Operating Systems

Introduction

1. Operating System Operations

Imagine your computer as a busy city. The **Operating System (OS)** is like the government that manages everything: traffic (CPU scheduling), housing (memory), resources (files, printers), and security (protection).

Definition: An Operating System (OS) is system software that manages hardware and software resources, and provides services to users/programs.

What an OS does:

1. **Process Management** –
   - A *process* = a running program.
   - OS creates processes, schedules them for execution, and kills them when finished.
   - Example: When you open Chrome + Spotify + Word → OS decides how CPU time is shared.
2. **Memory Management** –
   - Every program needs memory to run.
   - OS allocates memory when a program starts and frees it when the program ends.
   - Prevents one program from disturbing another.
3. **Storage / File Management** –
   - Manages files and folders on your hard disk/SSD.
   - Example: Reading a song file from your disk when you open a music player.
4. **I/O Device Management** –
   - Coordinates devices like keyboard, mouse, printers.
   - Example: You press a key → OS translates it into an instruction for the application.
5. **Security & Protection** –
   - Prevents unauthorized users/programs from accessing system resources.
   - Example: Login password, file permissions in Linux.
6. **User Interaction** –
   - Provides Command Line Interface (CLI) like Linux terminal or Graphical User Interface (GUI) like Windows.

Summary: OS acts as a **manager** for all computer activities.

2. Operating System Services

The OS offers a set of services to make computer usage easier and efficient.

Important Services:

- **Program Execution** → Load a program into memory and run it.
- **I/O Operations** → Handles reading/writing from devices.
- **File Management** → Create, read, write, delete, organize files.
- **Communication** → Between processes (e.g., chat apps where processes exchange data).
- **Error Detection** → Detects hardware/software errors.
- **Resource Allocation** → Fairly distributes CPU, memory, I/O among processes.
- **Accounting** → Keeps usage records for billing/performance (common in servers).
- **Protection & Security** → Stops one program from harming another.

Summary: Services are like **features of OS** that make user's life easier.

3. System Calls

Question: *How do applications talk to the OS?*

- Apps cannot directly access hardware (for safety).
- They request services from the OS through **system calls**.

Example:

- When you run `printf("Hello");` in C, it eventually calls the **write() system call** to send text to the screen.
- In Linux:
  - `fork()` → create a new process.
  - `exec()` → run a new program.
  - `open()`, `read()`, `write()` → file operations.

Summary: System calls = **bridge** between programs and OS.

4. Types of System Calls

1. **Process Control** – Create, terminate, execute programs (`fork()`, `exit()`).
2. **File Management** – Open, close, read, write files.
3. **Device Management** – Request/release devices, read/write data.
4. **Information Maintenance** – Get time, process ID, system info.
5. **Communication** – Message passing between processes (`pipe()`, `send()`, `recv()`).

5. Operating System Structure

Different ways to design an OS:

1. **Simple structure** – No proper modules, everything in one place (e.g., MS-DOS).
2. **Layered structure** – Divided into layers (hardware at bottom, UI at top).
3. **Monolithic kernel** – All OS services in one large kernel (e.g., UNIX).
4. **Microkernel** – Only essential services in kernel; others in user space (e.g., Minix).
5. **Modular (Hybrid)** – Combines both (Linux, Windows).

Summary: Structure decides **how OS is organized internally**.

6. Operating System Types

1. **Batch OS** → Jobs collected and run without user interaction (old mainframes).
2. **Time-sharing OS** → CPU time shared among users/programs (UNIX).
3. **Distributed OS** → Runs across multiple computers but appears as one (cluster systems).
4. **Real-time OS (RTOS)** → Responds to tasks within strict deadlines.
    - *Hard RTOS*: deadlines cannot be missed (air traffic control).
    - *Soft RTOS*: occasional misses allowed (multimedia).
5. **Multiprogramming OS** → Multiple programs in memory, CPU switches when one waits.
6. **Multitasking OS** → User runs many tasks “at once” (Windows, Linux).

7. OS Components

- **Kernel** – Core part, directly interacts with hardware. Manages CPU, memory, devices.
- **Shell** – Interface for users to interact with OS (command line or GUI).
- **Command Interpreter** – Reads and executes user commands (`bash` in Linux, `cmd.exe` in Windows).

