
# Operating System

## (LECTURE NOTES)

**Prepared by:**

**Er. AKSHAY KUMAR PATRA**

**(Assistant Professor)**

**DEPT. OF COMPUTER SCIENCE. & ENGINEERING**

**Modern Engineering and Management Studies**

**Banaparia, Kuruda, Balasore, Odisha.**

# MODULE-1

## Definition of Operating System (OS)

An **Operating System** is system software that manages **hardware** and **software** resources of a computer. It acts as an **intermediary between users and the computer hardware**.

---

## Types of Operating Systems

| Type | Description | Examples |
|---|---|---|
| **Batch OS** | No interaction with users; jobs processed in batches. | IBM OS/360 |
| **Time-Sharing OS** | Allows multiple users to use the system interactively. | UNIX, MULTICS |
| **Distributed OS** | Controls a group of distinct computers and makes them appear as a single system. | LOCUS |
| **Real-Time OS** | Provides immediate processing; used in critical systems. | VxWorks, RTLinux |
| **Embedded OS** | Used in embedded systems (small, dedicated devices). | Android (for mobile), QNX |
| **Network OS** | Manages networked computers and resources. | Novell NetWare, Windows Server |
| **Mobile OS** | Designed specifically for mobile devices. | iOS, Android |

---

## Functions of Operating System

- **Process Management**: Manages processes in a multitasking system.
- **Memory Management**: Allocates memory to processes and reclaims it when done.
- **File System Management**: Manages files on storage devices.
- **Device Management**: Manages device communication via drivers.
- **Security and Access Control**: Protects data and resources.
- **User Interface**: Provides GUI/CLI for user interaction.

### Diagram: Basic Functionality of OS

```
pgsql
CopyEdit
+------------------------+
|      User Programs     |
+------------------------+
           |
           v
+------------------------+
|     Operating System    |
|------------------------|
|  Process Management     |
|  Memory Management      |
|  File System            |
|  Device Management      |
|  Security & Interface   |
+------------------------+
           |
           v
+------------------------+
|    Hardware (CPU, RAM, |
|      I/O Devices)      |
+------------------------+
```

---

## Abstract View of an Operating System

The OS is like a **resource manager**. It abstracts hardware and presents a simplified interface to users and applications.

### Diagram: OS Abstraction Layer

```
csharp
CopyEdit
[User Applications]
       |
[System Calls / APIs]
       |
[Operating System]
       |
[Hardware]
```

## System Structures

System structure defines how OS components are organized.

- **Monolithic Structure**: All OS components are tightly integrated (e.g., early UNIX).
- **Layered Structure**: OS is divided into layers; higher layers use services of lower ones.
- **Microkernel**: Only essential services (memory, process scheduling) in kernel.
- **Modular**: Uses loadable modules (e.g., Linux).
- **Hybrid**: Combines various models (e.g., Windows NT).

---

## System Calls

System calls are **interfaces between user programs and OS**. Users use system calls to request OS services.

### Categories:

- **Process Control**: `fork()`, `exit()`
- **File Management**: `open()`, `read()`, `write()`
- **Device Management**: `ioctl()`, `read()`, `write()`
- **Information Maintenance**: `getpid()`, `alarm()`
- **Communication**: `pipe()`, `shmget()`

### Diagram: System Call Flow

```
sql
CopyEdit
[User Process]
     |
  System Call
     |
[Operating System Kernel]
     |
  Executes Request
```

---

## Virtual Machines (VMs)

A **Virtual Machine** is an abstraction of a physical machine. The OS creates an environment that appears like a complete system.

### Types:

- **System VMs**: Provide a complete system (e.g., VMware, VirtualBox).
- **Process VMs**: Support a single process (e.g., Java Virtual Machine).

### □ Diagram: Virtual Machine Layer

```
diff
CopyEdit
+---------------------+
| Guest OS / Software |
+---------------------+
| Virtual Machine     |
+---------------------+
| Host OS / Hypervisor|
+---------------------+
| Physical Hardware   |
+---------------------+
```

---

## Process Concepts

A **process** is a program in execution. It includes the program code, program counter, stack, data section.

### States of a Process:

- **New**
- **Ready**
- **Running**
- **Waiting**
- **Terminated**

### Diagram: Process State Transition

```
sql
CopyEdit
          +---------+
          |  New    |
          +---------+
               |
               v
          +---------+        +----------+
          | Ready   |<------>| Waiting  |
          +---------+        +----------+
               |                  ^
               v                  |
          +---------+       I/O or event
          |Running  |--------------
          +---------+
               |
               v
          +-----------+
          | Terminated|
          +-----------+
```

---

## Threads

A **thread** is the smallest sequence of programmed instructions that can be managed independently by the scheduler.

