1.what is os?

Ans: An Operating System (OS) is system software that acts as an intermediary between the user and the computer hardware.

It manages hardware resources (CPU, memory, disk, input/output devices) and provides services that allow applications to run smoothly.

3.ans:Functions of an Operating System:

Process Management – Handles creation, scheduling, and termination of processes.

Memory Management – Allocates and deallocates memory space as needed.

File System Management – Organizes and manages files on storage devices.

Device Management – Controls and coordinates input/output devices.

Security & Access Control – Protects data and resources from unauthorized access.

User Interface – Provides interaction methods (CLI, GUI, etc.).

Examples of OS:

Windows (e.g., Windows 10, 11)

Linux (Ubuntu, Fedora, Debian)

macOS

Android

iOS

2.what is kernel in os?

Ans: The Kernel is the core part of an Operating System (OS).

It sits between the hardware and the rest of the OS, controlling how software applications interact with the computer's hardware.

🧩 Types of Kernels:

Monolithic Kernel → Entire OS runs in kernel space (e.g., Linux, Unix).

Microkernel → Only essential functions in kernel; others run in user space (e.g., Minix, QNX).

Hybrid Kernel → Mix of monolithic and microkernel features (e.g., Windows NT, macOS).

Exokernel → Very minimal kernel, gives applications more direct hardware access.

4. System Software:

Definition: System software is the software that manages and controls the computer hardware so that application software can run.

It acts as a platform between the hardware and user applications.

✅ Examples:

Operating Systems (Windows, Linux, macOS, Android)

Utility Programs (antivirus, disk management tools, drivers, file managers, compilers, assemblers, linkers)

Application Software

Definition: Application software is the software designed to perform specific tasks for the user.

It runs on top of system software and directly helps the user do their work.

✅ Examples:

Productivity Software → MS Word, Excel, PowerPoint

Media Players → VLC, Windows Media Player

Browsers → Chrome, Firefox, Edge

Mobile Apps → WhatsApp, Facebook, Instagram

Games

5. 📁 File & Directory Commands

pwd → Show current working directory

ls → List files in directory

ls -l → Detailed list

ls -a → Show hidden files

cd dirname → Change directory

mkdir dirname → Create new directory

rmdir dirname → Remove empty directory

📄 File Operations

touch file.txt → Create empty file

cat file.txt → Show file content

nano file.txt or vim file.txt → Edit file

cp file1 file2 → Copy file

mv file1 file2 → Move or rename file

rm file.txt → Delete file

🔎 Viewing & Searching

head file.txt → Show first 10 line

tail file.txt → Show last 10 lines

grep "word" file.txt → Search for text inside file

find /path -name filename → Find file by name

🧑 User & Permission

whoami → Show current user

who → Show logged-in users

chmod 755 file → Change file permissions

chown user:group file → Change file ownership

⚙️ System & Process

ps → Show running processes

top → Monitor system processes in real-time

kill pid → Kill process by ID

df -h → Show disk usage

free -m → Show memory usage

uname -a → Show system info

🌐 Networking

ping google.com → Check network connectivity

ifconfig or ip a → Show network interfaces

wget url → Download file from internet

6. 🔷 **Process**

- A **process** is an **independent program** in execution.
- Each process has its own:
  - **Memory space** (code, data, stack, heap)
  - **Resources** (CPU time, files, registers, etc.)
- Processes are **heavyweight** (creating a new process takes more time and resources).
- Communication between processes requires **IPC (Inter-Process Communication)** like pipes, sockets, shared memory.

✅ Example:

- Running **Chrome** and **MS Word** are two separate processes.

🔷 **Thread**

- A **thread** is the **smallest unit of execution** within a process.
- Multiple threads of the same process **share the same memory space** but have their own stack and registers.
- Threads are **lightweight** (faster to create and switch than processes).
- Used for **parallelism and multitasking** inside a program.

