESSES

✅ What are Cooperating Processes?

Cooperating processes are processes that **can affect or be affected** by other processes. They **share data** and **communicate** with each other to solve a task.

✅ Reasons for Cooperation:

1. **Information Sharing** – e.g., multiple programs accessing the same database.
2. **Computation Speed-up** – break tasks into parallel sub-tasks.
3. **Modularity** – split system into smaller cooperating pieces.
4. **Convenience** – background processes handling specific tasks (e.g., file downloads).

🔗 INTER-PROCESS COMMUNICATION (IPC)

IPC is a mechanism that allows **processes to communicate and synchronize** their actions when working together.

✅ Two Main IPC Models:

IPC Model	Description
Shared Memory	Processes communicate by reading/writing to a shared region in memory. Fast but needs synchronization tools (like semaphores, mutexes).
Message Passing	Processes exchange messages using `send()` and `receive()` functions. Easier to implement in distributed systems.

✅ Direct vs Indirect Communication (Message Passing)

Type	Description
Direct	Processes send messages to a specific process (using PID or name).
Indirect	Messages sent to and received from **mailboxes** (message queues).

✅ Synchronization in IPC:

- **Blocking**: Sender/receiver waits until the message is received/sent.
- **Non-blocking**: Processes continue without waiting.

Process Scheduling:-

SCHEDULING CRITERIA

Scheduling criteria are the **goals or objectives** that scheduling algorithms try to achieve. They help evaluate how well a CPU scheduling algorithm performs.

✅ Common Scheduling Criteria:

Criteria	Description
CPU Utilization	Keep the CPU as busy as possible (ideally 100%).
Throughput	Number of processes completed per time unit. Higher is better.

Criteria	Description
Turnaround Time	Total time taken from process submission to completion.
Waiting Time	Time a process spends in the ready queue waiting for CPU.
Response Time	Time from process submission to the first CPU response (important in interactive systems).
Fairness	Each process should get a fair share of CPU.

□ SCHEDULING ALGORITHMS

Scheduling algorithms decide **which process gets the CPU** and for **how long**. They affect system performance, responsiveness, and fairness.

□ 1. First-Come, First-Served (FCFS)

- **Type**: Non-preemptive
- **Idea**: Processes are executed in the order they arrive.

□ Simple to implement
□ Can lead to**convoy effect** (short jobs wait behind long ones)

□ 2. Shortest Job First (SJF)

- **Type**: Non-preemptive or preemptive
- **Idea**: Process with the shortest burst time is scheduled first.

□ Minimum average waiting time
□ Requiresknowledge of future burst times

□ 3. Priority Scheduling

- **Type**: Can be preemptive or non-preemptive
- **Idea**: CPU assigned to the process with the **highest priority**.

 Good for important tasks
 Can cause**starvation** (low-priority processes may never run)

 Aging is used to prevent starvation (priority increases with time).

---

4. Round Robin (RR)

- **Type**: Preemptive
- **Idea**: Each process gets a **fixed time slice** (quantum) in a cyclic order.

 Good for**time-sharing** systems
 Performance depends on the length of thetime quantum

---

5. Multilevel Queue Scheduling

- **Type**: Preemptive or Non-preemptive
- **Idea**: Multiple queues for different types of processes (e.g., foreground, background), each with its own scheduling algorithm.

 Organized and structured
 No movement between queues (unless **multilevel feedback queue** is used)

---

6. Multilevel Feedback Queue Scheduling

- **Type**: Preemptive
- **Idea**: Like multilevel queue, but processes **can move between queues** based on behavior.

 Most flexible and adaptive
 Complex to implement

Evaluation of Scheduling Algorithms in Operating Systems

Scheduling Algorithm Evaluation is the process of **comparing** and **measuring** how well different CPU scheduling algorithms perform, based on **standard criteria**.

---

1. Evaluation Criteria

Criteria	Description
CPU Utilization	Keep CPU busy; higher is better (goal: 80–100%).
Throughput	Number of completed processes per time unit.
Turnaround Time	Total time from submission to completion.
Waiting Time	Time spent in the ready queue (idle).
Response Time	Time from submission to first CPU response.
Fairness	Equal treatment of all processes (avoiding starvation).
Predictability	Consistent performance under different loads.

