MATHEMATICAL AND COMPUTING SCIENCE DEPARTMENT

PRINCIPLES OF OPERATING SYSTEM

LECTURE NOTE

AYEPEKU F.O

Table of Contents

Chapter 6: I/O Management

- Basics of I/O devices and operations
- I/O device management techniques (polling, interrupts, DMA)
- I/O scheduling algorithms (FCFS, SSTF, SCAN)
- Device drivers and I/O subsystem organization

Chapter 7: Deadlock Avoidance

- Understanding deadlock and its causes
- Detection and recovery techniques for deadlock
- Deadlock prevention strategies (resource allocation graphs, Banker's algorithm)
- Deadlock avoidance using dynamic allocation and ordering resources

CHAPTER ONE

Introduction to Operating Systems

Definition and Purpose of Operating Systems:

An operating system (OS) is a software layer that acts as an intermediary between computer hardware and user applications. It manages hardware resources, provides essential services to applications, and facilitates user interaction with the computer system. The primary purposes of an operating system include:

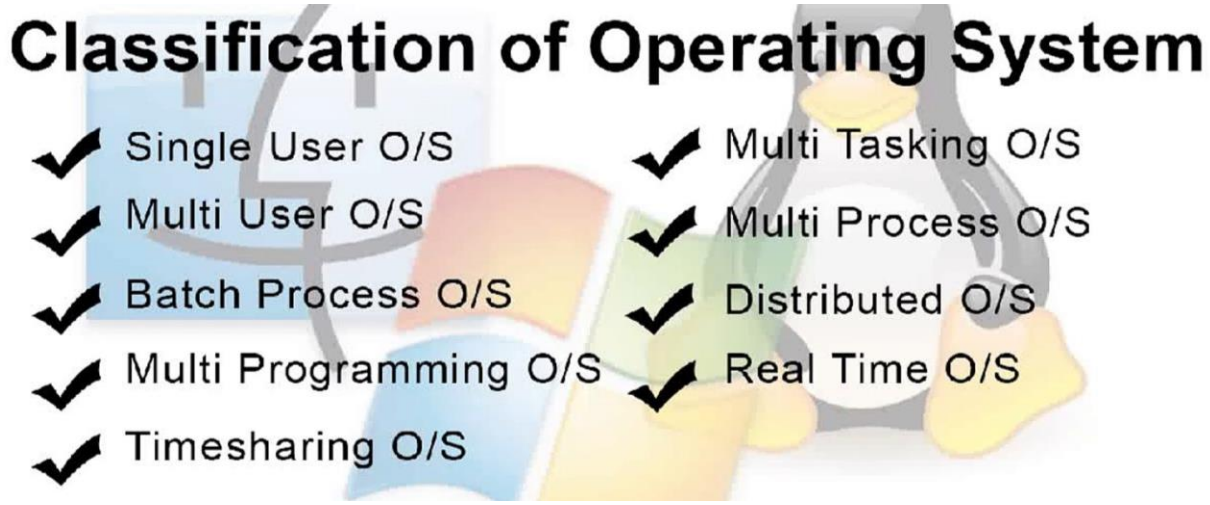
1. **Resource Management:** The OS manages hardware resources such as CPU, memory, storage devices, and input/output (I/O) devices. It allocates these resources efficiently among different processes and users to ensure optimal performance and utilization.
2. **Process Management:** It facilitates the creation, scheduling, and execution of processes or programs. The OS manages process synchronization, communication, and inter-process coordination to ensure smooth operation and prevent conflicts.
3. **Memory Management:** The OS allocates and deallocates memory space for processes, manages virtual memory, and handles memory protection to prevent unauthorized access and ensure data integrity.
4. **File System Management:** It provides a hierarchical structure for organizing and storing files on storage devices. The OS handles file creation, deletion, access control, and data storage/retrieval operations.
5. **Device Management:** The OS controls communication with input/output devices such as keyboards, mice, printers, and network interfaces. It provides device drivers and manages device access, ensuring efficient data transfer and device utilization.
6. **User Interface:** The OS provides a user-friendly interface for interacting with the computer system. This can include command-line interfaces (CLI), graphical user interfaces (GUI), or a combination of both, depending on the OS and user preferences.
7. **Security and Protection:** It implements security measures to protect system resources, data, and user privacy. This includes user authentication, access control mechanisms, encryption, and malware detection/prevention.

Evolution of Operating Systems:

The evolution of operating systems can be traced through several key stages, each marked by significant advancements in technology and functionality:

1. **Serial Processing Systems:** Early computers operated in a serial processing mode, executing one program at a time without multitasking capabilities. Examples include the ENIAC and UNIVAC systems.
2. **Batch Processing Systems:** Batch processing systems emerged in the 1950s, allowing multiple jobs to be executed sequentially without user intervention. Programs were submitted in batches, and the OS managed job scheduling, resource allocation, and I/O operations. IBM's OS/360 is a notable example of a batch processing OS.
3. **Time-Sharing Systems:** Time-sharing systems, developed in the 1960s, enabled multiple users to interact with a computer simultaneously. These systems divided CPU time among multiple processes, providing each user with the illusion of having a dedicated computer. The introduction of interactive terminals and multi-user operating systems like CTSS (Compatible Time-Sharing System) and UNIX revolutionized computing by enabling real-time interaction and collaboration.
4. **Distributed Systems:** Distributed operating systems emerged in the 1980s, enabling the coordination and management of resources across multiple interconnected computers. Distributed OSs facilitate distributed computing, allowing tasks to be divided among networked machines for improved scalability, fault tolerance, and performance. Examples include Google's Android, Microsoft's Windows Distributed File System (DFS), and various cluster computing systems.
5. **Real-Time Systems:** Real-time operating systems (RTOS) are designed to meet strict timing constraints and deadlines in applications where timely response is critical. RTOSs prioritize tasks based on their urgency and guarantee timely execution, making them suitable for embedded systems, control systems, and mission-critical applications. Examples include VxWorks, QNX, and FreeRTOS.

Types of Operating Systems:



1. Single-User Operating System:

- **Features:**
  - Designed to support only one user at a time.
  - Primarily runs on personal computers, laptops, and workstations.
  - Provides a user-friendly interface for individual users to interact with the system.
  - Manages resources for a single user's tasks and applications.
- **Usage:**
  - Personal computing, home use, and small businesses where only one user interacts with the computer at a time.
- **Examples:**
  - Microsoft Windows: Versions of Windows such as Windows 10, Windows 11, and Windows 7 are examples of single-user operating systems widely used on personal computers and laptops.
  - macOS: Apple's macOS operating system is used on Macintosh computers and is designed for single-user environments, providing a seamless user experience with a graphical user interface.

2. Multi-User Operating System:

- **Features:**
  - Supports multiple users to interact with the system simultaneously.

- Enables concurrent execution of processes and tasks for multiple users.
- Provides user isolation and access control mechanisms to protect user data and resources.
- Facilitates resource sharing and collaboration among users.

- **Usage:**
  - Servers, mainframes, shared computing environments, and networked systems where multiple users need access to shared resources and services.
- **Examples:**
  - UNIX and UNIX-like Systems: UNIX operating systems, such as Linux and FreeBSD, are designed to support multi-user environments with features for user authentication, access control, and resource sharing. They are commonly used in server environments and shared computing systems.
  - Windows Server: Microsoft's Windows Server operating system is specifically designed for server environments and supports multiple users, applications, and services concurrently. It provides features such as Active Directory for user authentication and access control in enterprise environments.

Comparison:

- **User Interaction:**
  - In single-user operating systems, only one user interacts with the system at a time, typically through a personal computer or workstation.
  - In multi-user operating systems, multiple users interact with the system simultaneously, accessing shared resources and services over a network.
- **Resource Management:**
  - Single-user operating systems manage resources for a single user's tasks and applications, optimizing performance and responsiveness for individual use.
  - Multi-user operating systems manage resources efficiently among multiple users, providing fair access and equitable distribution of resources for shared computing environments.
- **Security and Access Control:**
  - Single-user operating systems focus on securing resources and data for a single user, often relying on user accounts and passwords for access control.

- Multi-user operating systems implement robust security measures for user isolation, access control, and data protection to ensure confidentiality, integrity, and availability in shared computing environments.

3. Batch Operating System:

- **Features:**
  - Processes jobs in batches without user intervention.
  - Sequential execution of tasks.
  - Typically used in environments where large-scale repetitive tasks need to be executed efficiently.
- **Usage:**
  - Mainframes and server environments where repetitive tasks such as payroll processing, report generation, and batch data processing are common.
- **Examples:**
  - IBM OS/360, z/OS: IBM's mainframe operating systems that support batch processing.
  - UNIX Batch System: Early versions of UNIX featured batch processing capabilities.

4. Multiprocessing Operating System:

- **Features:**
  - Supports concurrent execution of multiple processes on multiple CPUs.
  - Utilizes multiprocessing capabilities for improved performance and throughput.
  - Provides efficient task scheduling and resource management across multiple processors.
- **Usage:**
  - High-performance computing environments, servers, and workstations with multiple CPU cores.
- **Examples:**
  - Linux: A popular open-source operating system that supports multiprocessing across multiple CPU cores.

- Windows Server: Microsoft's server operating system that efficiently utilizes multiple processors for server workloads.

5. Time-Sharing Operating System:

- **Features:**
  - Allows multiple users to interact with the system simultaneously.
  - Divides CPU time among active processes or users.
  - Provides responsive and interactive user experience through time-slicing.
- **Usage:**
  - Multi-user environments such as servers, mainframes, and shared computing systems.
- **Examples:**
  - UNIX: Early versions of UNIX introduced time-sharing capabilities, allowing multiple users to access a system concurrently.
  - Linux: Modern distributions of Linux support time-sharing features for multi-user environments.

6. Multitasking Operating System:

- **Features:**
  - Enables a single user to run multiple programs concurrently.
  - Rapid context switching between tasks to provide the illusion of parallel execution.
  - Efficient utilization of CPU resources.
- **Usage:**
  - Personal computers, laptops, and workstations where users perform multiple tasks simultaneously.
- **Examples:**
  - Microsoft Windows: Windows operating systems support multitasking, allowing users to run multiple applications concurrently.
  - macOS: Apple's macOS provides multitasking capabilities for users to switch between applications seamlessly.

7. Multiprocess Operating System:

- **Features:**
  - Supports concurrent execution of multiple processes within a single computer system.
  - Manages processes effectively, whether running on a single CPU core or multiple CPU cores.
  - Provides efficient process scheduling and resource management.
- **Usage:**
  - Personal computers, servers, and workstations with multi-core processors.
- **Examples:**
  - Windows NT: Microsoft's Windows NT operating system family supports multiprocess execution across multiple CPU cores.
  - Linux: Linux kernels are designed to efficiently manage multiple processes across multi-core systems.

8. Distributed Operating System:

- **Features:**
  - Manages resources and coordinates activities across multiple networked computers.
  - Treats networked computers as a single integrated system.
  - Provides transparent access to resources across distributed nodes.
- **Usage:**
  - Networked environments, cloud computing, peer-to-peer networks, and distributed systems.
- **Examples:**
  - Google's Android: Android is a distributed operating system used in mobile devices, managing resources across various hardware components.
  - Apache Hadoop: Hadoop is an open-source distributed operating system used for distributed storage and processing of large datasets.

9. Real-Time Operating System (RTOS):

- **Features:**
    - Guarantees timely responses to events within specified time constraints.
    - Provides deterministic behavior and predictable response times.
    - Used in time-critical applications where responsiveness is crucial.
- **Usage:**
    - Embedded systems, control systems, automotive, aerospace, and industrial applications.
- **Examples:**
    - VxWorks: VxWorks is a real-time operating system commonly used in embedded systems and mission-critical applications.
    - FreeRTOS: FreeRTOS is an open-source real-time operating system designed for embedded microcontrollers and IoT devices.