✅ Example:

- In **Chrome browser (process)**:
  - One thread loads a webpage
  - Another thread plays a video
  - Another thread handles user input

🔑 **Key Differences (Process vs Thread)**

Aspect	**Process**	**Thread**
Definition	Independent program in execution	Smallest unit of execution inside a process
Memory	Has separate memory space	Shares memory of the process
Creation	Slower, needs more resources	Faster, lightweight
Communication	Needs IPC (pipes, sockets, etc.)	Easy (since threads share memory)
Failure	Crash of one process usually doesn't affect others	Crash of one thread can affect entire process
Example	Chrome, Word, VLC (different processes)	Tabs in Chrome (different threads of same process)

👉 **In short:**

- **Process = Independent program**
- **Thread = A task within a process**

6.🔷 **Multiprogramming**

- **Definition**: Technique of keeping **multiple programs in memory** at the same time so the CPU always has one to execute.
- When one program waits for I/O (like reading from disk), the CPU switches to another program.
- Goal: **Maximize CPU utilization**.

✅ Example:

- In early batch systems, while one program waited for input/output, another program could use the CPU.

🔷 **Multitasking**

- **Definition**: Ability of the OS to allow **multiple tasks (programs/processes)** to run at the **same time** by quickly switching the CPU between them.

- Uses **time-sharing** → CPU gives a small time slice to each task, creating the illusion of parallel execution.
- Goal: **Improve user interaction & responsiveness**.

✅ Example:

- You can listen to music, browse the web, and download files at the same time on your PC.

🔑 **Key Differences (Multitasking vs Multiprogramming)**

Aspect	**Multiprogramming**	**Multitasking**
Definition	Multiple programs kept in memory, CPU switches when one is waiting	CPU switches rapidly between tasks to give illusion of parallelism
Focus	CPU utilization	User interaction & responsiveness
User Involvement	Mostly in older batch systems (no user interaction)	Direct user interaction (modern OS)
Execution	One program runs until it waits for I/O	Multiple tasks appear to run simultaneously
Example	Old batch OS like early UNIX	Modern OS like Windows, Linux, macOS

👉 **In short**:

- **Multiprogramming = Keep CPU busy (utilization).**
- **Multitasking = Keep user happy (responsiveness).**

7.ans:

🔷 **Open Source Software (OSS)**

- **Definition**: Software whose **source code is publicly available**.
- Anyone can **view, modify, and distribute** it under its license (like GPL, MIT, Apache).
- Usually **free to use**, though support or enterprise editions may cost money.

✅ Examples:

- **Linux** (Ubuntu, Fedora)
- **Apache Web Server**
- **LibreOffice**
- **Python, PHP, MySQL**

🔷 **Subscription-Based Software (SaaS / Proprietary)**

- **Definition**: Software that requires users to **pay regularly (monthly/yearly)** to access or continue using it.
- Source code is **not shared** (closed-source).
- Usually includes **support, updates, and cloud features** as part of the subscription.

✅ Examples:

- **Microsoft 365** (Word, Excel, Outlook)
- **Adobe Creative Cloud** (Photoshop, Premiere Pro)
- **Netflix, Spotify** (subscription services)
- **Zoom Pro, Google Workspace**

🔑 Key Differences

Aspect	**Open Source Software**	**Subscription-Based Software**
Source Code	Open, can be modified	Closed, not accessible
Cost	Usually free (may charge for support)	Requires recurring payment
Ownership	Community or organization	Owned by company
Customization	High (you can change code)	Limited (depends on vendor)
Support	Community forums, optional paid support	Official vendor support included
Updates	Community-driven, free	Provided regularly by company
Examples	Linux, LibreOffice, GIMP	Microsoft 365, Adobe CC, Netflix

---

👉 **In short**:

- **Open Source = Free, customizable, community-driven.**
- **Subscription-Based = Paid, closed, vendor-supported.**

8.Ans:

A **Mobile Operating System (Mobile OS)** is a type of **operating system** specially designed to run on **smartphones, tablets, and other portable devices**.

It manages the **hardware (CPU, memory, sensors, touchscreen, camera, etc.)** and provides a **user interface and platform** for running mobile applications.

🔑 Functions of a Mobile OS

1. **User Interface (UI)** → Touchscreen gestures, icons, voice commands.

2. **Application Management** → Install, update, and run apps.
3. **Connectivity** → Handles Wi-Fi, Bluetooth, 4G/5G, GPS, etc.
4. **Device Management** → Controls sensors (accelerometer, gyroscope, fingerprint scanner).
5. **Security** → Provides app sandboxing, permissions, encryption.
6. **Power Management** → Optimizes battery usage.

📱 Examples of Mobile Operating Systems

🔷 Popular (current)

- **Android** → Developed by Google, most widely used.
- **iOS** → Developed by Apple, runs on iPhone and iPad.
- **HarmonyOS** → Developed by Huawei.

