N.B.K.R. INSTITUTE OF SCIENCE AND TECHNOLOGY::VIDYANAGAR
(AUTONOMOUS)
*****
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

II-B.Tech. II SEM(R-23)

OPERATING SYSTEMS (23CS22T1)

(Common to CSE AND IT)

UNIT-I

Operating Systems Overview: Introduction, Operating system functions, Operating systems operations.

System Structures: Operating System Services, User and Operating-System Interface, system calls, Types of System Calls, system programs, Operating system Design and Implementation, Operating system structure.

UNIT-II

Processes: Process Concept, Process scheduling, Operations on processes, Inter-process communication.

Threads and Concurrency: Multithreading models, Thread libraries, Threading issues.

CPU Scheduling: Basic concepts, Scheduling criteria, Scheduling algorithms, Multiple processor scheduling.

UNIT-III

Synchronization Tools: The Critical Section Problem, Peterson's Solution, Mutex Locks, Semaphores, Monitors, Classic problems of Synchronization.

Deadlocks: system Model, Deadlock characterization, Methods for handling Deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from Deadlock.

UNIT-IV

Memory-Management Strategies: Introduction, Contiguous memory allocation, Paging, Structure of the Page Table, Swapping.

Virtual Memory Management: Introduction, Demand paging, Copy-on-write, Page replacement, Allocation of frames, Thrashing.

Storage Management: Overview of Mass Storage Structure, HDD Scheduling, RAID.

UNIT-V

File System: File System Interface: File concept, Access methods, Directory Structure; **File system Implementation:** File-system structure, File-system Operations, Directory implementation, Allocation method, Free space management; **File-System Internals:** File System Mounting, Partitions and Mounting, File Sharing.

Protection: Goals of protection.

UNIT-I

Operating Systems Overview: Introduction, Operating system functions, Operating systems operations.

System Structures: Operating System Services, User and Operating-System Interface, system calls, Types of System Calls, system programs, Operating system Design and Implementation, Operating system structure.

1.1 Operating Systems Overview: Introduction

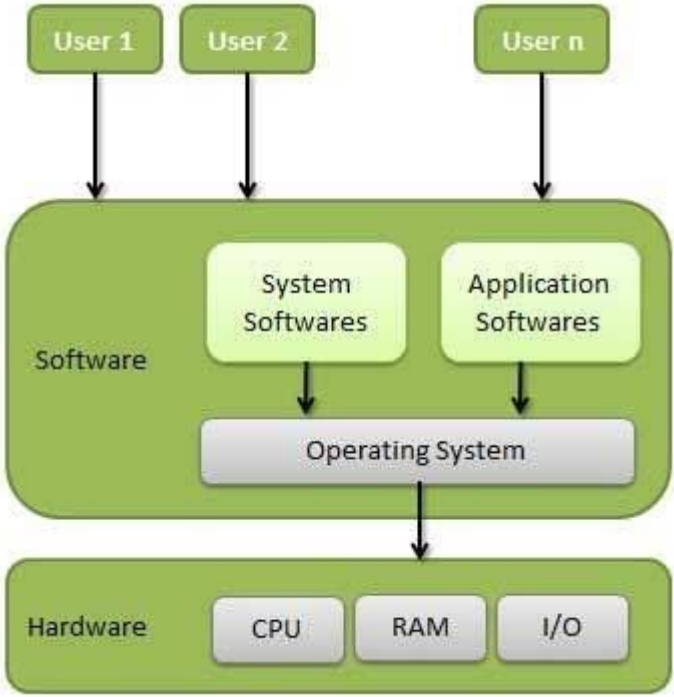
An Operating System (OS) is an interface between computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

The primary purposes of an Operating System are to enable applications to interact with a computer's hardware and to manage a system's hardware and software resources.

Some popular Operating Systems include Linux Operating System, Windows Operating System etc. Today, Operating system is found almost in every device like mobile phones, personal computers, mainframe computers, automobiles, TV, Toys etc.

Architecture

We can draw a generic architecture diagram of an Operating System which is as follows:



Operating System Generations

GENERATION	YEAR	TECHNOLOGY	OPERATING SYSTEM	SPECIFIC COMPUTERS
First Generation	**1946 – 1959**	Vacuum Tubes	None	ENIAC, IBM-701, IBM-650 etc.
Second Generation	**1959 – 1965**	Transistors	None	IBM 1401, B5000 etc.

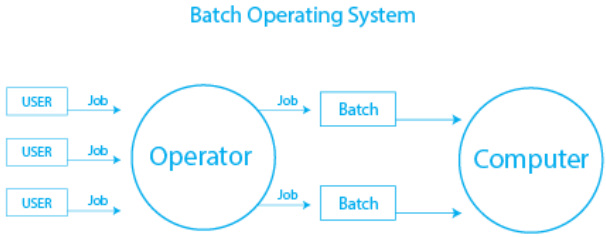
Third Generation	**1965 – 1971**	ICs	Yes	IBM-360, IBM-370, HP 2100A etc.
Fourth Generation	**1971 – 1980**	LSI and VLSI	Yes	IBM PC, STAR 1000 etc.
Fifth Generation	**1980– onwards**	Artificial Intelligence, ExpertSystems, Natural Language	Yes	Desktop, Laptop,Note Book, Ultra Book etc.

Types of Operating System

The operating system can be of different types. They are as follows:

a) Batch OS

Jobs with similar requirements are grouped together and processed as a batch. This saves time and makes the best use of the computer's resources.

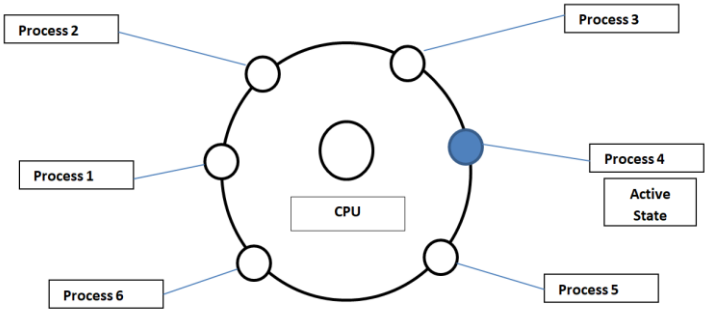


Example: payroll system and Bank statement

b) Time-Shared OS

Time sharing OS allows the user to perform more than one task at a time, each task getting the same amount of time to

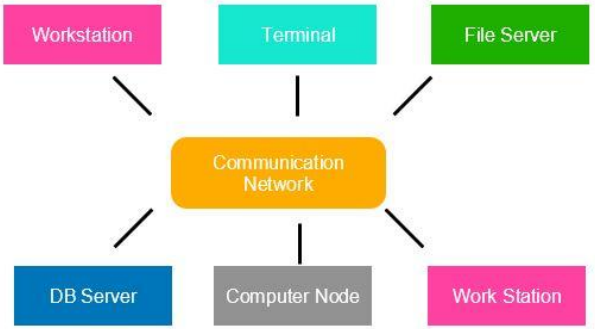
execute. Hence, the name time sharing OS. Multiple jobs are running at the CPU time and also, they use the CPU simultaneously.



Example: UNIX

c) Distributed OS

A distributed operating system (DOS) is a type of operating system that connects multiple independent computer systems through a single communication channel.



Example: LOCUS

d) Network OS

A network operating system (NOS) is software that connects multiple devices and computers on the network and allows them to share resources on the network.

Example: UNIX, LINUX and Microsoft Windows server 2008

e) Real-Time OS

A real-time operating system (RTOS) is a software program that manages tasks and system resources with a high degree of precision and reliability, and within strict time constraints.

Example: Missile systems, robots

1.2 Functions of Operating System

An operating system (OS) has many functions, including:

Managing resources: The OS manages the computer's resources, such as the CPU, RAM, and hard disk. It uses these resources to perform tasks like opening applications, printing documents, and accessing the internet.

Managing files: The OS manages files from creation to deletion, including their security, status, and storage location.

Security: The OS protects data and files from unauthorized access through access controls, user authentication, and data protection mechanisms.

Monitoring and improving performance: The OS monitors the system's performance and works to improve it. It also continuously regulates the system to detect bugs and errors.

Booting: The OS is loaded into the main memory when the computer starts, a process called booting.

User interface: The OS provides a graphical user interface (GUI) that allows the user to interact with the computer.

Managing processes: The OS oversees all running programs and processes.

Managing memory: The OS manages the computer's memory.

Device management: The OS controls peripheral devices like printers and disk drives.

Networking: The OS handles networking.

Task scheduling: The OS schedules tasks.

1.3 Operating systems operations

There are various components of an Operating System to perform well defined tasks. Though most of the Operating Systems differ in structure but logically they have similar components. Each component must be a well-defined portion of a system that appropriately describes the functions, inputs, and outputs.

There are following 8-components of an Operating System:

a. Process Management
b. I/O Device Management
c. File Management
d. Network Management
e. Main Memory Management
f. Secondary Storage Management
g. Security Management
h. Command Interpreter System

a) Process Management

Process management in an operating system (OS) is the process of creating, scheduling, and terminating programs, or processes, that are running within the system:

Process

A program that is currently running. Each process has a program counter that represents its current activity, and a set of resources allocated by the OS.

Process management

The process of managing the creation, scheduling, termination, and synchronization of processes within the system.

Purpose

Process management ensures that resources are used efficiently, the system remains stable, and applications run smoothly. It also controls how programs interact with each other.

b) I/O Device Management

Input/Output (I/O) device management in an operating system (OS) is how the OS handles input and output operations between the computer and its external devices. The OS acts as a mediator between the devices and the running programs, ensuring efficient access to these resources.

Here are some aspects of I/O device management in an OS:

Device drivers

Responsible for all aspects of device access, management, and control. This includes how requests are passed from higher level components to the device and how those components respond to errors or notifications from the device.

Buffering

Involves temporarily storing data in memory before it is written to a storage device, or after it is read from a storage device. This can improve performance by reducing the number of I/O operations required.

Direct Memory Access (DMA) controller

A control unit that can transfer blocks of data between I/O devices and main memory with minimal intervention from the processor.

Spooling

The process of managing the data flow to devices that can't handle interleaved data streams, like printers.

Device control

The management of I/O devices such as a keyboard, magnetic tape, disk, printer, microphone, USB ports, scanner, camcorder, etc.

c) File Management

File management in an operating system (OS) is the process of organizing, storing, and manipulating files on a computer. It's a fundamental and crucial component of an OS.

File management in an OS involves:

- Creating, modifying, and deleting files
- Storing files in a hierarchical file system
- Managing nonvolatile storage
- Ensuring data integrity and security
- Making files easily searchable and retrievable

Some fundamental concepts in file management include:

Consistency: Standardizing file naming, folder structure, and metadata use

Primary and backup: Protecting the primary copy of data with backup copies

Originals and derivatives: Storing camera originals separately from derivatives.

d) Network Management

Network management is the process of configuring, monitoring, and maintaining a network, and the tools used to do so. An operating system (OS) plays a key role in

network management by managing the network interface card (NIC) and network protocols.

Here are some ways an OS manages a network:

NIC management: The OS ensures the NIC works properly by managing its operation.

Network protocol management: The OS manages network protocols, which are rules and conventions that govern data transmission and reception.

Data transmission and reception management: The OS manages data transmission and reception.

Encapsulation process management: The OS controls the encapsulation process, which involves packaging data into packets for transmission on the network.

Network service management: The OS manages network services like email, file sharing, and print sharing.

Network security management: The OS manages the firewall, which prevents unauthorized access to a network or system. The OS also manages the authentication process, which involves identifying users before granting access to a network.

e) Main memory Management

Memory management is the process of controlling and coordinating a computer's main memory. It ensures that blocks of memory space are properly managed and allocated so the operating system (OS), applications and other running processes have the memory they need to carry out their operations.

Main memory management in an operating system (OS) is the process of managing a computer's primary memory to ensure that the OS, applications, and other processes have the memory they need to run:

Tracking memory status: Memory management keeps track of each memory location's status, whether it's allocated or free.

Allocating memory: Memory management decides which processes get memory, when they receive it, and how much they are allowed.

Deallocating memory: Memory management releases memory space when it's no longer needed.

Moving processes: Memory management moves processes between main memory and disk during execution.

Optimizing memory usage: Memory management strives to optimize memory usage so the CPU can access the instructions and data it needs.

Preventing unauthorized access: Memory management prevents unauthorized memory access to any process.

f) Secondary Storage Management

Secondary memory management in an operating system (OS) involves organizing data, controlling access, and handling data transfers. The OS manages secondary storage devices, such as hard drives and solid-state drives, using a variety of methods, including:

File systems: Organizes data on secondary storage devices, allowing for storage, retrieval, and updating. The file system also manages free space on the storage device.

Access control: Controls who can access data on a storage device and what they can do with it.

Buffering: Helps manage secondary storage.

The OS also manages secondary storage by:

- Allocating storage space when new files are written
- Scheduling requests for memory access

Secondary memory is non-volatile and is used for long-term data storage. It's much slower than RAM, so the OS needs to ensure that frequently accessed data stays in main memory. Less-used data is swapped out to secondary storage.

g) Security Management

Security management for an operating system (OS) is the process of protecting a computer system from internal and external threats. It involves:

Identifying assets: Cataloguing the assets of the organization, such as people, technology, data, and physical facilities

Developing policies: Creating and documenting policies and procedures to protect the assets

Implementing policies: Putting the policies and procedures into action

Maintaining policies: Regularly reviewing and improving the policies and procedures

Training staff: Ensuring that staff are aware of their security roles and are trained to support security efforts

Monitoring user activity: Assessing how well the security implementation is working

Some examples of security features in an OS include:

Secure Boot and Trusted Boot: These features work together to ensure that the system boots up safely and securely

h) Command Interpreter System

A command interpreter allows the user to interact with a program using commands in the form of text lines. It was frequently used until the 1970's. However, in modern times many command interpreters are replaced by graphical user interfaces and menu-driven interfaces.

1.4 System Structures: Operating System Services

An operating system is software that acts as an intermediary between the user and computer hardware. It is a program with the help of which we are able to run various applications. It is the one program that is running all the time. Every computer must have an operating system to smoothly execute other programs.

Here are the main categories and types of services typically provided by an OS:

a. Program execution
b. Input Output Operations
c. Communication between Processes

d. File Management
e. Memory Management
f. Process Management
g. Security and Privacy
h. Resource Management
i. User Interface
j. Networking
k. Error handling
l. Time Management

a) Program Execution

The OS provides an environment for executing applications. It includes:

- **Loading programs into memory**.
- **Running and terminating programs** when the user or another process instructs it to.
- **Memory and resource allocation** for applications and managing execution times.

b) Input Output Operations

The OS manages input and output devices and services for reading from and writing to devices. Examples include:

- Disk and file I/O for saving and retrieving data.

- Device management for handling printers, keyboards, display screens, and external storage.
- Device drivers that facilitate communication between hardware and software.

c) Communication between Processes

The Operating system manages the communication between processes. Communication between processes includes data transfer among them. If the processes are not on the same computer but connected through a computer network, then also their communication is managed by the Operating System itself.

d) File Management

The operating system helps in managing files also. If a program needs access to a file, it is the operating system that grants access. These permissions include read-only, read-write, etc. It also provides a platform for the user to create, and delete files. The Operating System is responsible for making decisions regarding the storage of all types of data or files, i.e, floppy disk/hard disk/pen drive, etc. The Operating System decides how the data should be manipulated and stored.

e) Memory Management

Memory management is crucial for allocating and organizing a system's memory resources, including:

- Allocating memory to processes and deallocating it when processes terminate.
- Tracking free and used memory to avoid conflicts.
- Virtual memory management, allowing applications to use more memory than is physically available by swapping data between RAM and storage.

f) Process Management

Process management services help the OS handle multiple tasks concurrently. These services include:

- Creating and terminating processes.
- Scheduling to determine which processes run when, based on priority and availability.
- Inter-process communication (IPC) to enable data exchange between processes.
- Process synchronization to prevent conflicts when multiple processes access shared resources.

g) Security and Privacy

Security services help prevent unauthorized access and ensure data integrity:

- User authentication, including passwords, biometrics, or multi-factor authentication.
- Access control to restrict access to files, processes, and devices.
- Encryption for protecting sensitive data, especially in networked or multi-user environments.

h) Resource Management

The OS monitors and optimizes the use of resources like CPU, memory, and disk storage:

- CPU and memory allocation based on application or user needs.
- Resource tracking and usage monitoring to manage system performance.
- Billing and accounting for resource usage, often used in cloud or multi-user environments to track resource consumption.

i) User Interface

- **Command-Line Interface (CLI):** Allows users to interact with the OS by typing commands. Common in UNIX/Linux and Windows (Command Prompt).
- **Graphical User Interface (GUI):** Provides a visual interface with windows, icons, and menus for user

interaction, making the system more accessible for users who prefer graphical navigation.

- **Touch Interface:** Common in mobile OSs, allowing users to interact with the device using touch gestures.

j) Networking

Modern OS provides services for network management, which is essential for connecting and communicating over the internet or local networks:

- Network protocols to enable communication, such as TCP/IP.
- Remote access to allow users to access and control a system from another location.
- File and resource sharing across the network, allowing users to share files, printers, and other resources.

k) Error handling

The OS continuously monitors the system for errors to ensure smooth operation:

- Detecting hardware and software errors, such as memory allocation issues or device failures.

- Recovering from errors by attempting corrective actions.
- Logging errors for system administrators and users to review

l) Time Management

Time management is the strategic process of organizing and planning how to allocate one's time effectively for maximum productivity. Time management is crucial for efficient handling of multiple processes, system resources, and task prioritization. The OS is responsible for keeping track of time and scheduling processes to ensure that each has fair access to the CPU and that critical tasks are completed on time.

1.5 User and Operating-System Interface

The User and Operating-System Interface refers to the layer through which users interact with the operating system (OS) and access its functionalities. This interface serves as a bridge between the user and the OS's core features, allowing users to execute commands, manage files, and configure system settings. There are two primary types of user

interfaces in an OS: the Command-Line Interface (CLI) and the Graphical User Interface (GUI).

1. Command-Line Interface (CLI)

The Command-Line Interface provides a text-based way for users to interact with the operating system by typing commands. The CLI is common in many operating systems, especially for technical or server environments where efficient, script-based management is preferred.

Key Features of CLI:

- **Direct Command Entry:** Users type commands directly to control the system, such as managing files, configuring settings, or running programs.
- **Scripting and Automation:** Users can create scripts to automate repetitive tasks, making CLI useful for system administration.
- **Efficiency:** For experienced users, the CLI can be faster and more flexible than a GUI, as it allows direct access to commands and shortcuts.

Examples of CLI:

- **UNIX/Linux Shells:** Such as Bash, Zsh, and C Shell.

- **Windows Command Prompt:** The command interpreter for Windows, also known as cmd.exe.
- **PowerShell:** An advanced command-line interface in Windows with powerful scripting capabilities.

2. Graphical User Interface (GUI)

The Graphical User Interface is a visual way for users to interact with the OS through windows, icons, menus, and other graphical elements. GUIs make the OS more accessible and user-friendly, especially for general users who may not be familiar with command-line commands.

Key Features of GUI:

- **Point-and-Click Interaction:** Users can interact with the OS by clicking, dragging, and selecting graphical elements with a mouse, touchpad, or touchscreen.
- **Icons and Windows:** Files, folders, applications, and system functions are represented by icons. Multiple applications can be opened in separate windows, making multitasking more intuitive.
- **User-Friendliness:** GUIs are designed to be visually intuitive, making them ideal for users with little technical knowledge.

- **Multi-Tasking and Desktop Management:** Users can manage and switch between multiple applications and windows easily.

Examples of GUI:

- **Windows OS Desktop Environment:** Provides a GUI with taskbars, start menus, and windows for different applications.
- **macOS Finder and Dock:** macOS provides a desktop environment with a dock for applications, Finder for file management, and other graphical tools.
- **Linux Desktop Environments:** GNOME, KDE Plasma, and Xfce are common desktop environments in Linux, each offering different GUI features and layouts.

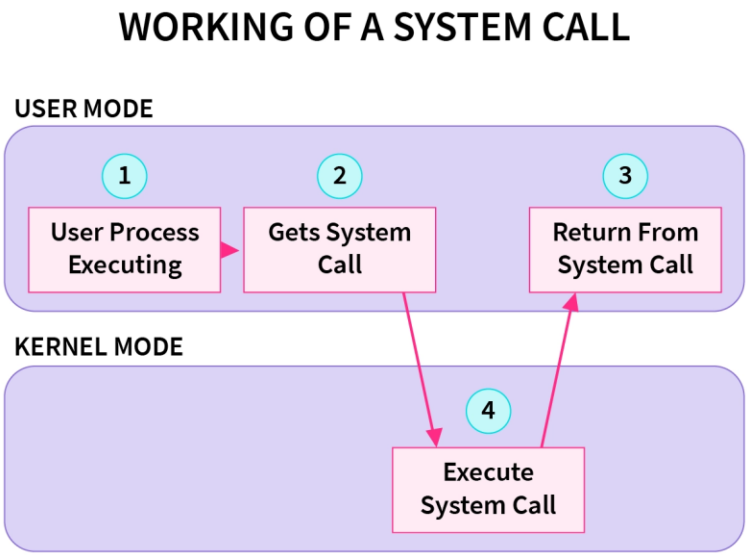
1.6 System Calls, Types of System Calls

A system call is a way for a computer program to request services from the operating system (OS) it's running on. System calls are an essential interface between the program and the OS.

In our computer system, we have two modes available.

User Mode: In this mode, execution is done on behalf of the user.

Monitor/Kernel-Mode: In this mode, execution is done on behalf of OS.



So when the process is under execution process executes under user mode, but when the process requires some OS resources to complete the task, the process makes a system call.

Here are some things that a system call can do:

- Access hardware, like a hard disk drive or camera

- Create and run new processes
- Communicate with kernel services, like process scheduling
- Manage resources, like memory

Some examples of system calls include:

Open: Opens or creates files, and specifies access mode and permissions

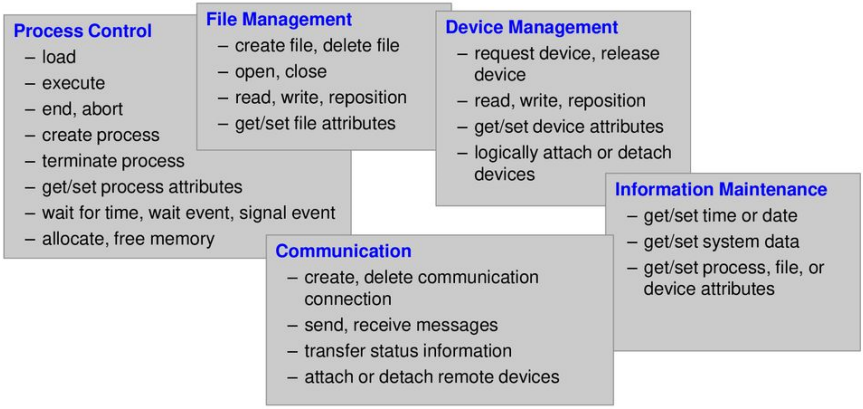
Read and write: Reads and writes data to files

Fork: Creates a child process, which is a copy of the parent process

Exec: Loads a new program into the current process's address space

Exit: Terminates a program and returns status to the parent process.

Types of System Calls



1. Process Control System Calls

Manage and control processes, including their execution and termination.

Examples include:

fork(): Creates a new process by duplicating an existing one.

exec(): Replaces the current process memory with a new program.

wait(): Suspends the calling process until a child process terminates.

exit(): Ends a process.

getpid(): Returns the process ID.

kill(): Sends a signal to terminate a process.

2. File Management System Calls

Handle files and directories, allowing creation, deletion, reading, and writing of files.

Examples include:

open(): Opens a file.

read(): Reads data from a file.

write(): Writes data to a file.

close(): Closes an open file.

unlink(): Deletes a file.

lseek(): Moves the read/write pointer within a file.

3. Device Management System Calls

Control device operations, allowing user processes to interact with hardware devices.

Examples include:

ioctl(): Performs device-specific input/output operations.

read() and write(): Also used for devices as they can be accessed as files.

open() and close(): Open and close device files.

4. Communication System Calls

Enable processes to communicate, typically in inter-process communication (IPC) or networked environments.

Examples include:

pipe(): Creates a unidirectional communication channel between processes.

shmget(), shmat(): Access shared memory.

msgget(), msgsnd(), msgrcv(): Create and access message queues.

socket(), connect(), send(), recv(): Used for network communication over sockets.

5. Information Maintenance System Calls

Provide information about processes, files, and system configuration.

Examples include:

getpid(): Returns the process ID of the current process.

alarm(): Sets a timer that sends a signal after a specified interval.

sleep(): Delays the process for a specified amount of time.

getcwd(): Gets the current working directory.

setuid(): Sets user ID of a process.

1.7 System Programs

System programs, also known as system software, are a collection of programs that control a computer's operations and run in the background. Some examples of system programs include:

Operating systems: Manages hardware resources and provides an environment for application software to run. examples like Windows, macOS, Linux, UNIX, and Ubuntu.

File management systems: These utility programs help manage a computer's files.

Device Drivers: Facilitate communication between the OS and hardware components like printers, graphics cards, and network adapters.

Utilities: Perform specific, often system maintenance, tasks, like disk cleanup, antivirus scanning, data backup, or system monitoring.

Compilers and Interpreters: Translate code written in programming languages (like C++ or Python) into machine language so the computer can execute it.

Assemblers: Convert assembly language code into machine code, essential in low-level programming tasks.

System Libraries: Contain code and data that other programs can use. They provide a standard way for applications to access system functions.

Antivirus programs: These programs help protect a computer from viruses and include examples like McAfee Antivirus, Quickheal Antivirus, and Windows Defender.

System programs are fundamental because they handle low-level operations, ensuring applications can run smoothly and efficiently on the hardware.

1.8 Operating system Design and Implementation

Designing and implementing an operating system (OS) involves creating software that manages hardware resources, runs applications, and provides user interfaces. OS design is complex because it requires balancing efficiency, reliability, security, and ease of use. Here's an overview of the key aspects involved:

1. Design Goals and Requirements

- **User Goals:** These focus on what users want from an OS, such as reliability, security, efficiency, and ease of use.

- **System Goals:** These include resource utilization, scalability, and maintainability. System goals often prioritize efficient use of CPU, memory, and other resources.

2. Types of Operating Systems

- **Batch Systems:** Process jobs without user interaction; commonly used in older systems.
- **Time-Sharing (Multitasking) Systems:** Allow multiple users or tasks to share system resources simultaneously.
- **Distributed Systems:** Enable multiple computers to work together over a network, appearing as one cohesive system.
- **Real-Time Systems:** Provide immediate processing for time-sensitive tasks, often used in embedded systems like medical or automotive devices.
- **Embedded Systems:** Tailored OS for specific devices, with limited resources, such as smartphones or IoT devices.

3. Core Components of OS Design

- **Kernel:** The core part of the OS, managing system resources and communication between hardware and software.
- **Monolithic Kernel:** A single large kernel that provides most OS functions (e.g., Linux).
- **Microkernel:** Minimalist design, only handling essential functions (e.g., Mach).
- **Process Management:** Manages processes and threads, including their scheduling, synchronization, and termination.
- **Memory Management:** Allocates and manages memory space, tracks free memory, and implements techniques like paging and segmentation.
- **File System Management:** Manages files and directories, providing access methods and maintaining metadata.
- **Device Management:** Coordinates the operation of peripheral devices through device drivers.
- **Security and Protection:** Implements security measures to protect against unauthorized access, ensuring safe multitasking.

4. Operating System Interfaces

- **Command-Line Interface (CLI):** Allows text-based commands to control the OS. Useful for advanced users.
- **Graphical User Interface (GUI):** Uses graphical elements for user interaction, focusing on ease of use.
- **Application Programming Interface (API):** Provides interfaces for application programs to access OS services.

5. Process and Thread Management

- **Processes:** An OS executes programs as processes. Each process has its memory, execution context, and resources.
- **Threads:** Lightweight processes within a single process. Threads allow multiple sequences of execution, improving efficiency in multicore processors.
- **Scheduling:** Determines the order of process or thread execution based on algorithms (e.g., round-robin, priority-based).

6. Memory Management Techniques

- **Paging:** Divides memory into fixed-sized pages to manage physical and virtual memory separately.
- **Segmentation:** Divides memory into segments based on logical divisions in a program (e.g., code, data).
- **Virtual Memory:** Provides the illusion of a larger memory size by using disk space as an extension of RAM.

7. File System Design

File Organization: Defines how data is stored, retrieved, and organized within files and directories.

- **Access Methods:** Determines how files are accessed, such as sequential or direct access.
- **File Allocation:** Techniques like contiguous, linked, or indexed allocation manage how disk space is assigned to files.
- **Data Structures:** Indexing, inodes, and directories help manage file metadata and improve access times.

8. Security and Protection Mechanisms

- **Authentication:** Verifies user identity with passwords, biometrics, or other methods.

- **Authorization:** Grants permissions based on user roles or access levels.
- **Encryption:** Secures data storage and communication.
- **Access Control:** Implements policies for user and process access to resources, often with mechanisms like Access Control Lists (ACLs).

Implementing an Operating System

Programming Language: OS is often implemented in C/C++ due to its performance and system-level access.

Development Phases:

- **Planning:** Defining the OS scope, design goals, and feature requirements.
- **Prototyping:** Building basic functionality to test key components.
- **Development:** Coding modules like kernel, device drivers, and system utilities.
- **Testing:** Running the OS on various configurations to detect and fix bugs.
- **Deployment:** Distributing the OS for use, often followed by periodic updates.

