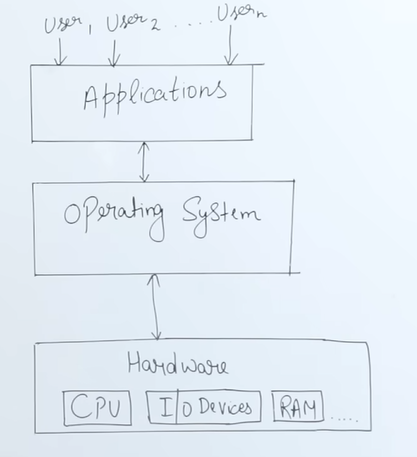
**Operating System :-**

1. Basic Introduction: - Type of OS, process diagram, System calls ,
2. Process Scheduling: - FIFOA, SJF, PReamtive, Round Robin Algorithms
3. Process Synchronization: - Semaphore,
4. Deadlock and Threads: - Bankers Algorithm
5. Memory Management: - Virtual Memory, Paging, Segmentation, Fragmentation, Page Replacement Algos.
6. Disk Scheduling: - SCAN, CSCNA, FCFS Algos
7. Unix Commands: - Sequential access, random access, linked access algos
8. File Management and Security.

**[LEC-1]**

**Introduction to Operating System and its Functions: -** An operating system is a piece of software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

****

**Why OS Important: -**

1. **What if there is no OS?**

**a. Bulky and complex app. (Hardware interaction code must be in app’s code base)**

**b. Resource exploitation by 1 App.**

**c. No memory protection.**

**2. What is an OS made up of?**

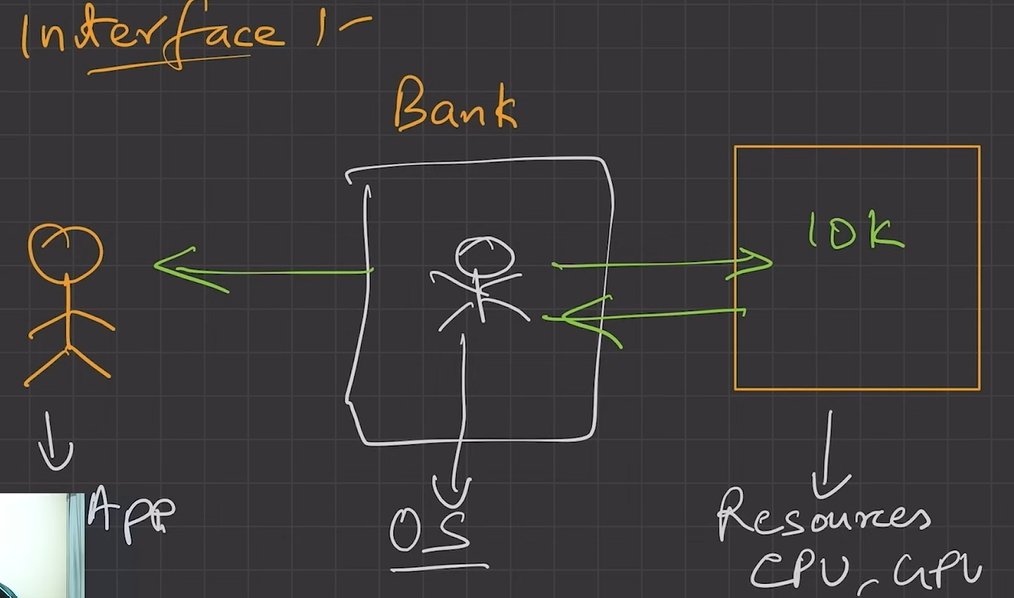
**a. Collection of system software.**

Let's take an example: Suppose we need to run an application like TikTok. To do this, we require CPU, GPU, memory, and disk resources.