- Threads within the same process share code, data, and resources.
- Each thread has its own stack, registers, and counter.

---

## Multithreading

**Multithreading** allows multiple threads to exist within the context of a single process. Useful for parallel execution and responsive applications.

### Benefits:

- Better resource utilization.
- Improved application performance.
- Lower overhead than multiple processes.

## Diagram: Single vs Multithreaded Process

```
lua
CopyEdit
Single-threaded:
+--------------+
|  Code        |
|  Data        |
|  Stack       |
+--------------+

Multithreaded:
+--------------+
|  Code        |
|  Data        |
| +----------+ |
| | Thread 1 | |
| |  Stack   | |
| +----------+ |
| | Thread 2 | |
| |  Stack   | |
| +----------+ |
+--------------+
```

# MODULE -2

## Process Scheduling

**Definition:**
Process scheduling is a key task of the operating system (OS) that determines which process runs at a given time. It's essential for multitasking and efficient CPU utilization.

### Types of Schedulers:

- **Long-term Scheduler**: Decides which processes are admitted to the system for processing.
- **Short-term Scheduler (CPU Scheduler)**: Decides which process gets the CPU next.
- **Medium-term Scheduler**: Swaps processes in and out of memory (used in systems with swapping).

### Scheduling Algorithms:

- **First-Come, First-Served (FCFS)**
- **Shortest Job Next (SJN)**
- **Priority Scheduling**
- **Round Robin (RR)**
- **Multilevel Queue Scheduling**

### Goals:

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

---

## Process Coordination

**Definition:**
Process coordination involves managing multiple processes so that they can operate correctly when accessing shared resources. It's critical in concurrent programming.

### Challenges:

- **Race Conditions**: Two or more processes access shared data concurrently and the outcome depends on the timing of their execution.
- **Critical Section Problem**: A critical section is a part of the code that accesses shared resources and must not be executed by more than one process at a time.

---

## Synchronization

**Definition:**
Synchronization ensures that two or more concurrent processes do not interfere with each other when sharing resources.

### Requirements:

1. **Mutual Exclusion**: Only one process can enter the critical section at a time.
2. **Progress**: If no process is in the critical section, and some processes wish to enter, one of them should be allowed to proceed.
3. **Bounded Waiting**: A bound must exist on the number of times other processes are allowed to enter their critical sections after a process has requested entry.

# Semaphores

**Definition:**
A semaphore is a synchronization tool that uses integer variables to control access to shared resources.

## Types:

- **Counting Semaphore**: Allows multiple processes to access a limited number of resources.
- **Binary Semaphore (Mutex)**: Only allows one process in the critical section (0 or 1 value).

## Operations:

- `wait()` or `P()` – Decrements the semaphore value. If the value becomes negative, the process is blocked.
- `signal()` or `V()` – Increments the semaphore value. If the value was negative, a blocked process is unblocked.

---

# Monitors

**Definition:**
A monitor is a high-level synchronization construct that allows safe access to shared data. It's a collection of procedures, variables, and data structures grouped together.

## Key Features:

- Only one process can be active inside the monitor at a time.
- Includes **condition variables** and operations:
  - `wait()`: Waits until a condition is true.
  - `signal()`: Wakes up a process waiting on a condition.

Monitors help abstract synchronization complexity by encapsulating shared variables and operations.

# Hardware Synchronization

**Definition:**
Hardware-based synchronization uses special machine-level instructions that are atomic (indivisible).

### Common Techniques:

- **Test-and-Set Instruction**
- **Compare-and-Swap (CAS)**
- **Exchange**

### Use:

These instructions ensure that checking and setting a lock happens atomically, preventing race conditions in low-level system programming.

---

## Deadlocks

**Definition:**
A deadlock is a situation where a set of processes are blocked because each process is waiting for a resource held by another, creating a cycle of dependencies.

### Necessary Conditions for Deadlock:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode.
2. **Hold and Wait**: A process holding at least one resource is waiting for more resources.
3. **No Preemption**: Resources cannot be forcibly taken away.
4. **Circular Wait**: A closed chain of processes exists, where each process holds a resource needed by the next.