Testing and Debugging

- **Simulators and Emulators:** Simulate hardware to test the OS without needing actual hardware.
- **Debugging Tools:** Tools like kernel debuggers, profilers, and analyzers help identify issues.
- **Continuous Integration and Testing:** Ensures that the OS is stable with new updates and modifications.

Challenges in OS Design

Concurrency: Managing multiple processes and resources without conflicts or deadlocks.

Resource Management: Balancing efficient use of CPU, memory, and other hardware with user demands.

Security: Constantly defending against vulnerabilities, attacks, and unauthorized access.

Portability and Scalability: Ensuring the OS works across various hardware configurations and scales with advancements.

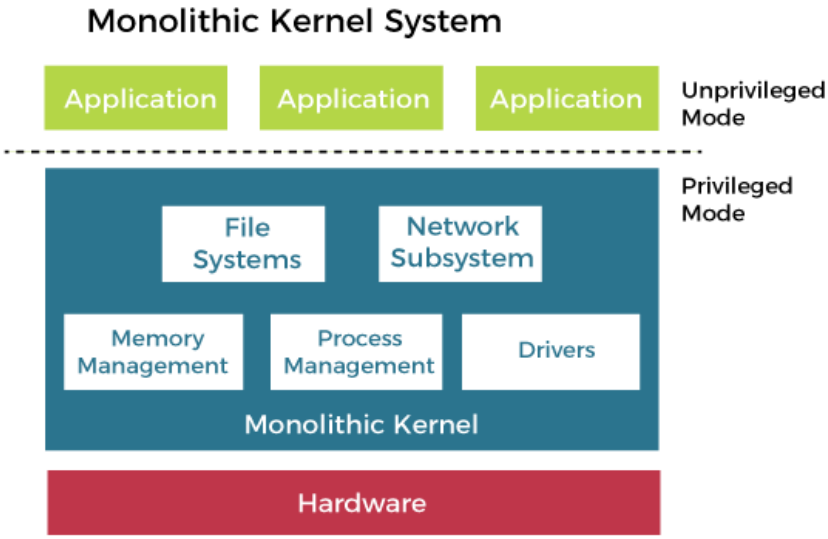
Developing an OS is a complex, iterative process requiring deep understanding of computer science, hardware, and software.

1.9 Operating System Structure

The structure of an operating system (OS) defines how its components are organized and interact. Different OS structures are designed to balance factors like performance, security, modularity, and ease of maintenance. Here are the main types of OS structures:

1. Monolithic Structure

Description: The monolithic operating system is a very basic operating system in which file management, memory management, device management, and process management are directly controlled within the kernel.



Advantages:

- Fast performance due to fewer boundaries between functions.
- Easy direct access to hardware and system calls.

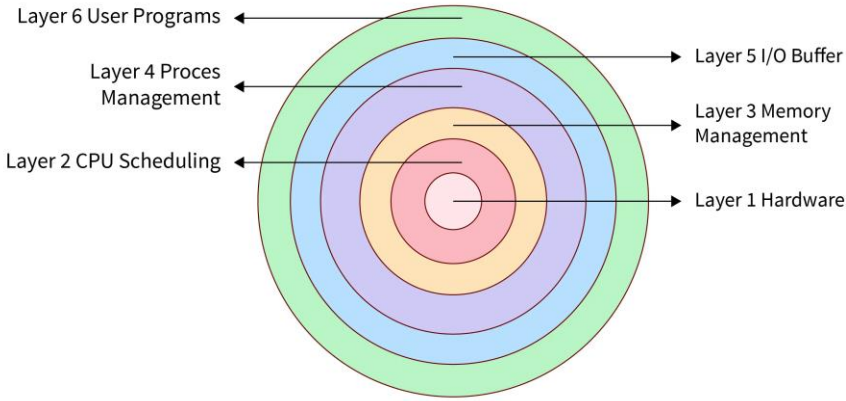
Disadvantages:

- Large and complex kernel, making it difficult to maintain and debug.
- Lack of modularity increases the risk of instability and crashes.

Examples: Early versions of UNIX, MS-DOS, and Linux (though Linux has since adopted modularity).

2. Layered Structure

Description: In a layered OS structure, the system is divided into layers, where each layer provides services to the layer above it and interacts only with the layer directly below. The bottom-most layer is the hardware, while the highest layer is the user interface.



Typical Layers:

- Hardware
- CPU scheduling, memory management
- Device management
- File system
- User interface

Advantages:

- Modularity enhances system organization and improves maintainability.
- Layers can be developed and tested independently, enhancing reliability.

Disadvantages:

- Potentially slower due to multiple layers of interaction.

- Rigid structure; each layer depends strictly on the lower one, which can be limiting.

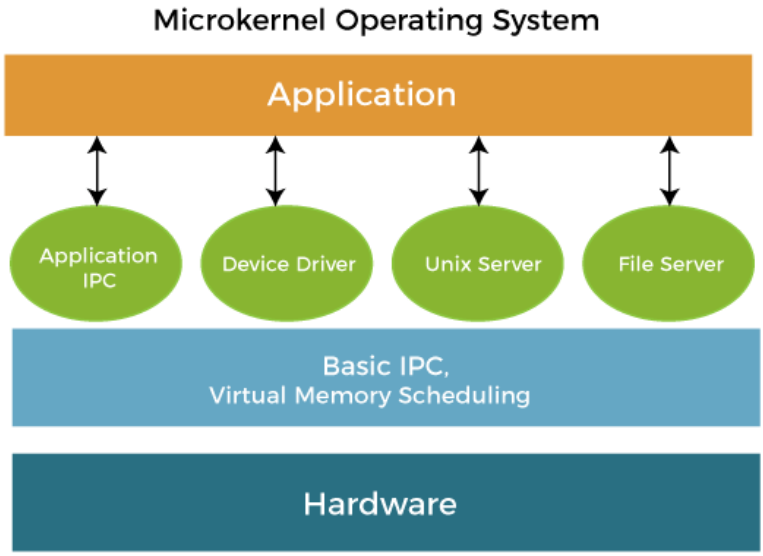
Examples: THE operating system and some versions of UNIX have adopted aspects of this design.

3. Microkernel Structure

What is a kernel?

It is the central component of an OS that handles system resources. It also acts as a bridge between the computer's application and hardware. It is one of the initial programs that is loaded when the computer boots up. When an OS is loaded, the kernel is the first component that loads into memory and rests there until the OS is shut down. It is in charge of various activities, including task management, disk management, and memory management.

Description: The microkernel structure keeps the kernel minimal, only containing essential services like communication between processes, basic memory management, and CPU scheduling. Additional services—like device drivers, file systems, and network protocols—run in user space.



In the above figure, the microkernel includes basic needs like process scheduling mechanisms, memory, and interprocess communication. It is the only program that executes at the privileged level, i.e., kernel mode. The OS's other functions are moved from the kernel-mode and execute in the user mode.

The microkernel ensures that the code may be easily controlled because the services are split in the user space. It means some code runs in the kernel mode, resulting in improved security and stability.

Advantages:

- Enhanced security and stability, as most OS services run in user mode.
- Easier to extend and maintain due to the smaller kernel.

Disadvantages:

- Potentially slower due to increased context switching and communication overhead between kernel and user services.

Examples: Minix, QNX, and macOS (based on the Mach microkernel).

4. Modular Structure

Description: The modular structure, also known as a hybrid kernel, combines the benefits of monolithic and microkernel structures. It has a core kernel but allows dynamic loading of additional modules, such as device drivers or file systems.

Advantages:

- High flexibility; modules can be loaded and unloaded as needed.
- Good performance as modules can still operate in kernel mode.

- Easier to debug and extend due to modular design.

Disadvantages:

- More complex than a simple monolithic design.

Examples: Linux, which allows dynamic loading of modules; Windows NT and its successors also use a hybrid model.

5. Client-Server Model (Distributed OS)

Description: In the client-server model, the OS is structured as a set of services that communicate over a network or across a system bus. Services (servers) provide functionality, while clients request and use these services.

Advantages:

- Scalability and modularity, as services can be distributed across multiple machines.
- Fault isolation; a crash in one service doesn't necessarily affect others.

Disadvantages:

- Can introduce significant overhead due to network latency and communication.
- More complex to design and manage as compared to traditional OS.

Examples: Distributed systems like Amoeba, Plan 9, and some aspects of Windows NT.

6. Virtual Machine Structure

Description: The virtual machine (VM) structure involves running an OS within a virtual environment, where each instance of the OS is isolated and can run multiple guest operating systems on a single hardware system. A hypervisor, or virtual machine monitor (VMM), manages these VMs.

Advantages:

- Excellent isolation and security, as each VM runs independently.
- Efficient resource utilization, allowing multiple OS instances on one physical machine.

Disadvantages:

- Adds a layer of complexity and overhead, which can affect performance.
- Challenging to manage and allocate resources among VMs.

Examples: VMware, VirtualBox, Microsoft Hyper-V, and Xen.

7. Exokernel Structure

Description: Exokernel design aims to minimize the role of the kernel by exposing low-level resources to applications. Instead of abstracting hardware resources, an exokernel provides minimal abstractions and lets applications manage their resources directly.

Advantages:

- Potentially higher efficiency, as applications can directly manage resources.
- Greater flexibility for applications needing fine-grained control over hardware.

Disadvantages:

- Complexity in application development due to low-level resource management.
- Less protection and security, as applications have direct hardware access.

Examples: MIT's Exokernel project.

Summary of OS Structures

Structure	Key Characteristics	Examples
Monolithic	Single large kernel, high performance, less modular	UNIX, MS-DOS
Layered	OS divided into hierarchical layers	THE OS, some UNIX
Microkernel	Minimal kernel, essential services only, user-mode services	Minix, macOS (Mach)
Modular	Core kernel with dynamically loaded modules	Linux, Windows NT
Client-Server	Distributed services, networked communication	Amoeba, Plan 9
Virtual Machine	Virtual OS instances with hypervisor	VMware, Hyper-V
Exokernel	Minimal kernel, direct hardware access for applications	MIT's Exokernel

Each OS structure has strengths and weaknesses, so the choice of structure depends on factors like the target application, required performance, security, modularity, and ease of maintenance.

UNIT-II

Processes: Process Concept, Process scheduling, Operations on processes, Inter-process communication.

Threads and Concurrency: Multithreading models, Thread libraries, Threading issues.

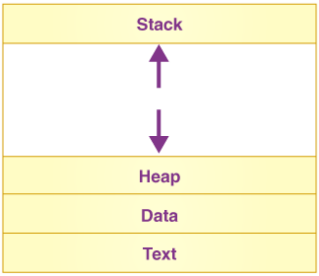
CPU Scheduling: Basic concepts, Scheduling criteria, Scheduling algorithms, Multiple processor scheduling

2.1 Processes: Process Concept

Basically, a process is a simple program. An active program which running now on the Operating System is known as the process. The Process is the base of all computing things. In other words, we write the computer programs in the form of a text file, thus when we run them, these turn into processes that complete all of the duties specified in the program.

a) Components of a Process

It is divided into the following four sections:



Stack

The process stack holds temporary data such as:

Function parameters: Arguments passed to functions within the program.

Return addresses: Addresses to which control should return after a function call.

Local variables: Variables defined within a function.

The stack grows and shrinks dynamically with function calls and returns, and it's unique to each process (and each thread within the process).

Heap

- The heap is a memory region used for dynamic memory allocation.
- It allows the process to request and allocate memory at runtime using functions like malloc() in C or new in C++.
- The heap can grow or shrink as needed, unlike the fixed size of the stack.

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

Text

This section contains the actual code or instructions of the program that the CPU will execute. It is typically read-only, ensuring that instructions don't get modified while the program runs.

Data

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

b) Process Life Cycle (States)

Processes in an operating system go through a series of states from creation to termination. Each state reflects the current activity and status of the process, helping the OS manage resources and scheduling. Here are the primary process states:

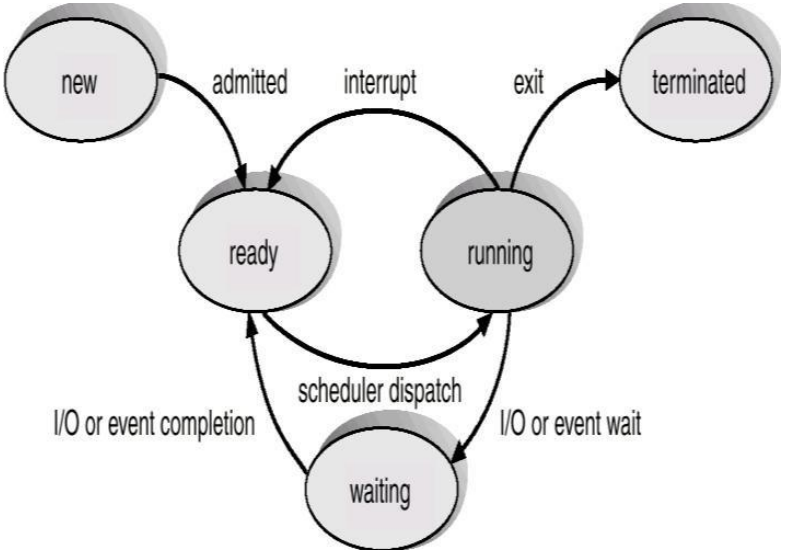
New: The process is being created.

Running: Instructions are being executed.

Waiting: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

Ready: The process is waiting to be assigned to a processor.

Terminated: The process has finished execution.



Summary of Process States

State	Description	Transition to Next State
New	Process is being created and initialized.	Moves to Ready once initialized.
Ready	Process is ready and waiting for CPU.	Moves to Running when CPU is available.
Running	Process is executing on the CPU.	Moves to Waiting (if needing I/O), Ready (if preempted), or Terminated (if

State	Description	Transition to Next State
		completed).
Waiting	Process is waiting for an event, like I/O.	Moves to Ready once the event occurs.
Terminated	Process has completed execution or been terminated.	Process is removed from memory after cleanup.

c) Process Control Block (PCB)

The information about each process is maintained in the operating system in a process control block, which is also called a task control block. Figure shows a PCB. The PCB contains extensive information about the process. The information present in the PCB includes the following:

Key Components of a Process Control Block (PCB)

Component	Description
Process ID (PID)	Unique identifier for the process.
Parent Process ID	Identifier of the parent process.

Component	Description
User and Group IDs	User and group ownership for security.
Process State	Current state of the process (e.g., Ready, Running).
Program Counter	Address of the next instruction to execute.
CPU Registers	Register values to restore during context switch.
Memory Management Info	Details on memory allocation (base, limit, page tables).
Accounting Information	Usage stats, priority, and resource limits.
I/O Status Information	Open files and I/O requests in progress.
Scheduling Information	Priority and queue pointers for scheduling decisions.
Privileges and Security	Access rights and security information.

d) Threads

Threads are a fundamental concept in operating systems, allowing a program to perform multiple tasks concurrently within a single process. Each thread represents a separate path of execution, enabling a process to accomplish tasks more efficiently by dividing work into parallel operations. Here's a comprehensive look at threads in an operating system:

Aspect	Description
Definition	A lightweight unit of execution within a process.
Types	User Threads and Kernel Threads.
Thread Models	One-to-One, Many-to-One, Many-to-Many.
Benefits	Improved responsiveness, efficient resource sharing, lower cost, and scalability on multi-cores.
Components	Each thread has its own PC, stack, and registers but shares code, data, and heap with the process.

Aspect	Description
Operations	Creation, termination, synchronization, and scheduling.
Challenges	Race conditions, deadlocks, debugging difficulties, concurrency control.

2.2 Process Scheduling

Process Scheduling is a core function of the operating system (OS) that determines the order in which processes are assigned to the CPU for execution. The scheduling mechanism is essential for managing multitasking and optimizing CPU utilization, system responsiveness, and fairness among processes.

a) Objectives of Process Scheduling

- **Maximize CPU Utilization:** Keep the CPU as busy as possible by always assigning a process to it when available.
- **Maximize Throughput:** Increase the number of processes that complete their execution in a given time frame.

- **Minimize Turnaround Time:** Reduce the time taken from process submission to completion.
- **Minimize Waiting Time:** Decrease the time a process spends waiting in the queue before it gets CPU time.
- **Minimize Response Time:** Ensure that interactive processes get quick responses, enhancing the user experience.
- **Fairness:** Ensure that all processes are given an appropriate share of the CPU based on priority or need.

b) Types of Scheduling

Pre-emptive Scheduling

- The OS can interrupt a currently running process to assign the CPU to another process, improving response time and allowing high-priority tasks to proceed quickly.

Non-pre-emptive Scheduling

- Once a process starts executing, it continues until it completes or voluntarily relinquishes control (e.g., waiting for I/O).

c) Scheduling Criteria

CPU Utilization: Keep the CPU active as much as possible.

Throughput: Number of processes completed per unit of time.

Turnaround Time: Time taken for a process to complete from submission.

Waiting Time: Total time a process spends in the ready queue.

Response Time: Time taken from submission of a request until the first response.

4. Types of Process Scheduling Algorithms

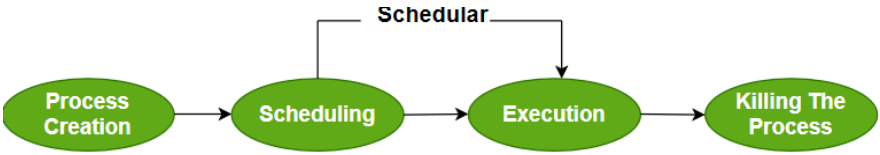
Algorithm	Preemptive	Fairness	Response Time	Waiting Time	Throughput	Starvation
FCFS	No	Poor for long jobs	Long	High	Low	Possible
SJF/SJN	No/Yes	Good	Short for short jobs	Low	High (for short jobs)	Yes, long jobs may starve

Algorithm	Preemptive	Fairness	Response Time	Waiting Time	Throughput	Starvation
Priority Scheduling	Yes/No	Good with aging	Varies	Varies	High for high-priority jobs	Yes, without aging
Round Robin (RR)	Yes	Good	Medium to short	Fair	Medium	No
Multilevel Queue	Yes	Fair with priority	Varies	Varies	Depends on policy	Yes
Multilevel Feedback Queue	Yes	Excellent	Short for high-priority jobs	Low for high-priority jobs	High for short jobs	No, with dynamic priority

Each scheduling algorithm is optimized for different performance criteria and system types, making process scheduling a critical component in achieving effective multitasking and resource utilization in operating systems.

2.3 Operations on Processes

Process operations in an operating system refer to the various activities the OS performs to manage processes. These operations include process creation, process scheduling, execution and killing the process. Here are the key process operations:



a) Process Creation

Process creation in an operating system (OS) is the act of generating a new process. This new process is an instance of a program that can execute independently.

Forking: The fork() system call is commonly used in UNIX-based systems to create a new process. The new process (child) is a duplicate of the calling process (parent) but with a unique process ID.

Exec: After a process is created, it can replace its memory space with a new program using exec() in UNIX. This is often used by the child process to start executing a different program than the parent.

Windows Process Creation: Windows has a CreateProcess() function to create a new process, allocate resources, and initialize the new process.

b) Scheduling

Once a process is ready to run, it enters the "ready queue." The scheduler's job is to pick a process from this queue and start its execution.

c) Execution

Execution means the CPU starts working on the process. During this time, the process might:

- Move to a waiting queue if it needs to perform an I/O operation.
- Get blocked if a higher-priority process needs the CPU.

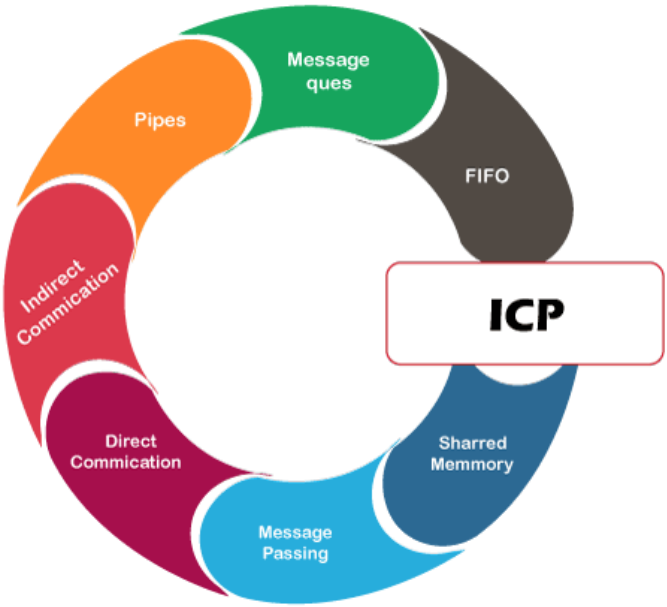
d) Killing the Process

After the process finishes its tasks, the operating system ends it and removes its Process Control Block (PCB).

2.4 Inter-process communication.

Inter-process communication (IPC) is a mechanism that allows processes to exchange data and synchronize their actions.

Here are the common IPC methods:



a) Pipes

b) Shared Memory

c) Message Queue

d) Direct Communication

e) Indirect communication

f) Message Passing

g) FIFO

a) Pipes

The pipe is a type of data channel that is unidirectional (unnamed pipes) in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel (Named Pipes) of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output.

Unnamed Pipes: Used primarily for communication between a parent and child process. Unnamed pipes are one-way communication channels that exist only for the process that created them.

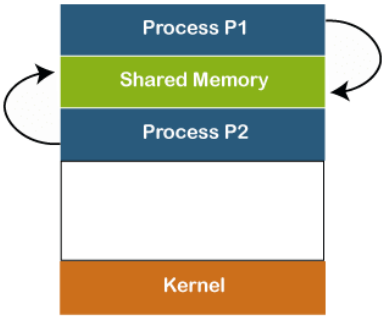
Named Pipes: These allow bidirectional communication between unrelated processes and have a name within the file system, making them accessible across processes.

b) Shared Memory

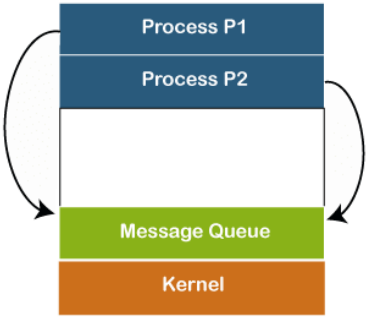
It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other.

c) Message Queue

Message queues are data structures that allow processes to send and receive messages in a queued manner.



Shared Memory



Message Queue

d) Direct Communication

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

e) Indirect communication

Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

f) Message Passing

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

- send (message)
- received (message)

g) FIFO

It is a type of general communication between two unrelated processes. It can also be considered as full-duplex, which means that one process can communicate with another process and vice versa.

Summary Table of IPC Mechanisms

IPC Mechanism	Communication Type	Pros	Cons
Pipes	Unidirectional (Unnamed), Bidirectional (Named)	Simple, lightweight	Limited to local communication
Message Queues	Bidirectional	Asynchronous, queued messages	Slower for large data
Shared Memory	Bidirectional	Fast, direct memory access	Needs synchronization
Sockets	Bidirectional	Network communication	Complex setup
Signals	Asynchronous	Quick notifications	Limited information transfer
Semaphores	Synchronization only	Effective for synchronization	No direct data communication

IPC Mechanism	Communication Type	Pros	Cons
Memory-Mapped Files	Bidirectional	Persistent storage	Requires file management
RPC	Synchronous	Remote procedure execution	Complex setup, network latency

2.5 Threads and Concurrency: Multithreading models

In operating systems, **threads** allow a process to split its tasks into multiple concurrent units of execution, improving efficiency and resource sharing. Multithreading models determine how threads are managed and how they interact with the OS. Here's an overview of the main multithreading models:

a) Many-to-One Model

b) One-to-One Model

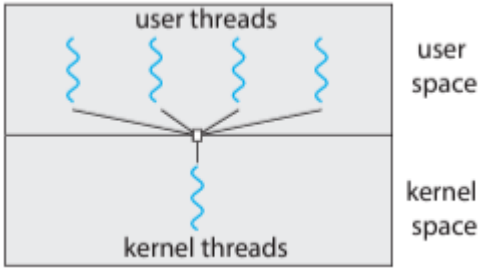
c) Many-to-Many Model

a) Many-to-One Model

Description: Multiple user-level threads are mapped to a single kernel thread.

In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time.

The thread management is done on the user level so it is more efficient.



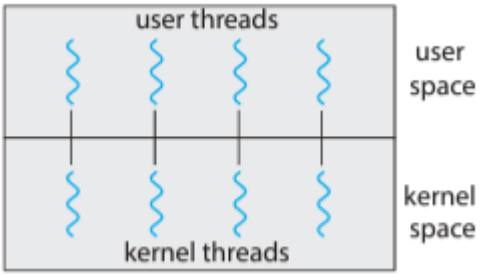
b) One-to-One Model

Description: Each user-level thread is mapped to a separate kernel thread.

In this model, one to one relationship between kernel and user thread. In this model multiple thread can run on

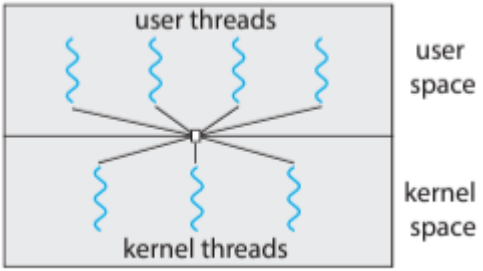
multiple processor. Problem with this model is that creating a user thread requires the corresponding kernel thread.

As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won't be blocked.



c) Many-to-Many Model

Description: Multiple user-level threads are mapped to multiple kernel threads, with the exact mapping determined by the OS or the thread library.



Summary Table of Multithreading Models

Model	User-Level Threads	Kernel-Level Threads	Concurrency	Parallelism	Complexity
Many-to-One	Multiple	One	Limited	None	Low
One-to-One	One per thread	One per thread	High	High	Moderate
Many-to-Many	Multiple	Multiple	High	Depends on mapping	High
Two-Level	Mix of both	Mix of both	High	High (when bound)	Very High

2.6 Thread libraries

Thread libraries provide APIs to create, manage, and control threads, making it easier for developers to implement multithreaded applications. Different operating systems and environments offer various thread libraries, each with

distinct features and capabilities. Here are some of the most widely used thread libraries:

1. POSIX Threads (Pthreads)

Description: Pthreads is a widely used thread library following the POSIX standard, primarily used on UNIX-based systems (Linux, macOS, etc.).

Key Functions:

pthread_create(): Creates a new thread.

pthread_join(): Waits for a thread to finish.

pthread_exit(): Terminates the calling thread.

pthread_mutex_lock() and pthread_mutex_unlock(): Lock and unlock mutexes for thread synchronization.

pthread_cond_wait() and pthread_cond_signal(): Use condition variables for signaling between threads.

2. Windows Threads

Description: Windows provides its own native threading library, integrated into the Windows API, specifically designed for the Windows operating system.

Key Functions:

CreateThread(): Creates a new thread.

ExitThread(): Ends the calling thread.

WaitForSingleObject(): Waits for a specific thread or other object to finish.

CreateMutex() and ReleaseMutex(): Create and release mutexes for synchronization.

CreateEvent(): Used for signaling between threads.

3. Java Threads (java.lang.Thread)

Description: Java provides its own threading library, allowing cross-platform multithreading as part of the Java API.

Key Classes:

Thread: Represents a thread of execution.

Runnable: Functional interface that represents a task to be executed by a thread.

ExecutorService: Manages a pool of threads for executing tasks.

synchronized keyword: Provides synchronized methods and blocks for thread-safe code.

wait() and notify(): Used for inter-thread communication.

4. C++ Standard Library Threads (C++11 and above)

Description: C++11 introduced standard threading support, providing basic thread management and synchronization mechanisms as part of the C++ Standard Library.

Key Classes and Functions:

std::thread: Represents an individual thread.

std::mutex and std::lock_guard: Basic mutexes for locking and synchronization.

std::condition_variable: Allows threads to wait for specific conditions.

std::async and std::future: Enable asynchronous function execution and result retrieval.

5. OpenMP (Open Multi-Processing)

Description: OpenMP is a portable, high-level API primarily for parallel programming on shared-memory systems, focusing on multithreading for scientific and engineering applications.

Key Directives:

#pragma omp parallel: Creates a team of threads.

#pragma omp for and #pragma omp sections: Divide tasks among threads in a team.

#pragma omp critical: Defines critical sections to avoid race conditions.

#pragma omp barrier: Synchronization point where threads wait until all reach it.

6. Intel Threading Building Blocks (TBB)

Description: Intel TBB is a C++ template library for parallel programming, focusing on task-based parallelism instead of explicit thread management.

Key Features:

- Task-based parallelism abstracts threads, allowing the library to handle workload distribution.
- Synchronization primitives like mutexes, condition variables, and atomic operations.
- Support for parallel algorithms and data structures, such as parallel loops, concurrent containers, and pipelines.

Summary Table of Thread Libraries

Library	Platform	Key Features	Typical Use Case

Library	Platform	Key Features	Typical Use Case
Pthreads	UNIX/Linux, macOS	Low-level, flexible, POSIX standard	UNIX-based systems, HPC, real-time applications
Windows Threads	Windows	Native Windows API, supports thread pooling	Windows-based applications, GUI, enterprise software
Java Threads	Cross-platform	High-level, managed by JVM, ExecutorService	Cross-platform, server-side, web services
C++ Standard Library Threads	Cross-platform	Simple C++ threading, async, futures	Real-time systems, games, system-level software
OpenMP	Cross-platform	Directives-based, shared-memory parallelism	HPC, scientific computing, numerical simulations
Intel TBB	Cross-platform	Task-based, optimized for multi-core CPUs	Data processing, scientific and financial modeling

2.7 Threading issues

Threading issues can arise when multiple threads operate concurrently, especially when they share resources or interact with each other. These issues often lead to unintended behavior, performance degradation, or system instability. Here are some common threading issues:

1. Race Conditions

Description: A race condition occurs when multiple threads access shared data simultaneously and the outcome depends on the order of execution.