🔷 Older / Less common

- **Windows Phone OS** → By Microsoft (discontinued).
- **BlackBerry OS** → By BlackBerry (discontinued).
- **Symbian OS** → Used in Nokia phones (discontinued).
- **Palm OS** → Early smartphones.

🔑 Difference from Desktop OS

Feature	**Mobile OS**	**Desktop OS**
Input	Touch, gestures, voice	Keyboard, mouse
Connectivity	Cellular, GPS, sensors	LAN, Wi-Fi
Power Mgmt.	Optimized for battery	Not battery-focused
Apps	Lightweight, sandboxed	Heavy, full-featured
Examples	Android, iOS	Windows, Linux, macOS

👉 **In short:**
A **Mobile OS** is the **software brain of your smartphone** that controls hardware, runs apps, and makes the device user-friendly.

8.Ans:

Scheduling in OS

🔷 What is Scheduling?

- Scheduling is the method by which the OS **allocates CPU time** and other resources to processes.
- Since the CPU can only execute **one process at a time (per core)**, the OS must schedule tasks efficiently.

🔑 Types of Scheduling

1. Long-Term Scheduling

- Controls which **jobs** are admitted to the system for processing.
- Determines **degree of multiprogramming** (how many processes stay in memory).
- Example: Batch jobs in older systems.

2. Medium-Term Scheduling

- Temporarily suspends or resumes processes.
- Used to improve CPU utilization and balance load.
- Example: Swapping processes in/out of memory.

3. Short-Term Scheduling (CPU Scheduling)

- Decides which process in the **ready queue** gets the CPU next.
- Runs **very frequently** (milliseconds).
- Example: Choosing which app runs on your phone when multiple apps are open.

🔑 CPU Scheduling Algorithms

1. **First Come, First Served (FCFS)**
   - Processes served in order of arrival.
   - **Non-preemptive** (once running, not interrupted).
   - Simple but can cause **convoy effect** (long job delays short ones).
2. **Shortest Job Next (SJN) / Shortest Job First (SJF)**
   - Picks the process with the **smallest burst time**.
   - Can be **preemptive (SRTF)** or **non-preemptive**.
   - Optimal in theory but needs **knowledge of burst time**.
3. **Priority Scheduling**
   - Each process has a priority; highest priority runs first.
   - May cause **starvation** (low-priority processes wait too long).
   - Solution: **Aging** (increase priority of waiting jobs).
4. **Round Robin (RR)**

    - Each process gets a **time slice (quantum)**.
    - **Preemptive**, fair for all users.
    - Common in time-sharing systems.
5. **Multilevel Queue Scheduling**
    - Processes divided into queues (system, interactive, batch, etc.).
    - Each queue has its own scheduling algorithm.
6. **Multilevel Feedback Queue**
    - Processes can move between queues based on behavior.
    - Very flexible and fair.

🔑 **Criteria for Scheduling**

- **CPU Utilization** → Keep CPU busy.
- **Throughput** → Max number of processes finished in time.
- **Turnaround Time** → Time from submission → completion.
- **Waiting Time** → Time spent waiting in the queue.
- **Response Time** → Time until first response in interactive systems.
- **Fairness** → Equal CPU share for all.

👉 **In short**:

- **Scheduling = Deciding "who gets the CPU next."**
- Types = **Long, Medium, Short-term**.
- Algorithms = **FCFS, SJF, Priority, RR, Multilevel Queue, etc.**

9.Ans:

🔷 **Definition**

Starvation occurs when a process **waits indefinitely** in the **ready queue** because other higher-priority processes keep getting preference.

- The process is **ready to execute** but **never gets CPU time**.

🔷 **Cause of Starvation**

- Happens mainly in **priority-based scheduling**.
- Low-priority processes can be **continuously postponed** if high-priority processes keep arriving.

✅ Example:

- Suppose there are 3 processes:

Process	**Priority**	**Burst Time**
P1	1 (high)	5
P2	2 (medium)	3
P3	5 (low)	2

- If new high-priority processes keep arriving, **P3 may never execute**, leading to starvation.

🔷 Solution to Starvation

1. **Aging** – Gradually **increase the priority** of waiting processes over time.
2. **Fair Scheduling** – Use algorithms like **Round Robin** that ensure all processes get CPU.
3. **Hybrid Scheduling** – Combine priority with time-sharing.

