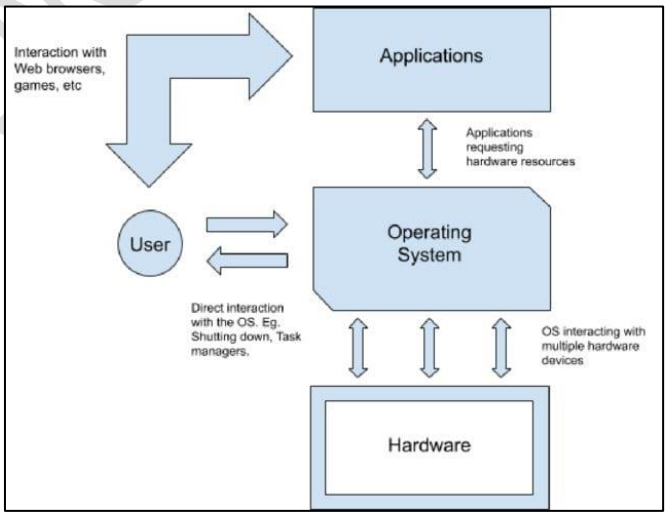
1. Overview of Operating System

Definition -:

An operating system (OS) is a software program that acts as an interface between a computer's hardware and its users. It is a fundamental component of any computer system and manages the overall operations of a computer or a network of computers.

The main purpose of an operating system is to provide a convenient and efficient way for users to interact with the computer system and to manage the hardware resources of the computer. It serves as a platform for running applications and provides services and utilities that enable users to perform tasks such as file management, process management, memory management, and device management.



Different types of operating systems exist, including general-purpose operating systems like **Windows**, **macOS**, and **Linux**, as well as specialized operating systems for specific devices or applications (e.g., real-time operating systems for embedded systems or mobile operating systems for smartphones).

Dual Mode Operation

Dual mode operation, also known as dual privilege mode or privileged mode, is a feature of modern operating systems that enables a clear distinction between privileged and non-privileged operations. It allows the operating system to protect critical system resources and maintain security by enforcing access control and preventing unauthorized access or manipulation.

In dual mode operation, the processor or CPU (Central Processing Unit) operates in two distinct modes:

1. User mode: In this mode, the CPU executes instructions on behalf of user applications or processes. The user mode provides a restricted environment where applications can run, but they have limited access to system resources. User mode applications cannot directly access hardware devices or perform privileged operations.

2. Kernel mode (also known as supervisor mode, system mode, or privileged mode): The kernel mode is the higher-privileged mode of operation. In this mode, the CPU executes instructions on behalf of the operating system kernel. The kernel has full access to system resources, including hardware devices, memory, and privileged instructions. It can perform critical system

operations such as modifying memory mappings, managing processes, controlling I/O devices, and enforcing security policies.

The transition between user mode and kernel mode occurs through system calls or exceptions. When a user application needs to perform a privileged operation or access a protected resource, it makes a request to the operating system through a system call. The system call interrupts the execution of the user mode code, transfers control to the kernel mode, and executes the requested operation on behalf of the application. After completing the privileged operation, control is returned to the user mode, and the application continues execution.

Dual mode operation provides several benefits:

1. Security: It allows the operating system to protect critical system resources and prevent unauthorized access or tampering. Only trusted code with appropriate privileges can perform privileged operations.

2. Stability: By isolating user applications from the kernel, the operating system can ensure that a faulty or malicious application does not disrupt the overall system stability. User mode applications operate within their own protected memory spaces and cannot directly interfere with the kernel or other applications.

3. Resource Management: The operating system can effectively manage and allocate system resources by controlling access from user applications. It can enforce policies to prevent resource contention, prioritize tasks, and optimize resource utilization.

OS as a Resource Manager

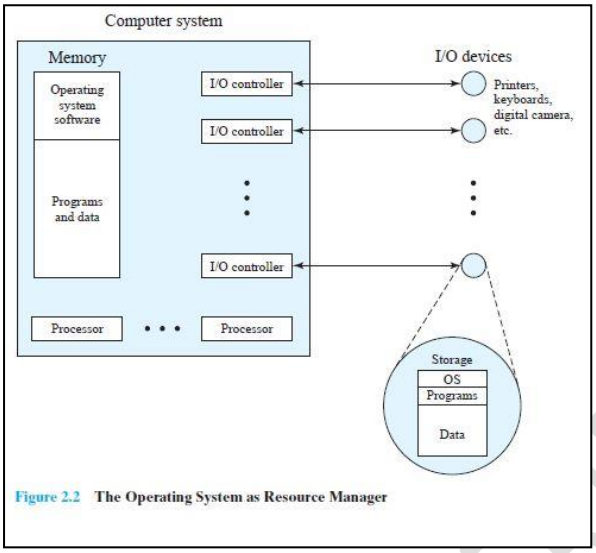


Figure 2.2 The Operating System as Resource Manager

An operating system (OS) acts as a resource manager, responsible for efficiently allocating and managing the various hardware and software resources of a computer system. It ensures that these resources are utilized effectively to fulfill the demands of user applications and provide a seamless computing experience. Here are some key aspects of how an OS functions as a resource manager:

By managing these resources, the OS ensures that they are allocated efficiently, conflicts are resolved, and the overall system operates in a stable and reliable manner. It serves as an intermediary layer between the hardware and software, abstracting the complexities of resource management and providing a unified interface for applications to interact with the system.

1. CPU Scheduling: The OS manages the allocation of CPU time among different processes or threads running on the system. It employs scheduling algorithms to determine which processes get to use the CPU and for how long.

2. Memory Management: The OS oversees the allocation and deallocation of memory resources, such as RAM (Random Access Memory), among active processes.

3. File System Management: The OS manages the organization, storage, and retrieval of files on storage devices such as hard drives. It provides an interface for creating, modifying, and deleting files, as well as controlling access permissions.

4. Device Management: The OS manages and controls access to various hardware devices connected to the computer system, such as printers, disk drives, network interfaces, and input/output devices.

5. Network Management: In networked environments, the OS facilitates communication between computers and network resources. It manages network connections, implements protocols for data transfer, handles network configuration, and provides security measures such as firewalls and encryption.

6. Process and Thread Management: The OS creates, schedules, and terminates processes and threads. It manages their execution states, assigns system resources, and facilitates interprocess communication and synchronization.

7. Security and Access Control: The OS enforces security policies to protect the system and its resources from unauthorized access and malicious activities.

8. System Performance Monitoring: The OS collects and analyzes system performance data, such as CPU usage, memory utilization, and disk I/O.

Imp Questions -:

1. Describe the concept of Operating System.
2. Explain dual mode operation.
3. Explain OS as a resource manager.

Different Types of Operating System

--- > Batch OS

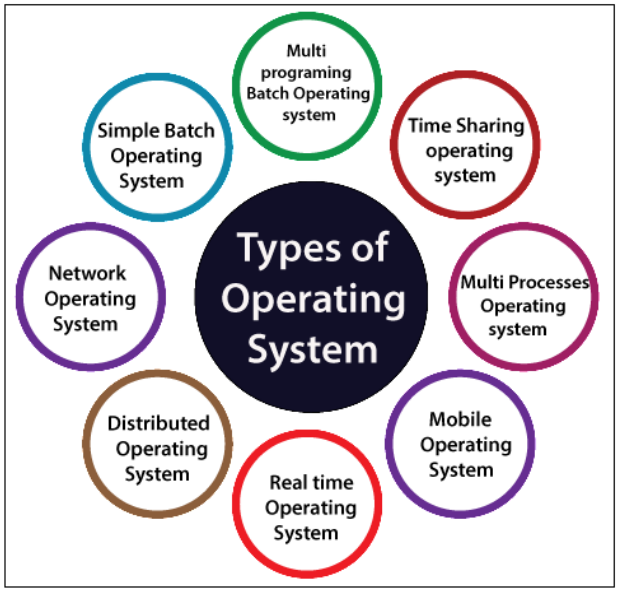
--- > Multi-Programmed OS

--- > Time-Shared OS

--- > Real Time OS

--- > Distributed OS

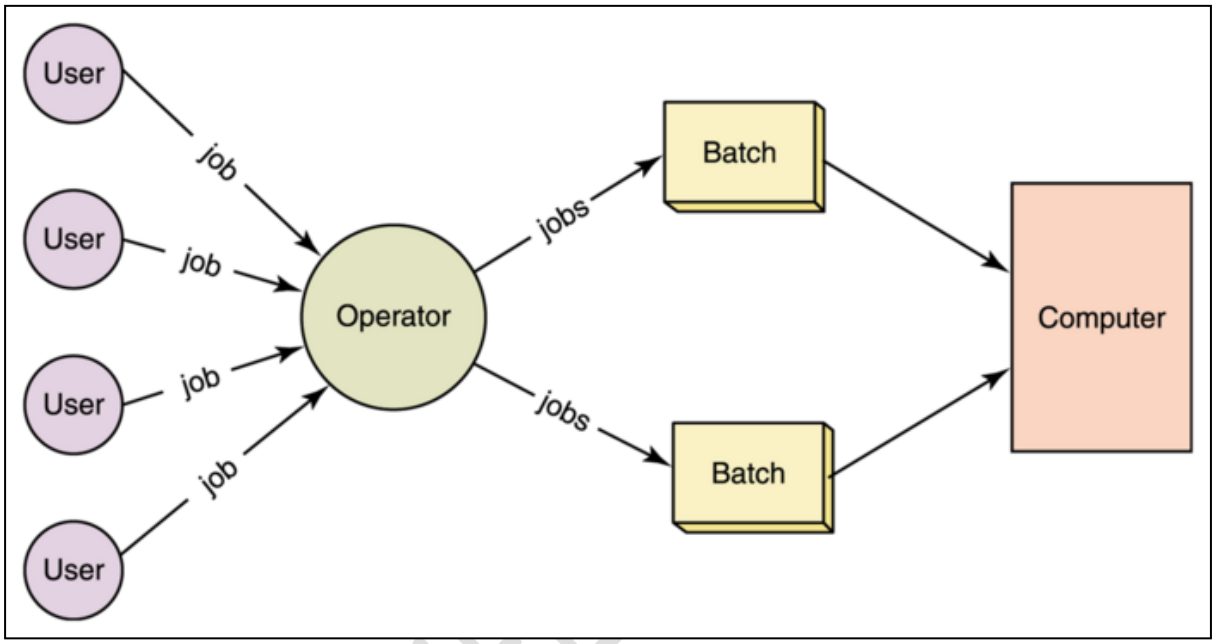
--- > Mobile OS



Batch Operating System

A batch operating system is a type of operating system that processes a series of jobs (also known as batch jobs) without requiring user interaction during their execution. In a batch processing environment, multiple jobs are submitted to the system as a batch, and the operating system executes them one after another, automatically.

Batch operating systems were prevalent in the early days of computing when computer resources were expensive and limited. They allowed organizations to maximize resource usage by submitting jobs in batches and efficiently utilizing the computer's processing power during off-peak hours.



Advantages of Batch Operating Systems:

1. Increased Efficiency: Batch operating systems maximize the utilization of computer resources by executing jobs in a sequential manner without requiring user interaction. This allows for continuous processing of jobs, minimizing idle time and improving overall system efficiency.

2. Automation: Batch systems automate the execution of jobs, eliminating the need for manual intervention. This reduces human errors and increases productivity by allowing users to submit multiple jobs for processing simultaneously.

3. Resource Optimization: By scheduling jobs based on priorities and available resources, batch operating systems can allocate system resources efficiently. This ensures that critical and high-priority jobs are processed promptly and that resource contention is minimized.

4. Large-Scale Processing: Batch systems are well-suited for handling large-scale processing tasks, such as payroll processing, report generation, or data processing. They can efficiently process a high volume of similar jobs without requiring manual intervention for each individual task.

5. Cost-Effectiveness: Batch processing allows organizations to make the most of their computer resources by optimizing their utilization. By efficiently utilizing the available resources, organizations can reduce operational costs and achieve better return on investment.

Disadvantages of Batch Operating Systems:

1. Lack of Interactivity: Batch operating systems lack interactivity and user interaction during job execution. Users have limited control over the execution of their jobs and must wait for the batch queue to process their jobs, which can result in longer response times.

2. Lack of Flexibility: Batch systems are designed for executing predefined jobs in a sequential manner. They are less flexible in handling ad-hoc or interactive tasks that require user inputs or real-time responses. Interactive applications, such as graphical user interfaces or real-time simulations, are not well-suited for batch systems.

3. Poor Responsiveness: Since batch systems prioritize processing jobs in the batch queue, they may not be responsive to immediate requests or critical tasks

that require immediate attention. Users may experience delays if their jobs are queued behind other jobs with higher priorities.

4. Limited Error Handling: In batch systems, errors or faults in one job may impact subsequent jobs in the batch queue. If a job encounters an error or failure, it may require manual intervention to correct the issue and resume processing, which can slow down the overall job execution.

5. Dependency on Job Order: The order in which jobs are submitted to the batch queue may impact overall performance. If high-priority or critical jobs are placed behind long-running jobs, it can delay their execution and affect the system's responsiveness.

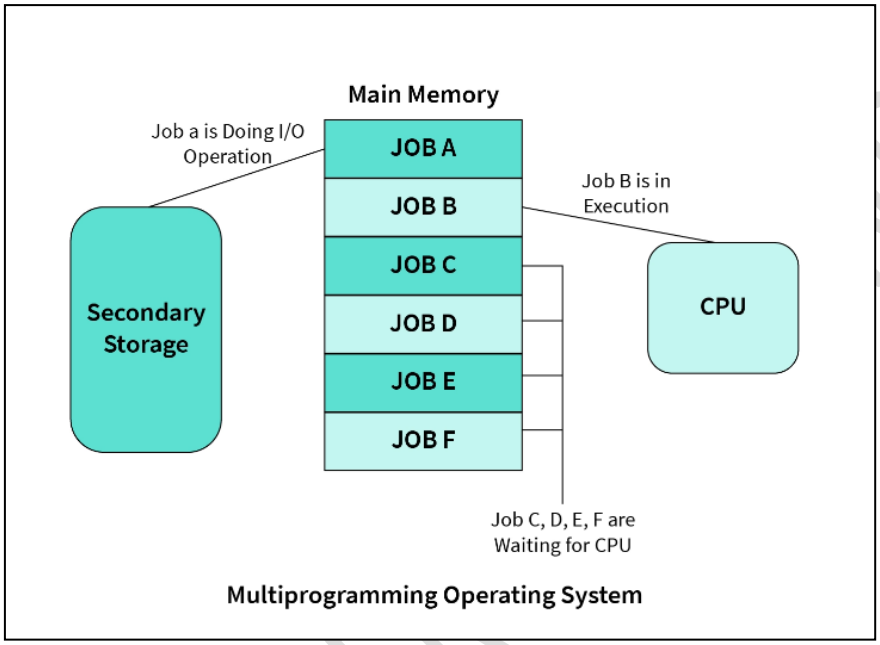
Multi-Programming Operating System

A multiprogramming operating system may run many programs on a single processor computer. If one program must wait for an input/output transfer in a multiprogramming operating system, the other programs are ready to use the CPU. As a result, various jobs may share CPU time. However, the execution of their jobs is not defined to be at the same time period.

When a program is being performed, it is known as a **"Task", "Process"**, and **"Job"**. Concurrent program executions improve system resource consumption and throughput as compared to serial and batch processing systems.

The primary goal of multiprogramming is to manage the entire system's resources. The key components of a multiprogramming system are the file

system, command processor, transient area, and I/O control system. As a result, multiprogramming operating systems are designed to store different programs based on sub-segmenting parts of the transient area. The resource management routines are linked with the operating system core functions.



A multiprogramming operating system, also known as a multitasking operating system, allows multiple programs or tasks to run concurrently on a computer system.

Advantages of Multiprogramming Operating System:

1. Increased CPU Utilization: With multiprogramming, the CPU can be utilized more efficiently. While one program is waiting for I/O operations or other tasks, another program can use the CPU, resulting in better overall CPU utilization.

2. Enhanced Throughput: By running multiple programs concurrently, a multiprogramming operating system can increase the overall throughput or the number of tasks completed in a given time period. This leads to improved efficiency and productivity.

3. Resource Sharing: Multiprogramming allows for effective sharing of system resources, such as memory, disk space, and peripherals, among multiple programs. This enables efficient utilization of resources and reduces idle time.

4. Increased Responsiveness: Users can experience better responsiveness as a result of multiprogramming. Even if one program is busy or unresponsive, other programs can continue to execute and respond to user requests, ensuring a smooth user experience.

5. Time-sharing: Multiprogramming facilitates time-sharing, where the CPU time is divided among multiple programs, giving each program a fair share of resources. This allows for the perception of simultaneous execution, making the system appear as if it is running multiple tasks concurrently.

Disadvantages of Multiprogramming Operating System:

1. Complexity: Managing multiple programs simultaneously adds complexity to the operating system. The system needs to handle task scheduling, memory allocation, and resource management efficiently, which can be challenging to implement and maintain.

2. Increased Overhead: The overhead involved in managing multiple programs can be significant. Context switching between programs, managing memory partitions, and coordinating resources require additional computational resources, which can reduce overall system performance.

3. Risk of System Instability: Running multiple programs concurrently increases the risk of system instability. A bug or error in one program can potentially affect other programs or the entire system, leading to crashes or unexpected behaviour.

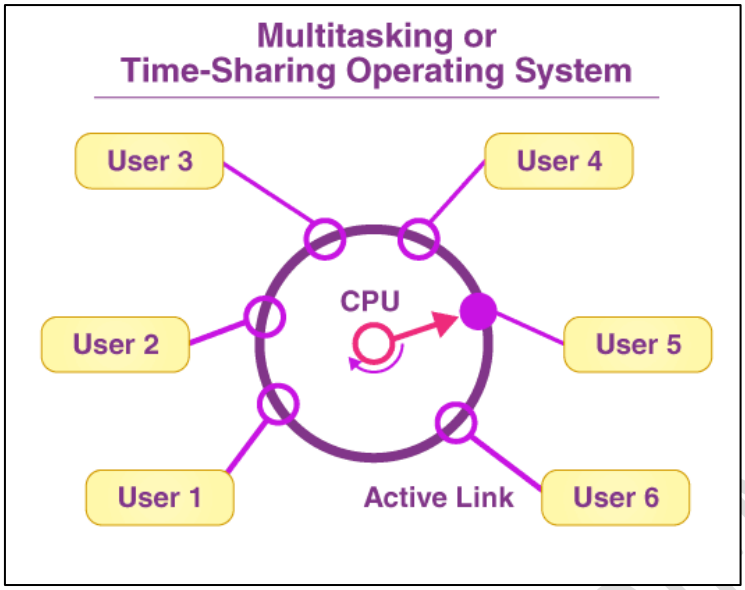
4. Resource Contentions: When multiple programs compete for the same resources, such as CPU time, memory, or I/O devices, resource contention can occur. This can result in delays, reduced performance, or even deadlocks if not managed properly.

5. Difficult Debugging: Debugging multiprogramming systems can be more challenging compared to single-program systems. Identifying and isolating issues related to a specific program or resource becomes complex when multiple programs are running simultaneously.

Time-Shared Operating System

A **time-shared operating system**, also known as a time-sharing operating system, is an operating system that allows multiple users or processes to share the CPU's time simultaneously. It provides the illusion of each user having dedicated access to the system's resources, even though the resources are actually shared among multiple users.

In a **time-shared operating system**, the CPU time is divided into small time intervals called time slices or quantum. Each user or process is allocated a time slice during which it can execute its tasks. When the time slice expires, the operating system interrupts the execution and switches to the next user or process in line.



The **significance of a time-sharing operating system** lies in its ability to provide several important benefits for both users and computer systems. Here are some key significances of time-sharing operating systems:

1. Resource Utilization: Time-sharing operating systems allow for efficient utilization of system resources. By allowing multiple users or processes to share the CPU time, memory, and other resources, the system can achieve higher resource utilization and maximize the overall throughput of the system.

2. Increased User Productivity: Time-sharing systems enhance user productivity by providing a responsive and interactive environment. Users can perform tasks concurrently, switch between applications seamlessly, and receive quick responses to their inputs. This enables users to accomplish more in less time, boosting their productivity.

3. Cost-Effectiveness: Time-sharing operating systems are cost-effective compared to providing individual dedicated systems to each user. By sharing system resources, organizations can optimize resource usage and reduce

hardware costs. It enables a larger number of users to access and utilize the same resources efficiently.

4. Multitasking Capabilities: Time-sharing systems support multitasking, allowing multiple tasks or programs to run concurrently. Users can execute different applications simultaneously, switch between them, and perform various tasks without the need for separate systems. This multitasking capability enhances productivity and efficiency.

5. Interactive and Real-time Responsiveness: Time-sharing systems provide an interactive and real-time responsive environment. Users can interact with the system, input commands, and receive immediate feedback. This is particularly important in scenarios such as command-line interfaces, where users need quick responses for efficient workflow.