Components of Operating Systems:

1. **Kernel:**
    - The kernel is the core component of an operating system, responsible for managing hardware resources and providing essential services to user programs.
    - It handles tasks such as process management, memory management, device management, and system calls.
    - The kernel operates in privileged mode, with direct access to hardware resources, and ensures proper isolation and protection between user processes.
2. **Process Management:**
    - Process management involves creating, scheduling, synchronizing, and terminating processes.
    - The OS maintains a process table containing information about active processes, including their state, priority, and resource usage.
    - It schedules processes for execution on the CPU, using algorithms such as round-robin, priority-based scheduling, or multi-level feedback queues.
    - Process synchronization mechanisms, such as semaphores, mutexes, and monitors, are used to coordinate access to shared resources and prevent race conditions.
3. **Memory Management:**

- Memory management involves managing system memory, including allocation, deallocation, and protection.
- The OS provides mechanisms for virtual memory, allowing processes to use more memory than physically available by swapping data between main memory and secondary storage.
- Memory protection mechanisms prevent processes from accessing memory locations outside their allocated address space, ensuring memory safety and security.
- Techniques such as paging, segmentation, and demand paging are used to optimize memory usage and minimize fragmentation.

4. **File System:**
   - The file system provides a hierarchical organization for storing and accessing files and directories on storage devices.
   - It manages file metadata, including file attributes (e.g., name, size, permissions, timestamps) and file allocation information.
   - The OS provides file system drivers to support different file system formats, such as FAT, NTFS, ext4, and HFS+.
   - File system operations, such as file creation, deletion, reading, and writing, are performed through system calls and file APIs.
5. **Device Management:**
   - Device management involves controlling input/output devices such as keyboards, mice, displays, printers, and network interfaces.
   - The OS provides device drivers to communicate with hardware devices and abstract device-specific details from user applications.
   - Device drivers handle device initialization, configuration, and communication, translating high-level I/O requests into device-specific commands.
   - Input/output operations are managed through device drivers and device controllers, ensuring efficient data transfer and error handling.
6. **User Interface:**
   - The user interface allows users to interact with the operating system and execute commands or applications.
   - Graphical user interfaces (GUIs) provide visual elements such as windows, icons, menus, and buttons for user interaction.

- Command-line interfaces (CLIs) allow users to interact with the system through text-based commands and shell programs.
- The OS provides user interface components, such as window managers, desktop environments, and command interpreters, to facilitate user interaction.

CHAPTER TWO

Process Management

Process

A process is a running program that serves as the foundation for all computation. The procedure is not the same as computer code, although it is very similar. In contrast to the program, which is often regarded as some 'passive' entity, a process is an 'active' entity. Hardware status, RAM, CPU, and other attributes are among the attributes held by the process.

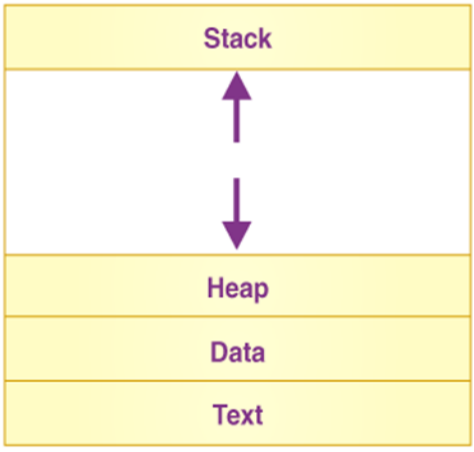
A process is essentially running software. The execution of any process must occur in a specific order. A process refers to an entity that helps in representing the fundamental unit of work that must be implemented in any system.

In other words, we write the computer programs in the form of a text file, thus when we run them, these turn into processes that complete all of the duties specified in the program.

A program can be segregated into four pieces when put into memory to become a process: stack, heap, text, and data. The diagram below depicts a simplified representation of a process in the main memory.

Components of a Process

It is divided into the following four sections:



Stack: Temporary data like method or function parameters, return address, and local variables are stored in the process stack.

Heap: This is the memory that is dynamically allocated to a process during its execution.

Text: This comprises the contents present in the processor's registers as well as the current activity reflected by the value of the program counter.

Data: The global as well as static variables are included in this section.

Program

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language −

```
#include <stdio.h>
int main() {
   printf("Hello, World! \n");
   return 0;
}
```

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

Key Components of Process Management

Below are some key components of process management.

- **Process mapping:** Creating visual representations of processes to understand how tasks flow, identify dependencies, and uncover improvement opportunities.

- **Process analysis:** Evaluating processes to identify bottlenecks, inefficiencies, and areas for improvement.

- **Process redesign:** Making changes to existing processes or creating new ones to optimize workflows and enhance performance.

- **Process implementation:** Introducing the redesigned processes into the organization and ensuring proper execution.

- **Process monitoring and control:** Tracking process performance, measuring key metrics, and implementing control mechanisms to maintain efficiency and effectiveness.

Importance of Process Management System

It is critical to comprehend the significance of process management for any manager overseeing a firm. It does more than just make workflows smooth. Process Management makes sure that every part of business operations moves as quickly as possible.

By implementing business processes management, we can avoid errors caused by inefficient human labor and cut down on time lost on repetitive operations. It also keeps data loss and process step errors at bay. Additionally, process management guarantees that resources are employed effectively, increasing the cost-effectiveness of our company. Process management not only makes business operations better, but it also makes sure that our procedures meet the needs of your clients. This raises income and improves consumer happiness.

Characteristics of a Process

A process has the following attributes.

- **Process Id:** A unique identifier assigned by the operating system.

- **Process State:** Can be ready, running, etc.

- **CPU registers:** Like the Program Counter (CPU registers must be saved and restored when a process is swapped in and out of the CPU)

- **Accounts information:** Amount of CPU used for process execution, time limits, execution ID, etc

- **I/O status information:** For example, devices allocated to the process, open files, etc

- **CPU scheduling information:** For example, Priority (Different processes may have different priorities, for example, a shorter process assigned high priority in the shortest job first scheduling)

Process Control Block (PCB)

Every process has a process control block, which is a data structure managed by the operating system. An integer process ID (or PID) is used to identify the PCB. As shown below, PCB stores all of the information required to maintain track of a process.

Process state: The process's present state, such as whether it's ready, waiting, running, or whatever.

Process privileges: This is required in order to grant or deny access to system resources.

Process ID: Each process in the OS has its own unique identifier.

Pointer: It refers to a pointer that points to the parent process.

Program counter: The program counter refers to a pointer that points to the address of the process's next instruction.

CPU registers: Processes must be stored in various CPU registers for execution in the running state.

CPU scheduling information

Process priority and additional scheduling information are required for the process to be scheduled.

Memory management information

This includes information from the page table, memory limitations, and segment table, all of which are dependent on the amount of memory used by the OS.

Accounting information

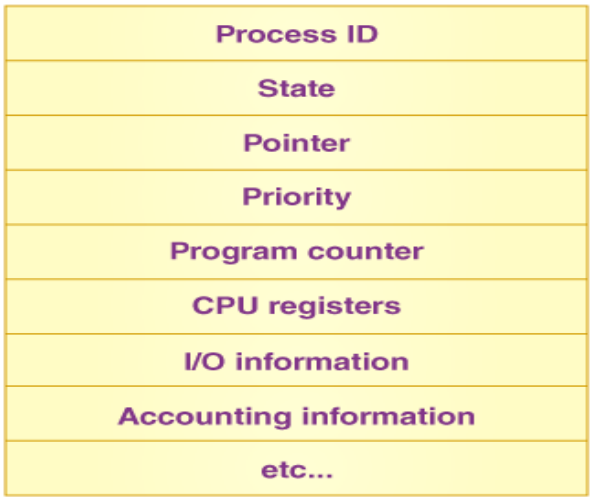
This comprises CPU use for process execution, time constraints, and execution ID, among other things.

IO status information

This section includes a list of the process's I/O devices.

The PCB architecture is fully dependent on the operating system, and different operating systems may include different information. A simplified diagram of a PCB is shown below.

The PCB is kept for the duration of a procedure and then removed once the process is finished.



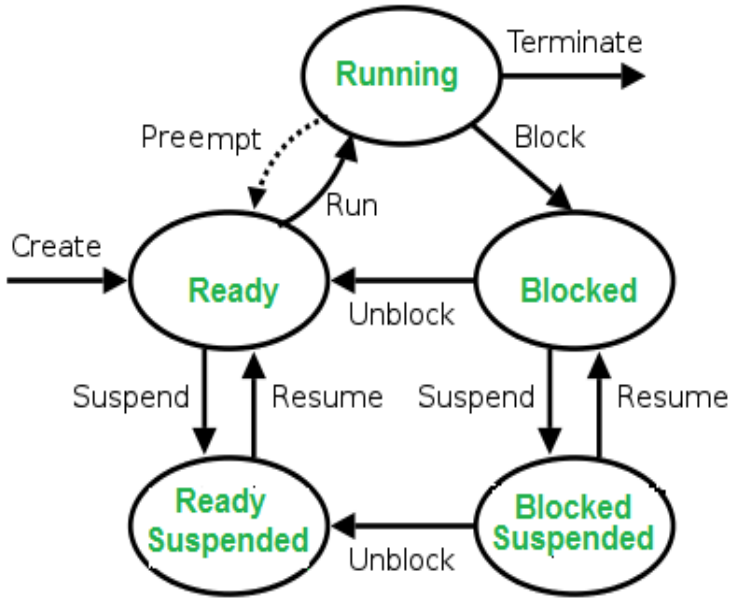
All of the above attributes of a process are also known as the **context of the process**. Every process has its own process control block (PCB), i.e. each process will have a unique PCB. All of the above attributes are part of the PCB.

States of Process

A process is in one of the following states:

- **New:** Newly Created Process (or) being-created process.

- **Ready:** After the creation process moves to the Ready state, i.e. the process is ready for execution.
- **Run:** Currently running process in CPU (only one process at a time can be under execution in a single processor)
- **Wait (or Block):** When a process requests I/O access.
- **Complete (or Terminated):** The process completed its execution.
- **Suspended Ready:** When the ready queue becomes full, some processes are moved to a suspended ready state
- **Suspended Block:** When the waiting queue becomes full.



Process vs Program

A program is a piece of code that can be as simple as a single line or as complex as millions of lines. A computer program is usually developed in a programming language by a programmer. The process, on the other hand, is essentially a representation of the computer program that is now running. It has a comparatively shorter lifetime.

Here is a basic program created in the C programming language as an example:

```
#include <stdio.h>

int main() {

printf(“Hi, Subhadip! \n”);

return 0;

}
```

A computer program refers to a set of instructions that, when executed by a computer, perform a certain purpose. We can deduce that a process refers to a dynamic instance of a computer program when we compare a program to a process. An algorithm is an element of a computer program that performs a certain task. A software package is a collection of computer programs, libraries, and related data.

Advantages of Process Management

- **Improved Efficiency:** Process management can help organizations identify bottlenecks and inefficiencies in their processes, allowing them to make changes to streamline workflows and increase productivity.
- **Cost Savings:** By identifying and eliminating waste and inefficiencies, process management can help organizations reduce costs associated with their business operations.
- **Improved Quality:** Process management can help organizations improve the quality of their products or services by standardizing processes and reducing errors.
- **Increased Customer Satisfaction:** By improving efficiency and quality, process management can enhance the customer experience and increase satisfaction.
- **Compliance with Regulations:** Process management can help organizations comply with regulatory requirements by ensuring that processes are properly documented, controlled, and monitored.

Disadvantages of Process Management

- **Time and Resource Intensive:** Implementing and maintaining process management initiatives can be time-consuming and require significant resources.

- **Resistance to Change:** Some employees may resist changes to established processes, which can slow down or hinder the implementation of process management initiatives.

- **Overemphasis on Process:** Overemphasis on the process can lead to a lack of focus on customer needs and other important aspects of business operations.

- **Risk of Standardization:** Standardizing processes too much can limit flexibility and creativity, potentially stifling innovation.
- **Difficulty in Measuring Results:** Measuring the effectiveness of process management initiatives can be difficult, making it challenging to determine their impact on organizational performance

CHAPTER THREE

Process Scheduling

Process Scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Categories of Scheduling

There are two categories of scheduling:

1. **Non-preemptive:** Here the resource can't be taken from a process until the process completes execution. The switching of resources occurs when the running process terminates and moves to a waiting state.
2. **Preemptive:** Here the OS allocates the resources to a process for a fixed amount of time. During resource allocation, the process switches from running state to ready state or from waiting state to ready state. This switching occurs as the CPU may give priority to other processes and replace the process with higher priority with the running process.

