Program vs. Process

- **Program** → Passive, stored on disk, a set of instructions (e.g., myprogram.exe).
- **Process** → Active, running instance in memory, executed by CPU.

  Example:

- notepad.exe (Program on disk)
- Opening Notepad creates a **process** (notepad.exe running in memory).

☑ *Processes switch between CPU-bound and I/O-bound states during execution.*

🔹 **CPU-Bound Process** 🔥

- Uses **more CPU** and does fewer I/O operations.
- Performance depends on **CPU speed**.
- Example: **Mathematical calculations, AI training.**

🔹 **I/O-Bound Process** 📂

- Spends **more time waiting** for I/O (disk, network).
- Performance depends on **I/O speed** (disk, network).
- Example: **File reading, database queries.**

Interrupts in OS

- **Interrupt: A signal to the CPU indicating an event needing attention.**
- **Purpose: Allows the OS to respond immediately to events (like I/O, errors, or timers).**
- **Types:**
  - **Hardware: From devices (e.g., keyboard).**
  - **Software: From programs (e.g., system calls).**
  - **Timer: For task scheduling.**
- **Process:**

1. **Interrupt occurs.**
2. **CPU pauses current task.**
3. **Executes Interrupt Service Routine (ISR).**

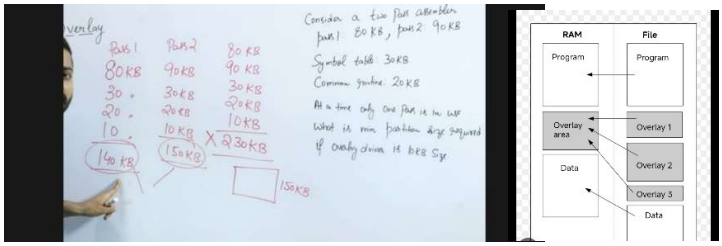
4. **Resumes previous task.**

👉 **Used for multitasking, better CPU usage, and real-time response.**

Overlay

Overlay in OS is a technique that runs large programs in limited memory by loading only one part (overlay) at a time. When another part is needed, It replaces the current one, saving memory. Modern systems use virtual memory instead.

Used in embedded device like washing machine if clean on clean code put in os and if dry on only dry mode code is put in os clean js 150 and dry is 140 so our max memory will be used



is 150 kb

Operating System

An **Operating System (OS)** is system software that acts as an interface between computer hardware and users.

Major Goals of an OS:

1. **User Interface** – Provides GUI (Graphical User Interface) for user interaction.
2. **Multitasking & Concurrency** – Allows multiple processes to run simultaneously.
3. **Resource Management** – Efficiently manages CPU, memory, storage, and I/O devices.
4. **Memory Management** – Allocates and deallocates memory as needed.
5. **Process Management** – Handles process scheduling, execution, and termination.
6. **File System Management** –stores, Organizes and retrieves files efficiently.
7. **Device Management – Controls and coordinates peripheral devices (printers, keyboards, etc.).**
8. Networking & Communication – Facilitates data exchange between connected systems.
9. **Security & Protection** – Prevents unauthorized access and ensures data integrity.

   Authentication vs Authorization

   - **Authentication** → Verifies **who** you are (e.g., login with a password).
   - **Authorization** → Determines **what** you can access (e.g., file permissions).
   - **Authentication comes first**, then **authorization**.

Pre-emptive vs non-pre-emptive Scheduling

1. Pre-emptive Scheduling:

- The CPU **can be taken** from a running process and given to another higher-priority process.
- Ensures better CPU utilization
- Used in **multi-tasking systems**.

Example: Shortest Remaining Time First (SRTF).

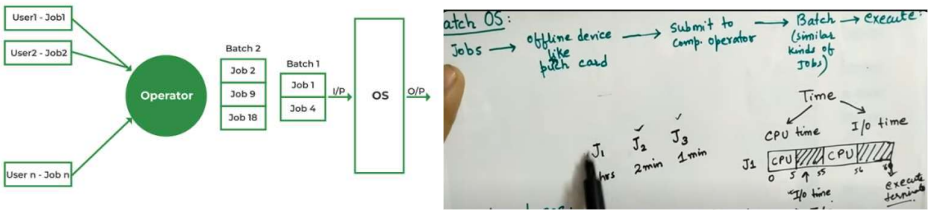
2. Non-Pre-emptive Scheduling:

- Once a process starts execution, it **cannot be interrupted** until it completes or switches to waiting state.
- Simpler but may lead to longer wait times.
- Used in **batch systems**.

Example: First Come First Serve (FCFS)

Types of Operating Systems

1. **Batch OS** – It is non-pre-emptive. Processes similar jobs in batches without direct user interaction.

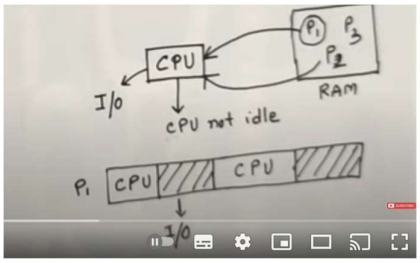


- ❖ **Single Batch System** → Runs **one job at a time** (slow, inefficient).

❖ **Multiprogramming Batch System** → Runs **multiple jobs together** by switching between them (faster, efficient).

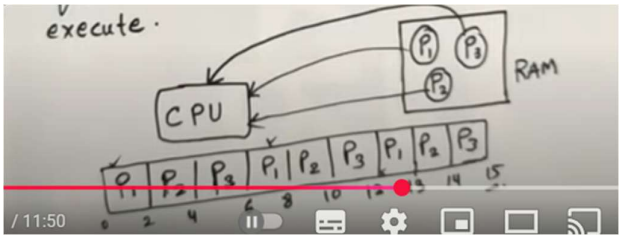
- **Pros:**
  - efficient for repeated tasks.
- **Cons:**
  - Inefficient CPU usage (CPU is being idle at the time of I/0 on which j2 and j3 can be performed but CPU is idle),
  - high waiting time (Starvation j2 and j3 wait until j1 finished which will take 1 hr),
  - not user interactive (because at time of execution user is not present.)
- **Example:** bank statements.

2. **Multiprogramming OS** – It is non-pre-emptive. Multiple programs reside in memory, improving resource use. For the time a process does I/0, the CPU can start the execution of other process. (if p1 is doing I/0 then p2 can be processed and CPU will not be idle like bath OS)



- **Pros:** Better CPU utilization
- **Cons:** Complex scheduling, Responsive time is max for the last process, starvation (if p1 will take 1hr and don't do any I/0 operation but p2 will take 1 mint then p2 will wait until p1 is finished)

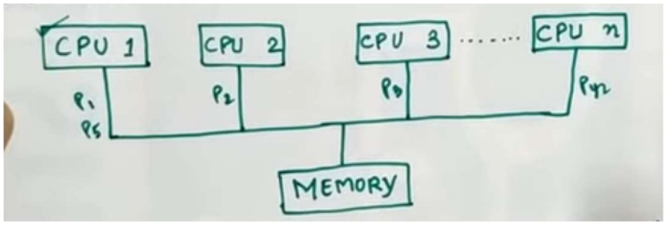
3. **Multi-tasking/Time-Sharing OS** – it is pre-emptive. Allocates CPU time (quantum) to multiple tasks in a round-robin manner for which a process is meant to execute.



(P1,p2,p3 each will take 5 mint, os allocate 2 min so look above 2 2 1 in this order each process execute parallelly therefore at 13 14 15 each take 1 mint but befpre this it take 2 mint)

- **Pros:** Equal CPU time
- **Cons:** Security risks

4. **Real-Time OS (RTOS)** – tasks are processed within strict time limits.
   - **Types:**
     - **Hard RTOS:** No delays allowed because minor delay may cause major loss (e.g., airbag systems, missile, satellite).
     - **Soft RTOS:** Minor delays acceptable (e.g., video streaming).
   - **Pros:** Fast response, high reliability.
   - **Cons:** Complex design
5. **Multiprocessing OS** – Uses multiple CPUs for execution. There is more than one processor present in the system which can execute more than one process at the same time (while in batch multitask and multiprogramming context switching happen means if one stop other run not all process works together)



   - **Pros:** Higher throughput (as max process done), fault tolerance/reliable (if CPU 1 fail then p1 and p5 will be executed by other CPU),fast processing because of multiple CPU
   - **Cons:** Expensive, Large main memory required
6. Parallel OS: A **Parallel OS** is a special type of multiprocessing OS designed for high-performance computing, where multiple processors work together on the same task (like supercomputers and distributed systems).

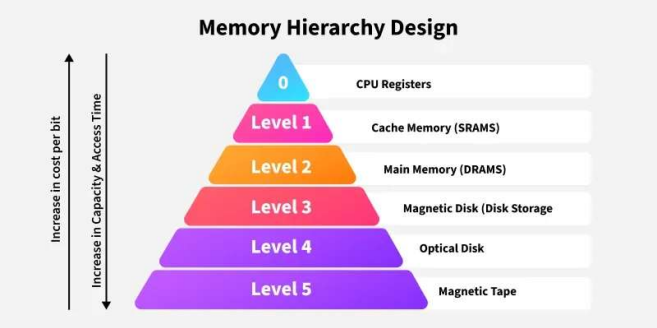
- A **Multiprocessing OS** runs multiple programs using multiple CPUs.
- A **Parallel OS** uses multiple CPUs to work on the same task faster.

7. **Distributed OS** – Uses interconnected computers for task execution.it follow client server architecture

- **Pros:** Resource sharing
- **Cons:** Network dependency
- **Example:** Windows Server

8. **Mobile OS** – Designed for mobile devices.**Examples:** Android, iOS.
9. **Embedded OS**: Designed for specific devices like ATMs, microwaves, and medical equipment.
10. **Personal OS:** Used in personal computers (PCs) e.g., Windows, macOS, Linux.

Memory Hierarchy Design and Characteristics



It arranges memory into multiple levels based on access time, with the fastest and smallest memory at the top and the slowest and largest at the bottom.

Why Memory Hierarchy is Needed?

- **It optimize access speed, cost, and capacity.**
- Faster memory (cache, RAM) is small and expensive, while slower memory (HDD, tape) is larger and cheaper.
- Ensures efficient data access and retrieval.

Types of Memory Hierarchy

1. **External Memory (Secondary Storage):** Includes HDDs, SSDs, optical disks, and magnetic tapes. Accessed via I/O modules.
2. **Internal Memory (Primary Storage):** Includes registers, cache, and main memory, directly accessible by the CPU.

Memory Hierarchy Levels

1. **Registers:**

2. **Cache Memory:** Faster than RAM, stores frequently accessed data to reduce access time. Between cpu and ram Registers hold data that the CPU is actively processing, whereas cache memory holds frequently accessed data to speed up processing.
3. **Main Memory (RAM):** Stores active programs and data.
    - **SRAM:** Fast, uses flip-flops, used in cache.
    - **DRAM:** Slower, uses capacitors, requires refreshing.
4. **Secondary Storage:** HDDs and SSDs for long-term data storage.
5. **Magnetic Disk:** Circular storage device used for frequent data operations.

6. **Magnetic Tape:** Slower, used for backups.

Characteristics of Memory Hierarchy

- **Capacity:** Increases from top (registers) to bottom (tape).
- **Access Time:** Increases as we move down the hierarchy.
- **Cost per Bit:** Higher for faster memory (registers > RAM > HDD).

Advantages of Memory Hierarchy

✓ **Cost Efficiency:** Combines expensive fast memory with cheaper large storage.

Disadvantages of Memory Hierarchy

✕ **Complex Design:** Managing different memory types is challenging.
✕ **High Cost:** Fast memory like cache and registers is expensive.

Memory Management in multiprogramming Operating Systems

OS occupies part of memory; the rest is shared among processes.

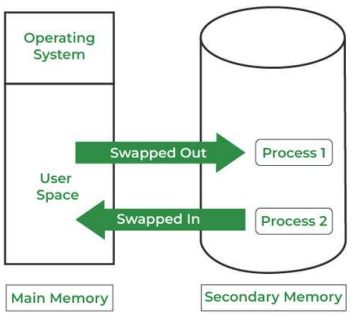
Maximizes processes in RAM for better CPU utilization.

Handles memory allocation, deallocation, and swapping.

Uses techniques like partitioning, paging, and segmentation.

Swapping

- Temporarily moves processes from main memory to secondary storage to optimize memory use and process execution.



Swapping (also called roll-out/roll-in) is a memory management technique where a lower-priority process is temporarily moved to disk to free up space for a higher-priority process.

Once the higher-priority process completes, the lower-priority process is swapped back into memory and resumes execution.

- Swap-in (Roll-in): Moving a process from disk (secondary storage) back into main memory (RAM) for execution.
- Swap-out (Roll-out): Moving a process from main memory (RAM) to disk (secondary storage) to free up space.

Advantages:

✓ Improves CPU efficiency by allowing multiple processes to run.

✓ Helps manage limited RAM by using swap space.

✓ Enables execution of large programs on systems with low memory.

Disadvantages:

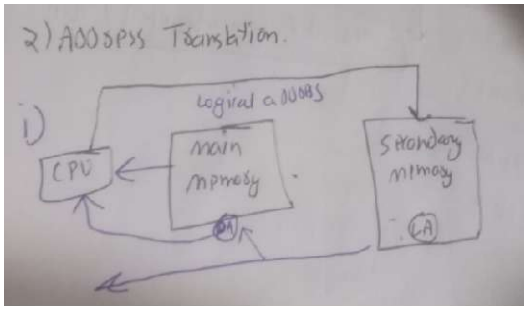
✕ Frequent swapping can slow down performance.

✕ Increases page faults due to constant data movement.

✕ Risk of data loss if power failure occurs before swapping completes.

Swapping ensures efficient memory management but can impact system speed if overused.

Logical and Physical Addresses in Operating Systems

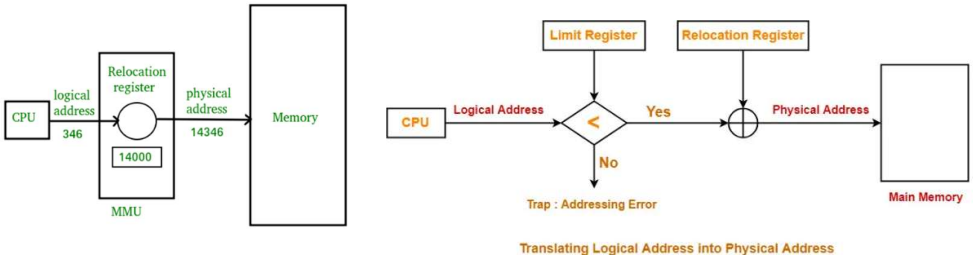


Memory Management Unit (MMU)

- A hardware component that **translates logical addresses** into physical addresses.
- Uses a **page table** to map logical addresses to corresponding memory locations.
- Found inside the **CPU** or as a separate chip.
- **Memory Management Unit (MMU)** uses two registers:
    - **Base Register (Relocation Register)** – Stores the **starting physical address** of the process.
    - **Limit Register** – Specifies the **size of the allocated memory** for the process.

Key Differences

Parameter	Logical Address	Physical Address
Generated by	CPU	MMU
Location	Virtual memory	Main memory (RAM)
Visibility	Visible to the user	Not directly visible
Access	Used by programs	Only accessed via translation
Editable	Can change	Fixed for a given execution



Translating Logical Address into Physical Address

Address Translation in Contiguous Allocation

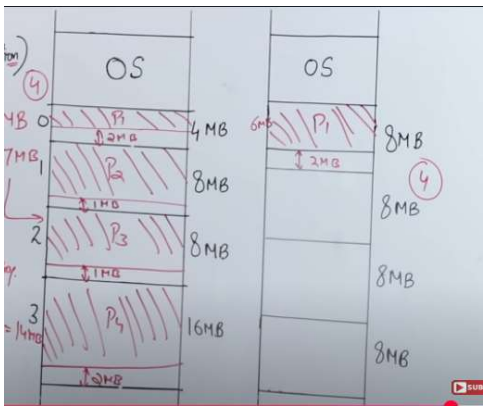
1. **CPU generates a logical address**.

2. **Limit Register checks** whether the address is within the process's allocated memory range.
3. **If valid**, the **Base Register's value is added** to the logical address to get the physical address.
4. **If invalid**, the OS **terminates the process** due to an out-of-bounds memory access error.

Memory Allocation

1. **Contiguous Allocation**: Assigns a continuous memory block to a process.

   ## Fixed Partitioning: The number of partitions is fixed, and each partition can have the same or different sizes. Processes can be placed in any available partition.



Problems of Fixed Partitioning:

- External Fragmentation: **Internal fragmentation:** Wasted space inside allocated memory blocks.*(E.g., if a 2MB process is placed in a 4MB partition, the remaining 2MB is wasted.)*
- **External fragmentation:** Non-contiguous free memory blocks prevent allocation*(E.g., if there are free blocks of 2MB, 1MB, 1MB, and 2MB, totalling 6MB, a new process cannot use it since processes must be loaded as a whole, not in parts.becuase spanning is not allowed)*

- **Process size limitation:** A process larger than the largest partition cannot be loaded (e.g., a 17MB process cannot fit in a 16MB partition).
- **Limited multiprogramming:** The number of processes in RAM is restricted by the number of partitions.

Dynamic/Variable Partitioning:

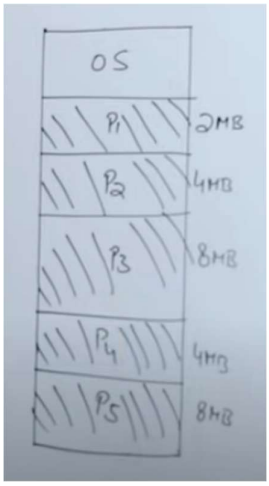
Allocates memory based on process requirements, allowing flexible memory usage.

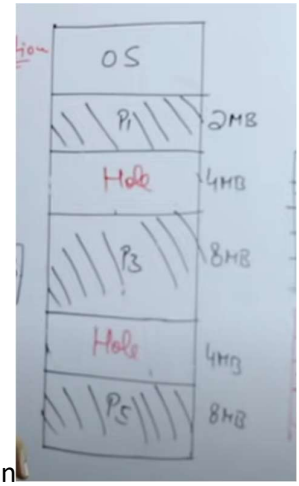
Advantages:

✅ **No internal fragmentation** (memory is allocated exactly as needed).
✅ **No limitation on process size** (as long as enough memory is available).
✅ **No limitation on the number of processes** (processes are allocated dynamically).