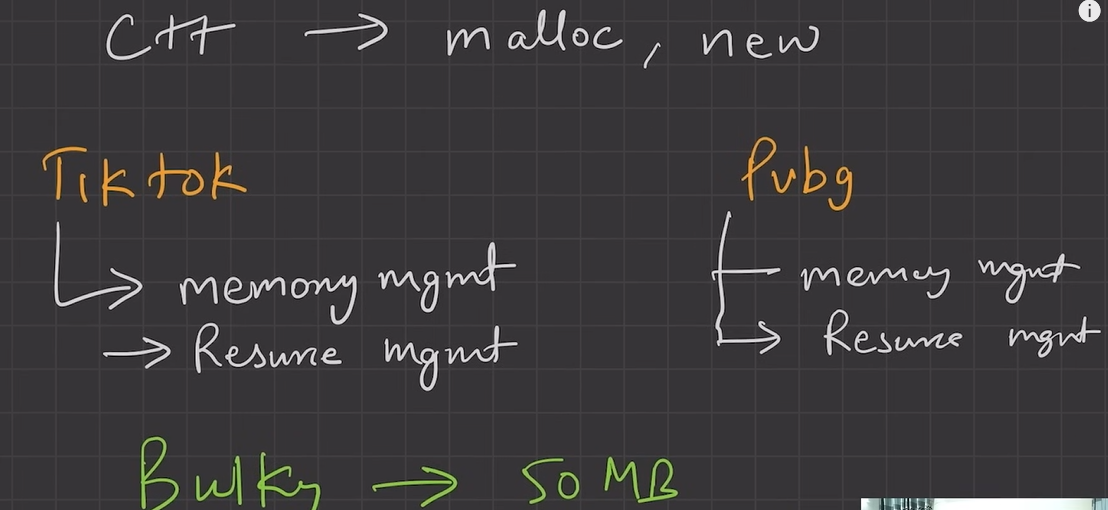
If there were no operating system (OS), TikTok would directly access the CPU, memory, GPU, and disk, consuming all available resources. Now, if we wanted to run another application, like PUBG, at the same time, it wouldn’t be possible because TikTok would have already occupied all the resources.

The OS solves this problem by allocating a specific portion of the CPU, memory, and GPU to TikTok while reserving some resources for other applications. This allows us to run PUBG simultaneously by distributing resources efficiently. This process is known as **resource management by the OS**.

The OS functions as an interface, similar to how a bank cashier facilitates cash withdrawals. When I go to a bank to withdraw money from my account, I first fill out a form and submit it to the cashier. The cashier then retrieves the cash from the bank and hands it over to me. In this scenario, the cashier acts as an interface between me and the bank, just as the OS serves as an interface between users and the computer's hardware.



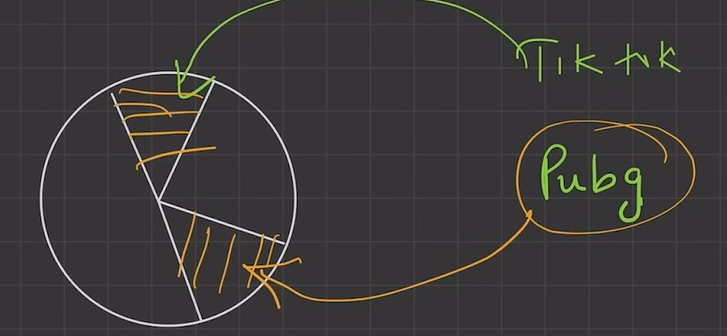
Without an OS, developers would need to write code for resource management, such as memory and CPU management, for applications like TikTok and PUBG. This would result in large, redundant code, as the same functionalities would have to be implemented repeatedly, violating the DRY (Don't Repeat Yourself) principle.

****

**Dry Principle: -** Do not repeat yourself: - The **DRY (Don't Repeat Yourself) Principle** is a software development concept that emphasizes reducing code duplication. It promotes reusability and maintainability by ensuring that a piece of logic or functionality is written only once and reused wherever needed (i.e Functions) . This helps prevent inconsistencies, reduces errors, and makes code easier to update and manage.

To stop DRY Principal violation, we need Operating System. We don’t need to write code for Resource management.

**ISOLATION & PROTECTION: -** The OS ensures isolation and protection. Without an OS, if applications like TikTok and PUBG are running and consuming memory, they could accidentally overwrite another application's memory, compromising security and leading to potential system instability.