□ 2. Methods of Evaluation

There are three main ways to evaluate scheduling algorithms:

□ a. Deterministic Modeling

- Use mathematical formulas and **known inputs** (arrival time, burst time).
- Example: Calculate **waiting time**, **turnaround time**, etc. for each algorithm.
- □ Simple and precise
- □ Not practical for large, real-world systems

□ b. Queueing Models

- Use **probability/statistics** to model arrival and service times.
- Represent the system as **queues** (e.g., using Poisson process for arrivals).
- □ Realistic and useful for**theoretical predictions**
- □ Complex mathematics involved

□ c. Simulation

- **Simulate** real process behavior using software.
- Feed actual or synthetic workloads into a **scheduler simulator**.
- □ Models real-world performance

- □ Time-consuming, needs programming or tools

□ 3. Sample Metrics Calculation

Example: Consider 3 processes with the following burst times:

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	8

You can simulate FCFS, SJF, and Round Robin for this data and compute:

- **Average Waiting Time**
- **Average Turnaround Time**
- **CPU Utilization** (if idle time is known)

□ 4. Comparing Algorithms (General Trends)

Algorithm	Avg Waiting Time	Response Time	Fairness	Use Case
FCFS	High (can vary)	Slow	Low	Simple batch jobs
SJF	Lowest (ideal)	Can vary	Poor	Known burst times
Priority	Can vary	Can be fast	Poor	Real-time tasks
Round Robin	Moderate	Fast	High	Interactive systems
Multilevel Queue	Varies by policy	Varies	Medium	System with distinct job types
Feedback Queue	Good	Fast	High	General-purpose OS

UNIT – 3

Memory Management:-

What is Memory Management?

Memory management is the function of an operating system (OS) that handles **allocation and deallocation of memory** space to various programs during their execution.

It ensures:

- Efficient memory use
- Safe memory access
- Isolation between processes

Concept of Memory Management

Memory Management is a core function of the operating system (OS) that handles the **allocation and deallocation** of memory to processes.

Key Goals:

- **Efficient memory utilization**
- **Safe and protected memory access**
- **Multiprogramming support** (multiple processes in memory)
- **Minimize fragmentation**

The OS must **track**:

- Which memory is in use
- Who is using it
- When to allocate/deallocate memory

Logical vs. Physical Address Space

These terms describe how addresses are handled in a system.

---

1. Logical Address (Virtual Address)

- Address generated by the **CPU** during program execution.
- Used by the **process (user)** to access memory.
- It is **virtual** because it needs to be **translated** into a physical address.

 Generated by: **CPU**
 Translated by: **Memory Management Unit (MMU)**

---

2. Physical Address

- Actual address in **physical RAM** (main memory).
- Used by the **hardware** to access data.

 Managed by: **Operating System + Hardware**

Address Translation

The **MMU (Memory Management Unit)** translates **logical address → physical address** using techniques like:

- **Paging**
- **Segmentation**
- **Offset + base registers**

Example:

Let's say:

- Logical Address = 1200
- Base Address (loaded into memory) = 4000
- Then,
  Physical Address = 4000 + 1200 = 5200

Address Spaces

Type of Address Space	Description
Logical Address Space	Set of addresses generated by a program (used by the CPU).
Physical Address Space	Set of addresses in main memory (actual RAM).

1. Swapping

Swapping is the process of **moving processes between RAM and disk** to manage memory efficiently.

✅ Key Points:

- When **RAM is full**, the OS may **swap out** a process to disk (swap space).
- When needed again, the process is **swapped back** into RAM.
- Enables **multiprogramming** even with limited memory.

✅ Frees memory temporarily
❌ Swapping overheadmay slow performance

✅ 2. Memory Allocation: Contiguous & Non-Contiguous

✅ A. Contiguous Memory Allocation

- Each process is assigned **one continuous block** in memory.
- Two types:
    1. **Fixed Partitioning** – Predefined memory sizes
    2. **Variable Partitioning** – Allocated based on process size

Pros	Cons
Simple to implement	External fragmentation
Fast access	Difficult resizing

✅ B. Non-Contiguous Memory Allocation

- A process is stored in **multiple blocks at different memory locations**.
- Uses **paging**, **segmentation**, or both.