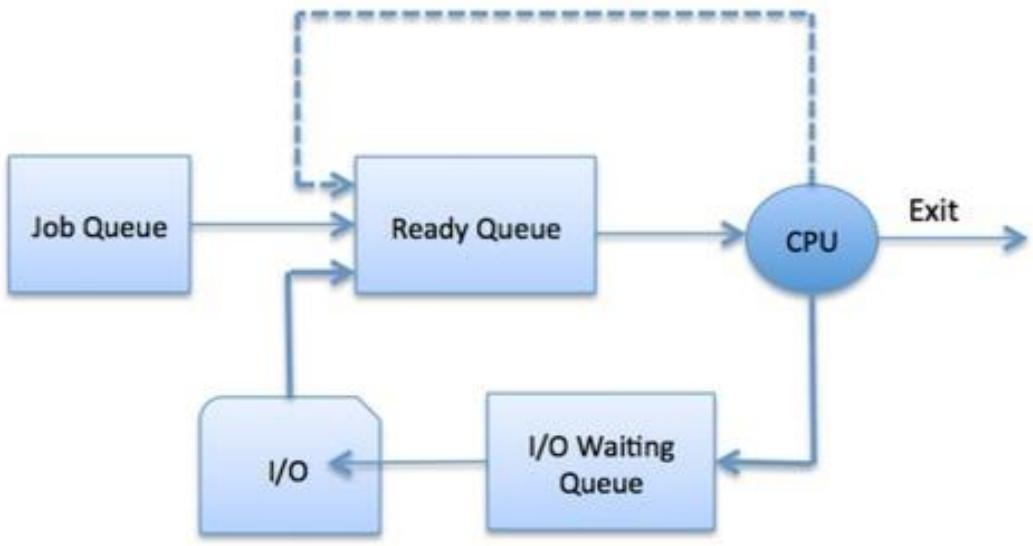
Process Scheduling Queues

The OS maintains all Process Control Blocks (PCBs) in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.

- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running and non-running states which are described below −

S.N. State & Description

1 **Running**

When a new process is created, it enters into the system as in the running state.

2 **Not Running**

Processes that are not running are kept in queue, waiting for their turn to execute. Each

entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the

process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.

5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

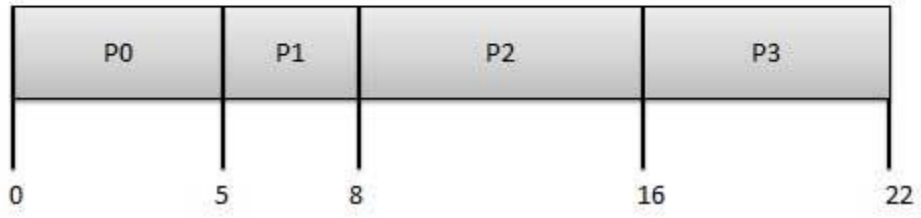
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job Next (SJN)

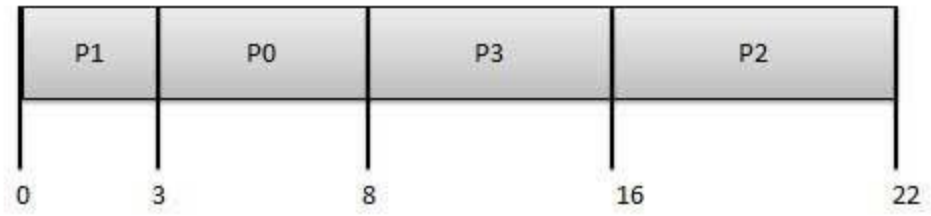
- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

Process Arrival Time Execution Time Service Time

P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8



Waiting time of each process is as follows −

Process Waiting Time

P0	0 - 0 = 0
P1	5 - 1 = 4
P2	14 - 2 = 12
P3	8 - 3 = 5

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

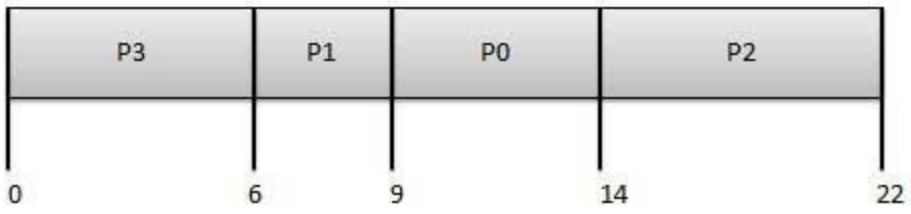
Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

Process	**Arrival Time**	**Execution Time**	**Priority**	**Service Time**
P0	0	5	1	0
P1	1	3	2	11
P2	2	8	1	14
P3	3	6	3	5

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Waiting time of each process is as follows −

Process	Waiting Time
P0	0 - 0 = 0
P1	11 - 1 = 10
P2	14 - 2 = 12
P3	5 - 3 = 2

Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

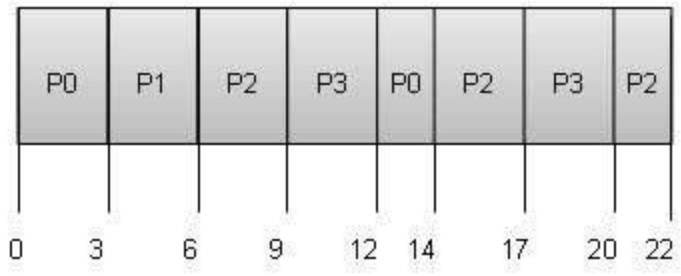
Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Quantum = 3



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

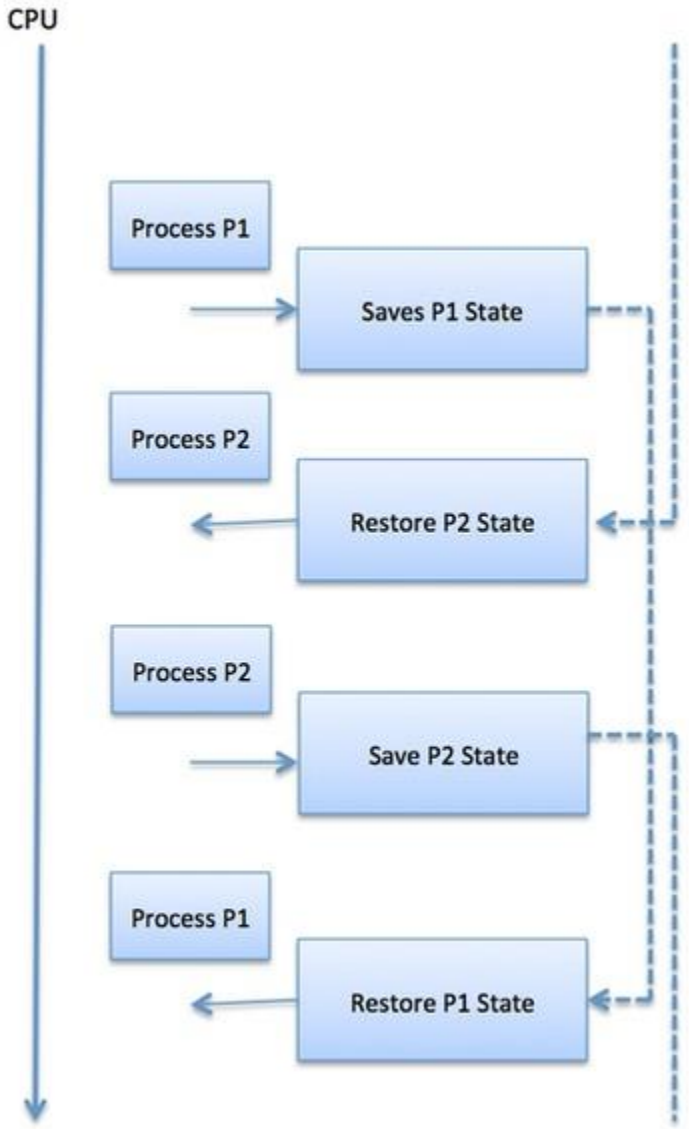
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Switching

A context switching is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing. Context



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

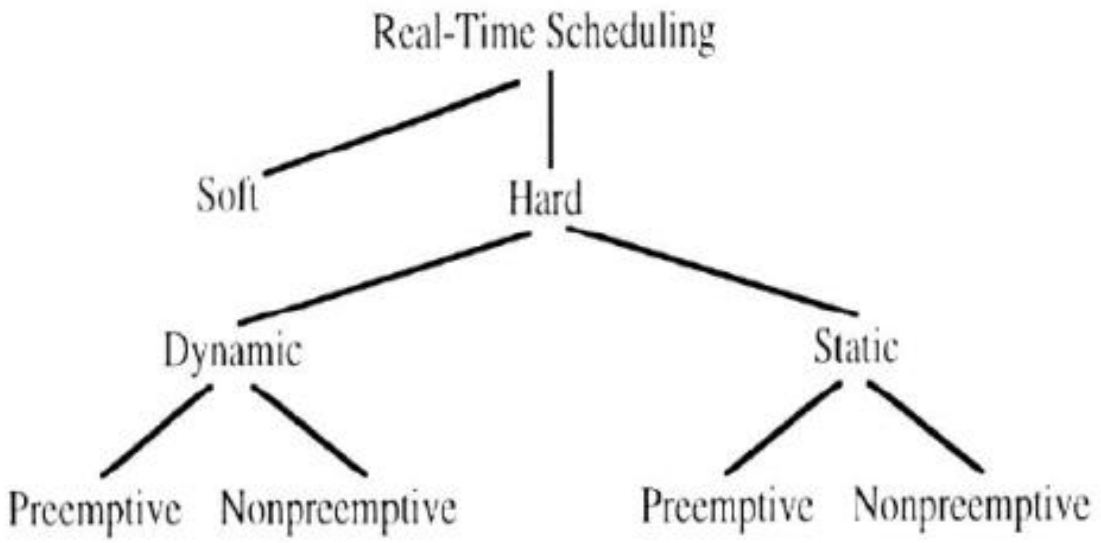
- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

Real-time scheduling is crucial in systems where tasks or processes must meet strict timing constraints to ensure correct operation. These systems include embedded systems, control systems, multimedia applications, and real-time data processing. Real-time scheduling considerations involve ensuring that tasks are executed within their deadlines, minimizing response times, and providing deterministic behavior. Here are some key considerations:

1. **Priority-Based Scheduling:**
    - Tasks are assigned priorities based on their importance and timing requirements.
    - Real-time operating systems (RTOS) use priority-based scheduling algorithms like Rate-Monotonic Scheduling (RMS) or Earliest Deadline First (EDF) to schedule tasks.
    - Higher-priority tasks preempt lower-priority ones to ensure critical tasks meet their deadlines.
2. **Determinism:**
    - Real-time systems must exhibit deterministic behavior, where the timing and outcome of tasks are predictable and repeatable.
    - Non-deterministic factors such as interrupt latency, task preemption, and resource contention should be minimized or accounted for in the scheduling algorithm.
3. **Task Deadline Management:**
    - Each task is associated with a deadline that specifies when it must complete its execution.

- Scheduling algorithms ensure that tasks are executed in a manner that meets their deadlines.
- Missed deadlines may result in system failures or degraded performance, especially in safety-critical or mission-critical applications.

4. **Interrupt Handling:**
   - Interrupts can disrupt the execution of tasks and introduce non-deterministic behavior.
   - Real-time systems should minimize interrupt latency, the time between the occurrence of an interrupt and its handling.
   - Critical interrupts may be prioritized, and non-essential interrupts may be temporarily disabled during critical task execution.
5. **Resource Management:**
   - Proper allocation and management of system resources (CPU time, memory, I/O devices) are essential for real-time scheduling.
   - Resource contention can lead to delays and affect the timing behavior of tasks.
   - Techniques such as resource reservation, priority inheritance, and priority ceiling protocols help manage resource access and prevent priority inversion.
6. **Response Time Analysis:**
   - Real-time systems must guarantee maximum response times for critical tasks.
   - Response time analysis evaluates the worst-case response time of tasks under different scheduling scenarios to ensure deadlines are met.
   - Techniques like Worst-Case Execution Time (WCET) analysis and schedulability analysis help determine if a system's scheduling policy can meet its timing requirements.
7. **Overhead and Efficiency:**
   - Real-time scheduling algorithms should be efficient and impose minimal overhead on system performance.
   - Context switching, scheduling overhead, and algorithm complexity should be kept low to maximize system throughput and responsiveness.