2. Deadlocks

Description: A deadlock happens when two or more threads are waiting indefinitely for resources held by each other, creating a cycle of dependency that halts execution.

3. Starvation

Description: Starvation occurs when a thread is unable to access necessary resources or execute because other threads monopolize them.

4. Priority Inversion

Description: Priority inversion happens when a high-priority thread is waiting for a resource held by a lower-priority thread, while a medium-priority thread preempts the

low-priority thread, effectively delaying the high-priority thread.

5. Concurrency Issues in Read-Write Operations

Description: When multiple threads read and write shared data without proper synchronization, inconsistencies can occur, leading to unpredictable behavior.

6. Thread Leaks

Description: A thread leak occurs when a program repeatedly creates threads without properly terminating or joining them, which can exhaust system resources.

Summary Table of Threading Issues and Solutions

Threading Issue	Description	Solution
Race Conditions	Concurrent access to shared data causes errors	Use locks or synchronization primitives
Deadlocks	Threads wait indefinitely for each other's resources	Avoid circular wait, use deadlock detection
Livelocks	Threads keep adjusting but make	Use random delays or prioritized

Threading Issue	Description	Solution
	no progress	handling
Starvation	Threads are indefinitely delayed by others	Fair scheduling, priority aging
Priority Inversion	High-priority thread waits for lower-priority thread	Use priority inheritance
Read-Write Concurrency Issues	Uncoordinated read-write leads to data inconsistency	Use reader-writer locks
Thread Leaks	Threads are created without proper termination	Use thread pooling, ensure proper termination
Context Switching Overhead	Frequent context switches slow performance	Use thread pools, fewer threads
Memory	Threads see	Use memory

Threading Issue	Description	Solution
Consistency Errors	inconsistent memory views	barriers, atomic variables
Improper Thread-Local Storage Use	Incorrect sharing of local data	Limit use of TLS, encapsulate properly

2.8 CPU Scheduling: Basic concepts

CPU scheduling is the process of determining which processes in the ready queue should be allocated CPU time for execution. Since CPU time is a limited resource, efficient CPU scheduling is essential to maximize CPU utilization, improve system throughput, and ensure fair allocation among processes.

Key Concepts in CPU Scheduling

1. CPU and I/O Bursts

CPU Burst: A period during which a process is actively using the CPU to execute instructions.

I/O Burst: A period during which a process is waiting for an I/O operation (e.g., reading data from a disk).

2. Scheduling Criteria

CPU Utilization: Aim to keep the CPU as busy as possible (ideally close to 100%).

Throughput: The number of processes completed per unit of time.

Turnaround Time: The total time taken from the process's arrival to its completion (including waiting, execution, and I/O).

Waiting Time: The total time a process spends waiting in the ready queue.

Response Time: The time from a request submission to the first response, particularly important in interactive systems.

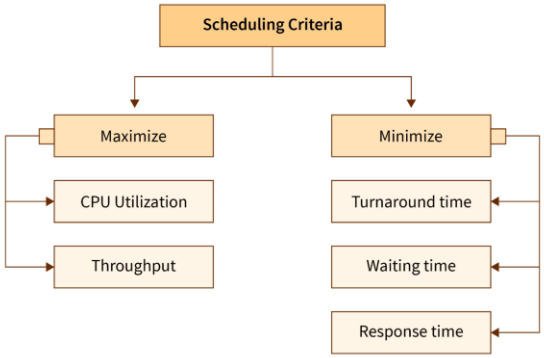
Fairness: Ensuring that each process gets a fair share of CPU time and avoids starvation.

3. Preemptive vs. Non-Preemptive Scheduling

Preemptive Scheduling: The scheduler can interrupt a currently running process to allocate the CPU to another process (e.g., due to higher priority or a time quantum expiring).

Non-Preemptive Scheduling: Once a process is given the CPU, it runs until it either finishes or enters the waiting state (e.g., for I/O).

2.9 Scheduling Criteria



Scheduling criteria are essential in evaluating and comparing different CPU scheduling algorithms. These criteria help determine how effectively an algorithm manages system resources, maximizes performance, and meets the requirements of various types of applications. Here are the primary scheduling criteria:

1. CPU Utilization

Definition: Measures the percentage of time the CPU is actively working on processes rather than being idle.

2. Throughput

Definition: The number of processes that are completed per unit of time.

3. Turnaround Time

Definition: The total time taken for a process to complete, from its arrival in the system to its completion.

4. Waiting Time

Definition: The total time a process spends in the ready queue waiting for CPU allocation.

5. Response Time

Definition: The time from when a request is submitted until the first response is produced (not the completion of the process).

6. Fairness

Definition: Ensuring that each process receives a fair share of CPU time and that no process is indefinitely postponed.

7. Predictability

Definition: The ability of a scheduling algorithm to consistently provide the same level of service, with stable waiting and turnaround times.

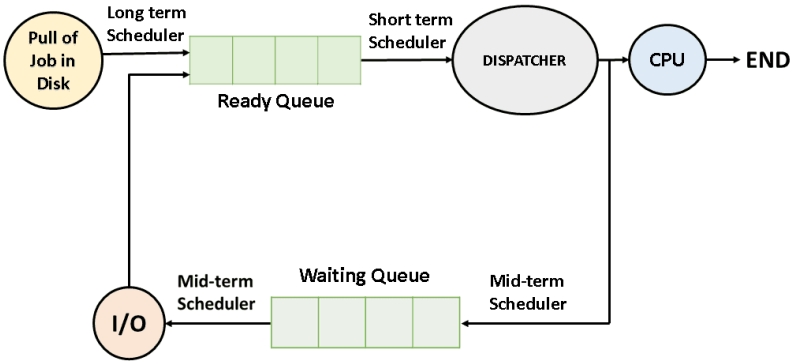
Summary of Scheduling Criteria

Scheduling Criterion	Description	Objective
CPU Utilization	Percentage of time CPU is actively working	Maximize

Scheduling Criterion	Description	Objective
Throughput	Number of processes completed per unit time	Maximize
Turnaround Time	Total time from process arrival to completion	Minimize
Waiting Time	Total time a process spends in the ready queue	Minimize
Response Time	Time from request submission to first response	Minimize
Fairness	Ensures all processes receive equitable CPU access	Maximize
Predictability	Consistent service levels, stable wait, and turnaround times	Maximize

2.10 Scheduling algorithms

CPU Scheduling is a process in operating systems that determines which process in the ready queue should be allocated to the CPU for execution. Since the CPU can only execute one process at a time, efficient scheduling is essential to maximize CPU utilization, improve system performance, and ensure processes run smoothly.



Key Goals of CPU Scheduling

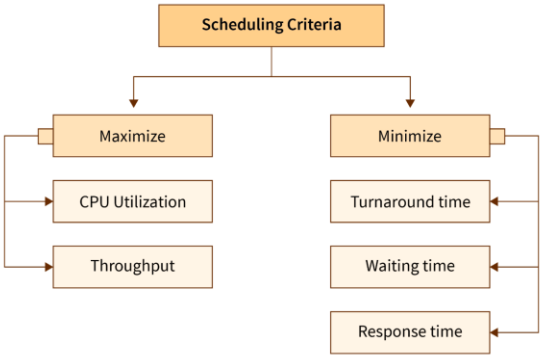
Maximize CPU Utilization: Keep the CPU as busy as possible.

Maximize Throughput: Increase the number of processes completed in a given time frame.

Minimize Waiting Time: Reduce the time processes spend in the ready queue.

Minimize Turnaround Time: Reduce the total time a process takes from arrival to completion.

Minimize Response Time: For interactive systems, it's crucial to minimize the time taken from when a request is submitted until the first response is produced.



Types of CPU Scheduling Algorithms

First-Come, First-Served (FCFS):

- Processes are scheduled in the order they arrive.
- Simple but may lead to high waiting times, especially with long processes.

Shortest Job First (SJF):

- Processes with the shortest burst time are scheduled first.

- Efficient but can cause starvation for longer processes.

Priority Scheduling:

- Each process is assigned a priority, and the CPU is allocated to the process with the highest priority.
- Works well but can lead to starvation for lower-priority processes.

Round Robin (RR):

- Each process gets a fixed time slice (quantum) in a cyclic order.
- Good for time-sharing and interactive systems; however, performance depends on the time quantum size.

Multilevel Queue Scheduling:

- Processes are grouped into different queues, each with its scheduling algorithm.
- Often used when there are distinct classes of processes, like system and user processes.

Multilevel Feedback Queue Scheduling:

- Similar to multilevel queue scheduling but allows processes to move between queues based on their behavior and requirements.

- Adaptive and efficient but complex to implement.

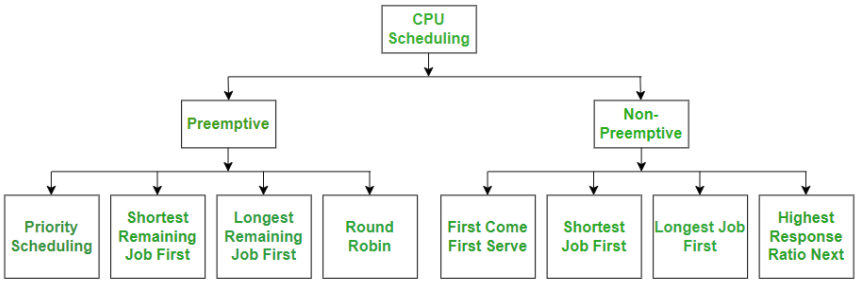
Preemptive vs. Non-Preemptive Scheduling

Preemptive Scheduling: Allows a process to be interrupted and moved to the ready queue if a higher-priority process arrives or based on other conditions.

Non-Preemptive Scheduling: Once a process is given the CPU, it keeps it until it finishes or voluntarily yields.

Algorithm	Type	Advantages	Disadvantages	Use Case
FCFS	Non-preemptive	Simple, fair in arrival order	Convoy effect	Batch processing
SJF/SJN	Non-preemptive	Minimizes average waiting time	Requires burst-time prediction, starvation	Known burst times, simulations
SRTF	Preemptive	Minimizes waiting time, more responsive	Hard to predict burst time, starvation	Short task scheduling
Round Robin (RR)	Preemptive	Fair, good for interactive systems	High context switching if quantum is too low	Time-sharing systems
Priority Scheduling	Can be both	Prioritizes important tasks	Starvation without aging	Real-time or mission-critical tasks
Multilevel	Can be	Separates and	Starvation for	Multi-purpose

Queue	both	prioritizes tasks by type	low-priority queues	systems with varied workloads
Multilevel Feedback Queue	Preemptive	Flexible, adaptive, reduces starvation	Complex configuration	General-purpose systems, OSes
Earliest Deadline First	Preemptive	Meets real-time deadlines	High overhead	Real-time systems
Lottery Scheduling	Preemptive	Fair distribution, allows priority assignment	Less predictable	Systems with approximate fairness needs



1. First Come First Serve

2. Shortest Job First

3. Priority Scheduling

4. Round Robin (RR)

1. First-Come, First-Served (FCFS) is one of the simplest CPU scheduling algorithms. In FCFS scheduling, the process that arrives first is executed first, without interruption, until it finishes. It follows a non-preemptive approach, meaning once a process starts executing, it runs until it completes.

Important Abbreviations

CPU - - - > Central Processing Unit

FCFS - - - > First Come First Serve

AT - - - > Arrival Time

BT - - - > Burst Time

WT - - - > Waiting Time

TAT - - - > Turn Around Time

CT - - - > Completion Time

FIFO - - - > First In First Out

How FCFS Scheduling Works

- Processes are scheduled in the order of their arrival times.

- Each process holds the CPU until it completes, after which the next process in line begins execution.

Advantages of FCFS

- Simple and easy to implement.
- Fair in terms of process arrival time (first come, first served).

Disadvantages of FCFS

- **Convoy Effect:** If a long process is ahead in the queue, shorter processes must wait, leading to increased waiting times and inefficient CPU utilization.
- **High Average Waiting Time:** Processes arriving later can experience significant delays if earlier processes have long burst times.

Example of FCFS Scheduling

Suppose we have the following processes arriving in the order shown below, with their respective CPU burst times:

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	8
P4	3	6

Step-by-Step Execution

Calculate Start and Finish Times for Each Process

Calculate Waiting and Turnaround Times

Waiting Time = Start Time - Arrival Time

Turnaround Time = Finish Time - Arrival Time

Process	Arrival Time	Burst Time	Start Time	Finish Time	Waiting Time	Turnaround Time
P1	0	5	0	5	0	5
P2	1	3	5	8	4	7
P3	2	8	8	16	6	14
P4	3	6	16	22	13	19

Calculate Average Waiting Time and Turnaround Time

- **Average Waiting Time** = (0 + 4 + 6 + 13) / 4 = 5.75
- **Average Turnaround Time** = (5 + 7 + 14 + 19) / 4 = 11.25

Gantt Chart

```
| P1 | P2 | P3 | P4 |
0    5    8    16   22
```

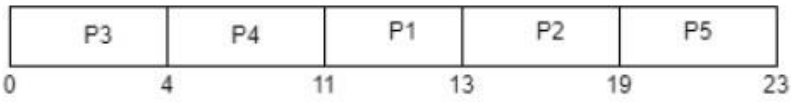
In this example, P4 has to wait a significant time due to the non-preemptive nature of FCFS, illustrating the convoy effect, where shorter jobs wait behind a longer one (P3).

Example2:

Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	5	6
P3	0	4
P4	0	7
P5	7	4

Gantt chart



For this problem CT, TAT, WT, RT is shown in the given table −

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	13	13-2= 11	11-2= 9	9
P2	5	6	19	19-5= 14	14-6= 8	8
P3	0	4	4	4-0= 4	4-4= 0	0
P4	0	7	11	11-0= 11	11-7= 4	4
P5	7	4	23	23-7= 16	16-4= 12	12

Average Waiting time = (9+8+0+4+12)/5 = 33/5 = 6.6 time unit (time unit can be considered as milliseconds)

Average Turn-around time = (11+14+4+11+16)/5 = 56/5 = 11.2 time unit (time unit can be considered as milliseconds).

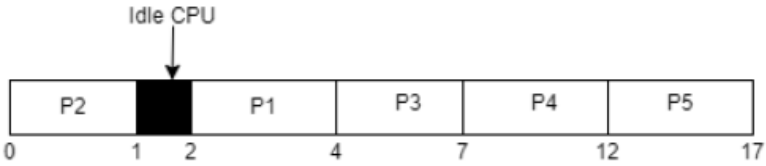
Example 3

Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	0	1
P3	2	3
P4	3	5
P5	4	5

Solution

Gantt chart –



For this problem CT, TAT, WT, RT is shown in the given table –

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	4	4-2= 2	2-2= 0	0
P2	0	1	1	1-0= 1	1-1= 0	0
P3	2	3	7	7-2= 5	5-3= 2	2
P4	3	5	12	12-3= 9	9-5= 4	4
P5	4	5	17	17-4= 13	13-5= 8	8

Average Waiting time = (0+0+2+4+8)/5 = 14/5 = 2.8 time unit (time unit can be considered as milliseconds)

Average Turn-around time = (2+1+5+9+13)/5 = 30/5 = 6 time unit (time unit can be considered as milliseconds)

2. Shortest Job First

- Shortest Job First (SJF) is a scheduling algorithm commonly used in operating systems to manage processes in the CPU. It selects the process with the

shortest execution time (burst time) to execute next, reducing the average waiting time for processes.

Types of SJF

Non-preemptive SJF: Once a process starts executing, it cannot be interrupted until it completes.

Preemptive SJF (Shortest Remaining Time First - SRTF): The currently running process can be interrupted if a new process arrives with a shorter burst time than the remaining time of the current process.

We'll solve both **Non-preemptive SJF** and **Preemptive SJF (Shortest Remaining Time First - SRTF)** using the same set of processes.

Example Problem

Consider the following set of processes with their **arrival time** and **burst time**:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

We'll solve this using both **Non-preemptive SJF** and **Preemptive SJF (SRTF)**.

1. Non-preemptive SJF

In Non-preemptive SJF, once a process starts executing, it cannot be interrupted until it completes.

- At time 0, P1 is the only process available, so it starts executing.
- P1 completes at time 8 (0 + 8).
- At time 8, all other processes (P2, P3, and P4) have arrived. The process with the shortest burst time is P2 (4), so P2 executes next.
- P2 completes at time 12 (8 + 4).
- Now at time 12, the remaining processes are P3 and P4. The shortest burst time is P4 (5), so P4 executes next.
- P4 completes at time 17 (12 + 5).
- Finally, P3 is the only process left, so it executes.
- P3 completes at time 26 (17 + 9).

Non-preemptive SJF Execution Order

The order of execution is P1→ P2→ P4→P3.

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	8	8	0
P2	1	4	12	11	7
P3	2	9	26	24	15
P4	3	5	17	14	9

Turnaround Time = Completion Time - Arrival Time

Waiting Time = Turnaround Time - Burst Time

Average Waiting Time

Average Waiting Time=0+7+15+9/4 =7.75 units

2. Preemptive SJF (Shortest Remaining Time First - SRTF)

- In Preemptive SJF (SRTF), the CPU can switch to a new process if it arrives with a shorter remaining time than the currently executing process.
- At time 0, P1 is the only process available, so it starts executing.
- At time 1, P2 arrives with a burst time of 4, which is less than the remaining time of P1 (7). So, P1 is preempted, and P2 starts executing.
- P2 completes at time 5 (1 + 4).

- At time 5, P1 (with 7 remaining time), P3 (burst time 9), and P4 (burst time 5) are available. The shortest remaining time is P4 (5), so P4 executes next.
- P4 completes at time 10 (5 + 5).
- At time 10, P1 (remaining time 7) and P3 (burst time 9) are available. P1 has the shorter remaining time, so P1 resumes execution.
- P1 completes at time 17 (10 + 7).
- Finally, P3 executes since it's the only process left and completes at time 26 (17 + 9).

Pre-emptive SJF Execution Order

The order of execution is P1 → P2→ P4→ P1→ P3**.**

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	17	17	9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

Average Waiting Time

Average Waiting Time=9+0+15+24/4=6.5 units

Summary

Algorithm	**Average Waiting Time**
Non-preemptive SJF	7.75 units
Preemptive SJF (SRTF)	6.5 units

In this example, the Preemptive SJF (SRTF) **algorithm provides a lower average waiting time than** Non-preemptive SJF**.**

3. Priority Scheduling

Priority Scheduling is a CPU scheduling algorithm in which each process is assigned a priority, and the CPU is allocated to the process with the highest priority. If two processes have the same priority, they are scheduled according to their arrival order (First-Come, First-Served basis). Priority scheduling can be implemented as either preemptive or non-preemptive.

Types of Priority Scheduling

Preemptive Priority Scheduling:

- If a new process arrives with a higher priority than the currently executing process, the CPU is preempted and allocated to the new process.
- This ensures that high-priority processes are always executed first.

Non-preemptive Priority Scheduling:

- Once a process starts execution, it continues until it finishes.
- The CPU only selects a new process from the ready queue when the current process finishes execution.

Example

Consider the following set of processes with arrival time, burst time, and priority (where a lower priority number means a higher priority):

Process	Arrival Time	Burst Time	Priority
P1	0	10	3
P2	1	1	1
P3	2	2	4
P4	3	1	5

Process Arrival Time Burst Time Priority

P5	4	5	2

1. Non-preemptive Priority Scheduling

- At **time 0**, **P1** arrives and starts executing because it's the only process.
- **P1** completes at time **10**.
- At **time 10**, the remaining processes (P2, P3, P4, and P5) have arrived. The process with the highest priority (lowest priority number) is **P2** (priority 1).
- **P2** completes at time **11** (10 + 1).
- Now at time **11**, the remaining processes are **P3**, **P4**, and **P5**. The process with the next highest priority is **P5** (priority 2).
- **P5** completes at time **16** (11 + 5).
- The remaining processes are **P3** and **P4**, with **P3** having a higher priority.
- **P3** completes at time **18** (16 + 2).
- Finally, **P4** executes and completes at time **19** (18 + 1).

Non-preemptive Priority Scheduling Execution Order

The order of execution is **P1→ P2 →P5→P3→P4**.

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
P1	0	10	3	10	10	0
P2	1	1	1	11	10	9
P3	2	2	4	18	16	14
P4	3	1	5	19	16	15
P5	4	5	2	16	12	7

- **Turnaround Time** = Completion Time - Arrival Time
- **Waiting Time** = Turnaround Time - Burst Time

Average Waiting Time

Average Waiting Time=50+9+14+15+7/5 =9 units

2. Preemptive Priority Scheduling

In preemptive priority scheduling, a new process with a higher priority will preempt the current running process.

- At time 0, P1 is the only process, so it starts executing.
- At time 1, P2 arrives with a higher priority than P1. P1 is preempted, and P2 starts executing.
- P2 completes at time 2.

- P1 resumes execution at time 2 since it has the next highest priority.
- At time 3, P4 arrives but has a lower priority than P1, so P1 continues.
- At time 4, P5 arrives with a higher priority than P1, so P1 is preempted, and P5 starts executing.
- P5 completes at time 9.
- At time 9, P1 resumes and completes at time 17.
- Now only P3 and P4 remain, with P3 having a higher priority, so P3 executes next.
- P3 completes at time 19, and finally, P4 executes and completes at time 20.

Preemptive Priority Scheduling Execution Order

The order of execution is P1→ P2→P1→P5→P1→P3→ P4**.**

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
P1	0	10	3	17	17	7

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
P2	1	1	1	2	1	0
P3	2	2	4	19	17	15
P4	3	1	5	20	17	16
P5	4	5	2	9	5	0

Average Waiting Time

Average Waiting Time=7+0+15+16+0/5=7.6 units

Summary

Algorithm	Average Waiting Time
Non-preemptive Priority	9 units
Preemptive Priority	7.6 units

4. Round Robin (RR)

Round Robin (RR) is a CPU scheduling algorithm designed for time-sharing systems. It allocates the CPU to each process in the ready queue for a fixed time slice or **quantum**, in a cyclic order. If a process does not complete within its time slice, it is moved to the end of the queue, and the next process is scheduled.

Example

Consider the following set of processes with **arrival time** and **burst time**. Assume a **time quantum** of 4 units.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Execution Steps

1. **At time 0**: **P1** starts execution (only process available). It runs for 4 units (quantum size).
    - Remaining burst time for **P1** = 8 - 4 = 4.
    - Time = 4.

2. **At time 4**: **P2**, **P3**, and **P4** have arrived. Next in the queue is **P2**.
    - **P2** executes for 4 units, completing its burst time.
    - Remaining burst time for **P2** = 4 - 4 = 0 (process completed).
    - Time = 8.
3. **At time 8**: Next in the queue is **P3**.
    - **P3** executes for 4 units (quantum size).
    - Remaining burst time for **P3** = 9 - 4 = 5.
    - Time = 12.
4. **At time 12**: Next in the queue is **P4**.
    - **P4** executes for 4 units (quantum size).
    - Remaining burst time for **P4** = 5 - 4 = 1.
    - Time = 16.
5. **At time 16**: The queue cycles back to **P1** (still has 4 units remaining).
    - **P1** executes for 4 units, completing its burst time.
    - Remaining burst time for **P1** = 4 - 4 = 0 (process completed).
    - Time = 20.

6. **At time 20**: Next in the queue is **P3** (still has 5 units remaining).
    - **P3** executes for 4 units (quantum size).
    - Remaining burst time for **P3** = 5 - 4 = 1.
    - Time = 24.
7. **At time 24**: Next in the queue is **P4** (still has 1 unit remaining).
    - **P4** executes for 1 unit, completing its burst time.
    - Time = 25.
8. **At time 25**: Finally, **P3** executes for 1 unit, completing its burst time.
    - Time = 26.

Execution Order

P1→P2→P3→P4→P1→P3→P4→P3

Summary Table

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0	8	20	20	12
P2	1	4	8	7	3
P3	2	9	26	24	15

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P4	3	5	25	22	17

Turnaround Time = **Completion Time - Arrival Time**

Waiting Time = **Turnaround Time - Burst Time**

Average Times

1. **Average Turnaround Time:**

   Average Turnaround Time=20+7+24+22/4=18.25 un its

2. **Average Waiting Time:**

   Average Waiting Time=12+3+15+17/4=11.75 units

Advantages of Round Robin

- Ensures fairness as every process gets an equal share of the CPU.
- Suitable for time-sharing and interactive systems.
- Avoids starvation since no process is left waiting indefinitely.

Disadvantages of Round Robin

- Context switching overhead can be high if the quantum is too small.

- Longer processes may take a long time to complete as they repeatedly cycle through the queue.

2.11 Multiple Processor Scheduling

Multiple-processor scheduling in operating systems refers to the techniques and strategies used to manage the execution of processes on systems with more than one processor (multi-core or multi-CPU systems). Its main objectives are to optimize resource utilization, increase throughput, and ensure fairness.

Key Concepts in Multiple Processor Scheduling

a) Processor Types:

- **Homogeneous Processors:** All processors are identical and can execute any process.
- **Heterogeneous Processors:** Processors have different capabilities, and processes may need to be assigned based on compatibility.

b) Scheduling Models:

- **Symmetric Multiprocessing (SMP):** All processors share the same responsibilities and operate independently.

- **Asymmetric Multiprocessing (AMP):** One processor (the master) manages scheduling (master and slave), while others execute assigned tasks.

c) Processor Affinity:

Tendency to keep a process on the same processor for improved cache performance.

- **Soft Affinity:** Preferred processor, but not mandatory.
- **Hard Affinity:** Strictly bound to a specific processor.

d) Load Balancing:

Ensures even distribution of processes across all processors.

Types:

- **Push Migration:** Overloaded processors push tasks to others.
- **Pull Migration:** Idle processors pull tasks from busy ones.

Scenario

Imagine a data center running an operating system with **four processors**: P1, P2, P3, and P4. The system receives the following jobs:

Job	Burst Time (ms)	Priority
J1	10	High
J2	20	Medium
J3	15	Low
J4	25	High
J5	30	Medium
J6	5	Low

Final Outcome

The jobs are executed across the processors in parallel, reducing the overall completion time for all tasks compared to single-processor scheduling.

Processor	Executed Jobs	Total Time Taken (ms)
P1	J1 → J3 → J6	30
P2	J4	25
P3	J2	20

Processor	Executed Jobs	Total Time Taken (ms)
P4	J5	30

Total Time for Completion: **30 ms**

In single-processor scheduling, the total time would have been **105 ms (sum of all burst times)**.

UNIT-III

Synchronization Tools: The Critical Section Problem, Peterson's Solution, Mutex Locks, Semaphores, Monitors, Classic problems of Synchronization.

Deadlocks: system Model, Deadlock characterization, Methods for handling Deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from Deadlock.

3.1 Synchronization Tools: The Critical Section Problem

Synchronization tools in an operating system are mechanisms used to coordinate the execution of processes or threads, ensuring safe access to shared resources and maintaining data integrity. Below are some of the most commonly used synchronization tools in operating systems:

1. Mutex (Mutual Exclusion Object)

Purpose: Ensures that only one thread or process accesses a critical section at a time.

Key Characteristics:

Binary state: Locked (1) or Unlocked (0).

A process acquires the mutex before entering the critical section and releases it after exiting.

Usage:

Thread synchronization.

Preventing race conditions.

Examples in OS:

POSIX Mutexes, Windows Critical Section.

2. Semaphores

Purpose: Synchronize access to shared resources.

Types:

Binary Semaphore: Similar to a mutex, it can take values 0 or 1.

Counting Semaphore: Keeps track of resource availability with a counter.

Operations:

wait() / P(): Decrements the semaphore value.

signal() / V(): Increments the semaphore value.

Examples:

Used in producer-consumer problems.

Managing multiple readers and writers.

3. Condition Variables

Purpose: Allow threads to wait for certain conditions to be true before proceeding.

Operations:

wait(): Releases the lock and puts the thread to sleep.

signal(): Wakes one waiting thread.

broadcast(): Wakes all waiting threads.

Examples:

Used with monitors for thread communication.

Often paired with mutexes.

4. Read-Write Locks

Purpose: Allow multiple threads to read simultaneously but ensure exclusive access for writing.

Advantages:

Improves performance in scenarios with frequent reads and rare writes.

Use Cases:

Managing databases or file systems.

5. Message Passing

Purpose: Synchronization through communication rather than shared memory.

Mechanisms:

Message queues.

Pipes.

Signals.

Advantages:

Avoids race conditions by not sharing memory directly.

Suitable for distributed systems.

6. Software Algorithms

Used when hardware support is unavailable:

Peterson's Algorithm: Ensures mutual exclusion between two processes.

Dekker's Algorithm: Handles synchronization between two processes.

Lamport's Bakery Algorithm: Manages multiple processes.

Conclusion

Synchronization tools are crucial for maintaining consistency and preventing race conditions in multi-threaded and multi-process systems. The choice of tool depends on the application's requirements, such as performance, fairness, and complexity.

3.2 The Critical Section Problem

Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

The critical section cannot be executed by more than one process at the same time; operating system faces the

difficulties in allowing and disallowing the processes from entering the critical section.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Key Issues in the Critical Section Problem

When multiple processes execute concurrently, the following issues can arise if access to the critical section is not managed properly:

Race Conditions:

Occur when the outcome of a process depends on the timing or sequence of other processes accessing shared resources.

Data Corruption:

Without proper synchronization, conflicting operations on shared resources can lead to inconsistencies.

Deadlocks:

Processes may block each other indefinitely if not managed correctly.