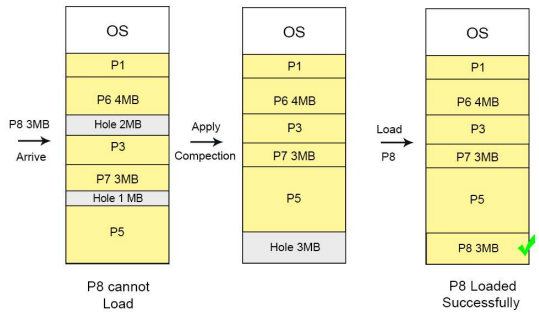
Problems:

❌ **External fragmentation** (*when processes finish execution, they leave holes in memory*).

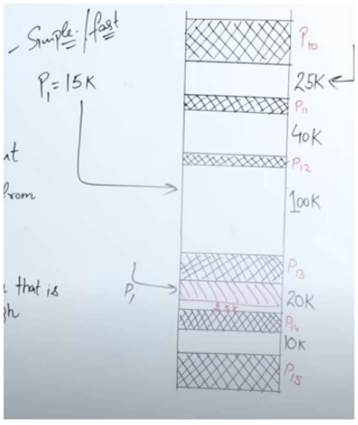


when process done execution


- *(E.g., if an 8MB hole exists but is split into smaller non-contiguous blocks, a 7MB process may not fit.)but* compaction *can be used we can separate process and hole differently*

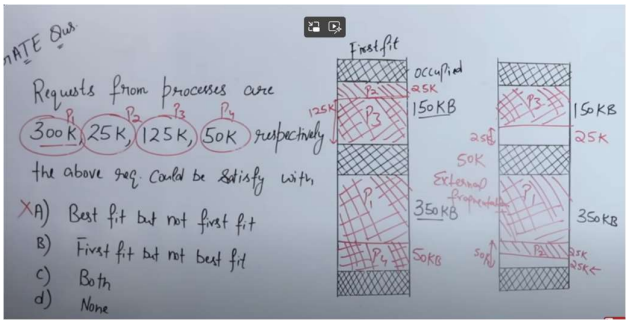


- ❌ **Complex allocation and deallocation** (*managing free space dynamically increases overhead*).
    - **Allocation Strategies for dynamic partition**:



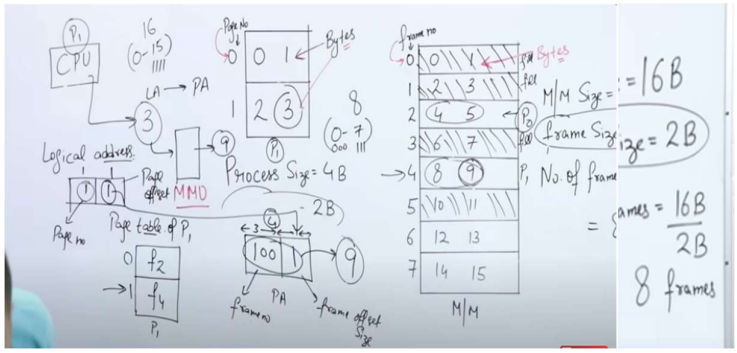
p1 to be placed in ram

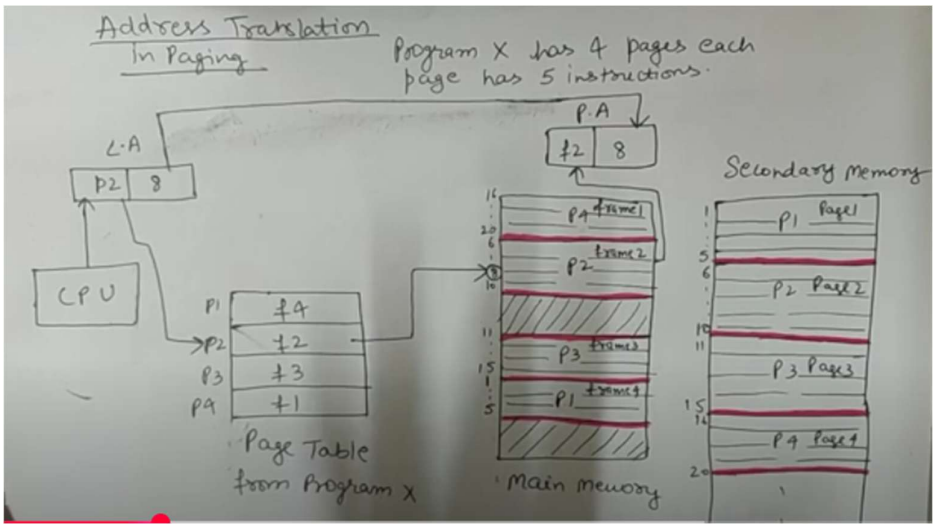
        - **First Fit**: Uses the first available block
        - **Best Fit**: Allocates the smallest sufficient block.
        - **Worst Fit**: Allocates the largest block to avoid fragmentation.
        - **Next Fit**: Searches for the next available block after the last allocation.



2. **Non-Contiguous Allocation**: Divides processes into smaller parts stored separately.
    - # Paging:

      ## It is Fixed-size memory blocks (pages).





Paging in Memory Management

- Paging is a memory management technique that allows a process's physical address space to be non-contiguous.
- It prevents external fragmentation and eliminates the need for compaction.
- Memory is divided into **fixed-size blocks**:
    - **Frames** (physical memory)
    - **Pages** (logical memory)
    - Page size = Frame size.

Logical Address (CPU Generated)

- **Page number(p): Number of bits required to represent the pages in Logical Address Space or Page number**
- Page offset(d): Number of bits required to represent a particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address (After Translation)

- **Frame Number(f): Number of bits required to represent the frame of Physical Address Space or Frame number frame**
- Frame Offset(d): Number of bits required to represent a particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

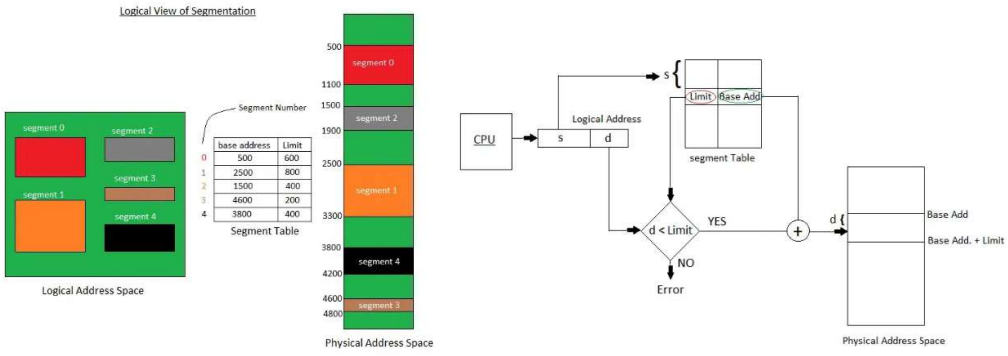
Page Table

- A **data structure** that maps **pages to frames**.

Segmentation:

- A process is divided into **segments** of varying sizes,(unlike paging, where all pages are of a fixed size.)

- Physical address= base address + offset



- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Segment Limit:** Also known as segment offset. It specifies the length of the segment.

Segment Table–

A **Segment Table** is used in **segmentation** to map **logical addresses** (segment number, offset) to **physical addresses** (memory location

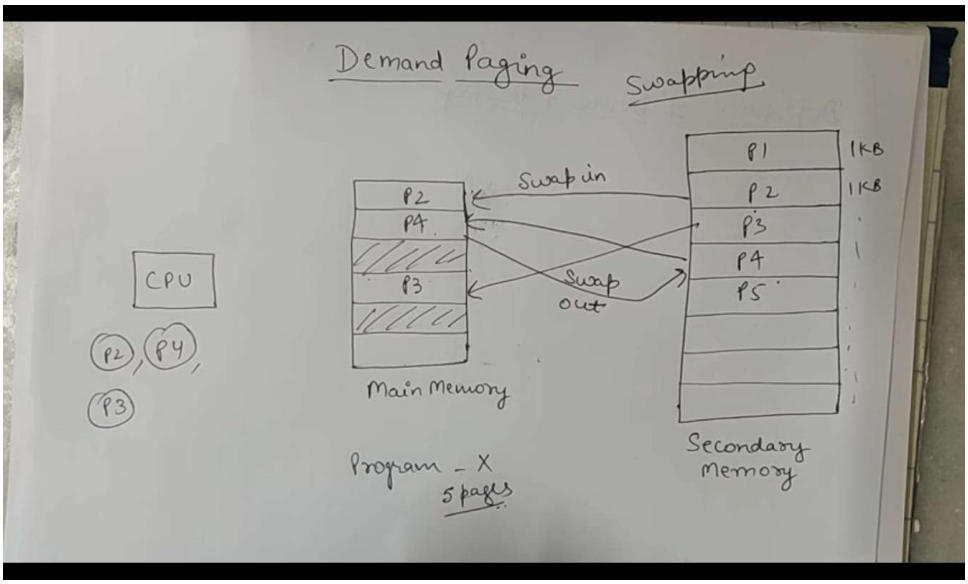
The address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the position of data within a segment.

Demand Paging

Demand paging is a memory management scheme in which programs reside in secondary memory, and pages are loaded into main memory only on demand, not in advance.

It is a type of swapping in which pages are copied from secondary memory to main memory when they are needed.

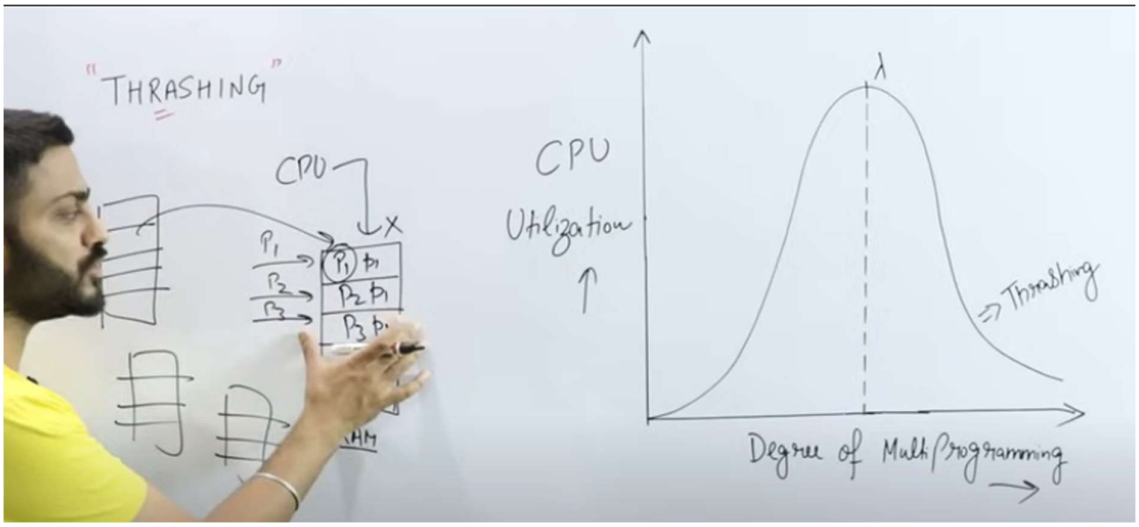


Page Fault/ page miss and Page Hit

- Page fault: When we want to load the page on the memory, and the page is not already on memory, then it is called a page fault. The page fault is also called page miss.
- Page hit: When we want to load the page on the memory, and the page is already available on memory, then it is called page hit.

Thrashing

Thrashing happens when the system spends more time swapping pages than executing processes, leading to performance degradation.



When only parts of processes (like the first parts of P1, P2, and P3) are loaded into RAM to maximize multiprogramming, the memory quickly fills up. If a process (P2 in this case) needs a part that isn't in RAM, a page fault occurs. Fetching the missing part from secondary storage takes time, causing the system to spend excessive time servicing page faults rather than performing useful work.

Causes of Thrashing:

✓ **High Degree of Multiprogramming** → Too many processes in memory.
✓ **Lack of Frames** → Not enough memory allocated per process.
✓ **Bad Page Replacement Policy** → Frequent evictions increase page faults.

Page Replacement Algorithm

Page replacement algorithms are used in operating systems to decide which memory pages to replace when a new page needs to be loaded into memory, but no free frames are available.

This is crucial for managing virtual memory efficiently.

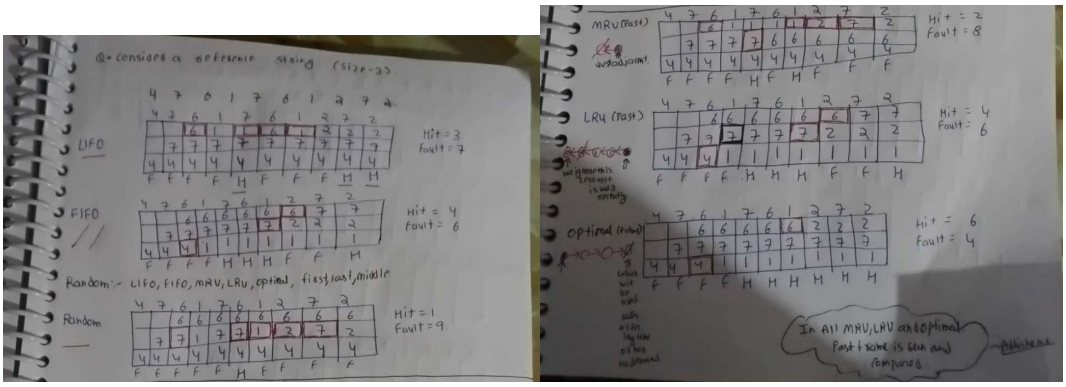
Here are some common page replacement algorithms:

Page Replacement Algorithms:

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

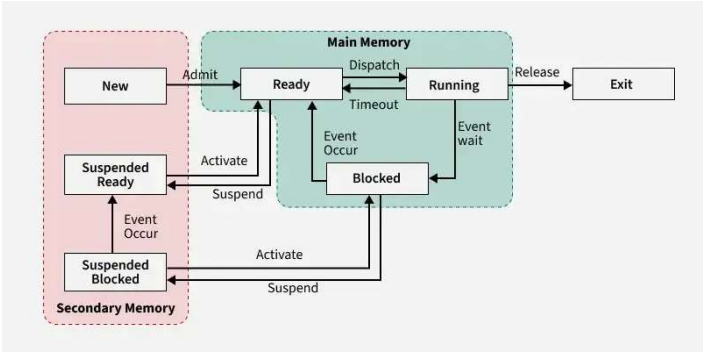
- Replaces the oldest page in memory.
- Uses a queue to track the order of arrival.
- Can suffer from **Belady's Anomaly** (more frames can lead to more page faults).

2. **LIFO (Last In, First Out)**
    - The most recently loaded page is replaced first.
    - Similar to a stack (push new pages, pop the most recent one).
    - Can be inefficient for some workloads.
3. **Random Replacement**
    - Selects a page randomly for replacement i.e last middle fast lru mru optical lifo fifo.
    - Simple but not optimal in performance.
4. **MRU (Most Recently Used)**
    - Replaces the page that was used **most recently**.
    - Useful when the most recently used pages are least likely to be needed again.
    - Opposite of LRU.
5. **LRU (Least Recently Used)**
    - Replaces the **least recently used** page.
    - Assumes that pages used recently will be needed again soon.
6. **Optimal Page Replacement (OPT or MIN)**
    - Replaces the page that **will not be used for the longest time** in the future.
    - Requires knowledge of future page references (not practical in real-world systems).



Unit 2

A process state represents the current status of a process in the operating system. A process goes through different states from creation to termination. (Pre-emptive scheduling)



Process States Explanation:

Active States (Main Memory Involvement)

1. **New:**
    - a program is being converted into a process
2. **Ready:**
    - The process is loaded into main memory and is ready to execute.
    - It waits for the CPU to become available.
3. **Running:**
    - The process is currently being executed by the CPU
    - Only one process can be in this state at a time (for a single-processor system).
4. **Blocked (Waiting):**

- The process is waiting for an event (e.g., I/O operation).
- It cannot proceed until the event occurs.

5. **Exit (Terminated):**
    - The process has finished execution and is removed from the system.

Suspended States (Secondary Memory Involvement)

6. **Suspended Ready:**
    - The process is ready but moved to secondary storage due to memory constraints.
    - It will return to the ready state when memory is available.
7. **Suspended Blocked:**
    - The process is blocked and moved to secondary storage.
    - It remains here until the event occurs and memory is available.

Scheduling Levels in Operating Systems

Scheduling levels refer to different stages in the **process scheduling hierarchy**. These levels define how processes are managed and which scheduler is responsible for a process at different times during its life cycle.

Process scheduling is essential in multiprogramming operating systems to manage CPU allocation efficiently. There are three types of schedulers:

1. **Long-Term Scheduler (Job Scheduler)**
    - Selects and loads jobs from secondary storage (e.g., disk) into main memory.
    - Controls the degree of multiprogramming by determining how many processes enter the system.
    - Prioritizes a balanced mix of CPU-bound and I/O-bound processes.
2. **Short-Term Scheduler (CPU Scheduler)**
    - Selects processes from the ready queue in main memory.
    - Runs frequently (high-speed) and is triggered by events like clock ticks or I/O interrupts.
    - Moves processes from the ready state to running.
3. **Medium-Term Scheduler (Process Swapping Scheduler)**

- Manages suspended processes by swapping them in and out of memory.
- Helps optimize memory usage and maintains the degree of multiprogramming.
- Reintroduces processes into memory once conditions allow execution to resume.

Speed: Sts>mts>lts

- **Speed:** Short-term is the fastest, medium-term is intermediate, and long-term is the slowest.

