**Record 1:**

* The CPU cannot directly access data from secondary storage (such as HDDs or SSDs) because they are too slow compared to the CPU's processing speed. Instead, instructions and data must first be **loaded into RAM** (main memory), which is much faster and allows the CPU to fetch and execute instructions efficiently.

### **Here's how it works:**

* + **Fetching from Storage**: When a program is executed, its instructions are copied from secondary storage (HDD/SSD) into RAM.
  + **Fetching to CPU**: The CPU fetches instructions from RAM using the **memory bus**.
  + **Decoding & Execution**: The CPU decodes the instruction and executes it.
  + **Storing Results**: Any required data is temporarily stored in RAM or registers for fast access.
* The name **Random Access Memory (RAM)** was introduced to distinguish it from older, **sequential** memory storage technologies like **magnetic tape** and **drum memory**.
* **Random Access (RAM)**The CPU can access any memory location **directly** and in **constant time** (O(1)).
  + No need to go through data sequentially.
  + Faster and more efficient for modern computing.
* **Sequential Access (Tape, Drum Memory)**
  + Data must be accessed **in order**, one after another.
  + To get to a specific piece of data, you must go through all the previous ones.
  + Much **slower** than RAM.

### **Why Does This Matter?**

* + Older computers used tape storage, making data retrieval slow.
  + RAM allows the CPU to work efficiently by directly fetching instructions and data.
  + This concept also applies to **HDDs vs. SSDs**, where HDDs have slower seek times compared to SSDs, which use **flash memory** (a form of RAM-like storage).
* The **Operating System (OS) works as an abstraction layer/ bridge** between the hardware and the user/software, making it easier to interact with the computer without dealing with complex hardware operations.
  + he **Operating System (OS)** is responsible for **resource management**, ensuring efficient use of hardware resources while avoiding conflicts and prioritizing tasks.
* The **OS is just software**, so it needs to be **loaded into memory (RAM)** before it can run. This is done by the **BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface)** using a process called **bootstrapping** (boot process).
  + **How the OS Loads into Memory (Boot Process)**
* **Power On & BIOS/UEFI Execution**
  + When you turn on the computer, the **BIOS/UEFI** is executed from **ROM (Read-Only Memory)** or **flash memory**.
  + It performs a **Power-On Self-Test (POST)** to check the hardware (CPU, RAM, disk, etc.).
* **Bootloader Execution**
  + The BIOS/UEFI locates the **bootloader** (e.g., GRUB, Windows Boot Manager) from the storage device.
  + The bootloader is then loaded into **RAM** and executed by the **CPU**.
* **Loading the OS into RAM**
  + The bootloader loads the **OS kernel** from the disk into RAM.
  + The CPU begins executing the kernel instructions, initializing system processes.
* **Hardware & Driver Initialization**
  + The OS detects connected hardware and loads the necessary **drivers**.
  + Example: Keyboard, mouse, storage, display, network, etc.
* **User Interface & Ready State**
  + The OS sets up the **Graphical User Interface (GUI)** or **Command Line Interface (CLI)**.
  + The system is now ready for user interaction!
* **OS Categorize:**

### **1. Batch Operating System**

* Processes are **executed in batches** without user interaction.
* Jobs are collected, grouped, and executed **one after another** until completion.
* **No real-time user input**—users submit jobs and wait for output.
* **Example:** Early mainframe OS, IBM’s batch processing systems.

### **2. Time-Sharing Operating System (Multitasking OS)**

* Allows multiple users or processes to **share CPU time** using **time-slicing**.
* The OS uses **multiplexing** to **switch between tasks quickly**, making it appear that multiple processes run simultaneously.
* **Examples:** Windows, Linux, macOS.

### **3. Parallel Operating System**

* Used in **multi-CPU systems** to execute multiple tasks **in parallel**.
* Improves performance by dividing workloads among **multiple processors**.
* **Examples:** High-performance servers, supercomputers (e.g., IBM Blue Gene, clusters running Linux).