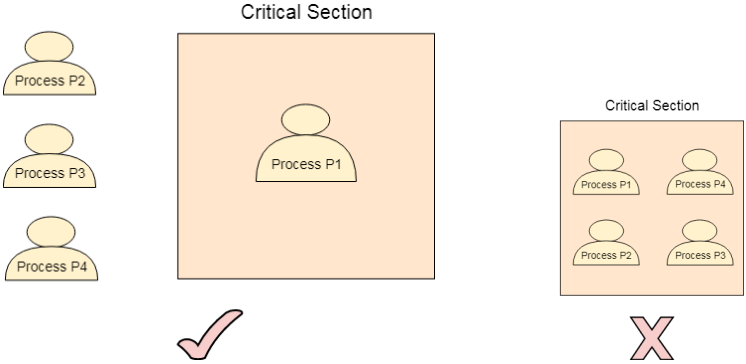
In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

Requirements of Synchronization mechanisms

Primary

a) Mutual Exclusion

Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.



b) Progress

Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Secondary

a) Bounded Waiting

A process won't wait indefinitely to enter the critical section.

.

b) Architectural Neutrality

Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

3.3 Peterson's Solution

Peterson's Solution is a classical algorithm used in operating systems to address the Critical Section Problem. It is a software-based solution designed to synchronize two processes that share a resource, ensuring mutual exclusion, progress, and bounded waiting without requiring special hardware support.

Key Features of Peterson's Solution

Mutual Exclusion: Ensures that only one process can enter the critical section at a time.

Progress: If no process is in the critical section, one of the waiting processes must be allowed to enter.

Bounded Waiting: A process won't wait indefinitely to enter the critical section.

Two-Process Limitation: Peterson's solution works for two processes. Extensions exist for more processes but become less practical.

How Peterson's Solution Works

It uses two shared variables:

flag[2]:

- Indicates whether a process wants to enter the critical section.
- flag[0] corresponds to process P0, and flag[1] corresponds to process P1.

turn:

Indicates whose turn it is to enter the critical section if both processes want to enter.

Algorithm for Two Processes

Let's assume two processes, P0 and P1, need to access a critical section.

Shared Variables:

```
int flag[2] = {0, 0}; // Indicates interest in critical section
int turn = 0;         // Determines whose turn it is
```

Process P0 Code:

```
flag[0] = 1;          // Indicate P0 wants to enter
turn = 1;             // Allow P1 the opportunity to enter
while (flag[1] == 1 && turn == 1); // Wait if P1 wants to enter and it's P1's turn
```

// Critical Section

...

flag[0] = 0; // Indicate P0 is done

Process P1 Code:

flag[1] = 1; // Indicate P1 wants to enter

turn = 0; // Allow P0 the opportunity to enter

while (flag[0] == 1 && turn == 0); // Wait if P0 wants to enter and it's P0's turn

// Critical Section

...

flag[1] = 0; // Indicate P1 is done

Visualization

Scenario 1: Both Processes Want to Enter

P0 sets flag[0] = 1 and turn = 1.

P1 sets flag[1] = 1 and turn = 0.

Both processes check the while condition:

P0 waits because flag[1] == 1 and turn == 1.

P1 enters the critical section because flag[0] == 1 but turn != 1.

After P1 exits, it sets flag[1] = 0, allowing P0 to proceed.

Scenario 2: Only One Process Wants to Enter

If P0 is the only process that wants to enter, it proceeds without waiting because flag[1] == 0

Advantages

- Simple and easy to understand.
- Does not require any special hardware support (purely software-based).
- Satisfies all three conditions of the critical section problem.

Disadvantages

Limited to Two Processes:

- The original algorithm is designed for only two processes. Extending it to multiple processes is complex and inefficient.

Busy Waiting:

- Processes spin in a loop while waiting, wasting CPU cycles.

Not Suitable for Modern Multicore Architectures:

- Memory consistency issues can arise in modern systems where caches are used.

Example Usage

Peterson's solution is mostly a teaching tool in operating systems. Practical implementations often use advanced

synchronization primitives like semaphores, mutexes, or hardware support (e.g., atomic instructions).

Conclusion

Peterson's Solution provides a simple, elegant way to understand and solve the critical section problem for two processes. While not widely used in modern systems due to its limitations, it is a foundational algorithm in the study of process synchronization.

3.4 Mutex Locks

A Mutex (Mutual Exclusion) Lock is a synchronization mechanism used in operating systems to protect critical sections, ensuring that only one thread or process can access shared resources at a time. It helps prevent race conditions and ensures mutual exclusion.

Key Features of Mutex Locks

a) Mutual Exclusion:

Ensures only one thread or process accesses the critical section at any time.

Lock and Unlock Mechanism:

A thread must acquire the lock before entering the critical section and release it after exiting.

c) Blocking:

If a thread attempts to acquire a lock that is already held, it is put in a waiting state until the lock becomes available.

d) No Busy Waiting:

Unlike simple spinlocks, mutex locks avoid busy waiting by putting threads in a waiting queue.

How Mutex Locks Work

Steps:

1) A thread requests the mutex lock before entering the critical section.
2) If the lock is free, the thread acquires it and enters the critical section.
3) If the lock is already held by another thread, the requesting thread waits until the lock is released.
4) Once the thread completes its task, it releases the lock, allowing other threads to proceed.

Mutex Lock Operations

Lock (acquire):

A thread/process requests the mutex. If the lock is available, it acquires it; otherwise, it waits.

The definition of acquire() is as follows:

```
acquire() {
while (!available)
```

; /* busy wait */

available = false;

}

Unlock (release):

The thread/process releases the mutex after completing its operation, making it available for others.

The definition of release() is as follows:

release() {

available = true;

}

Trylock (optional):

Attempts to acquire the lock without blocking. If the lock is unavailable, it returns immediately.

Advantages of Mutex Locks

Simple and Easy to Use:

- Provides a straightforward mechanism for ensuring mutual exclusion.

No Busy Waiting:

- Threads are blocked and do not consume CPU cycles while waiting.

Portable:

- Supported by most operating systems and threading libraries.

Disadvantages of Mutex Locks

Deadlocks:

- Improper use can result in deadlocks if multiple threads block each other indefinitely.

Priority Inversion:

- A higher-priority thread may wait indefinitely if a lower-priority thread holds the mutex.

Overhead:

- Blocking and unblocking threads involve overhead compared to spinlocks.

Applications of Mutex Locks

Multithreaded Programs:

- Used to synchronize threads accessing shared resources.

Operating Systems:

- Protects shared data structures in the kernel.

Database Systems:

- Ensures data consistency during transactions.

Conclusion

Mutex locks are a fundamental synchronization tool in operating systems and multithreaded programming. They provide a simple and effective way to prevent race conditions, ensuring the correct execution of critical sections

while avoiding busy waiting. Proper use of mutex locks is essential to avoid pitfalls like deadlocks and priority inversion.

3.5 Semaphores

A semaphore is a synchronization mechanism used in operating systems to control access to shared resources by multiple processes or threads. Semaphores help prevent race conditions, ensure mutual exclusion, and manage resource allocation in concurrent systems.

Types of Semaphores

a) Binary Semaphore:

- Can have only two values: 0 or 1.
- Acts like a mutex lock: 0 indicates the resource is locked, and 1 indicates it is available.

b) Counting Semaphore:

- Can have a non-negative integer value.
- Used to manage a limited number of resources. The value of the semaphore represents the number of available resources.

Semaphores Operations

Semaphores support two atomic operations:

a) Wait (P or Down):

- Decrements the semaphore value.
- If the value becomes less than zero, the process/thread is put to sleep (blocked).
- The definition of wait() is as follows:

```
wait(S)
 {
    while (S <= 0)
     ; // busy wait
      S--;
  }
```

b) Signal (V or Up):

- Increments the semaphore value.
- If there are processes/threads waiting, one is woken up to proceed.
- The definition of signal() is as follows:

```
signal(S)
{
S++;
```

}

Semaphore Pseudocode

Initialization:

Semaphore S = initial_value; // Semaphore initialized with a value

Wait Operation:

```
wait(S):
   S = S - 1;
   if (S < 0)
      block();  // Put the process to sleep
```

Signal Operation:

```
signal(S):
   S = S + 1;
   if (S <= 0)
      wakeup();  // Wake up a blocked process
```

Advantages of Semaphores

Supports Multiple Resources:

- Can manage more than one resource (via counting semaphore).

Synchronization:

- Effectively synchronizes processes or threads.

Flexibility:

- Can be used to implement different synchronization primitives like mut-exes or barriers.

Disadvantages of Semaphores

Complexity:

- Difficult to use correctly in large systems, leading to issues like deadlocks.

Deadlocks:

- Improper usage can cause processes to block indefinitely.

Priority Inversion:

- A high-priority thread can be blocked by a lower-priority thread holding the semaphore.

Applications of Semaphores

Producer-Consumer Problem:

- Synchronizing production and consumption of shared resources.

Reader-Writer Problem:

- Managing access to shared data for multiple readers and writers.

Operating Systems:

- Used in managing shared resources like CPU scheduling, memory management, and disk access.

Conclusion

Semaphores are a powerful tool for process and thread synchronization. They provide a robust mechanism for ensuring mutual exclusion and resource allocation. However, their complexity requires careful implementation to avoid synchronization issues like deadlocks or starvation.

3.6 Monitors

A monitor is a high-level synchronization construct used in operating systems and programming languages to manage access to shared resources by multiple threads or processes. Monitors encapsulate shared data, procedures to manipulate that data, and the synchronization mechanisms required to ensure mutual exclusion and condition synchronization.

Key Features of Monitors

Encapsulation:

- Monitors encapsulate shared data and operations (methods) that manipulate that data.
- Threads can access the shared resource only through the monitor's methods, ensuring controlled access.

Mutual Exclusion:

- Only one thread can execute a monitor's method at a time.
- This is achieved by associating a lock with the monitor. A thread must acquire the lock before executing any method within the monitor.

Condition Variables:

- Monitors include condition variables that allow threads to wait for certain conditions to be met.

Operations:

- **Wait():** A thread releases the monitor lock and waits for a signal.
- **Signal():** Wakes up one waiting thread.
- **Broadcast():** Wakes up all waiting threads.

Implicit Locking:

Monitors automatically handle locking and unlocking mechanisms, reducing the burden on the programmer and minimizing errors.

How Monitors Work:

- When a thread enters a monitor, it acquires the associated lock.
- Other threads attempting to access the monitor are blocked until the lock is released.
- A thread that completes its operation or calls Wait() releases the lock, allowing other threads to acquire it.

Advantages of Monitors:

- Simplifies synchronization in concurrent programming.
- Reduces the risk of errors like deadlocks and race conditions by centralizing resource management.

Limitations of Monitors:

Limited Flexibility:

- Monitors may not work well in situations requiring fine-grained control over locking mechanisms.

Potential for Deadlocks:

- Improper use of condition variables and nested monitors can lead to deadlocks.

Performance Overhead:

- Synchronization mechanisms in monitors can introduce latency in highly concurrent systems.

Monitor vs. Semaphore:

Aspect	Monitor	Semaphore
High-Level Abstraction	Yes	No
Encapsulation	Encapsulates data and methods	No encapsulation; operates with counters
Condition Variables	Built-in	Requires explicit implementation
Mutual Exclusion	Automatic	Must be manually implemented

3.7 Classic problems of Synchronization

Synchronization problems in operating systems arise when multiple processes or threads need to access shared resources without conflicts. Classic synchronization problems serve as fundamental examples for studying and solving concurrency issues. Here are the most well-known problems:

1) The Producer-Consumer Problem

2) The Readers-Writers Problem

3) The Dining Philosophers Problem

4) The Sleeping Barber Problem

5) The Cigarette Smokers Problem

1. The Producer-Consumer Problem

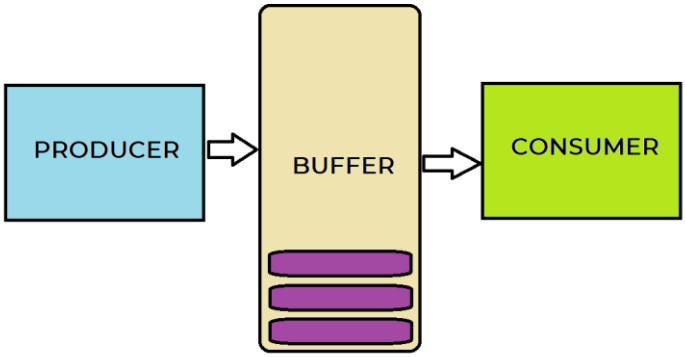
The producer-consumer problem is a well-known synchronization issue in operating systems that involves two types of processes: producers and consumers.

Producer

Creates data and puts it into a shared buffer

Consumer

Takes data out of the shared buffer and uses it



Scenario:

- A producer thread produces data items and places them in a bounded buffer.
- A consumer thread consumes items from the buffer.
- The producer must wait if the buffer is full, and the consumer must wait if the buffer is empty.

Key Challenges:

- Avoid race conditions when accessing the buffer.
- Ensure proper signaling between producer and consumer.

Solution:

- Use semaphores for mutual exclusion (mutex) and signaling (full and empty semaphores).

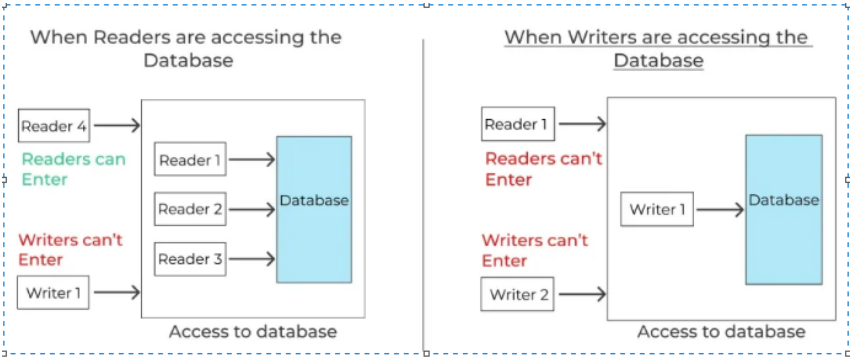
- Alternatively, monitors or condition variables can be used.

2) The Readers-Writers Problem

The readers-writers problem is a common problem in computer science and operating systems that occurs when multiple processes share access to a resource, but only one process can write to it at a time:

Readers: Processes that only want to read data from the shared resource

Writers: Processes that want to write data into the shared resource.



Scenario:

- Multiple readers and writers access a shared resource (e.g., a database).

- Readers can read simultaneously, but a writer needs exclusive access.

Key Variants:

- First Readers-Writers Problem: No reader is kept waiting unless a writer has acquired the resource.
- Second Readers-Writers Problem: No writer is kept waiting for a reader to release the resource.

Key Challenges:

- Allow multiple readers but ensure mutual exclusion for writers.
- Prevent starvation of either readers or writers.

Solution:

- Use semaphores or locks to synchronize readers and writers.
- Prioritize readers or writers to avoid starvation.

3) The Dining Philosophers Problem

The dining philosophers problem is a classic synchronization problem in operating systems that illustrates how to resolve issues that arise when multiple processes interact:

Problem

Five philosophers sit around a table and take turns eating and thinking. Each philosopher needs both their left and right chopsticks to eat, and can only eat if both are available. If a philosopher can't eat, they put their forks down and continue thinking.

Scenario:

- Five philosophers alternately think and eat.
- They share a circular table with five forks (one fork between each pair of philosophers).
- A philosopher needs both forks adjacent to them to eat.

Key Challenges:

- Prevent deadlock (e.g., if every philosopher picks up one fork and waits for the other).
- Avoid starvation (some philosophers never get to eat).

Solution:

- Introduce rules to reduce contention:
- Limit the number of philosophers who can pick up forks at the same time.
- Use a waiter process to manage fork allocation.
- Ensure that a philosopher picks up both forks simultaneously or none.

4) The Sleeping Barber Problem

The sleeping barber problem is a classic computer science example of how multiple operating system processes can cause synchronization issues. It was first proposed in 1965 by Edsger Dijkstra, a computer science pioneer.

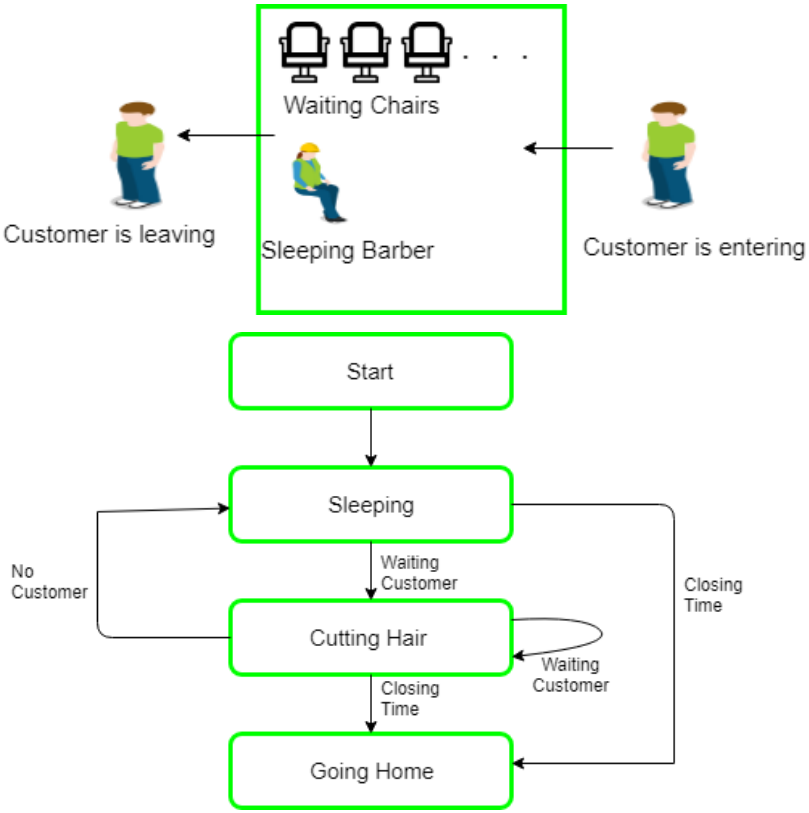
The problem is based on a barbershop with a barber and a waiting room:

No customers: The barber sleeps in the barber chair.

Customer arrives: The customer wakes up the barber to get a haircut.

Multiple customers: If there are other customers waiting, they sit in the waiting room if there are empty seats, or they leave if there are no empty seats.

The sleeping barber problem is a demonstration of inter-process communication and synchronization challenges in concurrent systems.



Scenario:

- A barber sleeps if no customers are present.
- When a customer arrives, they either wake the barber if sleeping or wait in a waiting room if the barber is busy.
- If the waiting room is full, the customer leaves.

Key Challenges:

- Synchronize the barber and customer threads.
- Manage the limited number of chairs in the waiting room.

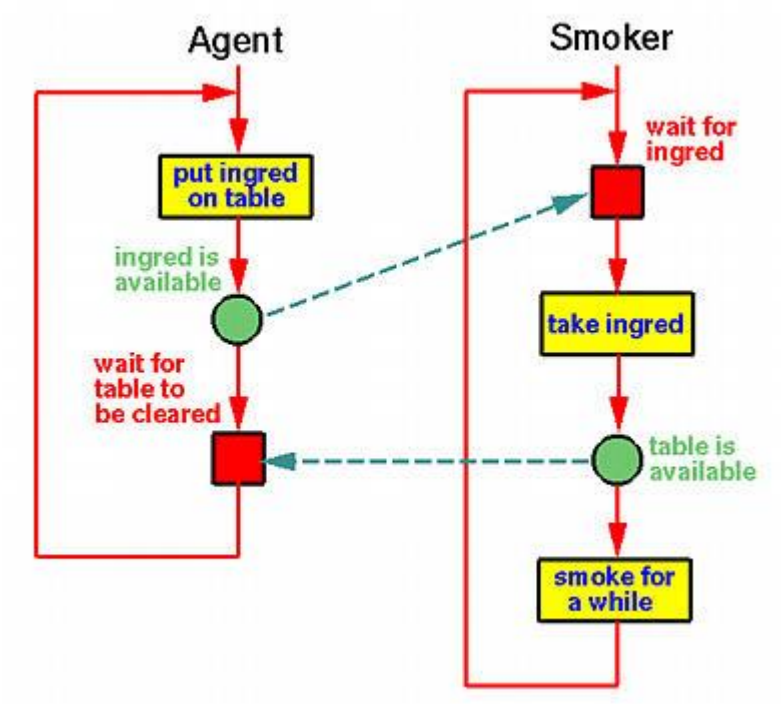
Solution:

- Use semaphores to coordinate sleeping, waking, and chair availability.
- Use mutex locks to ensure proper access to shared resources.

5) The Cigarette Smokers Problem

Suppose a cigarette requires three ingredients, tobacco, paper and match. There are three chain smokers. Each of them has only one ingredient with infinite supply. There is an agent who has infinite supply of all three ingredients. To make a cigarette, the smoker has tobacco (resp., paper and match) must have the other two ingredients paper and match

(resp., tobacco and match, and tobacco and paper). The agent and smokers share a table. The agent randomly generates two ingredients and notifies the smoker who needs these two ingredients. Once the ingredients are taken from the table, the agent supplies another two. On the other hand, each smoker waits for the agent's notification. Once it is notified, the smoker picks up the ingredients, makes a cigarette, smokes for a while, and goes back to the table waiting for his next ingredients.



Scenario:

- Three smokers need three ingredients (tobacco, paper, and matches) to make a cigarette.
- An agent places two random ingredients on a table, and the smoker with the third ingredient takes them.

Key Challenges:

- Ensure that only the smoker with the missing ingredient acts.
- Avoid deadlock and ensure fairness.

Solution:

- Use semaphores to signal which smoker should act.
- Use mutex locks to ensure mutual exclusion for the table.

Common Techniques to Solve Synchronization Problems:

Semaphores:

- Use counting and binary semaphores to manage resource access and signaling.

Monitors:

- Encapsulate shared resources and synchronization mechanisms in high-level constructs.

Locks:

- Use mutexes or spinlocks to enforce mutual exclusion.

Condition Variables:

- Facilitate signaling and waiting for specific conditions.

Deadlock Prevention Techniques:

- Resource hierarchy, timeout mechanisms, or avoiding circular waits.
- These problems are central to understanding and addressing synchronization issues in modern operating systems and multithreaded applications.

3.8 Deadlocks: system Model

A deadlock in an operating system occurs when a group of processes becomes stuck in a state where none of them can proceed because each process is waiting for a resource held by another process. Deadlocks are a fundamental issue in concurrent programming and resource management, as they can cause a system to hang indefinitely.

Conditions for Deadlock

For a deadlock to occur, the following four conditions must hold simultaneously:

Mutual Exclusion:

At least one resource must be held in a non-shareable mode. Only one process can use the resource at a time.

Hold and Wait:

A process holding at least one resource is waiting to acquire additional resources that are currently held by other processes.

No Preemption:

Resources cannot be forcibly removed from a process holding them; the process must release them voluntarily.

Circular Wait:

A set of processes exists where each process is waiting for a resource held by the next process in the set, forming a circular chain.

If any one of these conditions is broken, a deadlock cannot occur.

Summary Table

Aspect	Explanation
Definition	A state where processes are stuck waiting for resources held by one another.
Conditions	Mutual Exclusion, Hold and Wait,

Aspect	Explanation
	No Preemption, Circular Wait
Handling Strategies	Prevention, Avoidance, Detection & Recovery, Ignorance
Key Tools	Resource Allocation Graph, Banker's Algorithm
Examples	Resource Allocation Deadlock, Dining Philosophers Problem, Producer-Consumer Deadlock

System Model in Operating Systems

The system model in operating systems provides a framework to understand how resources are allocated to processes and how these allocations affect system performance and behavior. It helps in modeling and analyzing situations such as deadlocks, resource scheduling, and process synchronization.

Key Components of the System Model

Processes:

- Independent entities (programs in execution) that require resources to execute tasks.
- Processes may request, hold, and release resources during their lifecycle.

Resources:

- Entities required by processes to perform operations.
- Resources can be hardware (e.g., CPU, memory, I/O devices) or software (e.g., files, locks).

Classified into:

- **Reusable Resources:** Can be used by one process at a time and returned (e.g., CPU, memory, printers).
- **Consumable Resources:** Can be produced and consumed, disappearing after use (e.g., messages, signals).

States of Resources:

- Resources can exist in one of the following states:

  Free: Not allocated to any process.

  Allocated: Currently in use by a process.

  Requested: A process is waiting for the resource.

Resource Types and Instances:

- Each resource type may have one or more instances (e.g., a printer type may have three physical printers).

System Representation:

- The system can be represented using a Resource Allocation Graph (RAG) or a matrix-based approach for analysis and management.

Operations in the System Model

Resource Request:

A process requests a resource. If the resource is available, it is allocated; otherwise, the process is put in a waiting state.

Resource Allocation:

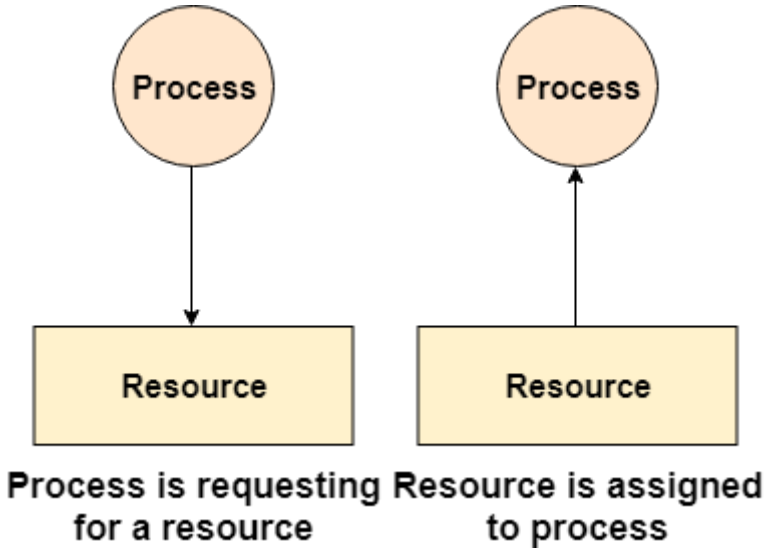
The resource is assigned to the requesting process.

Resource Release:

The process releases the resource after its operation is complete, making it available to other processes.

Resource Allocation Graph (RAG)

The Resource Allocation Graph, also known as RAG is a graphical representation of the state of a system. It has all the information about the resource allocation to each process and the request of each process.

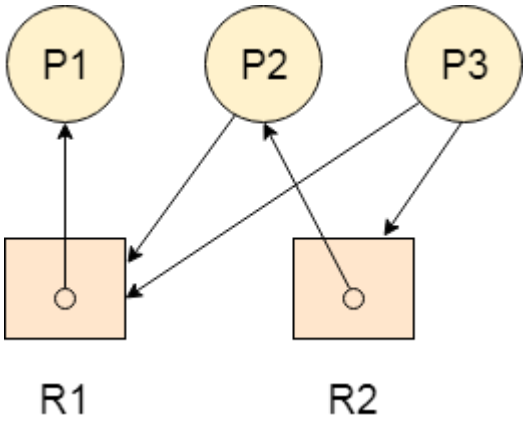


Example

Let'sconsider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.



System States in the Model

Safe State:

The system can allocate resources to all processes in some order without causing a deadlock.

Defined by algorithms like the Banker's Algorithm.

Unsafe State:

The system may not be able to allocate resources safely to all processes.

Unsafe states may lead to a deadlock but do not guarantee one.

Deadlocked State:

Processes are waiting for resources in a circular chain, and no progress is possible.

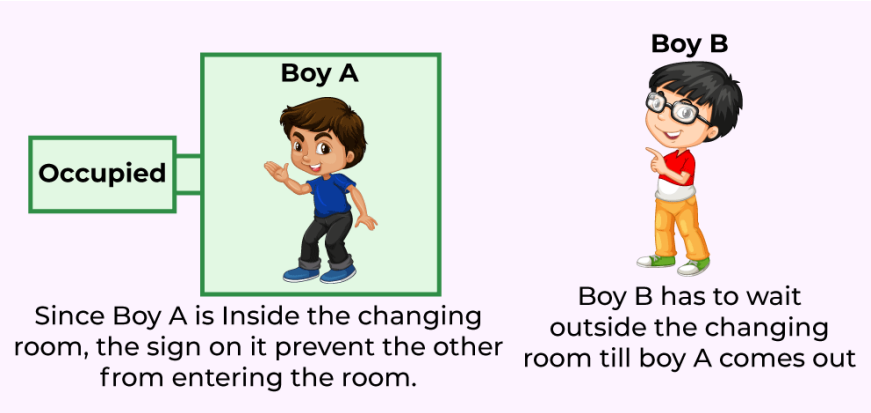
3.9 Deadlock characterization

A deadlock is a condition in a multi-process system where a group of processes is stuck because each process is waiting for a resource held by another process in the group. Deadlock characterization explains the conditions that lead to deadlock and provides the basis for its detection, prevention, and resolution.

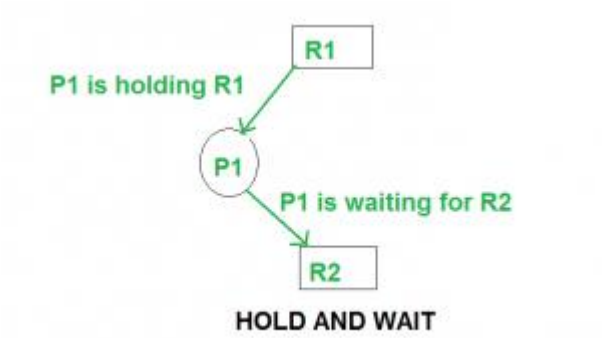
Necessary Conditions for Deadlock

Four conditions must be met simultaneously for a deadlock to occur:

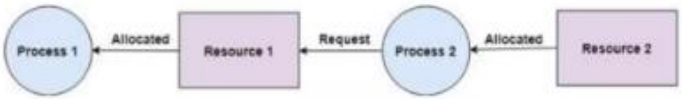
Mutual exclusion: Only one process can use a resource at a time.