6. Fair Resource Allocation: Time-sharing operating systems implement scheduling algorithms to allocate CPU time fairly among users or processes. This ensures that each user gets a fair share of the system's resources, preventing any single user from monopolizing the resources and promoting a sense of equity.

7. Convenient and Simultaneous Access: Time-sharing systems enable multiple users to access the system simultaneously. Users can share information, collaborate, and work on shared resources or files concurrently. This promotes collaboration and facilitates teamwork in various computing environments

8. Scalability: Time-sharing operating systems can easily scale to accommodate a growing number of users. As more users are added, the system can allocate resources and time slices accordingly, ensuring that each user receives their fair share. This scalability makes time-sharing systems suitable for large-scale computing environments.

Advantages of Time-shared Operating System:

- **Efficient Resource Utilization:** Time-sharing allows for better utilization of system resources by serving multiple users or processes concurrently.

- **Increased Productivity:** Users can perform tasks simultaneously and receive quick responses, enhancing productivity.

- **Cost-Effective**: Time-shared systems can be more cost-effective than providing individual dedicated systems to each user since resources are shared.

- **Sharing of Expensive Resources:** Expensive resources, such as high-performance CPUs or large memory capacities, can be shared among multiple users, making them accessible to a larger user base.

Disadvantages of Time-shared Operating System:

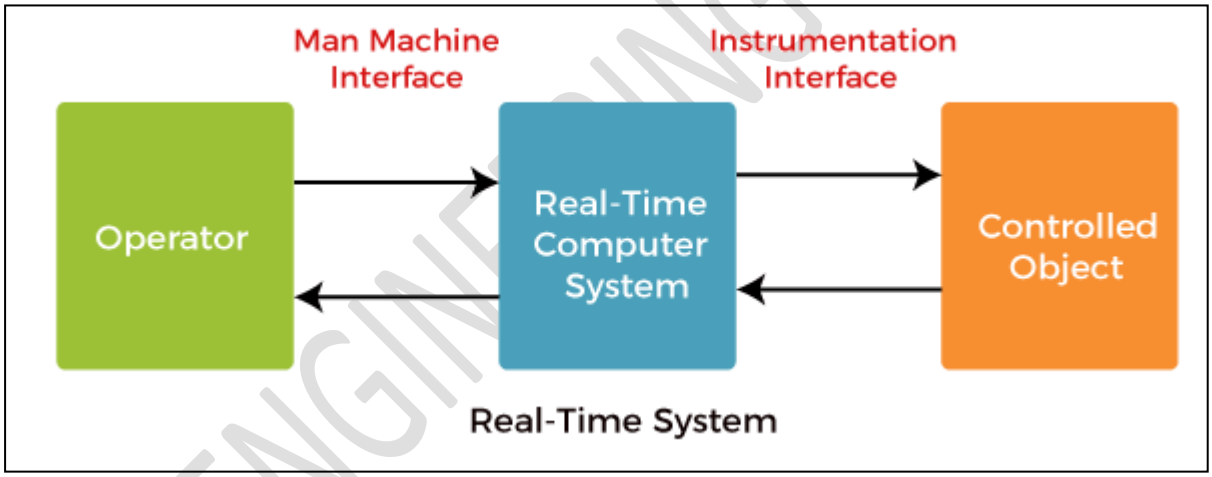
- **Performance Overhead**: The frequent context switching and resource allocation overhead in a time-shared system can impact overall performance and throughput.

- **Security and Privacy Concerns**: Sharing system resources raises security and privacy concerns, as one user's actions or programs may potentially affect others.

- **Complexity**: Time-shared systems require sophisticated scheduling and resource management algorithms, making their implementation and maintenance complex.

Real-time Operating System

A real-time operating system (RTOS) is an operating system designed to meet the specific requirements of real-time applications. Real-time systems are those that must respond to events within strict timing constraints. They are used in a wide range of applications, including industrial control systems, robotics, aerospace, medical devices, and embedded systems. The key characteristic of an RTOS is its ability to provide deterministic and predictable behaviour in meeting timing deadlines.



Hard Real Time OS

Soft Real Time OS

Hard Real Time OS

These operating systems guarantee that critical tasks be completed within a range of time.

For example, a robot is hired to weld a car body. If the robot welds too early or too late, the car cannot be sold, so it is a hard real-time system that requires complete car welding by robot hardly on the time., scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

Soft Real Time OS

This operating system provides some relaxation in the time limit.

For example – Multimedia systems, digital audio systems etc. Explicit, programmer-defined and controlled processes are encountered in real-time systems. A separate process is changed with handling a single external event. The process is activated upon occurrence of the related event signalled by an interrupt.

Multitasking operation is accomplished by scheduling processes for execution independently of each other. Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services. The processor is allocated to the highest priority processes. This type of schedule, called, priority-based preemptive scheduling is used by real-time systems.

Distributed Operating System

A **distributed operating system** is an operating system that runs on multiple machines and enables them to work together as a single system. It allows computers and devices in a network to communicate, share resources, and perform tasks in a coordinated manner. A distributed operating system provides transparency and abstraction, making the underlying network and hardware details transparent to the applications and users.

Distributed operating systems are commonly used in various scenarios, including large-scale computing environments, cloud computing, grid computing, and distributed database systems. They enable efficient resource utilization, fault tolerance, and collaboration among multiple machines, making it possible to tackle complex tasks that require distributed computing power and cooperation.

Distributed operating systems find applications in various domains where the coordination and sharing of resources across multiple machines are essential. Here are some common applications of distributed operating systems:

1. Distributed Computing: Distributed operating systems are extensively used in distributed computing environments. They enable the execution of computationally intensive tasks across multiple machines, harnessing the collective processing power to solve complex problems. Applications include scientific simulations, data analysis, and distributed rendering.

2. Cloud Computing: Distributed operating systems form the foundation of cloud computing platforms. Cloud service providers use distributed operating systems to manage and allocate computing resources dynamically. Users can

deploy and run applications on a distributed infrastructure without being concerned about the underlying hardware and network details.

3. Grid Computing: Distributed operating systems are employed in grid computing, where computing resources from multiple organizations or locations are pooled together. Grid systems use distributed operating systems to coordinate and schedule tasks across distributed resources for scientific research, high-performance computing, and data-intensive applications.

4. Distributed Database Systems: Distributed operating systems play a crucial role in distributed database systems. They provide mechanisms for data partitioning, replication, consistency, and coordination across multiple nodes to ensure efficient and reliable data storage and retrieval.

5. Content Delivery Networks (CDNs): CDNs utilize distributed operating systems to deliver web content efficiently to end-users. The content is cached and distributed across multiple servers located at strategic points in the network, reducing latency and improving content delivery performance.

6. Peer-to-Peer (P2P) Networks: P2P networks rely on distributed operating systems for peer coordination, resource sharing, and content distribution. Examples include file-sharing networks, decentralized cryptocurrency systems, and collaborative applications.

Types of Distributed Operating System

There are various types of Distributed Operating systems. Some of them are as follows

1. **Client-Server Systems**
2. **Peer-to-Peer Systems**

Client-Server System

This type of system requires the client to request a resource, after which the server gives the requested resource. When a client connects to a server, the server may serve multiple clients at the same time.

Client-Server Systems are also referred to as "Tightly Coupled Operating Systems". This system is primarily intended for multiprocessors and homogenous multicomputer. Client-Server Systems function as a centralized server since they approve all requests issued by client systems.

Server systems can be divided into two parts:

1. Computer Server System

This system allows the interface, and the client then sends its own requests to be executed as an action. After completing the activity, it sends a back response and transfers the result to the client.

2. File Server System

It provides a file system interface for clients, allowing them to execute actions like file creation, updating, deletion, and more.

Peer-to-Peer System

The nodes play an important role in this system. The task is evenly distributed among the nodes. Additionally, these nodes can share data and resources as needed. Once again, they require a network to connect.

The Peer-to-Peer System is known as a "Loosely Couple System". This concept is used in computer network applications since they contain a large number of processors that do not share memory or clocks. Each processor has its own local memory, and they interact with one another via a variety of communication methods like telephone lines or high-speed buses.

Mobile OS

Android -:

Android is an open source and Linux-based **Operating System** for mobile devices such as smartphones and tablet computers. Android was developed by the *Open Handset Alliance*, led by Google, and other companies.

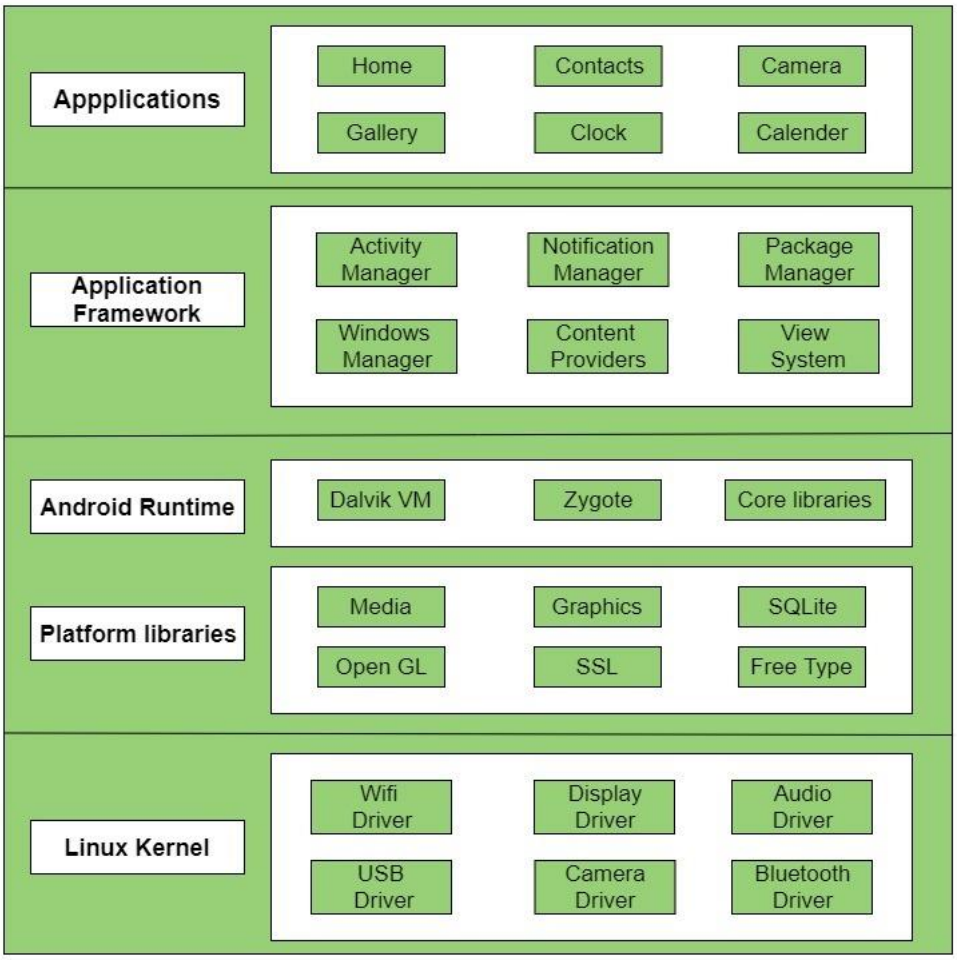
Android architecture contains different number of components to support any android device needs. Android software contains an open-source Linux Kernel having collection of number of C/C++ libraries which are exposed through an application framework services.

Among all the components Linux Kernel provides main functionality of operating system functions to smartphones and Dalvik Virtual Machine (DVM) provide platform for running an android application.

The main components of android architecture are following:-

- Applications
- Application Framework

- Android Runtime
- Platform Libraries
- Linux Kernel



Applications –

Applications is the top layer of android architecture. The pre-installed applications like home, contacts, camera, gallery etc and third party applications downloaded from the play store like chat applications, games etc. will be installed on this layer only.

It runs within the Android run time with the help of the classes and services provided by the application framework.

Application framework –

Application Framework provides several important classes which are used to create an Android application. It provides a generic abstraction for hardware access and also helps in managing the user interface with application resources. Generally, it provides the services with the help of which we can create a particular class and make that class helpful for the Applications creation.

It includes different types of services activity manager, notification manager, view system, package manager etc. which are helpful for the development of our application according to the prerequisite.

Application runtime –

Android Runtime environment is one of the most important part of Android. It contains components like core libraries and the Dalvik virtual machine(DVM). Mainly, it provides the base for the application framework and powers our application with the help of the core libraries.

Like Java Virtual Machine (JVM), **Dalvik Virtual Machine (DVM)** is a register-based virtual machine and specially designed and optimized for android to ensure that a device can run multiple instances efficiently. It depends on the layer Linux kernel for threading and low-level memory

management. The core libraries enable us to implement android applications using the standard JAVA or Kotlin programming languages.

Platform libraries –

The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.

- **Media** library provides support to play and record an audio and video formats.
- **Surface manager** responsible for managing access to the display subsystem.
- **SGL** and **OpenGL** both cross-language, cross-platform application program interface (API) are used for 2D and 3D computer graphics.
- **SQLite** provides database support and **FreeType** provides font support.
- **Web-Kit** This open source web browser engine provides all the functionality to display web content and to simplify page loading.
- **SSL (Secure Sockets Layer)** is security technology to establish an encrypted link between a web server and a web browser.

Linux Kernel –

Linux Kernel is heart of the android architecture. It manages all the available drivers such as display drivers, camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc. which are required during the runtime.

The Linux Kernel will provide an abstraction layer between the device hardware and the other components of android architecture. It is responsible for management of memory, power, devices etc.

The features of Linux kernel are:

- **Security:** The Linux kernel handles the security between the application and the system.

- **Memory Management:** It efficiently handles the memory management thereby providing the freedom to develop our apps.
- **Process Management:** It manages the process well, allocates resources to processes whenever they need them.
- **Network Stack:** It effectively handles the network communication.
- **Driver Model:** It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

Imp Questions :

1. Explain Mobile OS.
2. Explain with example real time os.
3. Explain client-server model and peer-to-peer model.

Assignment 1

1. Define Operating System with the help of a diagram.

2. List out any 4 functions of OS.

3. Explain dual mode operation in OS.

4. Describe OS as a resource manager.

5. Explain batch operating system with the help of a diagram.

6 Explain multi-programming operating system with the help of a diagram.

7. Explain time sharing operating system. List out any 2 advantages and disadvantages.

8. Explain multiprocessing operating system with its types.

9. Explain distributed OS in detail.

10. With neat diagram, explain real time os.

11. Explain android os with its architecture.

12. Write a short note on DOS and UNIX.

Chapter 1 Checklist

Sr. No.	Topics	Status (clear / doubt)
1.	Concept of operating system	
2.	Views of OS	
3.	Dual mode operation	
4.	Resource Management	
5.	Batch OS	
6.	Multi-programming OS	
7.	Time sharing OS	
8.	Real time OS	
9.	Distributed OS	
10.	Mobile OS	
11.	Command-line OS	
12.	GUI Based OS	

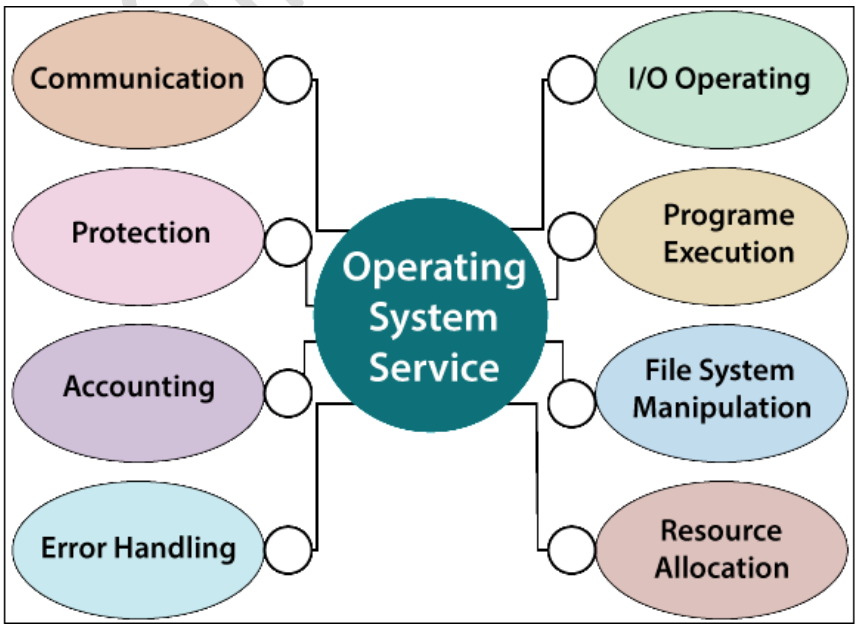
Doubts Column :

2. Services & Components of OS

An operating system (OS) is a software program that acts as an interface between a computer's hardware and its users. It is a fundamental component of any computer system and manages the overall operations of a computer or a network of computers.

Services of Operating System

Operating systems provide various services to manage and coordinate the resources and operations of a computer system. Here are some of the different services typically offered by operating systems:



1. Process Management: The operating system manages processes, which are instances of executing programs. It schedules and allocates system resources to processes, handles process creation and termination, and provides mechanisms for inter-process communication and synchronization.

2. Memory Management: This service is responsible for managing the computer's memory resources. It tracks and allocates memory space to processes, handles memory allocation and deallocation, and provides mechanisms for virtual memory management and memory protection.

3. File System Management: The file system service provides a hierarchical structure for organizing and storing files on storage devices. It manages file creation, deletion, and access permissions, as well as directory management and file metadata (e.g., file size, timestamps).

4. Device Management: Operating systems handle interactions with hardware devices, such as keyboards, mice, printers, and storage devices. They provide device drivers to communicate with these devices, handle input/output operations, and manage device resources.

5. User Interface: The operating system provides a user interface that allows users to interact with the computer system. This can include command-line interfaces (CLI), graphical user interfaces (GUI), or a combination of both, enabling users to execute commands, launch applications, and manage files.

6. Network Management: Operating systems include networking services that enable communication and data transfer over computer networks. They provide network protocols, manage network connections, handle data transmission, and support various network configurations.

7. Security Services: Operating systems implement security mechanisms to protect the system and its resources from unauthorized access and ensure data integrity. These services include user authentication, access control, encryption, and auditing.

8. Error Handling: The operating system monitors system errors and exceptions, such as hardware faults or software crashes, and provides error handling mechanisms to recover from or mitigate these issues. This can involve logging errors, generating error messages, and taking appropriate actions to maintain system stability.

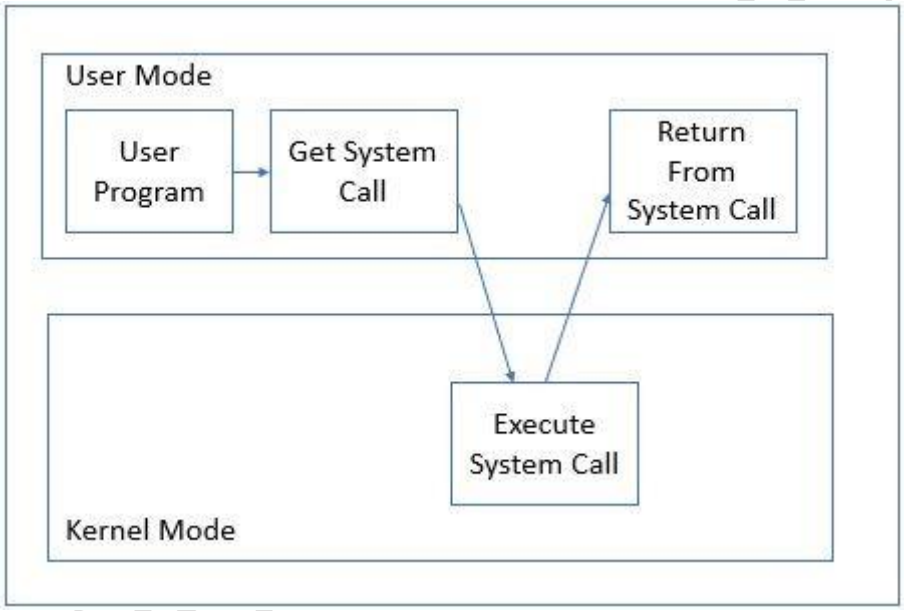
9. Scheduling: The operating system employs scheduling algorithms to determine the order and priority of executing processes. It optimizes resource allocation, ensures fair access to resources, and manages the execution of concurrent processes.

10. System Utilities: Operating systems offer a range of utility programs that assist in system management and maintenance. These utilities may include disk management tools, performance monitoring tools, backup and restore utilities, and system configuration tools.

System Calls

System calls are fundamental functions provided by an operating system that allow user-level programs to request services from the operating system kernel. They act as an interface between user applications and the underlying operating system, enabling applications to access and utilize various system resources and services.