Summary: Kernel = heart, Shell = mediator, Command Interpreter = translator.

Process Management

1. Process

- A **process** is simply a *program in execution*.
- Example:
    - The `chrome.exe` file on disk = program.
    - When you double-click it and it runs = process.

Difference between Program & Process

- Program = Passive (stored instructions on disk).
- Process = Active (program loaded in memory + running).

2. Process State

A process changes its state while executing. Common states are:

1. **New** → Process is being created.
2. **Ready** → Process is waiting for CPU.
3. **Running** → Instructions are being executed.
4. **Waiting/Blocked** → Process is waiting for an event (like I/O).
5. **Terminated** → Process has finished execution.

Think of it like waiting in a queue for a bus:

- *Ready* = standing in line.
- *Running* = you got on the bus.
- *Waiting* = bus stopped for traffic light.
- *Terminated* = you reached destination.

3. Process Control Block (PCB)

- The **PCB** is a data structure that stores all information about a process.
- It's like the *ID card* of a process.
- Information stored in PCB:
    1. Process ID (PID)
    2. Process state (Ready/Running/Waiting)
    3. Program counter (next instruction address)
    4. CPU registers
    5. Memory info (base & limit registers, page tables)
    6. I/O info (files open, devices used)

4. Process Scheduling

- CPU can execute only **one process at a time** (per core).
- Scheduling = deciding **which process runs next**.
- Goal: maximize CPU usage, minimize waiting time.

5. Scheduling Queues

- Processes are kept in **queues** before execution.

1. **Job Queue** → All processes in the system.
2. **Ready Queue** → Processes waiting for CPU.
3. **Device Queue** → Waiting for I/O devices.

Example: Like waiting lines in a hospital:

- Registration queue, Doctor queue, Pharmacy queue.

6. Schedulers

- Decide which process moves from one queue to another.

1. **Long-term scheduler** → Selects processes from *job queue* → moves to ready queue.
    - Controls *degree of multiprogramming*.
2. **Short-term scheduler (CPU Scheduler)** → Selects which process from *ready queue* will run next.
    - Runs frequently (milliseconds).
3. **Medium-term scheduler** → Suspends or resumes processes to balance CPU load.

7. Context Switch

- When CPU switches from one process to another, it must **save the old process's state** and **load the new one's state**.
- Saved info = PCB.
- **Context switch is overhead** (no work done, just switching).

Example: Like a cricket umpire recording player stats when a batsman changes.

8. Scheduling Criteria

When designing scheduling algorithms, OS considers:

- **CPU utilization** → Keep CPU as busy as possible.
- **Throughput** → # of processes completed per unit time.
- **Turnaround time** → Time from submission → completion.
- **Waiting time** → Time spent in ready queue.
- **Response time** → Time to first response (important for interactive systems).
- **Fairness** → No process should starve forever.

9. Scheduling Algorithms

(a) First Come First Serve (FCFS)

- Processes executed in order of arrival.
- Simple, but **convoy effect** (slow process delays others).

(b) Shortest Job First (SJF)

- Process with shortest CPU burst runs first.
- Optimal for average waiting time.
- Problem: requires knowing job length in advance.

(c) Round Robin (RR)

- Each process gets a fixed time slice (quantum).
- Good for time-sharing OS.
- Example: CPU → P1 (10ms) → P2 (10ms) → P3 (10ms).

(d) Priority Scheduling

- Each process has a priority → highest priority runs first.
- Problem: starvation (low-priority may never run).
- Solution: **Aging** (gradually increase waiting process's priority).

(e) Multilevel Queue

- Different queues for different types of processes (e.g., system, interactive, batch).
- Each queue has its own scheduling algorithm.

(f) Multilevel Feedback Queue

- Processes can move between queues depending on behavior.
- Best balance between responsiveness and efficiency.

10. Operations on Processes

- **Creation** → New process made using `fork()` (in UNIX).
- **Termination** → Process ends by `exit()` or killed by another process.

11. Inter-Process Communication (IPC)

Processes need to communicate. Two ways:

1. **Direct / Indirect Communication**
   - Direct: send(P, message) → to process P.
   - Indirect: Using *mailboxes* or *ports*.
2. **Message Passing**
   - Send & receive messages via OS.
   - Slower but safer.
3. **Shared Memory**
   - Two processes share a common memory region.
   - Faster but needs synchronization.