Process Management Attributes

A process in an operating system has several attributes that help manage and track its execution. These include:

1. **Process ID (PID)** – A unique number assigned to each process. Helps the OS distinguish between multiple running processes.
2. **Process State** – The current status of the process (e.g., Ready, Running, Waiting).
3. **Program Counter** – Holds the address of the next instruction to be executed.
4. **Priority** – Determines the scheduling order; higher priority processes run first.
5. **General-Purpose Registers** – Store temporary data during execution. Few example are:-

- **Accumulator (AX, EAX, RAX) → Stores arithmetic & logic results.**
- **Base Register (BX, EBX, RBX) → Holds memory addresses.**
- **Stack Pointer (SP, ESP, RSP) → Points to the top of the stack.**

6. **List of Open Files** – Tracks files currently being accessed by the process and ensures proper file handling
7. **List of Open Devices** – Keeps record of hardware devices being used (e.g., printers, disks).

Inter-Process Communication (IPC)

Inter-Process Communication (IPC) allows processes to exchange data and synchronize their execution in an operating system.

Types of IPC:

1. **Independent Processes**

- Do not share data with other processes.
- Execution is unaffected by other processes and don’t affect other processes

2. **Cooperating Processes**
    - Share data, variable, code and resources with other processes.
    - Execution is affected by other processes and it affect other processes

---

IPC Communication Types:

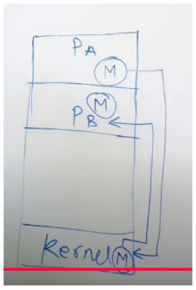
1. **Unicast**
    - One-to-one communication.
    - Example: Sending a message from one process to another.
2. **Multicast**
    - One-to-many communication.
    - Example: A process sends data to multiple processes simultaneously.

IPC Models:

1. **Message Passing Model**
    - Processes communicate by sending and receiving messages.
    - Slower but useful for distributed systems.

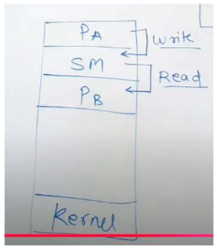
Process Communication:

- **Process A (Pa)** sends a message to the kernel.
- The **kernel** forwards the message to **Process B (Pb)**.
- **Process B (Pb)** receives and processes the message.

- Example: Message Queues. 

2. **Shared Memory Model**
    - Processes communicate by sharing a common memory region.
    - Requires synchronization
    - Faster but needs careful memory management.

Data Flow:

3. **Process A (Pa)** writes data to shared memory.
4. **Process B (Pb)** reads data from shared memory.
    - Example: POSIX shared memory
    - 

Ipc mechanism

- Semaphore : used to synchronised process so that they can't access critical section simultaneously
- Message queue : process send and receive message between process
- Shared memory: process share memory region to exchange data

Process vs Thread

Aspect	**Process**	**Thread**
Definition	An independent program in execution/heavy weight	A smaller part of a process/light weight
Memory	Has its own separate address space.	Shares memory with other threads in the same process.

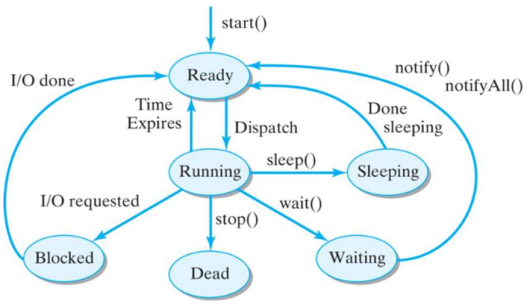
Aspect	Process	Thread
Creation & context Switching	Slower to create, terminate, and switch.	Faster to create, terminate, and switch.
Blocking	A blocked process does not affect others.	A blocked thread may affect other threads in the same process.
Communication	Less efficient because it uses IPC (message queues, pipes, or shared memory).	More efficient because it uses shared memory communication within the same process.

Context switching is when the CPU stops running one process or thread and switches to another. It saves the current process's state so it can continue later without losing progress.

Context switching occurs in these main scenarios:

- **When a process is busy in I/O** – If a process is waiting for input/output (e.g., reading a file, waiting for user input), the CPU switches to another process instead of staying idle.
- **When a higher-priority process arrives** – If a process with higher priority needs the CPU, the current process is paused, and the CPU switches to the high-priority task.
- **When time-slicing happens (Multitasking)** – In time-sharing systems, each process gets a fixed time to execute. When its time is up, the CPU switches to another process.
- **When an interrupt occurs** – If an external event (e.g., a keyboard press, network request) requires immediate attention, the CPU switches to handle it.

Life cycle of Thread



Ready – The thread is created and ready to run but is waiting for CPU time. (Triggered by start())

Running – The thread is currently executing in the CPU. (Triggered by Dispatch)

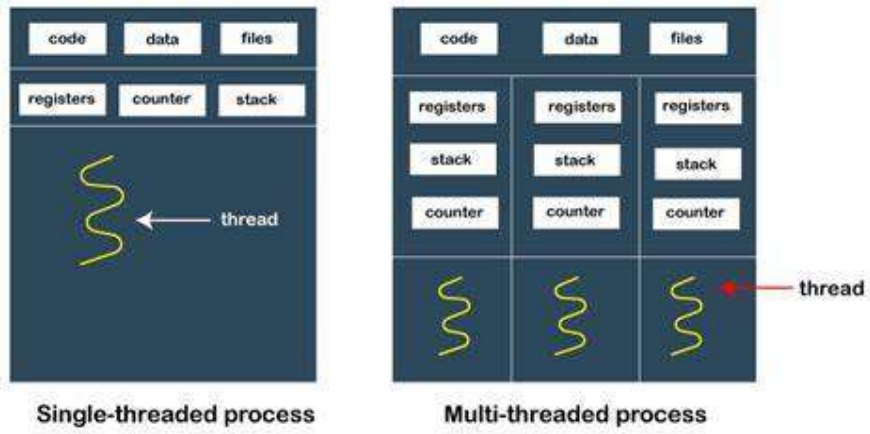
Dead – The thread has completed execution or was stopped. (Triggered by stop())

Blocked – The thread is waiting for I/O or a resource to become available. (Triggered by I/O requested, resumes when I/O done)

Sleeping – The thread is temporarily paused for a fixed time, , resumes automatically. (Triggered by sleep(), resumes when Done sleeping)

Waiting – The thread is waiting indefinitely for another thread's signal. (Triggered by wait(), resumes when notify() or notifyAll() is called)

Single Thread vs Multithread



- **Stack → Stores function calls & local variables**
- **Registers → Hold temporary data**
- **Counter/program Counter→ Keeps track of the next instruction to execute in the thread.**

Difference Between Single-Threading and Multi-Threading

Feature	Single-Threading	Multi-Threading
Number of Threads	Contain Only one thread that executes in the process	Multiple threads execute simultaneously within a same process
Resource Usage	Uses separate code, data, stack, and registers	Threads share code, data, and files but have separate stacks and registers
Execution Type	Sequential execution, tasks run one after another	Parallel execution, improving efficiency
Performance	Slower, as tasks must wait for completion	Faster, as multiple threads execute together
CPU Utilization	Less, as CPU may stay idle during I/O	More, as CPU can execute another thread while one waits
Context Switching	No thread switching, so no thread management overhead	Requires thread management but improves speed

Difference Between User-Level Thread and Kernel-Level Thread

Feature	User-Level Thread (ULT)	Kernel-Level Thread (KLT)
Managed By	Managed by user-level libraries (without OS intervention)	Managed directly by the operating system
Speed	Faster, as no kernel involvement	Slower, as system calls are required
Context Switching	Quick and does not require mode switching	Slower due to kernel intervention

Feature	User-Level Thread (ULT) 📝	Kernel-Level Thread (KLT) 📝
Dependency on Kernel	OS is unaware of user threads	OS is fully aware and manages kernel threads
Blocking	If one thread blocks, all threads in the process block	One thread blocking does not affect others
System Call Overhead	No system call overhead	Involves system call overhead
Example	Java Green Threads	Windows threads

Context switching time: process>kernel level thread>user level thread

Process takes more time and user level thread take less time for context switching less time means **Better CPU utilization**

Threading Models in Operating Systems

A threading model defines how user-level threads are mapped to kernel-level threads by the OS. It affects performance, concurrency, and scheduling.

📝 1. Many-to-One Model

- **Many user threads → One kernel thread**
- **Thread management is done by user-level library.**
- **Not truly concurrent on multicore systems.**
- **If one thread blocks, all block.**

✅ **Advantage: Fast and lightweight**
❌ **Disadvantage: No parallelism, blocking affects all**

📝 2. One-to-One Model

- **One user thread → One kernel thread**
- **True concurrency, uses OS-level scheduling.**
- **More overhead due to kernel management.**

✅ **Advantage: True parallelism**
❌ **Disadvantage: More memory and slower thread creation**

📝 **3. Many-to-Many Model**

- **Many user threads ↔ Many kernel threads**
- **OS maps user threads to a smaller or equal number of kernel threads.**
- **Combines the benefits of the above two models.**

✅ **Advantage: Good concurrency + efficient resource use**
❌ **Disadvantage: Complex to implement**

What is CPU Scheduling?

CPU scheduling is the process used by the OS to decide which process gets CPU time when multiple processes are waiting. It helps maximize CPU utilization and minimize response and waiting time.

Types of CPU Scheduling:

1. **Preemptive Scheduling**
    - CPU can be taken from a running process.
    - Ensures better CPU utilization but increases context switching overhead.
2. **Non-Preemptive Scheduling**
    - CPU is not taken from a process until it finishes.
    - Simpler but can cause delays (e.g., long-running processes blocking others).
3. **Demand Scheduling:**

    Definition: A scheduling approach where tasks are executed only when explicitly requested or triggered by an event.

    Execution: Tasks run **on demand** based on **user input**, **interrupts**, or **events**.

    Efficiency: Saves CPU and resources since tasks run only when needed.

    Example: Event-driven systems like embedded devices, sensors, or user-interface apps.

4. ⚙️ **Real-Time Scheduling:**

Definition: A scheduling approach used in **real-time systems** where tasks must meet **strict timing constraints**.

Execution: Tasks must complete before their **deadlines** (e.g., pacemaker, avionics systems).

Types:

Hard Real-Time: Missing a deadline results in **failure** (e.g., life-critical systems).

Soft Real-Time: Missing a deadline is **acceptable** occasionally (e.g., video streaming).

Example: Operating systems for **autonomous vehicles**, **robotics**, **industrial control systems**

Scheduling Criteria

These are **metrics** used to evaluate scheduling algorithms:

Criterion	Description
CPU Utilization	% of time CPU is busy
Throughput	Number of processes completed per unit time
Turnaround Time	Completion time − Arrival time
Waiting Time	Time spent in ready queue
Response Time	First response time − Arrival time
Fairness	Avoid starvation, give each process a chance

Scheduling Objectives

These are **goals** the OS tries to achieve through scheduling:

- **Maximize CPU utilization**: Keep CPU busy.
- **Maximize throughput**: Complete as many processes as possible.
- **Minimize turnaround time**: Time from submission to completion.

- **Minimize waiting time**: Time a process waits in the ready queue.
- **Minimize response time**: Time from submission to first response.
- **Ensure fairness**: All processes get a fair share of CPU.

Key Terminologies:

- **Arrival Time** → Time when a process enters the ready queue.
- **Burst Time** → Time required by a process to execute on CPU.
- **Turnaround Time (TAT)** → Completion Time - Arrival Time.
- **Waiting Time (WT)** → Turnaround Time - Burst Time.
- **Response Time** → Time from process submission to first response.

CPU Scheduling Algorithms Categorized

1. Non-Pre-emptive Scheduling Algorithms (here Waiting time=response time)

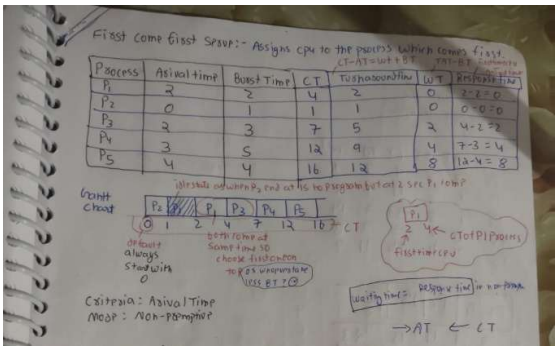
- **First Come, First Serve (FCFS)** – Executes processes in arrival order.
- **Shortest Job First (SJF)** – Executes the shortest burst time process first.
- **Priority Scheduling (Non-Pre-emptive)** – Executes the highest priority process first.
- **Highest Response Ratio Next (HRRN)** – Selects processes based on response ratio.

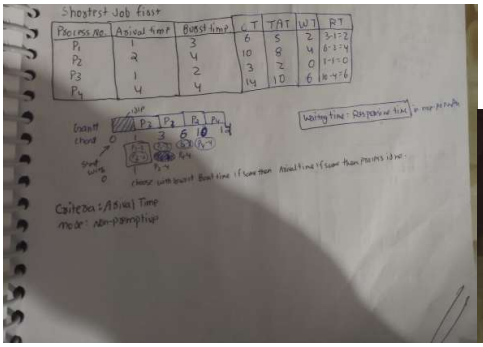
2. Pre-emptive Scheduling Algorithms

- **Shortest Remaining Time First (SRTF)** – Pre-emptive version of SJF.
- **Round Robin (RR)** – Processes get a fixed time slice before switching.
- **Priority Scheduling (Pre-emptive)** – CPU switches to a higher-priority process if it arrives.

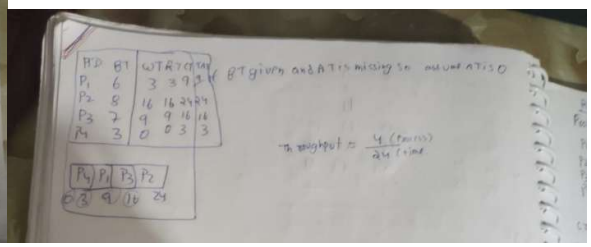
3. Hybrid Scheduling Algorithms (Mix of Pre-emptive & Non-Pre-emptive)

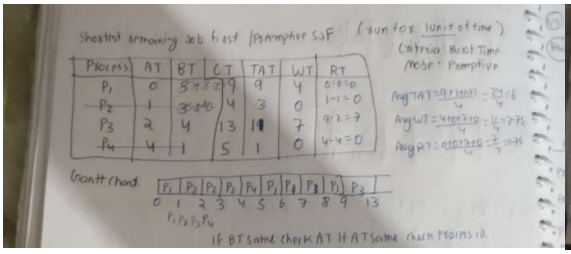
- Multi-**Level Queue (MLQ) Scheduling:** Fixed queues with no movement between them.
- Multi-**Level Feedback Queue (MLFQ) Scheduling:** Processes can move between queues based on behaviour.

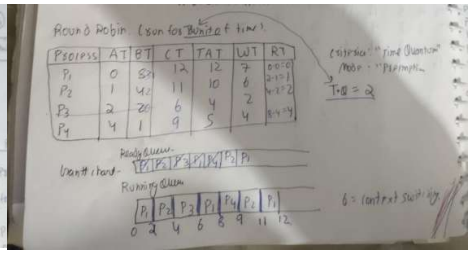

First come first serve:- Assigns cpu to the process which comes first.
Process | Arrival time | Burst Time | CT | TurnAroundTime | WT | Response Time
P1 | 2 | 2 | 4 | 2 | 0 | 2-2=0
P2 | 0 | 1 | 1 | 1 | 0 | 0-0=0
P3 | 2 | 3 | 7 | 5 | 2 | 4-2=2
P4 | 3 | 5 | 12 | 9 | 4 | 7-3=4
P5 | 4 | 4 | 16 | 12 | 8 | 12-4=8
Gantt Chart
P2 P1 P3 P4 P5
0 1 2 4 7 12 16 CT
Criteria: Arrival Time
Mode: Non-Preemptive
→AT ← CT

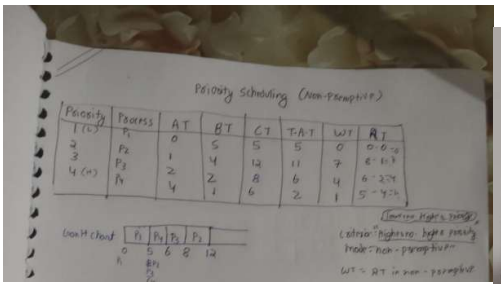

Shortest Job first
Process No. | Arrival time | Burst time | CT | TAT | WT | RT
P1 | | 3 | 6 | 5 | 2 | 3-1=2
P2 | 2 | 4 | 10 | 8 | 4 | 6-2=4
P3 | 1 | 2 | 3 | 2 | 0 | 1-1=0
P4 | 4 | 4 | 14 | 10 | 6 | 10-4=6
Gantt chart
P3 P2 P1 P4
0 3 6 10 14
Criteria: Arrival Time
mode: non-preemptive

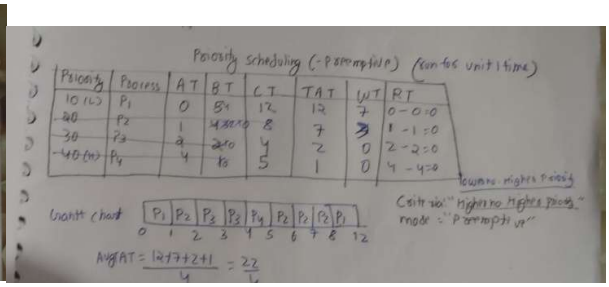
SJF W/O AT

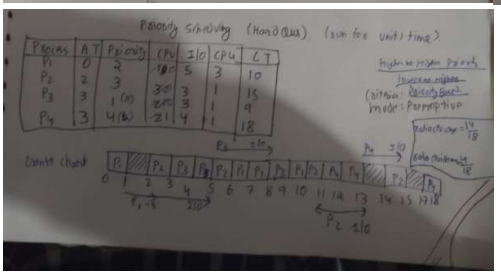

PID | BT | WT | TAT | CT
P1 | 6 | 3 | 9 | 9
P2 | 8 | 16 | 24 | 24
P3 | 7 | 9 | 16 | 16
P4 | 3 | 0 | 3 | 3
BT given and AT is missing so assume AT is 0
P4 P1 P3 P2
0 3 9 16 24