System calls provide a controlled and secure mechanism for applications to interact with the underlying operating system, ensuring that critical operations and access to protected resources are regulated and managed. They abstract the complexity of low-level hardware operations and provide a high-level interface for application developers to utilize the features and services offered by the operating system.



When a program needs to perform a privileged operation or access a system resource, it invokes a system call, which transfers control from user space to the kernel. The kernel then performs the requested operation on behalf of the program and returns the result back to the program.

Here are some common examples of system calls:

1. File System Operations: System calls such as "open," "read," "write," and "close" are used for file manipulation. They allow programs to create, open, read from, write to, and close files.

2. Process Management: System calls like "fork," "exec," "exit," and "wait" are used for managing processes. They allow programs to create new processes, replace the current process with a different program, terminate processes, and wait for process termination.

3. Device Management: System calls such as "open," "read," "write," and "ioctl" are used to interact with hardware devices. They enable programs to access devices such as printers, network interfaces, and serial ports.

4. Memory Management: System calls like "brk" and "mmap" are used for memory management. They allow programs to allocate and deallocate memory dynamically, map files into memory, and modify memory protection settings.

5. Network Operations: System calls such as "socket," "connect," "send," and "recv" are used for network communication. They enable programs to establish network connections, send and receive data over networks, and perform network-related operations.

6. Time and Date Management: System calls like "time," "gettimeofday," and "sleep" are used to obtain and manipulate system time and dates.

7. Process Control: System calls like "kill" and "signal" are used for process control. They allow programs to send signals to processes, handle signal events, and modify signal behavior.

8. Interprocess Communication (IPC): System calls such as "pipe," "shmget," "msgget," and "semget" are used for interprocess communication. They enable communication and synchronization between processes using mechanisms like pipes, shared memory, message queues, and semaphores.

Imp questions

1. Explain different services of operating system.

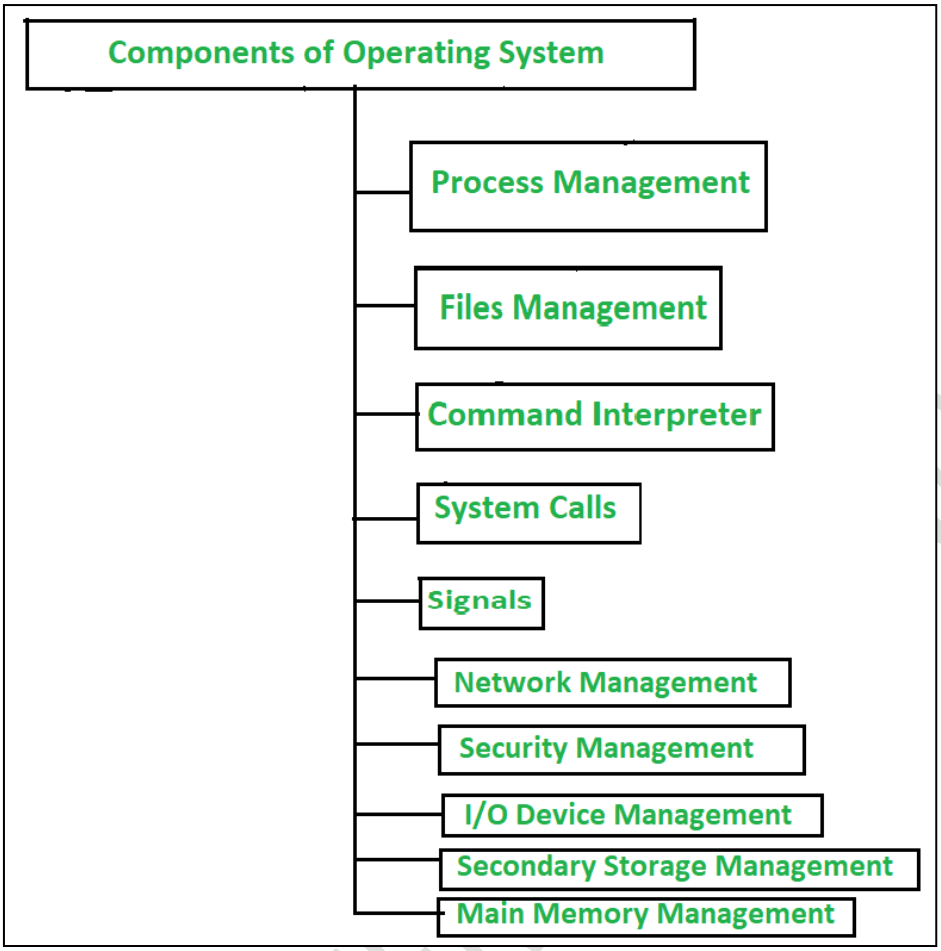
2. Explain types of system calls with example of each category.

Components of OS

An Operating system is an interface between users and the hardware of a computer system. It is a system software that is viewed as an organized collection of software consisting of procedures and functions, providing an environment for the execution of programs.

Important Components of the Operating System:

- Process management
- Files management
- Command Interpreter
- System calls
- Signals
- Network management
- Security management
- I/O device management
- Secondary storage management
- Main memory management



Process Management :

A process is a program in execution. It consists of the followings:

- Executable program
- Program's data
- Stack and stack pointer
- Program counter and other CPU registers
- Details of opened files

Files Management :

Files are used for long-term storage. Files are used for both input and output. Every operating system provides a file management service. This file management service can also be treated as an abstraction as it hides the

information about the disks from the user. The operating system also provides a system call for file management. The system call for file management includes –

- File creation
- File deletion
- Read and Write operations

Command Interpreter :

There are several ways for users to interface with the operating system. One of the approaches to user interaction with the operating system is through commands. Command interpreter provides a **command-line interface**. It allows the user to enter a command on the command line prompt (cmd).

System Calls :

System calls provide an interface to the services made by an operating system. The user interacts with the operating system programs through System calls. These calls are normally made available as library functions in high-level languages such as C, Java, Python etc.

Network Management :

Network management is a fundamental concept of computer networks. Network Management Systems is a software application that provides network administrators with information on components in their networks. It ensures the quality of service and availability of network resources. It also examines the operations of a network, reconstructs its network configuration, modifies it for improving performance of tasks.

Security Management:

The security mechanisms in an operating system ensure that authorized programs have access to resources, and unauthorized programs have no access to restricted resources. Security management refers to the various processes where the user changes the file, memory, CPU, and other hardware resources that should have authorization from the operating system.

I/O Device Management :

The I/O device management component is an I/O manager that hides the details of hardware devices and manages the main memory for devices using cache and spooling. This component provides a buffer cache and general device driver code that allows the system to manage the main memory and the hardware devices connected to it. It also provides and manages custom drivers for particular hardware devices.

The purpose of the I/O system is to hide the details of hardware devices from the application programmer. An I/O device management component allows highly efficient resource utilization while minimizing errors and making programming easy on the entire range of devices available in their systems.

Secondary Storage Management :

Broadly, the secondary storage area is any space, where data is stored permanently and the user can retrieve it easily. Your computer's hard drive is the primary location for your files and programs. Other spaces, such as CD-ROM/DVD drives, flash memory cards, and networked devices, also provide secondary storage for data on the computer. The computer's main memory (RAM) is a volatile storage device in which all programs reside, it provides only temporary storage space for performing tasks. Secondary storage refers to the media devices other than RAM (e.g. CDs, DVDs, or hard disks) that provide additional space for permanent storing of data and software programs which is also called non-volatile storage.

Main memory management :

Main memory is a flexible and volatile type of storage device. It is a large sequence of bytes and addresses used to store volatile data. Main memory is also called Random Access Memory (RAM), which is the fastest computer storage available on PCs. It is costly and low in terms of storage as compared to secondary storage devices. Whenever computer programs are executed, it is temporarily stored in the main memory for execution. Later, the user can permanently store the data or program in the secondary storage device.

Operating System Tools

Task Scheduler -:

Task Scheduler is a crucial component of the Windows operating system that allows users to automate the execution of tasks at predetermined times or in response to specific events. It provides a convenient way to schedule and manage various system tasks, programs, scripts, and administrative functions without the need for manual intervention. Here are some key aspects and functionalities of the Task Scheduler in Windows:

1. Task Creation and Management: Task Scheduler allows users to create tasks by specifying the desired triggers, actions, and conditions.

2. Task Monitoring and History: Task Scheduler provides a comprehensive interface for monitoring and managing scheduled tasks. Users can view the list

of tasks, check their status, modify task properties, enable or disable tasks, and delete tasks.

3. Security and Permissions: Task Scheduler incorporates security features to control access to tasks and ensure their execution in a secure environment. Users can assign different security permissions to tasks, determining who can create, modify, or delete tasks.

4. Advanced Task Settings: Task Scheduler offers various advanced settings to fine-tune task behavior. These settings include options to delay task execution, configure task repetition patterns, set task priorities, define task timeouts, and specify how to handle task conflicts. Users can also set up notifications to receive email alerts or display custom messages upon task completion or failure.

Performance Monitor -:

The Performance Monitor is a built-in utility in the Windows operating system that allows users to monitor and analyze the performance of various system components and processes. It provides valuable insights into the health and efficiency of a computer system, helping users diagnose performance issues and optimize system resources.

The Performance Monitor is a powerful tool for system administrators, IT professionals, and advanced users who need to monitor, analyze, and optimize system performance in Windows. By using Performance Monitor effectively, users can identify performance bottlenecks, troubleshoot issues, and make informed decisions to enhance the overall performance and stability of their Windows-based systems.

Assignment 2

1. Describe services of Operating system.
2. Explain all components of os.
3. What is system call ? List out all types of system call.
4. List out all types of system call and explain any one of them.
5. Write a short note on Task Scheduler & Performance Monitor.

Chapter 2 Checklist

Sr. No.	Topics	Status (clear / doubt)
1.	Services of Operating System	
2.	System calls	
3.	Components of Operating System	
4.	Task scheduler	
5.	Performance monitor	

Doubts Column :

3. Process Management

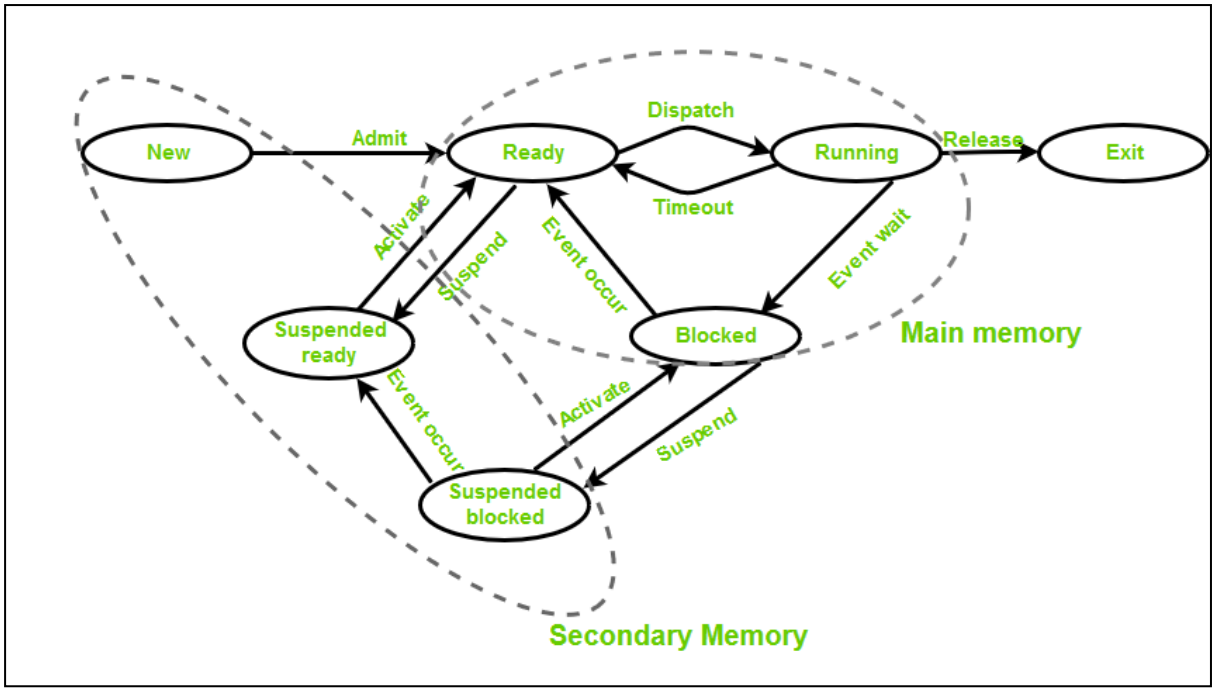
Process -:

In an operating system, a process is a unit of activity that represents the execution of a program or a task. It is an instance of a computer program that is being executed and managed by the operating system. Processes are essential for multitasking, as they allow the operating system to execute multiple tasks concurrently.

Program Execution: A process corresponds to the execution of a program. When a program is launched, the operating system creates a process to manage its execution. Each process has its own memory space, program counter, and resources.

Process States -:

A process has several stages that it passes through from beginning to end. There must be a minimum of five states. Even though during execution, the process could be in one of these states, the names of the states are not standardized. Each process goes through several stages throughout its life cycle.



Process States in Operating System

The states of a process are as follows:

- **New (Create):** In this step, the process is about to be created but not yet created. It is the program that is present in secondary memory that will be picked up by OS to create the process.
- **Ready:** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue called ready queue for ready processes.
- **Run:** The process is chosen from the ready queue by the CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or Wait:** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already

acquired) it enters the blocked or waits for the state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.

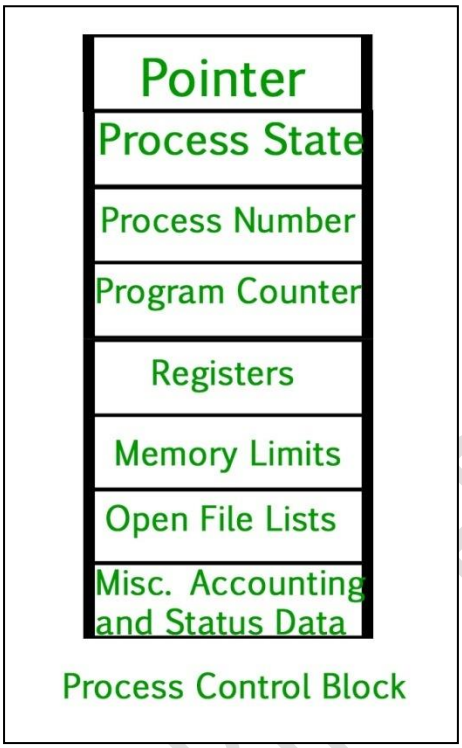
- **Terminated or Completed:** Process is killed as well as PCB is deleted. The resources allocated to the process will be released or deallocated.
- **Suspend Ready:** Process that was initially in the ready state but was swapped out of main memory(refer to Virtual Memory topic) and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a ready state whenever the process is again brought onto the main memory.
- **Suspend wait or suspend blocked:** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.

Process Control Block

The Process Control Block (PCB), also known as the Task Control Block (TCB), is a data structure used by an operating system to manage and track information about a specific process. It contains essential details and control information that the operating system needs to manage the process effectively. Each process in the system has its own PCB, and the operating system uses the PCB to perform process management and scheduling tasks.

The Process Control Block is a vital data structure used by the operating system to manage and control processes efficiently. It allows the operating system to maintain and retrieve the necessary information for process scheduling, context switching, resource allocation, and interprocess communication. By maintaining

a PCB for each process, the operating system can effectively manage the execution and control of multiple processes concurrently.



Here are the key components and information typically found in a Process Control Block:

1. Process ID (PID): A unique identifier assigned to each process in the system. It distinguishes one process from another and is used by the operating system to reference and manage the process.

2. Process State: Indicates the current state of the process, such as running, ready, blocked, or terminated. The state is updated as the process moves through different phases of execution and interacts with the operating system.

3. Program Counter (PC): The Program Counter holds the address of the next instruction to be executed by the process. It allows the operating system to keep track of the execution progress of the process.

4. CPU Registers: The PCB contains the values of various CPU registers associated with the process, such as the accumulator, stack pointer, and index registers. These registers hold the intermediate results, program variables, and execution context of the process.

5. Process Priority: A priority level assigned to the process, indicating its relative importance or scheduling preference. The operating system uses the priority value to determine the order in which processes are scheduled and allocated CPU time.

6. Memory Management Information: Information related to the memory allocation of the process, including the base and limit registers or pointers that define the process's memory address space. This information helps the operating system manage memory and ensure memory protection between processes.

7. I/O Status Information: Contains details about I/O operations associated with the process, such as open file descriptors, pending I/O requests, and I/O device status. This information allows the operating system to manage and coordinate I/O operations effectively.

8. Accounting Information: The PCB may include accounting data, such as the amount of CPU time used by the process, the number of times it has been executed, or other statistics related to resource utilization. This information assists in performance analysis, billing, and system monitoring.

9. Process Scheduling Information: Additional information related to process scheduling, including scheduling queues or pointers to the next process in a scheduling queue. This information aids the operating system in making scheduling decisions and managing the order of process execution.

Imp Questions -:

1. Define process . List out different process states.
2. Describe process state diagram.
3. Explain process control block with the help of a diagram.

What is Process Scheduling ?

Process scheduling is a fundamental function of an operating system that involves determining the order and allocation of CPU time to processes in a multitasking environment. It is the process by which the operating system selects and assigns processes for execution on the CPU, ensuring efficient utilization of system resources and meeting performance objectives.

Process scheduling aims to achieve the following goals:

1. Fairness: The scheduler ensures that each process gets a fair share of CPU time, preventing any particular process from monopolizing system resources. Fairness promotes an equitable distribution of CPU resources among processes.

2. Efficiency: The scheduler aims to maximize CPU utilization, keeping the CPU busy with productive work as much as possible. By minimizing idle time and maximizing the number of executed processes, efficiency is improved.

3. Responsiveness: The scheduler strives to provide a responsive system by minimizing the time it takes for processes to start execution or respond to

events. It prioritizes processes that require immediate attention, ensuring that interactive tasks and critical operations are handled promptly.

4. Throughput: The scheduler aims to maximize the number of processes completed per unit of time, known as throughput. Higher throughput indicates improved system efficiency and productivity.

5. Resource Optimization: Process scheduling considers various system resources, such as CPU, memory, and I/O devices, when making scheduling decisions. It aims to balance the allocation of these resources to meet overall performance goals.

Scheduling Queue

Ready Queue:

The ready queue is a fundamental concept in process scheduling within an operating system. It is a data structure that holds the processes that are ready and waiting to be executed on the CPU. Processes in the ready queue are in a state where they can be allocated CPU time and can potentially start executing.

Key points about the ready queue:

1. Purpose: The primary purpose of the ready queue is to hold processes that are waiting to be scheduled for execution on the CPU. These processes have met the necessary criteria to run, such as having their required resources available or completing any required I/O operations.

2. Organization: The ready queue is typically implemented as a queue data structure, where processes are added to the back of the queue and removed from the front. This follows the First-Come, First-Served (FCFS) principle, where the process that arrives first is scheduled first.

3. Scheduling Decision: When the CPU becomes available, the scheduler selects a process from the ready queue based on the scheduling algorithm in use. The chosen process is then dispatched to the CPU for execution.

4. Process State: Processes in the ready queue are in a ready state, indicating that they are prepared to execute but are waiting for CPU allocation. Once a process is selected from the ready queue, it transitions to the running state and starts executing on the CPU.

Device Queue:

The device queue, also known as the I/O queue or waiting queue, is a data structure used by the operating system to manage processes that are waiting for access to I/O devices. It holds processes that are waiting for I/O operations to complete before they can proceed with their execution.

Key points about the device queue:

1. Purpose: The device queue is used to hold processes that are waiting for I/O operations to be performed on a specific device, such as reading from or writing to a disk, accessing a printer, or interacting with other peripherals. These processes are unable to proceed until the requested I/O operation is completed.

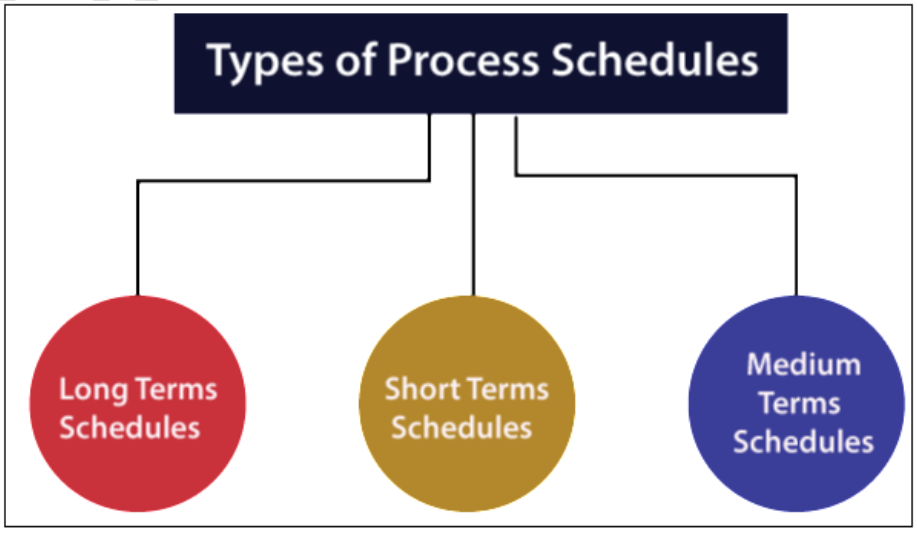
2. Organization: Similar to the ready queue, the device queue is typically implemented as a queue data structure. Processes are added to the end of the queue when they are waiting for an I/O operation and are removed from the front when the operation is completed.

