III Semester
Course 8: Operating Systems

UNIT – I

What is an Operating System (OS)?

An **Operating System (OS)** is **system software** that acts as an **interface** between computer hardware and the user. It manages hardware resources and provides services for computer programs.

Functions of an Operating System:

1. **Process Management** – Manages execution of processes, scheduling, and multitasking.
2. **Memory Management** – Allocates and deallocates memory for processes.
3. **File System Management** – Organizes and controls file storage and access.
4. **Device Management** – Controls and communicates with input/output devices.
5. **User Interface** – Provides CLI (Command Line Interface) or GUI (Graphical User Interface).
6. **Security and Protection** – Manages user authentication, file permissions, and system security.

Examples of Operating Systems:

- **Windows** (Windows 10, 11)
- **Linux** (Ubuntu, Fedora)
- **MacOS**
- **Android** (Mobile OS)
- **iOS** (Apple's Mobile OS)

History and Evolution of Operating Systems

Operating Systems (OS) have evolved significantly over time, adapting to new hardware and computing needs. The evolution can be divided into different generations:

1st Generation (1940s – Early 1950s): No Operating System

- Computers were **vacuum tube-based** and programmed using machine language.
- No OS existed; users manually operated the machine using switches and punch cards.
- **Example Machines:** ENIAC, UNIVAC

2nd Generation (1950s – Early 1960s): Batch Processing Systems

- **Introduction of OS:** First OS was developed to automate job execution.
- **Batch Processing:** Programs were grouped into batches and executed sequentially.
- No direct interaction between users and computers.
- **Example OS:** IBM's GM-NAA I/O

3rd Generation (1960s – 1970s): Multiprogramming & Time-Sharing

- **Multiprogramming:** OS could run multiple programs at the same time by switching between them.
- **Time-Sharing:** Allowed multiple users to interact with the system simultaneously.
- **Introduction of UNIX (1969):** First portable OS, developed at Bell Labs.
- **Examples:** UNIX, IBM OS/360

4th Generation (1980s – 1990s): Personal Computers (PC) & GUI-Based OS

- **Personal Computers (PCs) became common**, leading to the development of user-friendly OS.
- **Graphical User Interface (GUI):** First used in Xerox Alto, later popularized by Windows and MacOS.
- **Networking & Multi-user OS:** OS started supporting networking capabilities.
- **Examples:** MS-DOS, Windows 3.1, MacOS, Linux

5th Generation (2000s – Present): Mobile, Cloud, and AI-Driven OS

- **Mobile OS:** Android, iOS revolutionized smartphones.
- **Cloud Computing:** OS supports virtualization (e.g., AWS, Google Cloud).
- **AI and Automation:** OS includes AI-powered assistants (e.g., Cortana, Siri).
- **Examples:** Windows 10/11, Linux, Android, iOS, macOS

Future Trends

- **Quantum Computing OS** (for quantum processors).
- **AI-driven OS** (self-healing and adaptive).
- **Edge Computing OS** (optimized for IoT and real-time processing).

Basic Functions of an Operating System (OS)

An **Operating System (OS)** performs several essential functions to manage computer hardware and software efficiently.

---

1. Process Management

- Creates, schedules, and terminates processes.
- Manages CPU scheduling (e.g., FCFS, Round Robin).
- Handles **multitasking** and **multi-threading**.

🛠 **Example:** Running multiple applications simultaneously (browser, music player).

---

2. Memory Management

- Allocates and deallocates memory to processes.
- Manages RAM usage and virtual memory.
- Uses paging and segmentation techniques.

🛠 **Example:** Swapping inactive processes to disk to free up RAM.

---

3. File System Management

- Organizes and manages files and directories.
- Provides file permissions and access control.
- Supports different file systems (e.g., NTFS, FAT32, ext4).

🛠 **Example:** Storing, renaming, or deleting files on a hard drive.

---

4. Device Management

- Controls input/output (I/O) devices like keyboards, printers, and hard drives.
- Uses **device drivers** to communicate with hardware.
- Implements **I/O scheduling** to optimize performance.

🛠 **Example:** Detecting and configuring a USB drive automatically.

---

5. User Interface Management

- Provides a **Graphical User Interface (GUI)** or **Command Line Interface (CLI)**.
- Manages user interactions with the system.

🛠 **Example:** Windows Explorer (GUI) or Linux Terminal (CLI).

---

6. Security and Access Control

- Manages user authentication and permissions.
- Protects against viruses, malware, and unauthorized access.
- Implements encryption and firewall mechanisms.

🛠 **Example:** Login authentication using passwords or biometrics.

---

7. Networking and Communication

- Manages network connections (Wi-Fi, LAN, VPN).
- Supports protocols like TCP/IP for internet communication.
- Allows remote access to resources.

🛠 **Example:** Connecting to a Wi-Fi network and accessing the internet.

---

8. Error Detection and Handling

- Detects and recovers from system errors (e.g., memory overflow, disk failure).
- Logs system errors for troubleshooting.

🛠 **Example:** Blue Screen of Death (BSOD) in Windows when a critical error occurs.

Conclusion

The OS is the backbone of a computer, managing hardware, processes, and users efficiently.

Types of Operating Systems (OS)

Operating Systems can be classified based on their functionality, user interaction, and system requirements. Here are the major types:

---

1. Batch Operating System

- **Jobs are processed in batches without user interaction.**
- Users submit jobs, and the OS executes them sequentially.
- Efficient for large-scale processing but lacks real-time execution.

🛠 **Example:** IBM's early batch OS, punched card-based systems.

---

2. Time-Sharing Operating System

- **Multiple users share CPU time simultaneously.**
- Uses **CPU scheduling** and **multiprogramming** to ensure responsiveness.
- Improves system utilization and allows interactive computing.

🛠 **Example:** UNIX, Linux

---

3. Distributed Operating System

- **Manages multiple computers as a single system.**
- Nodes communicate over a network and share resources.
- Provides fault tolerance and load balancing.

🛠 **Example:** Amoeba, Windows Server, Google's Cluster OS

---

4. Real-Time Operating System (RTOS)

- **Processes tasks with strict time constraints.**
- Used in mission-critical applications like aerospace, robotics, and medical devices.
- **Types:**
  - **Hard RTOS:** Ensures deadlines are met strictly (e.g., missile guidance systems).
  - **Soft RTOS:** Prioritizes tasks but allows flexibility (e.g., video streaming).

🛠 **Example:** VxWorks, QNX, RTLinux

---

5. Network Operating System (NOS)

- **Manages network resources and communication.**
- Provides file sharing, user authentication, and security.
- Supports multiple connected systems over LAN/WAN.

🛠 **Example:** Windows Server, Linux Server, Novell NetWare

---

6. Mobile Operating System

- **Designed for smartphones and tablets.**
- Optimized for touch input, sensors, and power efficiency.
- Supports app ecosystems and cloud services.

🛠 **Example:** Android, iOS

---

7. Embedded Operating System

- **Used in specialized hardware systems (e.g., IoT devices, ATMs, medical equipment).**
- Small footprint and optimized for specific functions.

🛠 **Example:** FreeRTOS, Embedded Linux, Windows CE

---

8. Multiprocessing Operating System

- **Uses multiple CPUs for parallel execution of processes.**
- Increases system performance and reliability.

🛠 **Example:** Linux, Windows with multi-core CPUs

---

9. Multi-User Operating System

- **Allows multiple users to access the system simultaneously.**
- Implements user authentication and resource sharing.

🛠 **Example:** UNIX, Windows Server

---

10. Cloud Operating System

- **Runs on cloud infrastructure and provides virtualization.**
- Manages cloud resources and supports distributed computing.

🛠 **Example:** Google Chrome OS, Microsoft Azure OS

Conclusion

Each OS type serves a different purpose, from real-time processing to cloud computing.

Multiprogramming Systems

What is Multiprogramming?

Multiprogramming is an operating system technique that allows multiple programs (or processes) to reside in **main memory (RAM)** at the same time and execute concurrently. The CPU switches between these programs to improve system utilization and efficiency.

📌 **Key Idea: When one program waits for I/O, the CPU executes another program, ensuring that the CPU is never idle.**

How Multiprogramming Works?

1. Multiple programs are loaded into memory.
2. The OS selects one process and assigns CPU time.
3. If the process needs I/O (e.g., reading from disk), the CPU switches to another process.
4. This cycle continues, ensuring maximum CPU utilization.

Advantages of Multiprogramming

✅ **Increased CPU Utilization** – The CPU never remains idle, as it switches between processes.
✅ **Faster Execution** – Multiple programs run concurrently, reducing waiting time.

✅ **Better Resource Utilization** – Memory, CPU, and I/O devices are efficiently used.
✅ **Improved System Throughput** – More jobs are completed in less time.

Disadvantages of Multiprogramming

❌ **Complex OS Design** – Requires advanced CPU scheduling and memory management.
❌ **Higher Memory Usage** – Multiple processes need to be stored in RAM.
❌ **CPU Overhead** – Context switching between processes consumes processing time.
❌ **Deadlock & Starvation Risks** – Poor scheduling can cause deadlocks or indefinite waiting.

Multiprogramming vs. Multitasking

Feature	Multiprogramming	Multitasking
Definition	Multiple programs in memory; CPU executes one at a time	Multiple tasks running simultaneously
User Interaction	No direct user interaction	Supports interactive users
Context Switching	Occurs only when I/O happens	Frequent context switching
Example OS	Early UNIX, IBM OS/360	Windows, Linux, MacOS

Examples of Multiprogramming OS

- **IBM OS/360** – One of the first multiprogramming OS.
- **UNIX** – Supports process scheduling and memory management.
- **Modern OS (Windows, Linux, macOS)** – Use multiprogramming alongside multitasking.

Conclusion

Multiprogramming improves system efficiency by keeping the CPU busy with multiple processes. It is the foundation of modern OS, enabling multitasking, multiprocessing, and time-sharing features.

Batch Operating Systems

What is a Batch Operating System?

A **Batch Operating System** is an early type of OS where jobs (tasks) are grouped together (batch processing) and executed **sequentially** without direct user interaction.

📌 **Key Idea:** The system processes batches of jobs automatically, improving efficiency in environments where user interaction is minimal.

How Batch Processing Works?

1. Users submit jobs (e.g., payroll processing, bank transactions) to an operator.
2. The operator groups similar jobs into a **batch** and loads them into the system.
3. The OS executes the batch sequentially without interruption.
4. Output is generated and returned to the user after processing.

Features of Batch Operating System

✅ **Automatic Job Execution** – Jobs run without manual intervention.
✅ **Job Scheduling** – The OS schedules jobs to optimize resource usage.
✅ **Efficient for Repetitive Tasks** – Best for large-scale repetitive operations.
✅ **Uses Spooling** – Jobs are stored in disk queues before execution.

Advantages of Batch Operating System

✓ **Better Resource Utilization** – CPU and memory are used efficiently.
✓ **Reduces Idle Time** – The system executes jobs continuously.
✓ **Handles Large Jobs Efficiently** – Ideal for banking, payroll, and billing systems.
✓ **Less Manual Intervention** – Reduces human errors in job execution.

Disadvantages of Batch Operating System

❌ **No Direct User Interaction** – Users must wait for job completion.
❌ **Slow Debugging** – Errors are found after execution, requiring re-processing.
❌ **High Turnaround Time** – Some jobs may have to wait long before execution.
❌ **Requires Job Scheduling** – The system must efficiently manage job queues.

Examples of Batch Operating Systems

- **IBM OS/360** – Early batch processing OS used in mainframes.
- **UNIX (in early days)** – Used batch processing for running jobs.
- **Windows Task Scheduler** – Automates background batch jobs.

Comparison: Batch OS vs. Modern OS

Feature	Batch OS	Modern OS (Windows, Linux)
User Interaction	None	Interactive
Job Processing	Sequential (one batch at a time)	Parallel & real-time
Error Handling	After job completion	Real-time error handling
Example Use Case	Payroll, billing, banking	Personal computing, gaming

---

Conclusion

Batch Operating Systems were crucial in the early computing era, mainly used for bulk data processing. While modern OS have replaced them for general computing, batch processing is still used in **large-scale data processing, banking, and automation tasks.**

Time-Sharing Operating Systems

What is a Time-Sharing System?

A **Time-Sharing Operating System** allows multiple users to share the CPU simultaneously by **allocating small time slices (quantums)** to each process. It enables **interactive computing**, where users can interact with programs in real time.

📌 **Key Idea: The CPU rapidly switches between tasks, giving users the illusion of concurrent execution.**

How Time-Sharing Works?

1. **Multiple users/programs** are loaded into memory.
2. The OS assigns each process a **time slice** (quantum).

3. **CPU switches** between processes using **context switching**.
4. If a process is not completed, it is moved back to the **ready queue** and will resume execution in the next cycle.

Features of Time-Sharing Systems

☑ **Multi-user support** – Many users can work on the system simultaneously.
☑ **Preemptive Scheduling** – The OS switches tasks based on a time quantum.
☑ **Interactive Processing** – Users receive instant feedback from their applications.
☑ **Resource Sharing** – CPU, memory, and I/O devices are shared among users.

Advantages of Time-Sharing OS

✓ **Efficient CPU Utilization** – The CPU is not idle and always processes tasks.
✓ **Faster Response Time** – Users receive quick responses from running programs.
✓ **Minimizes CPU Idle Time** – The system continuously executes processes.
✓ **Better User Experience** – Allows multiple users to interact simultaneously.

Disadvantages of Time-Sharing OS

✕ **Overhead of Context Switching** – Switching between processes adds processing delay.
✕ **Security and Data Integrity Issues** – Multiple users sharing resources can lead to security risks.
✕ **Complex OS Design** – Requires advanced scheduling algorithms.
✕ **Response Time Variation** – If too many processes run, performance may degrade.

Examples of Time-Sharing Operating Systems

- **UNIX** – One of the first time-sharing OS.
- **Linux** – Supports multi-user and multi-tasking.
- **Windows Server** – Supports multiple remote desktop users.

Comparison: Time-Sharing vs. Other Systems

Feature	Time-Sharing OS	Batch OS	Multiprogramming OS

Feature	Time-Sharing OS	Batch OS	Multiprogramming OS
User Interaction	Yes (interactive)	No	Limited
Job Execution	Concurrent execution with time slices	Sequential execution	Multiple programs in memory
Response Time	Fast	Slow	Faster than batch but not interactive
Example OS	UNIX, Linux, Windows Server	IBM OS/360	Early UNIX, Windows

---

Conclusion

Time-Sharing OS revolutionized computing by enabling **interactive, multi-user environments**, forming the foundation of modern operating systems. Today's OS (Windows, Linux, macOS) integrate time-sharing with multitasking and multiprocessing to enhance performance.

Operating Systems for Personal Computers (PCs)

An **Operating System (OS)** for personal computers is software that manages hardware resources and provides essential services for applications and users. It supports multitasking, user interaction, and system security.

1️⃣ Types of PC Operating Systems

1. Single-User, Single-Task OS

- Supports only **one user and one task** at a time.
- Example: **MS-DOS (Microsoft Disk Operating System)**

2. Single-User, Multi-Tasking OS

- Supports **one user but multiple tasks** at once.
- Example: **Windows, macOS, Linux**

3. Multi-User OS

- Supports **multiple users** accessing the system simultaneously.
- Example: **Windows Server, UNIX, Linux**

4. Real-Time OS (RTOS)

- Used for applications that require **immediate response** and precise timing.
- Example: **QNX, RTLinux**

---

2 Popular PC Operating Systems

1. Windows OS

📌 **Developed by:** Microsoft
📌 **Latest Version:** Windows 11
✅ **Features:**

- User-friendly **Graphical User Interface (GUI)**
- Supports multitasking, gaming, and business applications
- Strong software compatibility
  ❌ **Cons:** Requires frequent updates and high system resources

2. macOS

📌 **Developed by:** Apple
📌 **Latest Version:** macOS Sonoma
✅ **Features:**

- Optimized for Apple devices (**MacBook, iMac**)
- Secure, stable, and efficient **performance**
- High-quality **graphics and UI**
  ❌ **Cons:** Works only on Apple hardware

3. Linux OS

📌 **Developed by:** Open-source community
📌 **Popular Versions:** Ubuntu, Fedora, Debian
✅ **Features:**

- Free and **open-source**
- High security and **customization**
- Runs on low-end as well as high-end PCs
  ❌ **Cons:** Learning curve for beginners

4. UNIX

📌 **Developed by:** AT&T Bell Labs
✅ **Features:**

- Multi-user and multitasking capabilities
- Mostly used in **servers and workstations**
- High stability and security

❌ **Cons:** Not commonly used for personal computing

---

3 Comparison of PC Operating Systems

Feature	Windows	macOS	Linux	UNIX
User-Friendliness	☆☆☆☆☆	☆☆☆☆	☆☆☆	☆☆
Security	☆☆☆	☆☆☆☆	☆☆☆☆☆	☆☆☆☆☆
Customization	☆☆	☆	☆☆☆☆☆	☆☆☆
Software Availability	☆☆☆☆☆	☆☆☆	☆☆☆	☆☆
Cost	Paid	Paid	Free (Open-Source)	Mostly Paid

---

4 Conclusion

Operating Systems for personal computers provide different features based on user needs.

- **Windows** is best for general users and gaming.
- **macOS** is ideal for Apple ecosystem users.
- **Linux** is great for developers and security-conscious users.
- **UNIX** is mainly used in professional and server environments.

Operating Systems for Workstations and Handheld Devices

Operating systems are designed based on the type of computing device they run on. Workstations and handheld devices have specialized OS optimized for performance, efficiency, and user experience.

---

1 Workstation Operating Systems

What is a Workstation?

A **workstation** is a high-performance computer designed for professional or technical applications such as graphics rendering, engineering, scientific computing, and software development.

Features of Workstation OS:

- ✅ **High processing power** – Supports advanced computations and simulations.
- ✅ **Multi-user & multitasking** – Enables concurrent processes and users.
- ✅ **Security & Stability** – Designed for enterprise use with strong security features.
- ✅ **Supports advanced hardware** – Optimized for GPUs, multiple CPUs, and large memory.

Popular Workstation Operating Systems:

OS Name	Description
Windows 10/11 Pro & Enterprise	Business-oriented OS with advanced security & virtualization.
macOS (Mac Pro, iMac Pro)	Optimized for creative professionals and software developers.
Linux (Ubuntu, Fedora, CentOS)	Preferred for programming, AI, and scientific computing.
UNIX (Solaris, AIX)	Used in high-end research, industrial, and financial applications.

---

2 Operating Systems for Handheld Devices

What are Handheld Devices?

Handheld devices include **smartphones, tablets, PDAs, and embedded systems** like smartwatches and IoT devices. Their OS is designed for **low power consumption, touch input, and real-time communication.**

Features of Handheld OS:

- ✅ **Lightweight & optimized for mobility**
- ✅ **Touchscreen & voice recognition support**
- ✅ **Battery efficiency & power management**
- ✅ **App store & software ecosystem**
- ✅ **Seamless connectivity (Wi-Fi, Bluetooth, 4G/5G, NFC)**

Popular Handheld Operating Systems:

OS Name	Device Type	Description
Android	Smartphones, Tablets, Smart TVs	Open-source, customizable OS by Google.
iOS	iPhones, iPads	Secure, optimized OS for Apple devices.
iPadOS	Tablets (iPad)	A tablet-optimized version of iOS.
watchOS	Smartwatches (Apple Watch)	Designed for wearable Apple devices.
Wear OS	Smartwatches (Google)	Google's OS for smartwatches.
HarmonyOS	Huawei Devices	Developed by Huawei as an alternative to Android.
KaiOS	Feature Phones	A lightweight OS for smart feature phones.
Windows Mobile (Discontinued)	Older Windows-based smartphones	Discontinued but was used in business devices.

---

3 Comparison: Workstation OS vs. Handheld OS

Feature	Workstation OS	Handheld OS
Processing Power	High (supports complex computations)	Low to Medium (optimized for efficiency)
User Input	Keyboard & mouse	Touchscreen, voice, gestures
Battery Dependency	Not a concern (plugged in)	Crucial (optimized for power saving)

Feature	Workstation OS	Handheld OS
Multitasking	Advanced (supports high-end tasks)	Limited (optimized for mobile apps)
Software Ecosystem	Professional applications (CAD, AI, Development)	Mobile apps (social media, productivity, gaming)
Connectivity	Ethernet, Wi-Fi	Wi-Fi, 4G/5G, Bluetooth, NFC

---

4 Conclusion

- **Workstation OS** is designed for power users who require **high performance, security, and stability.**
- **Handheld OS** is optimized for **portability, battery efficiency, and touchscreen interfaces.**

Process Control & Real-Time Systems

1 Process Control in Operating Systems

What is Process Control?

Process control refers to how an **Operating System (OS)** manages and schedules processes (running programs) efficiently. It ensures proper execution, synchronization, and communication between processes.

Key Functions of Process Control:

✅ **Process Creation & Termination** – Starting and stopping processes.
✅ **Process Scheduling** – Allocating CPU time using scheduling algorithms.
✅ **Inter-Process Communication (IPC)** – Allowing processes to share data.
✅ **Synchronization & Deadlock Handling** – Managing shared resources.
✅ **Process States** – Tracking execution status (Ready, Running, Waiting, etc.).

Process States in OS:

1. **New** – Process is being created.
2. **Ready** – Process is waiting for CPU time.
3. **Running** – Process is executing.
4. **Waiting** – Process is waiting for I/O or another event.
5. **Terminated** – Process has completed execution.

Types of Process Scheduling:

- **Long-Term Scheduler** – Controls job admission to the system.
- **Short-Term Scheduler** – Selects processes for CPU execution.
- **Medium-Term Scheduler** – Manages swapping of processes to and from memory.

---

2 Real-Time Systems in Operating Systems

What is a Real-Time System?

A **Real-Time Operating System (RTOS)** is an OS designed to **process data and execute tasks within a strict time limit.** It is commonly used in embedded systems, industrial automation, robotics, medical devices, and telecommunications.

Types of Real-Time Systems:

1. **Hard Real-Time System**
   - Guarantees task completion **within a strict deadline**.
   - Example: **Aircraft control systems, medical devices, industrial automation**.
2. **Soft Real-Time System**
   - Prioritizes tasks but **allows some flexibility** in deadlines.
   - Example: **Multimedia streaming, online transaction processing, gaming**.
3. **Firm Real-Time System**
   - Missing deadlines **does not cause failure**, but it reduces system efficiency.
   - Example: **Weather forecasting, video conferencing**.

Key Features of Real-Time Operating Systems (RTOS):

✅ **Deterministic** – Ensures predictable task execution.
✅ **Low Latency** – Quick response to external inputs.
✅ **Priority Scheduling** – Higher priority tasks execute first.
✅ **Concurrency Control** – Handles multiple real-time tasks simultaneously.
✅ **Efficient Memory Management** – Optimized for minimal delays.

---

3 Comparison: Process Control vs. Real-Time Systems

Feature	Process Control	Real-Time Systems
Main Focus	Managing multiple processes	Ensuring real-time task execution
Response Time	May vary (depends on scheduling)	Strict timing constraints

Feature	Process Control	Real-Time Systems
Use Case	General-purpose OS (Windows, Linux)	Embedded systems, automation
Scheduling Algorithm	Round Robin, Priority Scheduling	Earliest Deadline First (EDF), Rate Monotonic
Examples	Process management in OS (Windows, Linux)	Medical devices, robotics, flight control

---

4 Conclusion

- **Process Control** manages process execution, scheduling, and synchronization in general-purpose OS.
- **Real-Time Systems** ensure timely execution of tasks, crucial for time-sensitive applications like aviation, healthcare, and automation.

UNIT – II

Processor and User Modes

In an **Operating System (OS)**, **Processor and User Modes** refer to different privilege levels that determine how a process can interact with system resources. These modes help in system protection and security by restricting direct access to hardware and critical system operations.

---

1. Processor Modes (Execution Modes)

These modes determine the privilege level of a process executing on the CPU.

(a) User Mode

- A process runs in **User Mode** when executing application-level code.
- It has **restricted access** to system resources (e.g., it cannot directly access hardware or execute privileged instructions).
- If a user-mode process needs to perform a **privileged operation**, it must request the OS through a **system call**.
- If a user-mode process attempts a privileged operation, it results in a **trap (exception)**.

(b) Kernel Mode (Supervisor Mode / System Mode)

- The OS runs in **Kernel Mode** with full access to hardware and system resources.
- It can execute **privileged instructions**, perform **I/O operations**, and manage memory.
- When a user process requests a system service, the mode switches from **User Mode to Kernel Mode** using a **system call** (e.g., file operations, process management).
- Kernel mode is necessary for handling **interrupts, process scheduling, and device management**.

---

2. Mode Switching

Mode switching occurs in the following cases:

1. **System Call (User → Kernel Mode)**
    - When a process in **User Mode** requests a privileged operation (e.g., reading a file), the CPU switches to **Kernel Mode**.
2. **Interrupts (User → Kernel Mode)**
    - If an **interrupt** (hardware or software) occurs, the CPU switches to **Kernel Mode** to handle the event.
3. **Exception Handling (User → Kernel Mode)**
    - If an **exception** (e.g., divide by zero) occurs, the CPU switches to **Kernel Mode** for error handling.

4. **Process Context Switch (Kernel → User Mode)**
    - After handling a request, the CPU switches back to **User Mode** to continue user process execution.

---

3. Dual-Mode and Multi-Mode Operations

- **Dual-Mode (User Mode & Kernel Mode)** is common in general-purpose OS like Windows, Linux.
- **Multi-Mode** includes additional modes (e.g., Hypervisor Mode for virtualization).

Summary Table

Mode	Privileges	Access to Hardware	Privileged Instructions
User Mode	Limited	No	No
Kernel Mode	Full	Yes	Yes

Conclusion

Processor modes ensure **system security and stability** by restricting user applications from directly accessing critical system resources. The **mode switching mechanism** allows controlled execution of privileged operations via system calls.

Kernels in Operating Systems

A **kernel** is the **core component** of an operating system (OS) that manages system resources and enables communication between hardware and software. It runs in **Kernel Mode**, having full control over the system.

---

1. Functions of a Kernel

The kernel provides several essential services:

1. **Process Management** – Schedules processes, handles multitasking, and allocates CPU time.
2. **Memory Management** – Manages RAM allocation and virtual memory.
3. **Device Management** – Communicates with hardware through drivers.
4. **File System Management** – Handles file operations and storage access.
5. **System Call Handling** – Provides an interface for user applications to access hardware safely.

---

2. Types of Kernels

Kernels are classified based on their architecture and functionality:

(a) Monolithic Kernel

- The entire OS runs as a single large program in **Kernel Mode**.
- All services (process management, memory, file systems, device drivers) run in the kernel space.
- Faster but **less secure** due to direct communication between components.
- Example: **Linux, UNIX, MS-DOS**

🔷 **Advantages**: High performance, direct access to hardware.
🔷 **Disadvantages**: A bug in one module can crash the entire system.

(b) Microkernel

- Only **essential** functions (like CPU scheduling and memory management) run in **Kernel Mode**.
- Other services (like device drivers, file systems, and network management) run in **User Mode** as separate processes.
- More **secure and stable** but slightly slower due to inter-process communication (IPC).
- Example: **MINIX, QNX, macOS X (partially)**

🔷 **Advantages**: Improved security, easier maintenance.
🔷 **Disadvantages**: Performance overhead due to IPC.

(c) Hybrid Kernel

- Combines **Monolithic and Microkernel** approaches.
- Some services run in **Kernel Mode** (like monolithic kernels), while others run in **User Mode** (like microkernels).
- Used in modern OS for a balance between **performance and stability**.
- Example: **Windows NT, macOS, Linux (modular approach)**

🔷 **Advantages**: Good balance of speed and security.
🔷 **Disadvantages**: Increased complexity.

(d) Exokernel

- Minimalist kernel that provides only hardware abstraction.
- Applications directly manage system resources with minimal kernel interference.
- Used in specialized systems and research.
- Example: **MIT Exokernel, Nemesis OS**

🔷 **Advantages**: Maximum flexibility and efficiency.
🔷 **Disadvantages**: Complex to program, lacks general-purpose usability.

(e) Nanokernel

- A smaller version of the microkernel that provides only minimal hardware control.
- Example: Some **real-time operating systems (RTOS)**.

---

3. Kernel Space vs. User Space

- **Kernel Space**: Runs privileged operations and has full access to hardware.
- **User Space**: Runs applications with restricted privileges.

---

4. Kernel Mode and System Calls

- **System Calls** allow user applications to request services from the kernel.
- Examples: `open()`, `read()`, `write()`, `fork()`, `exec()`.

Summary Table

Kernel Type	Speed	Security	Stability	Examples
Monolithic	✅ Fast	❌ Low	❌ Moderate	Linux, UNIX
Microkernel	❌ Slower	✅ High	✅ High	MINIX, QNX
Hybrid	✅ Fast	✅ Moderate	✅ High	Windows NT, macOS
Exokernel	✅ Fastest	❌ Low	❌ Low	MIT Exokernel
Nanokernel	✅ Fast	✅ High	✅ High	RTOS

Conclusion

The **kernel is the heart of an OS**, responsible for resource management and hardware interaction. Different types of kernels offer **trade-offs between performance, security, and stability** depending on system requirements.

System Calls and System Programs in Operating Systems

1. System Calls

A **System Call** is a mechanism that allows user-level processes to request services from the **operating system's kernel**. Since user programs run in **User Mode** with restricted access, they need **System Calls** to perform privileged operations like accessing files, communicating with hardware, and managing processes.

Types of System Calls

System calls are categorized based on their functions:

1. **Process Control**
    - Create, terminate, and manage processes.
    - Examples:
        - `fork()` – Create a new process.
        - `exit()` – Terminate a process.
        - `exec()` – Replace the current process with another program.
        - `wait()` – