1 Introduction Operating System

1.1 What's An Operating System ?

An operating system (OS) is a software program designed to manage and optimize the use of a machine's resources. It provides an easy to use **VM** virtual Machine to interact with the hardware and efficiently execute tasks. A typical machine consists of three main components:

- **Memory:** Includes random access memory (RAM), registers, and storage devices such as hard drives and solid-state drives (SSD).
- **CPU:** The central processing unit (CPU) consists of the Arithmetic Logic Unit (ALU) and the Control Unit (CU), which process and execute instructions.
- **Peripherals:** External devices such as keyboards, mice, displays, printers, and storage media.

1.2 Types Of Operating Systems

1.2.1 Batch Operating System

In a batch operating system, the user does not directly interact with the machine. Instead, the user prepares a stack of punch cards, referred to as **jobs**, which represent the user's programs. The operating system collects these jobs and **sorts them into batches** based on certain criteria, such as similar I/O requirements or resource needs. These batches are then processed sequentially, one after the other.
The main limitation of this system is that **it can only execute one job at a time** , since the CPU only loads one programe in the memory then execute it free the space and loads the next one and so on. If a job is waiting for an I/O operation (e.g., reading from a disk), the CPU enters an **idle state**, leaving its full potential and resources underutilized.

1.2.2 Multi-Programmed Operating System

Similar to a batch operating system, the user still submits jobs to the machine for execution. However, the key difference is that in a multi-programmed operating system, the CPU **loads multiple programs into memory at the same time**.
When the CPU encounters an I/O operation in Job 1, instead of going idle, it can **switch to Job 2 and continue executing instructions**. This allows the CPU to be **fully utilized**, as it can work on other jobs while waiting for I/O operations to complete. By handling multiple jobs concurrently, the system improves overall efficiency and reduces CPU idle time.

1.2.3 Multi-Processing Operating System

Similar to batch and multi-programmed operating systems, users still provide jobs for the machine to execute. However, unlike the previous systems, a multi-processing operating system has **multiple CPUs connected to the same system**, allowing for the **parallel execution** of programs. In some cases, the execution of a **single program can be split across multiple CPUs** . Like the multi-programmed operating system, this system also benefits from minimizing idle time since multiple CPUs can work on **different tasks simultaneously**. Additionally, having multiple CPUs provides **fault tolerance** if one CPU fails, the system can continue functioning with the remaining CPUs, ensuring greater reliability.

1.2.4 Distributed Operating System

In this setup, a set of machines is connected to each other through a local area network (**LAN**). The **parts of the operating system and tasks are spread among the machines** rather than being centralized. If one of the machines fails, the others will detect it and finish any incomplete tasks. The role of the failed machine will then be assigned to another machine, ensuring that the OS is always running. The crucial data is **available on all machines to prevent loss** and provide even more **robust fault tolerance**, ensuring high availability and reliability for users and applications.

1.2.5 Multi-tasking Operating System

Unlike Batch, Multi-programmed, and Multi-processing operating systems that rely on jobs and don't allow direct interaction between the user and the machine, Multi-tasking OS allows users to interact with the machine in real-time. It can be seen as an extension of the Multi-programmed OS. In a multi-tasking system, the CPU loads all programs into memory at once and switches between them rapidly, creating the illusion of parallel execution. However, in reality, the CPU is handling only one program at a time, switching between tasks so fast that it appears as if they are running simultaneously. The operating system ensures that the CPU is fully utilized and doesn't remain idle.

1.2.6 Time Sharing Operating System

In a Time Sharing Operating System, much like Multi-tasking OS the user directly interact with the machine . the CPU allocates a specific time slice to each program or task. Once the allocated time expires, the CPU switches to the next program in a cyclic manner, allowing multiple programs to share the CPU effectively. This process continues until all programs have completed their execution. The main goals of a time-sharing system are to provide interactive user experiences, ensure that all programs receive fair access to CPU resources, and minimize the time users spend waiting for responses from the system.