Example:

- Message passing = writing a letter to a friend.
- Shared memory = sharing a notebook.

12. Multithreading & Thread Models

- A **thread** = lightweight process.
- Multiple threads inside one process share memory but run independently.
- Example: In Chrome, one thread handles UI, another loads web pages.

Thread Models:

1. **User-level threads**
   - Managed by user libraries, OS not aware.
   - Fast but can't utilize multiple CPUs well.
2. **Kernel-level threads**
   - Managed by OS.
   - Slower but supports true parallelism.
3. **Hybrid (Two-level)** → Combination.

13. Multicore Programming & Parallelism

- **Multicore processors** have multiple CPUs on one chip.
- OS must schedule tasks to exploit **parallel execution**.
- Types of parallelism:
  - **Data parallelism** – Same task on different data (e.g., matrix multiplication).
  - **Task parallelism** – Different tasks run at the same time (UI + background download).

Example: Modern games → One core handles graphics, another AI, another sound.

Memory Management Strategies

1. Swapping

- **Definition:** A process of temporarily moving an entire process from **main memory (RAM)** to **secondary storage (disk)** and bringing it back later when needed.
- **Purpose:** Allows CPU to execute other processes when RAM is full.
- **Steps:**
    1. Process in memory becomes idle or low priority → swapped out to disk.
    2. Later, swapped back into RAM for execution.
- **Advantage:** Increases degree of multiprogramming.
- **Disadvantage:** Slow due to disk I/O overhead.

```
+--------------------------+       +----------------------------+

|   Main Memory      | <--> |  Secondary Storage   |

|     [P1][P2][P3]       |        |  [Swapped Out P4]    |

+---------------------------+      +----------------------------+
```

2. Contiguous Memory Allocation

- **Definition:** Each process is allocated a single **continuous block** of memory.
- **Two parts of memory:**
    - **OS memory** (low part of RAM, reserved for OS).
    - **User memory** (remaining part, divided among processes).
- **Problem: Fragmentation** occurs.
    - **External fragmentation:** Free memory exists but not in a continuous block.
    - **Internal fragmentation:** Allocated block has unused space.

```
+-------------------------------+

| OS | P1 | P2 | Free | P3 |

+-------------------------------+
```

3. Paging

- **Definition:** Memory management scheme that eliminates external fragmentation by dividing:
    - **Logical memory (process)** → into **pages** (fixed size).
    - **Physical memory (RAM)** → into **frames** (same size as pages).

- **Working:**
    - Page Table maps **pages → frames**.
    - Example: Page 0 → Frame 5, Page 1 → Frame 9, etc.
- **Advantages:**
    - Removes external fragmentation.
    - Efficient memory utilization.
- **Disadvantage:** Overhead of page table management.

```
Logical Address Space       Physical Memory

+------------+------------+         +------------+------------+

|  Page 0   |  Page 1   | -----> | Frame 5 | Frame 9 |

|  Page 2   |  Page 3   | -----> | Frame 2 | Frame 7 |

+------------+------------+         +------------+------------+
```

4. Segmentation

- **Definition:** Divides logical memory into **segments** (variable size, e.g., code, stack, data).
- **Each segment has:**
    - Base address.
    - Limit (length).
- **Advantage:** Matches programmer's logical view of memory.
- **Disadvantage:** Leads to **external fragmentation**.

```
Logical Address Space
+------------------------+
| Segment 0: Code |
| Segment 1: Data  |
| Segment 2: Stack |
+-----------------------+

Physical Memory
+-----------------------+
| Code in Frame 3  |
| Data in Frame 5   |
| Stack in Frame 7  |
+-----------------------+
```

5. Virtual Memory Management

- **Definition:** Technique that allows execution of processes **not completely in memory**.
- Uses **secondary storage (disk)** as an extension of RAM.
- **Main concept:** Only required part of the process is loaded (on demand).

- **Advantage:** Can run large programs with less RAM.
- **Disadvantage:** Too much use of disk may slow down system.