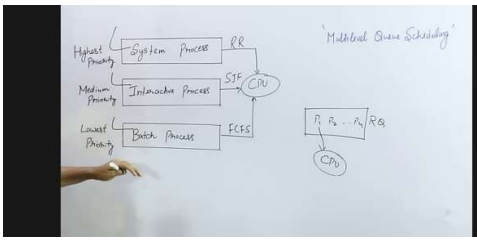

Shortest remaining Job first (Preemptive SJF) (Run for 1 unit of time)
Process | AT | BT | CT | TAT | WT | RT
P1 | 0 | 5 | 9 | 9 | 4 | 0-0=0
P2 | 1 | 4 | 13 | 12 | 8 | 1-1=0
P3 | 2 | 4 | 13 | 11 | 7 | 9-2=7
P4 | 4 | 1 | 5 | 1 | 0 | 4-4=0
Gantt chart
P1 P2 P2 P3 P4 P1 P2 P3 P1 P3
0 1 2 3 4 5 6 7 8 9 13
Criteria: Burst Time
Mode: Preemptive
IF BT same check AT IF AT same (Rank process id)

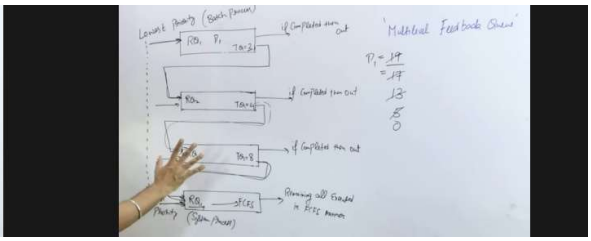

Round Robin (Run for 2 unit of time)
Process | AT | BT | CT | TAT | WT | RT
P1 | 0 | 5 | 12 | 12 | 7 | 0-0=0
P2 | 1 | 4 | 11 | 10 | 6 | 2-1=1
P3 | 2 | 2 | 6 | 4 | 2 | 4-2=2
P4 | 4 | 1 | 9 | 5 | 4 | 8-4=4
Criteria: Time Quantum
Mode: Preemptive
TQ = 2
Ready Queue
Gantt chart
Running Queue
P1 P2 P3 P1 P4 P2 P1
0 2 4 6 8 9 11 12
δ = context switching


Priority Scheduling (Non-Preemptive)
Priority | Process | AT | BT | CT | TAT | WT | RT
3 | P2 | 0 | 5 | 5 | 5 | 0 | 0-0=0
3 | P2 | 1 | 4 | 12 | 11 | 7 | 8-1=7
4 (n) | P4 | 2 | 2 | 8 | 6 | 4 | 6-2=4
4 | P4 | 4 | 1 | 6 | 2 | 1 | 5-4=1
towards higher priority
Criteria: Higher no. higher priority
mode: non-preemptive
WT = RT in non-preemptive

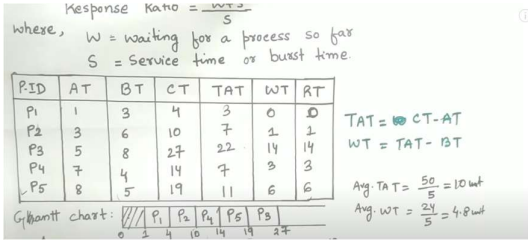

Priority scheduling (Preemptive) (Run for unit 1 time)
Priority | Process | AT | BT | CT | TAT | WT | RT
10 (L) | P1 | 0 | 5 | 12 | 12 | 7 | 0-0=0
20 | P2 | 1 | 4 | 8 | 7 | 3 | 1-1=0
30 | P3 | 2 | 2 | 4 | 2 | 0 | 2-2=0
40 (H) | P4 | 4 | 1 | 5 | 1 | 0 | 4-4=0
towards higher priority
Gantt chart
P1 P2 P3 P3 P4 P2 P2 P1
0 1 2 3 4 5 6 7 8 12
Criteria: Higher no. higher priority
mode: Preemptive
Avg AT = 12+7+2+1 / 4 = 22/4

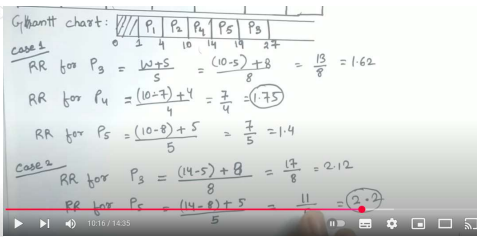

Priority Scheduling (Round Robin) (Run for unit 1 time)
Process | AT | Priority | CPU | I/O | CPU | CT
P1 | 0 | 2 | 1 | 5 | 3 | 10
P2 | 2 | 3 | 3 | 3 | 1 | 15
P3 | 3 | 1 (H) | 2 | 3 | 1 | 9
P4 | 3 | 4 (L) | 2 | 4 | 1 | 18
higher no higher priority
Criteria: Priority Based
mode: Preemptive
Gantt chart
P1 P2 P3 P3 P2 P1 P1 P2 P4 P1 P4 P2 P1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 18
P1 → 1
P2 → 10
P4 → 10
P2 → 10





Highest response ratio next





Process Synchronization

Process Synchronization is used in a computer system to ensure that multiple processes or threads can run concurrently without interfering with each other.

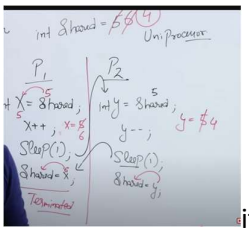
We need it for cooperative process not for independent process

☑ Goal of Synchronization:

- Avoid **Race Conditions**
- Maintain **Data Consistency**
- Ensure **Correct Execution**

Race condition in OS

A **race condition** happens when **two or more programs (or threads)** try to use or change the **same data at the same time**, and the **final result depends on who gets there first**.



if p1 start first then output 4 if p2 start first then output 6 which is correct? therefore we need synchronization

When p1 is sleeped means paused its excuted data till x++ are stored in pcb so that when it get cpu it can excute fro m it left and not do from start

What is a Critical Section?

It is a part of a program (usually in multithreaded applications) that **accesses shared resources** and **must not be concurrently accessed** by more than one thread or process.

❌ Problems Without Handling Critical Sections:

1. **Race Conditions** – Final result depends on the sequence of execution.
2. **Data Corruption** – Shared data gets overwritten or mismanaged.

Basic flow

```
do {
    ENTRY SECTION
    CRITICAL SECTION
    EXIT SECTION
    REMAINDER SECTION
} while (true)
```

🔐 **Entry Section**

- Code where a process **asks for permission** to enter the **critical section**.
- Makes sure **no two processes** enter at the same time.

⚙️ **Critical Section**

- The part where the process **accesses shared resources** (like variables, files, etc).
- Needs to be protected so that **only one process** can use it at a time.

🚪 **Exit Section**

- Code that **releases the lock** or signal.
- Lets other processes know they can now enter the critical section.

🧘 **Remainder Section**

- Code that runs **normally** (doesn't access shared resources).

☑ Properties to Follow (Criteria for Synchronization):

To handle the critical section properly, any solution must satisfy the following **four conditions**:

1. Mutual Exclusion

- Only **one process/thread** can enter the critical section at a time.
- Prevents race conditions.

2. Progress

- If no process is in the critical section and some processes wish to enter, **one of them must be allowed** to enter without unnecessary delay.
- Ensures system keeps moving forward.

3. Bounded Waiting

- There must be a limit on how many times other processes are allowed to enter their critical sections before a waiting process is granted access.
- Prevents **starvation**.

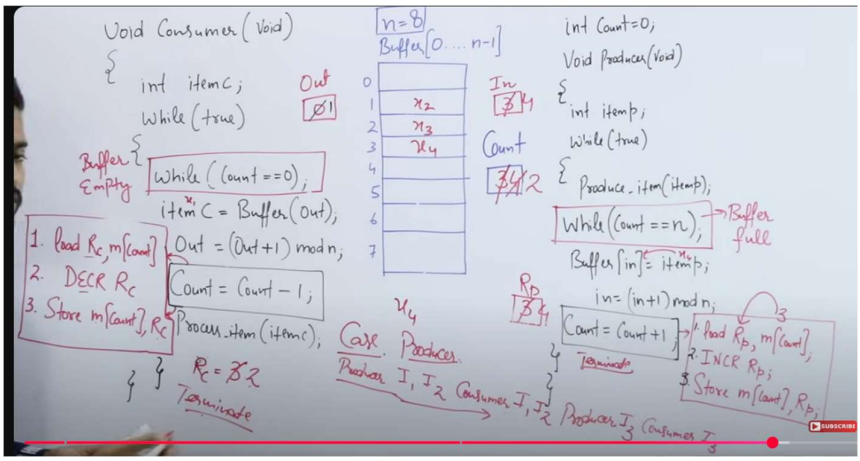
4. your solution must not be hardware dependent like solution work best in i5 but not in i3

producer and consumer problem

The **Producer-Consumer Problem** is a classic **synchronization problem** in operating systems and concurrent programming.

🔄 **What is the Producer-Consumer Problem?**

- There is a **shared buffer** (fixed-size storage).
- A **Producer** creates data and puts it into the buffer.
- A **Consumer** takes data from the buffer and processes it.
- The challenge is to make sure:
    - The **producer doesn't add** data when the buffer is **full**.
    - The **consumer doesn't remove** data when the buffer is **empty**.
    - They **don’t access the buffer at the same time** (to prevent race conditions).



Wrong count 2 but it is actually there is 3 item happen due to preemtiveness case is given i1 i2 run but i3 stop and premptive and then I1 i2 of producer run agin premptivre i3 of producer works

Solution of consumer producer problem: sempahores

Semaphores

A **semaphore** is an integer variable and synchronization primitive used in concurrent programming to control access to shared resources by multiple processes or threads.

Semaphores help prevent race conditions and ensure that multiple processes can safely interact with shared resources without causing data corruption or inconsistent states.

Semaphore Operations:

Semaphores provide two key operations, usually referred to as **wait** (or **P** operation) and **signal** (or **V** operation).

1. **wait(semaphore)** or **P() or down()**: for entry section
    - Decreases the semaphore value by 1.

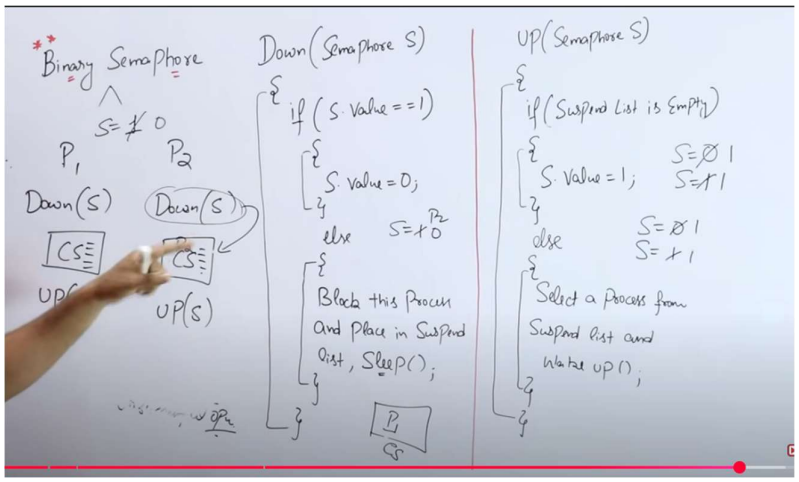
- If the semaphore value is greater than 0, the process continues; if the semaphore value is 0, the process is blocked until the semaphore becomes positive again.

2. **signal(semaphore)** or **V() or up() or post()**:for exit section
    - Increases the semaphore value by 1.
    - If there are any processes waiting on the semaphore (i.e., the value is 0), one of the waiting processes is awakened.

Types of Semaphores:

1. **Binary Semaphore** (also known as a Mutex):

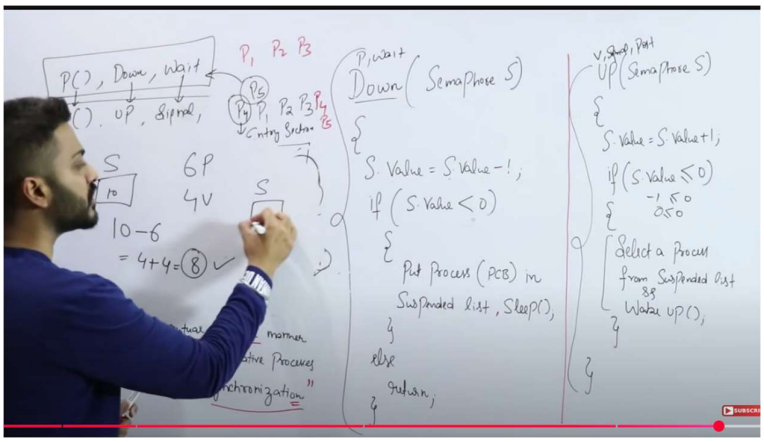
- Value: Only 0 or 1
- Purpose: Used for mutual exclusion (lock/unlock a critical section)



If process go in cs then successful operation if process blocked then unsuccessful operation

2. **Counting Semaphore**:

- Value: Any non-negative integer (0 to N)
- Purpose: Used to manage access to a pool of resources



final semaphore value is 4+4=8

Dining philosophers Problem and Solution using Semaphore in Operating System

The Dining Philosophers Problem is a classic example in operating systems that shows how multiple processes can safely share limited resources without causing deadlock, starvation, or race conditions.

📌 Problem Statement

- **N philosophers** sit at a round table.
- Each needs **2 forks (chopsticks)** (left and right) to eat.
- Only **N forks** are available (1 between each pair).
- Philosophers alternate between **thinking** and **eating**.

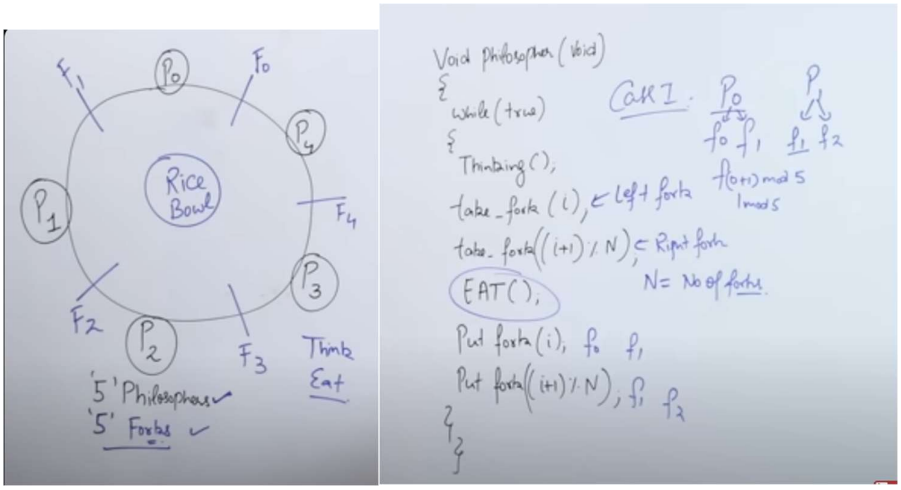
🔐 Constraints

- One philosopher can pick up **only one fork at a time**.
- A philosopher **eats only if both forks** are available.
- Forks are **shared** between adjacent philosophers.

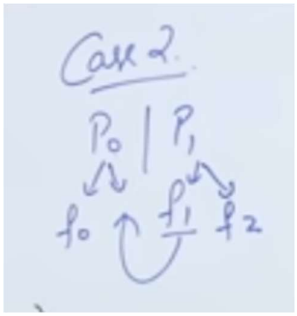
🧩 Problems to Solve

1. **Mutual Exclusion:** No two philosophers can use the same fork.
2. **Deadlock Prevention:** Avoid circular wait where each philosopher holds one fork and waits for the other.

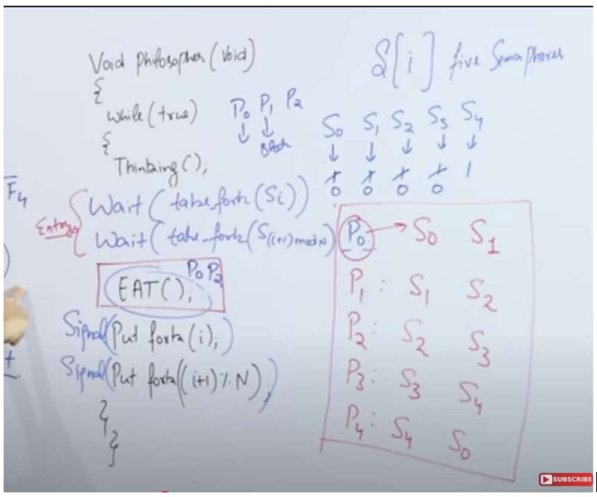
3. **Starvation Avoidance:** Every philosopher eventually gets to eat.



Case 1: Po come take fo and f1 complete it task and leave fo and f1 then p1 come take f1 and f2 and done its task and leave f1 and f2



Case 2: Po come take fo and suddenly p1 come and take f1 and f2 before po take f1 so there may be case of race condition so to avoid this we use binary semaphores



here first s0 and s1 has to down means set to 0 only then it can take fork else it block p0 come set s0 and s1 to 0 and take fork f0 and f1

and then p2 come it has to down s1 and s2 but s1 is already down so it get block in this manner only po and p2 run simultaneously because there fork are independent

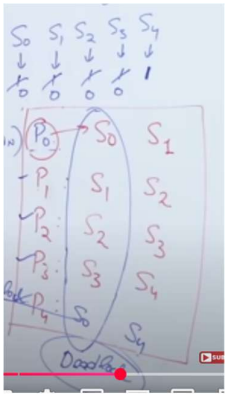
⚖️ **Asymmetric Solution to prevent circular wait or deadlock**

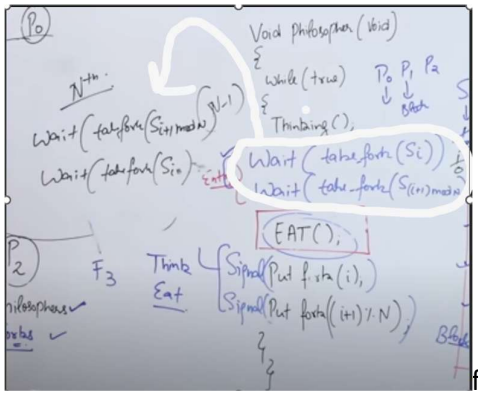
- Philosopher 0 to 3: pick left then right
- Philosopher 4: pick right then left

This avoids circular wait → no deadlock.

deadlock occur po come it take down so and take fo fork and before it down s1 it is pre-empted and same done for p2 p3 p4 then every semaphore is down but none of process get two fork to execute it task then deadlock occur

Its solution is to reverse p4 order then s4 will still 1 because p4 first try to take so which is already down to 0 so p4 block now p3 has f3 fork need f4 so s4 is now down to 0 so now p3 execute and leave its fork and then p2 then p1 then in last p4 execute





for last process we change the order of take fork

💈 Sleeping Barber Problem

The **Sleeping Barber Problem** is a classic **synchronization problem** in Operating Systems used to understand **process synchronization** and **inter-process communication** using semaphores.