1.2.7 Real Time Operating System

A Real-Time Operating System (RTOS) is designed to prioritize and execute tasks within strict time constraints, known as deadlines. It loads multiple tasks into memory and manages their execution on the CPU(s). RTOS allows direct interaction with external inputs, including users or devices, ensuring that critical tasks are completed on time.
RTOS can either:

- Multi-task : by using a single CPU that rapidly switches between tasks (similar to a Multi-Tasking OS) .
- Multi-process : by using multiple CPUs to run tasks in parallel (similar to a Multi-Processing OS).

The key feature of an RTOS is that it guarantees timely execution, making it suitable for systems where **meeting deadlines is crucial** , such as in embedded systems, medical devices, or automotive control systems.

1.3 Core Functions Of An Operating System

1.4 History And Evolution Of Operating System

Chapter 5: Deadlock

1 Processes & Resources

S

Processes in an **(OS)** use various resources such as peripherals, variables, and files, following these steps:

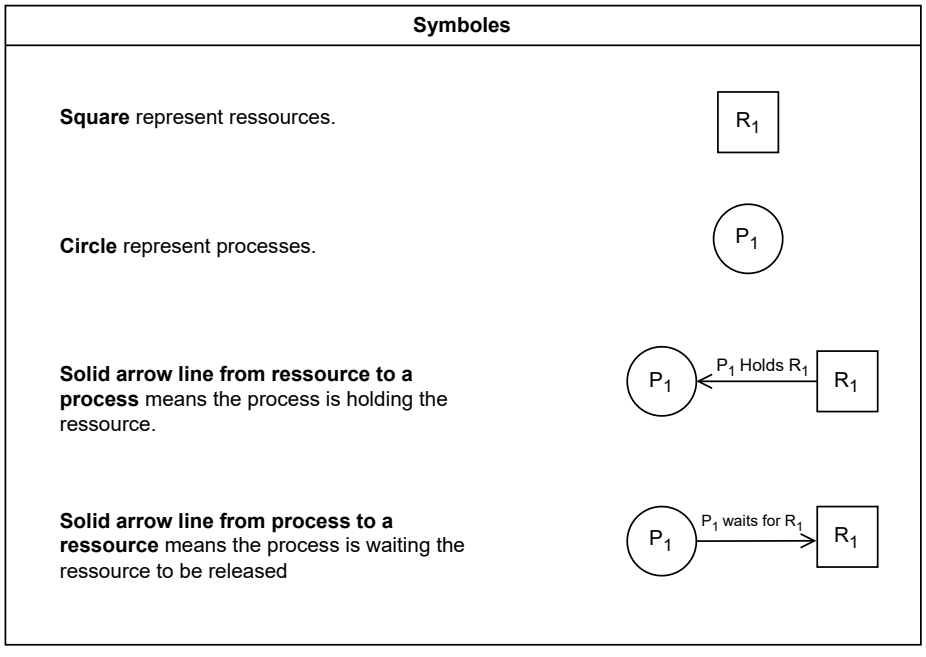
1. Request a resource, if it is unavailable the process is put on wait.
2. Hold and use the resource.
3. Release the resource once the operation is completed.

2 Deadlock

What is Deadlock?

Deadlock is a situation where a set of processes becomes blocked because each process is holding a resource and waiting for another resource held by another process to be released.