🔑 Key Points

- Starvation = **indefinite waiting**
- Mainly occurs in **priority scheduling**
- Avoided by **aging** or **time-sharing techniques**

👉 In short:

Starvation is when a process is **ready but never gets CPU** due to other processes always taking priority.

10.Ans:

🔷 Definition

A **Process Control Block (PCB)** is a **data structure maintained by the OS** for **each process**.

- It **contains all the information about a process** needed for the OS to manage and schedule it.
- Whenever a process is created, the OS creates a **PCB** for it.

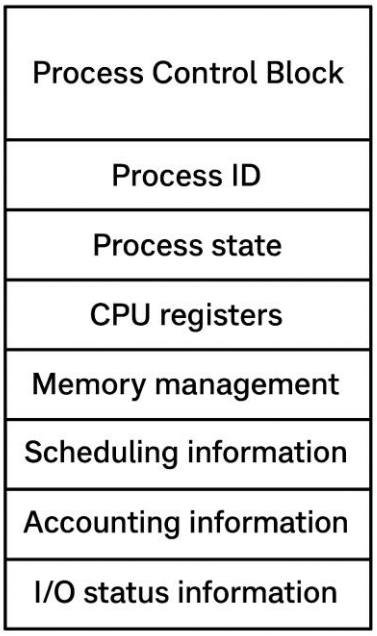
🔷 Contents of PCB

A PCB typically contains the following information:

1. **Process Identification**
    - Process ID (PID)
    - Parent process ID

- User ID, group ID

2. **Process State**
   - New, Ready, Running, Waiting, Terminated
3. **CPU Registers & Program Counter**
   - Current state of CPU registers
   - Address of next instruction to execute
4. **Memory Management Information**
   - Base and limit registers
   - Page tables or segment tables
5. **Scheduling Information**
   - Priority of the process
   - Scheduling queues it belongs to
6. **Accounting Information**
   - CPU usage, execution time, start time
7. **I/O Status Information**
   - List of I/O devices allocated
   - List of open files



🔷 **Role of PCB**

- **Context Switching** → PCB stores process state; OS can save and restore CPU state.
- **Process Management** → Helps OS keep track of all processes.
- **Resource Allocation** → Tracks what resources the process is using.

🔑 **In short**

PCB = **"Identity card" of a process** in the OS.
It keeps **all the info OS needs** to manage and schedule a process.

11. Deadlock is a situation in **operating systems** (or concurrent computing) where a set of processes are **unable to proceed** because each process is **waiting for a resource that is held by another process** in the same set. Essentially, every process in the group is waiting indefinitely for resources, and none can make progress.

Let's break it down:

Conditions for Deadlock

A deadlock can occur only if **all four of these conditions** hold simultaneously (known as **Coffman conditions**):

1. **Mutual Exclusion:**
   At least one resource must be held in a non-sharable mode; only one process can use the resource at a time.
2. **Hold and Wait:**
   A process is holding at least one resource and is waiting to acquire additional resources held by other processes.
3. **No Preemption:**
   Resources cannot be forcibly taken from a process; they must be released voluntarily.
4. **Circular Wait:**
   There exists a circular chain of processes where each process is waiting for a resource held by the next process in the chain.

Example

Imagine **two processes P1 and P2** and **two resources R1 and R2**:

- P1 holds R1 and waits for R2.
- P2 holds R2 and waits for R1.

Neither can proceed → **deadlock occurs**.

Deadlock Handling Methods

1. **Prevention** – Make at least one of the Coffman conditions impossible.
   - Example: Deny circular wait by imposing a resource ordering.
2. **Avoidance** – Dynamically check resource allocation to ensure the system never enters an unsafe state.
   - Example: **Banker's Algorithm**.
3. **Detection and Recovery** – Allow deadlocks but detect them and recover.
   - Example: Use a **wait-for graph** to detect cycles, then terminate or roll back processes.
4. **Ignore** – Sometimes, especially in small systems, deadlocks are rare and ignored.

12.ans:

Memory Management is a fundamental concept in operating systems that deals with **efficiently allocating and managing the computer's memory (RAM) among processes**. Its goal is to ensure that programs run smoothly without conflicts, while maximizing performance.

Let's break it down:

---

1. Objectives of Memory Management

- **Allocate memory** to processes when needed.
- **Track memory usage** (which parts are free/occupied).
- **Deallocate memory** when processes terminate.
- **Prevent conflicts** like overlapping memory or illegal access.
- **Optimize performance** by minimizing fragmentation and maximizing utilization.

---

2. Types of Memory

1. **Primary Memory (RAM)** – Fast access, volatile, directly used by CPU.
2. **Secondary Memory (Disk/SSD)** – Slower, non-volatile, used for storage.
3. **Cache Memory** – Very fast memory between CPU and RAM to speed up access.

---

3. Memory Allocation Techniques

a. Contiguous Memory Allocation

- Each process is loaded into a **single contiguous block** of memory.
- **Advantages:** Simple, fast access.
- **Disadvantages:**
    - Leads to **external fragmentation**.
    - Hard to allocate memory for large processes if space is fragmented.

b. Non-Contiguous Memory Allocation

- Process memory can be scattered in different blocks.
- Implemented via **paging** or **segmentation**.

---

4. Memory Management Techniques

a. Paging

- Divides memory into fixed-size **frames** and processes into **pages**.
- Pages are loaded into any available frame (non-contiguous).
- **Advantages:** No external fragmentation, easy to manage.
- **Disadvantages:** Some internal fragmentation.

b. Segmentation

- Memory is divided according to **logical segments** like code, data, stack.
- Each segment can grow independently.
- **Advantages:** Logical organization, easier for modular programming.

- **Disadvantages:** Can still cause **external fragmentation**.

c. Virtual Memory

- Allows processes to use more memory than physically available.
- Uses **disk as an extension of RAM**.
- Implemented using **paging or segmentation**.
- **Advantages:** More processes can run simultaneously.
- **Disadvantages:** Slower access (page faults if data not in RAM).

---

5. Memory Management Policies

- **First Fit:** Allocate the first available block that fits the process.
- **Best Fit:** Allocate the smallest block that fits the process (minimizes wasted space).
- **Worst Fit:** Allocate the largest block (may leave medium-sized blocks free).

---

6. Problems in Memory Management

1. **Fragmentation**
    - **External:** Free memory is split into small blocks that cannot be used efficiently.
    - **Internal:** Allocated memory may have unused space within blocks.
2. **Thrashing**
    - Excessive paging in virtual memory slows down the system.

💡 **Summary:**
Memory management ensures **efficient and safe use of RAM**, decides **which process gets memory, when, and where**, and optimizes system performance.

13.ans

Page Replacement Algorithms are used in operating systems when **a page fault occurs** and there is **no free frame** in memory. The OS must **replace an existing page** with the new one. The goal is to **minimize page faults**.

Here's a detailed explanation:

---

1. What is a Page Fault?

A **page fault** occurs when a process tries to access a page that is **not currently in main memory (RAM)**.
The OS must **bring the page from disk into memory**, possibly replacing an existing page.

---

2. Page Replacement Algorithms

a. FIFO (First-In, First-Out)

- Replace the page that has been in memory **the longest**.
- **Implementation:** Queue (first loaded → first replaced).
- **Pros:** Simple.
- **Cons:** Can suffer from **Belady's anomaly** (more frames → more page faults sometimes).

Example:
Frames = 3, Reference String = 7, 0, 1, 2, 0, 3, 0, 4

- Pages loaded in order, oldest removed first.

---

b. LRU (Least Recently Used)

- Replace the page that **has not been used for the longest time**.
- **Implementation:** Use counters or stack.
- **Pros:** Generally better than FIFO.
- **Cons:** More complex to implement.

---

c. Optimal (OPT)

- Replace the page that **will not be used for the longest time in the future**.
- **Pros:** Lowest possible page fault rate (theoretical).
- **Cons:** Cannot be implemented in practice (requires future knowledge).

---

d. LFU (Least Frequently Used)

- Replace the page with the **lowest access frequency**.
- **Pros:** Good for workloads with stable working sets.
- **Cons:** Can keep old but rarely used pages, causing unnecessary faults.

---

e. NRU (Not Recently Used)

- OS keeps track of **referenced (R) and modified (M) bits**.
- Classifies pages into four categories and replaces a page from the **lowest class**.
- Efficient but approximate.