3. Process State: Processes in the device queue are in a blocked or waiting state. This state indicates that they are unable to continue execution until the requested I/O operation is finished.

4. I/O Scheduling: The device queue, in conjunction with the I/O scheduler, manages the order in which processes access I/O devices. The scheduler determines the sequence in which processes are granted access to the device, aiming to optimize device utilization and minimize waiting times.

5. Device Completion: When an I/O operation is completed for a process in the device queue, the process is moved to the ready queue, indicating that it can resume execution on the CPU.

Schedulers



Schedulers in an operating system are responsible for making decisions about process execution, resource allocation, and process management. They determine which processes should run, in what order, and for how long. Schedulers play a crucial role in achieving efficient utilization of system resources, responsiveness, fairness, and meeting performance objectives. Here are the main types of schedulers found in operating systems:

1. Long-Term Scheduler (Admission Scheduler or Job Scheduler):

The long-term scheduler decides which processes are admitted into the system from the pool of waiting processes. Its primary goal is to maintain an optimal number of processes in the system, taking into account factors such as available resources, system load, and overall system performance. The long-term scheduler controls the degree of multiprogramming by determining when to bring new processes into the system.

2. Short-Term Scheduler (CPU Scheduler):

The short-term scheduler determines which processes in the ready queue should be allocated CPU time for execution. It is responsible for making frequent and fast scheduling decisions, typically on a millisecond or microsecond timescale. The short-term scheduler selects a process from the ready queue and dispatches it to the CPU, often using scheduling algorithms such as Round Robin, First-Come, First-Served, Shortest Job Next, or Priority Scheduling.

3. Medium-Term Scheduler:

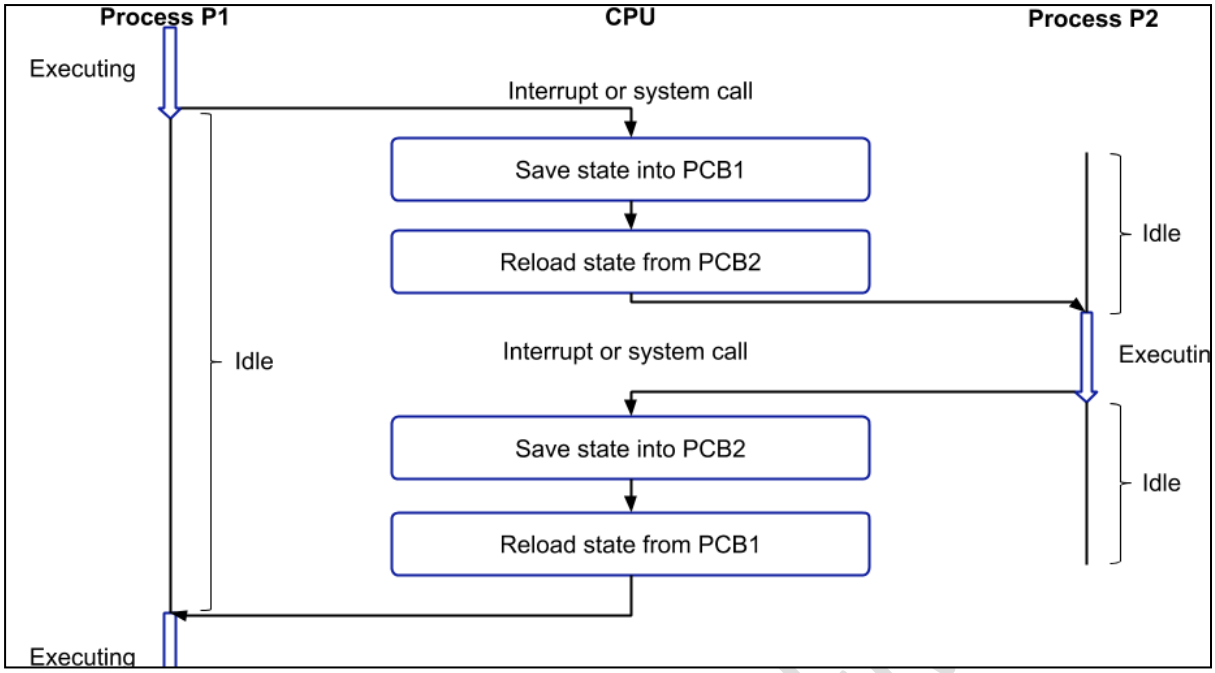
The medium-term scheduler, also known as the swapping scheduler, is an optional scheduler found in some operating systems. Its primary function is to handle processes that are partially or fully swapped out from main memory to secondary storage (e.g., disk). The medium-term scheduler decides when to swap processes in and out of main memory based on factors like memory availability, process priority, or I/O demands. This helps manage the overall memory usage and improve system performance.

Each scheduler performs a specific function and operates at a different level of granularity in the operating system. They work together to manage process execution, resource allocation, and ensure the efficient operation of the system.

Context Switch

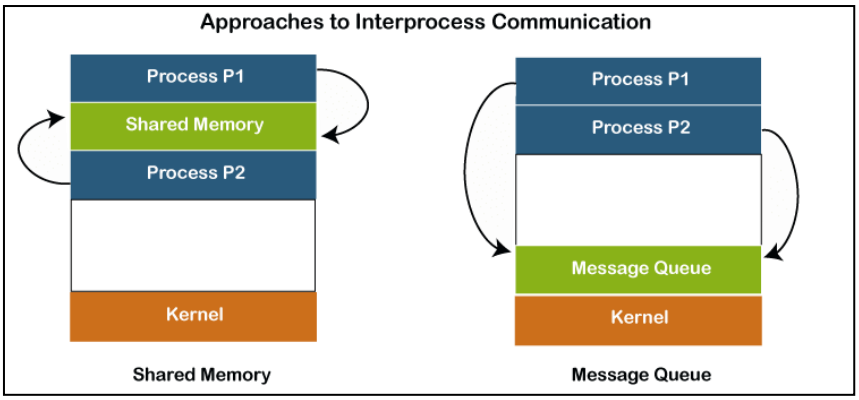
A context switch is a fundamental operation performed by an operating system to save and restore the state of a process or thread when there is a need to switch the CPU's execution from one process to another. It involves storing the current execution context of a process, including its register values, program counter, and other relevant information, and loading the saved context of another process to resume its execution. Context switches allow the operating system to provide the illusion of concurrent execution and support multitasking in a system.

Context switches are essential for multitasking and providing a responsive and concurrent environment in an operating system. They allow the CPU to efficiently allocate its processing power to multiple processes or threads, enabling concurrent execution, time-sharing, and the illusion of parallelism in a system.



Inter-process Communication

Inter-process communication (IPC) refers to the mechanisms and techniques used by operating systems to enable communication and data exchange between different processes running concurrently. IPC plays a vital role in coordinating and synchronizing the activities of various processes, allowing them to share information, cooperate, and achieve common goals. There are several methods of IPC, including the following:



1. Shared Memory : Shared memory is a fast and efficient method of IPC. It involves creating a region of memory that can be accessed by multiple processes. These processes can read from and write to the shared memory space, allowing them to share data and communicate with each other. However, careful synchronization mechanisms, such as semaphores or mutexes, are necessary to prevent conflicts when multiple processes access the shared memory simultaneously.

2. Message Passing : Message passing involves sending and receiving messages between processes. In this method, processes explicitly send data to one another through communication primitives provided by the operating system. Messages can be sent using either **synchronous** or **asynchronous** communication. Synchronous communication requires the sender to wait until the recipient acknowledges the receipt of the message, whereas asynchronous communication allows the sender to continue its execution immediately after sending the message, without waiting for a response.

3. Pipes and FIFOs : Pipes and FIFOs (First-In-First-Out) are forms of inter-process communication that are typically used for communication between related processes. A pipe is a unidirectional communication channel that allows data to flow in one direction. FIFOs, also known as named pipes, are similar to pipes but can be accessed by unrelated processes. Pipes and FIFOs provide a simple and straightforward method of IPC, with one process writing data into the pipe and another process reading it.

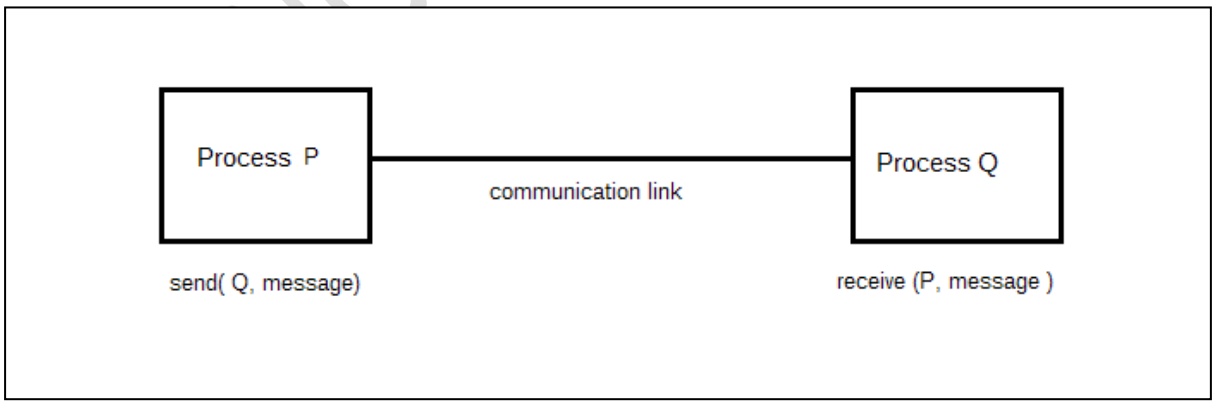
4. Sockets : Sockets are a network-based form of IPC that allows communication between processes running on different machines connected over a network. Sockets can be used for both inter-process communication on the same machine (using local loopback addresses) and communication between

processes on different machines. They provide a flexible and powerful means of IPC and are widely used in client-server applications.

5. Signals : Signals are software interrupts used by an operating system to notify a process of an event or an exceptional condition. Processes can send and receive signals to communicate with each other. For example, one process can send a signal to another process to request its termination or to handle a specific event. Signals are often used for simple forms of IPC but are limited in the amount of data they can convey.

Message Passing :

Message passing is a form of inter-process communication (IPC) that involves the exchange of messages between processes. In this method, processes communicate by sending and receiving messages through predefined communication primitives provided by the operating system or programming language.



Message passing can be achieved through various mechanisms, such as direct function calls, remote procedure calls (RPC), or message queues. The sending process encapsulates data or instructions into a message and sends it to the

receiving process. The receiving process then extracts and processes the message contents.

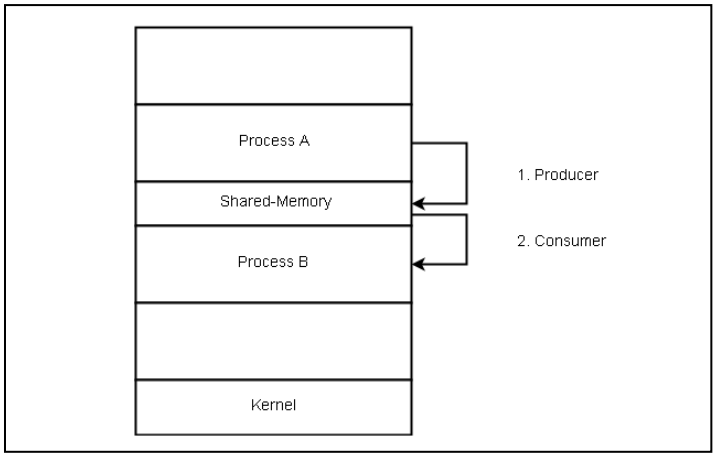
Message passing offers several advantages. First, it provides a clear separation between processes, as they do not need to share memory directly. This enhances modularity, encapsulation, and security, as processes can operate independently and communicate only through well-defined interfaces. Additionally, message passing is generally platform-independent, allowing processes running on different machines or operating systems to communicate with each other.

Shared Memory :

Shared memory is an IPC mechanism that allows multiple processes to access and share a common region of memory. In shared memory communication, a specific portion of memory is allocated and mapped into the address space of multiple processes. These processes can read from and write to the shared memory, enabling them to exchange data and communicate efficiently.

Shared memory provides several benefits. It is generally faster than other IPC methods since it avoids the need for data copying between processes. As processes can directly access the shared memory, it is suitable for scenarios that require high-performance data sharing, such as inter-process coordination or inter-thread communication within a single machine.

However, shared memory also presents challenges. It requires careful synchronization and coordination mechanisms, such as locks, semaphores, or mutexes, to avoid data races and ensure consistent access. Proper synchronization is essential to prevent conflicts when multiple processes access the shared memory simultaneously.



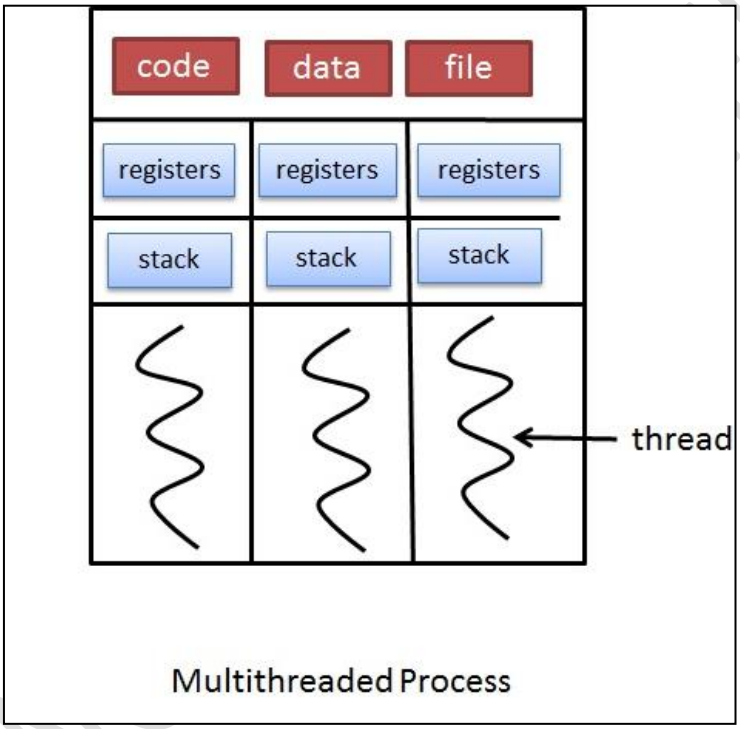
Imp Questions -:

1. Explain process scheduling.
2. Define schedulers with its types.
3. Explain context switch.
4. Explain inter-process communication with the help of a diagram.

Threads

In operating systems, a thread refers to a lightweight unit of execution within a process. A thread is also known as a "lightweight process" because it shares the same memory space as other threads within the same process, allowing them to access and modify the same data.

Threads are used to achieve concurrent execution within a single process. Unlike processes, which are independent entities with their own memory space, threads exist within a process and share resources such as memory, file descriptors, and other process-related attributes. Threads within a process can communicate and coordinate with each other more efficiently compared to separate processes, as they can directly access shared data.



Threads have the following characteristics:

1. Concurrency : Threads allow multiple units of execution to progress concurrently within a single process. This concurrency enables tasks to be performed simultaneously, leading to increased efficiency and responsiveness.

2. Shared Resources : Threads within the same process share the same resources, including memory, file descriptors, and open files. This shared memory enables efficient communication and data exchange between threads.

3. Lightweight : Threads are lightweight compared to processes. They require fewer system resources to create, switch, and manage. This lightweight nature allows for faster thread creation and context switching.

4. Scheduling : Threads within a process are scheduled by the operating system's scheduler. The scheduler allocates CPU time to each thread, allowing them to execute in a time-sliced manner. Thread scheduling can be either pre-emptive (where the scheduler determines when to switch between threads) or cooperative (where threads yield control voluntarily).

Threads provide several advantages in operating systems and application development:

1. Responsiveness : Threads allow for concurrent execution, making applications more responsive to user interactions. For example, a user interface can remain responsive while a background thread performs a time-consuming task.

2. Efficiency : Threads enable efficient utilization of system resources. They can take advantage of multiprocessor systems by running on different cores simultaneously, improving overall system performance.

3. Resource Sharing : Threads within a process can share data directly, avoiding the need for costly inter-process communication mechanisms. This simplifies the development of shared data structures and enhances communication and coordination between different parts of an application.

4. Modularity : Threads provide a modular approach to program design. By dividing complex tasks into multiple threads, each responsible for a specific aspect, applications become easier to design, understand, and maintain.

Thread Lifecycle -:

The life cycle of a thread describes the different stages that a thread goes through during its existence. The thread life cycle typically includes the following states:

Thread life cycle management involves activities such as creating threads, starting them, synchronizing their execution, and terminating them appropriately. Proper synchronization mechanisms and coordination are crucial to ensure correct and efficient thread interactions and to avoid race conditions or deadlocks.

1. New : In the new state, a thread is created but has not yet started executing. The necessary resources for the thread, such as its stack space and program counter, have been allocated, but the thread has not been scheduled by the operating system to run.

2. Runnable : Once the thread is ready to execute, it enters the runnable state. In this state, the thread is eligible to run and can be scheduled by the operating system's thread scheduler. However, it does not guarantee immediate execution as it competes with other runnable threads for CPU time.

3. Running : When a thread is selected by the scheduler to execute on a CPU core, it enters the running state. In this state, the actual execution of the thread's instructions takes place. The thread remains in the running state until it voluntarily yields the CPU or its time slice expires.

4. Blocked : Threads can transition to the blocked (or waiting) state if they need to wait for a particular event or condition to occur. For example, a thread might block if it requests I/O operations or synchronization primitives like locks or semaphores. While blocked, the thread is not eligible for CPU time and remains in this state until the event or condition it is waiting for is satisfied.

5. Terminated : A thread enters the terminated state when it finishes its execution or is explicitly terminated by the program. In this state, the thread has completed its tasks, and its resources are released by the operating system. The terminated thread can no longer be scheduled for execution.

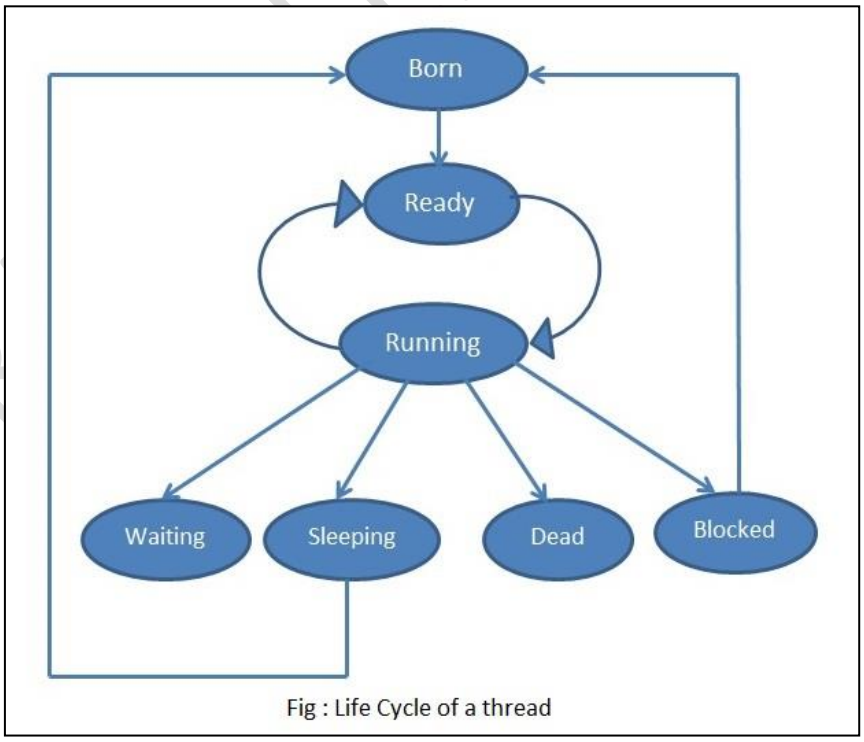
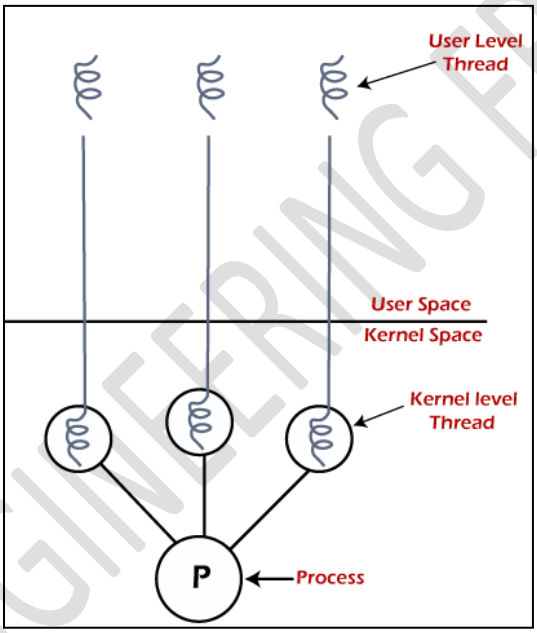


Fig : Life Cycle of a thread

Types of Threads

The multithreading model in an operating system allows multiple threads to exist within a single process, enabling concurrent execution and sharing of resources. In a multithreaded environment, each thread represents an independent flow of execution with its own program counter, stack, and thread-specific data, but they share the same memory space and other process-related resources.