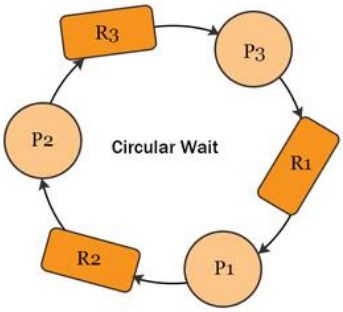
Hold and wait: A process is holding at least one resource and waiting for other resources.



No preemption: A process can't use a resource until it is released by other processes.



Circular wait: A set of processes are waiting for each other in a circular manner.



P1 is waiting for P2 to release R2 , P2 is waiting for P3 to release R3 and P3 is waiting for P1 to release R1

Resource Allocation Graph (RAG) and Deadlocks

- A Resource Allocation Graph (RAG) is a directed graph representing processes and resource relationships.

Vertices:

Processes (circles) and resources (rectangles).

Edges:

- **Request Edge:** From a process to a resource (indicating the process is waiting for the resource).
- **Assignment Edge:** From a resource to a process (indicating the resource is allocated to the process).

Cycle Detection in RAG:

- If there is no cycle, there is no deadlock.

If a cycle exists:

- If each resource has only one instance, a deadlock is guaranteed.
- If resources have multiple instances, further analysis is required to confirm deadlock.

Deadlock Example

Scenario:

Processes: P1,P2

Resources: R1,R2

State:

P1 holds R1 and requests R2.

P2 holds R2 and requests R1.

RAG Representation:

P1→R2→P2→R1→P1

Analysis:

A cycle exists, and since each resource has one instance, the system is in a deadlock state.

3.10 Methods for handling Deadlocks

Handling deadlocks in a system is crucial to maintain the system's stability and prevent performance issues. Here are several methods to handle deadlocks:

1. Prevention

Mutual Exclusion: Make sure that at least one resource is not shared among processes. This prevents circular wait by

ensuring that at least one process does not hold resources and wait for others.

Hold and Wait: This method breaks the hold-and-wait condition by forcing processes to release all their resources before requesting any new ones.

No Preemption: Preemption can be used to forcibly take back resources from processes. However, this approach can lead to additional complexity and should be used with caution.

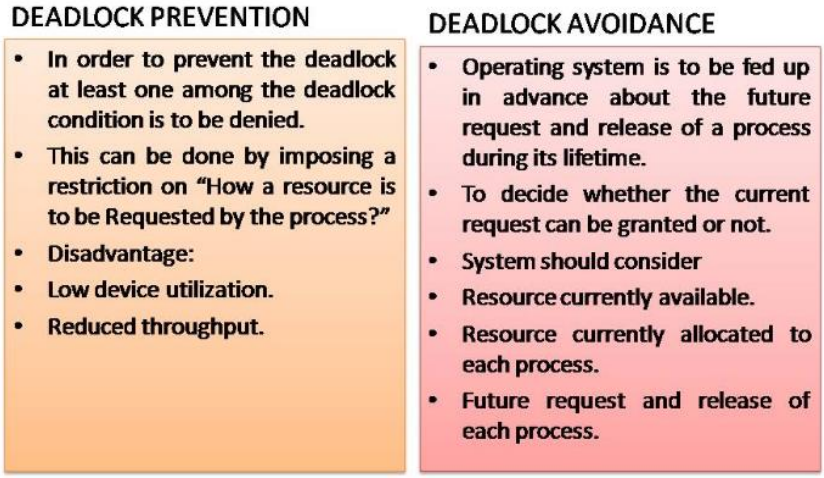
Circular Wait: A simple technique is to require that all processes request resources in a predefined order, which prevents circular wait by ensuring that processes can only request resources in a specific sequence.

2. Avoidance

Resource Allocation Graph: By maintaining a resource allocation graph, the system can determine if a new resource allocation would lead to a deadlock. The Banker's Algorithm is a classic method for deadlock avoidance.

Safe State: Only grant a request if it does not lead to an unsafe state (where the system might enter a deadlock). This approach requires that the system maintains information

about the resources currently allocated and the potential future requests of all processes.



3. Detection and Recovery

Detection: By periodically checking for deadlocks in the system using algorithms like Dijkstra's or Raghavan and Singhal's algorithm, the system can identify a deadlock cycle.

Recovery: Once a deadlock is detected, the system can resolve it by aborting one or more processes or by rolling back their execution to a safe state.

4. Deadlock Timeout

Timeouts: The system can be configured to abort processes that wait for a resource for too long, thereby preventing

deadlocks. This approach does not guarantee deadlock resolution but can alleviate the system's resource contention issues.

5. Resource Preallocation

Preallocate Resources: By pre-allocating the maximum required resources to a process upfront, the system reduces the chances of deadlock as processes do not need to request additional resources once they start executing.

Each method has its trade-offs and is often used in combination to effectively handle deadlocks in a system.

UNIT-IV

Memory-Management Strategies: Introduction, Contiguous memory allocation, Paging, Structure of the Page Table, Swapping.

Virtual Memory Management: Introduction, Demand paging, Copy-on-write, Page replacement, Allocation of frames, Thrashing.

Storage Management: Overview of Mass Storage Structure, HDD Scheduling, RAID.

4.1 Memory-Management Strategies: Introduction

Memory management is a critical function of an operating system (OS) that involves managing the computer's memory hierarchy to optimize system performance. Memory serves as a bridge between the CPU and storage, allowing programs and data to be accessed efficiently. Effective memory-management strategies ensure optimal utilization of the available memory resources, supporting multitasking, process isolation, and efficient execution.

Key Objectives of Memory Management

Efficient Resource Utilization: Ensure memory is allocated and utilized effectively without wastage.

Process Isolation: Protect processes from interfering with each other's memory space.

Multiprogramming Support: Allow multiple processes to execute simultaneously by sharing memory resources.

Minimized Latency: Reduce the time spent in accessing and allocating memory.

Security and Protection: Safeguard memory against unauthorized access or malicious attacks.

Components of Memory Management

Logical and Physical Addressing

- Logical addresses are generated by the CPU and are independent of the physical memory.
- Physical addresses refer to actual locations in the memory hardware.
- The Memory Management Unit (MMU) translates logical addresses to physical addresses.

Memory Allocation

- Allocating memory to processes dynamically or statically during their execution.
- Strategies include contiguous and non-contiguous memory allocation.

Memory Hierarchy

- Memory consists of layers: registers, cache, main memory (RAM), and secondary storage (disk).

- Memory management optimizes data flow between these layers to enhance performance.

Fragmentation Management

- Fragmentation (internal or external) occurs when memory is inefficiently utilized.
- Memory-management strategies aim to reduce fragmentation.

Types of Memory-Management Strategies

a) Contiguous Memory Allocation

- Memory is allocated in a single contiguous block for each process.

Includes:

- **Fixed Partitioning:** Divides memory into fixed-size partitions.
- **Variable Partitioning:** Allocates memory dynamically to fit process sizes.

b) Paging

- Divides memory into fixed-size blocks called pages.
- Logical memory is divided into pages, and physical memory into frames.

- Eliminates external fragmentation but introduces page table overhead.

c) Segmentation

- Divides memory into segments based on logical divisions (e.g., code, data, stack).
- Segments vary in size, improving logical organization.

d) Virtual Memory

- Enables execution of processes larger than physical memory by using disk storage as an extension of RAM.
- Uses paging or segmentation to manage the virtual address space.

e) Swapping

Temporarily moves processes between main memory and secondary storage to manage limited memory.

Importance of Memory-Management Strategies

System Stability: Ensures that processes do not interfere with one another, preventing crashes.

Enhanced Performance: Optimizes memory usage to support multitasking and quick process execution.

Scalability: Supports increasing demands on system resources as workloads grow.

User Experience: Minimizes delays, making applications and the system more responsive.

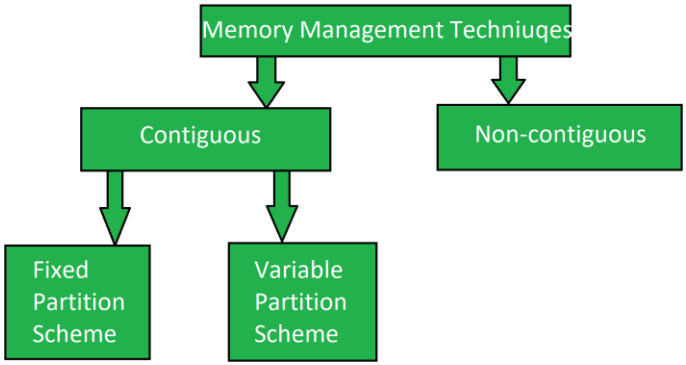
By employing appropriate memory-management strategies, operating systems maintain balance between resource allocation, protection, and system efficiency.

4.2 Contiguous memory allocation

Contiguous memory allocation is a memory management technique used in operating systems to allocate memory in a single continuous block. This method is one of the simplest and oldest ways to manage memory, and it ensures that a process is assigned a single contiguous block of memory for execution.

Continuous Memory Management Techniques. Memory Management Techniques are classified broadly into two categories:

- Contiguous
- Non-contiguous



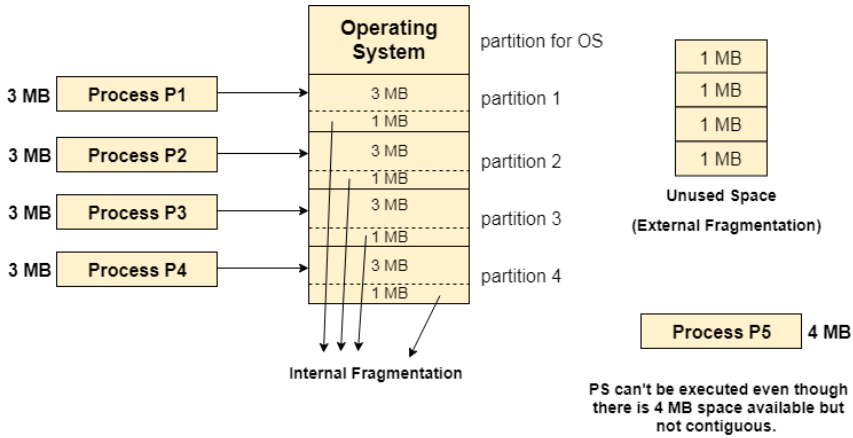
Contiguous Memory Management Techniques

Below are two Contiguous Memory Management Techniques.

1. Fixed Partition Scheme
2. Variable Partition Scheme

1. Fixed Partition Scheme

The **Fixed Partition Scheme** is a memory management technique in contiguous memory allocation where the memory is divided into fixed-sized partitions at the time of system initialization. Each partition can hold exactly one process, and the size of partitions is predetermined and remains constant throughout system operation.



Fixed Partitioning

(Contiguous memory allocation)

Advantages:

Simple Implementation:

The simplicity of design and implementation makes it easier to manage memory allocation and deallocation.

Efficient for Small Systems:

Ideal for systems with predictable workloads where processes have relatively uniform memory requirements.

Fast Allocation:

Allocation decisions are quick since partitions are pre-defined.

Disadvantages:

Internal Fragmentation:

Occurs when the process size is smaller than the allocated partition, leading to wasted memory within the partition.

External Fragmentation:

If no partition is large enough to accommodate a process, it results in wasted space, even if the total free memory is sufficient.

Inflexibility:

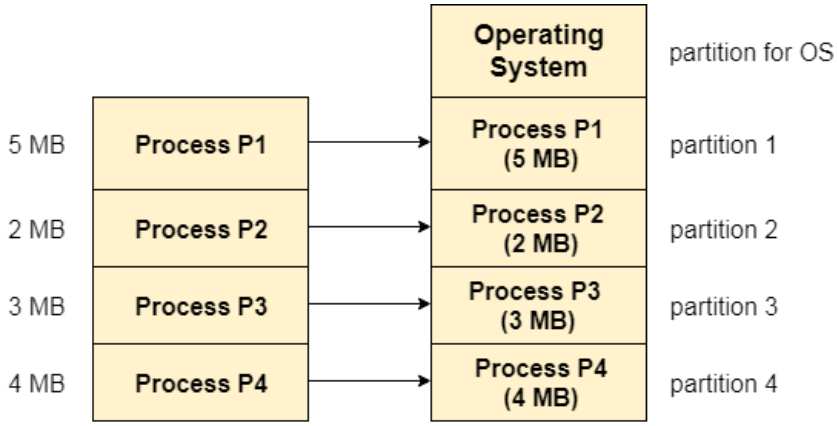
Predefined partition sizes cannot adapt to processes with highly variable memory demands, reducing overall memory utilization.

Fixed Number of Processes:

The number of processes that can be accommodated at a time is limited by the number of partitions, regardless of whether free memory is available.

2. Variable Partition Scheme

The Variable Partition Scheme is a memory management technique where memory is allocated dynamically based on the size of the process. Unlike the fixed partition scheme, memory is not divided into fixed-size blocks; instead, partitions are created as needed, and their sizes are tailored to the memory requirements of processes.



Dynamic Partitioning

(Process Size = Partition Size)

Advantages:

Reduced Internal Fragmentation:

Memory is allocated exactly as needed, leaving little to no wasted space within partitions.

Better Utilization of Memory:

The scheme adapts to process requirements, making better use of available memory.

No Predefined Limits:

There is no fixed limit on the number of processes, as partitions are created dynamically.

Disadvantages:

External Fragmentation:

Over time, free memory becomes fragmented into small, non-contiguous blocks, making it difficult to allocate memory to large processes.

Compaction Overhead:

To address external fragmentation, the operating system may need to periodically perform memory compaction, which is resource-intensive.

Allocation Overhead:

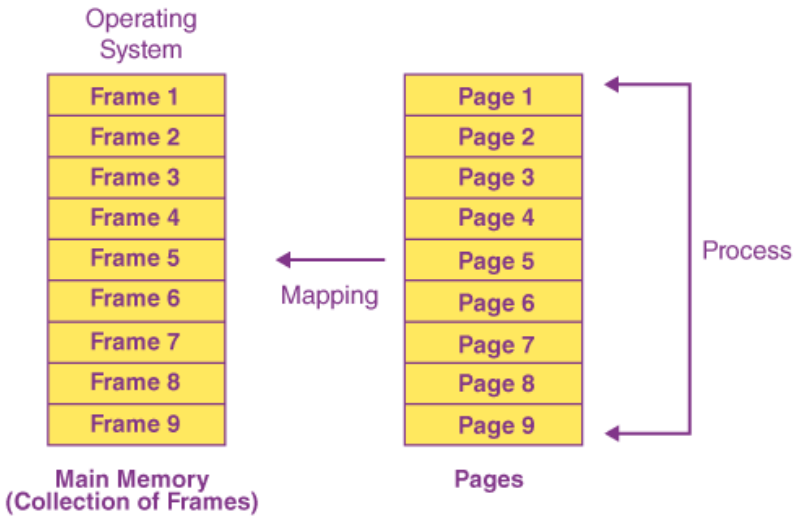
Finding a suitable free memory block for a process can take time, especially as fragmentation increases.

Comparison:

Aspect	Fixed Partitioning	Variable Partitioning
Partition Size	Fixed and pre-defined	Dynamic and variable
Fragmentation	Internal fragmentation	External fragmentation
Flexibility	Inflexible	Flexible
Efficiency	Lower memory utilization	Higher memory utilization
Complexity	Simple implementation	More complex to manage

4.3 Paging

Paging is a memory management technique in operating systems that eliminates the need for contiguous memory allocation and minimizes fragmentation by dividing both the physical memory and a process's logical memory into fixed-size blocks. These blocks are called pages (in logical memory) and frames (in physical memory).



Key Components:

Page:

A fixed-size block of logical memory.

Frame:

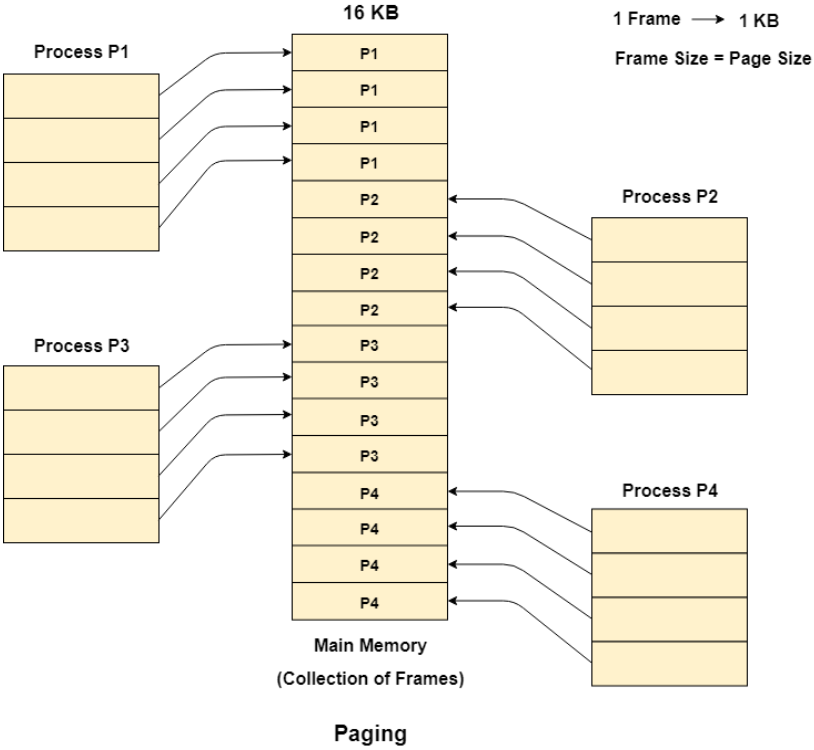
- A fixed-size block of physical memory, matching the page size.

Page Table:

- Maintains the mapping between page numbers (logical addresses) and frame numbers (physical addresses).
- Each process has its own page table.

Logical and Physical Address:

- **Logical Address:** An address generated by the CPU, consisting of a page number and an offset within the page.
- **Physical Address:** The actual location in physical memory, calculated using the frame number and offset.



Paging

Example

A process has a logical memory size of 8 KB, and the physical memory size is 16 KB. The page size is 1 KB.

Given:

Logical memory = 8 KB

Physical memory = 16 KB

Page size = 1 KB

4. Logical memory pages = $\frac{\text{Logical Memory Size}}{\text{Page Size}} = \frac{8}{1} = 8$ pages.
5. Physical memory frames = $\frac{\text{Physical Memory Size}}{\text{Page Size}} = \frac{16}{1} = 16$ frames.

Logical to Physical Mapping:

Suppose the operating system assigns the process's pages to the following physical frames:

Page Number	Frame Number
0	5
1	8
2	2
3	12
4	7
5	10

Page Number	Frame Number
6	3
7	14

Logical Address:

A logical address is represented as:

- **Page Number (P):** Identifies the page.
- **Page Offset (d):** Specifies the position within the page.

Example Logical Address:

Logical Address: **2050**

- Page size = 1 KB = 1024 bytes.
- Calculate

1. Page Number (P) = $\lfloor \text{Logical Address}/\text{Page Size} \rfloor = \lfloor 2050/1024 \rfloor = 2.$
2. Offset (d) = $\text{Logical Address} \mod \text{Page Size} = 2050 \mod 1024 = 2.$

Physical Address Translation:

Using the page table, Page 2 maps to Frame 2. The physical address is calculated as:

$$\text{Physical Address} = (\text{Frame Number} \times \text{Page Size}) + \text{Offset}$$

$$\text{Physical Address} = (2 \times 1024) + 2 = 2048 + 2 = 2050$$

Result:

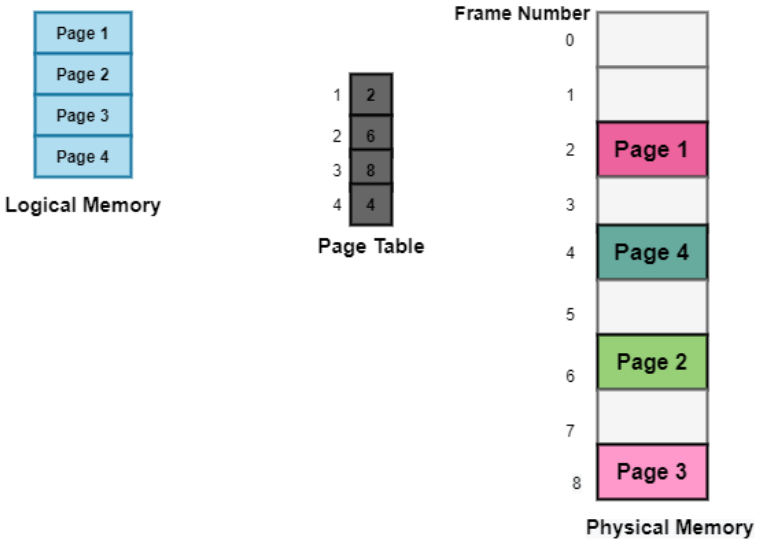
Logical Address 2050 maps to Physical Address 2050.

4.4 Structure of the Page Table

Page Table is a data structure used by the virtual memory system to store the mapping between logical addresses and physical addresses is commonly known as Page Table.

Logical addresses are generated by the CPU for the pages of the processes therefore they are generally used by the processes.

Physical addresses are the actual frame address of the memory. They are generally used by the hardware or more specifically by RAM subsystems.



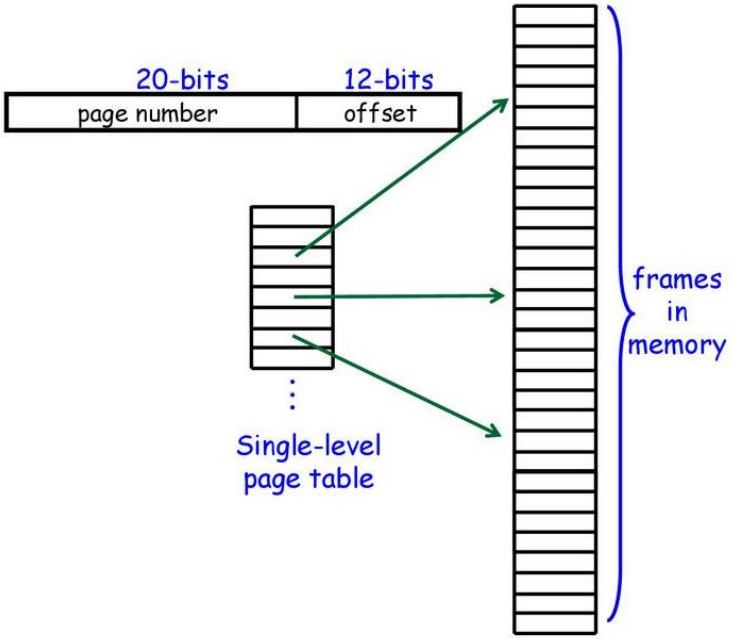
Techniques used for Structuring the Page Table

Some of the common techniques that are used for structuring the Page table are as follows:

1. Single-Level Page Table

Description:

- A single table contains entries mapping each logical page to its corresponding physical frame.
- Suitable for systems with small logical address spaces.



Advantages:

- Simple and easy to implement.

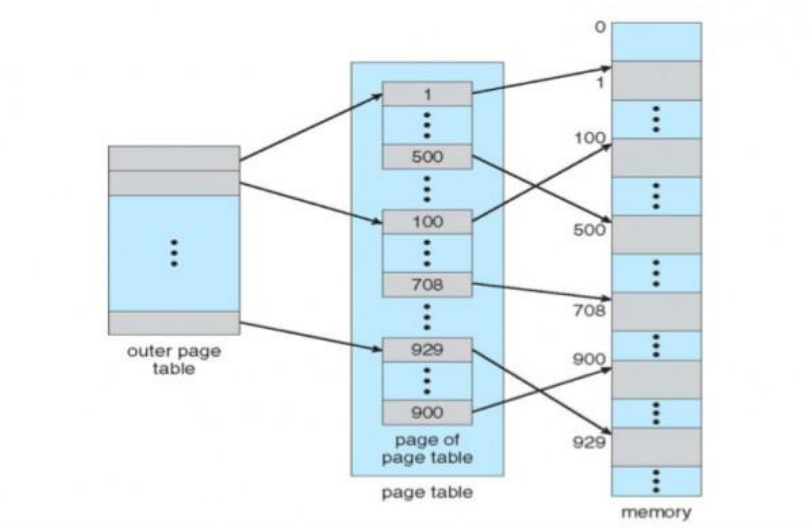
Disadvantages:

- Becomes impractical for large address spaces due to the size of the table.

2. Multi-Level Page Table

Description:

- Divides the page table into levels. The first-level table points to second-level tables, and so on.
- Reduces memory overhead by only allocating space for page table entries that are needed.



Example:

Logical address is divided into:

Index 1: Points to a second-level page table.

Index 2: Points to the frame number in the second-level table.

Offset: Specifies the location within the page.

Advantages:

Saves memory by allocating page table entries on demand.

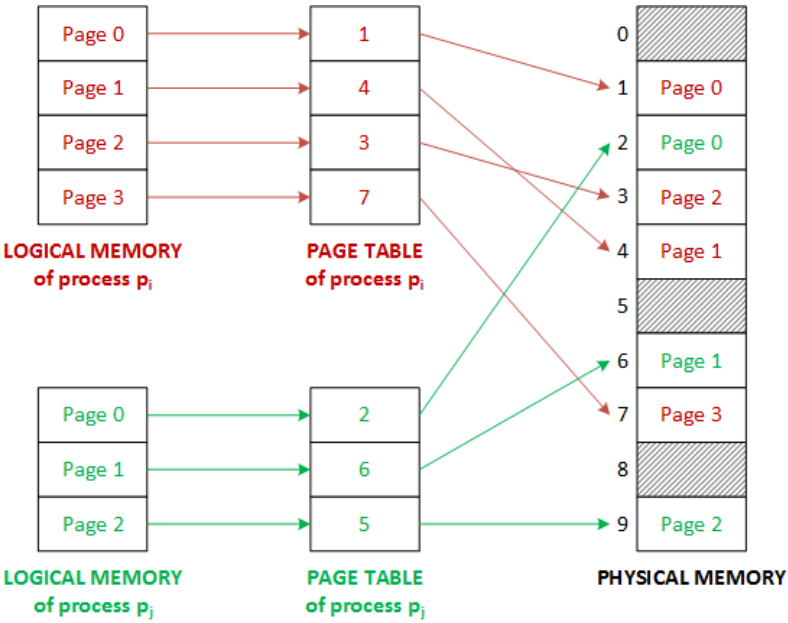
Disadvantages:

Increases translation time as multiple memory lookups are required.

3. Inverted Page Table

Description:

- A single global table shared by all processes.
- Each entry maps a physical frame to the corresponding logical page and process.



Advantages:

- Reduces memory overhead by having only one page table for the entire system.

Disadvantages:

- Slower address translation since a search is required for each reference.
- Typically combined with a hash table for faster lookups.

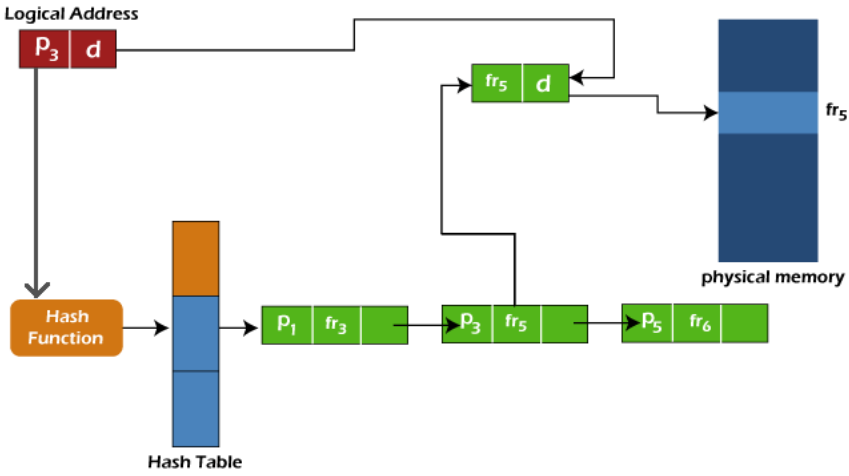
4. Hashed Page Table

Description:

- Uses a hash function to map logical page numbers to physical frame numbers.
- Each entry includes a linked list to handle collisions.

We would understand the working of the hashed page table with the help of an example. The CPU generates a logical address for the page it needs. Now, this logical address needs to be mapped to the physical address. This logical address has two entries, i.e., a page number (P3) and an offset, as shown below.

- The page number from the logical address is directed to the hash function.
- The hash function produces a hash value corresponding to the page number.
- This hash value directs to an entry in the hash table.
- As we have studied earlier, each entry in the hash table has a link list. Here the page number is compared with the first element's first entry. If a match is found, then the second entry is checked.



In this example, the logical address includes page number P3 which does not match the first element of the link list as it includes page number P1. So we will move ahead and check the next element; now, this element has a page number entry, i.e., P3, so further, we will check the frame entry of the element, which is fr5. We will append the offset provided in the logical address to this frame number to reach the page's physical address. So, this is how the hashed page table works to map the logical address to the physical address.

Advantages:

- Efficient for sparse address spaces.

- Reduces the size of the page table by focusing only on mapped pages.

Disadvantages:

- Hashing can introduce overhead, and collisions can increase lookup time.

5. Hierarchical (Tree-Based) Page Table

Description:

- Treats the page table as a tree structure, with intermediate nodes pointing to lower-level nodes.
- The root points to higher-level page tables, and leaves contain frame numbers.

Advantages:

Dynamic and scalable for large address spaces.

Disadvantages:

More complex implementation and longer address translation times.