Pros	Cons
No external fragmentation	Needs complex mapping
Efficient memory use	Overhead of page tables

□ 3. Paging (with Hardware Support)

Paging is a non-contiguous memory management technique where:

- **Logical memory** is divided into **pages**
- **Physical memory** is divided into **frames**
- Each page maps to a frame via the **Page Map Table**

□ Paging Hardware Support

1. **Page Table** – Maps logical page numbers to physical frame numbers.
2. **MMU (Memory Management Unit)** – Performs address translation.
3. **TLB (Translation Lookaside Buffer)** – Fast cache to store recent translations.

□ Address Translation Example:

```
mathematica
CopyEdit
Logical Address = (Page Number, Offset)
MMU → Translates → Frame Number + Offset = Physical Address
```

□ 4. Page Map Table (PMT)

- Stores **mappings between logical pages and physical frames**
- Each entry contains:
  - **Frame number**
  - **Valid/Invalid bit**
  - **Protection bits** (read/write/execute)
  - **Present/absent bit** (if using virtual memory)

□ 5. Protection in Paging

- The OS uses **protection bits** in the page table:

    - **Read-only / Read-Write / Execute** permissions
- Illegal memory access → triggers a **trap/interrupt**
- Each process has its **own page table** for isolation

Segmentation in Operating Systems

❓ What is Segmentation?

Segmentation is a **memory management scheme** in which a process is divided into **variable-sized segments** based on **logical divisions**, such as:

- Code segment
- Data segment
- Stack segment
- Heap segment

Each segment has:

- Its own **base** (starting address in physical memory)
- Its own **limit** (length of segment)

❓ Segmentation Hardware Support

To manage and translate segment-based addresses, the system uses a **Segment Table** and hardware support from the **MMU (Memory Management Unit)**.

❓ Segment Table Contains:

Field	Description
Base	Starting physical address of segment
Limit	Size of the segment
Access rights	Read/Write/Execute protection

❓ Address Translation in Segmentation:

Logical address = ⟨segment number, offset⟩
Physical address = **base[segment number] + offset**

 Offset must be < limit, or a **segmentation fault** occurs

□ Protection in Segmentation

Segmentation provides **natural protection** because:

- Each segment is **independent**, with its **own access control**
- **Illegal access** (e.g., stack trying to access code) causes a **trap**
- OS sets **access rights** (read, write, execute) in the segment table

Segment Common Rights

Code	Execute only
Data	Read/Write
Stack	Read/Write

□ Sharing in Segmentation

Segmentation allows **easy sharing** of memory between processes.

Example:

- Multiple processes can **share a code segment** (read-only).
- OS maps **the same segment** into different processes' segment tables.

 Efficient memory use
 No duplication of shared code

Virtual Memory :-

1. Need for Virtual Memory

❓ What is Virtual Memory?

Virtual Memory is a memory management technique that allows execution of processes that **may not be completely in main memory**.

❓ Why is it needed?

Reason	Description
✅ **Run large programs**	Programs can be larger than physical memory
✅ **Multiprogramming**	More processes can be loaded at once
✅ **Isolation**	Each process has its own address space
✅ **Efficient use of memory**	Only needed parts are loaded

✅ 2. Demand Paging

❓ What is Demand Paging?

Demand paging is a technique where **pages are loaded into memory only when they are needed**, not in advance.

- Pages not in memory cause a **page fault**
- OS loads the page from disk into RAM

❓ Components:

- **Page Table**: Keeps track of page locations and validity
- **Valid/Invalid Bit**: Indicates if the page is in memory

✅ 3. Pure Demand Paging

In **pure demand paging**:

- **No pages** are loaded initially
- Pages are brought in **only on first reference**
- Every first access to a page causes a **page fault**

 Saves memory
 Causes more initial page faults

---

4. Handling Page Faults

Steps to Handle a Page Fault:

1. **Trap to OS** due to invalid page access
2. **Check** if the reference is valid
3. **Locate** the page on disk
4. **Choose** a free frame (or use page replacement)
5. **Load** the page into RAM
6. **Update** the page table
7. **Restart** the instruction

If no free frame:

- Use a **page replacement algorithm** (e.g., FIFO, LRU)

---

5. Performance of Demand Paging

Key Factors:

Metric	Description
Page Fault Rate	Percentage of memory accesses that cause a fault
Effective Access Time (EAT)	Time based on memory + page fault delay

Effective Access Time Formula:

$$EAT = (1 - p) \times \text{memory access time} + p \times \text{page fault time}$$

Where:

- ppp = Page fault rate ($0 \leq p \leq 1$)
- Page fault time is usually **much higher** (ms vs ns)

□ Too many page faults? → Thrashing

- System spends more time swapping than executing

1. Page Replacement Algorithms

When there is **no free frame** in RAM, and a page fault occurs, the OS uses a **Page Replacement Algorithm** to decide **which page to remove** from memory.