6. Demand Paging

- **Definition:** Pages are loaded into memory **only when needed** (not in advance).
- If page not in memory → **Page Fault** occurs → OS loads page from disk.
- **Advantage:** Reduces memory wastage.
- **Disadvantage:** Frequent page faults → system slowdown.

CPU -> Virtual Address -> Page Table -> Physical Frame

(If not in memory → Page Fault → Load from Disk)

7. Page Replacement Algorithms

When page fault occurs and no free frame is available → **replacement needed**.

1. **FIFO (First-In First-Out):**
   - Oldest page in memory is replaced.
   - Simple but may not be optimal.

   Frames: [ ] → [7] → [7,0] → [7,0,1] → [0,1,2] → [1,2,3]

2. **LRU (Least Recently Used):**
   - Replace page that hasn't been used for the longest time.
   - More efficient, needs hardware support.
3. **Optimal Page Replacement:**
   - Replace the page that will not be used for the longest time in future.
   - Theoretical best but **not practical** (requires future knowledge).
4. **Clock Algorithm (Second-Chance):**
   - Pages arranged in circular list with a reference bit.
   - Gives "second chance" to pages before replacement.

   [Frame1*] → [Frame2] → [Frame3] → [Frame4]

   *Hand points to victim frame

8. Allocation of Frames

- **Equal allocation:** Every process gets same number of frames.
- **Proportional allocation:** Frames allocated based on process size.
- **Priority allocation:** Higher priority processes get more frames.

```
Process P1 → 3 Frames
Process P2 → 2 Frames
Process P3 → 1 Frame
```

9. Thrashing

- **Definition:** When CPU spends more time handling **page faults** than executing instructions.
- Occurs due to **too many processes with insufficient frames**.
- **Solution:** Reduce degree of multiprogramming or use **working set model**.

```
CPU -------------> Page Faults ↑↑

Execution ↓↓
```

10. Translation Lookaside Buffer (TLB) & Address Translation

- **TLB:** Special fast hardware cache that stores recent page table entries.
- **Process:**
  1. CPU generates logical address → checked in TLB.
  2. If found → frame number returned (TLB hit).
  3. If not found → check page table in RAM (TLB miss).
- **Advantage:** Speeds up memory access.

```
CPU Logical Address
        ↓
  +----------------+
  |      TLB       | → If Hit → Frame Number
  +----------------+
      ↓ Miss
  +----------------+
  |  Page Table  |
  +----------------+
```

11. Fragmentation

- **Internal Fragmentation:** Wasted space **inside allocated block**.
- **External Fragmentation:** Wasted space because free memory is scattered, not contiguous.

  Process requires 18KB → Allocated Block 20KB

  [Process 18KB | Unused 2KB]

12. Memory Allocation Strategies

1. **First-Fit:** Allocate the first block of free space large enough for process.
2. **Best-Fit:** Allocate the smallest block that fits the process (minimizes waste).
3. **Worst-Fit:** Allocate the largest available block (may leave large leftover space).

First Fit

Free Blocks: [10KB][20KB][15KB]

Request: 12KB → Allocated in 20KB

Best Fit

Free Blocks: [10KB][20KB][15KB]

Request: 12KB → Allocated in 15KB

Worst Fit

Free Blocks: [10KB][20KB][15KB]

Request: 12KB → Allocated in 20KB

Deadlocks and Mass-Storage Structure

1. System Model & Deadlock Characterization

- **Deadlock**: A situation where a set of processes are blocked because each process is holding a resource and waiting for another resource held by another process.
- **Necessary conditions for Deadlock (Coffman's conditions):**
    1. **Mutual Exclusion** – At least one resource must be held in a non-sharable mode.
    2. **Hold and Wait** – A process is holding at least one resource and waiting for others.
    3. **No Preemption** – Resources cannot be forcibly taken away.
    4. **Circular Wait** – A set of processes are waiting in a circular chain.

Diagram – Deadlock with 4 processes in circular wait:

P1 → R1 → P2 → R2 → P3 → R3 → P4 → R4 → P1

2. Methods for Handling Deadlocks

1. **Deadlock Prevention** – Ensure at least one Coffman condition cannot hold.
2. **Deadlock Avoidance** – Use algorithms (like Banker's Algorithm) to avoid unsafe states.
3. **Deadlock Detection and Recovery** – Allow deadlocks, detect them, and recover (terminate/restart processes).