---

3. Summary Table

Algorithm	Principle	Pros	Cons
FIFO	Replace oldest page	Simple	Belady's anomaly
LRU	Replace least recently used page	Good performance	Hard to implement
OPT	Replace page used farthest in future	Optimal	Needs future knowledge
LFU	Replace least frequently used page	Stable workloads	Can keep old pages
NRU	Replace based on R/M bits	Simple & efficient	Approximate

💡 **Tip:**

- **FIFO** → "Oldest out"
- **LRU** → "Least recent out"
- **OPT** → "Future out"

13. **Virtual Memory** is a memory management technique in operating systems that allows a process to **use more memory than is physically available** in RAM by temporarily transferring data to disk storage. It creates an **illusion of a very large main memory** for processes.

---

1. Purpose of Virtual Memory

- Allows **execution of programs larger than physical memory**.
- **Multiprogramming:** More processes can run concurrently.
- **Isolation:** Each process gets its own address space, improving **security and stability**.
- **Efficient memory use:** Only required pages are loaded into RAM.

---

2. How Virtual Memory Works

- Each process has a **logical (virtual) address space**.
- The OS and **Memory Management Unit (MMU)** translate virtual addresses into **physical addresses** in RAM.
- Pages not currently in RAM are stored in **secondary storage (disk)**.

Process flow:

1. CPU generates a **virtual address**.
2. MMU translates it to a **physical address**.
3. If the page is in RAM → **access succeeds**.
4. If the page is not in RAM → **page fault occurs** → OS loads page from disk into RAM.

---

3. Implementation Techniques

a. Paging

- Memory is divided into **fixed-size frames**, and processes into **pages**.
- Non-contiguous allocation → no external fragmentation.
- OS keeps a **page table** mapping virtual pages → physical frames.

b. Segmentation

- Memory is divided into **logical segments** like code, data, stack.
- Each segment can grow independently.
- Can cause **external fragmentation**.

c. Demand Paging

- Only **load pages when needed** (on demand).
- Reduces **memory load** and improves efficiency.

---

4. Page Fault

- Occurs when the process tries to access a **page not in RAM**.
- OS uses a **page replacement algorithm** (FIFO, LRU, OPT, LFU) to free a frame and bring in the required page.

---

5. Advantages

- Runs **large programs** without requiring equivalent RAM.
- Improves **CPU utilization** by keeping active pages in RAM.
- Supports **process isolation** and security.
- Enables **multiprogramming**.

---

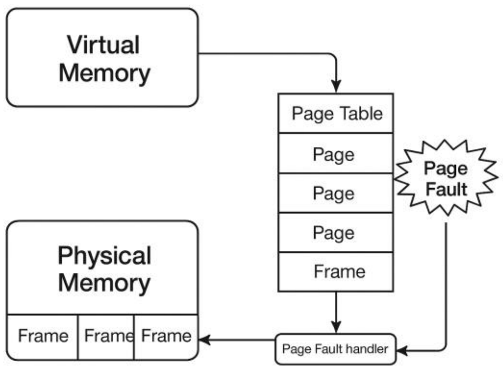
6. Disadvantages

- Accessing disk is much slower than RAM → **performance penalty**.
- If too many page faults occur → **thrashing** (CPU spends more time swapping pages than executing).

💡 **Analogy:**
Virtual memory is like **having a desk and a filing cabinet**:

- Your desk = RAM (fast access, limited space)

- Filing cabinet = Disk (slower, large space)
- You only keep the papers you are currently working on on the desk, and swap others from the cabinet when needed.



16. Swapping is a **memory management technique** used in operating systems (OS) to manage processes in main memory and optimize CPU utilization. Let me explain it step by step.

Definition

Swapping is the process of temporarily moving a process **from main memory (RAM) to secondary storage (usually disk)**, and later bringing it back into memory for execution. This allows the OS to free up memory for other processes.

Purpose of Swapping

1. **Efficient memory utilization**: Keeps the CPU busy by allowing other processes to run when memory is limited.
2. **Multiprogramming**: Enables multiple processes to share memory, even if RAM is insufficient.
3. **Process suspension**: Helps in handling long-running or idle processes by moving them out temporarily.

How Swapping Works

1. A process is **loaded into memory** from disk to start execution.
2. If another process needs memory and there is none available, the OS may **swap out** a process from RAM to disk (usually to a **swap space** or **page file**).
3. Later, the swapped-out process can be **swapped in** from disk back into RAM when needed.
4. The OS maintains a **process table** to track where each process resides (memory or disk).