6. Segmented Page Table

Description:

- Combines segmentation and paging.
- Each segment has its own page table.

Advantages:

- Provides flexibility of segmentation with the efficiency of paging.
- Allows variable-sized segments.

Disadvantages:

- Adds complexity due to managing segments and page tables simultaneously.

7. Clustered Page Table

Description:

- Modifies traditional page tables to map multiple pages (a cluster) to a single frame table entry.
- Reduces the number of entries for large address spaces.

Advantages:

Saves memory space for systems with clustered or contiguous allocation needs.

Disadvantages:

Less granular control, as clusters may lead to internal fragmentation.

8. Shadow Page Tables

Description:

- Used in virtualized environments.
- Maintains a shadow copy of the guest's page table for use by the hypervisor.

Advantages:

Allows efficient memory management in virtualized systems.

Disadvantages:

Adds overhead for synchronization between guest and shadow page tables.

Comparison of Techniques

Technique	Memory Usage	Address Translation Speed	Complexity
Single-Level	High	Fast	Low
Multi-Level	Moderate	Moderate	Moderate
Inverted	Low	Slow	Moderate
Hashed	Moderate	Fast (with good	Moderate

Technique	Memory Usage	Address Translation Speed	Complexity
		hash function)	
Hierarchical	Low	Slow	High
Segmented	Moderate	Moderate	High
Clustered	Low	Moderate	Moderate
Shadow (Virtualization)	High	Moderate	High

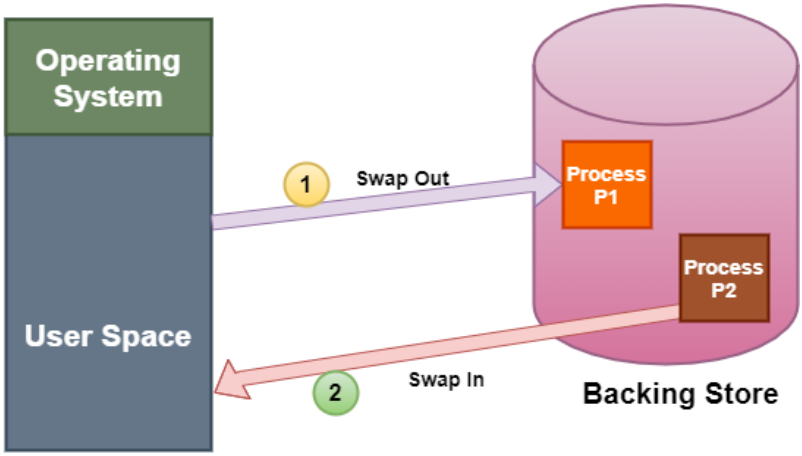
4.5 Swapping

Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

The purpose of the swapping in operating system is to access the data present in the hard disk and bring it to RAM so that the application programs can use it. The thing to

remember is that swapping is used only when data is not present in RAM.

Although the process of swapping affects the performance of the system, it helps to run larger and more than one process. This is the reason why swapping is also referred to as memory compaction.



The concept of swapping has divided into two more concepts: Swap-in and Swap-out.

- **Swap-out** is a method of removing a process from RAM and adding it to the hard disk.
- **Swap-in** is a method of removing a program from a hard disk and putting it back into the main memory or RAM.

Swapping Process Flow

1. The process is selected for swapping based on scheduling or priority.
2. The entire process image is copied from RAM to the swap space.
3. The process is removed from memory, and memory is freed for other processes.
4. When required, the process is reloaded into RAM, and execution resumes from where it was swapped out.

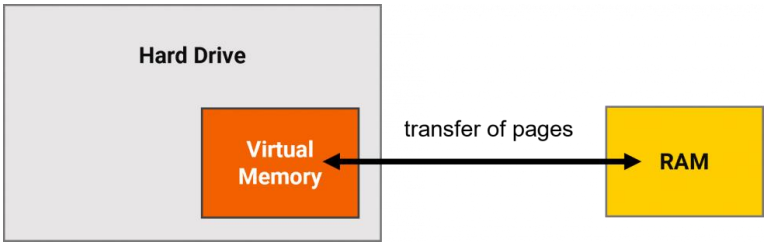
Swapping vs. Paging

Feature	Swapping	Paging
Granularity	Entire process	Individual pages
Overhead	High	Moderate
Efficiency	Less efficient for large processes	More efficient due to smaller swaps
Disk Usage	Requires larger disk space	Requires smaller disk space

4.6 Virtual Memory Management: Introduction

Virtual memory is a critical concept in modern computing that allows a computer system to extend its physical memory using disk storage. This technique provides a mechanism for the execution of processes that may not fully fit into the physical memory available. It offers a larger, more flexible address space for programs and enhances system security and efficiency.

Virtual memory is a memory management technique that allows computers to run programs when they don't have enough physical memory.



- All open programs require a share of RAM, as do the files you are working on.
- Some files and programs are simply too big to fit into the RAM available. This is particularly true when creating videos and large graphics.

- Alternatively, you may just have too many files or programs open at any one time.
- The operating system will utilise virtual memory (your hard drive) when the physical RAM in a computer system is not sufficient to cope with the files and applications currently in use.
- Virtual memory will allow you to continue multitasking and accessing large files despite your RAM being limited or full.

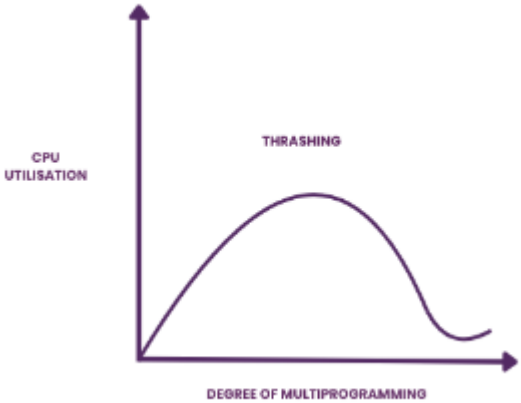
How it works

Virtual memory uses a computer's hard disk to extend the amount of available memory. It temporarily moves data from a computer's RAM to the hard disk or solid-state drive (SSD) when it's not being used. This frees up RAM so that the computer can run multiple programs at once.

Thrashing

Thrashing in an operating system (OS) is when a system spends more time swapping pages between memory and disk than performing useful tasks. This can be caused by a high number of page faults, which happen when the system needs to retrieve a page from the disk because it is not

present in memory. Thrashing can lead to low CPU utilization and make the system unresponsive and slow.



Advantages of Virtual Memory

Efficient Memory Use:

Allows multiple processes to run simultaneously, even if the combined memory requirements exceed physical memory.

Flexibility:

Applications can use more memory than what is physically available on the system.

Isolation:

Enhances security by isolating processes from one another.

Simplified Programming:

Developers do not need to worry about managing physical memory allocation.

Disadvantages of Virtual Memory

Performance Overhead:

Frequent page swaps between RAM and disk (thrashing) can slow down the system.

Storage Dependency:

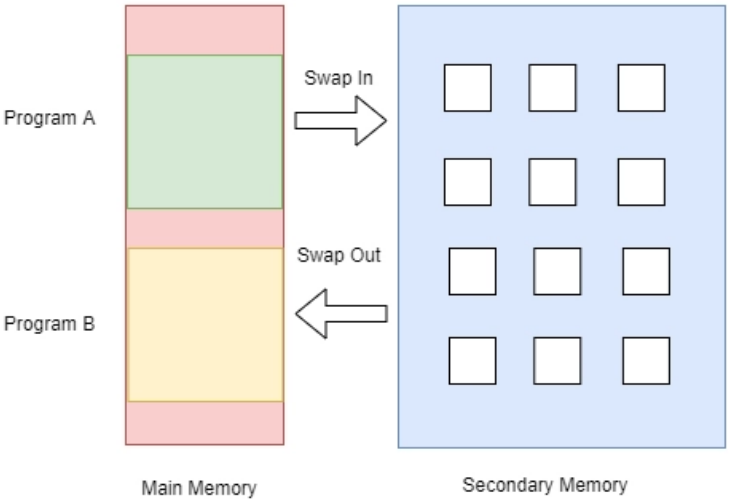
Requires significant disk space for swap areas.

Complexity:

Implementation of virtual memory increases the complexity of the operating system.

4.7 Demand paging

Demand paging is a memory management technique used in operating systems where pages of a program are loaded into physical memory (RAM) only when they are required, rather than preloading all pages at once. This method is part of the virtual memory system and helps efficiently utilize limited physical memory resources.



How Demand Paging Works

Initial State:

- When a program starts, none (or only a minimal number) of its pages are loaded into memory.
- The operating system keeps track of the pages in a page table.

Page Access:

- When the program accesses a page, the OS checks if the page is in memory.
- If the page is already in memory, the program continues execution.

Page Fault:

- If the page is not in memory, a page fault occurs.
- The OS pauses the program, fetches the required page from secondary storage (e.g., disk), loads it into memory, updates the page table, and resumes execution.

Steps in Demand Paging

Check for Page:

- The system checks the page table for the requested page.
- Handle Page Fault:
- If the page is not in memory, a page fault is triggered.

Load Page:

- The required page is loaded into memory, potentially replacing another page if memory is full.

Resume Execution:

- The program resumes from where it was interrupted.

Advantages of Demand Paging

Efficient Memory Use:

- Only the required pages are loaded into memory, leaving space for other processes.

Fast Program Startup:

- Programs can start running without loading the entire memory footprint.

Supports Large Programs:

- Allows programs to use more memory than the system's physical RAM by loading pages on demand.

Disadvantages of Demand Paging

Page Fault Overhead:

Frequent page faults can slow down performance.

Disk Dependency:

Heavy reliance on disk I/O for page fetching may lead to thrashing if memory is overcommitted.

Complexity:

Managing page tables and handling faults adds complexity to the operating system.

4.8 Copy-on-write

Copy-on-Write (COW) is an optimization strategy used in computer science, particularly in operating systems and memory management, to efficiently handle resource sharing. The main idea behind COW is to delay copying data until a

modification is required. This helps conserve memory and improves performance by avoiding unnecessary duplication of data.

Let us take an example where Process A creates a new process that is Process B, initially both these processes will share the same pages of the memory.

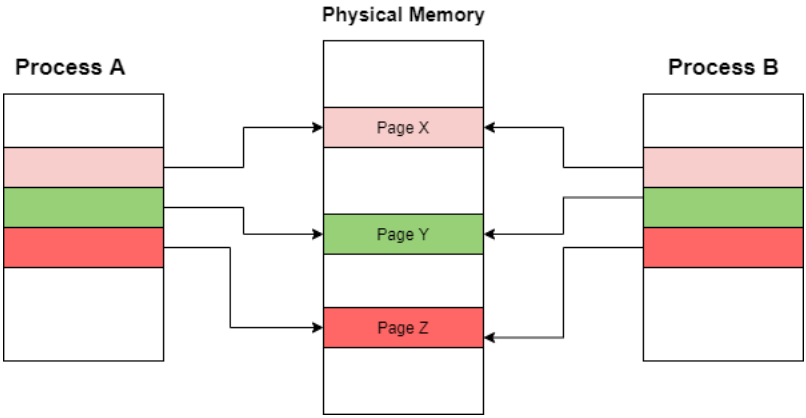
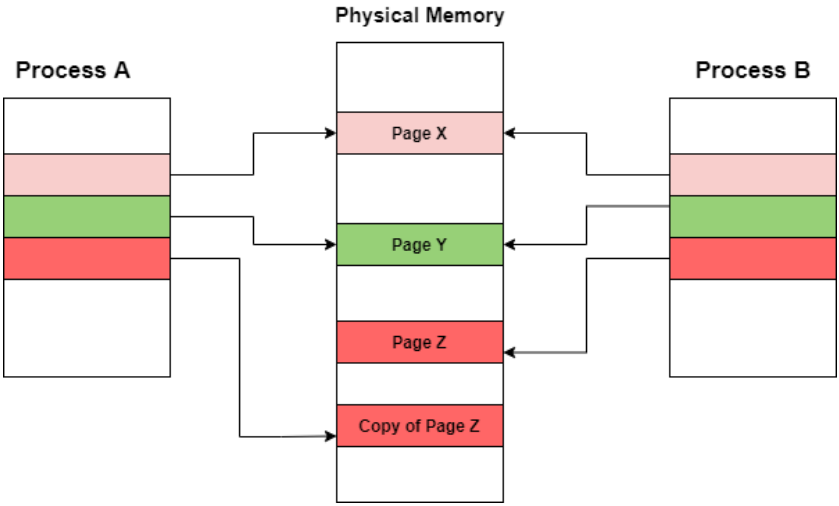


Figure: Above figure indicates parent and child process sharing the same pages

Now, let us assume that process A wants to modify a page in the memory. When the Copy-on-write(CoW) technique is used, only those pages that are modified by either process are copied; all the unmodified pages can be easily shared by the parent and child process.



Whenever it is determined that a page is going to be duplicated using the copy-on-write technique, then it is important to note the location from where the free pages will be allocated. There is a pool of free pages for such requests; provided by many operating systems. And these free pages are allocated typically when the stack/heap for a process must expand or when there are copy-on-write pages to manage.

Advantages of Copy-on-Write

Memory Efficiency:

Reduces memory usage by sharing data until it is modified.

Performance:

Avoids unnecessary copying, especially for read-only data.

Fast Process Creation:

Used during process creation (e.g., with fork() in Unix-like systems), allowing the child process to share the parent's memory until modifications occur.

Disadvantages of Copy-on-Write

Complexity:

Increases the complexity of the memory management system.

Page Fault Overhead:

Writing to a shared page triggers a page fault, adding overhead.

4.9 Page replacement

Page replacement is a mechanism used in operating systems to manage memory efficiently when a program tries to access a page not currently in physical memory (a page fault) and all memory frames are occupied. The operating system must decide which page to remove from memory to make room for the new page.

Why Page Replacement is needed

Limited Physical Memory:

- Systems typically have less RAM than the total memory demands of running processes.

Virtual Memory:

- Programs can use more memory than physically available using virtual memory. When memory is full, page replacement ensures the necessary pages are loaded.

Page Replacement Process

Page Fault Occurs:

A process requests a page not currently in physical memory.

Identify a Victim Page:

The OS selects a page to evict based on a page replacement algorithm.

Swap Out:

The selected page is moved to the swap space on disk.

Load New Page:

The required page is loaded into the freed frame.

Update Page Table:

The page table is updated with the new page's location.

Page Replacement Algorithms

First-In-First-Out (FIFO):

Replaces the oldest page in memory.

Least Recently Used (LRU):

Replaces the page that has not been used for the longest time.

Optimal Page Replacement (OPT):

Replaces the page that will not be needed for the longest time in the future.

Page fault

When the processor need to execute a particular page, that page is not available in main memory, this situation is said to be “Page fault”. When the page fault is happened, the page replacement will be need.

Page replacement algorithms

- FIFO
- Optimal
- LRU

1.FIFO

The idea behind this is “ Replace a page that page is the oldest page of all the pages of main memory”

Example

Reference string: 0 1 2 3 0 1
2 3 0 1 2 3 4
5 6 7 for a memory with 3 frames.

Sol:

Table 7.1 FIFO Behaviour

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	3*	3	3	2*	2	2	1*	1	1	4*	4	4	7*
1		1*	1	1	0*	0	0	3*	3	3	2*	2	2	5*	5	5
2			2*	2	2	1*	1	1	0*	0	0	3*	3	3	6*	6
	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√

Page fault rate = Number of page faults / Number of bits in the reference string

16/16=100%

Advantages of FIFO

Simple to Implement:

Easy to maintain a queue of pages.

Predictable Behavior:

Always removes the oldest page.

Disadvantages of FIFO

Belady's Anomaly:

Increasing the number of frames can sometimes result in more page faults.

Ignores Page Usage:

Does not account for how frequently or recently a page is accessed.

2. Optimal Page replacement algorithm

The idea behind this is "Replace a page that will not be used for the longest period of time"

Example

Reference string: 1 2 3 2 5 6 3 4 6 3 4 6 3

7 3 1 5 3 6 3

4 2 4 3 4 5 1

assume that the memory size is 4 frames.

Page fault: 11/24

Table 7.3 Optimal Behavior

Frame	1	2	3	2	5	6	3	4	6	3	7	3	1
0	1*	1	1	1	1	1	1	1	1	1	1	1	1*
1		2*	2	2*	2	6*	6	6	6*	6	6	6	6
2			3*	3	3	3	3*	3	3	3*	3	3*	3
3					5*	5	5	4*	4	4	7*	7	7
	√	√	√	×	√	√	×	√	×	×	√	×	×

Frame	5	3	6	3	4	2	4	3	4	5	1
0	1	1	1	1	1	2*	2	2	2	2	1*
1	6	6	6*	6	4*	4	4*	4	4*	4	4
2	3	3*	3	3*	3	3	3	3*	3	3	3
3	5*	5	5	5	5	5	5	5	5	5*	5
	√	×	×	×	√	√	×	×	×	×	√

∴ Page fault = 11/24

Advantages of OPT

Optimal Performance:

Minimizes page faults in theory.

Ideal for Predictive Systems:

Useful in simulated environments where future memory access patterns can be predicted.

Disadvantages of OPT

Theoretical and Impractical:

Real systems cannot foresee future page accesses.

Complex to Implement:

Requires significant overhead to track and predict future accesses.

3. LRU (Least Recently Used) Algorithm

The idea behind this is “Replace a page that has not been used for the longest period of time”. This strategy is the “Page replacement algorithm looking backward in time, rather than forward”.

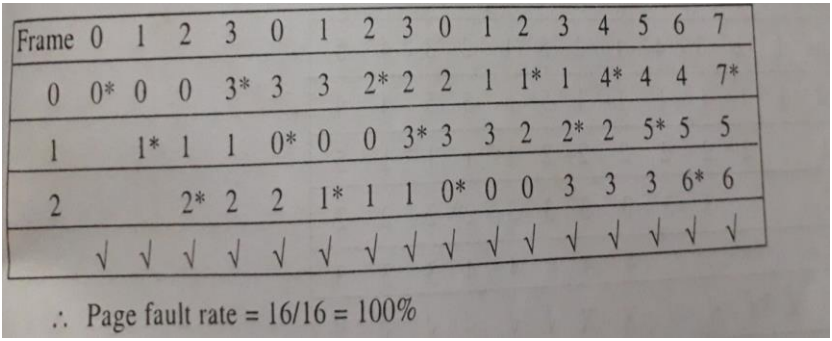
Example

Reference string: 0 1 2 3 0 1

2 3 0 1 2 3 4

5 6 7 with 3 main frames.

Sol: 16/16 =100%



Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	3*	3	3	2*	2	2	1	1*	1	4*	4	4	7*
1		1*	1	1	0*	0	0	3*	3	3	2	2*	2	5*	5	5
2			2*	2	2	1*	1	1	0*	0	0	3	3	3	6*	6
	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√

∴ Page fault rate = 16/16 = 100%

Advantages of LRU

Efficient Memory Use:

- Minimizes page faults in comparison to simpler algorithms like FIFO.

Practical and Intuitive:

- Based on recent usage patterns, which align with real-world access patterns.

Disadvantages of LRU

Implementation Complexity:

Requires additional data structures like stacks, linked lists, or timestamps.

Overhead:

Tracking and updating usage information adds overhead.

Not Always Optimal:

It doesn't account for future access patterns, unlike the theoretical Optimal Page Replacement algorithm.

Comparison Table of FIFO, OPT, and LRU Page Replacement Algorithms

Feature	FIFO	OPT	LRU
Definition	Evicts the oldest page in memory.	Selects the page that will not be used for the longest time in the future.	Evicts the least recently used page.
Page Fault Rate	Generally higher than LRU and OPT.	Ideal scenario, yields the least page faults with future knowledge.	Efficient, minimizes page faults in real scenarios.
Complexity	Simple to implement, uses a queue.	High complexity, requires tracking future	Moderate complexity, requires tracking usage

Feature	FIFO	OPT	LRU
		accesses.	patterns.
Future Knowledge	Does not require knowledge of future memory access.	Requires knowledge of future memory access.	Does not require knowledge of future memory access.
Predictive Power	No predictive power; relies on FIFO order.	Best predictive power in theory, but impractical.	Good predictive power based on recent history.
Performance	Poor for workloads with non-uniform access patterns.	Best theoretical performance but not feasible in real systems.	Good for real-world access patterns, but not optimal.
Implementation	Straightforward,	Complex,	Practical,

Feature	FIFO	OPT	LRU
	uses a simple queue.	requires sophisticated data structures.	uses stacks or counters for tracking.
Real-World Use	Limited; used when other methods are too complex.	Rarely used due to impracticality.	Widely used in modern operating systems due to its balance of simplicity and effectiveness.

4.10 Allocation of frames

In an operating system, the memory available for processes is divided into smaller sections called frames. The allocation of frames to processes is a critical aspect of memory management and can affect system performance significantly. There are different strategies to allocate

frames to processes, each with its advantages and disadvantages.

Types of Frame Allocation Strategies

1. Fixed Allocation:

Description: A fixed number of frames is allocated to each process.

Advantages:

- Simple to implement and manage.
- Easy to calculate the maximum memory required by each process.

Disadvantages:

- Inefficient use of memory if processes do not use all their allocated frames.
- Fixed allocation may lead to fragmentation, causing wasted memory.

2. Variable Allocation:

Description: The number of frames allocated to each process can vary depending on demand.

Advantages:

- More efficient use of memory compared to fixed allocation.

- Allows processes to grow or shrink in memory requirements dynamically.

Disadvantages:

- Complexity increases due to the need for dynamic memory management.
- Potential for fragmentation if the system does not allocate frames efficiently.

3. Proportional Allocation:

Description: Each process is allocated memory in proportion to its needs. For example, a process requiring 30% of the memory might be allocated 30% of the total frames.

Advantages:

- Ensures fair memory allocation among processes.
- Useful for systems where processes have different memory requirements.

Disadvantages:

- Can lead to inefficiencies if the allocation ratio is not well calculated.
- May not provide enough memory for processes with high memory demands.

4. Priority Allocation:

Description: Frames are allocated based on the priority of the process. High-priority processes are given more memory.

Advantages:

- Allows critical processes to have more memory.
- Prioritizes processes based on importance.

Disadvantages:

- Can lead to starvation if lower-priority processes are repeatedly denied memory.
- Inefficient memory usage if not managed carefully.

5. Demand Paging:

Description: Frames are allocated on demand, meaning a process gets a frame only when it is required.

Advantages:

- Reduces page faults by allocating frames only when necessary.
- Efficient use of memory by swapping out unused pages.

Disadvantages:

- Complexity in managing memory as frames are dynamically allocated.

- Higher overhead due to frequent checks for page faults.

Example Scenario

Imagine a system with 10 frames and three processes: P1, P2, and P3.

P1 needs 5 frames.

P2 needs 3 frames.

P3 needs 2 frames.

Depending on the chosen strategy:

Fixed Allocation: Allocate a fixed number of frames (e.g., 3, 3, 4).

Variable Allocation: Adjust frames dynamically based on process demand.

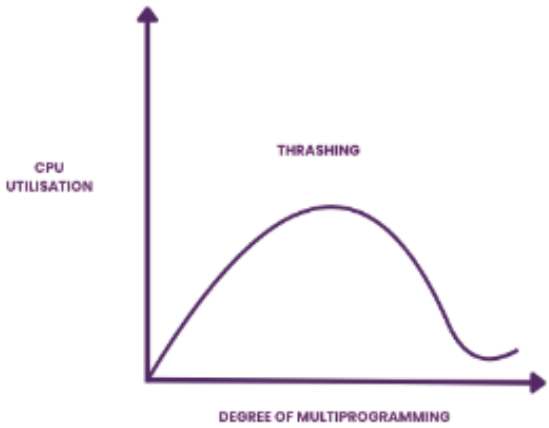
Proportional Allocation: Distribute frames in proportion to each process's needs.

Priority Allocation: Allocate more frames to higher priority processes.

Demand Paging: Frames are allocated only when a page fault occurs.

4.11 Thrashing.

Thrashing in an operating system (OS) is when a system spends more time swapping pages between memory and disk than performing useful tasks. This can be caused by a high number of page faults, which happen when the system needs to retrieve a page from the disk because it is not present in memory. Thrashing can lead to low CPU utilization and make the system unresponsive and slow.



Example Scenario

Consider a system with 4GB of physical memory and three processes (P1, P2, P3) competing for memory. If the system only has 2GB of physical memory and allocates insufficient frames:

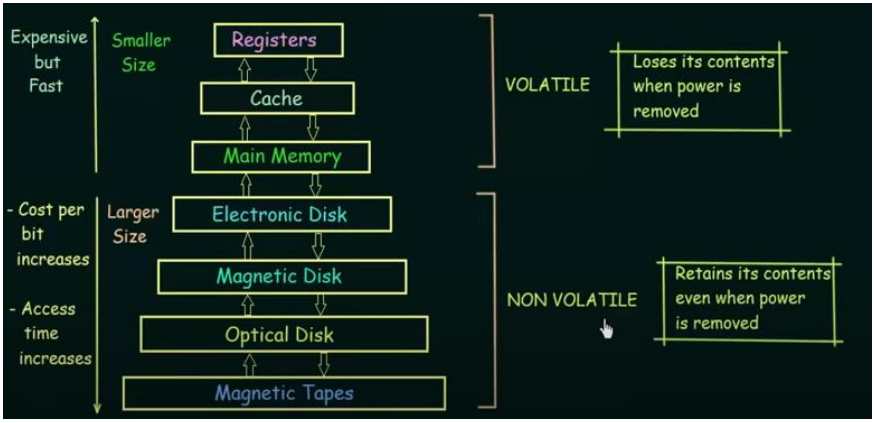
- P1 needs 1GB but is allocated 256MB.
- P2 needs 1GB but is allocated 256MB.
- P3 needs 1GB but is allocated 256MB. Each process will experience frequent page faults because it does not have enough memory.
- P1 will continuously load and unload pages as it accesses more memory than it can hold.
- This can lead to thrashing, where the system is constantly swapping pages between disk and memory without actually executing useful instructions.

In such a case, even though the system appears to have enough memory, the lack of sufficient frames causes performance degradation.

4.12 Storage Management: Overview of Mass Storage Structure

Mass storage is the backbone of computer systems, providing long-term data storage and retrieval capabilities. It plays a crucial role in ensuring data persistence, system performance, and access speed. Understanding the structure

and components of mass storage is essential for effective storage management in operating systems.



Mass storage is a collection of devices and systems that store large amounts of data. The structure of mass storage devices is made up of platters, tracks, sectors, and cylinders:

Platters: Rigid metal platters for hard disk drives, or flexible plastic platters for floppy disks

Tracks: Concentric rings on the working surface of a platter

Sectors: Divisions within a track that typically contain 512 bytes of data

Cylinders: Collection of tracks that are the same distance from the edge of the platter

The storage capacity of a disk drive is calculated by multiplying the number of heads, tracks, sectors, and bytes per sector.

Some examples of mass storage devices include:

Hard disk drives (HDDs)

A common internal storage device with a magnetized surface that spins at 60–120 revolutions per second

Magnetic tape drives

A cheap way to store large amounts of data, often used as a backing store for mainframe computers

Optical disc drives

Can store more data than floppy disks, with CD-ROMs, CD-Rs, and CD-RWs being the three main types

Solid-state drives (SSDs)

A type of mass storage device

Storage management is the process of ensuring that data is accessible quickly while also maintaining data integrity, compliance with policies and regulations, and efficient use of storage resources.

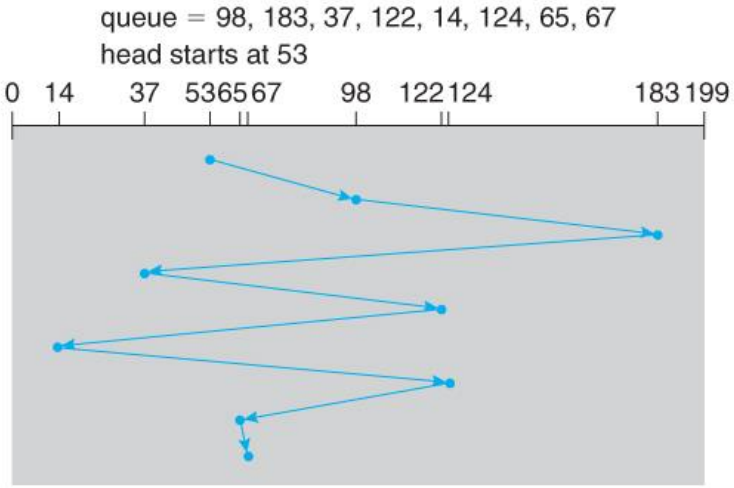
4.13 HDD Scheduling

Hard Disk Drive (HDD) scheduling refers to the methods used by operating systems to manage and prioritize requests for read and write operations on a hard disk. The goal of HDD scheduling is to minimize the seek time and rotational latency, thereby optimizing the overall performance of disk access and reducing wait times for processes.