There are several multithreading models used in operating systems, including:

1. User-Level Threads (ULT) : User-level threads are managed entirely by user-level thread libraries or programming language runtimes, without direct support from the operating system kernel. The thread management and scheduling are implemented in user space, making thread creation, switching, and synchronization faster. However, ULT models are typically less efficient

when it comes to blocking system calls or accessing operating system resources, as a blocked thread can potentially block all other threads in the same process.

2. Kernel-Level Threads (KLT) : Kernel-level threads are managed by the operating system kernel. The kernel provides explicit support for thread creation, scheduling, and synchronization. Each thread is treated as a separate entity by the operating system, and the kernel is responsible for allocating CPU time to each thread. KLT models are more robust and provide better support for concurrency, as blocking operations in one thread do not affect other threads in the process. However, thread management operations may involve higher overhead due to system calls and kernel involvement.

Multi-threading Model

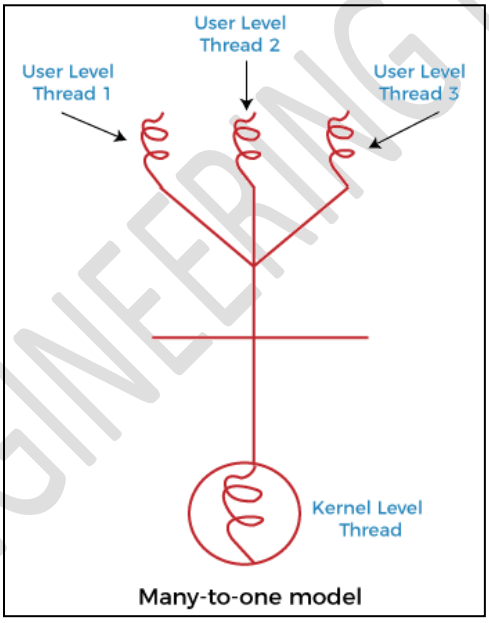
A multithreading model in an operating system (OS) allows multiple threads of execution to exist within a single process. Multithreading enables concurrent execution of multiple threads, allowing tasks to be performed simultaneously and efficiently utilizing available system resources.

The choice of the multithreading model depends on various factors, such as the application requirements, performance goals, and the level of control and concurrency desired. Each model has its advantages and trade-offs in terms of concurrency, overhead, resource usage, and ease of programming.

Multithreading models are crucial for developing efficient concurrent applications, as they enable tasks to be performed simultaneously, improve responsiveness, and fully utilize the available processing power in modern systems.

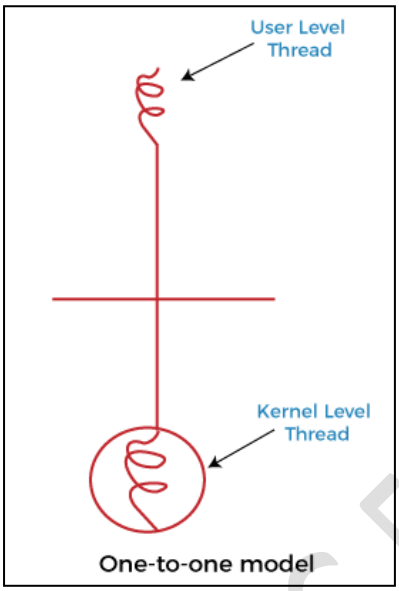
There are several multithreading models used in operating systems, including the following:

1. Many-to-One (User-Level Threads) : In this model, multiple user-level threads are managed by a single kernel-level thread. The OS is unaware of the existence of user-level threads and schedules the process as a whole. User-level thread management is handled by a thread library or runtime environment within the process. Context switching between user-level threads is fast since it doesn't involve kernel involvement, but blocking system calls can potentially block all threads in the process.

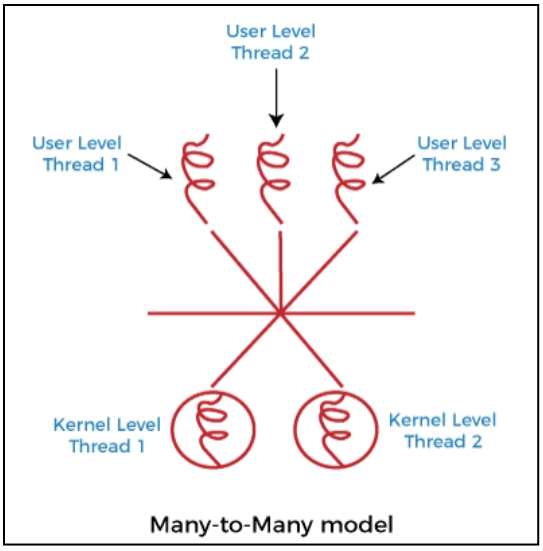


2. One-to-One (Kernel-Level Threads) : In this model, each user-level thread is mapped to a separate kernel-level thread by the operating system. Kernel-level threads are scheduled and managed by the OS scheduler. This model provides more concurrency as each user-level thread can run in parallel on separate processor cores, allowing true parallel execution. However, creating

and managing kernel-level threads involves more overhead due to frequent context switches and potential synchronization with kernel data structures.



3. Many-to-Many (Hybrid Model) : The many-to-many model is a hybrid approach that combines aspects of the previous two models. In this model, multiple user-level threads are mapped to an equal or smaller number of kernel-level threads. The OS scheduler can schedule kernel-level threads across multiple processor cores, providing both concurrency and control over scheduling. This model provides a balance between concurrency and flexibility, but it involves additional complexity in managing both user-level and kernel-level threads.



Process Commands

ps command -:

The `ps` command is a commonly used command in operating systems, especially UNIX-like systems, to provide information about the currently running processes. It allows users to view various attributes and details of processes running on the system. The exact options and output of the `ps` command may vary slightly depending on the specific operating system and its version. Here are some commonly used options -:

- `ps -u username`: This option allows you to filter and display processes owned by a specific user. Replace "username" with the actual username.
- `ps -p PID`: Use the `-p` option followed by a process ID (PID) to retrieve information about a specific process.

wait command -:

The "wait" command in an operating system is used to make a process wait or pause its execution until a specific event or condition occurs. It allows the process to temporarily suspend its execution and relinquish the CPU to other processes.

sleep command -:

The "sleep" command is a utility found in various operating systems and programming languages that allows for a specified delay or pause in the execution of a program or script. It is commonly used to introduce a time delay between different actions or to wait for a specific period before proceeding with the next set of instructions. The sleep command is typically used in command-line interfaces and shell scripts.

kill command -:

In operating systems, the `kill` command is a system utility used to terminate or send signals to processes. Its primary purpose is to allow users or system administrators to manage and control running processes. The `kill` command is available in various Unix-like operating systems, including Linux.

The basic syntax of the `kill` command is as follows

kill [options] <PID>

Assignment 3

1. Define Process.
2. List out any two difference between process and program.
3. Explain PCB.
4. With the help of a neat labelled diagram explain process states.
5. Explain types of schedulers with the help of a diagram.
6. What do you mean by scheduling ?
7. What is a thread ? Explain types of thread.
8. Explain multi-threading model.
9. Explain thread life cycle diagram with the help of a diagram.
10. Explain context switching.
11. Explain Inter-process communication. List out all the techniques used in IPC and explain any one of them.
12. Write a short note on : ps command , sleep , wait , kill

Chapter 3 Checklist

Sr. No.	Topics	Status (clear / doubt)
1.	Basic concept of process	
2.	Process states	
3.	Process control block	
4.	Schedulers	
5.	Threads	
6.	Multi-threading Models	
7.	Process Commands	

Doubts Column :

4. CPU Scheduling & Algorithms

Process

In the context of an operating system, a process can be defined as a program in execution. It is an instance of a computer program that is being executed or run by the operating system on a computer's central processing unit (CPU). A process represents a unit of work or a task that can be performed by the computer system.

Here are key characteristics and components of a process in an operating system:

1. Program Code: The executable instructions of the program to be executed are stored in the process.

2. Data: This includes variables, constants, and other data required by the program during execution.

3. Execution Context: It includes the current values of the CPU registers, program counter, and other relevant information that allows the program to resume execution from where it was interrupted.

4. Resources: Processes may require various resources such as memory, I/O devices, and file handles. The operating system manages and allocates these resources to processes as needed.

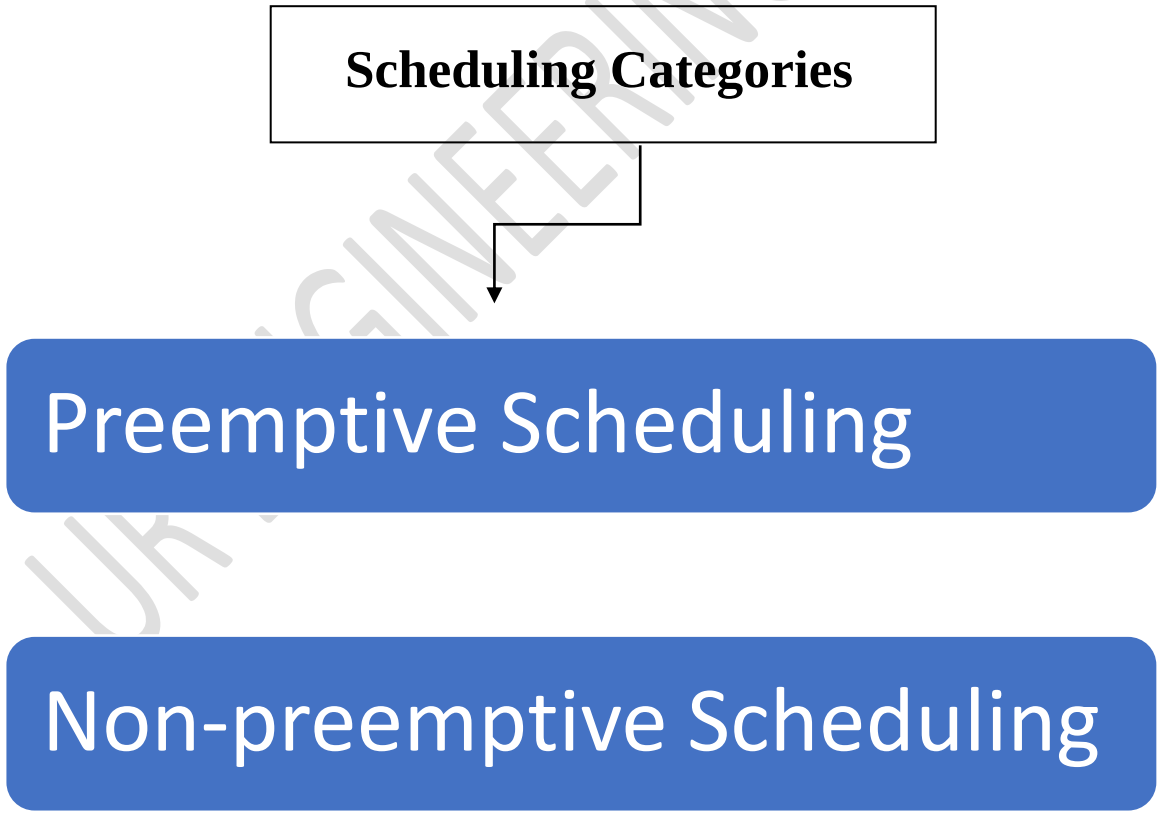
5. State: A process can be in various states during its execution lifecycle, including new, ready, running, blocked, and terminated. The operating system

transitions processes between these states based on their execution progress and resource availability.

6. Scheduling Information: Information used by the operating system to determine the order and priority in which processes are executed on the CPU.

7. Process Control Block (PCB): The operating system maintains a data structure known as the PCB for each process. It contains all the information needed to manage and control the process, including process ID, process state, program counter, CPU registers, and more.

8. Parent-Child Relationship: A process may create new processes, resulting in a parent-child relationship. The parent process can communicate with and control its child processes.



Preemptive and non-preemptive scheduling are two types of process scheduling mechanisms used by operating systems to manage the execution of multiple processes in a multitasking environment.

These mechanisms determine how the operating system decides which process to run and for how long.

1. Preemptive Scheduling:

In preemptive scheduling, the operating system can interrupt a running process and allocate the CPU to another process. The scheduler decides when to preempt a running process based on certain criteria, such as priority or time slice. If a higher-priority process becomes available or the current process exceeds its allocated time slice, the operating system will interrupt the running process and schedule a new one.

Advantages:

- Enhances responsiveness and reduces latency for high-priority tasks.
- Allows for better utilization of system resources by swiftly responding to new tasks or higher-priority processes.

Disadvantages:

- Requires careful handling to prevent race conditions and ensure data consistency.

- Context switching (switching between processes) involves overhead, potentially affecting system performance.

2. Non-preemptive (or Cooperative) Scheduling:

In non-preemptive scheduling, a process holds the CPU until it voluntarily relinquishes control or completes its execution. The operating system does not interrupt the running process to allocate the CPU to another process. The process itself decides when to release the CPU, typically at certain predefined points or after completing a specific task.

Advantages:

- Simpler to implement as it doesn't involve forced context switches.
- Easier to manage and reason about in terms of program execution flow.

Disadvantages:

- May lead to inefficiencies if a process doesn't release the CPU in a timely manner, causing delays for other processes.
- A misbehaving or long-running process can potentially block other processes from executing.

Scheduling Algorithm

Scheduling algorithms in operating systems determine the order in which processes are executed on the CPU. The goal is to optimize resource utilization, ensure fairness, minimize response time, and prioritize certain processes based on various criteria.

Here are some common scheduling algorithms:

1. First-Come-First-Served (FCFS):

- Processes are scheduled in the order they arrive in the ready queue.
- Simple and easy to implement, but may lead to longer waiting times for shorter processes (convoy effect).

2. Shortest Job First (SJF):

- Processes are scheduled based on their burst time (the time taken to complete their execution).
- Minimizes average waiting time and is optimal for minimizing waiting time for the shortest processes.
- However, it requires knowledge of the burst times in advance, which may not always be available.

3. Shortest Remaining Time First (SRTF):

- A preemptive version of SJF where a new process can preempt a running process if its remaining burst time is shorter.
- Minimizes average waiting time and improves response time for short processes.
- May lead to high context switch overhead due to frequent preemptions.

4. Round Robin (RR):

- Each process is allocated a fixed time slice (quantum) in a cyclic order.
- Provides fair execution to all processes and prevents any single process from monopolizing the CPU.
- Response time is usually higher compared to preemptive algorithms.

5. Priority Scheduling:

- Each process is assigned a priority, and the highest priority process is scheduled first.
- Can be either preemptive or non-preemptive.
- May suffer from starvation (low-priority processes never getting CPU time) if not implemented carefully.

6. Priority Aging:

- A variation of priority scheduling where the priority of a process increases as it waits in the ready queue.
- Prevents starvation by gradually increasing the priority of waiting processes.

7. Multi-Level Queue (MLQ):

- Divides the ready queue into multiple separate queues with different priorities.
- Each queue can have its own scheduling algorithm.
- Processes move between queues based on their characteristics or system-defined criteria.

8. Multi-Level Feedback Queue (MLFQ):

- Similar to MLQ, but allows processes to move between different priority queues based on their behaviour

- Higher-priority queues for short CPU bursts and lower-priority queues for longer CPU bursts.
- Prevents aging and allows for dynamic changes in process priorities.

9. Lottery Scheduling:

- Assins processes "lottery tickets" based on some criteria (e.g., priority, CPU usage).
- The scheduler then randomly selects a ticket, and the process associated with that ticket gets to run.

10. Guaranteed Scheduling:

- Provides guaranteed minimum CPU time to each process in a round-robin manner.
- Ensures that no process is starved of CPU time.

FCFS (First Come First Serve) -:

First-Come-First-Served (FCFS) is a simple and straightforward scheduling algorithm where processes are executed based on the order of their arrival in the ready queue. The process that arrives first is scheduled first, and subsequent processes are executed in the order they enter the ready queue.

Advantages :

1. Simplicity : FCFS is straightforward and easy to understand and implement. It doesn't require complex data structures or calculations.

2. No Starvation : Every process gets a chance to execute, ensuring that no process is starved of CPU time indefinitely. It follows a fair order based on arrival time.

3. Optimal for CPU-bound Processes : FCFS is optimal for processes that have long CPU bursts or are CPU-bound, as it allows each process to complete its execution without preemptions.

Disadvantages:

1. Convoy Effect : FCFS can suffer from the convoy effect, where shorter processes are stuck behind a long-running process. This increases the average waiting time, especially for shorter processes.

2. High Average Waiting Time : The average waiting time can be relatively high, especially if shorter processes arrive after longer processes. This can lead to suboptimal performance in terms of response time.

3. Inefficient for I/O-bound Processes : FCFS is not efficient for I/O-bound processes. If a process spends a significant amount of time waiting for I/O operations, other ready processes may get delayed unnecessarily.

4. Not Suitable for Real-Time Systems : FCFS is not suitable for real-time systems where strict timing requirements need to be met. It doesn't prioritize processes based on their urgency or importance.

Example -:

Shortest Job First (SJF)

The Shortest Job First (SJF) algorithm is a non-preemptive or preemptive scheduling policy used in operating systems for process scheduling. It schedules processes based on their burst time, which is the time required for a process to complete its execution.

Here's an overview of how SJF operates:

1. Non-Preemptive SJF:

- The operating system schedules the process with the shortest burst time first.
- When a new process arrives, its burst time is compared with the remaining burst times of other processes in the ready queue. The process with the shortest burst time is selected to run next.

2. Preemptive SJF (also known as Shortest Remaining Time First - SRTF):

- If a new process arrives with a shorter burst time than the remaining time of the currently running process, the operating system preempts the running process and schedules the new process.
- The algorithm compares the remaining burst times of all processes and selects the one with the shortest remaining burst time to run next.

Advantages of SJF:

1. Optimal Average Waiting Time : SJF provides the minimum average waiting time among all scheduling algorithms. It gives priority to shorter jobs, which minimizes the overall waiting time.

2. Efficient Utilization of CPU : When the process with the shortest burst time is scheduled first, it optimizes CPU utilization by executing processes quickly.

3. High Throughput for Short Processes : Short processes complete execution swiftly, allowing the system to handle a higher number of processes in a given time frame.

Disadvantages of SJF:

1. Prediction of Burst Time : In non-preemptive SJF, accurate prediction of burst times for each process is challenging. If burst times are estimated inaccurately, it can lead to suboptimal scheduling.

2. Starvation for Long Processes : Longer processes may face starvation if many short processes keep arriving. The long processes will have to wait until all shorter processes complete.

3. Unrealistic Assumption : SJF assumes that the burst time of each process is known in advance, which is not always the case in real-world scenarios.

Example -:

Round Robin

Round Robin CPU Scheduling is the most important CPU Scheduling Algorithm which is ever used in the history of CPU Scheduling Algorithms. Round Robin CPU Scheduling uses Time Quantum (TQ). The Time Quantum is something which is removed from the Burst Time and lets the chunk of process to be completed.

Time Sharing is the main emphasis of the algorithm. Each step of this algorithm is carried out cyclically. The system defines a specific time slice, known as a time quantum.

Round Robin scheduling is a widely used scheduling algorithm in operating systems and computer systems. It's a pre-emptive scheduling algorithm that treats each process or task in a circular queue, allowing each process to run for a fixed time slice or quantum before moving on to the next one in the queue. This ensures fair allocation of CPU time to all processes in the system, preventing any single process from monopolizing the CPU.

Here's how Round Robin scheduling works:

1. Initialization: Create a queue to hold the processes that are ready to execute. Each process is assigned a fixed time quantum (time slice) that determines how long it can run on the CPU before being moved to the back of the queue.

2. Arrival of Processes: As processes arrive or are created, they are added to the end of the queue.

3. Scheduling: The CPU scheduler selects the process at the front of the queue to run on the CPU for the time quantum specified. If a process completes its execution within the time quantum, it is removed from the queue. If it doesn't finish, it's moved to the back of the queue, making room for the next process.