### **4. Network Operating System (NOS)**

* Designed for **networked systems**, allowing multiple computers to **work together**.
* Provides **file sharing, remote access, and communication protocols** (e.g., TCP/IP).
* Used in **client-server** and **peer-to-peer** architectures.
* **Examples:** Windows Server, Linux Ubuntu Server, Novell NetWare.

### **5. Real-Time Operating System (RTOS)**

* Designed for **real-time, time-sensitive tasks**.
* Used in **embedded systems**, robotics, and industrial control.
* **Two types:**
  + **Hard Real-Time OS**: Strict timing constraints (e.g., medical devices, avionics).
  + **Soft Real-Time OS**: Timing is important but not critical (e.g., video streaming, gaming consoles).
* **Examples:** FreeRTOS, QNX, VxWorks.

### **Conclusion**

* **Batch OS** → No user interaction, jobs executed sequentially.
* **Time-Sharing OS** → Multiple users/processes get CPU time using time-slicing.
* **Parallel OS** → Uses multiple CPUs for increased performance.
* **Network OS** → Supports communication and resource sharing in a network.
* **Real-Time OS** → Used in time-sensitive embedded systems.
* **The Operating System (OS) acts as an abstraction layer** between hardware and software, managing resources and ensuring smooth operation.

### **OS as the Middle Layer (Abstraction Layer)**

* It sits **between the hardware (lower layer) and user applications (upper layer)**.
* It provides **interfaces** for applications to interact with hardware **without dealing with complexities** like CPU scheduling, memory management, and device handling.

### **Layers of an Operating System**

1. **Hardware Layer (Lowest Layer)**
   * Includes CPU, RAM, storage, I/O devices.
   * Directly interacts with the OS through device drivers.
2. **Operating System Layer (Middle Layer)**
   * **CPU Scheduler**: Decides which process gets CPU time.
   * **Memory Manager**: Allocates RAM efficiently.
   * **File System**: Manages data storage.
   * **I/O Management**: Handles communication with peripherals.
3. **Application Layer (Upper Layer)**
   * Includes user applications like browsers, games, and office software.
   * Uses **APIs (Application Programming Interfaces)** to request OS services.

* As a **developer**, understanding the **Operating System (OS)** is crucial because different OS environments (Windows, Linux, macOS) **handle system resources differently**. When writing applications, like Paths Difference conventions.

**Record 2:**

* When we are dealing with processes in our computer, the OS handles them using the **scheduler.** What is actually done is prioritizing a single task, and after finishing it, it goes to the next one at a fast speed. We can not notice that.
  + The **OS scheduler** manages processes by rapidly switching between them, creating the illusion that multiple tasks are running simultaneously (even on a single-core CPU).
* Text section
  + Program instructions: where the written code is.
* Program counter
  + Next instruction address: track the line of code for execution.
* Stack Data Section
  + Local variables
  + Return addresses
  + Method parameters
* Heap Data section
  + Global variables
* Process Control Block(PCB): is the OS way to handle and track the different processes while executing where each process has its PCB.
  + **Process Control Block (PCB)** is a data structure used by the OS to manage and track processes during execution. Each process has its own PCB, which stores essential information such as the **process ID (PID), process state, CPU registers, memory allocation, scheduling information, and I/O status**. The OS uses the PCB to switch between processes efficiently, ensuring proper execution and resource management.
* Virtual memory is used for stretching the memory using a hard disk to handle the idle and non-running process with no need for the main memory
  + **Virtual memory** extends the available RAM by using a portion of the hard disk (or SSD) as **temporary memory storage**. It helps manage idle or less frequently used processes by moving them from **main memory (RAM) to disk storage**, allowing the system to run more applications than the physical RAM can hold. This process, called **paging or swapping**, ensures efficient memory management but may slow down performance due to the slower speed of disk access compared to RAM.

**Record 3:**

* The **Interrupt Vector** is a table stored in memory that helps the OS handle interrupts efficiently. It consists of two columns: one for the **interrupt ID** and the other for the **address of the corresponding interrupt service routine (ISR)**. When an interrupt occurs, the CPU checks the interrupt vector to find the appropriate ISR, ensuring quick and organized handling of hardware and software interrupts.