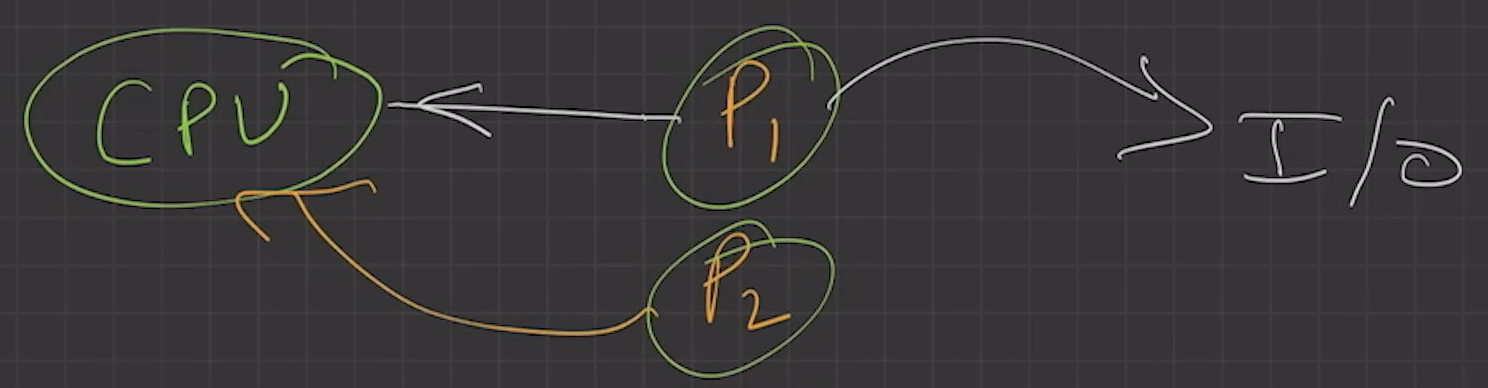
****

Imagine you are playing PUBG with your player's health at 100%. Suddenly, TikTok runs and accidentally changes your health to 0%, instantly killing your player. This happens because there’s no OS to keep applications separate, allowing one app to interfere with another, which is a security breach.

**[LEC-2]**

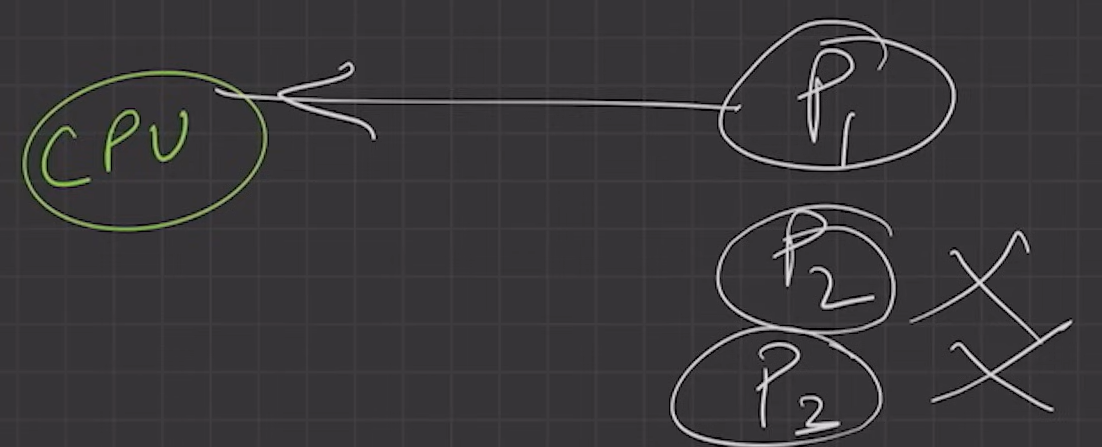
**OS goals –**

* **Maximum CPU utilization: -**

****

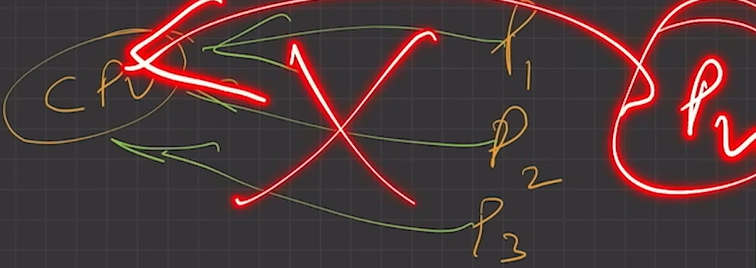
There are two processes, P1 and P2. When P1 is using the CPU and later needs to perform input/output (I/O) operations, the CPU switches to P2 so that it doesn't sit idle. This helps in making the best use of the CPU.

* **Less process starvation: -**

****

If P1 keeps using the CPU for a long time and P2 and P3 don’t get a turn, it is called process starvation. This means some processes are stuck waiting while one process keeps running.