□ Common Algorithms:

□ 1. FIFO (First-In, First-Out)

- Oldest loaded page is removed first.

□ Simple
□ Can remove frequently used pages
□ Suffers from**Belady's Anomaly**

□ 2. LRU (Least Recently Used)

- Removes the page that was **least recently used**.

□ Good performance
□ Expensive to implement (needs time-stamp or stack)

□ 3. Optimal Page Replacement

- Replaces the page that **won't be used for the longest time in the future**.

□ Best possible result
□ Cannot be implemented in practice (requires future knowledge)

□ 4. Clock (Second Chance)

- Circular buffer; gives pages a **second chance** before replacement.

 Efficient and approximates LRU
 Uses a **reference bit**

Example:

For a reference string: `7, 0, 1, 2, 0, 3, 0, 4`

- Run these algorithms to compare page faults for different frame counts.

---

2. Allocation of Frames

Frame allocation determines **how many frames** to assign to each process.

Types of Allocation:

1. Equal Allocation

- Every process gets the **same number of frames**.

 May be unfair to large processes.

---

2. Proportional Allocation

- Each process gets frames **based on its size**:

Frames for process i=(size of process itotal size of all processes)×total frames\text{Frames for process i} = \left(\frac{\text{size of process i}}{\text{total size of all processes}}\right) \times \text{total frames}Frames for process i=(total size of all processessize of process i)×total frames

 Fairer allocation
 Needs calculation

---

3. Priority Allocation

- Frames given **based on priority level** of each process.

 High-priority processes get more frames.

□ 3. Global vs Local Allocation

Feature	Global Allocation	Local Allocation
Frames scope	Shared across all processes	Fixed to each process
Page replacement	Can replace any process's page	Replaces only within the process
Flexibility	More flexible	More predictable
Performance	Better in light load	Better in high load

□ 4. Thrashing

□ What is Thrashing?

Thrashing occurs when the OS spends **more time swapping pages** in and out than executing actual processes.

□ Causes:

- **Too many processes**
- **Too few frames per process**
- High **page fault rate**

□ Effects:

- CPU usage drops
- System slows down severely

□ Solution:

- Use **working set model**
- **Reduce degree of multiprogramming**
- Use **local replacement** instead of global
- Increase RAM if possible

UNIT - 4

Introduction to I/O Management:-

✅ What is I/O Management?

I/O (Input/Output) Management is the component of the operating system responsible for **handling communication between the computer system and external devices**, such as:

- Keyboards
- Mice
- Monitors
- Disks
- Printers
- USB drives
- Network cards

Basic I/O Devices & Types of I/O Devices

✅ What is an I/O Device?

An **I/O (Input/Output) device** is any hardware used by a computer to **communicate with the outside world** (input or output data).

✅ Types of I/O Devices

Type	Description	Examples
Input Devices	Send data **to** the computer	Keyboard, Mouse, Scanner
Output Devices	Receive data **from** the computer	Monitor, Printer, Speaker
Storage Devices	Store data permanently	HDD, SSD, USB drives
Communication Devices	Allow data transfer between systems	Network card, Modem, Bluetooth

---

✅ Block vs. Character Devices

🔹 Block Devices

- Store and access data in **fixed-size blocks**
- Support **random access**
- Often used for **storage**

Examples: Hard disk, SSD, CD/DVD, USB

🔹 Character Devices

- Handle data **one character (or byte) at a time**
- Support only **sequential access**
- Often used for **I/O streams**

Examples: Keyboard, Mouse, Serial Port, Printer

🧱 I/O Software Layers

Operating systems use **layered software** to manage I/O efficiently.