🧠 The Scenario:

- A **barber shop** has:
  - **1 barber**
  - **1 barber chair**
  - **N waiting chairs** for customers
- **If no customers**, the **barber sleeps**.
- **If a customer arrives**:
  - If **chairs are available**, they **wait**.
  - If **no chairs**, they **leave**.
  - If the **barber is sleeping**, they **wake him up**.

🎯 Goal:

Synchronize the **barber** and the **customers** to avoid:

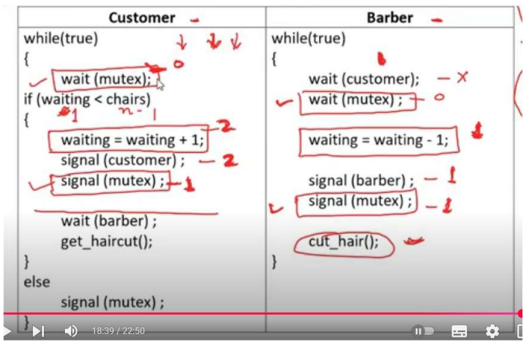
- Waking up the barber unnecessarily
- Having customers wait when there's no space
- More than one customer in the barber’s chair

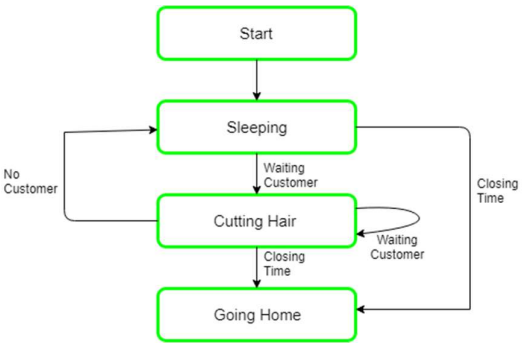
🔧 Used Concepts:

- **Semaphore** (for mutual exclusion and signaling)
- **Mutex lock** (to protect shared resources like chairs)

📌 Key Components:

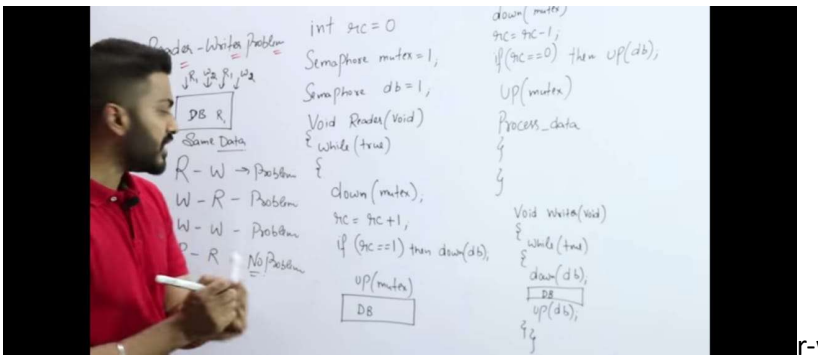
```
chairs = n
semaphore customer = 0   // no of customers in waiting room
semaphore barber = 0     // barber is idle (sleeping)
semaphore mutex = 1      // mutual exclusion
int waiting = 0          // no of waiting customer
```





Reader-Writer Problem

The **Reader-Writer Problem** is a classic **synchronization problem** in Operating Systems that deals with **accessing shared data** (like a file or database) by **multiple readers and writers**



r-w ,w-r,w-w cause problem if using same database but r-r and r-w,w-r,w-w using different db don't cause probkem

Mutual Exclusion Problem

Goal: Ensure **only one process** accesses the **critical section** at a time to avoid data inconsistency.

⚙️ **Hardware Solutions**

1. **Disabling Interrupts**
    - A process turns off all interrupts while entering the critical section.
    - ✅ Ensures uninterrupted access.
    - ❌ Not suitable for multiprocessor systems.
2. **Test-and-Set Instruction**
    - A special hardware instruction that atomically checks and sets a lock.
    - ✅ Works well on modern CPUs.
3. **Compare-and-Swap Instruction**
    - Atomically compares a memory value and updates it only if it matches expected value.
    - ✅ Prevents race conditions.
4. **Exchange Instruction**
    - Atomically swaps register and memory values to create a lock.

💻 **Software Solutions**

1. **Peterson's Algorithm** (for two processes)
    - Uses flags and turn variables to allow only one process in the critical section.
    - ✅ Works without hardware help.
    - ❌ Works for only 2 processes.
2. **Bakery Algorithm** (for multiple processes)
    - Like taking a token; the process with the smallest number gets the turn.
    - ✅ Fair for many processes.
3. **Flag Variables with Turn**
    - Each process sets a flag and waits for its turn.

Unit 3

Resource Concept in OS

In an **Operating System (OS)**, a **resource** refers to any component (hardware or software) that a process or thread requires to perform its tasks. These resources are limited and must be allocated and managed efficiently.

Types of Resources:

1. **CPU Time** - Processor time for executing tasks.
2. **Memory** - RAM for storing data and instructions.
3. **I/O Devices** - Devices like printers, disks, and keyboards.
4. **Files** - Data storage and access.
5. **Network Resources** - Bandwidth and network interfaces.
6. **Synchronization Resources** - Semaphores, locks, etc.

Resource Allocation and Management:

- The OS allocates resources to processes based on factors like priority and availability.
- Resources are deallocated after use, ensuring no conflicts.

Goals:

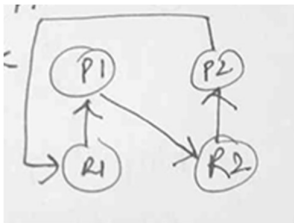
- **Fair allocation**, **efficient utilization**, **minimized wait time**, and **deadlock avoidance**

What is Deadlock?

Deadlock is a situation in computer systems (especially in OS or DBMS) where **two or more processes** are **waiting for each other to release resources**, and **none can proceed**.

🔗 **Necessary Conditions for Deadlock (All must be true):**

1. **Mutual Exclusion** – Only one process can use a resource at a time.
2. **Hold and Wait** – A process holding one resource is waiting for another.
3. **No Preemption** – A resource can't be forcibly taken; only released voluntarily.
4. **Circular Wait** – A closed chain exists where each process waits for a resource held by the next one.



🛠️ **Methods to Handle Deadlock**

1. Deadlock Ignorance(Ostrich Method) commonly used

- The system **does nothing** about deadlocks.
- Example: **Most operating systems like UNIX, Windows follow this** approach (they hope it never happens).
- If deadlock happens: Restart system or kill processes manually.

2. Deadlock Prevention

- Prevent at least **one of the four necessary conditions**.
- **Conservative** but safe.

Mutual Exclusion

🔧 **Solution**:

Use **sharable resources** where possible.

E.g., allow multiple processes to **read** a file simultaneously (read-only access).

Hold and wait

🔧 **Solution**:

Require each process to **request all required resources at once**, before execution.

Or, make it **release all resources** before requesting new ones.

No Preemption

🔧 **Solution**:

Allow the system to **take back (preempt)** resources from a process when needed.

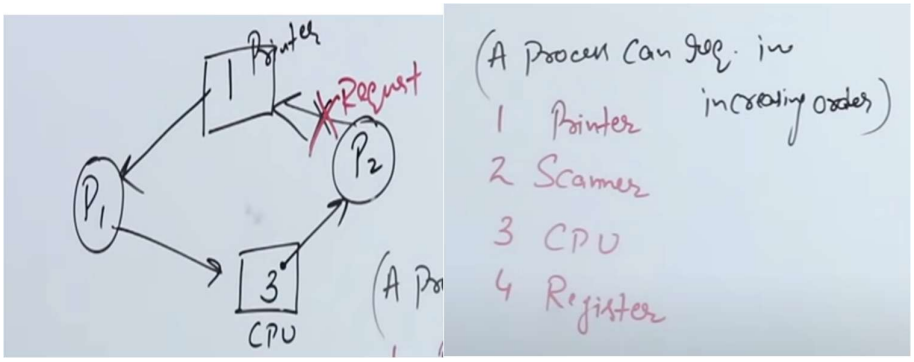
E.g., if a process is waiting, the system can force it to release its held resources.

Circular Wait

🔧 **Solution**:

Number all resources and require each process to **request resources in increasing order only**.

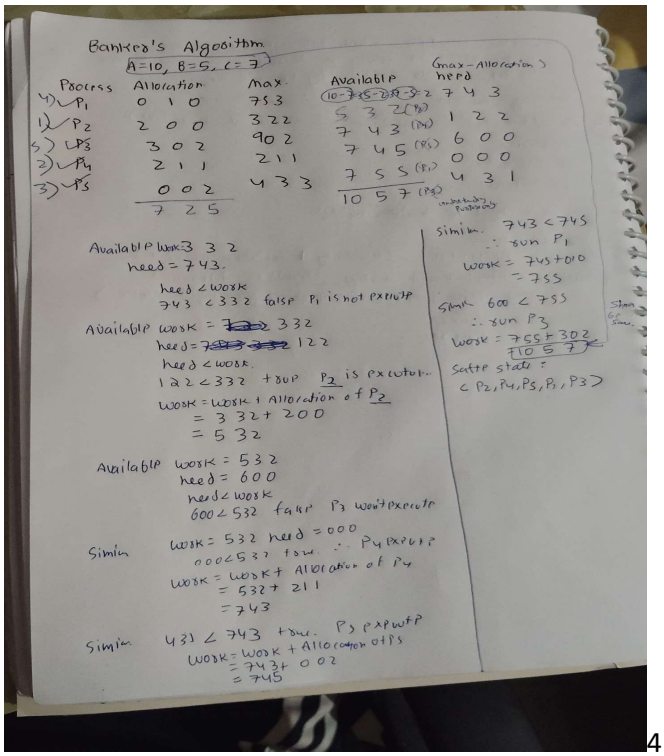
This breaks the cycle, so circular wait is not possible.



P1 is assigned printer and request for cpu p2 is assigned cpu and request for printer which has low number so wont assigned

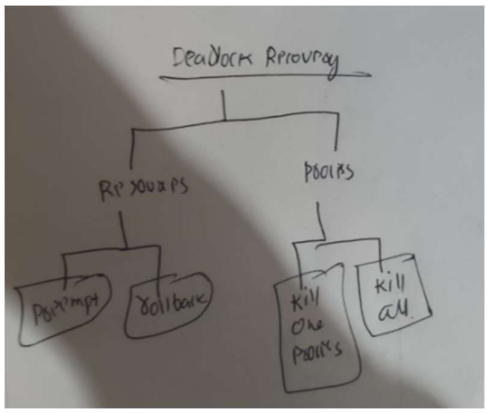
3. Deadlock Avoidance

- System checks **each request** and only grants if it **won't lead to a deadlock**.
- Example: **Banker's Algorithm**
- Needs prior knowledge of max resource needs.



4

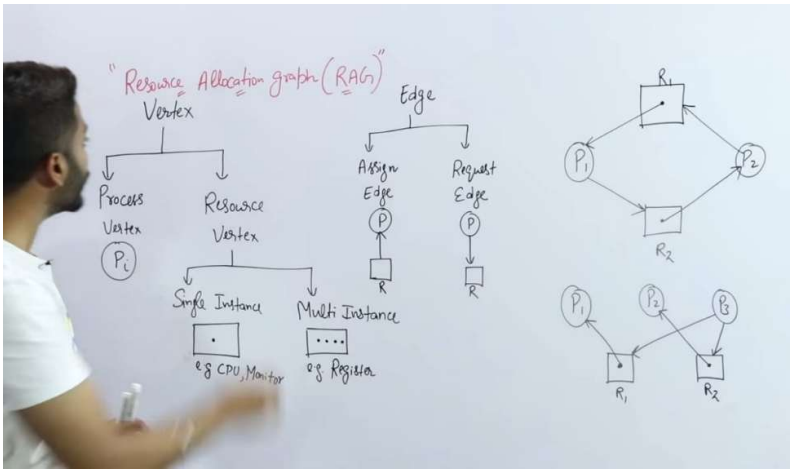
4. Deadlock Detection and Recovery (Deadlock Assurance)

- Allow deadlocks to occur, but **detect them** using algorithms.
- After detection, system **recovers**:
  - Kill a process one by one until deadlock vanished
  - Rollback a process to an earlier safe state and restart it.
  - Preempt resources from a process and give to others.
  - 

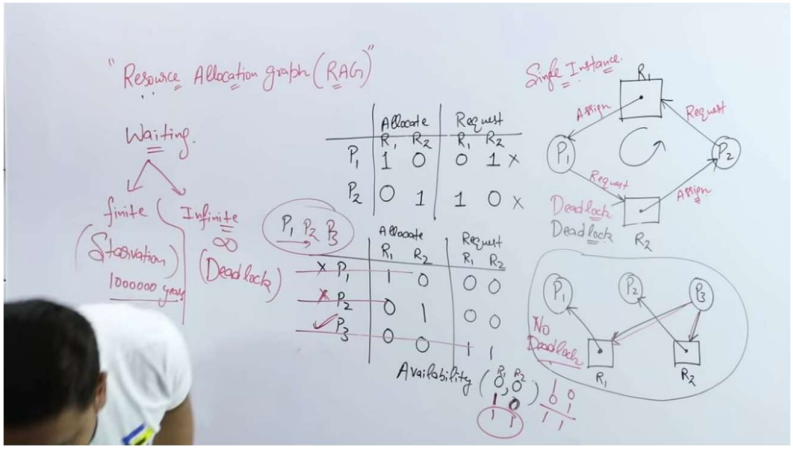
- Example: Resource Allocation Graph

RAG (Resource Allocation Graph)

A graphical tool used to detect deadlocks by showing the relationship between processes and resources.

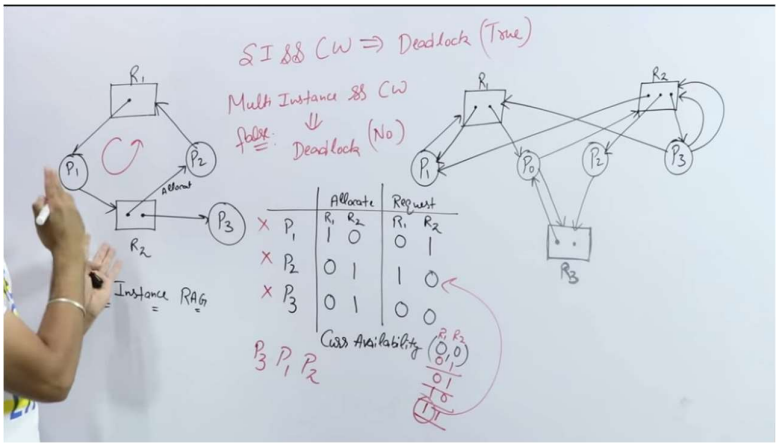


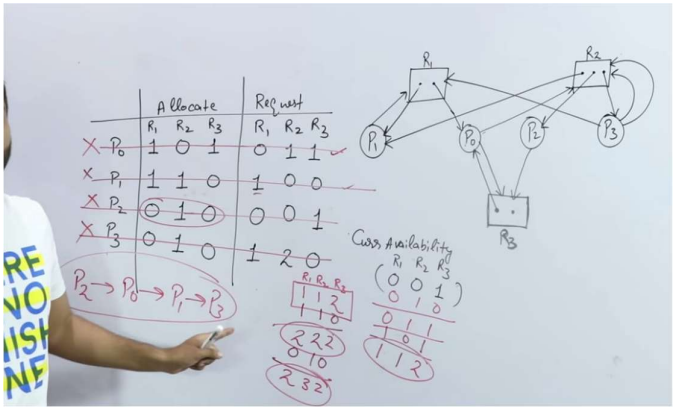
Single Instance Resource Graph



- Each resource has **only one instance**.
- **Deadlock is detected** if there is a **cycle** in the graph.
- **Ex-1 has deadlock(because cycle exist) and ex-2 has no deadlock**
- If waiting time is finite then starvation no deadlock look example two where p3 wait for finite time until p1 and p2 done and leave resource for p3 to be used so no deadlock
- In single instance graph if cycle exist then deadlock if cycle don't exist then no deadlock

Multi Instance Resource Graph





- Each resource can have **multiple copies/instances**.
- **Deadlock is possible** if there is a **cycle** and **not enough instances** to fulfil all requests.
- Cycle ≠ always deadlock here.
- **Ex-1,2 both has no deadlock although cycle exist**

Device Management

In **device management**, the operating system manages hardware resources and ensures smooth interaction between processes and devices (like printers, disk drives, and input/output devices). The system considerations involve:

1. **Efficiency**: Ensuring that devices are used effectively without wasting resources.
2. **Fairness**: Distributing device access fairly among processes, preventing resource starvation.
3. **Security**: Protecting devices from unauthorized access or misuse.
4. **Concurrency**: Handling simultaneous access requests to devices without conflict.

5. **Reliability**: Regularly checking device status (e.g., battery level, connectivity, memory usage) to catch problems early.
6. **Performance**: Optimizing response time and throughput for devices like disk drives and printers.

Rotational Optimization in Device Management

Rotational optimization focuses on improving disk access by minimizing **seek time** (time to move the disk arm) and **rotational latency** (time for the desired sector to rotate under the read/write head).