3. Deadlock Prevention

- Break one of the four conditions:
    - **Mutual Exclusion**: Allow some resources to be shared.
    - **Hold and Wait**: Request all resources at once.
    - **No Preemption**: Take resources away when needed.
    - **Circular Wait**: Impose resource ordering.

4. Deadlock Avoidance – Banker's Algorithm

- Used when resource requests are known in advance.
- Ensures the system never enters an **unsafe state**.

Diagram – Banker's Algorithm workflow:

Request → Check Available → If (Need <= Available) → Pretend Allocation → Safe State?
  Yes → Grant
  No → Wait

5. Deadlock Detection & Recovery

- Build a **Wait-for Graph**:
    - Nodes = Processes
    - Edges = Process waiting for another
- Cycle = Deadlock.

Diagram – Wait-for Graph showing deadlock:

P1 → P2 → P3 → P1 (cycle exists = deadlock)

- **Recovery**:
    - Kill process(es).
    - Preempt resources.

6. Mass-Storage Structure

- **Secondary storage** (disks, SSDs) plays a major role in OS.

Disk Scheduling Algorithms

- **FCFS**: Serve requests in order of arrival.
- **SSTF**: Shortest Seek Time First → closer requests first.
- **SCAN**: Disk arm moves back and forth like an elevator.
- **C-SCAN**: Circular SCAN → moves in one direction only.
- **LOOK & C-LOOK**: Variants of SCAN, stop at last request.

Diagram – Disk head movement (SCAN vs C-SCAN):

```
SCAN:      |---->----|----<----|---->----|
C-SCAN:    |---->----| reset → |---->----|
```

Disk Management

- Formatting, partitioning, free space management, bad-block replacement.

RAID Levels

- **RAID 0** – Striping (fast, no redundancy).
- **RAID 1** – Mirroring (data duplicated).
- **RAID 5** – Block-level striping + parity.
- **RAID 10** – Combination of RAID 1 + 0.

Diagram – RAID:

```
RAID 0: A1 A2 A3 A4
RAID 1: A1 A1 A2 A2
RAID 5: A1 A2 P A3 (P=parity)
```

SSD vs HDD

- **HDD**: Magnetic disk, slower, mechanical parts.
- **SSD**: Flash memory, faster, no moving parts, durable.

Diagram – HDD vs SSD:

```
HDD: Spinning platters + read/write head
SSD: Flash chips + controller
```

Synchronization

The Critical Section Problem

- A **critical section** is a part of the program where shared resources (like variables, files, or databases) are accessed.
- Problem: If multiple processes access the critical section **simultaneously**, it may cause data inconsistency.

Requirements for a solution:

1. **Mutual Exclusion** – Only one process in the critical section at a time.
2. **Progress** – If no process is in CS, some waiting process should enter.
3. **Bounded Waiting** – No process should wait forever to enter CS.

Diagram – Critical Section

```
 -------------------------
| Entry Section       |
 -------------------------
| Critical Section    |  → Shared Resource Access
 -------------------------
|     Exit Section       |
 -------------------------
| Remainder Section |
 -----------------------------
```

Peterson's Solution

- Software-based algorithm for **two processes**.
- Uses two variables:
    - flag[i] = true → process i wants to enter CS
    - turn → indicates whose turn it is

➡ Guarantees **mutual exclusion** and **bounded waiting**.

P0	P1
flag[0] = true;	flag[1] = true;
turn = 1;	turn = 0;
while(flag[1] && turn==1);	while(flag[0] && turn==0);
---- Critical Section ----	---- Critical Section ----
flag[0] = false;	flag[1] = false;

Synchronization Hardware

- Some CPUs provide **atomic instructions** for synchronization:
    - **Test-and-Set (TAS)** – Locks a variable in one step.
    - **Compare-and-Swap (CAS)** – Compares memory value and swaps atomically.

Example: Spinlocks are implemented using these.

Shared Lock Variable → 0 (Unlocked), 1 (Locked)

```
┌─────────────────────┐
|   Process tries TSL   |
|   if Lock==0 → enter  |
|   else → wait (busy)  |
└─────────────────────┘
```

Semaphores

- An integer variable used for synchronization.
- Two operations:
  - **wait(P)** → decreases semaphore (block if < 0).
  - **signal(V)** → increases semaphore (wakes waiting process).