1.

Soft Real-Time Scheduling:

- In soft real-time scheduling, meeting deadlines is desirable but not mandatory.
- If a deadline is missed, it may lead to degraded performance but not system failure.
- Examples include multimedia streaming, online gaming, and some types of data processing applications.

2. **Hard Real-Time Scheduling:**
    - In hard real-time scheduling, meeting deadlines is critical.
    - Missing a deadline can lead to system failure or catastrophic consequences.
    - Examples include aircraft control systems, medical devices, and automotive safety systems.
3. **Static Scheduling:**
    - In static scheduling, task priorities and scheduling decisions are determined at system design time and remain fixed.
    - Task execution times and deadlines are known and constant.
    - Examples include Rate-Monotonic Scheduling (RMS) and Deadline Monotonic Scheduling (DMS).
4. **Dynamic Scheduling:**
    - In dynamic scheduling, task priorities and scheduling decisions can change dynamically at runtime.
    - Task priorities may be adjusted based on system conditions or runtime events.

- Examples include Earliest Deadline First (EDF) scheduling and Least Slack Time First (LST) scheduling.

5. **Preemptive Scheduling:**
    - In preemptive scheduling, higher-priority tasks can interrupt the execution of lower-priority tasks.
    - When a higher-priority task becomes ready to execute, it preempts the currently running task and begins execution immediately.
    - Preemptive scheduling ensures that critical tasks meet their deadlines even if lower-priority tasks are running.
    - Example: A real-time operating system (RTOS) using a preemptive scheduling algorithm like RMS or EDF.
6. **Non-preemptive Scheduling:**
    - In non-preemptive scheduling, once a task starts execution, it continues until it completes or voluntarily yields the CPU.
    - Higher-priority tasks cannot interrupt the execution of lower-priority tasks.
    - Non-preemptive scheduling may be simpler to implement but can lead to missed deadlines if a high-priority task is blocked by a lower-priority task.
    - Example: A single-threaded application using a non-preemptive scheduling policy.

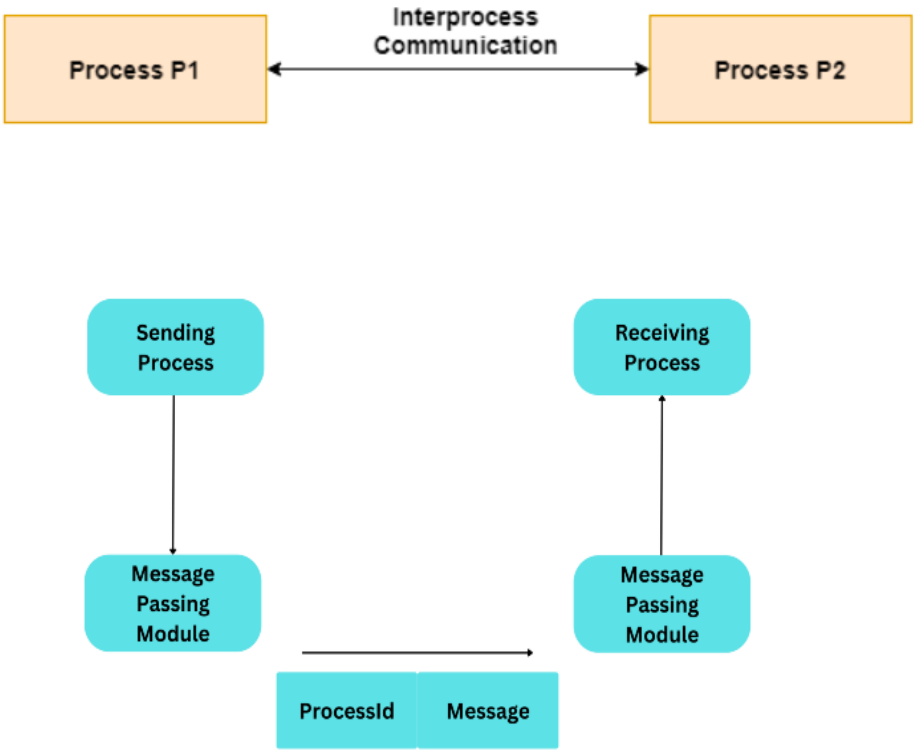
CHAPTER FOUR

Inter-Process Communication (IPC)

Inter process communication (IPC) was the transfer of information and interaction between multiple processes in an electronic system. Every operation in a tasking process structure runs on its own, as well as communication between them is required if these processes require to exchange of information or coordination of what they are doing

IPC is an essential part of contemporary operating systems and can be employed in a variety of applications, which include simple control-line appliances to complicated systems with distributed components. The primary goal of IPC is to make the transmission of knowledge among processes more private and effective.

A diagram that illustrates inter-process communication is as follows −



Synchronization in Interprocess Communication

Synchronization is a necessary part of interprocess communication. It is either provided by the interprocess control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows −

- **Semaphore**

  A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.

- **Mutual Exclusion**

  Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.

- **Barrier**

  A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.

- **Spinlock**

  This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Approaches to Interprocess Communication

The different approaches to implement interprocess communication are given as follows −

- **Pipe**

  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.

- **Socket**

The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.

- **File**

  A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.

- **Signal**

  Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.

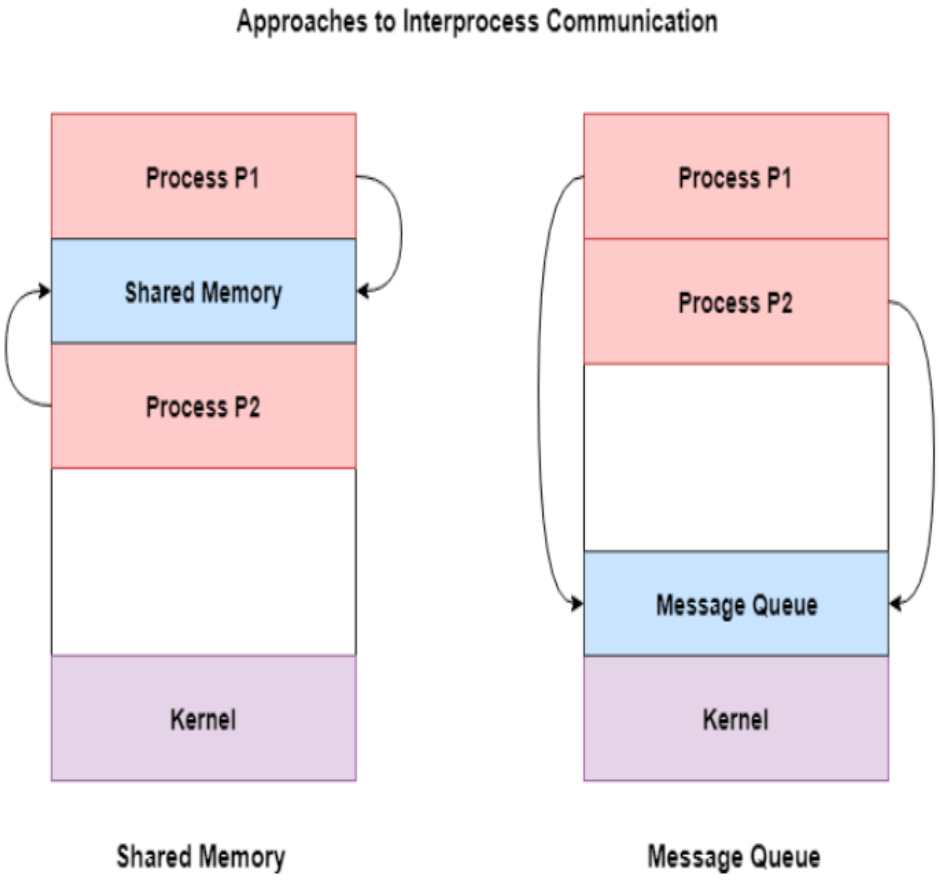
- **Shared Memory**

  Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

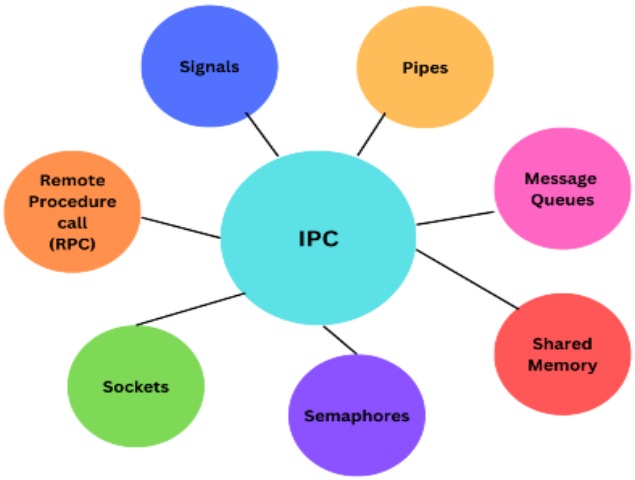
- **Message Queue**

  Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message queue and shared memory methods of interprocess communication is as follows −



Different Methods of IPC



Methods of IPC Info graphic

There are several IPC methods accessible, each of which has its own set of benefits and drawbacks. Several of the most common IPC techniques are −

- **Pipes** − A pipe is a channel of communication that is one-way that enables a single procedure to transmit data to a different one. Pipes can be identified as or unidentified. The operations running in anonymous pipes have to be associated (i.e., both parent and child processes). Known pipes, on the contrary together, may be utilized by processes that are separate from one another.
- **Message Queues** − Message queues are employed for inter-process interaction when both the sending and getting processes do not need to be present at the same time. The asynchronous communications may be sent and received. A message in a queue possesses a particular final destination and is accessible to multiple processes
- **Shared Memory** − Shared memory is an inter process communication method that enables various programs to make use of a single storage region. This allows them to effectively and effectively share data. Sharing memory is frequently employed in applications that are extremely fast.
- **Semaphores** − Semaphores serve to keep the utilization of resources that are shared synchronized. Their companies serve as responses that limit the number of procedures that may utilize a resource that is shared at any given time. Semaphores are useful for implementing critical sections in which only one process has access to a resource that is shared at a time.
- **Socket** − Sockets constitute an internet-based communications process that enables procedures to interact with one another over a network. Someone can communicate both locally and remotely. In client-server relationships applications, ports are frequently used.
- **Remote Procedure call(RPC)** − RPC is a procedure that enables a single process to call an operation in another. It allows procedures to call treatments in distant systems as though they were actually local, enabling distributed computing. In systems with distributed components, RPC is frequently used.
- **Signals** − Asynchronous IPC signals are employed for informing an operator of an occurrence or interference. The Operating System (OS) sends communication through

processes as well as between processes. Programming based on events can be implemented using signals.

Advantages

The following are many benefits of employing inter process communication (IPC) techniques for procedure interaction, such as −

- **Increased Modularity** − IPC enables developers to divide large applications into smaller parts that are easier to manage.
- **Improved Performance** − IPC may enhance the efficiency of applications by enabling handles to share information and convey it directly to one another.
- **Improved Scalability** − IPC may assist enhance an implementation's adaptability by enabling various processes to collaborate to complete a task.
- **Improved Fault Tolerance** − IPC may be employed to enhance an implementation's fault tolerance by allowing procedures to identify and recuperate from mistakes.
- **Increased Security** − IPC may be employed to enhance security for applications by managing the utilization of resources that are shared.

Disadvantages

Although there are numerous benefits associated with employing Inter process Communication (IPC) techniques, that are also a number of potential drawbacks to keep in mind, such as −

- **Increased Complexity** − IPC may complicate a program by needing creators to handle process interaction and synchronization.
- **Increased Overhead** − IPC can increase usage overhead by needing extra processing time as well as memory assets.
- **Increased Risk of Race Condition** − IPC may boost the likelihood of race circumstances, which occur when several programmers access a resource that is shared at the same period of time potentially resulting in corruption of information or other problems.
- **Security Risks** − Given that interactions between procedures can be seized or controlled by unauthorized individuals, IPC can pose safety hazards.

- **System Dependency** − In accordance with the system that underlies it, distinctive IPC techniques might come with distinct needs and constraints.
- **Debugging and Troubleshooting** − Debugging and troubleshooting IPC related problems can prove difficult because they might involve numerous procedures with intricate relationships.