Optimization Strategies:

- **SSTF (Shortest Seek Time First),**
- **SCAN**,**C-SCAN**:
- **LOOK & C-LOOK**:

Caching in OS

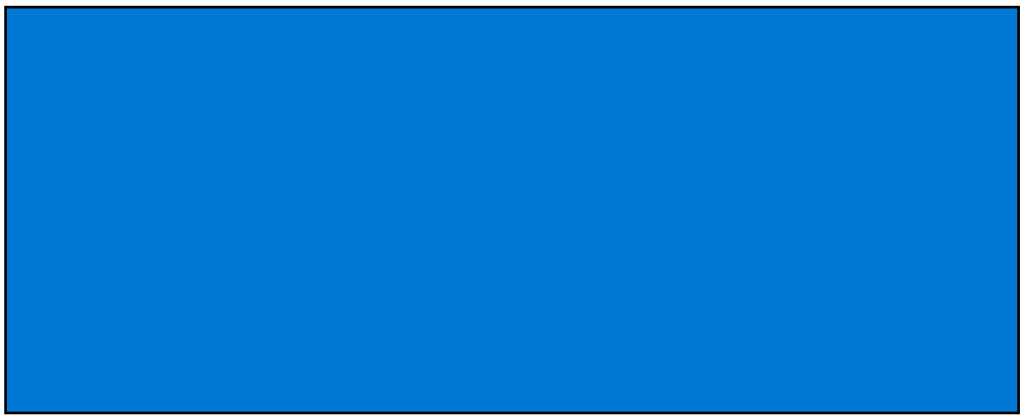
In disk management, caching helps in reducing the number of disk accesses by keeping frequently used data in **cache memory**.

- **Disk Caching**: The OS stores frequently used disk blocks in RAM or disk cache inside hdd/sdd for quicker access.
- **Write Caching**: Data is written to the cache first and then written to disk, improving performance.

Buffering in OS

Buffering refers to temporarily storing data in memory to handle differences in the speed of data production and consumption..

- **Input Buffering**: Stores data read from a device (like a keyboard or network) before it is processed. handle slow input device
- **Output Buffering**: Stores data to be written to a device (like a printer or disk) before the device [illegible]. sending to output device handle slow output device



Unit 4

◇ What is a File System?

A file is a collection of related information that is recorded on secondary storage.

The name of the file is divided into two parts as shown below:

- Name
- Extension, separated by a period.

A **file system** is a method used by an **operating system** to store, organize, retrieve, and manage data on storage devices (HDDs, SSDs, USBs). It serves as an interface between the OS and the storage hardware.

- **Common File Systems**
  - **FAT (File Allocation Table)**: Simple, older file system.
  - **NTFS (New Technology File System)**: Used in Windows, supports security, compression, encryption.
  - **ext (Extended File System)**: Popular in Linux (ext2, ext3, ext4).

- HFS (Hierarchical File System): A file system used by macOS.
- APFS (Apple File System): A new file system for Macs and iOS devices.

Logical File System (LFS) vs Physical File System (PFS)

Feature	Logical File System (LFS)	Physical File System (PFS)
Function	Manages files, directories, and metadata	Manages actual data storage on disk
User Interaction	Provides interface for file access and manipulation	Hidden from users; interacts with disk hardware
View	Logical view (e.g., /docs/report.txt)	Physical view (e.g., blocks, sectors, cylinders)
Responsibilities	- File naming - Directory structure - Access control	- Free space management - Block allocation - Disk buffering
Example Task	Checking if a user has permission to read a file	Writing file data to disk blocks
Layer	Higher-level file system layer	Lower-level storage management layer
Visibility to User	Visible (e.g., file explorer)	Invisible (handled by OS kernel and drivers)

🔧 File Operations

These are the actions that can be performed on files by the user or the system:

Operation	Meaning
Create	Make a new file in the file system.
Read	Retrieve data from a file.
Write	Add or modify data in a file.
Delete	Remove a file from the file system.
Truncate	Shorten a file by removing its content (makes size zero, file still exists means its attribute stay but content deleted).

Operation	Meaning
Reposition (Seek)	Change the current position of the file pointer (for reading/writing).

File Attributes

These are properties that describe a file and help the system manage it:

Attribute	Meaning
Name	The name of the file.
Extension	The type/format of file (like .txt, .jpg, .exe).
Identifier (ID)	A unique number used internally to identify the file.
Location	The disk address (or block) where the file is stored.
Size	The file's size in bytes.
Created Date	The date and time the file was originally created.
Modified Date	The date the file was last modified.
Accessed Date	The last time the file was opened/read.
Protection/Permission	Specifies who can read, write, or execute the file.
Encryption	Whether the file content is encrypted for security.
Compression	Whether the file is stored in a compressed form to save space.

File Allocation Methods in File System/ Types of File Organization/ File Allocation Strategies:

File Organization

File organization refers to the way data is stored and arranged on storage media (e.g., hard drives, SSDs). It determines how files are mapped to the storage device and influences performance, accessibility, and efficiency. There are several methods of organizing files based on access patterns and storage requirements.

These methods determine **how blocks of space are assigned to files** on storage devices:

in the context of an HDD, a **block** and a **sector** are essentially the same thing. The **sector** is the basic unit of storage on the disk, and it is also referred to as a **block** in many contexts, especially in the file system.

---

1. 📏 Contiguous Allocation

How it works:

- Each file is stored in a **single continuous block** of memory.

Pros:

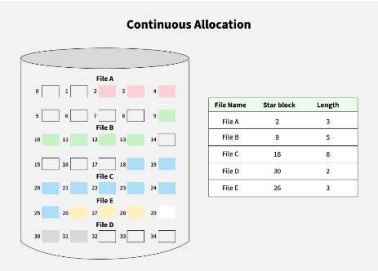
- Fast access — ideal for **sequential and direct access**.
- Simple implementation.

Cons:

- **External fragmentation** (free space is scattered).
- Need to know file size at creation.
- Difficult to grow file (file is stored at block 1 2 3 and file 2 on 4 5 6 now file 1 size increased as user add some extra data on it so it need 4 block but is it used) The **only solution** is to:**Find a new larger free space** that can fit the entire updated File 1.**Move the entire file** there (e.g., to blocks 10, 11, 12, 13) — which is time-consuming.

Example:

- File starts at block 5 and occupies 4 blocks → blocks 5, 6, 7, 8.



2. 🔗 Linked Allocation (Non-contiguous)

How it works:

- Each file is a **linked list of blocks**. Each block has a pointer to the next one.

Pros:

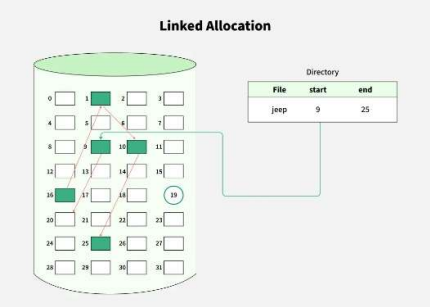
- No external fragmentation.
- File can grow easily.

Cons:

- **Larger seek time**
- **Random access/direct access is difficult**
- Overhead due to pointers.
- If one pointer is lost, **part of file is lost**.

Example:

- Block 3 → 12 → 7 → 22 (each block points to the next).



(start block of file is stored in linked allocation)

---

3. Indexed Allocation((Non-contiguous))

How it works:

- Each file has an **index block** that stores pointers to all its data blocks.

Pros:

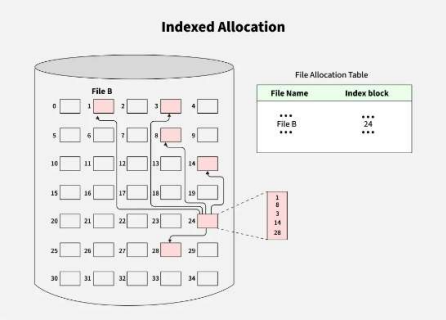
- Supports both **sequential and direct access**.
- No external fragmentation.

Cons:

- Overhead of maintaining the index block.
- If the file is very large, **multiple index blocks** may be needed.

Example:

- Index block → [block 2, block 7, block 15, block 28].



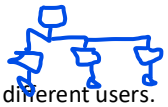
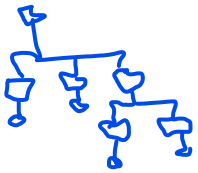
("In indexed allocation in OS, all the **addresses** (or **pointers**) of logically divided blocks of a particular file are stored in an index block.")

🎯 Goal of Allocation Methods

- **Efficient disk space utilization**.
- **Fast access** (sequential and/or direct).
- **Easy file growth** and dynamic allocation.
- **Minimize fragmentation** (internal or external).

◇ File Directories

- A **directory** stores information about files. **It** contains metadata about the file, such as the file name, file type, and a pointer to the location of the file's index block (if indexed allocation is used).
- in a 1TB storage HDD, the actual usable space will be slightly less than 1TB because some of the storage is reserved for **metadata** and **file system structures**.
- Types:
  - **Single-Level**: All files in one directory.
    - ❌ No grouping, naming conflicts.
  - **Two-Level**: One directory per user.
    - ✅ Supports same filenames for different users.
  - **Tree-Structured**: Hierarchical.
    - ✅ Efficient searching, supports grouping.

🔹 **Advantages of File Systems**

- 🗂️ Organized file storage
- 🔐 Data protection

Provides access control and security..

Supports file operations like read, write, delete.

- ⚡ Better performance

🔹 **Disadvantages of File Systems**

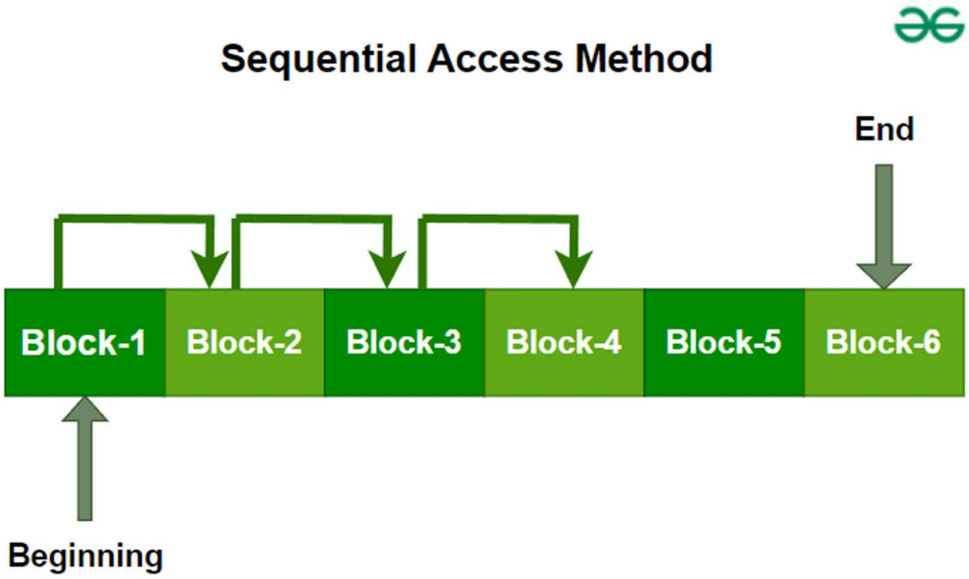
- ⚠ Compatibility issues between OSs
- 🧱 Disk space used by metadata
- 🛡 Vulnerable to corruption and attacks

Difficult to manage large volumes of data.

File access method/file data method/data access method

1. Sequential Access

- **Definition**: Accesses data **in order**, from start to end.
- **Operations**:
  - READ NEXT: Read the next piece of data in sequence.
  - WRITE NEXT: Append data to the end of the file.
  - REWIND: Go back to the beginning of the file. (0^{th} position)
  - SKIP: Move forward by a certain number of records.

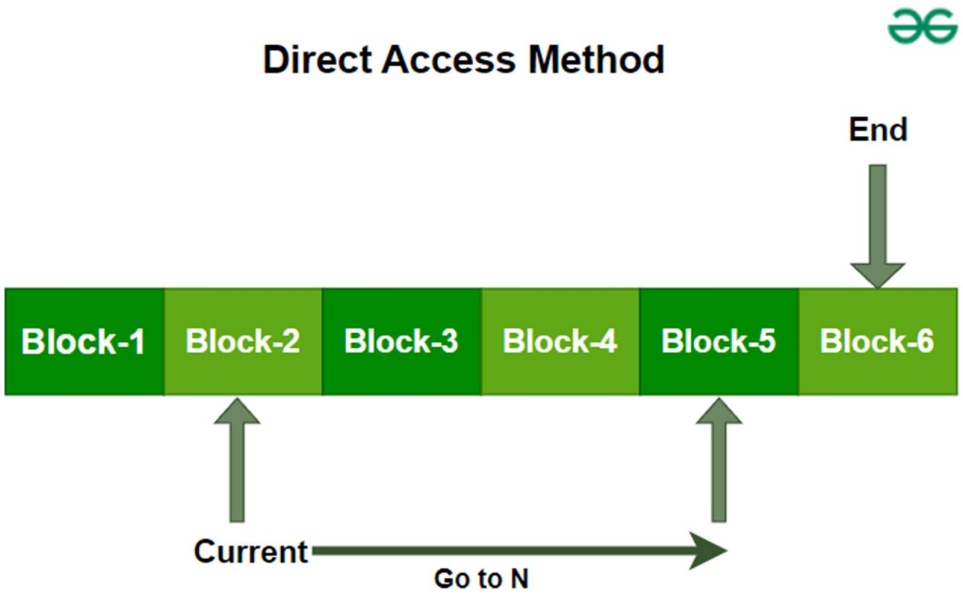


---

2. Direct (Random) Access

- **Definition**: Access data using a **specific record or block number**, not in order.

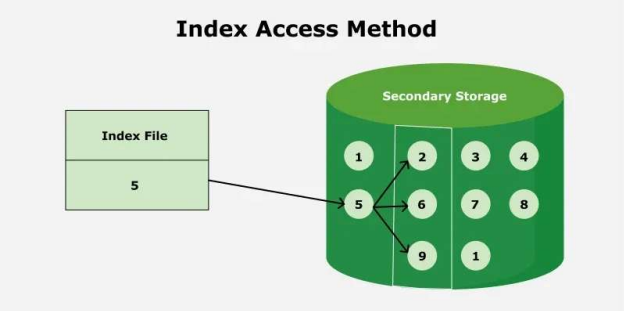
- **Operations**:
    - READ n: read the **nth record directly**
    - WRITE n: Overwrite data to the **nth position**.
    - JUMP TO record n: Move cursor to the **nth record** without reading previous ones.
    - QUERY CURRENT RECORD: Get the current record or position info
- **Example**: Accessing an element in an **array** or a **record in a database**.



---

3. Indexed Access(similar to index allocation(non-contiguous))

- **Definition**: Uses an **index** to first locate the position of data, then accesses it directly.
- **Process**:
    - Search **index**
    - Access corresponding **data block**
- **Efficient for**: **Large files** and fast lookups.
- **Example**: **Database indexing** using **B-Trees** or **hash indexes**.



What is Free Space Management?

Free space management refers to **how an OS keeps track of unallocated (free) disk blocks** so that they can be reused for storing files or data.

- **1. Bit Vector (Bit Map / Bit Array)**
  - **Concept**: Each block on the disk is represented by a **bit**:
  - *0 indicates that the block is free* and 1 indicates an allocated block. (but some programmer uses 1 as free and 0 is alloccated)
  - The OS uses this vector to find free blocks.
  - **Example**:

Bit Vector: [1, 0, 0, 1, 1, 0, 0]

Block # : 0 1 2 3 4 5 6

Free blocks are: **1,2,5,6**

- **Pros**: Simple and efficient for small disks.
- **Cons**: Large memory needed for large disks (1 bit per block).

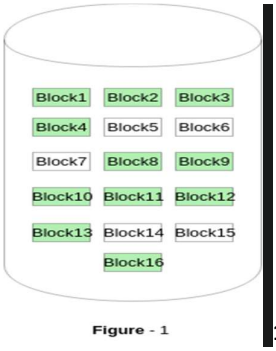


Figure - 1

1111000111111001. (bit map of disk sector each block is sector)

- **2. Linked List**
  - **Concept**: All free blocks are linked using pointers.

- Each free block contains a pointer to the next free block.

- **Example**:

Free blocks: 5 → 8 → 11 → 14 → NULL

- **Pros**: No need for extra large space like bit vector.
- **Cons**: Slow for locating multiple free blocks quickly.

🔹 **3. Grouping**

- **Concept**: Store addresses of free blocks in groups.
  - First block contains addresses of a few free blocks and pointer to the next group block.
- **Example**:

Block A:

Contains: [Block B, Block C, Block D, Pointer to Block E]

- **Pros**: Fast allocation of multiple blocks.
- **Cons**: Slightly more complex structure.

🧱 Total Free Blocks:

[5, 7, 9, 11, 13, 15, 17]

Block 5 → [7, 9 | 11]

Block 11 → [13, 15 | 17]

Block 17 → [-1, -1 | -1](no more free block)

🔹 **4. Counting**

- **Concept**: Free blocks are often found in **contiguous** chunks.
  - Keep track of the **starting block** and **count of free blocks**.
- **Example**:

(Start = 100, Count = 5) → Blocks: 100 to 104 are free

- **Pros**: Efficient for systems where free blocks are clustered. less memory used
- **Cons**: Not suitable for fragmented free space. not used in non contiguous

Case study on FAT32, NTFS, EXT2/EXT3, and exFAT file systems:

Feature	FAT32	NTFS	EXT2/EXT3	exFAT
Introduced	1996 (by Microsoft)	1993 (by Microsoft)	1993 (EXT2), 2001 (EXT3, for Linux)	2006 (by Microsoft)
Max File Size	4 GB	16 TB	16 GB (EXT3)	16 EB (exabytes)
Max Partition Size	8 TB	256 TB	16 TB	128 PB (petabytes)
Compatibility	Windows, Linux, macOS (highly)	Primarily Windows; limited on macOS/Linux	Primarily Linux; limited Windows support	Windows, macOS, some Linux, and devices
Journaling	❌ No	✅ Yes	✅ EXT3 only	❌ No
Security Features	❌ None	✅ File permissions, encryption, quotas	✅ (EXT3 has basic journaling)	❌ None
Advantages	Simple, widely compatible	Secure, reliable, feature-rich	Reliable (EXT3), simple (EXT2)	No 4GB limit, good for flash memory
Disadvantages	4 GB file limit, no security	Limited cross-platform support, complex	No native Windows support, slower (EXT2)	No journaling, fewer features than NTFS
Use Cases	Flash drives, external drives	Internal drives, Windows system partitions	Linux system storage, backups	Flash storage, external devices, SD cards

File access control

File access control in an OS manages who can access a file and what actions they can perform (e.g., read, write, execute). Key concepts include:

1. Permissions

Define allowed actions (read, write, execute) for the file owner, group, and others.
Example: Alice creates a file. She can read and edit it (write), her team can only read it, and others can't access it.