Types:

1. **Binary Semaphore (Mutex):** 0 or 1 (like a lock).
2. **Counting Semaphore:** Multiple resources controlled.

Diagram – Semaphore Working

```
        Resource
            |
      +----------+
      | Queue    |
      +----------+
   P() /        \ V()
```

Binary Semaphore:

```
S = 1 (Resource Free)
wait(S) → if S==1 → enter CS, set S=0
signal(S) → set S=1 (Release)
```

Counting Semaphore:

```
S = n (n resources available)
wait(S) → decrease S
signal(S) → increase S
```

Classic Synchronization Problems

1. **Producer-Consumer (Bounded Buffer Problem):**
   - Producer adds items, Consumer removes items.
   - Semaphores prevent buffer underflow/overflow.

- **Producer–Consumer Problem**

```
                  Buffer
|   Producer   |  ──────►  | Consumer |
                  ◄──────
  (Produce items, put in buffer)
  (Consumer takes items from buffer)
```

2. **Readers-Writers Problem:**
    - Multiple readers allowed, but only one writer at a time.

- **Readers–Writers Problem**

```
        ┌──────────────┐
        | Shared Data  |
        └──────────────┘
               |
               |
       ┌───────┴───────┐
       |               |
Readers (many)   Writers (only one)
```

3. **Dining Philosophers Problem:**
    - Five philosophers share five chopsticks → Deadlock possible.
    - Solved using semaphores or monitors.
4. **Dining Philosophers Problem**

```
[P0]—F0—[P1]
 |       |
F4      F1
 |       |
[P4]—F3—[P2]
     |
     F2
     |
    [P3]
(Pi = Philosopher, Fi = Fork)
```

Monitors

- High-level abstraction (like a class in programming).
- Contains **shared variables, procedures, and synchronization** automatically.
- Only **one process** can execute inside monitor at a time.

Monitor Example:

```
┌──────────────────────────────┐
|     monitor BoundedBuffer {  |
|     condition notFull, notEmpty |
|     procedures insert/remove |
| }                            |
└──────────────────────────────┘
```

Deadlock in Synchronization

- If processes wait on semaphores forever → deadlock occurs.
- Example: Dining philosophers all pick left chopstick → deadlock.

Spinlocks & Mutexes

- **Spinlock:** Process keeps "spinning" (busy waiting) until lock is free.
- **Mutex:** Like binary semaphore, but provides ownership (only locker can unlock).

| Thread 1 | ◄——► | Mutex | ◄——► | Thread 2 |

Condition Variables

- Used inside monitors.
- Two operations:
  - **wait()** – process waits on condition.
  - **signal()** – wake one waiting process.

Diagram – Condition Variable inside Monitor

```
Monitor {
  Shared data
  Procedure P() {
    if (condition not met) wait(cond);
    ...
    signal(cond);
  }
}
```

| Wait(cond, lock) |
| Signal(cond) |
| Broadcast(cond) |

File System Interface

1. Concept of a File

- A **file** is a collection of related information stored on secondary storage.
- Attributes: name, type, size, location, protection, timestamps.
- Operations: create, read, write, delete, open, close.

2. Access Methods

1. **Sequential Access**
   - Data is accessed in order (one after another).
   - Common for text files.

   Diagram:

   Record 1 → Record 2 → Record 3 → ... → Record N
2. **Direct (Random) Access**
   - Any block can be accessed directly using its index.
   - Used in databases.

Diagram:

Access Block 5 → Block 20 → Block 1 (in any order)

3. **Indexed Access**
   - An index is maintained to locate blocks quickly.

Diagram:

Index Table → Points to → Actual File Blocks

3. Directory Structure

1. **Single-Level Directory**
   - All files in one directory.
   - Simple but name conflicts occur.

Diagram:

```
Root
 ├── File1
 ├── File2
 └── File3
```

2. **Two-Level Directory**
   - Each user has their own directory.

Diagram:

```
Root
 ├── User1 → {File1, File2}
 ├── User2 → {File3, File4}
```

3. **Tree Structure**
   - Hierarchical, like modern OS.