CHAPTER FIVE

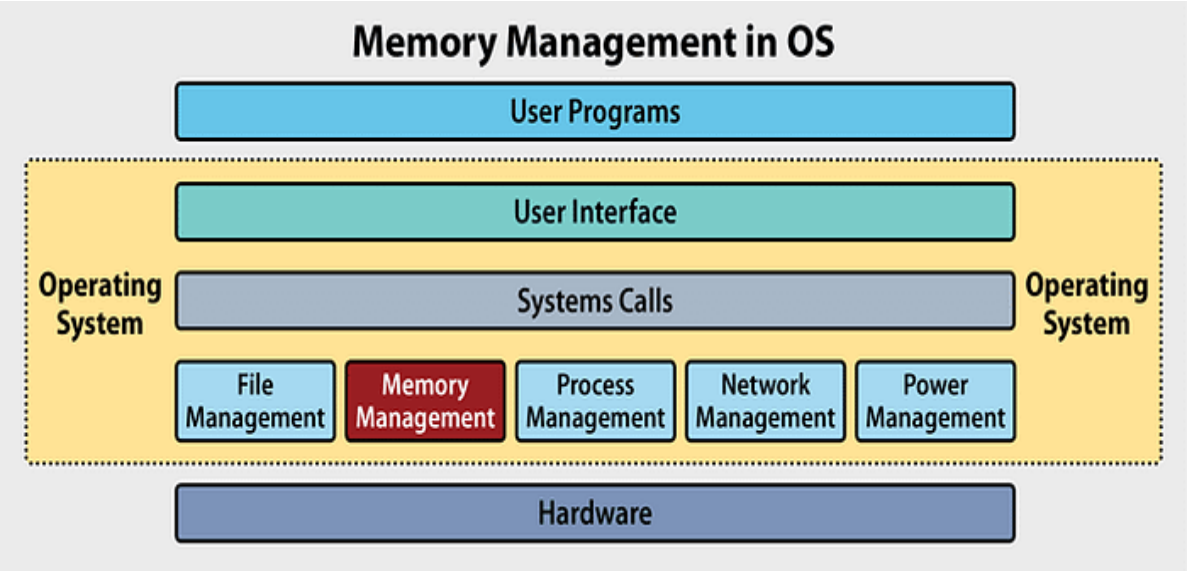
Memory Management Techniques

What Is Memory Management?

Memory management in an operating system (OS) is a critical function that involves **coordinating and optimizing the use of computer memory** to ensure efficient and secure execution of programs and processes.

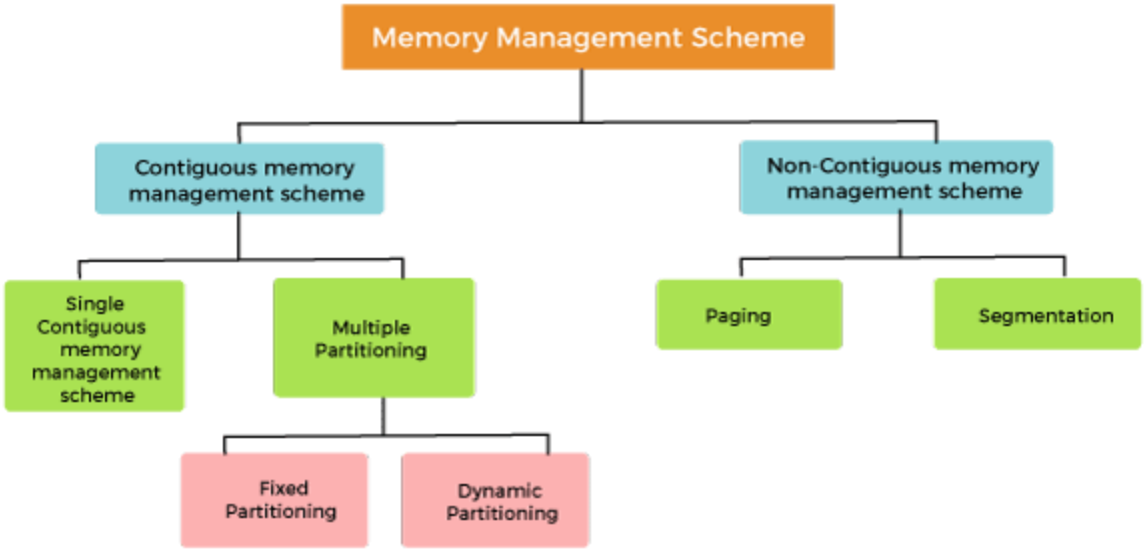
Memory management is critical to the computer system because the amount of main **memory available in a computer system is very limited**. At any time, many processes are competing for it.

Memory management encompasses several key aspects, including memory allocation, protection, sharing, and swapping.



Memory Management Techniques:

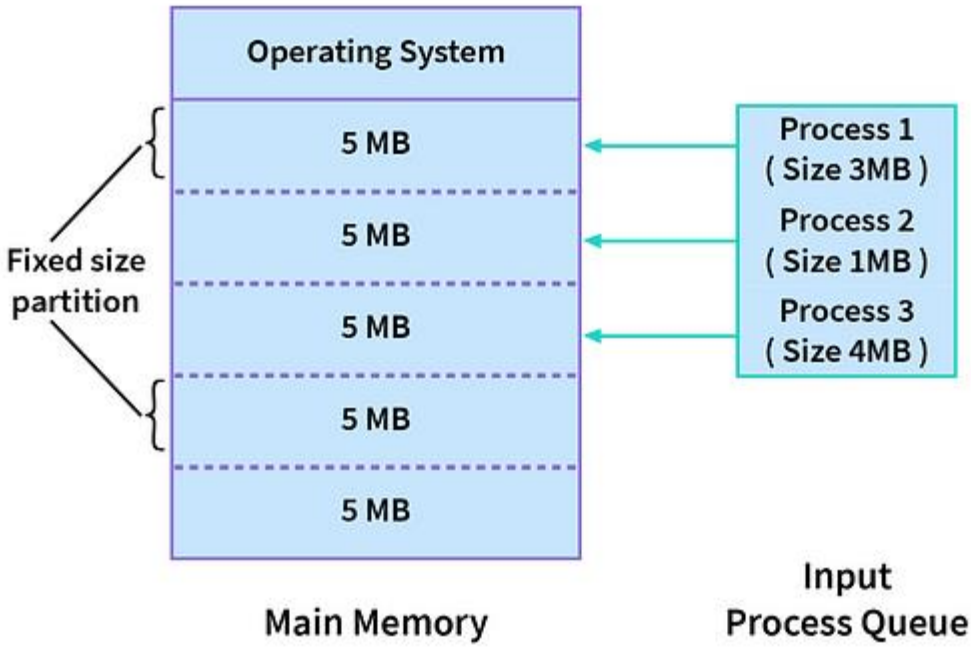
It can categorize in two.



Classification of memory management schemes

Contiguous memory management schemes

Contiguous memory management schemes are memory allocation techniques that involve allocating a contiguous block of memory to a process or program. In these schemes, each process is given a single, contiguous block of memory in which it can load and execute.



Contiguous memory management schemes

Single contiguous memory management schemes:

The Single contiguous memory management scheme is the simplest memory management scheme used in the earliest generation of computer systems. In this scheme, the main memory is divided into two contiguous areas or partitions. The operating systems reside permanently in one partition, generally at the lower memory, and the user process is loaded into the other partition.

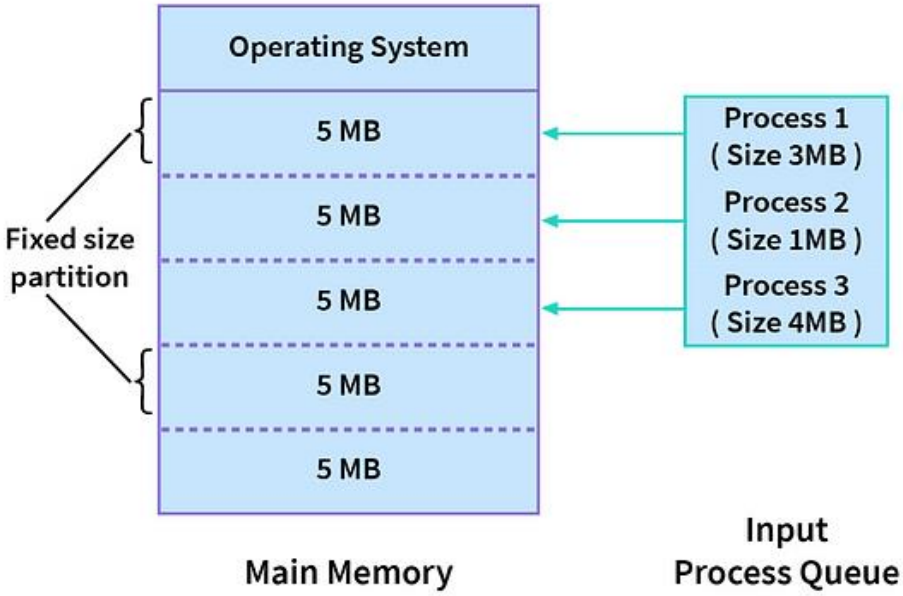
Multiple Partitioning:

The single Contiguous memory management scheme is inefficient as it limits computers to execute only one program at a time resulting in wastage in memory space and CPU time. The problem of inefficient CPU use can be overcome using multiprogramming that allows more than one program to run concurrently. To switch between two processes, the operating systems need to load both processes into the main memory. The operating system needs to divide the available main memory into multiple parts to load multiple processes into the main memory. Thus multiple processes can reside in the main memory simultaneously.

1. **Fixed Partitioning:**

- In fixed partitioning, the memory is divided into a fixed number of partitions or segments, each of a predefined size.
- Each partition can hold one process or program. The size of the partitions is determined during system configuration.
- Processes are assigned to partitions based on their size. Small processes may share a partition, while larger processes require entire partitions.
- Fixed partitioning is relatively simple to implement but can lead to inefficient memory utilization, as there may be internal fragmentation (unused memory within a partition).
- It is typically used in older systems where memory requirements were relatively small and fixed

2. **Variable Partitioning:**



Variable Partitioning:

Variable partitioning is a more flexible version of contiguous memory management, where partitions can vary in size.

- Memory is divided into variable-sized partitions, and processes are allocated memory based on their actual size, with no fixed partition sizes.
- A process is allocated the smallest available partition that can accommodate it.
- Variable partitioning helps reduce internal fragmentation, as processes are allocated memory more precisely. However, it requires dynamic memory allocation and management.
- This scheme is commonly used in modern operating systems to handle varying memory requirements of processes efficiently.

Non-Contiguous memory management schemes:

Non-contiguous memory management schemes, also known as **dynamic memory management schemes,** allow processes to be allocated memory in a non-contiguous manner. These schemes are more flexible and efficient in terms of memory utilization compared to contiguous memory

management schemes like fixed and variable partitioning. Here are two common non-contiguous memory management schemes:

1. **Paging**:

- In paging, both physical memory and the process's logical address space are divided into fixed-size blocks called "pages."
- Physical memory is divided into page frames, which are also of the same size as pages.
- When a process is loaded into memory, it is divided into fixed-size blocks, or pages, and these pages can be scattered throughout physical memory.
- A page table is used to map logical pages to physical page frames. Each entry in the page table contains the mapping information.
- Paging eliminates external fragmentation because pages can be allocated in any available page frame, and internal fragmentation is minimal.
- It allows for efficient memory allocation, and it simplifies memory management. However, it may incur some overhead due to the page table.