---

## Methods for Handling Deadlocks

There are four primary strategies:

### 1. Deadlock Prevention

- Ensure that at least one of the necessary conditions for deadlock does not occur.
- Example: Prevent circular wait by ordering resource allocation.

## 2. Deadlock Avoidance

- Use algorithms to check if resource allocation leads to a safe state.
- Example: **Banker's Algorithm**

## 3. Deadlock Detection and Recovery

- Allow deadlocks to occur, detect them using algorithms, and then recover.
- **Recovery methods**: Kill one or more processes, or forcibly take resources back.

## 4. Ignore the Problem (Ostrich Algorithm)

- Often used in practice when deadlocks are rare.
- Example: Many operating systems (like UNIX) ignore deadlocks and assume they won't occur often.

---

## Summary Table:

| Topic | Focus | Key Tool/Concept |
|---|---|---|
| Process Scheduling | CPU Allocation | FCFS, RR, Priority etc. |
| Process Coordination | Proper interaction between processes | Critical Section Problem |
| Synchronization | Controlled access to resources | Mutual Exclusion |
| Semaphores | Signal-based resource control | `wait()`, `signal()` |
| Monitors | High-level synchronization | `wait()`, `signal()` |
| Hardware Synchronization | Atomic operations | Test-and-Set, CAS |
| Deadlocks | Circular resource waiting | Resource allocation graphs |
| Deadlock Handling | Methods to resolve or prevent deadlocks | Banker's algorithm, prevention |

MODULE-3

# Memory Management Strategies

Memory management refers to the way an operating system handles and organizes memory (RAM) usage for various processes. Two primary strategies are:

## A. Contiguous Memory Allocation

In contiguous allocation, each process is allocated a single contiguous block of memory.

**Types:**

- **Single Contiguous Allocation:** All processes share memory in a single block. Often used in early systems.
- **Partitioned Allocation:**
  - **Fixed Partitioning:** Memory is divided into fixed-size blocks or partitions.
  - **Variable Partitioning:** Memory is divided based on the process size.

**Advantages:**

- Simple and easy to implement.
- Low overhead.

**Disadvantages:**

- **Fragmentation:**
  - **Internal:** Wasted space within allocated memory.
  - **External:** Unused memory outside allocated blocks.
- Limits flexibility in handling varying process sizes.

---

## B. Non-Contiguous Memory Allocation

Memory is allocated in multiple non-adjacent blocks, allowing better utilization and flexibility.

**Techniques:**

- **Paging**
- **Segmentation**
- **Paged Segmentation**

**Advantages:**

- Reduces external fragmentation.
- Allows efficient use of memory.

**Disadvantages:**

- Overhead of maintaining page/segment tables.
- Can cause page table management complexity.

---

## Virtual Memory Management

Virtual memory allows execution of processes that may not be completely in RAM. It gives an illusion of a very large memory space.

### Key Concepts:

- Uses **secondary storage** (like hard disk) as an extension of RAM.
- Divides memory into **pages** and **frames**.
- Requires **address translation** using page tables.

### Benefits:

- Enables larger programs to run.
- Enhances multiprogramming.
- Reduces the need for contiguous memory.

---

## Demand Paging

Demand paging is a type of virtual memory where pages are only loaded into RAM when needed.

### Working:

- If a process accesses a page not in memory, a **page fault** occurs.
- The OS loads the page from disk into RAM.
- If RAM is full, a **replacement policy** selects a page to evict.

### Advantages:

- Reduces memory usage.
- Allows more processes to be resident in memory.

### Disadvantages:

- Page faults cause performance overhead.
- May lead to **thrashing** (excessive paging activity).

---

## Page Placement and Replacement Policies

### Page Placement:

Determines where in physical memory a new page will be loaded. Typically, any free frame can be used (random placement), but managing frame allocation efficiently is crucial.

### Page Replacement Policies:

When no free frame is available, the OS needs to **replace an existing page**. Common algorithms include:

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

- Removes the oldest page in memory.
- Easy to implement, but may remove frequently used pages.

#### 2. LRU (Least Recently Used)

- Removes the page that has not been used for the longest time.
- Closer to optimal, but expensive to implement without hardware support.

#### 3. Optimal Page Replacement

- Replaces the page that will not be used for the longest time in the future.
- Theoretically best, but not implementable in real systems.

#### 4. Clock Algorithm (Second-Chance)

- Circular queue of pages with reference bits.
- Gives a second chance to pages before replacing them.