🔹 1. Device-Independent I/O Software

- Provides **uniform access** to different devices
- Hides **device-specific details**
- Handles:
  - Naming (e.g., `/dev/usb0`)
  - Buffering
  - Error reporting
  - Blocking/Non-blocking I/O

✅ Makes it easier for apps to work with any device

🔹🔹 2. User-Space I/O Software

- Part of the **application layer**
- Uses **system calls** (like `read()`, `write()`, `open()`)
- Invokes kernel-level I/O
- Includes libraries like `stdio.h` in C (`printf`, `scanf`)

3. Kernel I/O Subsystem

Runs in the **kernel space** and manages:

- Device drivers (device-specific control)
- Interrupt handling
- I/O scheduling
- Caching and buffering
- DMA (Direct Memory Access) control

 Ensures**safe and efficient** communication with hardware

Summary Table

Concept	Description
I/O Device	Hardware for data input/output
Block Device	Data in blocks, random access
Character Device	Byte-by-byte, sequential access
Device-Independent I/O	Abstracts hardware details
User-Space I/O Software	Interfaces with user programs
Kernel I/O Software	Manages low-level hardware interaction

Device Controllers, Device Drivers, and Interrupt Handlers

---

1. Device Controller

- A **hardware component** that manages a specific type of I/O device.
- Responsible for:
  - Receiving commands from CPU
  - Controlling the I/O device
  - Transferring data to/from memory
  - Signaling the CPU when tasks are complete

 Each type of device (disk, printer, etc.) has a corresponding controller.

---

2. Device Driver

- A **software module** that communicates between the **OS** and a **device controller**.
- Converts generic OS instructions into device-specific commands.
- Runs in **kernel mode**.

 Role: Makes I/O devices appear standard to the OS.

---

3. Interrupt Handler

- A **kernel routine** that executes when a device sends an **interrupt signal**.
- Used to:
  - Notify CPU when I/O is complete
  - Handle errors or status changes
  - Resume the process waiting for I/O

 Improves efficiency by**avoiding busy waiting**.

Communication Approaches to I/O Devices

---

1. Special Instruction I/O (Programmed I/O)

- Uses **special CPU instructions** (e.g., `IN`, `OUT` in x86) to communicate with devices.
- Sends data directly from CPU to the device.

 Inefficient: CPU is busy waiting during the entire I/O operation.

---

2. Memory-Mapped I/O

- I/O device registers are **mapped into the address space** of the CPU.
- CPU uses **standard read/write instructions** to interact with devices like memory.

 Simplifies CPU design
 Can reduce available addressable memory

□ Direct Memory Access (DMA)

□ What is DMA?

DMA allows devices to transfer data **directly to/from main memory** without involving the CPU.

□ How DMA Works:

1. CPU sets up the DMA controller with:
   - Source and destination addresses
   - Data length
2. DMA controller transfers data while CPU performs other tasks
3. DMA sends **interrupt** when transfer is complete

□ Advantages of DMA:

Feature	Benefit
High speed	Faster than CPU-based transfers
Low CPU usage	CPU is free for other tasks
Efficient for large blocks	Ideal for disk, network, and graphics I/O

□ Summary Table

Component/Concept	Description
Device Controller	Hardware that interfaces with I/O devices
Device Driver	Software that controls device-specific actions
Interrupt Handler	Responds to signals from I/O devices
Programmed I/O	CPU actively transfers data (inefficient)
Memory-Mapped I/O	Devices accessed like memory using load/store

Component/Concept	Description
DMA	Data transfer between memory and devices without CPU

1. Secondary Storage Structure

✅ What is Secondary Storage?

Secondary storage is **non-volatile** storage used to **store data permanently** (unlike RAM which is temporary).

✅ Examples of Secondary Storage:

- Hard Disk Drives (HDDs)
- Solid-State Drives (SSDs)
- CDs, DVDs
- USB Flash Drives
- Magnetic Tapes

✅ Purpose:

- Long-term data storage
- Backup and archiving
- Holding OS, software, files, etc.

□ 2. Disk Structure

✅ Disk Organization (in HDDs):

Term	Meaning
Platter	A circular disk where data is stored magnetically
Track	A concentric circle on a platter
Sector	A segment of a track (smallest data unit)
Cylinder	All tracks aligned vertically through platters

Term	Meaning
Read/Write Head	Reads or writes data by moving across the disk

---

⏱ Disk Access Time = Seek Time + Rotational Latency + Transfer Time

Component	Description
Seek Time	Time to move the head to the correct track
Rotational Latency	Time waiting for the desired sector to rotate under head
Transfer Time	Time to actually transfer data

🧠 3. Disk Scheduling Algorithms

Disk scheduling is used to decide **the order in which disk I/O requests** are processed to **minimize seek time**.

🔄 Common Disk Scheduling Algorithms:

---

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

- Requests are handled in the order they arrive.