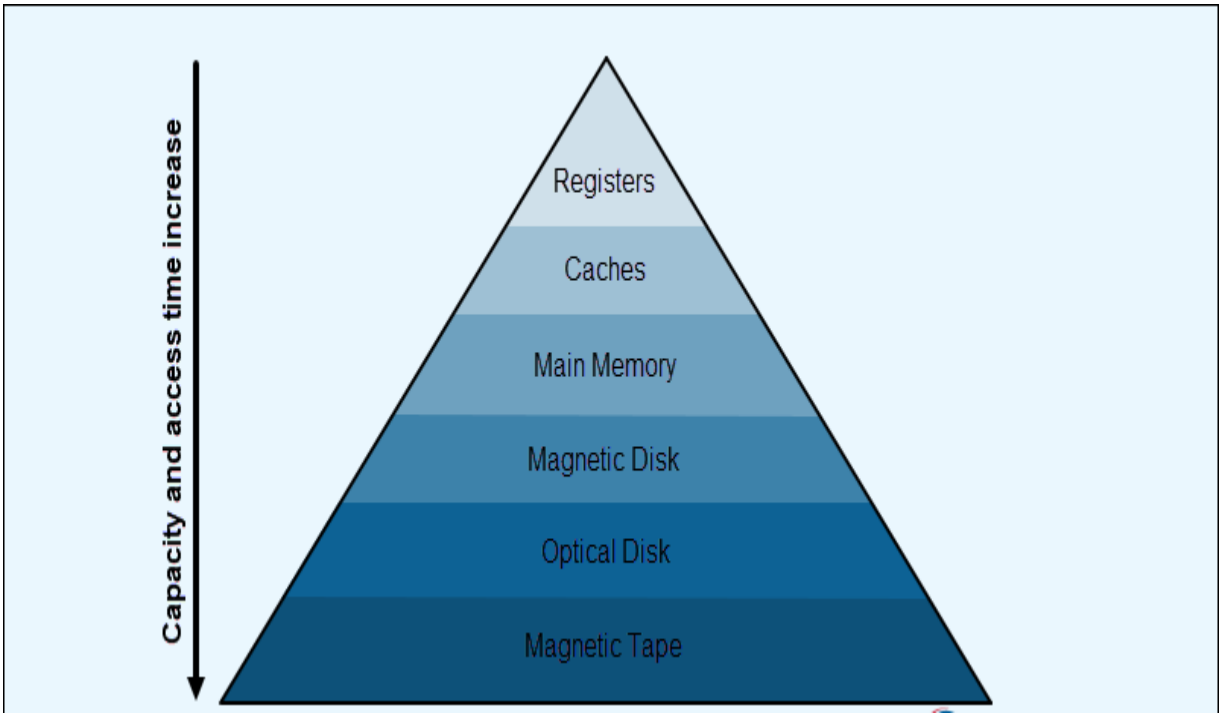
Segmentation:

- Segmentation divides the logical address space of a process into variable-sized segments, each with its own attributes.
- Each segment represents a different part of a program or data (e.g., code segment, data segment).
- Unlike paging, segments are not of uniform size, and they can grow or shrink dynamically.
- A segment table is used to map logical segments to physical memory addresses. Each entry in the segment table contains the base address and the length of the segment.
- Segmentation provides better memory utilization than paging for processes with varying memory requirements, as segments can expand or contract as needed.
- However, it may introduce external fragmentation when segments are deallocated or resized.

The main components in memory management are a processor and a memory unit. The efficiency of a system depends on how these two key components interact.

Efficient memory management depends on two factors:

1. **Memory unit organization**. Several different memory types make up the memory unit. A computer's memory hierarchy and organization affect data access speeds and storage size. Faster and smaller caches are closer to the CPU, while larger and slower memory is further away.



2. **Memory access**. The CPU regularly accesses data stored in memory. Efficient memory access influences how fast a CPU completes tasks and becomes available for new tasks. Memory access involves working with addresses and defining access rules across memory levels.

Memory management balances trade-offs between speed, size, and power use in a computer. Primary memory allows fast access but no permanent storage. On the other hand, secondary memory is slower but offers permanent storage.

Why Is Memory Management Necessary?

Main memory is an essential part of an operating system. It allows the CPU to access the data it needs to run processes. However, frequent read-and-write operations slow down the system.

Therefore, to improve CPU usage and computer speed, several processes reside in memory simultaneously. Memory management is necessary to divide memory between processes in the most efficient way possible.

As a result, memory management affects the following factors:

- **Resource usage**. Memory management is a crucial aspect of computer resource allocation. RAM is the central component, and processes use memory to run. An operating system decides how to divide memory between processes. Proper allocation ensures every process receives the necessary memory to run in parallel.
- **Performance optimization**. Various memory management mechanisms have a significant impact on system speed and stability. The mechanisms aim to reduce memory access operations, which are CPU-heavy tasks.
- **Security**. Memory management ensures data and process security. Isolation ensures processes only use the memory they were given. Memory management also enforces access permissions to prevent entry to restricted memory spaces.

Operating systems utilize memory addresses to keep track of allocated memory across different processes.

Memory Addresses

Memory addresses are vital to memory management in operating systems. A memory address is a unique identifier for a specific memory or storage location. Addresses help find and access information stored in memory.

Memory management tracks every memory location, maps addresses, and manages the memory address space. Different contexts require different ways to refer to memory address locations.

The two main memory address types are explained in the sections below. Each type has a different role in memory management and serves a different purpose.

Physical Addresses

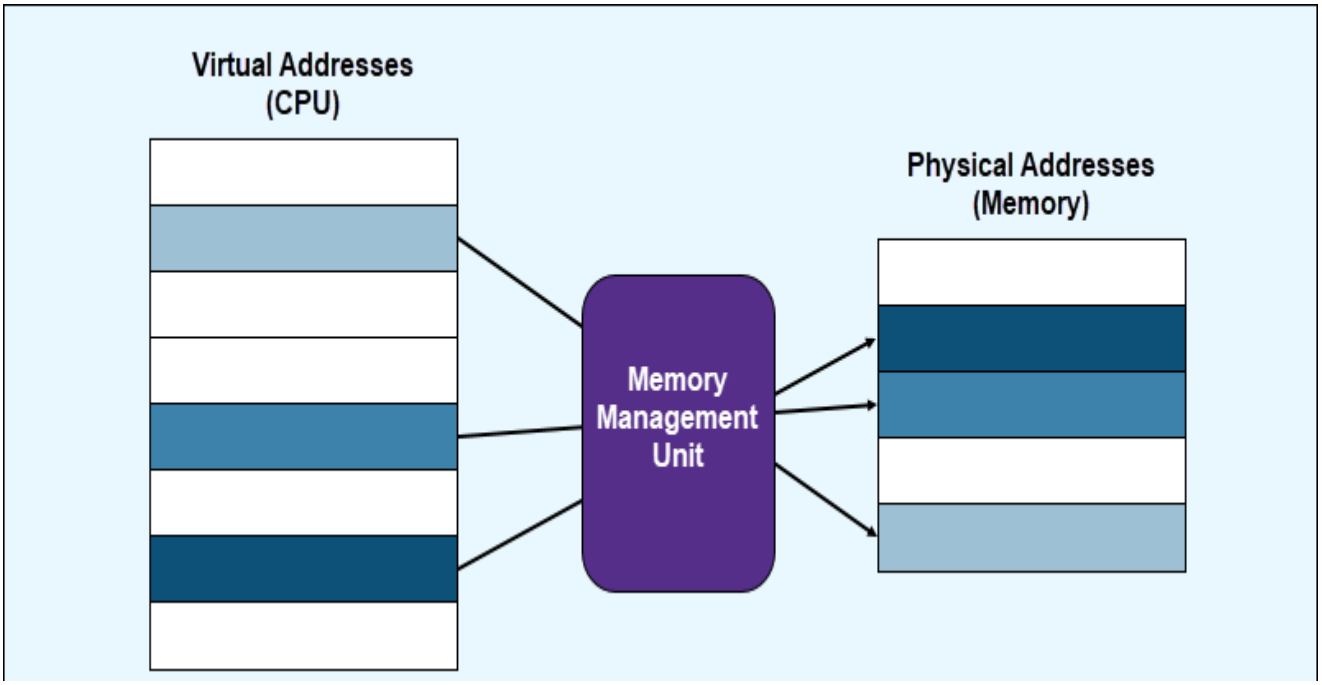
A physical address is a numerical identifier pointing to a physical memory location. The address represents the actual location of data in hardware, and they are crucial for low-level memory management.

Hardware components like the CPU or memory controller use physical addresses. The addresses are unique and have fixed locations, allowing hardware to locate any data quickly. Physical addresses are not available to user programs.

Virtual Addresses

A virtual address is a program-generated address representing an abstraction of physical memory. Every process uses the virtual memory address space as dedicated memory.

Virtual addresses do not match physical memory locations. Programs read and create virtual addresses, unaware of the physical address space. The main memory unit (MMU) maps virtual to physical addresses to ensure valid memory access.



The virtual address space is split into segments or pages for efficient memory use.

Static vs. Dynamic Loading

Static and dynamic loading are **two ways to allocate memory for executable programs**. The two approaches differ in memory usage and resource consumption. The choice between the two depends on available memory, performance results, and resource usage.

- **Static loading** allocates memory and addresses during program launch. It has predictable but inefficient resource usage where a program loads into memory along with all the necessary resources in advance. System utilities and applications use static loading to simplify program distribution. Executable files require compilation and are typically larger files. Real-time operating systems, bootloaders, and legacy systems utilize static loading.
- **Dynamic loading** allocates memory and address resolutions during program execution, and a program requests resources as needed. Dynamic loading reduces memory consumption and enables a multi-process environment. Executable files are smaller but have added complexity due to memory leaks, overhead, and runtime errors. Modern operating systems (Linux, macOS, Windows), mobile operating systems (Android, iOS), and web browsers use dynamic loading.

Static vs. Dynamic Linking

Static and dynamic linking are two different ways to handle libraries and dependencies for programs. The memory management approaches are similar to static and dynamic loading:

- **Static linking** allocates memory for libraries and dependencies before compilation during program launch. Programs are complete, and do not seek external libraries at compile time.
- **Dynamic linking** allocates memory for libraries and dependencies as needed after program launch. Programs search for external libraries as the requirement appears after compilation.

Static loading and linking typically unify into a memory management approach where all program resources are predetermined. Likewise, dynamic loading and linking create a strategy where programs allocate and seek resources when necessary.

Combining different loading and linking strategies is possible to a certain extent. The mixed approach is complex to manage but also brings the benefits of both methods.

Swapping

Swapping is a memory management mechanism operating systems use to free up RAM space. The mechanism moves inactive processes or data between RAM and secondary storage (such as HDD or SSD).

The swapping process utilizes virtual memory to address RAM space size limits, making it a crucial memory management technique in operating systems. The technique uses a section from a computer's secondary storage to create swap memory as a partition or file.

Swap space enables exceeding RAM space by dividing data into fixed-size blocks called pages. The paging mechanism tracks which pages are in RAM and which are swapped out through page faults.

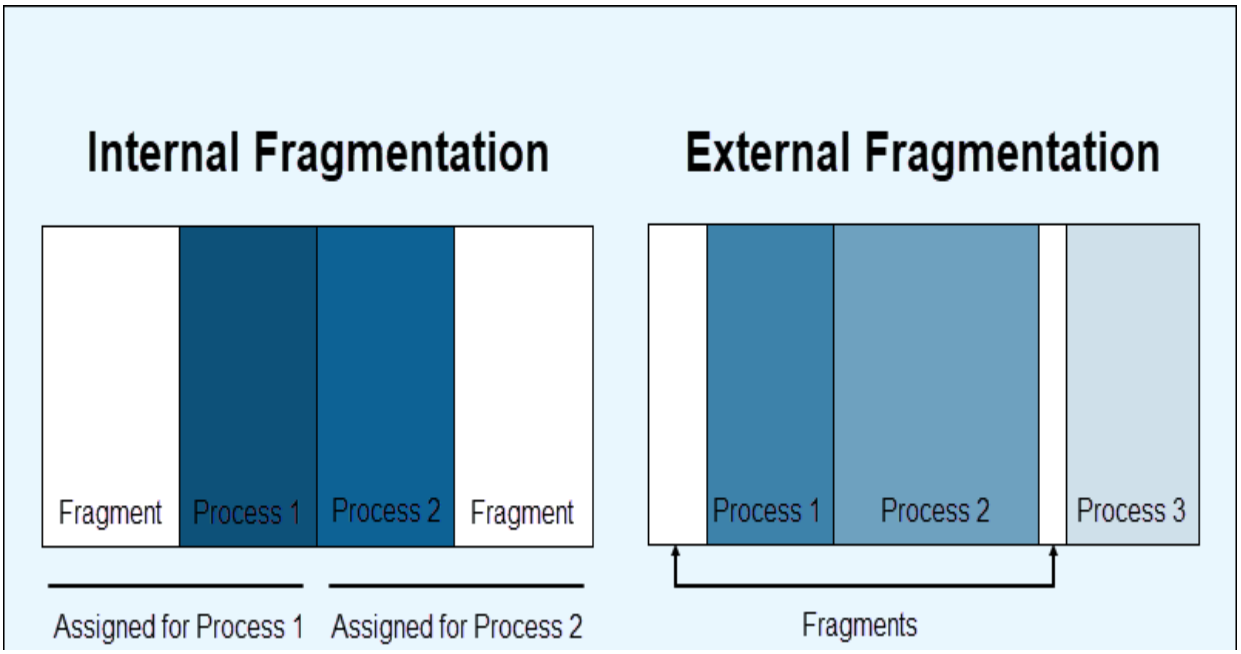
Excessive swapping leads to performance degradation due to secondary memory being slower. Different swapping strategies and swappiness values minimize page faults while ensuring that only essential data is in RAM.

Fragmentation

Fragmentation is a consequence that appears when attempting to divide memory into partitions. An operating system takes a part of the main memory, leaving the rest available for processes which divides further into smaller partitions. Partitioning does not utilize virtual memory.

There are two approaches to partitioning remaining memory: **into fixed or dynamic partitions**. Both approaches result in different fragmentation types:

- **Internal**. When remaining memory is divided into equal-sized partitions, programs larger than the partition size require overlaying, while smaller programs take up more space than needed. The unallocated space creates internal fragmentation.
- **External**. Dividing the remaining memory dynamically results in partitions with variable sizes and lengths. A process receives only the memory it requests. The space is freed when it is completed. Over time, unused memory gaps appear, resulting in external fragmentation.



Internal fragmentation requires design changes. The typical resolution is through the paging and segmentation mechanism.

External fragmentation requires the operating system to periodically defragment and free up unused space.

CHAPTER SIX

I/O Management

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

- **Block devices** − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices** − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

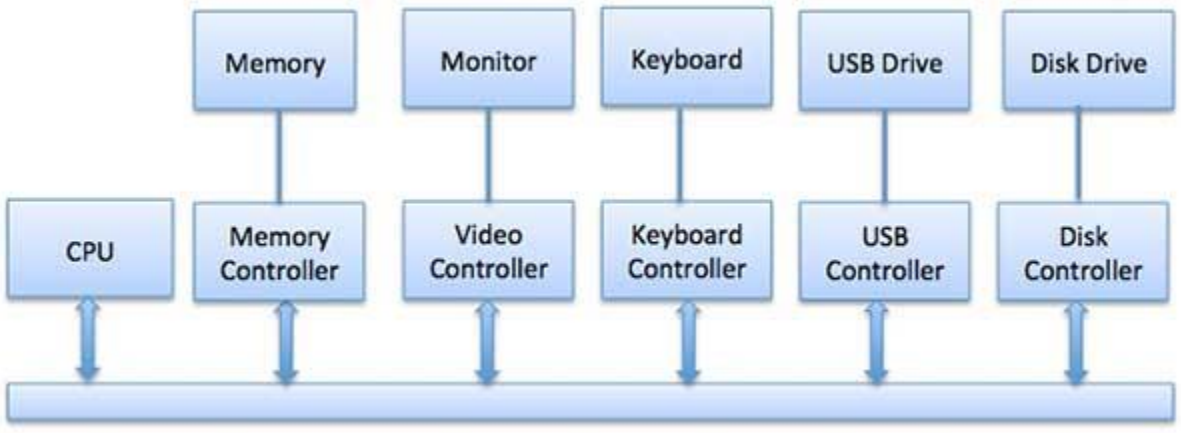
Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



Synchronous vs asynchronous I/O

- **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

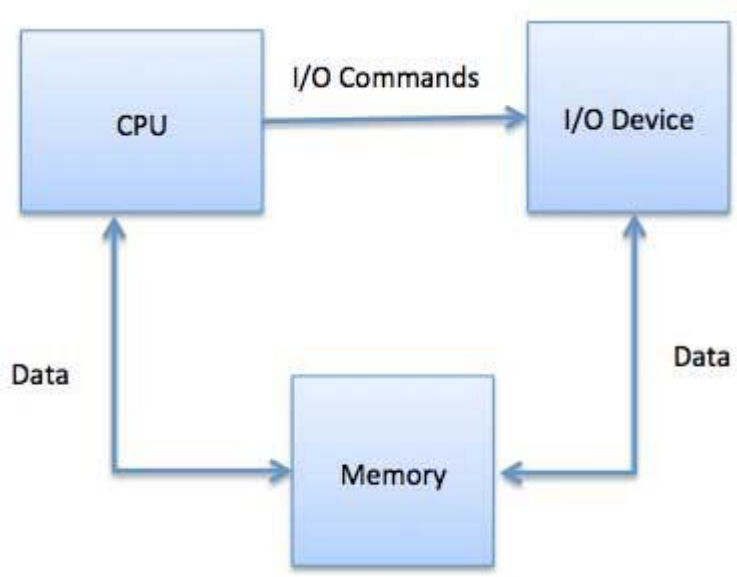
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

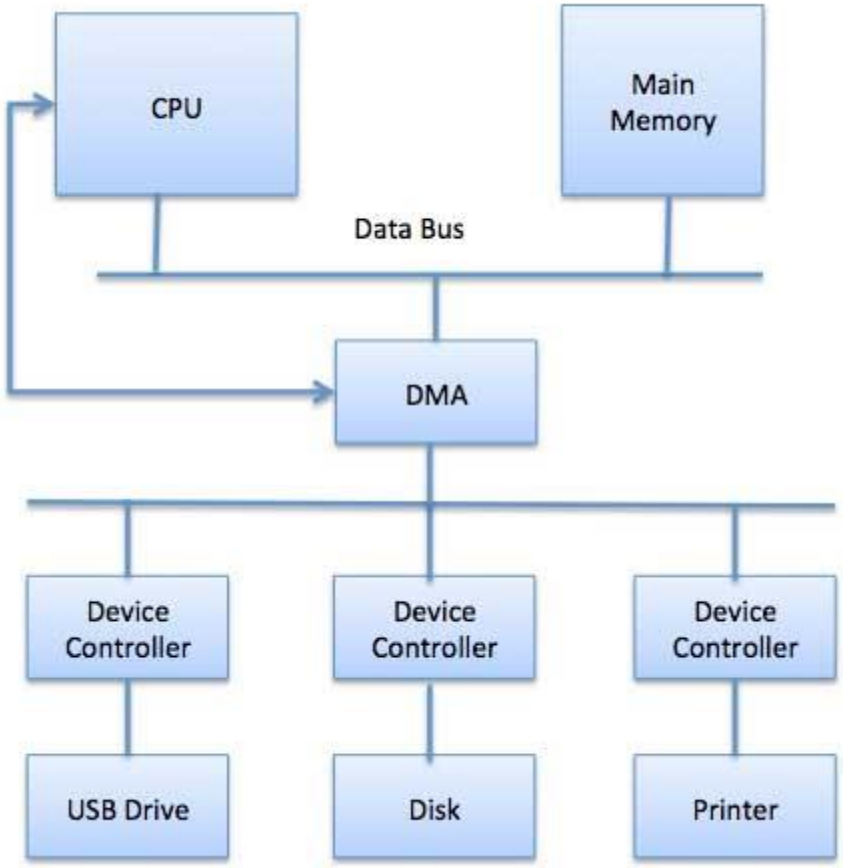
Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of

transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.



The operating system uses the DMA hardware as follows −

Step Description

1 Device driver is instructed to transfer disk data to a buffer address X.

2 Device driver then instruct disk controller to transfer data to buffer.

3 Disk controller starts DMA transfer.

4 Disk controller sends each byte to DMA controller.

5 DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.

6 When C becomes zero, DMA interrupts CPU to signal transfer completion.

Polling vs Interrupts I/O

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

Polling I/O

Polling is the simplest way for an I/O device to communicate with the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

Interrupts I/O

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

CHAPTER SEVEN

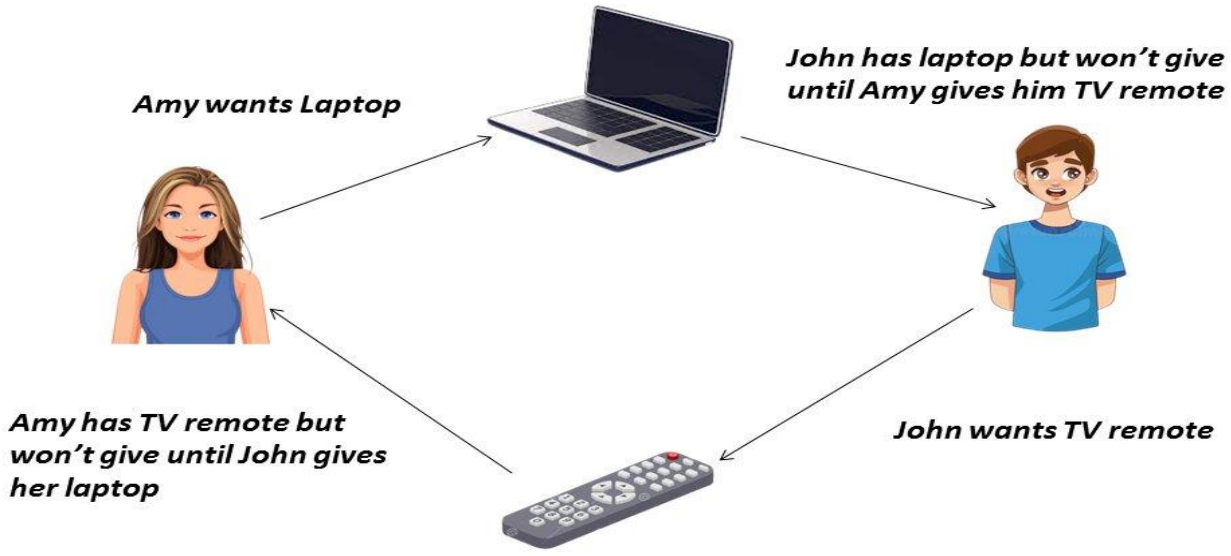
Deadlock Avoidance

What is Deadlock?

Deadlock is when two or more processes wait for each other to release a resource. This creates a standstill, and the system becomes unresponsive until one of the processes is killed.

Let's understand with a real-life analogy.

1. Consider a very narrow one-way road where two persons, A and B, coming from opposite directions, have blocked each other's passage. Consider road as a resource. And the other people moving on the road can be taken as processes. As the path is very narrow, none of these two-person can proceed further and is blocked. Now they are in a deadlock state.
2. John and Amy are brother-sister. They both are in a deadlock state as Amy wants a laptop which John is having and is not in the mood to give it to his sister. And Amy has a TV remote which John wants, But Amy is not giving it to him.



Also read: What is Operating Systems (OS) – Types, Functions, and Examples

Necessary Conditions for Deadlock

These four conditions must be met for a deadlock to happen in an operating system.

1. Mutual Exclusion

In this, two or more processes must compete for the same resources. There must be some resources that can only be used one process at a time. This means the resource is non-sharable. This could be a physical resource like a printer or an abstract concept like a lock on a shared **data structure**.

2. Hold and Wait

Hold and wait is when a process is holding a resource and waiting to acquire another resource that it needs but cannot proceed because another process is keeping the first resource. Each of these processes must have a hold on at least one of the resources it's requesting. If one process doesn't have a hold on any of the resources, it can't wait and will give up immediately.

3. No Preemption

Preemption means temporarily interrupting a task or process to execute another task or process. Preemption can occur due to an external event or internally within the system. If we take away the resource from the process that is causing deadlock, we can avoid deadlock. But is it a good approach? The answer is NO because that will lead to an inconsistent state. For example, if we take away memory from any process(whose data was in the process of getting stored) and assign it to some other process. Then will lead to an inconsistent state.

4. Circular Wait

The circular wait is when two processes wait for each other to release a resource they are holding, creating a deadlock. There must be a cycle in the graph below. As you can see, process 1 is holding on to a resource R1 that process 2 in the cycle is waiting for. This is an example of a circular wait.To better understand let's understand with another example. For example, Process A might be holding on to Resource X while waiting for Resource Y, while Process B is holding on to Resource Y while waiting for Resource Z, and so on around the cycle.

What are the Consequences of a Deadlock?

When a deadlock occurs, it can cause your computer to freeze up, making it difficult to even restart. This can cause you to lose important work or data and in some cases, may even damage your computer.