* **Higher priority job execution: -**



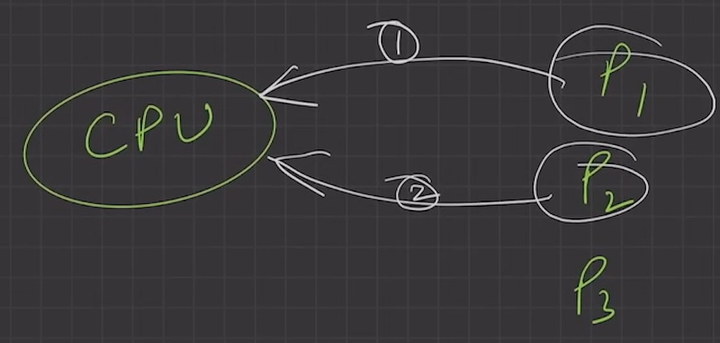
If multiple processes are running on the CPU and a high-priority task arrives, the CPU should pause the others and run the high-priority task first.

**Types of operating systems –**

* Single process operating system
* Batch-processing operating system
* Multiprogramming operating system
* Multitasking operating system
* Multi-processing operating system
* Distributed system
* Real time OS

1. **Single process OS**, only 1 process executes at a time from the ready queue. [Oldest – MS DOS]

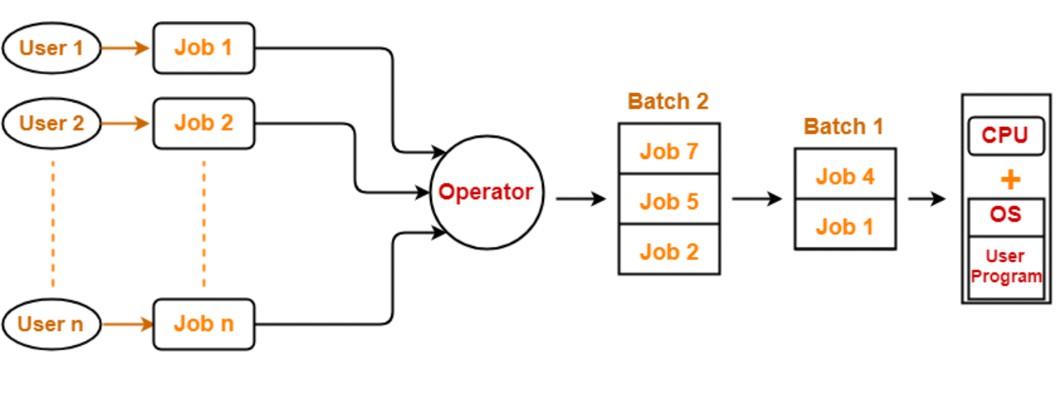
### In a single-process OS, only one process runs at a time. For example, if there are three processes—P1, P2, and P3—first, P1 runs. After P1 finishes, P2 starts. Once P2 is done, P3 begins.



The disadvantage of this OS is that it doesn’t fully utilize the CPU, can cause process starvation, and doesn’t allow high-priority tasks to run first.

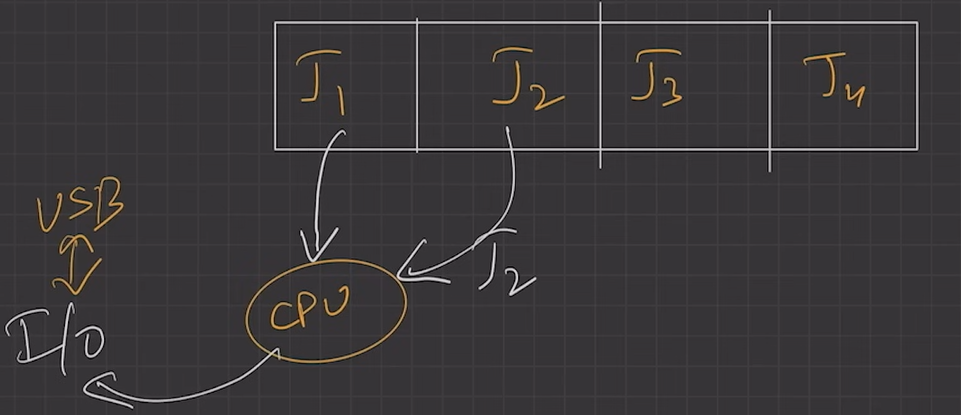
### **Batch-processing OS,**

1. Firstly, user prepares his job using **punch cards**.
2. Then, he submits the job to the computer operator.
3. Operator collects the jobs from different users and sort the jobs into batches with similar needs.
4. Then, operator submits the batches to the processor one by one.
5. All the jobs of one batch are exec uted together.
   * Priorities cannot be set, if a job comes with some higher priority.
   * May lead to starvation. (A batch may take more time to complete)
   * CPU may become idle in case of I/O operations.