✅ Simple
❌ May cause long waits

---

2. SSTF (Shortest Seek Time First)

- Chooses the request **closest to current head position**.

✅ Faster than FCFS
❌ May cause**starvation** of far requests

---

3. SCAN (Elevator Algorithm)

- Head moves in one direction, servicing all requests, then reverses.

 Fair and efficient
 Services requests in both directions

---

4. C-SCAN (Circular SCAN)

- Like SCAN but only services in one direction, then quickly returns to the beginning.

 More uniform wait times than SCAN

---

5. LOOK and C-LOOK

- Like SCAN/C-SCAN, but head only goes **as far as the last request**, then reverses.

 Avoids scanning unnecessary empty tracks

? Example Comparison:

For head at track 53 and requests: 98, 183, 37, 122, 14, 124, 65, 67

Algorithm	Total Head Movement
FCFS	High (depends on order)
SSTF	Lower than FCFS
SCAN	Medium
C-SCAN	Medium–High
LOOK	Lower than SCAN if requests are concentrated
C-LOOK	More efficient than C-SCAN

□ Summary Table

Topic	Key Point
Secondary Storage	Non-volatile storage like HDD, SSD
Disk Structure	Organized into platters, tracks, sectors
Disk Scheduling	Optimizes I/O request order
SSTF	Chooses closest request
SCAN/C-SCAN	Services in order across disk
LOOK/C-LOOK	Services only as far as needed

FILE SYSTEM INTERFACE:

1. File Concept

A **file** is a collection of related data stored on secondary storage (like a hard drive), identified by a **filename**.

Files are used to store:

- Programs
- Documents
- Images, Videos, etc.

□ 2. File Attributes

Every file has **metadata** (data about data), such as:

Attribute	Description
Name	Human-readable identifier

Attribute	Description
Type	File extension (e.g., `.txt`, `.exe`)
Location	Address on disk
Size	Number of bytes in the file
Protection	Permissions (read/write/execute)
Time & Date	Creation, modification timestamps
Owner	User ID or owner of the file

□ 3. File Operations

The OS provides system calls to perform operations on files:

- `create()`
- `open()`
- `read()`
- `write()`
- `close()`
- `delete()`
- `seek()`
- `truncate()`

□ 4. File Types

Type	Description
Text	Human-readable files
Binary	Machine-readable (e.g., `.exe`)
Executable	Ready-to-run programs
Multimedia	Images, videos, audio

□ 5. File Access Methods

□ a) Sequential Access

- Data is read **in order**, one record after another.
- Like a cassette tape.

□ Used in text files, logs.

□ b) Direct Access (Random Access)

- Jump to any block or location directly.
- Like a CD or DVD.

□ Used in databases, media files.

□ c) Indexed Sequential Access

- Combines **indexing** with sequential access.
- Index points to blocks for faster access.

□ Used in large files like dictionaries.

□ 6. Free Space Management

OS tracks **free blocks** on disk to allocate files.

Techniques:

Method	Description
Bit Vector	1 = used, 0 = free (bitmap)
Linked List	Each free block points to next
Grouping	Stores addresses of free blocks in groups
Counting	Stores start block and number of free blocks

□ 7. Directory Structure

Directories organize files into a **hierarchy**.

□ a) Single-Level Directory

- All files in **one directory**.

□ Simple
□ Name conflicts; poor organization

□ b) Two-Level Directory

- Each user gets a **separate directory**.

□ No name conflicts
□ No subdirectories for organizing files

□ c) Tree-Structured Directory

- Allows **nested directories** (subdirectories).

□ Organized, scalable
□ Supports pathnames (e.g., `/home/user/docs`)

□ 8. File Protection and Sharing

□ File Protection:

- Prevents **unauthorized access**.
- Permissions (Unix style): `r` (read), `w` (write), `x` (execute)
- Access Control Lists (ACLs)

🔗 File Sharing:

- Users can share files with **controlled access**.

Methods:

- **By links** (symbolic or hard)
- **By network file sharing (e.g., NFS)**
- With **access control**

📌 Summary Table

Concept	Key Point
File Attributes	Metadata about file
File Operations	Create, read, write, etc.
Access Methods	Sequential, direct, indexed
Free Space Management	Bitmap, list, grouping
Directory Structures	Single, two-level, tree
Protection & Sharing	Permissions, ACL, multi-user sharing