4. Execution: The selected process is allowed to execute on the CPU until its time quantum expires or it voluntarily releases the CPU, such as when it performs an I/O operation or finishes its execution.

Example -:

Priority Scheduling

Priority scheduling is a type of scheduling algorithm used in computer operating systems and multitasking systems to determine the order in which tasks or processes are executed based on their priority levels. Each task or process is assigned a priority, which indicates its relative importance or urgency.

In priority scheduling, higher-priority tasks are given preference and are scheduled to run before lower-priority tasks. If multiple tasks have the same priority, they are typically scheduled in a first-come-first-served (FCFS) or a round-robin manner.

Priority scheduling is widely used in real-time systems, where tasks have strict timing requirements and need to be completed within specific deadlines. It helps in efficiently managing resources and ensuring that critical tasks are given higher precedence, ultimately enhancing system performance and responsiveness.

Priority scheduling is a popular scheduling algorithm used in operating systems and computer systems to determine the order in which processes are executed based on their priority levels. Here are some advantages and disadvantages of priority scheduling:

Advantages:

1. Responsive to Priority: Priority scheduling ensures that high-priority tasks or processes are executed first, allowing critical tasks to be completed promptly. This can be crucial in real-time systems and environments where certain tasks must be given immediate attention.

2. Efficient Resource Utilization: It can lead to efficient utilization of resources as high-priority processes are scheduled and completed quickly. This can enhance the overall system performance and throughput.

3. Customizable Allocation: Priorities can be assigned based on system requirements, allowing for customization and adaptation to specific needs or applications. Different processes or tasks can be categorized into priority levels based on their importance.

4. Support for Multitasking: Priority scheduling facilitates multitasking by ensuring that time-critical or important tasks get sufficient CPU time, even in a system with numerous competing processes.

5. Flexibility in Task Execution: The scheduler can adapt to changes in priorities dynamically. If a process's priority changes during its execution (e.g., due to a change in its importance), the scheduler can adjust the order of execution accordingly.

Disadvantages:

1. Potential Starvation: Lower-priority processes may face starvation, where they receive very little or no CPU time if higher-priority processes constantly arrive. This can lead to a degradation in system performance and fairness.

2. Priority Inversion: Priority inversion can occur when a lower-priority task holds a resource that a higher-priority task needs. This can lead to a scenario where a higher-priority task is blocked by a lower-priority task, causing delays and affecting system responsiveness.

3. Complexity and Overhead: Implementing and managing a priority-based scheduling system can be complex. Assigning and managing priorities for various processes require additional overhead and administrative effort, which can impact the efficiency and simplicity of the scheduler.

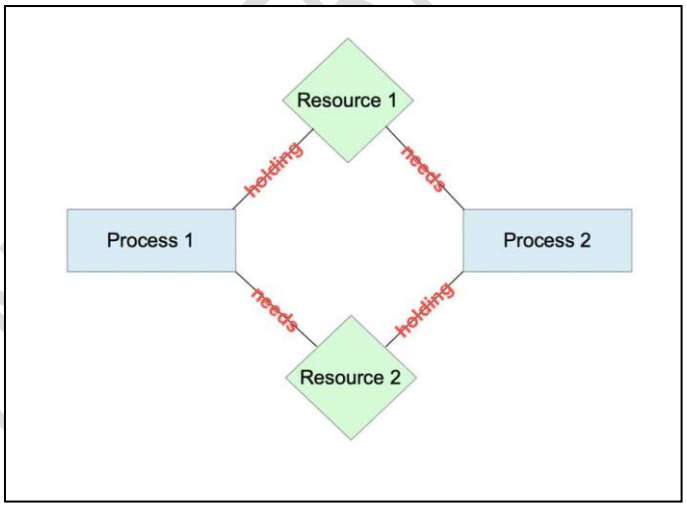
4. Possibility of Process Starvation: If a process continually receives a low priority and there are always higher-priority processes in the system, that process may never get the chance to execute, causing process starvation and potential performance issues.

5. Risk of Priority Inversion Deadlocks: If not handled carefully, priority inversion can lead to deadlocks, where processes end up waiting indefinitely for resources, causing system lockups and an inability to proceed with further processing.

Example -:

Deadlock

A deadlock in operating system terms refers to a state where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. This creates a circular waiting scenario where none of the processes can proceed, effectively halting the entire system.



Here are the key components involved in a deadlock:

1. Processes: Multiple processes are competing for resources like CPU time, memory, or input/output devices.

2. Resources: Resources can be either reusable (e.g., memory, CPU) or consumable (e.g., files, database records).

3. Requests and Holds: A process can request a resource or hold a resource. If a process requests a resource that is currently held by another process, it may be forced to wait until the resource becomes available.

4. Circular Wait: A deadlock occurs when a set of processes are each waiting for a resource acquired by one of the other processes, forming a circular chain of dependencies.

Here's an example scenario to illustrate a deadlock:

- Process 1 holds Resource A and requests Resource B.

- Process 2 holds Resource B and requests Resource C.

- Process 3 holds Resource C and requests Resource A.

In this scenario, none of the processes can proceed because they are all waiting for a resource held by another process. This forms a circular wait, leading to a deadlock.

Preventing and resolving deadlocks is crucial in operating system design. Strategies to handle deadlocks include:

1. Deadlock Prevention: Ensure that at least one of the necessary conditions for deadlock (mutual exclusion, hold and wait, no preemption, and circular wait) is not satisfied.

2. Deadlock Avoidance: Use algorithms that carefully allocate resources to processes to avoid the possibility of deadlock. Examples include the Banker's algorithm and optimistic concurrency control.

3. Deadlock Detection and Recovery: Allow the system to enter a deadlock state, then detect and recover from it by either killing processes or rolling back to a previous state.

Assignment #4

1. Define preemptive and non-preemptive scheduling with the help of an example..

2. What is the need of scheduling algorithms ?

3. List types of scheduling algorithms and explain any one of them.

4. Explain fcfs with the help of an example.

5. Explain sjf with the help of an example. (preemptive and non-preemptive)

6. Explain round robin with the help of an example.

7. Explain priority scheduling with the help of an example.

8. Explain deadlock

5. Memory Management

Memory Management

Memory management in an operating system involves various mechanisms and techniques to efficiently manage a computer system's memory resources. The memory management module is a critical component within the operating system responsible for controlling and organizing the computer's memory, ensuring that programs and processes can access and use memory in a structured and efficient manner. Here's a breakdown of its main functions and components:

1. Memory Allocation and Deallocation:

- Allocating memory to processes when requested and deallocating it when no longer needed.

2. Memory Organization:

- Managing the available memory space, dividing it into partitions or segments to accommodate multiple processes.

3. Address Translation:

- Mapping logical addresses used by processes to physical addresses in the actual hardware memory.

4. Memory Protection:

- Implementing measures to prevent unauthorized access to memory regions, ensuring the security and integrity of data.

5. Memory Sharing:

- Allowing multiple processes to share certain memory regions, enhancing efficiency and reducing resource usage.

6. Memory Paging and Segmentation:

- Implementing paging or segmentation techniques to optimize memory usage and improve memory access times.

7. Virtual Memory:

- Providing an illusion of a much larger main memory by using disk storage as an extension of RAM, allowing efficient utilization of available memory.

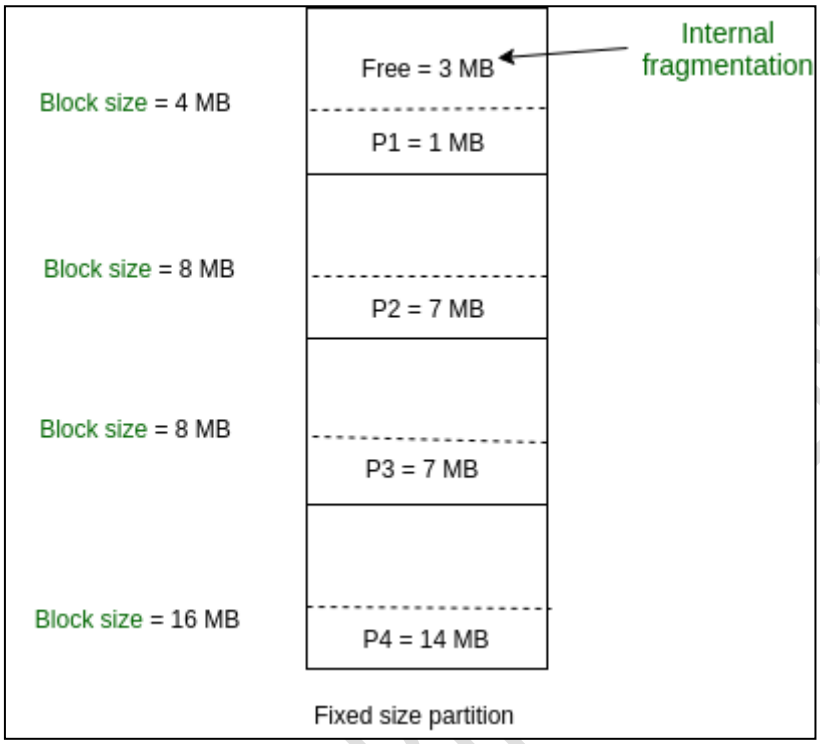
Memory Partitioning

Memory partitioning is a technique used in operating systems to manage and allocate memory to multiple processes running concurrently. It involves dividing the physical memory into fixed or variable-sized partitions to accommodate different processes and their memory requirements. There are two main types of memory partitioning: fixed partitioning and dynamic partitioning.

1. Fixed Partitioning:

In fixed partitioning, the physical memory is divided into fixed-sized partitions during system initialization. Each partition can accommodate one process. Processes are loaded into these partitions based on their size, and any remaining

unused space in a partition is wasted. Fixed partitioning is simple but may lead to inefficient memory utilization.



Advantages:

- Simplicity and ease of implementation.
- Quick and straightforward allocation of memory.

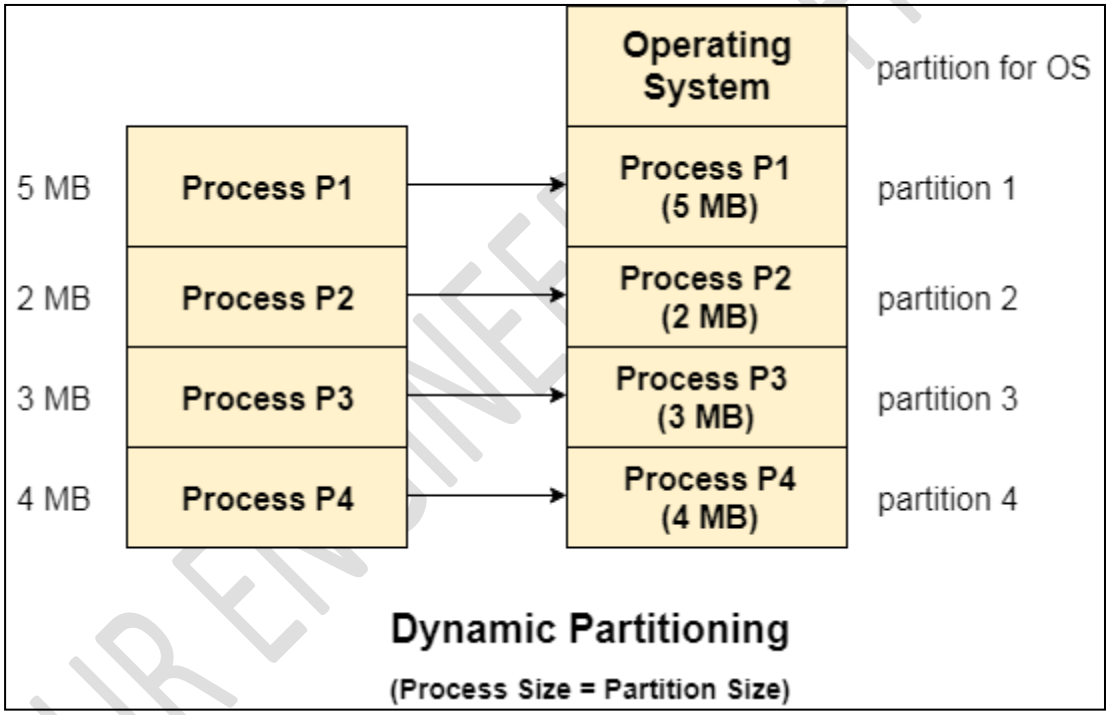
Disadvantages:

- Inefficient memory utilization due to internal fragmentation.
- Difficulty accommodating processes of varying sizes.

2. Dynamic Partitioning:

Dynamic partitioning involves allocating memory dynamically based on the size of the processes. When a new process arrives, the operating system finds a suitable partition that can accommodate the process and allocates memory accordingly. The partition size is adjusted dynamically based on the size of the incoming processes.

In modern operating systems, dynamic partitioning is more commonly used due to its flexibility and ability to adapt to varying process memory requirements. Advanced memory management techniques like paging and segmentation are often combined with dynamic partitioning to further optimize memory utilization and address fragmentation issues.



Advantages:

- Efficient memory utilization by reducing internal fragmentation.
- Ability to accommodate processes of varying sizes.

Disadvantages:

- Overhead of managing and updating information about partitions and free spaces.
- Possibility of external fragmentation, where small blocks of free memory become scattered and cannot be used.

To address the issue of external fragmentation in dynamic partitioning, memory compaction and relocation techniques can be employed. Compaction involves moving processes and consolidating fragmented free memory to create contiguous blocks of free space. Relocation involves moving a process during execution to a different memory location to eliminate external fragmentation.

Fragmentation

Fragmentation in the context of operating systems refers to the phenomenon where the total memory space is broken up into smaller, non-contiguous blocks, making it challenging to efficiently allocate these fragmented blocks for new processes or data storage. Fragmentation can occur in both main memory (RAM) and secondary storage (disk).

There are two main types of fragmentation:

1. Internal Fragmentation:

Internal fragmentation occurs when memory is allocated in fixed-size blocks or partitions, and the allocated memory may be larger than what the process actually needs. The unused portion within a memory block is wasted, resulting in inefficient memory utilization.

- **Fixed Partitioning:**

In fixed partitioning, where memory is divided into fixed-size partitions, internal fragmentation occurs when a process is allocated a partition larger than its actual memory requirements.

- **Variable Partitioning:**

Even in dynamic partitioning (variable-sized partitions), internal fragmentation occurs when a process is allocated a partition that is larger than its size, resulting in unused memory within the partition.

- Example:

Suppose a process needs 30 units of memory and is allocated a partition of 40 units. The remaining 10 units are wasted, representing internal fragmentation.

2. External Fragmentation:

External fragmentation occurs when there is enough total free memory to satisfy a memory request, but this free memory is scattered across the system in small, non-contiguous blocks. As a result, a request for a specific amount of memory cannot be fulfilled, even though the aggregate free memory is sufficient.

- Example:

Imagine you have 100 units of free memory, divided into blocks of 30, 10, 25, and 35 units. If a process requests 50 units of memory, it cannot be accommodated even though there is enough free memory available.

- Methods to Handle External Fragmentation:

- **Compaction:** Compaction involves relocating processes and rearranging memory contents to create larger contiguous blocks of free memory, reducing external fragmentation
- **Memory Compaction:** Periodically shifting processes to consolidate free memory blocks and reduce fragmentation.
- **Memory Allocation Algorithms:** Using efficient memory allocation algorithms (e.g., best fit, worst fit, first fit) to minimize fragmentation.

Both internal and external fragmentation can impact system performance and memory utilization efficiency. Effective memory management strategies, like compaction and appropriate memory allocation algorithms, are crucial to mitigate the effects of fragmentation and enhance overall system performance.

Paging

Paging is a memory management scheme used by modern operating systems to efficiently manage and utilize the computer's physical memory (RAM). It's a technique that allows the operating system to store and retrieve data from the main memory in fixed-size blocks called "pages."

If a requested page is not present in physical memory (a page fault), the operating system fetches the required page from the disk (where it's stored when not in RAM) and swaps it into an available frame. This process allows efficient utilization of physical memory by loading only the necessary pages into memory while keeping the rest on disk.

The key idea behind paging is to divide the virtual address space of a process into fixed-size pages and map these pages to frames in the physical memory. When a process accesses a memory address, the operating system uses the page table to translate the virtual address into a physical address.

Here's a breakdown of how paging works:

1. Page: A fixed-size block of memory, typically 4 KB in size, used by the operating system to manage memory. The size of a page is defined by the hardware and is a power of 2.

2. Frame: A fixed-size block of physical memory corresponding to the size of a page. Frames are also a power of 2 and match the size of pages.

3. Page Table: A data structure maintained by the operating system that maps virtual addresses (used by the processes) to physical addresses (locations in RAM). Each entry in the page table corresponds to a page and stores the frame number where the page is located in physical memory.

4. Virtual Address Space: The range of addresses that a process can use, as seen from the perspective of the process. This address space is divided into fixed-size pages.

5. Physical Address Space: The range of actual memory addresses in the physical RAM.

Page Replacement Algorithms

Page replacement algorithms are techniques used by the operating system to manage the limited amount of physical memory (RAM) effectively when there is a need to swap pages in and out of memory due to memory demand. When a page fault occurs and a new page needs to be brought into memory, but there are no free frames, the operating system must decide which page(s) to remove from memory to make space for the new page. This decision is based on a page replacement algorithm. Common page replacement algorithms include:

1. FIFO (First-In-First-Out):

This algorithm removes the page that has been in memory the longest. It's like a queue where the page that came in first is the first to be replaced. However, FIFO does not consider how frequently a page is accessed or how recently it was used.

2. LRU (Least Recently Used):

LRU replaces the page that has not been used for the longest time. It's based on the principle that pages that have not been accessed recently are less likely to be used soon. LRU requires maintaining a timestamp or a counter for each page to track its last access time.

3. Optimal Page Replacement:

This is an idealized algorithm that replaces the page that will not be used for the longest time in the future. It's used as a reference to measure the efficiency of other page replacement algorithms, although it's not practically implementable because it requires knowledge of future memory accesses.

First-In-First-Out

The FIFO (First-In-First-Out) page replacement algorithm is one of the simplest and most intuitive memory management schemes used in operating systems. It operates based on the principle of a queue, where the page that has been in memory the longest is the one selected for replacement.

Here's a step-by-step explanation of how the FIFO page replacement algorithm works:

1. Initialization:

The operating system maintains a queue, which initially is empty, to track the order of page arrivals into the memory.

2. Page Request:

When a page fault occurs (a requested page is not in memory), the operating system selects the oldest page in the memory for replacement.

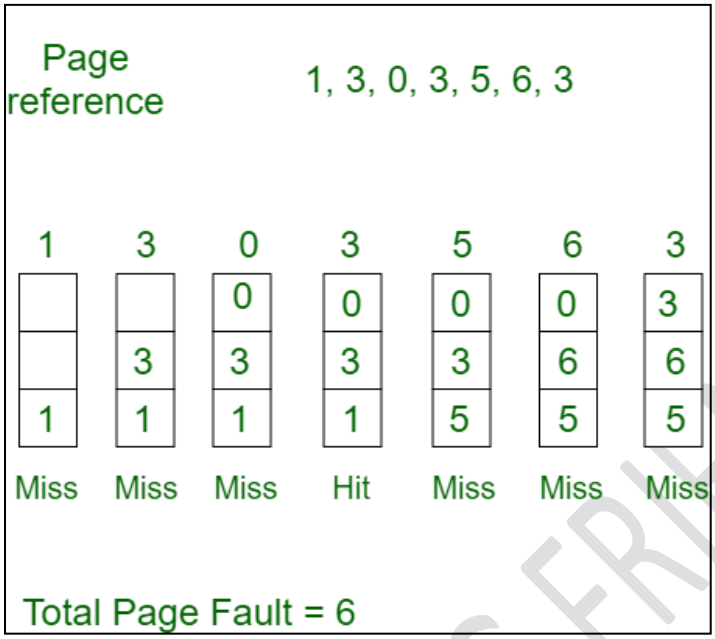
3. Page Replacement:

The page at the front of the queue (the one that has been in memory the longest) is replaced with the new page that caused the page fault.