1. **Multiprogramming** increases CPU utilization by keeping multiple jobs (code and data) in the memory so that the CPU always has one to execute in case some job gets busy with I/O.
   * Single CPU
   * Context switching for processes.
   * Switch happens when current process goes to wait state.
   * CPU idle time reduced.

In a multiprogramming OS, there is only one CPU, but it uses a **ready queue** to hold multiple jobs. If one process goes into waiting or needs I/O, the CPU picks another job to run, keeping the system active.



* + **Context switching for processes. :-**

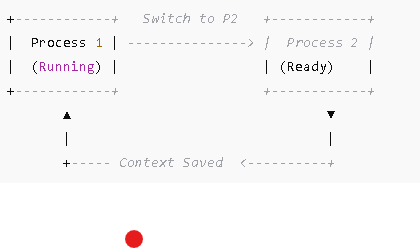
Context switching is when the CPU switches from one process to another. This happens when multiple processes are running, and the CPU needs to stop one process and start another. The OS saves the state of the current process and loads the state of the next process so it can continue from where it left off.

Example:

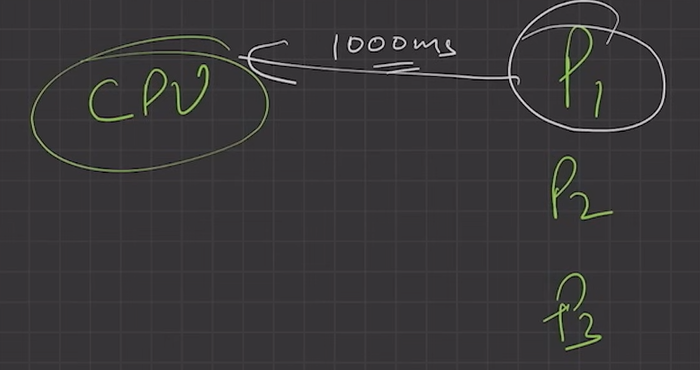
Imagine you are watching a video on your phone, and suddenly you get a call. Your phone pauses the video (saves its state) and switches to the call. After the call, you go back to the video, and it resumes from where it stopped.

Steps in Context Switching:

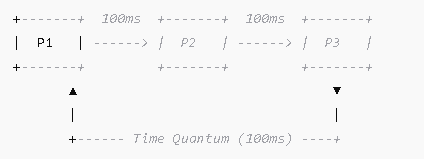
1. **Save the current process state (PCB – Process Controll Block) .**
2. **Load the new process state from memory.**
3. **Start executing the new process.**



1. **Multitasking** is a logical extension of multiprogramming.
   1. Single CPU
   2. Time Sharing (Time Quantum i.e 100 ms)
   3. Able to run more than one task simultaneously.
   4. Context switching and time sharing used.
   5. Increases responsiveness.
   6. CPU idle time is further reduced.



In this diagram, we show **time-sharing**, where each process gets a fixed amount of time to execute. Suppose **P1** starts running and gets **100 ms** to complete its work. After **100 ms**, the CPU switches to **P2** for another **100 ms**, and then to **P3**. This fixed time slice is called a **time quantum**.



1. **Multi-processing OS,** more than 1 CPU in a single computer.
   * 1. Increases reliability, 1 CPU fails, other can work
     2. Better throughput.
     3. Lesser process starvation, (if 1 CPU is working on some process, other can be executed on other CPU.

### **Distributed OS,**

* OS manages many bunches of resources,

>=1 CPUs, >=1 memory, >=1 GPUs, etc

- Loosely connected autonomous,

- interconnected computer nodes.

* collection of independent, networked, communicating, and physically separate computational nodes.

### **RTOS**

* Real time error free, computations within tight-time boundaries.
* Air Traffic control system, ROBOTS etc.