3 Resource-Process Graph

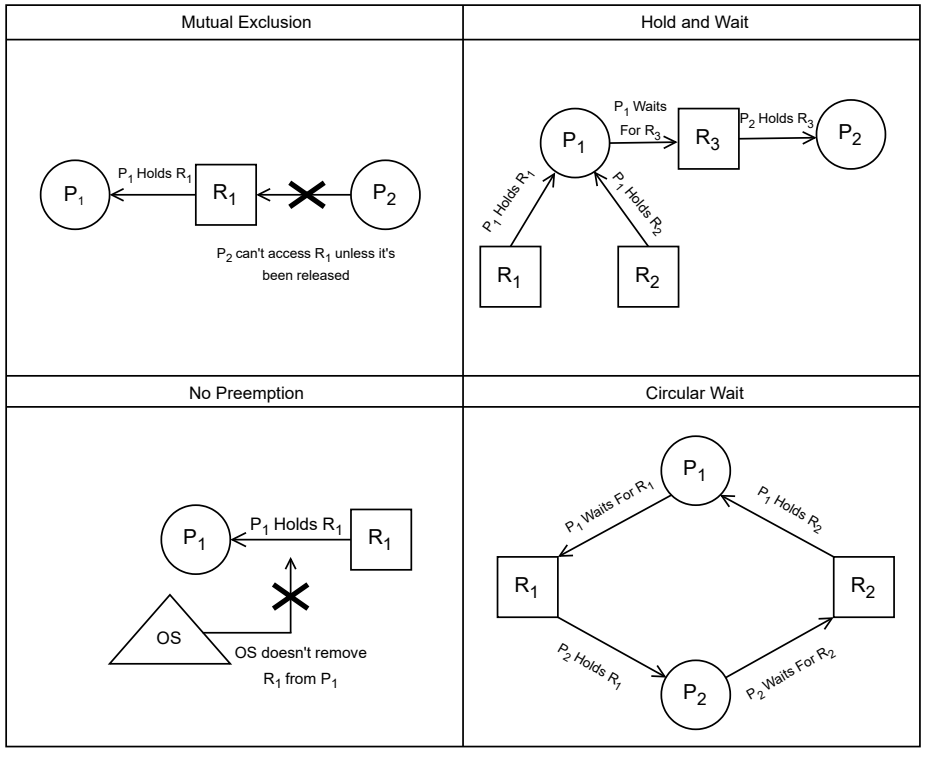


4 Conditions for Deadlock

Coffman Conditions

Deadlock can arise if all the following conditions are met:

- **Mutual Exclusion**: Only one process can hold a resource at a time.
- **Hold and Wait**: A process holding at least one resource is waiting for additional resources held by other processes.
- **No Preemption**: The **OS** cannot forcibly remove a resource from a process.
- **Circular Wait**: There is a cycle in the graph of two or more processes waiting for each other to release ressources.



5 Handling Deadlocks

Methods

- **Ignore the Problem**: If resolving deadlocks is too costly, the **OS** may adopt a laissez-faire approach, ignoring deadlocks and requiring a system reboot if necessary.
- **Detection and Resolution**: The system detects deadlocks, often using algorithms like **DFS** on the resource allocation graph, and resolves them through one of the following strategies:
  - **Preemption**: Reassign a resource from one process to another, which may lead to issues like inconsistency or lost progress.
  - **Rollback**: Restore the system to a previously saved state and reallocate resources to avoid deadlock.
  - **Termination**: Kill one or more processes involved in the deadlock to free resources.
- **Avoidance**: Dynamically allocate resources in a way that avoids unsafe states, using the **Banker's Algorithm** to check the safety of each potential allocation before it is made.
- **Prevention**: Prevent deadlocks by ensuring at least one of Coffman's conditions cannot occur:
  - **Mutual Exclusion**: Use serialized access (e.g., queues) to eliminate contention by ensuring resources are accessed in an orderly fashion. This doesn't remove mutual exclusion, but it allows the system to control priority, avoiding chaotic competition between processes.
  - **Hold and Wait**: Require processes to request all the resources they need at the beginning. While this prevents deadlocks, it can lead to inefficiency because resources may remain idle, and a resource may need to be released to be used by another.
  - **No Preemption**: Allow preemption for certain resources by forcibly reallocating them. This approach is rarely practical due to the complexity of saving and restoring resource states.
  - **Circular Wait**: Resources are assigned increasing numerical labels. Processes must request resources in increasing order of their labels. If a process needs a previously allocated resource, it must first release any resources with higher labels before requesting the lower-numbered one. This ensures that no cycles can form in the resource allocation graph, preventing deadlock.