4. Updating the Queue:

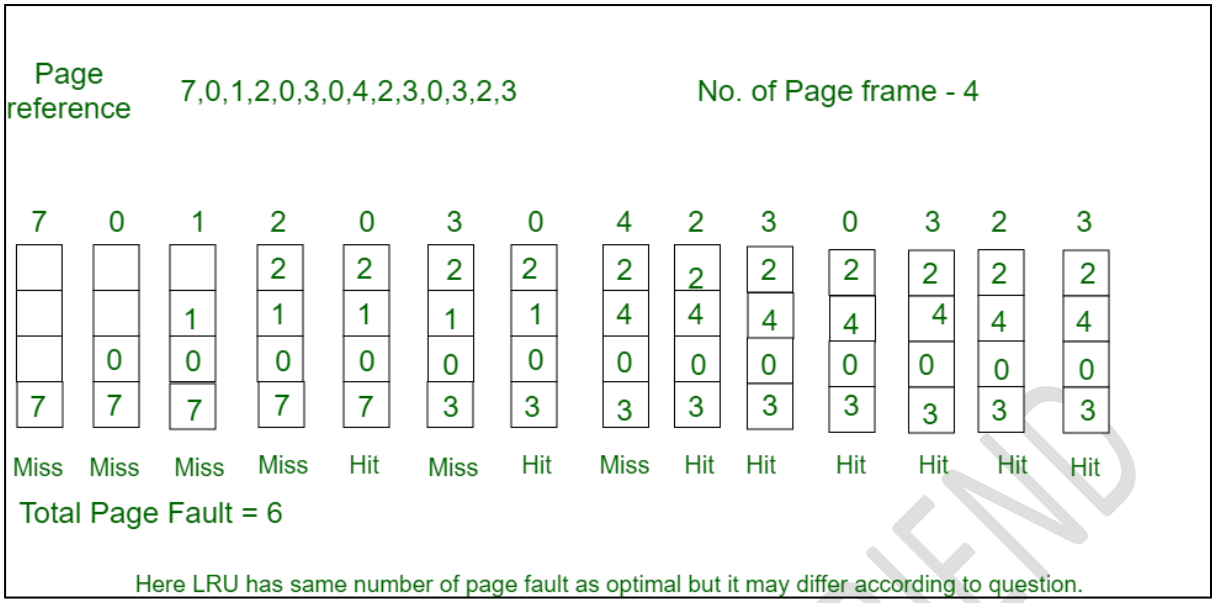
After replacing the old page, the new page is added to the end of the queue, representing the most recently brought into memory.

Example -:



LRU (Least Recently Used

The LRU stands for the **Least Recently Used**. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods.



LRU, which stands for "Least Recently Used," is a page replacement algorithm used in computer operating systems to manage and optimize the use of physical memory (RAM) when executing multiple processes or programs. The primary goal of the LRU algorithm is to minimize page faults by evicting the page that has not been used for the longest period of time.

Here's how the LRU page replacement algorithm works:

1. Maintaining a Page Reference List: LRU keeps track of the order in which pages are accessed by creating a list or queue, which is sometimes called a "page reference list." This list is usually implemented as a data structure that records the recent page accesses.

2. Page Access: Whenever a page is accessed, it is moved to the front of the page reference list to indicate that it's the most recently used page. If a page is already in the list, it is promoted to the front. If a page is not in the list, it is added to the front.

3. Page Replacement Decision: When the operating system needs to bring in a new page (due to a page fault) and there's no free space in RAM, the LRU algorithm selects the page at the end of the list (the one that hasn't been used for the longest time) for replacement.

4. Maintaining List Size: To ensure that the page reference list doesn't grow infinitely, it is usually limited to a fixed size, which means that when a new page is added to the front of the list, the page at the end of the list is removed.

Optimal Page Replacement

The Optimal Page Replacement algorithm, also known as the Belady's Algorithm, is a theoretical memory management algorithm used to evaluate the performance of other page replacement algorithms.

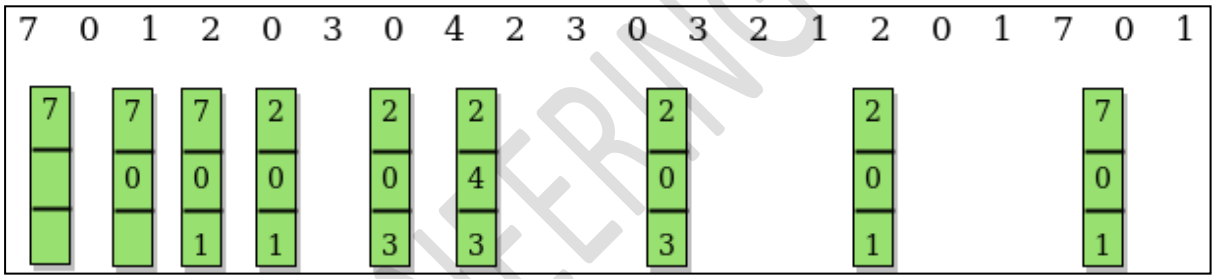
- It is not a practical algorithm for real-world systems due to its reliance on future knowledge, which is impossible to obtain in most situations. Instead, it helps us establish an upper bound on the performance of other page replacement algorithms.
- The goal of the Optimal Page Replacement algorithm is to minimize the number of page faults.
- To achieve this, it assumes perfect knowledge of future memory access patterns, meaning it knows which pages will be accessed and when.
- This idealized knowledge allows the algorithm to make the best possible decisions.

Here's how the Optimal Page Replacement algorithm works:

1. At each memory access request, the algorithm examines the future references to determine which page will be used furthest in the future.

2. It selects the page that will not be used for the longest time (the page that has the farthest future reference).

3. If the memory is full, and a page replacement is required, the algorithm replaces the page that will not be needed for the longest time, according to its future knowledge.



6. File Management

Concept of File

In an operating system, a "file" is a fundamental unit of data storage that contains information, data, or instructions. Files are organized into directories or folders, and they serve as a means to store, retrieve, and manage data. Each file is characterized by several attributes that provide information about the file and its properties. These attributes can include:

1. File Name: This is the human-readable name of the file, which is used to identify and reference it.

2. File Extension: The file extension is typically a part of the file name and indicates the file type or format. For example, ".txt" is often used for plain text files, ".jpg" for image files, and ".exe" for executable programs.

3. File Size: This attribute specifies the size of the file in bytes or another appropriate unit of measurement, indicating how much storage space the file occupies.

4. File Location: The file's path or directory location specifies where the file is stored within the file system's hierarchy. It includes information about the folder(s) in which the file is contained.

5. Date Created: This attribute indicates the date and time when the file was initially created.

6. Date Modified: This attribute indicates the date and time when the file was last modified or updated.

7. Date Accessed: Some operating systems track the date and time when the file was last accessed, although this attribute is often disabled by default due to performance considerations.

8. File Permissions: File permissions define who can access, read, write, or execute the file. These permissions are usually categorized into read, write, and execute permissions for the owner, group, and others.

9. File Type: This attribute provides information about the type or format of the file, which can be used by the operating system and associated applications to determine how to handle the file.

10. File Owner: Every file is associated with an owner, typically the user account that created the file. The owner has special privileges and control over the file's permissions.

File Operations

In an operating system, file operations are a set of actions that can be performed on files to create, read, write, update, delete, and manage them. These operations are essential for managing and manipulating data within the file system. Here are some of the most common file operations:

1. File Creation: Creating a file involves specifying a file name and optionally choosing a location within the file system. The operating system reserves space for the file and assigns initial attributes such as creation date and permissions.

2. File Reading: Reading from a file allows you to retrieve data from an existing file. Reading can be done sequentially or randomly, depending on the file access method. The operating system provides system calls and APIs for reading data from files.

3. File Writing: Writing to a file involves adding or modifying data within an existing file. You can append data to the end of the file or overwrite existing content. File writing is often subject to file permissions, which control who can write to a file.

5. File Updating: Updating a file means modifying specific parts of its content, typically within the file's structure. For instance, updating a database file might involve changing a record's data without affecting the rest of the file.

6. File Deletion: Deleting a file removes it from the file system, freeing up storage space. Care should be taken when deleting files, as they may not always be recoverable from the recycling bin or trash.

File System Structure

In operating systems, file systems can be structured in different ways to organize and store data. Three common structures are byte sequence, tree sequence, and record sequence. Each of these structures has its own characteristics and use cases:

1. Byte Sequence File System:

- In a byte sequence file system, data is organized as a continuous stream of bytes.
- This structure is the simplest and most straightforward, as it treats files as unstructured sequences of bytes with no inherent hierarchy.
- There are no directories or folders in a byte sequence file system, only files.
- Files are identified by their position within the sequence of bytes, usually starting from the beginning of the storage medium.
- This structure is often used in embedded systems or where simplicity and minimal overhead are required. However, it lacks the ability to efficiently organize and categorize data.

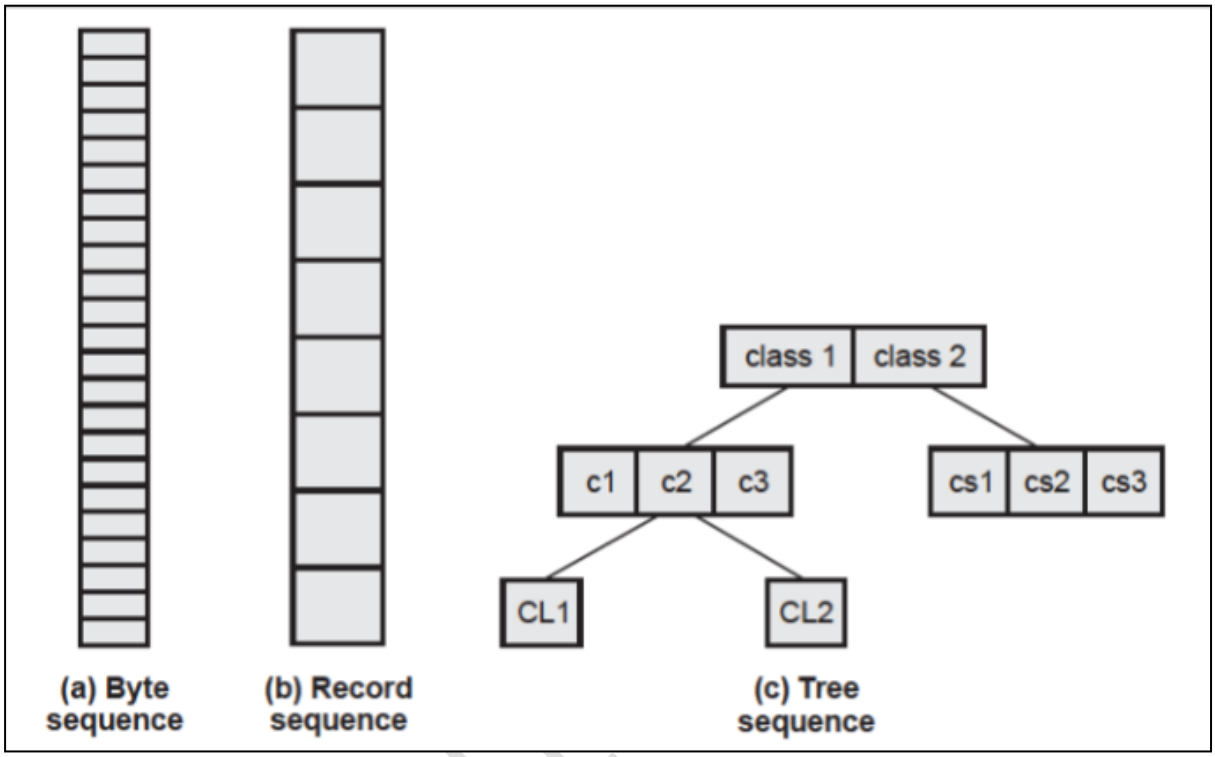
2. Tree Sequence File System:

- In a tree sequence file system, data is organized in a hierarchical tree-like structure.
- Files and directories (folders) are represented as nodes in the tree, with the root node at the top and subdirectories branching off from parent directories.
- This structure allows for a logical organization of files and makes it easy to navigate and locate specific data.
- Each file or directory has a unique path that specifies its location within the tree (e.g., "C:\Users\John\Documents\file.txt" in Windows or "/home/jane/documents/file.txt" in Unix-like systems).
- Tree sequence file systems are commonly used in desktop and server operating systems, providing efficient data organization and retrieval.

3. Record Sequence File System:

- A record sequence file system stores data as records, where each record consists of a fixed-size block or entry.
- This structure is well-suited for applications where data is organized into records, such as databases.
- Records are typically identified by a record number or key and can be read, written, or updated individually.
- Record sequence file systems provide efficient access to specific data within a file without the need to read or write the entire file.

- This structure is commonly used in database management systems (DBMS) and other data-centric applications.



File Access Methods

File access methods in operating systems refer to the techniques or approaches used to read and write data within files. These methods determine how data is accessed and retrieved from storage media, such as hard drives or solid-state drives. There are primarily two common file access methods: sequential access and direct (random) access.

1. Sequential Access:

In sequential access, data is read or written one item at a time in a linear or sequential manner. To access a specific piece of data, you must start at the beginning of the file and read or write sequentially until you reach the desired location.

- Sequential access is similar to reading a book page by page, starting from the first page and moving forward until you reach the desired page.
- This method is suitable for tasks that involve reading or writing data in a specific order, such as scanning a text file line by line or processing a data file sequentially.
- Example: Reading a text file line by line in a sequential manner. You start at the beginning of the file and read each line until you find the one you're looking for.

File Contents:

Line 1: This is the first line.

Line 2: This is the second line.

Line 3: This is the third line.

Line 4: This is the fourth line.

2. Direct (Random) Access:

Direct access, also known as random access, allows you to read or write data at a specific location within the file without the need to traverse the entire file sequentially.

- Each data item within the file is associated with an address or index, allowing you to directly access the desired item.
- This method is suitable for tasks that require quick access to specific data within a file, such as database systems or indexed data structures.
- Example: Accessing a specific record in a database file by its record number. You don't need to read through all the records to find the one you're interested in.

Record 1: User: Alice, Age: 28, Email: alice@example.com

Record 2: User: Bob, Age: 35, Email: bob@example.com

Record 3: User: Carol, Age: 42, Email: carol@example.com

File Allocation

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- **Contiguous Allocation**
- **Linked Allocation**
- **Indexed Allocation**

The main idea behind these methods is to provide:

- Efficient disk space utilization.
- Fast access to the file blocks.

1. Contiguous Allocation

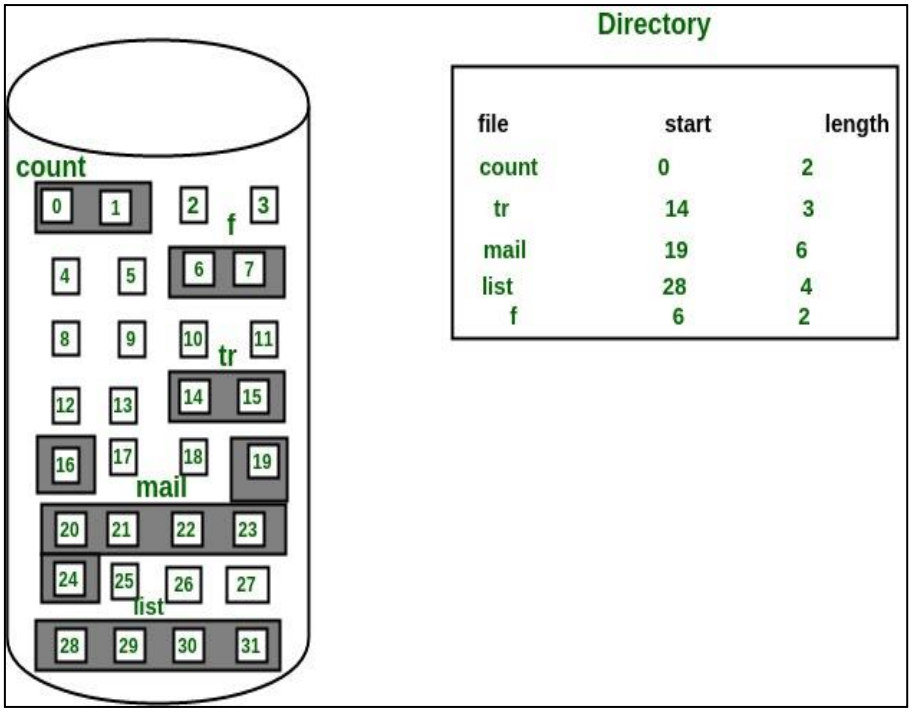
In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.*

- This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

  The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.



Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

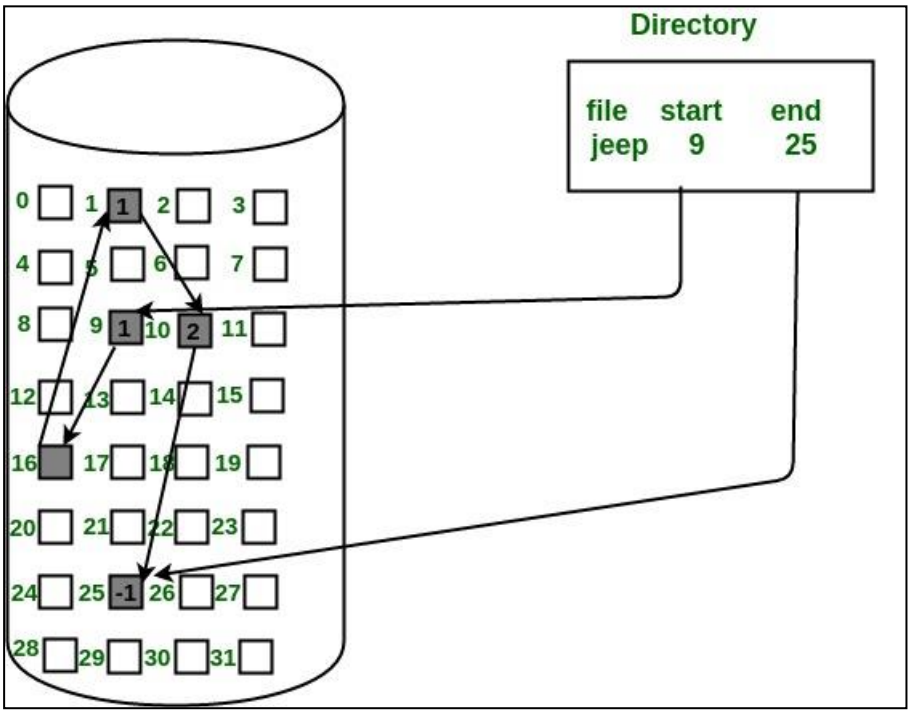
Disadvantages:

- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

- *The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.*



Advantages:

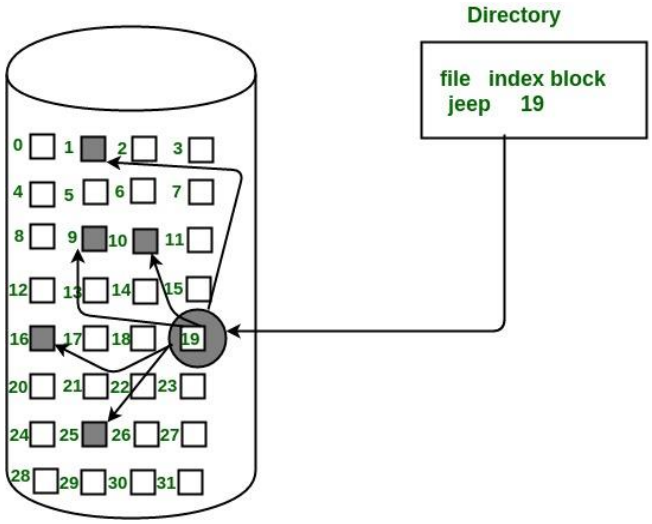
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:



Advantages:

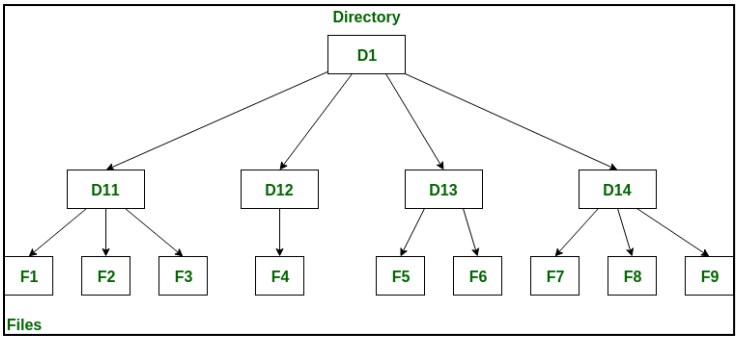
- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Directory Structure

A **directory** is a container that is used to contain folders and files. It organizes files and folders in a hierarchical manner.