Key Scheduling Algorithms

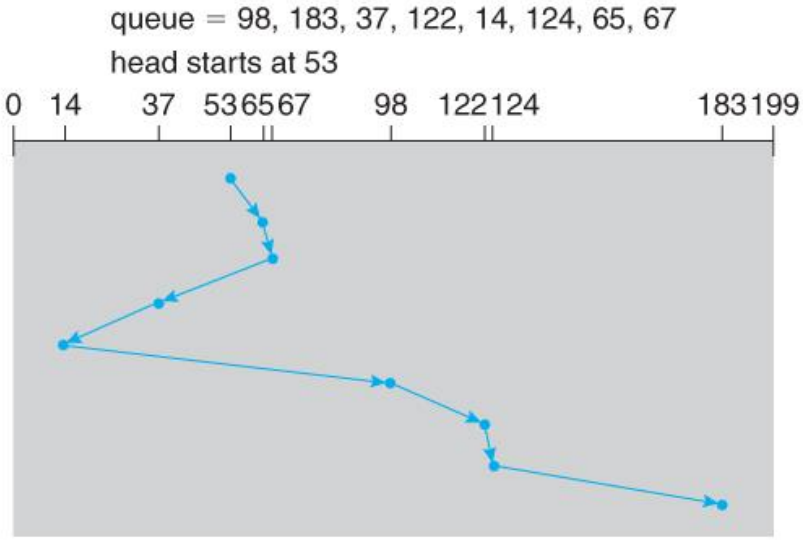
1. First-In, First-Out (FIFO):

Description: Requests are processed in the order they arrive. The oldest request is served first. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:



2. Shortest Seek Time First (SSTF):

Description: Selects the request with the shortest seek time (distance) from the current head position.

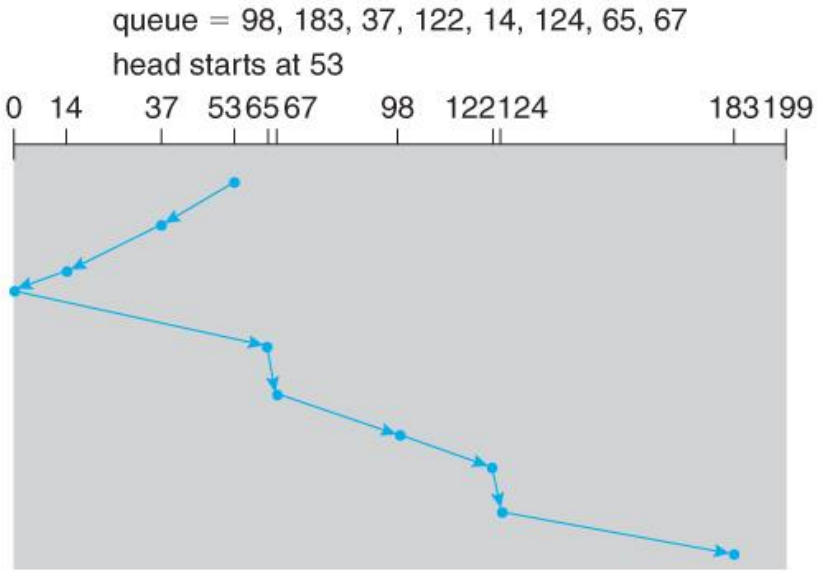


3.SCAN (Elevator Algorithm):

Description: Moves the disk arm from one end to the other, servicing requests along the way.

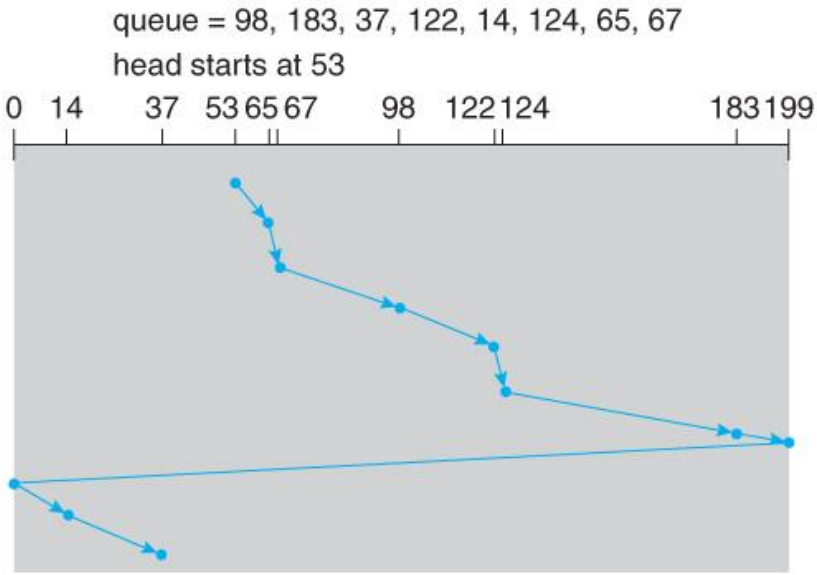
Upward Direction: Moves from one end of the disk to the other in one continuous motion.

Downward Direction: After reaching the end, it reverses direction and continues servicing requests.



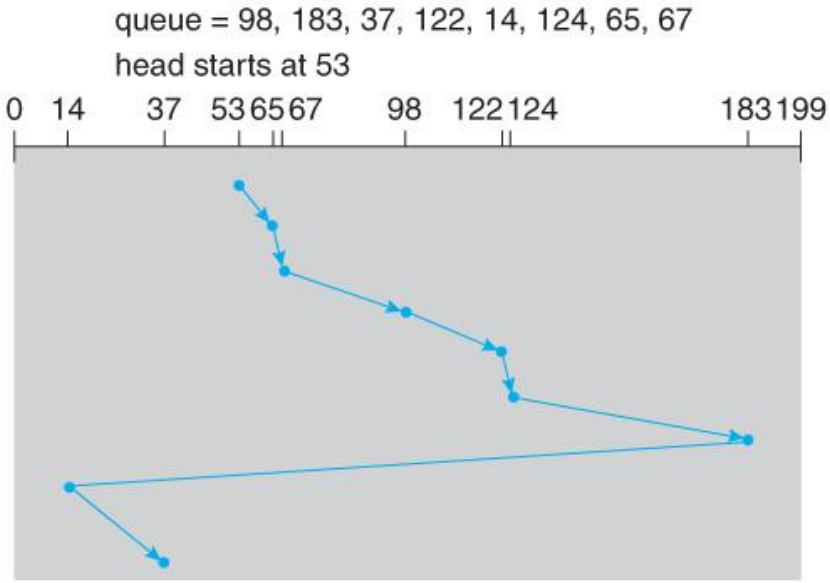
4. C-SCAN (Circular SCAN):

Description: Similar to SCAN, but after reaching the end, the head immediately jumps back to the beginning of the disk without serving requests between.



5. C-LOOK (Circular LOOK):

Description: A variation of LOOK, where the disk arm moves only as far as the last request, then jumps back to the beginning, ignoring idle requests.



4.14 RAID

RAID (redundant array of independent disks) is a way of storing the same data in different places on multiple hard disks or solid-state drives (SSDs) to protect data in the case of a drive failure. There are different RAID levels

RAID (Redundant Array of Independent Disks) is a data storage virtualization technology that combines multiple physical hard drives into a single logical unit to achieve performance improvement, fault tolerance, or both. It is widely used in operating systems for data management

and reliability. RAID can be implemented in software (via the OS) or hardware (via a dedicated RAID controller).

Key Concepts of RAID:

Redundancy: Provides data duplication to prevent data loss in case of disk failure.

Performance: Improves read/write operations by distributing data across multiple drives.

Logical Storage: Abstracts multiple physical drives into a single logical volume visible to the OS.

RAID Levels

There are several RAID levels, each serving different purposes:

1. RAID 0 (Striping)

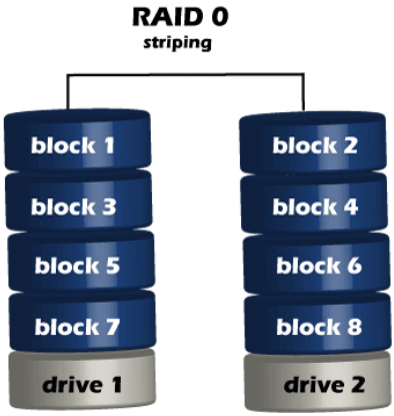
Purpose: Performance.

How it works:

- This configuration has striping but no redundancy of data. It offers the best performance, but it does not provide fault tolerance.
- Data is split into blocks and written across all drives.

Advantages: High read/write speeds.

Disadvantages: No redundancy; failure of one drive leads to data loss.



2. RAID 1 (Mirroring)

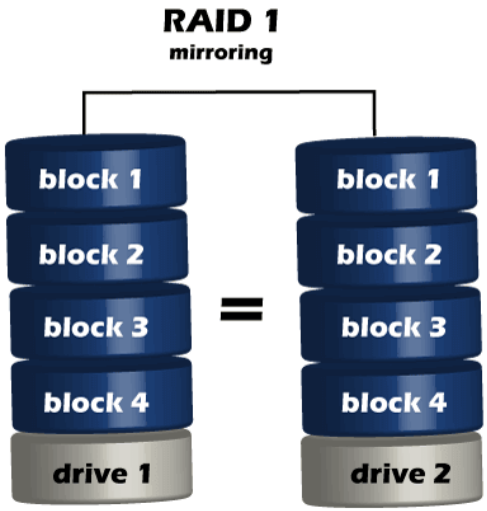
Purpose: Fault tolerance.

How it works:

- Also known as disk mirroring, this configuration consists of at least two drives that duplicate the storage of data. There is no striping. Read performance is improved, since either disk can be read at the same time. Write performance is the same as for single disk storage.
- Data is duplicated across two or more drives.

Advantages: High reliability; data remains accessible if one drive fails.

Disadvantages: Storage cost doubles since each bit of data is stored twice.



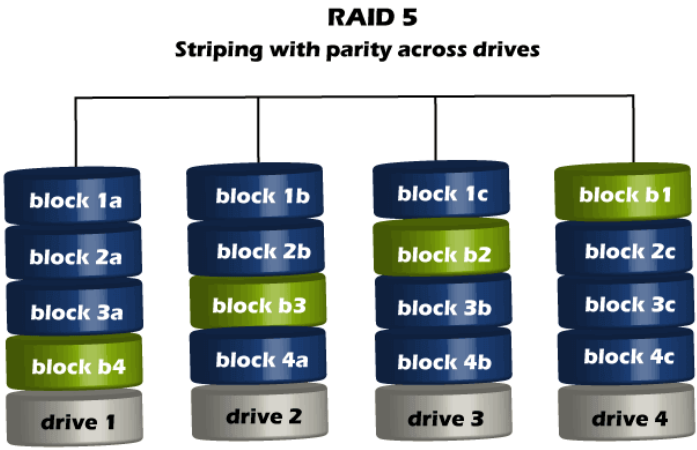
3. RAID 5 (Striping with single Parity)

Purpose: Fault tolerance and performance.

How it works: Data and parity information are striped across all drives. Parity allows recovery in case of a single drive failure.

Advantages: Efficient storage and fault tolerance.

Disadvantages: Slightly slower write speeds due to parity calculations.



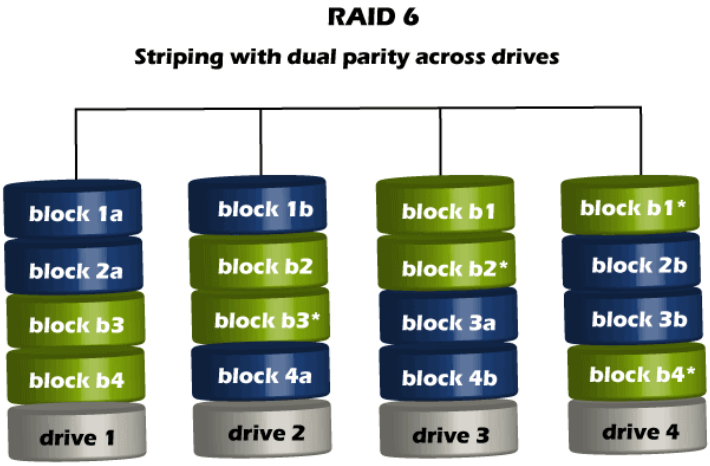
4. RAID 6 (Striping with Double Parity)

Purpose: Enhanced fault tolerance.

How it works: Similar to RAID 5 but with two sets of parity.

Advantages: Can tolerate two simultaneous drive failures.

Disadvantages: Slower writes and requires more storage.



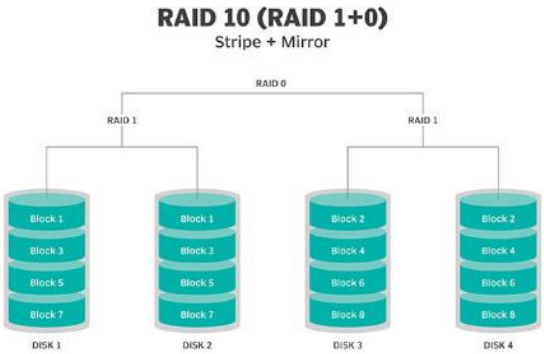
5. RAID 10 (1+0, or Mirroring and Striping)

Purpose: High performance and fault tolerance.

How it works: Combines RAID 1 and RAID 0. Data is mirrored and then striped.

Advantages: High speed and redundancy.

Disadvantages: Expensive due to high storage overhead.



RAID Implementation in Operating Systems

RAID controller

A RAID controller is a device used to manage hard disk drives in a storage array. It can be used as a level of abstraction between the OS and the physical disks, presenting groups of disks as logical units. Using a RAID controller can improve performance and help protect data in case of a crash.

A RAID controller may be hardware- or software-based. In a **hardware-based RAID** product, a physical controller manages the entire array. The controller can also be designed to support drive formats such as Serial Advanced Technology Attachment and Small Computer System Interface. A physical RAID controller can also be built into a server's motherboard.

With **software-based RAID**, the controller uses the resources of the hardware system, such as the central processor and memory. While it performs the same functions as a hardware-based RAID controller, software-based RAID controllers may not enable as much of a

performance boost and can affect the performance of other applications on the server.

If a software-based RAID implementation is not compatible with a system's boot-up process and hardware-based RAID controllers are too costly, firmware, or driver-based RAID, is a potential option.

Firmware-based RAID controller chips are located on the motherboard, and all operations are performed by the central processing unit (CPU), similar to software-based RAID. However, with firmware, the RAID system is only implemented at the beginning of the boot process. Once the OS has loaded, the controller driver takes over RAID functionality. A firmware RAID controller is not as pricey as a hardware option, but it puts more strain on the computer's CPU. Firmware-based RAID is also called hardware-assisted software RAID, hybrid model RAID and fake RAID.

1. Software RAID

- Configured at the operating system level.

- No special hardware required.

Example tools:

Linux: Logical Volume Manager (LVM).

Windows: Disk Management or Storage Spaces.

Pros:

- Cost-effective.
- Flexible configuration.

Cons:

- Relies on CPU for RAID processing, potentially reducing system performance.

2. Hardware RAID

- Managed via a dedicated RAID controller card.
- Transparent to the OS.

Pros:

- Offloads RAID processing to the hardware.
- Often faster and more reliable.

Cons:

- Higher cost.
- Potential vendor lock-in.

UNIT-V

File System: File System Interface: File concept, Access methods, Directory Structure; File system Implementation: File-system structure, File-system Operations, Directory implementation, Allocation method, Free space management; File-System Internals: File System Mounting, Partitions and Mounting, File Sharing.

Protection: Goals of protection.

5.1 File System: File System Interface: File concept

A file system is a critical component of an operating system that manages how data is stored and retrieved on storage devices like hard drives or SSDs. The file concept is foundational to how users and applications interact with data on a computer.

a) What is a File?

A file is a logical collection of related data, stored on secondary storage, such as a hard disk, SSD, or removable media.

b) Attributes of a File

Each file typically has a set of attributes that describe its **properties:**

Name: A unique identifier for the file within its directory.

Type: Indicates the kind of data in the file (e.g., .txt, .jpg, .exe).

Location: The physical or logical location on the storage device.

Size: The current size of the file, often measured in bytes.

Protection: Permissions that control who can read, write, or execute the file.

Timestamps: Information about when the file was created, last modified, and last accessed.

Ownership: Identifies the user or group that owns the file.

c) File Types

Operating systems often classify files into types based on their purpose:

Regular Files: Contain user data like text, images, or executables.

Directories: Special files that store lists of file names and their attributes.

Device Files: Represent hardware devices (e.g., /dev files in Linux).

Special Files: Pipes, sockets, or symbolic links used for interprocess communication or shortcuts

d) Common Operations in File Systems

Create: Allocates space and initializes file metadata.

Read: Retrieves file content.

Write: Updates or appends data to a file.

Delete: Removes file entries and reclaims space.

Open/Close: Prepares a file for access and releases resources afterward.

e) Importance of the File Concept

The file concept abstracts the complexity of hardware and presents a user-friendly interface, enabling:

Data Organization: Simplifies storing and retrieving data.

Collaboration: Allows multiple users or processes to interact with the same data.

Data Persistence: Ensures data is saved even after the system shuts down.

d) Advantages of File Systems

Efficient Data Access: Optimized for quick reads and writes.

Data Organization: Hierarchical structure simplifies navigation.

Data Persistence: Ensures data remains intact across reboots.

Fault Tolerance: Protects against hardware and software failures

By providing these capabilities, the file system interface is essential for the functionality and usability of modern operating systems.

5.2 Access methods

Access methods in a file system determine how data stored in files can be retrieved and manipulated.

File Access Methods:

Sequential Access: Data is read/written in sequence.

Random Access: Allows jumping to any location within a file.

Indexed Access: Uses indexes for fast data retrieval.

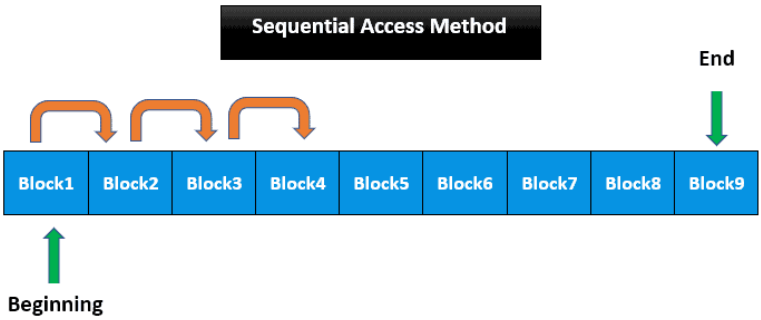
Sequential-Indexed Access: Combines sequential access and indexing for efficient data retrieval.

1. Sequential Access:

In an operating system, sequential access is a file access method that reads a file word by word in a linear order. It's similar to flipping through a book, where you can't jump directly to a specific page.

How it works

A pointer is maintained which initially points to the base address of the file. If the user wants to read first word of the file then the pointer provides that word to the user and increases its value by 1 word. This process continues till the end of the file.



Advantages

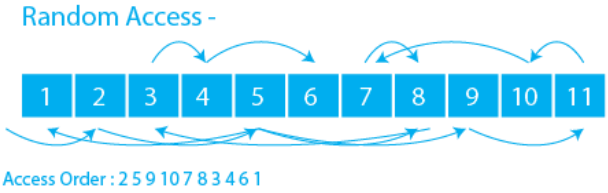
Simple and efficient for sequentially structured data.

Disadvantages

Inefficient for non-sequential access

2. Random Access:

In an operating system, random access is the ability to access data at any location within a file or data structure without having to go through other elements sequentially. This is in contrast to sequential access, where data is accessed in a predetermined, linear sequence.



Description: Data can be accessed directly by specifying its location in the file.

How it works: A file is treated as a series of fixed-size logical blocks, each with a unique address. Applications use these addresses to read or write data directly.

Advantages:

Fast access to specific parts of the file.

Disadvantages:

More complex to implement compared to sequential access.

3. Indexed Access:

The indexed access method is a file access technique in an operating system that uses an index to access records in a file:

Description: Uses an index to keep track of the locations of data blocks within a file, similar to an index in a book.

How it works: An index is maintained, and each entry points to the data's physical location in the file.

Advantages:

Combines fast access with structured organization.

Disadvantages:

Additional storage is required for maintaining the index.

4. Sequential-Indexed Access

Description: Combines sequential access and indexing for efficient data retrieval.

Operations:

Use the index to locate a starting point.

Read sequentially from that point.

Characteristics:

Optimized for datasets requiring both types of access patterns.

Use Cases:

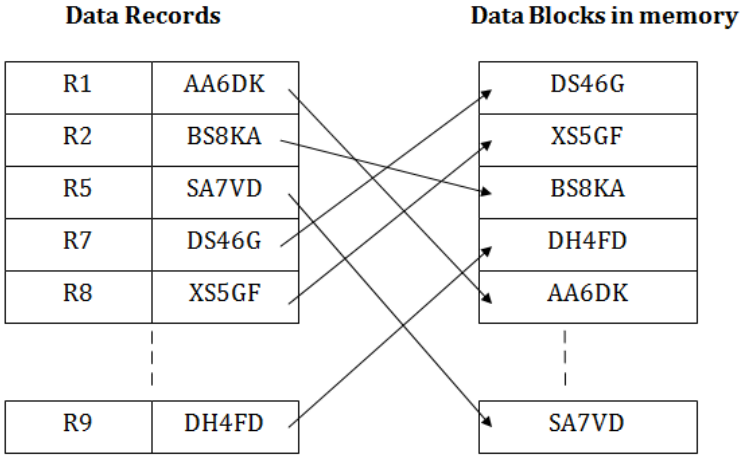
Large, sorted datasets (e.g., customer records, sorted logs).

Advantages:

Balances flexibility and efficiency.

Disadvantages:

Index management introduces overhead.



5. Content-Based Access

Description: Files are accessed based on their content, often using keywords or metadata.

Operations:

Search for files or records matching a specific query.

Characteristics:

Requires full-text indexing or metadata tagging.

Use Cases:

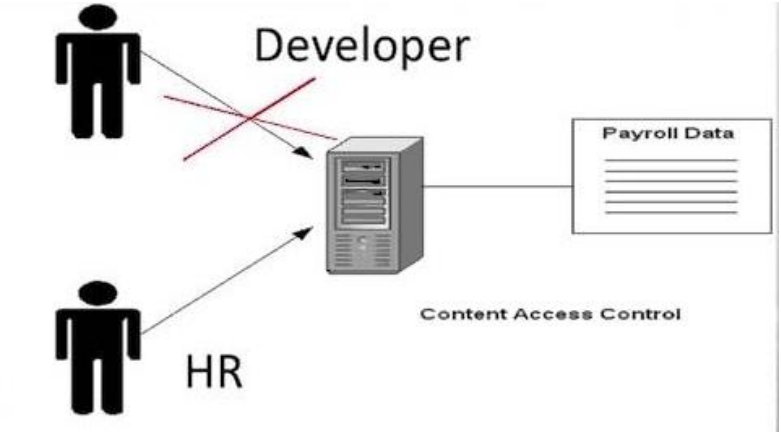
Search engines, document management systems.

Advantages:

Flexible and powerful queries.

Disadvantages:

High computational and storage overhead for indexing.

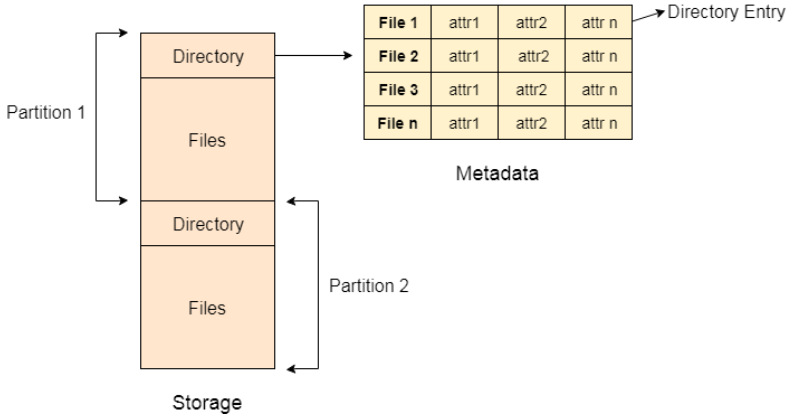


Comparison Table

Access Method	Speed	Complexity	Use Case	Drawback
Sequential Access	Slow for random access	Simple	Log files, media playback	Inefficient for random access
Direct Access	Fast for random access	Moderate	Databases, media editing	Requires fixed block size
Indexed Access	Fast for large files	Complex	Databases, inventory systems	Overhead for index storage
Sequential Indexed	Moderate to fast	Moderate	Sorted datasets	Index maintenance overhead
Hierarchical Access	Moderate	Simple to use	File navigation	Slower navigation in deep hierarchies
Content-Based Access	Fast for searches	High	Search engines, documents	Computationally expensive

5.3 Directory Structure

A directory structure is a hierarchical or organizational framework used by operating systems to organize files and directories (or folders). It provides a way to manage, access, and retrieves data efficiently, ensuring that the file system remains navigable and structured.



a) Directory Operations

Create: Add a new directory.

Delete: Remove a directory (usually requires it to be empty).

Rename: Change the name of a directory.

Traverse: Navigate through directories to locate files.

Search: Find files or directories based on criteria.

List: Display the contents of a directory.

b) Components of a Directory Structure

Root Directory:

- The top-most directory in a hierarchical structure.
- All other files and directories branch out from here.
- Denoted as / in Unix/Linux and C:\ (or similar drive letters) in Windows.

Subdirectories:

- Directories within another directory.
- Allow hierarchical organization, enabling logical grouping of related files.

Files:

- Actual data stored within directories.

Paths:

- Specifies the location of a file or directory.

Absolute Path: Full path from the root (e.g., /home/user/file.txt).

Relative Path: Path relative to the current directory (e.g., ../documents/file.txt).

c) Types of Directory Structures

1. Single-Level Directory

2. Two-Level Directory

3. Hierarchical Directory

4. Acyclic-Graph Directory

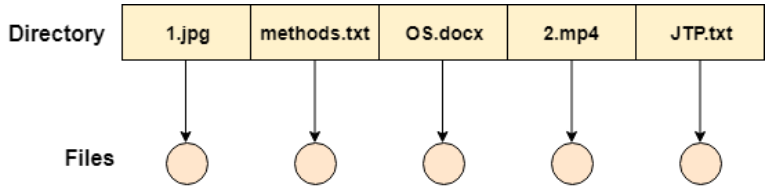
5. General Graph Directory

1. Single-Level Directory

Description:

All files are stored in a single directory.

The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.



Single Level Directory

Advantages:

- Simple and easy to implement.

Disadvantages:

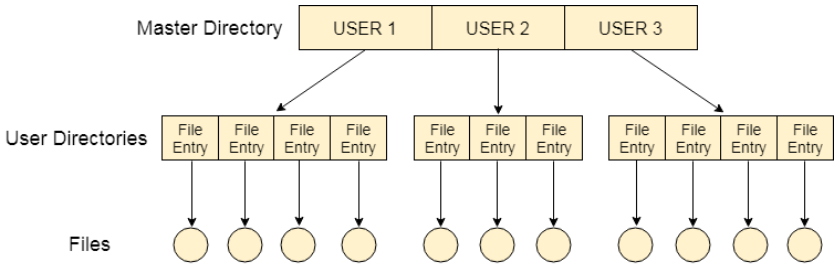
- Difficult to manage when the number of files increases.
- No logical grouping or user separation.

2. Two-Level Directory

Description:

Each user has a separate directory within the root directory.

In two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.



Two Level Directory

Advantages:

- Better file organization.
- User separation ensures privacy and avoids name collisions.

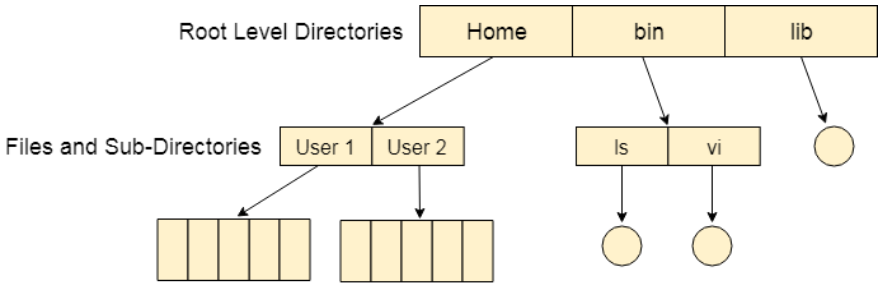
Disadvantages:

- Limited to two levels; cannot organize files hierarchically within a user's directory.

3. Hierarchical Directory

Description:

- Directories are organized in a tree-like structure.
- Users can create subdirectories within directories, allowing unlimited levels.



The Structured Directory System

Advantages:

- Flexible and scalable.
- Logical grouping of files and directories.

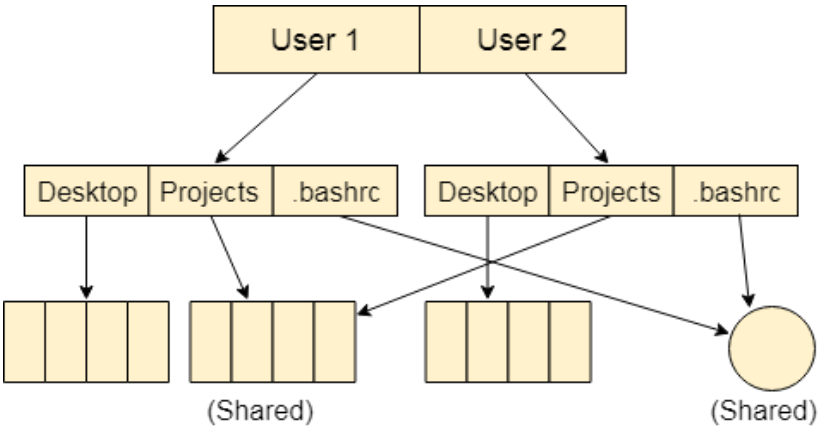
Disadvantages:

- Complexity increases with deep structures.

4. Acyclic-Graph Directory

Description:

- Allows directories to share files and subdirectories via links (hard links or symbolic links).



Acyclic-Graph Structured Directory System

Advantages:

- Avoids duplication of data by enabling sharing.