Diagram:

```
Root
 ├── Home
 │    ├── User1
 │    │     └── File1
 │    └── User2
 │          └── File2
 └── System
      └── Config
```

4. **Acyclic Graph**
   - Allows sharing of subdirectories/files without cycles.
5. **General Graph**
   - More flexible, but cycles may create problems.

4. File System Mounting

- Process of attaching a file system to a directory tree at a mount point.

Diagram:

```
Root FS
 |
 └── /mnt → Mounted File System (USB / Disk / Network Drive)
```

5. File Sharing and Protection

- **Sharing**: Multiple users may access same file (read/write modes).
- **Protection**: Controlled using **Access Control Lists (ACLs)**, permissions (R, W, X).

Diagram:

```
File Permissions:  rwx  rw-  r--
          Owner Group Others
```

6. File Allocation Methods

1. **Contiguous Allocation**
    - Files stored in consecutive blocks.
    - Fast but causes external fragmentation.

    Diagram:

    ```
    File A → [Block 1, 2, 3, 4]
    File B → [Block 5, 6, 7]
    ```
2. **Linked Allocation**
    - Each file is a linked list of blocks.
    - No external fragmentation, but slow random access.

    Diagram:

    ```
    File A → Block 1 → Block 7 → Block 12 → Block 20
    ```
3. **Indexed Allocation**
    - Uses index block that contains pointers to file blocks.

    Diagram:

    ```
    Index Block → [5, 9, 13, 20] → Actual Data Blocks
    ```

7. Free Space Management

1. **Bit Vector**
    - Each bit represents a block (0 = free, 1 = allocated).

Diagram:

Blocks: [0 1 2 3 4 5 6 7]
BitMap: [1 0 0 1 1 0 0 1]

2. **Linked List**
   - Free blocks linked together.

Diagram:

Free List Head → Block 2 → Block 5 → Block 8

3. **Grouping**
   - Stores addresses of free blocks in a block.
4. **Counting**
   - Tracks free blocks as contiguous runs (block + count).

8. Journaling & Crash Recovery Basics

- **Journaling**: Keeps a log of file system changes before applying them, helps in crash recovery.
- **Crash Recovery**: Uses log (journal) to restore consistency.

Diagram:

Operation → Write to Journal → Commit → Apply to FS

File System Implementation & I/O Systems

1. File System Implementation

- **File Control Block (FCB):** Data structure to store metadata (file name, size, type, location, permissions).
- **Inode (Unix/Linux):** Index node storing file metadata + block addresses.

File Allocation Methods

- **Contiguous Allocation:**
  - Files stored in a sequence of blocks.
  - Fast access, ❌External fragmentation.
- **Linked Allocation:**
  - Each block points to the next block.
  - No external fragmentation, ❌Slow random access.
- **Indexed Allocation:**
  - Index block holds all block addresses.
  - Direct access, ❌Overhead of index blocks.

Free Space Management

- **Bit Vector:** 1 = free, 0 = allocated.
- **Linked List:** Free blocks linked together.
- **Grouping:** Store addresses of free blocks in a block.
- **Counting:** Store start + count of free blocks.

2. File System Mounting

- Process of attaching a file system to a directory structure.
- Example: `mount /dev/sda1 /mnt/data`.

3. File Sharing & Protection

- **Access Control Lists (ACLs):** Fine-grained permissions.
- **User/Group/Other (UGO) model:** Read, Write, Execute.
- **Encryption & Authentication:** Extra layer for protection.

4. Journaling & Crash Recovery

- **Journaling FS (e.g., ext4, NTFS):**
  - Log operations before applying them.
  - Ensures recovery after a crash.
- **Write-ahead logging:** Changes written to journal first.

5. I/O Systems

- **I/O Hardware:** Controllers, device drivers.
- **Device Drivers:** Interface between OS and hardware.

I/O Scheduling

- **FCFS (First Come First Serve):** Simple, fair.
- **SSTF (Shortest Seek Time First):** Nearest request first.
- **SCAN (Elevator Algorithm):** Head moves back & forth.
- **C-SCAN (Circular SCAN):** Head moves in one direction only.

Buffering & Caching

- **Buffering:** Temp storage to handle speed mismatch.
- **Caching:** Storing frequently accessed data in fast memory.
- **Spooling:** Queue of jobs for devices like printers.