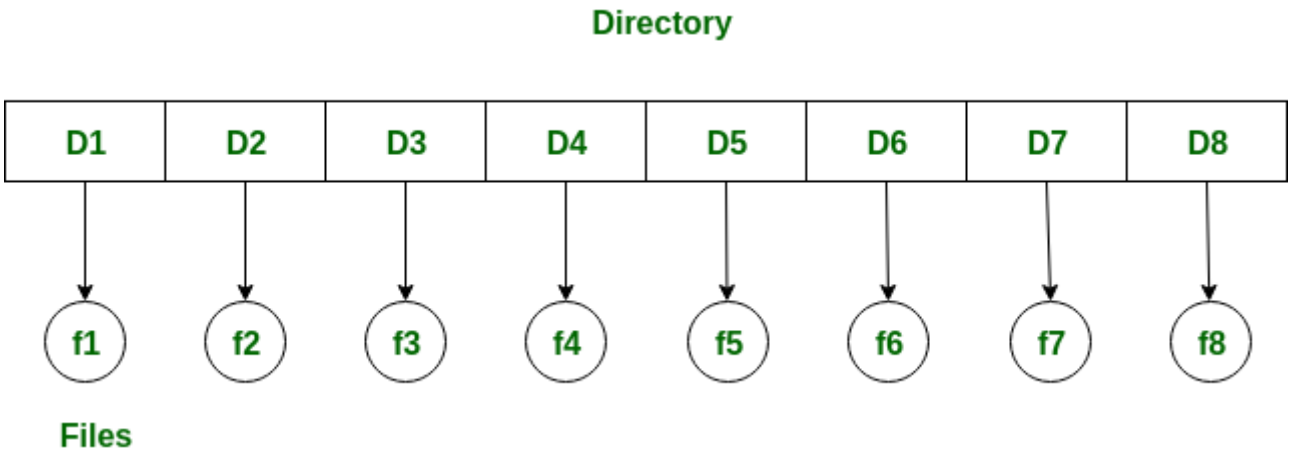


Following are the logical structures of a directory, each providing a solution to the problem faced in previous type of directory structure.

1) Single-level directory:

The single-level directory is the **simplest directory structure**. In it, all files are contained in the same directory which makes it easy to support and understand.

A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. Since all the files are in the same directory, they must have a **unique name**. If two users call their dataset test, then the unique name rule violated.



Advantages:

- Since it is a single directory, so its implementation is very easy.
- If the files are smaller in size, searching will become faster.
- The operations like file creation, searching, deletion, updating are very easy in such a directory structure.
- **Logical Organization**: Directory structures help to logically organize files and directories in a hierarchical structure. This provides an easy way to

navigate and manage files, making it easier for users to access the data they need.

- **Increased Efficiency:** Directory structures can increase the efficiency of the file system by reducing the time required to search for files. This is because directory structures are optimized for fast file access, allowing users to quickly locate the file they need.
- **Improved Security**: Directory structures can provide better security for files by allowing access to be restricted at the directory level. This helps to prevent unauthorized access to sensitive data and ensures that important files are protected.
- **Facilitates Backup and Recovery**: Directory structures make it easier to backup and recover files in the event of a system failure or data loss. By storing related files in the same directory, it is easier to locate and backup all the files that need to be protected.
- **Scalability:** Directory structures are scalable, making it easy to add new directories and files as needed. This helps to accommodate growth in the system and makes it easier to manage large amounts of data.

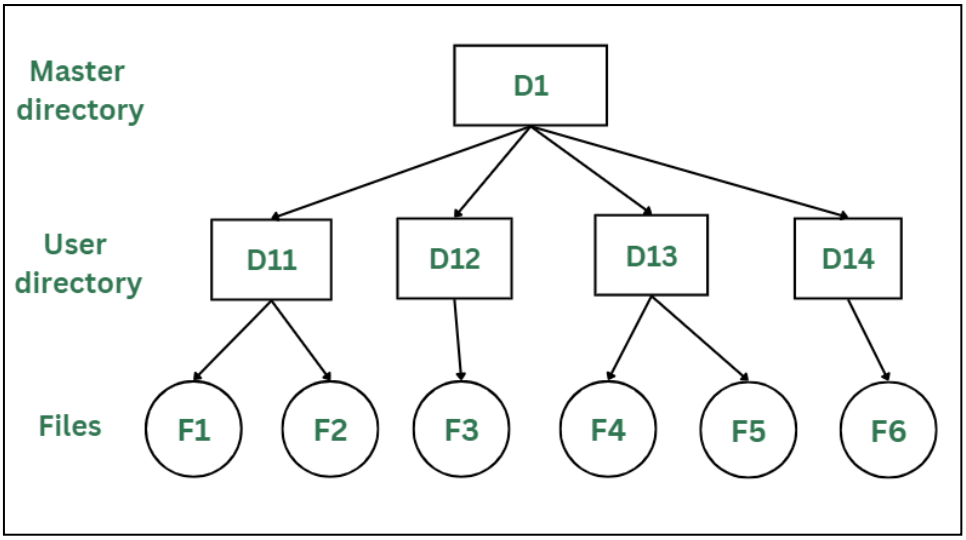
Disadvantages:

- There may chance of name collision because two files can have the same name.
- Searching will become time taking if the directory is large.
- This can not group the same type of files together.

2) Two-level directory:

As we have seen, a single level directory often leads to confusion of files names among different users. The solution to this problem is to create a **separate directory for each user**.

In the two-level directory structure, each user has their own **user files directory (UFD).** The UFDs have similar structures, but each lists only the files of a single user. System's **master file directory (MFD)** is searched whenever a new user id is created.



Advantages:

- The main advantage is there can be more than two files with same name, and would be very helpful if there are multiple users.
- A security would be there which would prevent user to access other user's files.
- Searching of the files becomes very easy in this directory structure.

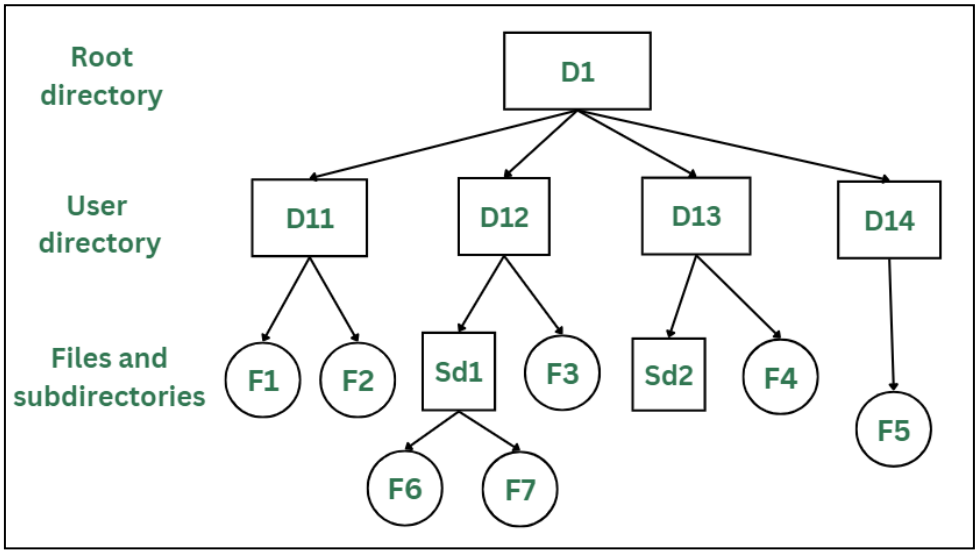
Disadvantages:

- As there is advantage of security, there is also disadvantage that the user cannot share the file with the other users.
- Unlike the advantage users can create their own files, users don't have the ability to create subdirectories.
- Scalability is not possible because one use can't group the same types of files together.

3) Tree Structure/ Hierarchical Structure:

Tree directory structure of operating system is most commonly used in our **personal computers**. User can create files and subdirectories too, which was a disadvantage in the previous directory structures.

- This directory structure resembles a real tree upside down, where the **root directory** is at the peak. This root contains all the directories for each user. The users can create subdirectories and even store files in their directory.
- A user do not have access to the root directory data and cannot modify it. And, even in this directory the user do not have access to other user's directories. The structure of tree directory is given below which shows how there are files and subdirectories in each user's directory.



Advantages:

- This directory structure allows subdirectories inside a directory.
- The searching is easier.
- File sorting of important and unimportant becomes easier.
- This directory is more scalable than the other two directory structures explained.

Disadvantages:

- As the user isn't allowed to access other user's directory, this prevents the file sharing among users.
- As the user has the capability to make subdirectories, if the number of subdirectories increase the searching may become complicated.
- Users cannot modify the root directory data.
- If files do not fit in one, they might have to be fit into other directories.

Raid Levels -:

RAID, or "Redundant Arrays of Independent Disks" is a technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both. The term was coined by David Patterson, Garth A. Gibson, and Randy Katz at the University of California, Berkeley in 1987.

It is a way of storing the same data in different places on multiple hard disks or solid-state drives to protect data in the case of a drive failure. A RAID system consists of two or more drives working in parallel. These can be hard discs, but there is a trend to use SSD technology (Solid State Drives).

RAID combines several independent and relatively small disks into single storage of a large size. The disks included in the array are called **array members**. The disks can combine into the array in different ways, which are known as **RAID levels**.

How RAID Works

RAID works by placing data on multiple disks and allowing input/output operations to overlap in a balanced way, improving performance. Because various disks increase the mean time between failures (MTBF), storing data redundantly also increases fault tolerance.

RAID arrays appear to the operating system as a single logical drive. RAID employs the techniques of disk mirroring or disk striping.

- Disk Mirroring will copy identical data onto more than one drive.
- Disk Striping partitions help spread data over multiple disk drives. Each drive's storage space is divided into units ranging from 512 bytes up to several megabytes. The stripes of all the disks are interleaved and addressed in order.
- Disk mirroring and disk striping can also be combined in a RAID array.

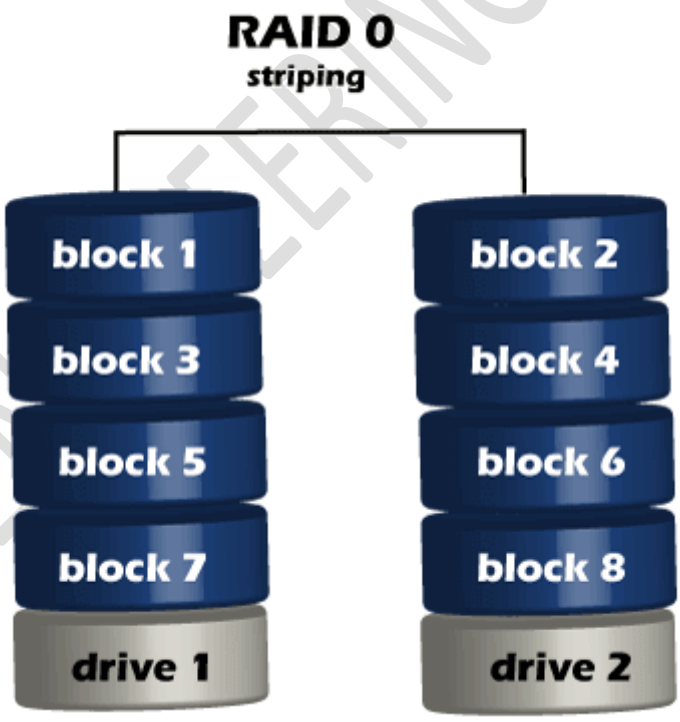
Levels of RAID

Many different ways of distributing data have been standardized into various RAID levels. Each RAID level is offering a trade-off of data protection, system

performance, and storage space. The number of levels has been broken into three categories, standard, nested, and non-standard RAID levels.

1. RAID 0 (striped disks)

RAID 0 is taking any number of disks and merging them into one large volume. It will increase speeds as you're reading and writing from multiple disks at a time. But all data on all disks is lost if any one disk fails. An individual file can then use the speed and capacity of all the drives of the array. The downside to RAID 0, though, is that it is NOT redundant. The loss of any individual disk will cause complete data loss. This RAID type is very much less reliable than having a single disk.

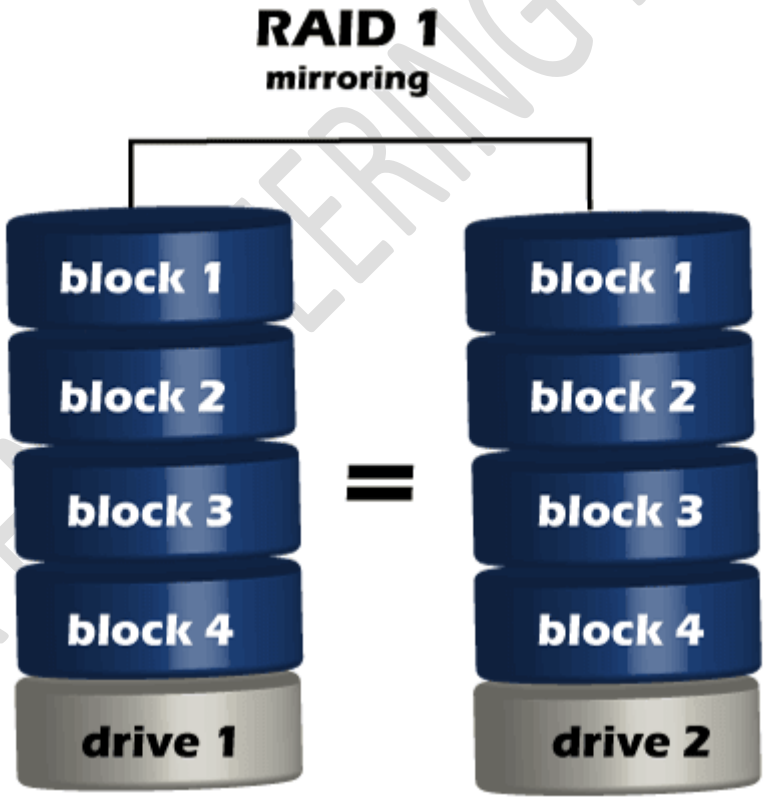


There is rarely a situation where you should use RAID 0 in a server environment. You can use it for cache or other purposes where speed is essential, and reliability or data loss does not matter at all.

2. RAID 1 (mirrored disks)

It duplicates data across two disks in the array, providing full redundancy. Both disks are store exactly the same data, at the same time, and at all times. Data is not lost as long as one disk survives. The total capacity of the array equals the capacity of the smallest disk in the array. At any given instant, the contents of both disks in the array are identical.

RAID 1 is capable of a much more complicated configuration. The point of RAID 1 is primarily for redundancy. If you completely lose a drive, you can still stay up and running off the other drive.

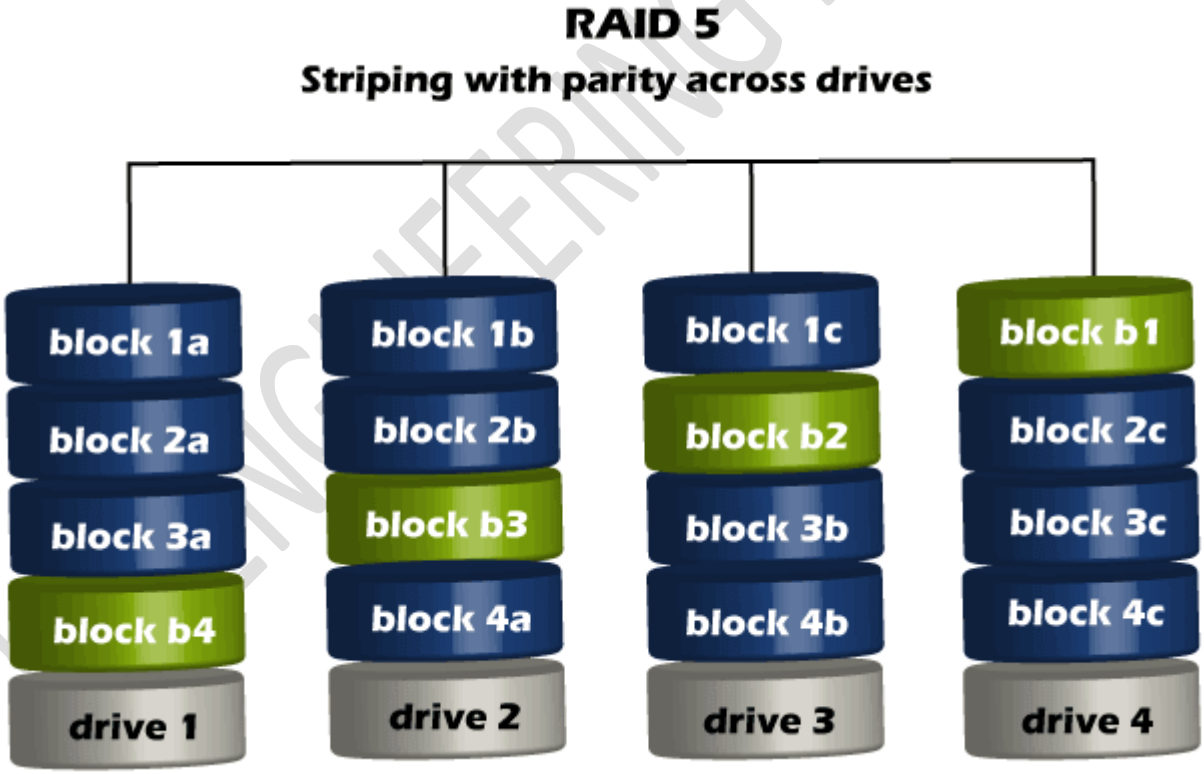


If either drive fails, you can then replace the broken drive with little to no downtime. RAID 1 also gives you the additional benefit of increased read performance, as data can read off any of the drives in the array. The downsides

are that you will have slightly higher write latency. Since the data needs to be written to both drives in the array, you'll only have a single drive's available capacity while needing two drives.

3. RAID 5(striped disks with single parity)

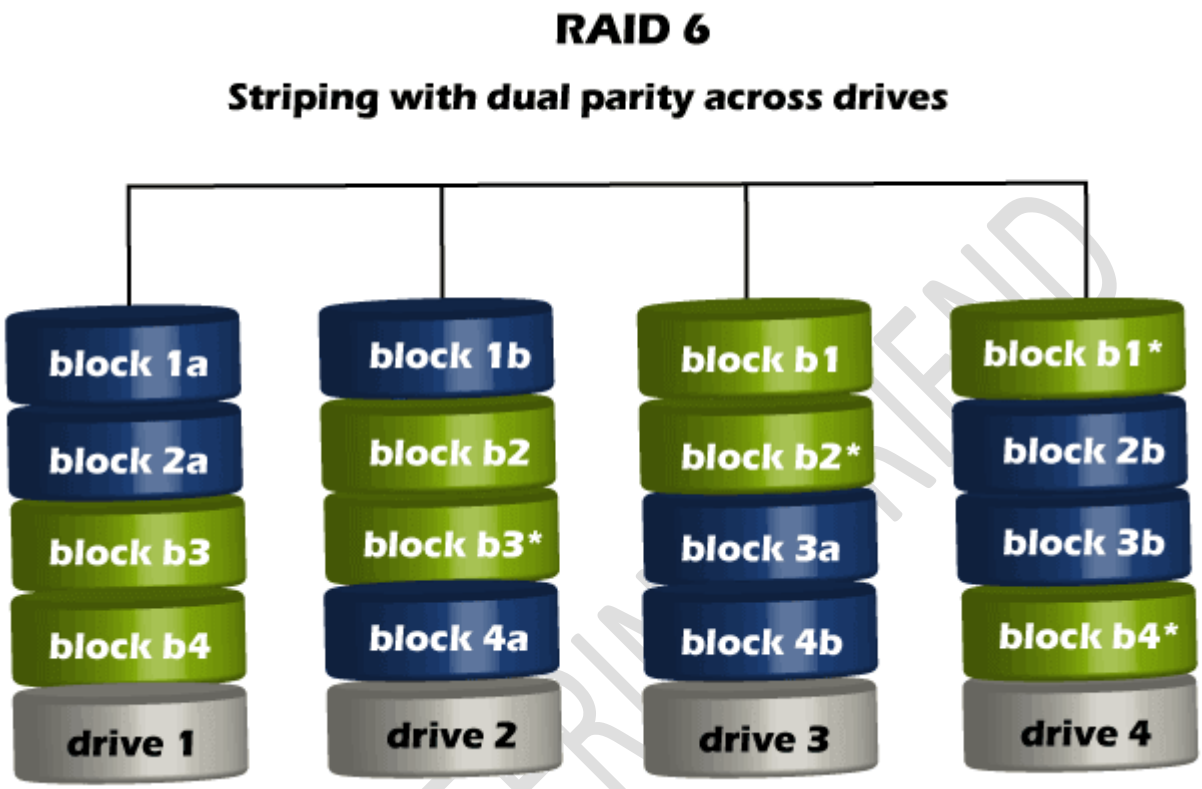
RAID 5 requires the use of at least three drives. It combines these disks to protect data against loss of any one disk; the array's storage capacity is reduced by one disk. It strips data across multiple drives to increase performance. But, it also adds the aspect of redundancy by distributing parity information across the disks.



4. RAID 6 (Striped disks with double parity)

RAID 6 is similar to RAID 5, but the parity data are written to two drives. The use of additional parity enables the array to continue to function even if two

disks fail simultaneously. However, this extra protection comes at a cost. RAID 6 has a slower write performance than RAID 5.



The chances that two drives break down at the same moment are minimal. However, if a drive in a RAID 5 system died and was replaced by a new drive, it takes a lot of time to rebuild the swapped drive. If another drive dies during that time, you still lose all of your data. With RAID 6, the RAID array will even survive that second failure also.