To prevent a deadlock state, it's important to be aware of what causes deadlocks and how to avoid them.

Methods For Handling Deadlocks

1. Deadlock avoidance

Deadlock avoidance is the process of taking steps to prevent deadlock from occurring. Operating system uses the deadlock Avoidance method to ensure the system is in a safe state(when the system can allocate resources and can avoid being in a deadlock state). We have a Deadlock avoidance algorithm-**Banker's algorithm** for this. When a new process is to be executed, it requires some resources. So banker's algorithm needs to know

- How many resources the process could request
- Which processes hold many resources.
- How many resources the system has.

And accordingly, resources are being assigned if available resources are more than requested to avoid deadlock. Tell the operating system about the maximum number of resources a process can request to complete its execution. The deadlock avoidance graph(shown in fig-2) assesses the resource-allocation state to check if a circular wait situation is not occurring.

If a deadlock does occur, it can sometimes be resolved by terminating one of the processes involved. However, this can cause data loss or corruption, so it's always preferable to try and prevent the deadlock from happening in the first place.

Bankers algorithm Pseudocode:

1. In **s**tarting all the processes are to be executed. Define two data structure finish and work:

Finish[n]=False.

Work=Available

Where n is a number of processes to be executed.

2. Find the process for which Finish[i]=False

And ***Need <=Work***(This means a request is valid as the number of requested resources of each resource type is less than the **available resources**, In case no such process is there then go to step

3. ***Work=Work+Allocation***

Finish[i]=True

Go to step 2 to find other processes

Any process says process 'i' finishes its execution. So that means the resources allocated to it previously, get free. So these resources are added to Work and Finish(i) of the process is set as true.

4. If ***Finish[i]=True*** for n processes then the system is in a safe state(If all the processes are executed in some sequence). Otherwise, it is in an unsafe state

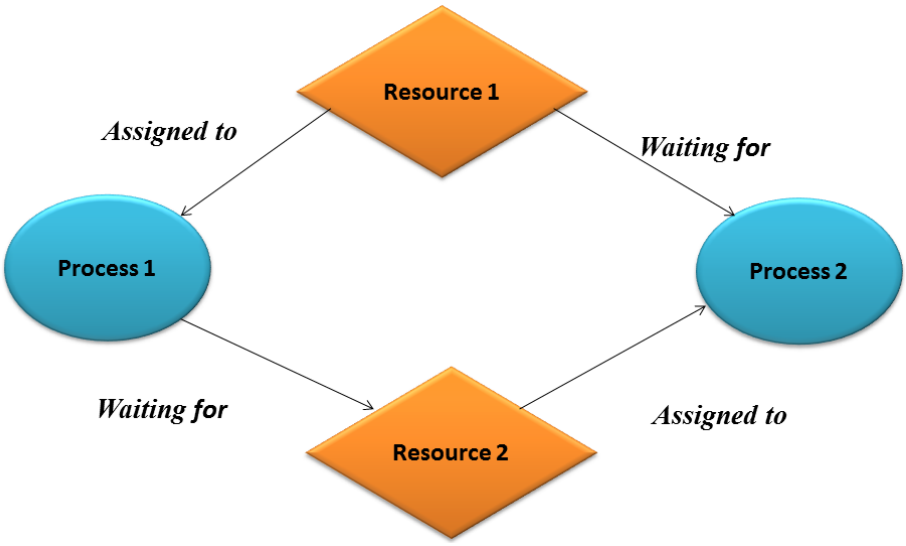
Also read: Real-time operating system

Must explore: Distributed operating system

2. Deadlock Detection

Detecting deadlocks is one of the most important steps in preventing them. A deadlock can happen anytime when two or more processes are trying to acquire a resource, and each process is waiting for other processes to release the resource.

The deadlock can be detected in the resource-allocation graph as shown in fig below.



This graph checks if there is a cycle in the **Resource Allocation Graph** and each resource in the cycle provides only one instance, If there is a cycle in this graph then the processes will be in a **deadlock state.**

So always remember detecting deadlocks is one of the most important steps in preventing them.

Also read: Operating System Online Courses & Certifications

3. Deadlock Prevention

The best way to prevent deadlocks is by understanding how they form in the first place. Deadlock can be prevented by eliminating the necessary conditions for deadlock(explained above).

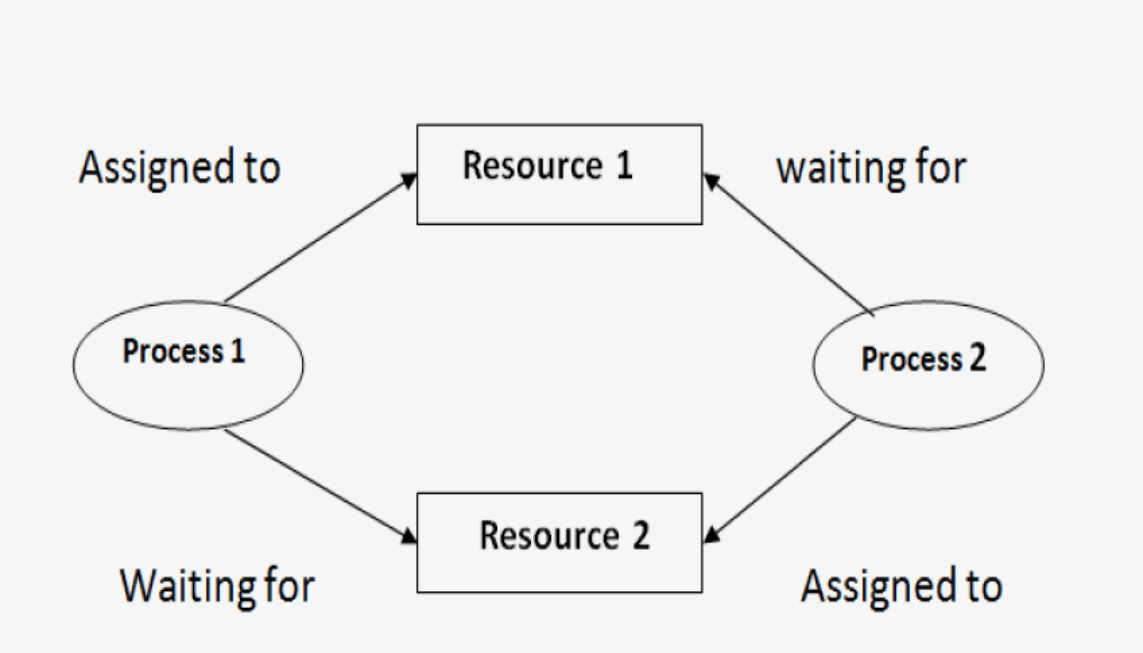
Some ways of prevention are as follows

1. **Preempting resources:** Take the resources from the process and assign them to other processes.
2. **Rollback:** When the process is taken away from the process, roll back and restart it.
3. **Aborting:** Aborting the deadlocked processes.
4. **Sharable resource:** If the resource is sharable, all processes will get all resources, and a deadlock situation won't come.

Example of Deadlock

Let's understand what deadlock is in the operating system better with this example. Let's assume that Process 1 and Process 2 are two processes. In the same vein, Resource 1 and Resource 2 are two resources. Supposedly,

- Process 1 is assigned to "Resource 1" but ends up waiting for "Resource 2".
- Process 2 needs "Resource 2" and waits for "Resource 1".



Subsequently, neither Process 1 nor Process 2 gets executed because the resources needed by them are held up. So, this was a deadlock example resulting in a potential deadlock in the operating system.

Necessary Conditions for Deadlock in OS

It isn't necessary that a deadlock will happen whenever more than one process, along with more than one resource, is involved. This is because a deadlock is earmarked by four conditions that are to be fulfilled anyhow. These include

1. Mutual Exclusion

Mutually exclusive means that for every process, there is a designated resource that cannot be shared. Henceforth, no two processes can ask for the same resource.

2. Hold and wait

As the name suggests, this deadlock condition in OS requires a process to wait for an occupied resource. This condition cannot be truer. Deadlock example

3. No preemption

Only one process can be scheduled at a point in time. So, it is said that no two processes can be executed simultaneously. This implies that only when a process is completed, then only the allocated or occupied resource will be duly released.

4. Circular wait

Every set of processes waits in a cyclic manner. This keeps them waiting forever, and they never get executed. As a result, deadlocks are called “circular wait” since they get a process stuck in a circular fashion. Every set of processes waits in a cyclic manner. This keeps them waiting forever, and they never get executed. As a result, deadlocks are also called “circular wait” since they get a process stuck in a circular fashion.

So, these were the four conditions of deadlock.

Methods for Handling Deadlock in OS

Are you brainstorming about the ways of handling a deadlock in an operating system? Well, it’s time to address your significant concern about handling deadlocks. These include

1. Deadlock Detection and Recovery

This method involves identifying the situation of deadlock and following some ways to recover your system from a deadlock. A user can simply abort all the processes that can lead to further potential deadlocks. It is recommended to abort one process at a time rather than ending all the processes simultaneously. Keep aborting the processes till the system returns back to normalcy

from the situation of deadlock. Another way out could be of freeing the resources from allocation until the deadlock perishes. This is called “resource preemption”.

2. Deadlock Prevention

As the name goes, if a user prevents any one of the four conditions for a deadlock from taking place, a deadlock will be prevented.

3. Deadlock Avoidance

Whenever a process asks for a resource to be allocated, an algorithm is exercised to examine whether this resource allocation is safe for the system or not. If it isn’t safe, then the request is denied. This very algorithm is known as the “deadlock avoidance” method.

4. Deadlock Ignorance

UNIX, Windows, and some other operating systems often are oblivious to deadlocks and ignore the situation of a deadlock, whenever it arises. This approach is termed the “Ostrich algorithm”. So, whenever a deadlock occurs, you can simply reboot the PC, and the deadlock will get resolved in no time.

Difference Between Starvation and Deadlock in the Operating System

Some users often confuse deadlocks for starvation in OS. However, the two are as different as day and night. Very commonly asked questions “What is starvation in OS” and “what is a deadlock in ” are also answered below. The table below lucidly traces the difference between starvation and deadlock:

Basis	Starvation	Deadlock
Definition	It is a process wherein resource-allocation is never done to low-priority resources and subsequently, they are never executed.	It is a situation where more than one process taking place in a system is prevented from getting executed because no resource is allocated to it.
Resource allocation	Here, resources are allocated to high-priority processes only.	Resource allocation is not done.
Ways to handle	A way to handle starvation is via aging.	By avoiding any of the four necessary conditions for deadlock.
Execution	Because of resource allocation, only high-priority resources are executed.	No process ever gets executed.
Other names	Starvation in OS is also termed as “lived lock”.	Also termed as “circular wait”.

Deadlock	Starvation
In this, two or more processes are each waiting for the other to release a resource, and neither process is able to continue.	In this, a process is unable to obtain the resources it needs to continue running.
Different processes are unable to proceed because they are each waiting for the other to do something.	A process is unable to proceed due to the unavailability of that resource.
Deadlock is also called Circular	Starvation is also called lived lock.

Basis	Starvation	Deadlock
wait.		
Avoiding the necessary conditions for deadlock can be prevented.	Starvation can be easily prevented by Aging.	

Advantages of the Deadlock Method

Since we are well-versed with what is a deadlock in OS now, let's get through the following advantages of it:

- **Perfect for a single task:** The deadlock method is suitable for performing single tasks.
- **No preemptions:** There is no need to block a process from accessing that resource and preempting it to some other process.
- **Lucrative method:** If your resource can get stored as well as restored quickly, then this is the method for you.
- **No computations required:** Sincere thanks to all the problem-solving done in the system design, you can now bid adieu to run-time algorithmic computations. Wondering the role system design has to play in this? You can learn all about it in this Web Designing and Development course online.

Disadvantages of the Deadlock Method

While the pros of the deadlock method exist, so do the cons:

- **Late initiation:** One of the biggest flaws of the deadlock method is that it lags in process initiation.
- **Forestall losses:** There is no iota of doubt that the deadlock method has inherent forestalled losses.
- **Preemptions are encountered frequently:** This method cannot be deemed good since it results in taking away the resource from the process making the output fetched up till now totally inconsistent.

- **No piecemeal resources:** The deadlock method doesn't approve any request asking for gradational resources.
- **Unawareness about future needs:** Usually, processes ain't aware of their future resource requirements.