## Summary Table

| Feature | Contiguous | Non-Contiguous |
|---|---|---|
| Memory Blocks | One continuous block | Multiple scattered blocks |
| Fragmentation | Internal & External | Mostly internal only |
| Flexibility | Low | High |
| Complexity | Low | High |
| Suitable for | Simple systems | Modern OS with large programs |

# MODULE-4

# File System – Basic Concepts

## What is a File System ?

A **file system** is a method used by operating systems to **store, organize, retrieve, and manage data** on storage devices like hard drives, SSDs, or USBs.

## Basic Terminology:

- **File**: Logical unit of storage (text, binary, executable).
- **Directory**: Collection of files, like a folder.
- **Path**: Location of a file/directory.
- **Metadata**: Information about a file (size, timestamps, permissions).

## Functions:

- Organizing files into directories.
- Naming and accessing files.
- Ensuring security and access rights.
- Managing space on storage devices.

# File System Design and Implementation

### Key Components:

1. **File Structure**: Defines how files are logically organized (e.g., flat, hierarchical).
2. **Allocation Methods**:
    - **Contiguous**: Files stored in a continuous block.
    - **Linked**: Files stored in blocks linked via pointers.
    - **Indexed**: Uses an index block to store addresses of file blocks.
3. **Directory Structures**:
    - Single-level
    - Two-level
    - Tree-structured
    - Acyclic graph
4. **File Access Methods**:
    - **Sequential**: Read/write in order.
    - **Direct (Random)**: Access any part of file directly.
5. **File System Mounting**: Making file systems available for use.
6. **File Protection**: Using permissions (read/write/execute).
7. **Crash Recovery**: Journaling or log-based recovery techniques.

---

## Case Study: Linux File Systems

### Common Linux File Systems:

- **ext2/ext3/ext4**: Traditional file systems.
- **Btrfs**: Advanced file system with features like snapshots.
- **XFS**: High-performance journaling file system.

### Features of ext4:

- Journaling for crash recovery.
- Delayed allocation for performance.
- Extents for efficient large file storage.
- Backward compatibility with ext2/ext3.

### Tools:

- `fsck`: File system check utility.
- `mount`/`umount`: Commands to mount/unmount file systems.
- `chmod`, `chown`, `ls`, `stat`: File permissions and metadata tools.

---

## Mass Storage Structure

### Types of Mass Storage:

- **HDD (Hard Disk Drive)**: Rotating platters, magnetic storage.
- **SSD (Solid-State Drive)**: No moving parts, faster, flash memory.
- **RAID**: Redundant Array of Independent Disks – improves performance and reliability.

### Key Concepts:

- **Sector**: Smallest physical storage unit.
- **Track**: Circular path on disk surface.
- **Cylinder**: Set of tracks aligned vertically.
- **Seek Time**: Time to move the disk arm.
- **Latency**: Delay waiting for the sector to rotate under the head.

---

## Disk Scheduling

### Purpose:

To determine the order in which disk I/O requests are serviced.

### Algorithms:

1. **FCFS (First Come First Serve)**
2. **SSTF (Shortest Seek Time First)**
3. **SCAN (Elevator Algorithm)**: Moves head in one direction then reverses.
4. **C-SCAN (Circular SCAN)**: Services in one direction, jumps back to start.
5. **LOOK/C-LOOK**: Variants that look ahead to last request rather than going to the end.

### Goal:

Reduce seek time and improve I/O throughput.

## Disk Management

### Responsibilities:

- Disk initialization and partitioning.
- Bad block management.
- File system formatting.
- Disk quotas to limit storage per user.
- Backup and recovery procedures.

### Partitioning:

Dividing a disk into multiple logical storage units.

### Volume Management:

Using Logical Volume Manager (LVM) to manage disk space dynamically.

---

## I/O Systems

### Purpose:

Interface between hardware devices and the operating system.

### I/O Techniques:

1. **Programmed I/O**: CPU controls all data transfer.
2. **Interrupt-driven I/O**: Devices interrupt CPU to get attention.
3. **DMA (Direct Memory Access)**: Transfers data directly between device and memory.

### I/O Components:

- **Device drivers**: Interface between hardware and OS.
- **I/O Controllers**: Manage device-specific operations.
- **Buffers**: Temporary storage to handle data transfer.

## System Protection and Security

### System Protection:

Ensures that processes do not interfere with each other.

- **Access Control**: Determines who can access what.
- **Memory Protection**: Ensures one process doesn't corrupt another.
- **CPU Protection**: Timer interrupts prevent monopolization.