Diagram

```
+-------------------+
|   Main Memory   |
+------------------+
| Process A (RUN)  |
| Process B (RUN)  |
+------------------+
    ⇅ Swap
+------------------+
|  Secondary Disk  |
+------------------+
| Process C (SWAP)  |
+------------------+
```

Characteristics

- Swapping is **time-consuming** because disk I/O is much slower than memory access.
- It requires **OS support**, like a swap space or swap partition.
- It allows **preemption** of processes in memory to accommodate higher-priority processes.

Example

Suppose your RAM can hold only **2 processes** at a time, but **3 processes** are ready to run:

1. Load Process P1 and P2 into memory.
2. P3 arrives → RAM is full → swap out P1 to disk.
3. Load P3 into memory.
4. Later, when P1 is scheduled again → swap out P2 or P3 → bring P1 back into memory.

Pros and Cons

Pros	**Cons**
Better CPU utilization	Swapping overhead (slow disk access)
Supports multiprogramming	Increased latency for swapped processes
Helps manage memory dynamically	Thrashing can occur if swapping is excessive

---

In short, **swapping allows the OS to run more processes than physical memory would normally allow**, but it can slow down the system if overused.

17. **Definition**

Hit Ratio (HR) is the fraction (or percentage) of memory references that are **found in the main memory (RAM or cache)** without requiring access to slower secondary storage (disk).

It measures **how effectively a memory or cache system is performing**.

Formula

Hit Ratio=Number of HitsTotal Memory Accesses\text{Hit Ratio} = \frac{\text{Number of Hits}}{\text{Total Memory Accesses}}Hit Ratio=Total Memory AccessesNumber of Hits

- **Hit:** When the data requested by the CPU is found in memory.
- **Miss:** When the data is **not in memory**, and the OS must fetch it from disk.

Miss Ratio=1−Hit Ratio\text{Miss Ratio} = 1 - \text{Hit Ratio}Miss Ratio=1−Hit Ratio

Example

Suppose a CPU accesses memory **100 times**, and **80 of those accesses** are found in RAM:

\text{Hit Ratio} = \frac{80}{100} = 0.8 \text{ (or 80%)} \text{Miss Ratio} = 1 - 0.8 = 0.2 \text{ (or 20%)}

When a CPU accesses memory:

- If the data is **in RAM (hit)** → access time = **memory access time (MA)**
- If the data is **not in RAM (miss / page fault)** → access time = **memory access time + time to fetch from disk (swap time)**

The **Effective Memory Access Time (EMAT)** combines both cases weighted by **hit ratio (h)** and **miss ratio (1 – h)**:

EMAT=(h×Memory Access Time)+((1−h)×Page Fault Time)

18.

In **Operating Systems**, **fragmentation** is a problem that occurs when **memory is used inefficiently**, leaving small, unusable gaps that prevent new processes from being loaded even though there is enough total free memory. Let's break it down.

Definition

Fragmentation is the phenomenon where **free memory is broken into small pieces**, which cannot be used efficiently by processes that require contiguous memory.

Types of Fragmentation

1. External Fragmentation

- Occurs in **main memory** when free memory is **split into small blocks** scattered between allocated blocks.
- Even if the total free memory is enough for a process, it **cannot be allocated** if the block is too small.

- Common in **contiguous memory allocation**.

Example:

Memory blocks (in KB):

[10][20][15][5][30] <- allocated/free blocks

If a process needs 25 KB, there is **enough total free memory (20+15+5+30)**, but **no single contiguous block of 25 KB** → external fragmentation occurs.

---

2. Internal Fragmentation

- Occurs when **allocated memory is larger than the requested memory**, leaving **unused space inside the allocated block**.
- Common in **fixed-size partitions or paging with larger page frames**.

Example:

- Process requests 18 KB, but memory is allocated in 20 KB blocks → **2 KB wasted** inside → internal fragmentation.

Difference Between External and Internal Fragmentation

Aspect	**External Fragmentation**	**Internal Fragmentation**
Location of wasted space	Outside allocated memory blocks	Inside allocated memory blocks
Cause	Variable-sized allocation	Fixed-sized allocation
Solution	Compaction, paging, segmentation	Better partition sizing

---

Solutions

1. **Compaction** – Move allocated blocks together to create larger contiguous free blocks.
2. **Paging** – Divides memory into fixed-size pages, eliminating external fragmentation.
3. **Segmentation** – Can reduce internal fragmentation but may still have external fragmentation.