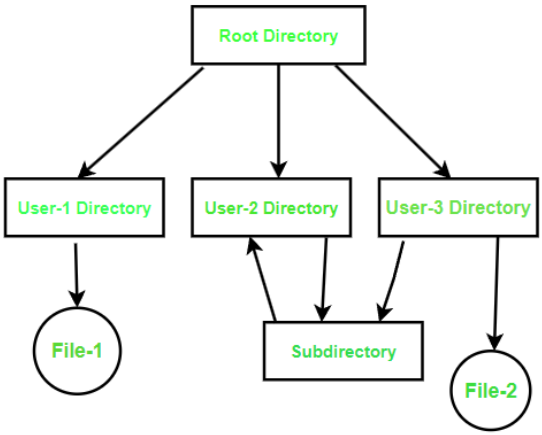
- Saves storage space.

Disadvantages:

- Complicated management due to multiple references.
- Deletion must be handled carefully to avoid dangling references.

5. General Graph Directory

A general-graph directory structure in an operating system (OS) is a flexible directory structure that allows files and directories to have multiple parent directories. This structure is adaptable for complex organizational needs and permits cycles, which allows for more intricate and flexible linking of directories.



Description:

Directories can contain cycles, meaning files and directories can refer to each other in loops.

Advantages:

Highly flexible.

Disadvantages:

- Complex to manage.
- Requires garbage collection to deal with loops and ensure proper deletion.

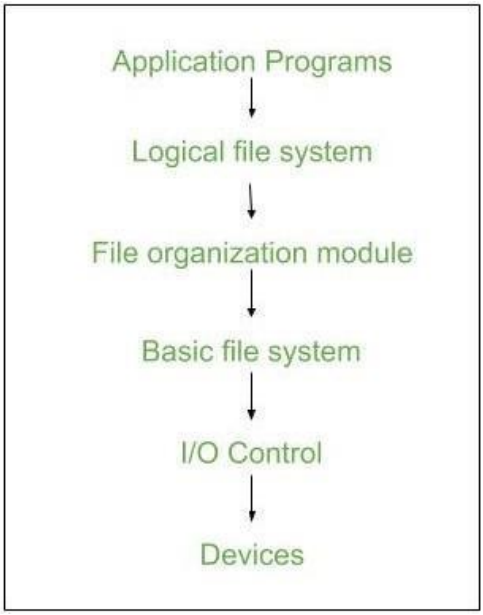
5.4 File System Implementation: File-system structure

File system implementation encompasses various aspects of designing, organizing, and managing data on storage devices. It involves the following key components:

1. File-system structure

The file-system structure is organized into layers, each responsible for specific tasks. Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

The image shown below, elaborates how the file system is divided in different layers, and also the functionality of each layer.



The file-system structure is organized into layers, each responsible for specific tasks.

Logical Layer:

- Provides abstraction for file operations like create, read, write, and delete.
- Manages metadata, including file names, access permissions, and timestamps.

File-Organization Layer:

- Maps logical files to physical storage blocks.
- Handles file allocation, storage structure, and metadata storage.

Storage Management Layer:

- Deals with block allocation and deallocation.
- Maintains free-space lists and optimizes storage usage.

I/O Control Layer:

- Interfaces with device drivers to interact with hardware.
- Handles low-level operations like reading/writing physical blocks.

Physical Storage Layer:

- Represents the actual storage devices like hard disks, SSDs, and flash drives.

2. File-System Operations

File-system operations can be categorized into file operations and directory operations:

a) File Operations

Create: Allocates metadata and space for a new file.

Open: Maps a file to an in-memory data structure.

Read/Write: Access or modify file contents.

Close: Releases in-memory data structure after use.

Delete: Frees allocated space and removes metadata.

c) Directory Operations

Create Directory: Adds a new directory in the hierarchy.

List Directory: Displays the contents of a directory.

Delete Directory: Removes a directory (requires it to be empty).

Rename Directory: Changes the name of a directory.

Traverse: Access files and subdirectories within a directory.

Other Operations

Mount: Integrates a file system into the operating system's directory structure.

Unlink: Removes a file's directory entry without deleting its data immediately (useful for shared files).

3. Directory Implementation

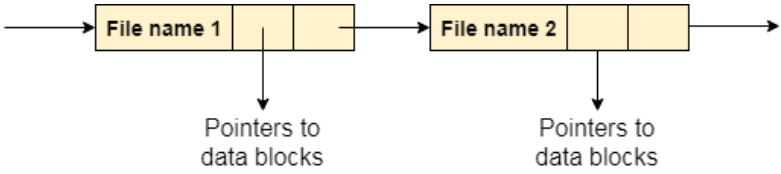
There is the number of algorithms by using which, the directories can be implemented. However, the selection of an appropriate directory implementation algorithm may significantly affect the performance of the system.

A directory is a special file that stores information about other files.

The directory implementation algorithms are classified according to the data structure they are using. Different methods can be used to implement directories:

a. Linear List

In this algorithm, all the files in a directory are maintained as singly lined list. Each file contains the pointers to the data blocks which are assigned to it and the next file in the directory.(A simple list of file names and metadata).



Linear List

Advantages:

Easy to implement.

Disadvantages:

Slow for searching in large directories.

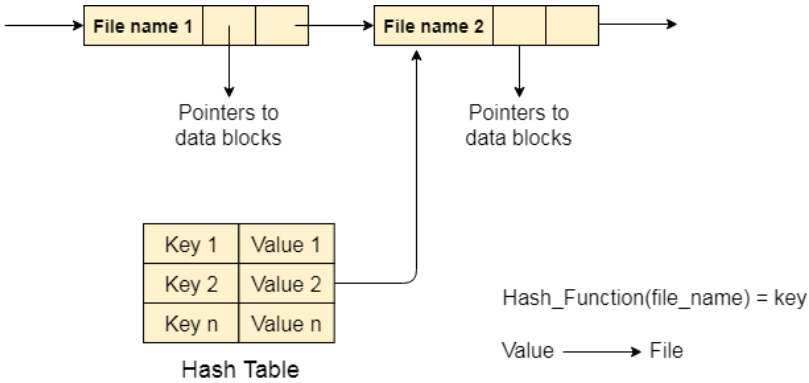
b. Hash Table

To overcome the drawbacks of singly linked list implementation of directories, there is an alternative

approach that is hash table. This approach suggests to use hash table along with the linked lists.

A key-value pair for each file in the directory gets generated and stored in the hash table. The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory.

Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.(Uses a hash function to map file names to metadata).



Advantages:

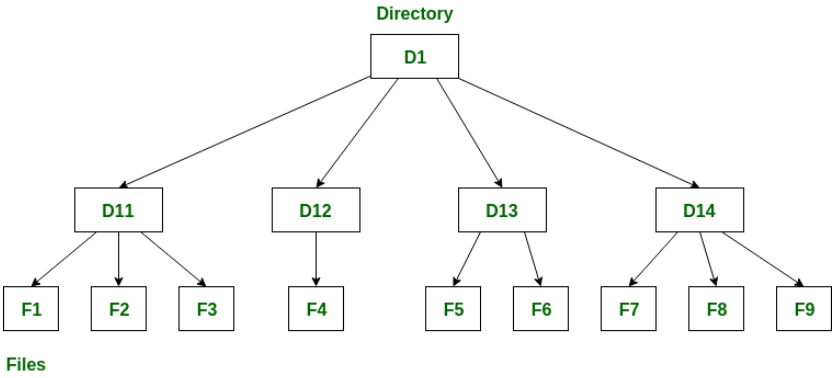
Faster lookups compared to a linear list.

Disadvantages:

Hash collisions require extra handling.

c. Tree-Based Structure

Organizes directories hierarchically (e.g., binary search tree or B+ tree).



The tree-structured directory structure in an operating system (OS) is a hierarchical way to organize files and directories, similar to a tree upside down. It's commonly used in personal computers and offers several advantages, including:

Efficient searching: Files can be located using either absolute or relative paths.

Logical organization: Files of the same type can be grouped together in the same directory.

Scalability: It's easy to create or remove directories and subdirectories.

Reduced name collisions: The likelihood of name collisions within a directory is lower.

Flexibility: Users can create any number of directories within their User File Directory (UFD)

Advantages:

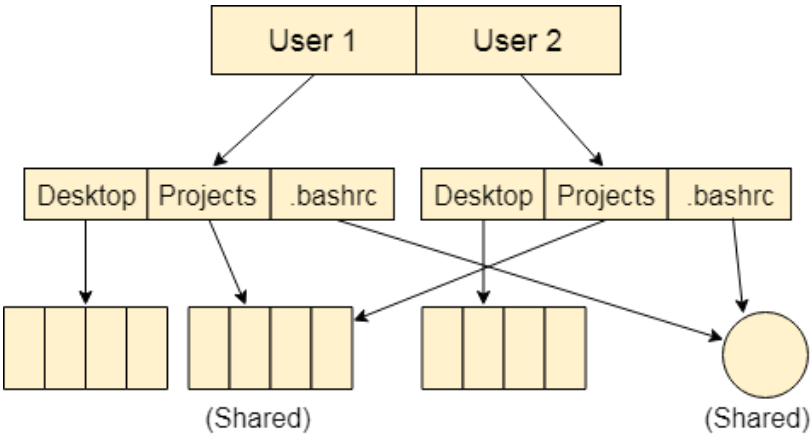
Efficient searching, insertion, and deletion.

Disadvantages:

Slightly more complex to implement.

d. Acyclic Graph

An acyclic graph directory structure in an operating system allows a file to be accessed from multiple directories, which is a generalization of the tree directory structure. This structure is useful when multiple users need access to the same file, such as when two programmers are working together.(Allows files or directories to be linked across directories).



Acyclic-Graph Structured Directory System

Advantages:

Supports shared files and directories.

Disadvantages:

Risk of dangling links; requires careful management.

4. File Allocation Methods

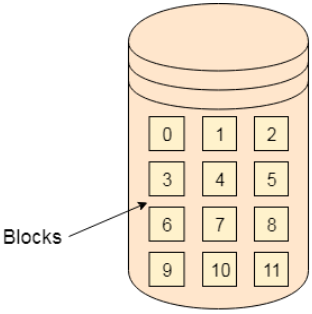
There are various methods which can be used to allocate disk space to the files. Selection of an appropriate allocation method will significantly affect the performance and efficiency of the system. Allocation method provides a way in which the disk will be utilized and the files will be accessed.

There are following methods which can be used for allocation

a) Contiguous Allocation:

If the blocks are allocated to the file in such a way that all the logical blocks of the file get the contiguous physical block in the hard disk then such allocation scheme is known as contiguous allocation.

In the image shown below, there are three files in the directory. The starting block and the length of each file are mentioned in the table. We can check in the table that the contiguous blocks are assigned to each file as per its need.



Hard Disk

File Name	Start	Length	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Directory

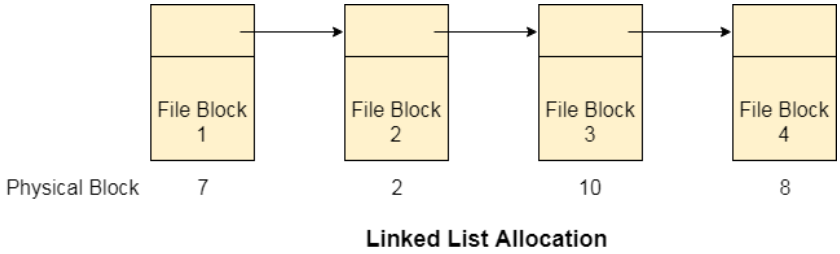
Contiguous Allocation

Advantages: Simple and fast.

Disadvantages: Prone to fragmentation and resizing issues.

b) Linked Allocation:

Linked List allocation solves all problems of contiguous allocation. In linked list allocation, each file is considered as the linked list of disk blocks. However, the disks blocks allocated to a particular file need not to be contiguous on the disk. Each disk block allocated to a file contains a pointer which points to the next disk block allocated to the same file. (Each block points to the next block in the file).



Linked List Allocation

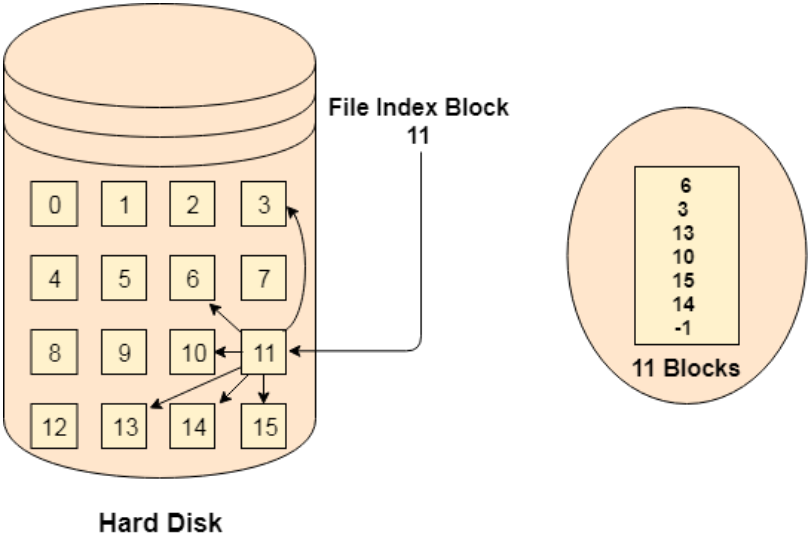
Advantages: Flexible, no fragmentation.

Disadvantages: Slower access due to sequential traversal.

c) Indexed Allocation:

Instead of maintaining a file allocation table of all the disk pointers, Indexed allocation scheme stores all the disk pointers in one of the blocks called as indexed block.

Indexed block doesn't hold the file data, but it holds the pointers to all the disk blocks allocated to that particular file. Directory entry will only contain the index block address. (An index block stores pointers to all file blocks).

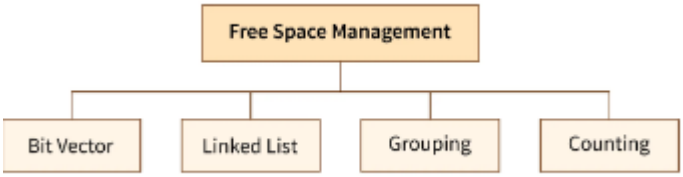


Advantages: Efficient random access.

Disadvantages: Additional overhead for maintaining index blocks.

5. Free space management

A file system is responsible to allocate the free blocks to the file therefore it has to keep track of all the free blocks present in the disk. There are mainly two approaches by using which, the free blocks in the disk are managed. (Efficient free-space management ensures that unused blocks are reused effectively).



a. Bitmap or Bit vector

Each bit represents a block: 1 if allocated, 0 if free.

A Bitmap or Bit Vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: 0 indicates that the block is free and 1 indicates an allocated block. The given instance of disk blocks on the disk in Figure 1 (where green blocks are allocated) can be represented by a bitmap of 16 bits as: 1111000111111001.

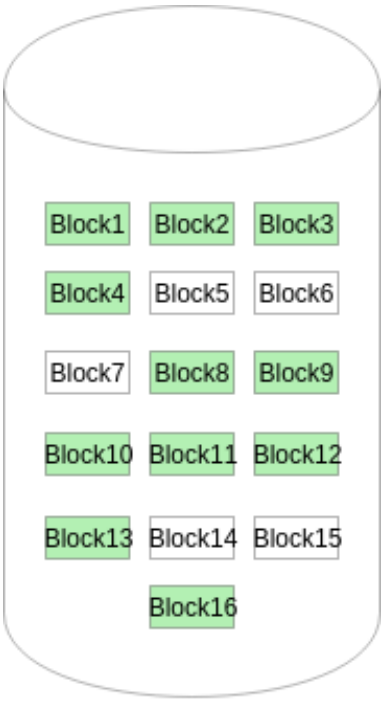


Figure - 1

Advantages:

- Compact representation.
- Fast search using bitwise operations.

Disadvantages:

Requires scanning for contiguous free blocks.

b. Linked List

In this approach, the free disk blocks are linked together i.e. a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in memory.

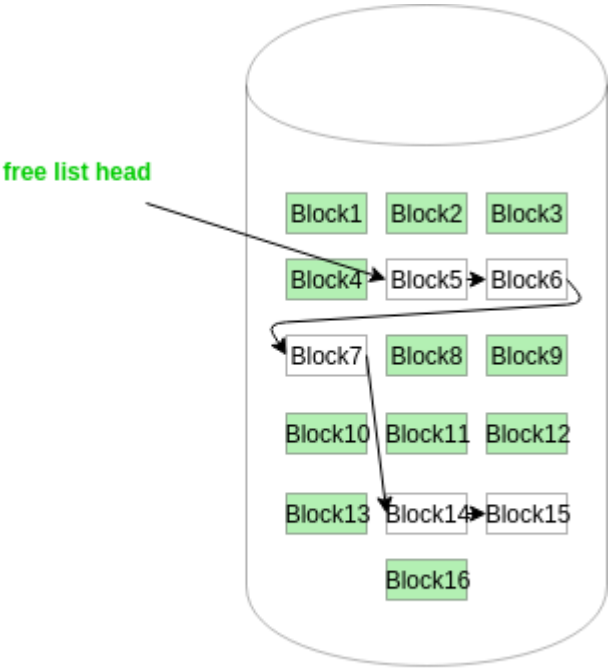


Figure - 2

In Figure-2, the free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of

free list. A drawback of this method is the I/O required for free space list traversal.

Free blocks are linked together.

Advantages:

Simple and dynamic.

Disadvantages:

Slow traversal for large disks.

c) Grouping

Maintains linked lists of free blocks in groups.

This approach stores the address of the free blocks in the first free block. The first free block stores the address of some, say n free blocks. Out of these n blocks, the first n-1 blocks are actually free and the last block contains the address of next free n blocks

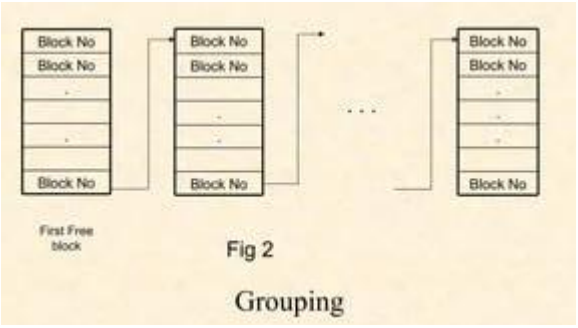


Fig 2

Grouping

Advantages:

Faster allocation than simple linked lists.

Disadvantages:

Slightly more complex.

d) Counting

This approach stores the address of the first free disk block and a number n of free contiguous disk blocks that follow the first block. Every entry in the list would contain:

- Address of first free disk block.
- A number n.

(Tracks contiguous free blocks as ranges).

Advantages:

Efficient for large contiguous free spaces.

Disadvantages:

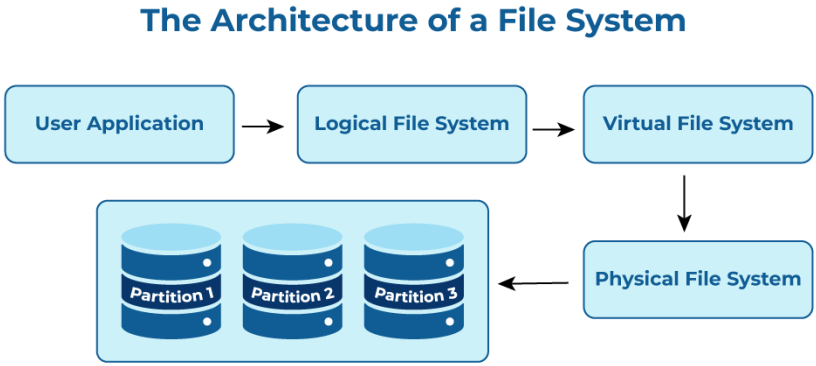
Inefficient for highly fragmented storage.

File-System Internals: File System Mounting, Partitions and Mounting, File Sharing.

Protection: Goals of protection.

5.5 File-System Internals

A file system is a software layer that manages files and folders on a storage device, such as a hard disk or flash memory. It organizes files and directories, and provides a way for users to access them.



Common file systems include:

- Windows: NTFS, FAT32, exFAT.
- Linux: ext4, XFS, Btrfs.
- Mac: APFS, HFS+.

File System Structure

Boot Block: Contains bootstrapping information, necessary to start the operating system.

Superblock: Holds metadata about the file system (e.g., size, block counts, free blocks).

Inode Table: Maintains information about files, such as ownership, permissions, and locations on disk.

Data Blocks: Store the actual content of files.

1. File System Mounting

Definition: File system mounting is the process of making a file system accessible to the operating system and its users. It involves associating a storage device (or partition) with a directory in the system's file hierarchy.

Process:

- Identify the file system type (e.g., NTFS, ext4, FAT32).
- Use a system call (e.g., mount in Unix/Linux) to link the file system to a directory (called a **mount point**).
- The system verifies the file system structure to ensure it's valid and consistent.

Dynamic Mounting: Modern systems allow on-demand mounting, such as when plugging in a USB device.

2. Partitions and Mounting

Partitions: A disk is divided into sections called partitions, each of which can hold a different file system or data structure.

Primary Partition: Can boot the system or hold the primary file system.

Extended Partition: A special partition that can contain multiple logical partitions.

Logical Partition: A partition within an extended partition.

Why Partitions?

- Separate system and user data.
- Enable multi-boot setups.
- Improve data management and security.

Mounting Partitions: Each partition must be mounted to a directory in the file system hierarchy for access. For example, /dev/sda1 might be mounted to /home.

Mounting Indifferent Operating Systems

a. Linux-Unix based OS

We want to mount /dev/sdb1 to an existing directory /mnt.

sudo mount /dev/sdb1 /mnt/mydisk

After mounting, we have to unmount after use

sudo umount /mnt/mydisk

b. Windows OS

In windows mounting is very easy for a user. When we connect the external storage devices, windows automatically detect the file system and mount it to the drive letter. Drive letter may be **D:** or **E:**.

Steps:

- Connect an external storage device to your PC.
- Windows detects the file system on the drive (e.g., FAT32 or NTFS) and assigns it a drive letter, such as "E:".
- You can access the derive by going through, THIS PC --> FILE EXPLORER -->"E:" drive
- Access the data.

c. Mac OS

In Mac OS when we connect an external storage it will automatically mount, and it will be accessible via Finder. As an advanced mounting method user can also use the command **diskutil** in Terminal.

Method 2(Using diskutil):

To mount a drive with a known identifier: **disk2s1**

diskutil mount /dev/disk2s1

To unmount:

diskutil unmount /dev/disk2s1

3. File Sharing

File Sharing in an Operating System(OS) denotes how information and files are shared between different users, computers, or devices on a network; and files are units of data that are stored in a computer in the form of documents/images/videos or any others types of information needed.

For Example: Suppose letting your computer talk to another computer and exchange pictures, documents, or any useful data. This is generally useful when one wants to work on a

project with others, send files to friends, or simply shift stuff to another device. Our OS provides ways to do this like email attachments, cloud services, etc. to make the sharing process easier and more secure.

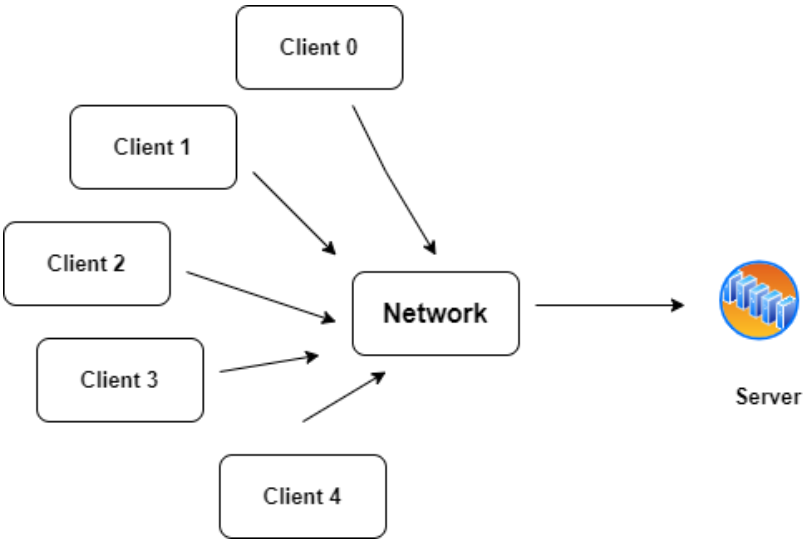
Local File Sharing: Multiple users on the same machine can access shared files with proper permissions.

Remote File Sharing:

a) Network File System (NFS): Allows file access over a network as if it were local.

NFS is a distributed based file sharing protocol mainly used in Linux/Unix based operating System. It allows a computer to share files over a network as if they were based on local. It provides a efficient way of transfer of files between servers and clients.

Example: Many Programmer/Universities/Research Institution uses Unix/Linux based Operating System. The Institutes puts up a global server datasets using NFS. The Researchers and students can access these shared directories and everyone can collaborate on it.

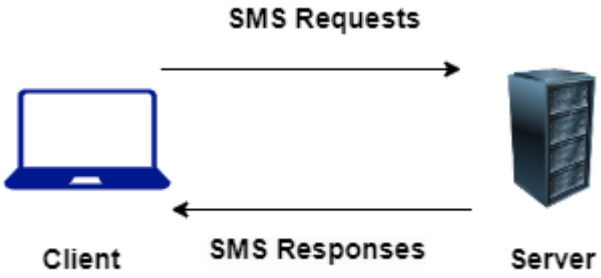


b)Server Message Block (SMB): Used for sharing files between Windows systems and others.

SMB is like a network based file sharing protocol mainly used in windows operating systems. It allows our computer to share files/printer on a network. SMB is now the standard way for seamless file transfer method and printer sharing.

Example: Imagine in a company where the employees have to share the files on a particular project . Here SMB is employed to share files among all the windows based operating system.orate on projects. SMB/CIFS is employed

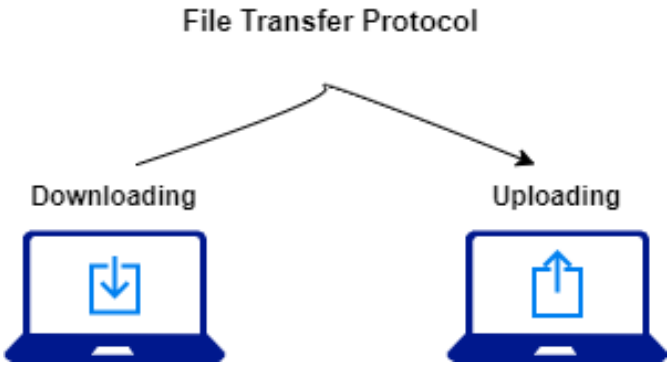
to share files between Windows-based computers. Users can access shared folders on a server, create, modify, and delete files.



3. File Transfer Protocol (FTP)

It is the most common standard protocol for transferring of the files between a client and a server on a computer network. FTPs supports both uploading and downloading of the files, here we can download,upload and transfer of files from Computer A to Computer B over the internet or between computer systems.

Example: Suppose the developer makes changes on the server. Using the FTP protocol, the developer connects to the server they can update the server with new website content and updates the existing file over there.



4. Cloud-Based File Sharing

It involves the famous ways of using online services like Google Drive, DropBox , One Drive ,etc. Any user can store files over these cloud services and they can share that with others, and providing access from many users. It includes collaboration in realtime file sharing and version control access.

Example: Several students working on a project and they can use Google Drive to store and share for that purpose. They can access the files from any computer or mobile devices and they can make changes in realtime and track the changes over there.

Challenges:

- Synchronization of access.
- Security and access control.
- Latency in networked environments.

5.6 Protection: Goals of protection

Protection in operating systems refers to mechanisms that control access to resources and ensure that users and programs interact with system resources in a controlled, safe, and predictable manner. The goals of protection are fundamental to maintaining system integrity, confidentiality, and availability. These goals include:

Goals of Protection

1. Preventing Unauthorized Access

- Ensure that only authorized users, processes, and programs can access system resources (e.g., files, memory, devices).
- Protect against malicious or accidental misuse of resources.
- Example: A payroll file should only be accessible to authorized HR personnel.

2. Ensuring Data Integrity

- Protect data from unauthorized modification or corruption.
- Prevent processes from altering critical system data or another user's data.
- Example: A banking application should ensure that only valid transactions modify account balances.

3. Confidentiality

- Ensure sensitive information is not disclosed to unauthorized users or processes.
- Protect secrets like passwords, personal data, and proprietary information.

- Example: Encryption mechanisms ensure that intercepted communication remains confidential.

4. Controlled Access

- Regulate access to resources based on predefined policies.
- Access should be granted based on principles like need-to-know or least privilege.
- Example: A user with "read" permission should not be able to delete files.

5. Isolation of Resources

- Prevent one process or user from interfering with others.
- Ensure that a failure in one part of the system does not compromise the entire system.
- Example: Sandboxing isolates untrusted code to prevent it from affecting the broader system.

6. Accountability

- Keep a record of actions performed by users and processes.
- Detect and track malicious or unauthorized activity.
- Example: Audit logs can show which user modified a file and when.

7. Availability

- Ensure that legitimate users have uninterrupted access to resources when needed.
- Protect against Denial-of-Service (DoS) attacks and resource starvation.
- Example: A web server must serve pages to authorized users even under heavy load.

8. Flexibility

- Support dynamic adjustments to protection policies as system requirements evolve.
- Example: Temporarily grant administrative access to a user during maintenance.

9. Enforcement of Policy

- Ensure that all resource accesses strictly follow the specified security and access control policies.
- Example: A system should deny a user's attempt to modify a read-only file, regardless of the method used.

Mechanisms for Achieving Protection Goals

Authentication: Verify the identity of users or processes (e.g., passwords, biometrics).

Authorization: Define permissions and determine what actions are allowed.

Access Control: Implement mechanisms like Access Control Lists (ACLs) or Role-Based Access Control (RBAC).

Encryption: Protect data confidentiality during storage and transmission.

Auditing: Log system activities to detect and prevent violations.