### System Security:

Protects system from unauthorized access and attacks.

- **Authentication**: Verifies identity (passwords, biometrics).
- **Authorization**: Grants access based on identity.
- **Encryption**: Protects data during storage/transmission.
- **Firewall & Antivirus**: Defend against malware and intrusions.

---

## Summary Table

| Topic | Key Focus |
|---|---|
| **File System Basics** | Files, directories, metadata |
| **Design & Implementation** | Allocation, access, protection |
| **Linux File Systems** | ext4, Btrfs, tools |
| **Mass Storage** | HDD, SSD, RAID |
| **Disk Scheduling** | FCFS, SSTF, SCAN |
| **Disk Management** | Partitioning, formatting, LVM |
| **I/O Systems** | Programmed I/O, DMA, interrupts |
| **Protection & Security** | Access control, authentication, encryption |

# MODULE-5

## Distributed Systems

A **Distributed System** is a collection of independent computers that appear to the users of the system as a single coherent system. Each node (computer) runs its own local operating system and communicates with others through a network.

### Key Characteristics:

- **Transparency**: Users don't see the complexity (location, access, replication, etc.)
- **Scalability**: Easy to expand by adding more nodes.
- **Fault Tolerance**: If one node fails, others can continue the task.
- **Concurrency**: Multiple processes can run concurrently on different nodes.

### Examples:

- Cloud computing platforms (e.g., AWS, Google Cloud)
- Peer-to-peer networks (e.g., BitTorrent)
- Distributed databases (e.g., Cassandra)

---

## Distributed Operating Systems (DOS)

### Definition:

A **Distributed Operating System** is a software layer that manages a distributed system and makes it appear as a single system to users and applications.

### Responsibilities:

- **Process Management**: Scheduling and executing processes across nodes.
- **Resource Management**: Coordinating access to shared hardware/software resources.
- **Communication**: Facilitating message passing and data sharing between nodes.
- **Fault Handling**: Detecting and recovering from failures.

### Types:

- **Tightly Coupled Systems**: More synchronized, e.g., cluster systems.
- **Loosely Coupled Systems**: Independent systems communicating via network.

### Example DOS:

- Amoeba, Plan 9, Sprite, and some flavors of UNIX clusters.

## Distributed File Systems (DFS)

### Definition:

A **Distributed File System** enables files stored on multiple machines to be accessed and managed as if they are on a local disk, providing users and applications with a seamless view of file access.

### Features:

- **Location Transparency**: Files can be accessed without knowing their physical location.
- **Replication**: Files can be duplicated across nodes for fault tolerance.
- **Concurrency Control**: Multiple users can access or modify files concurrently.
- **Caching**: Files/data are cached locally to improve performance.

### Common DFS Examples:

- **NFS (Network File System)** by Sun Microsystems
- **AFS (Andrew File System)**
- **Google File System (GFS)**
- **HDFS (Hadoop Distributed File System)**

---

## Distributed Synchronization

### Definition:

**Distributed Synchronization** ensures that operations performed on multiple nodes happen in a coordinated fashion, preserving consistency and correctness.

### Key Concepts:

#### a. Clock Synchronization:

In distributed systems, each node has its own clock. To ensure coordinated actions, their clocks must be synchronized.

- **Techniques**: NTP (Network Time Protocol), Cristian's Algorithm, Berkeley Algorithm.

**b. Mutual Exclusion:**

Ensures that only one process accesses a critical section/resource at a time.

- **Algorithms**: Ricart-Agrawala, Lamport's Mutex, Token Ring

**c. Election Algorithms:**

Used when nodes need to agree on a coordinator.

- **Examples**: Bully Algorithm, Ring Algorithm

**d. Deadlock Handling:**

Deadlocks can happen when processes wait indefinitely for resources.

- **Solutions**: Detection and Recovery, Avoidance (Banker's Algorithm), Prevention

**e. Global State:**

Capturing a consistent global state is important for fault detection and recovery.

- **Example**: Chandy-Lamport Algorithm

---

## Summary Table

| Component | Purpose/Role |
|---|---|
| **Distributed System** | Enables multiple computers to work together as one system |
| **Distributed OS** | Manages the distributed resources and presents a unified system |
| **Distributed File System** | Provides file access across multiple nodes transparently |
| **Distributed Synchronization** | Coordinates timing and access among nodes to maintain consistency |

Thank You…