2. Access Control Lists (ACLs)

Lists specifying users/groups and their permissions on a file.
Example: Alice gives Bob read access and Charlie write access to her file.

3. Ownership

Files are owned by a user and group, who control access.
Example: A file created by Alice belongs to her and the "Editors" group. She decides who can use it.

4. Discretionary Access Control (DAC)

File owners control permissions.
Example: Alice removes Bob's access to her file because she owns it.

5. Mandatory Access Control (MAC)

The system enforces access control rules.
Example: Even though Alice owns the file, the system blocks her from sharing it outside the company.

6. Role-Based Access Control (RBAC)

Access is granted based on user roles.
Example: All managers can edit a report, but regular staff can only view it.

This ensures secure and organized access to files in an OS.

Data Integrity Protection in OS

Data integrity protection ensures that data is **accurate, consistent, and not changed accidentally or maliciously**.

🔑 **Key Methods of Protection:**

1. **Checksums and Hashing**
    - Detect if data has been changed.
    - **Example**: A file has a digital fingerprint (like a hash). If the fingerprint changes, it means the file was altered.
2. **Access Control**
    - Only authorized users can modify data.

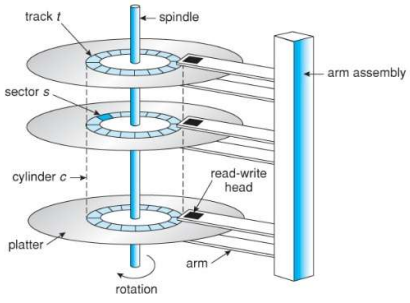
- **Example**: Only a manager can update salary records; others can only view them.

3. **Backups**
    - Original data can be restored if corrupted or lost.
    - **Example**: If a file is accidentally deleted, a backup copy can bring it back.
4. **File System Permissions**
    - Prevent unauthorized editing or deletion.
    - **Example**: A system file is marked read-only for normal users.
5. **Digital Signatures**
    - Ensure data came from a trusted source and was not changed.
    - **Example**: A signed PDF proves it's from your college and hasn't been modified.
6. **Transaction Controls (in databases)**
    - Ensure multiple operations complete fully or not at all (atomicity).
    - **Example**: While transferring money, the system ensures the amount is both debited from one account and credited to another. If not, nothing changes.

✅ **Goal:**

Unit 5

Disk Architecture

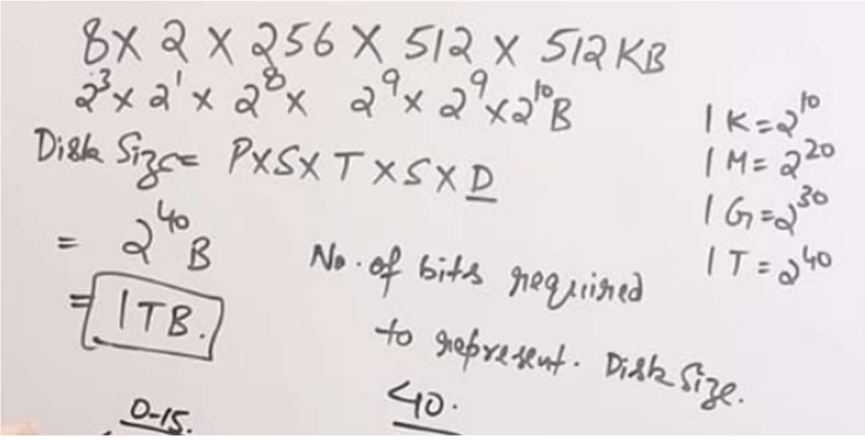


platter rotates in clockwise directon only outermost is 0th track inside deep at center 0th trach when power is cut read write head go to landing zone that not contain any data it could be outside or inside of disk platter for n plate 2n read/write head exist

(In above diagram we have 3 platter means 6 surface upside and downside)

Inside platter->surface(upper and lower)->track->sector->data

Disk size: platter*surface*track*sector*data



🧠 **Disk Access Time Components**

1. **Seek Time (ST)**
    - Time taken by the Read/Write (R/W) head to reach the desired track.
2. **Rotation Time (RT)**
    - Time taken for **one full rotation** (360°) of the disk.
3. **Rotational Latency(half of the rotation time.)**
    - Time taken to reach the desired **sector**.
4. **Transfer Time (TT)**
    - Time taken to transfer the actual data.
    - **Formula:**

$$\text{Transfer Time} = \frac{\text{Data to be transferred}}{\text{Transfer Rate}}$$

⚙ **Transfer Rate (Data Rate)**

$$\text{Transfer Rate} = \text{No. of Heads} \times \text{Capacity of one Track} \times \text{No. of Rotations/sec}$$

Disk access time=Seektime+rotational latency+transfertime+control_unit_time(drive)+Queue(time)

Disk Scheduling Algorithms

Disk scheduling helps decide **which request to process next** from a queue of I/O requests to the disk.

Goal: to minimise the seek time

Best case: When the **R/W head is already on the desired track** → **Seek time = 0**

Worst case: When the **desired location is farthest** (i.e., **nth track away**) from the current R/W head position → **Seek time is maximum**

---

1. FCFS (First Come First Serve)

- Serves requests in the order they arrive.
- **Simple but not efficient** (can cause long seek times).

---

2. SSTF (Shortest Seek Time First)

- Picks the request **closest** to the current head position.
- Faster than FCFS but can cause **starvation**.

---

3. SCAN (Elevator Algorithm)

- Head moves in one direction, services requests, then **reverses direction**.
- Like an elevator: goes up, comes down.

---

4. C-SCAN (Circular SCAN)

- Like SCAN but after reaching one end, it **jumps to the beginning** without servicing requests on the return.

---

5. LOOK

- Like SCAN, but the head only goes **as far as the last request** in one direction, then reverses.

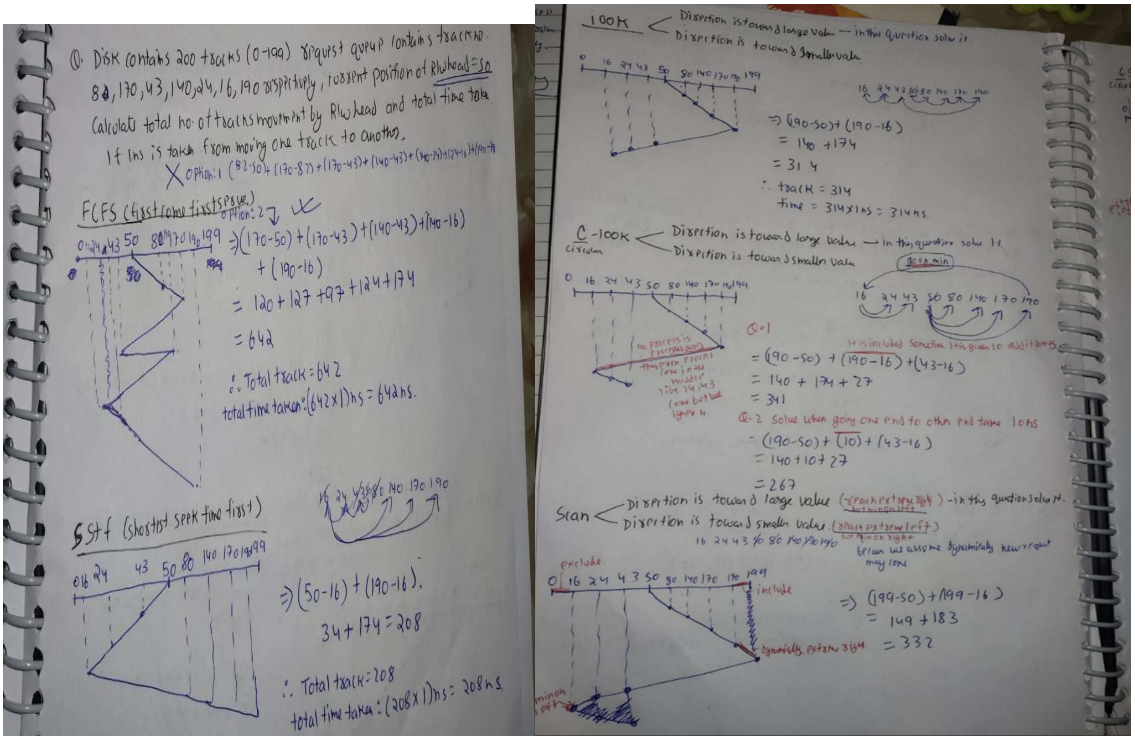
---

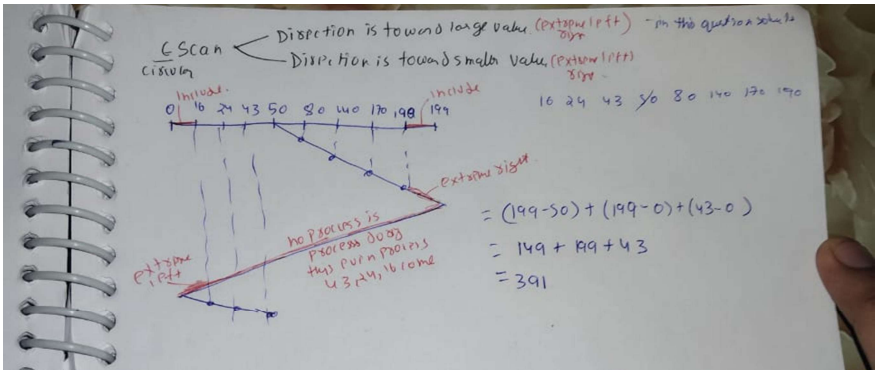
6. C-LOOK

- Like C-SCAN, but **only goes as far as the last request**, then jumps back to the lowest.

✨ Comparison Table

Algorithm	Performance	Starvation	Direction Aware
FCFS	❌ Slow	❌ No	❌ No
SSTF	✅ Good	⚠️ Yes	❌ No
SCAN	✅ Better	❌ No	✅ Yes
C-SCAN	✅ Better	❌ No	✅ Yes
LOOK	✅ Better	❌ No	✅ Yes
C-LOOK	✅ Better	❌ No	✅ Yes





Disk Reliability in OS

Disk reliability refers to the ability of a disk to function properly without failure over time. It is crucial for ensuring the safety of stored data and the stability of the system. Several factors influence disk reliability:

1. **Error Detection and Correction**:
    - Disks use techniques like **parity**, **checksums**, and **ECC (Error Correcting Code)** to detect and correct errors that might occur during reading or writing.
2. **Redundancy**:
    - Using **RAID (Redundant Array of Independent Disks)** systems or similar setups provides redundancy, ensuring that if one disk fails, data is not lost.
3. **Bad Sector Management**:
    - Disks can develop **bad sectors** over time, where data cannot be reliably stored. OS handles this by marking bad sectors and reassigning data to healthy areas.
4. **Disk Health Monitoring**:
    - Utilities like **SMART (Self-Monitoring, Analysis, and Reporting Technology)** help monitor the disk's health and predict failure, enabling preventive maintenance.
5. **Data Backup**:
    - Regular backups ensure that data can be restored in case of disk failure, improving data reliability.

Disk Formatting

Disk formatting is the process of preparing a disk for use by creating a file system, which organizes data on the disk in a way the OS can manage. There are two main types of disk formatting:

1) Low-Level Formatting (LLF):
Done by the manufacturer; sets up physical sectors and tracks on the disk.

2) High-Level Formatting (HLF):
Done by the user/OS; creates partitions and creates a file system (e.g., NTFS, FAT, ext4).

Partitioning: Dividing the disk into logical sections to organize data storage.

UNIX vs. Windows (Comparison Table)

Feature	UNIX	Windows
Origin	Developed in the 1970s at AT&T Bell Labs	Developed by Microsoft in the 1980s

Feature	UNIX	Windows
Interface	Primarily command-line, customizable	Graphical User Interface (GUI), user-friendly
Use Case	Servers, development, secure environments	Desktops, gaming, business environments
Performance	Lightweight, efficient, stable for servers	Can be resource-intensive, especially for GUI tasks
Customization	Highly customizable via shell and scripts	Limited customization compared to UNIX

Boot Block:

- **Definition**: The boot block is a special block on the storage device (e.g., hard disk or SSD) that contains the code or information needed to boot up the system.
- **Function**: When the computer is powered on, the BIOS or UEFI firmware looks for the boot block, which contains the bootloader—a small program that loads the operating system into memory.
- **Location**: On traditional hard drives, the boot block is typically located in the first sector of the storage device, known as the Master Boot Record (MBR) or GUID Partition Table (GPT) for modern systems.
- **Importance**: If the boot block is damaged or missing, the system will fail to boot.

Bad Block:

- **Definition**: A bad block is a section of storage that has become faulty and is unable to reliably store or retrieve data.
- occur due to physical damage, wear over time (especially in SSDs), or manufacturing defects.
- **Handling**: Operating systems or disk management tools often mark bad blocks to prevent them from being used. In some cases, data may be moved to good blocks automatically.
- **Types**: Bad blocks can be classified into two types:
    1. **Hard bad blocks**: These are permanently damaged and cannot be repaired , hard to recover caused by physical damage or scratches
    2. **Soft bad blocks**: These may be recoverable through error correction techniques or reallocation of data. caused by software error or file system issue

1. Two-Level Scheduling

Two-level scheduling is used in multiprogramming and multitasking systems to manage both long-term and short-term scheduling.

- **Long-term scheduler**: Selects which processes should be admitted into the ready queue (controls degree of multiprogramming).
- **Short-term scheduler (CPU scheduler)**: Selects from the ready processes and allocates the CPU to one of them.
  This approach balances system load and ensures efficient use of CPU and memory.

---

2. Shell

A **shell** is a command-line interpreter that provides a user interface to interact with the operating system.

- It takes user commands, interprets them, and passes them to the OS for execution.
- Examples: **Bash**, **sh**, **csh**, and **PowerShell**.
- Shells can execute built-in commands and also support scripting to automate tasks.

---

3. System Call

A **system call** is a programmatic way for user programs to request services from the operating system.

- Acts as an interface between user applications and the OS kernel.
- Common types: read(), write(), fork(), exec(), open(), close().
- Used for operations like file handling, process control, memory management, etc.

h) 💳 Smart Card OS

- A lightweight OS for secure chips in cards.
- Handles authentication, encryption, data storage.
- Highly secure and minimalistic.
- Example: SIM cards, ATM cards, ID cards.

f) 📱 Handheld System

- Designed for mobile or portable devices.
- Requires low power, touch interface, and wireless access.
- Examples: Smartphones, tablets.
- OS Example: Android, iOS

✅ **Advantages of Multiprocessor System:**

- 🚀 Faster processing (parallel execution)
- 🔁 More reliable (one CPU fails, others work)
- ⚙️ Efficient resource sharing
- 📊 Better performance for multitasking

❌ **Disadvantages of Multiprocessor System:**

- 🔧 Complex hardware & software
- ⚠️ OS becomes harder to manage
- 💰 Expensive setup
- 📡 Communication between CPUs can slow things down

short and clear comparison of Symmetric vs Asymmetric Multiprocessing:

- ---

Feature	🟩 Symmetric Multiprocessing (SMP)	🟧 Asymmetric Multiprocessing (AMP)
Processor Role	All CPUs are **equal**	One CPU is **master**, others are **slaves**
Task Assignment	Tasks are **shared equally** among CPUs	Master assigns tasks to slave CPUs

Feature	🟩 Symmetric Multiprocessing (SMP)	🟧 Asymmetric Multiprocessing (AMP)
Control	**Shared control** among all processors	**Master-controlled** system
Communication	**More complex**, needs synchronization	**Simpler**, as master handles control
Reliability	More **fault-tolerant**	Less fault-tolerant (master failure = system fail)
Usage	Modern systems, servers	Older or simple embedded systems

🧠 Operating System as an Extended Machine – Detailed Explanation

---

⚙️ **Without OS (Bare Machine):**

- User must write code to control hardware directly.
- Very **complex, error-prone**, and **hardware-dependent**.
- Example: To print something, you'd need to control the printer port using low-level machine instructions.

🛠️ **With OS (Extended Machine):**

- The OS provides **high-level commands** (like print("Hello")).
- Users don’t worry about **memory, I/O, CPU scheduling, or interrupts**.
- Makes programming **easier, safer, and portable**.

☑️ OS Features Required for Multiprogramming:

Multiprogramming means **multiple programs are loaded in memory and executed by the CPU one after another** to maximize CPU utilization.

Here are the essential OS features required:

---

1. 🧠 **Memory Management**
    - Efficiently allocate memory to multiple processes without conflict.
2. ⚙️ **CPU Scheduling**
    - Select which process gets the CPU when multiple are ready.
3. 🔐 **Protection and Security**
    - Prevent processes from interfering with each other's memory/data.
4. 📂 **Job Scheduling**
    - Decide the order in which jobs are picked for execution.
5. 📈 **Efficient I/O Management**
    - Handle I/O devices and allow CPU to switch to another job while waiting.
6. 📊 **Accounting**
    - Track CPU and resource usage per process.
7. 📝 **Process Management**
    - Create, manage, and terminate processes efficiently.

1) Key Difference Between a Trap and an Interrupt:

- **Trap**: A **synchronous event** triggered by the program itself (e.g., division by zero or invalid memory access). It's typically used for **error handling** or **system calls**.
- **Interrupt**: An **asynchronous event** triggered by external sources (e.g., hardware or devices like keyboard input). It interrupts the current process to signal the CPU to handle the event.

---

2) Types of System Calls:

1. **Process Control**: Manage processes (e.g., fork(), exec(), exit()).
2. **File Management**: Manage files (e.g., open(), read(), write(), close()).

3. **Device Management**: Interact with hardware devices (e.g., ioctl(), read(), write()).
4. **Information Maintenance**: Get or set system info (e.g., getpid(), getuid()).
5. **Communication**: Establish communication between processes (e.g., pipe(), msgsnd()).

---

3) Four Process Management System Calls:

1. fork(): Creates a new process.
2. exec(): Replaces the current process with a new process.
3. wait(): Makes a process wait for the completion of another process.
4. exit(): Terminates the current process.

---

4) User Mode and Kernel Mode:

- **User Mode**: The mode where **user applications** run with limited access to system resources.
- **Kernel Mode**: The mode where the **operating system kernel** runs with **full access** to all system resources.

Why two modes?:

- **User mode** ensures that programs cannot directly access or modify critical system resources, **preventing errors or security issues**.
- **Kernel mode** allows the OS to manage hardware and execute sensitive instructions.

---

5) Difference Between Loosely Coupled and Tightly Coupled System:

Feature	Loosely Coupled System	Tightly Coupled System
Definition	Systems that **operate independently**, communicate via **network**.	Systems where **processors share memory** and resources.

Feature	Loosely Coupled System	Tightly Coupled System
Communication	Communication through **message passing**.	Communication through **shared memory** or direct access.
Example	Distributed systems (e.g., cloud computing).	Multiprocessor systems (e.g., SMP).

---

6) Requirements of Hard Real-Time and Soft Real-Time Systems:

- **Hard Real-Time**:
  - **Critical deadlines** must be met, or failure occurs.
  - Example: **Airbag system** in cars.
- **Soft Real-Time**:
  - Deadlines are **desirable**, but not mandatory.
  - Example: **Video streaming**, where slight delays are acceptable.

---

7) Drawbacks of Monolithic Systems:

Monolithic Systems:

A monolithic system refers to an architecture where the entire operating system is built as a single, unified block. In this architecture, all the essential components of the operating system (such as process management, memory management, device drivers, file systems, and system calls) are tightly integrated into a single kernel.

1. **Complexity**: Difficult to maintain and debug.
2. **Poor Modularity**: Lack of clear separation of functions.
3. **Large Size**: The kernel is large and more error-prone.
4. **Scalability Issues**: Adding new features may affect existing ones.

---

8) Advantages of Layered Structure Over Monolithic Structure:

1. **Modularity**: Layers are independent, easier to maintain.
2. **Separation of Concerns**: Each layer handles distinct tasks.
3. **Easier Debugging**: Bugs are localized to specific layers.
4. **Portability**: Easier to adapt and modify layers without affecting others.

---

9) Examples of Microkernels:

- **MINIX**: A Unix-like operating system that uses a microkernel.
- **QNX**: A real-time operating system that uses a microkernel architecture.
- **L4**: A family of second-generation microkernels.

---

10) Differences Between Macro Kernel and Micro Kernel:

Feature	Macro Kernel	Micro Kernel
Size	Larger, **all functionalities** in one kernel.	Smaller, **only essential services** in the kernel.
Services	Provides services like **file management, memory** management, etc.	Only provides basic services like **IPC**, **scheduling**, etc.
Performance	Faster, as all services run in kernel space.	Slower due to communication between user space and kernel.
Complexity	More complex and harder to maintain.	Simpler, more modular and easier to extend.
Example	Linux, Windows NT	MINIX, QNX, L4

---

11) True or False Statements:

- a) **The user application interacts directly with O.S.**
  False. The user application typically interacts with the OS through system calls, not directly.
- b) **Shell is part of operating System.**
  True. The shell is a command-line interface that acts as a **user interface** for interacting with the OS.

59. Objectives and Minimal Set of Requirements for the File Management System:

Objectives:

1. **Data Storage**: Efficient and reliable storage of files.
2. **File Retrieval**: Quick and easy retrieval of files when needed.
3. **Security**: Protect files from unauthorized access.
4. **Integrity**: Ensure data consistency and correctness.
5. **Access Control**: Allow different access rights to users for file manipulation.
6. **Sharing**: Enable file sharing between users with proper controls.

Minimal Requirements:

1. **File Naming**: A mechanism to name and reference files.
2. **Directory Structure**: Efficient organization of files within directories.
3. **Access Mechanism**: Permissions to read, write, and execute.
4. **File Storage Management**: Allocation and deallocation of disk space.

---

60. Criteria for Choosing File Organization:

1. **Efficiency**: How fast can data be stored and retrieved? This is important for performance.
2. **Space Utilization**: How well is the disk space utilized (e.g., minimizing fragmentation)?
3. **Access Method**: The type of operations required (e.g., sequential, direct access).
4. **Security**: Ensuring that the file organization supports secure access controls.
5. **Portability**: Compatibility of the file organization across different systems.

6. **Maintainability**: How easy is it to update and modify the file system structure?

---

61. File System Architecture & File Management Functions:

File System Architecture:

- **File System Layer**: The file system provides an interface between the physical storage and user applications.
- **File Management**: Manages how files are created, stored, and accessed. It includes structures such as directories, file metadata, and storage blocks.
- **Storage Layer**: This is the underlying hardware or device (e.g., hard disk) that physically stores data.

File Management Functions:

1. **File Creation**: Allocating space and assigning a name.
2. **File Access**: Allowing reading or writing of data.
3. **File Deletion**: Removing files when no longer needed.
4. **File Maintenance**: Ensuring file integrity, consistency, and organization.
5. **Directory Management**: Creating, deleting, and updating directory entries.

---

62. Five File Organizations:

1. **Sequential File Organization**:
    - Data is stored one after another. Access is only possible in a sequential manner, making it slow for random access.
2. **Indexed File Organization**:
    - Files are stored in blocks, with an index to allow fast access to records. This allows both sequential and direct access.
3. **Hashed File Organization**:

- Uses a hash function to store records in a specific location based on a key value. It allows for fast retrieval of data by key.

4. **Clustered File Organization**:
    - Groups related records together in a block to optimize performance and reduce disk seek time.
5. **Tree-based File Organization**:
    - Uses tree structures (e.g., B-trees) to store records, allowing efficient searching, insertion, and deletion.

---

63. Typical Operations Performed on a Directory:

1. **Create**: Add a new file or subdirectory.
2. **Delete**: Remove an existing file or subdirectory.
3. **Rename**: Change the name of a file or directory.
4. **Search**: Locate a file or subdirectory.
5. **Traverse**: Navigate through the directory to access files or subdirectories.

---

64. Typical Access Rights Granted or Denied to a File:

1. **Read**: Allows a user to view the contents of a file.
2. **Write**: Grants the ability to modify the contents of the file.
3. **Execute**: Permits running the file if it is an executable program.
4. **Delete**: Allows the user to remove the file from the system.
5. **Append**: Allows adding data to the end of the file.
6. **Rename**: Lets the user change the file's name.

---

66. File System Consistency:

File System Consistency refers to ensuring that the file system remains **consistent** and **reliable** after system crashes or failures. The file system should always present a coherent and accurate view of data.

- **Consistency Checks**: The system ensures that the file system's data structures (e.g., directories, inodes) are not corrupted.
- **Atomicity**: File operations (e.g., writes) should be atomic—either they completely succeed or not at all.
- **Journaling**: Some file systems use journaling to log changes before they are made, ensuring that if a crash occurs, the system can roll back to a consistent state.
- **Backup and Recovery**: Regular backups and mechanisms for recovery are employed to ensure consistency in case of failures.

By maintaining consistency, the file system avoids issues like **data corruption** or **orphaned files**.

59. Memory Management Requirements:

Memory management in an operating system is essential for ensuring that computer memory is used efficiently and fairly. The requirements for effective memory management are:

1. **Memory Allocation**: Ensures that memory is allocated to processes as they need it and deallocated when it is no longer required.
2. **Address Binding**: Ensures that program addresses are mapped to physical memory locations, enabling processes to run properly.
3. **Memory Protection**: Prevents one process from accessing or modifying the memory allocated to another process.
4. **Memory Sharing**: Allows processes to share memory when needed for efficient resource use.
5. **Memory Fragmentation Management**: Deals with external and internal fragmentation to ensure that memory is used optimally.
6. **Swapping**: Enables processes to be moved in and out of physical memory to allow multiple processes to execute in limited memory.
7. **Virtual Memory Support**: Enables a system to appear to have more memory than is physically available by using disk storage.

---

60. Relocation Problem for Multiprogramming with Fixed Partitions:

The relocation problem arises when multiple programs (or processes) need to share memory, but the memory allocation is done with fixed partitions. When a program is loaded into memory, it is placed in a fixed partition, and the addresses are hardcoded to those memory locations.

- **Challenge**: If a process needs to be moved (relocated) due to memory fragmentation or other reasons, the memory addresses it uses may become invalid, causing the program to crash or malfunction.
- **Solution**: This problem can be resolved using **dynamic relocation** techniques, such as **base and limit registers**, which allow the operating system to adjust addresses when moving programs around in memory.

---

61. Static Partitioned Allocation with Partition Sizes 300, 150, 100, 200, and 20:

Using the **First Fit** allocation method, we have the following partition sizes: 300, 150, 100, 200, and 20.

Let's allocate the processes of size 80, 180, 280, 380, and 30 in the following manner:

- **Memory Status**:
    - Partition 1 (300): First process of size 80 is allocated here.
    - Partition 2 (150): Second process of size 180 cannot fit, but the third process of size 280 can. It goes into Partition 3.
    - Partition 3 (100): Third process of size 280 cannot fit.
    - Partition 4 (200): Fourth process of size 380 cannot fit.
    - Partition 5 (20): Fifth process of size 30 cannot fit.

Memory status after allocation:

Partition Size	Process Allocated	Remaining Space
300	80	220

Partition Size	Process Allocated	Remaining Space
150	-	150
100	-	100
200	-	200
20	-	20

---

62. What is Virtual Memory? How is it Implemented?

Virtual memory is a memory management technique that gives an "idealized abstraction" of the storage resources that are actually available on a given machine. It allows programs to access more memory than is physically available by using secondary storage (like a hard disk) to simulate additional RAM.

Implementation:

1. **Paging**: The memory is divided into fixed-sized blocks called **pages**, and the virtual address space is divided into **page frames** in physical memory.
2. **Page Tables**: A page table is maintained by the OS to map virtual pages to physical frames.
3. **Swap Space**: When physical memory is full, inactive pages are swapped to disk (swap space), freeing up space in RAM for active pages.
4. **Demand Paging**: Pages are loaded into memory only when needed, rather than all at once.

b. Page Fault Frequency & Thrashing:

- **Page Fault Frequency**: Refers to the rate at which page faults occur. A high frequency can indicate that the system is constantly accessing data from disk, slowing down performance.
- **Thrashing**: A situation where the system spends most of its time swapping data between the disk and RAM instead of executing processes, leading to severe performance degradation.
- **Control**: The operating system can control thrashing by adjusting the degree of multiprogramming or using page replacement algorithms to reduce page fault frequency.

---

64. Memory Allocation (First Fit, Best Fit, Worst Fit):

Given the memory holes (15K, 10K, 5K, 25K, 30K, 40K) and process sizes (12K, 2K, 25K, 20K), the allocation will proceed as follows:

- **First Fit**:
    - Process 12K goes into the first 15K hole.
    - Process 2K goes into the 5K hole.
    - Process 25K goes into the 30K hole.
    - Process 20K goes into the 40K hole.

Internal Fragmentation: 15K (hole) - 12K (process) = 3K, 5K (hole) - 2K (process) = 3K, 30K (hole) - 25K (process) = 5K, 40K (hole) - 20K (process) = 20K.

External Fragmentation: 10K hole remains unallocated.

- **Best Fit**:
    - Process 12K goes into the 15K hole (smallest that fits).
    - Process 2K goes into the 5K hole.
    - Process 25K goes into the 30K hole.
    - Process 20K goes into the 40K hole.

Internal Fragmentation: Similar to First Fit (3K, 3K, 5K, 20K).

External Fragmentation: 10K hole remains unallocated.

- **Worst Fit**:
    - Process 12K goes into the 40K hole (largest hole).
    - Process 2K goes into the 5K hole.
    - Process 25K goes into the 30K hole.
    - Process 20K goes into the 15K hole.

Internal Fragmentation: 40K (hole) - 12K (process) = 28K, 15K (hole) - 20K (process) = -5K (invalid), 30K (hole) - 25K (process) = 5K.

External Fragmentation: 10K hole remains unallocated.

63. Issues related to device-independent I/O software:

a. Uniform interfacing for device drivers:

It standardizes how the operating system communicates with different hardware devices, providing a consistent interface, so applications don't need to know specifics about each device.

b. Buffering:

Buffering stores data temporarily in memory during transfer between devices and processes to improve efficiency and prevent data loss due to speed mismatches between devices and the CPU.

---

64. Device Drivers:

Device drivers are specialized programs that enable the operating system to interact with hardware. They translate high-level I/O requests into device-specific commands and manage hardware communication.

---

65. Short notes on various topics:

a. Device-independent I/O Software:

Software that abstracts hardware details and provides a uniform interface for accessing various devices, making I/O operations easier and more efficient.

b. Goals of I/O Software:

- **Abstraction** of hardware details.
- **Efficiency** in resource usage.
- **Concurrency** in handling multiple I/O operations.
- **Error handling** for I/O failures.
- **Device management** and access control.

c. Interrupt Handler:

A function executed in response to an interrupt from hardware, temporarily halting the CPU's current task to handle the interrupt.

d. I/O Devices:

Hardware used for input and output operations, such as keyboards, printers, disk drives, and network interfaces.

e. Device Drivers:

Software that allows the OS to interact with hardware devices by converting high-level commands into device-specific instructions.

f. Device Controllers:

Hardware components that control the operation of I/O devices, acting as intermediaries between the device and the computer.

g. Disk Space Management:

Managing the allocation and deallocation of disk space, ensuring efficient use and organization of data on storage devices.

h. Disk Arm Scheduling Algorithm:

Algorithms that manage the movement of the disk arm in hard drives to minimize seek time, such as FCFS, SSTF, and SCAN.

---

66. Discussion on various storage technologies:

a) Magnetic Disk:

A storage medium using magnetic fields to read and write data, typically hard drives.

b) CDs:

Optical storage medium using laser to read/write data.

c) RAID:

A technology for combining multiple disk drives to improve performance and fault tolerance (e.g., RAID 0, RAID 1, RAID 5).

d) DVDs:

An optical disk format for storing larger amounts of data than CDs.

e) Formatting Disk:

The process of preparing a disk for use by creating a file system, initializing sectors, and organizing data storage.

---

67. Deadlock conditions, detection, and recovery:

Conditions for deadlock are: **Mutual exclusion, Hold and wait, No preemption, Circular wait.**

- **Detection**: The system periodically checks for cycles in the resource allocation graph.
- **Recovery**: Terminate processes or preempt resources to break the deadlock.

---

68. Deadlock Modeling:

Models use resource allocation graphs to represent the relationships between processes and resources, helping to identify deadlocks by detecting cycles in the graph.

---

69. Banker's Algorithm for multiple resources:

It checks if a system can safely allocate resources without leading to deadlock by ensuring that there is a safe sequence of process execution.

---

70. Deadlock detection with one resource of each type:

With only one resource of each type, deadlock detection is straightforward by checking if processes are waiting indefinitely for resources held by others.

---

71. Swap Space Management:

Swap space management involves using disk space as virtual memory when physical RAM is full, ensuring that the OS can handle memory demands by swapping data between RAM and the disk.

---

72. Memory and I/O Management:

- **Memory Management**: Managing the allocation, deallocation, and organization of memory (RAM) in a system.
- **I/O Management**: Handling input/output operations between the system and peripherals through device drivers, buffers, and efficient scheduling algorithms.

1) Process State Transition Diagram & PCB

States:

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

PCB Fields:

- PID, Process State, Program Counter, CPU Registers, Memory Info, I/O status, Priority.

Switching Overhead:

- Time/effort needed to save and load process states during context switch.

---

2) Round Robin – Time Quantum Effects

Large Quantum:

- Behaves like FCFS, poor response time.

Small Quantum:

- High context switch overhead, CPU wasted in switching.

Example:
3 processes (P1, P2, P3) with burst times 5, 5, 5

- Quantum = 5 → Each runs once (FCFS behavior).
- Quantum = 1 → Many switches, high overhead.

---

37) Mutual Exclusion Solutions

a) Lock Variable:

- Boolean flag; not reliable (race condition possible).

b) TSL Instruction (Test-and-Set Lock):

- Atomic hardware instruction to set a lock safely.

---

38) Mutual Exclusion Solutions for interrupt

a) Disabling Interrupts:

- Only one process runs, but not usable in multi-user OS.

b) Strict Alternation:

- Forces turn-taking using a turn variable; inefficient.

---

39) Monitor

Monitor:

- High-level sync construct with shared variables + condition variables.

---

40) Message Passing & Producer-Consumer

Message Passing:

- Processes communicate via send() and receive() functions.

Producer-Consumer via Message Passing:

- Producer sends items to consumer using send(),
- Consumer waits and reads items via receive().

---

37. Explain static partitioned allocation with partition sizes 300,150, 100, 200 and 20. Assuming first fit method indicate the memory status after memory request for sizes 80, 180, 280, 380, 30.

38. Free memory holes of sizes 15K, 10K, 5K, 25K, 30K, 40K are available. The processes of size 12K, 2K, 25K, 20K is to be allocated. How processes are placed in first fit, best fit, worst fit.

Calculate internal as well as external fragmentation.

39. Consider the following snapshot-

Allocated	Max				Available							
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Answer the following questions using banker's algorithm:

a) What are contents of matrix end?

b) Is the system in safe state?

c) If request for process p1 arrives for (0, 4, 2, 0) .Can the request be granted immediately?

1) Consider following processes with length of CPU burst time in milliseconds

Process	Burst time
P1	5
P2	10
P3	2
P4	1

All process arrived in order p1, p2, p3, p4 all time zero

a) Draw Gantt charts illustrating execution of these processes for SJF and round robin (quantum=1)

b) Calculate waiting time for each process for each scheduling algorithm

c) Calculate average waiting time for each scheduling algorithm

40. Consider following processes with length of CPU burst time in millisecond

Process	Burst time	Priority
P1	10	3

P2	1	1
P3	2	3
P4	1	4
P5	5	2

All processes arrived in order p1, p2, p3, p4, p5 all at time zero.

1) Draw Gant charts illustrating execution of these processes for SJF, non-preemptive priority (smaller priority number implies a higher priority) & round robin(quantum=1)

2) Calculate turnaround time for each process for scheduling algorithm in part (1)

3) Calculate waiting time for each scheduling algorithm in part (1)

41. Suppose a disk drive has 400 cylinders , numbered 0 to 399.The driver is currently serving a request at cylinder 143 and previous request was at cylinder 125 .The queue of pending request in FIFO order is: 86,147,312,91,177,48,309,222,175,130.

Starting from the current head position what is the total distance in cylinders that the disk to satisfy all the pending request for each of the following disk scheduling algorithms?

1] SSTS 2] SCAN 3] C-SCAN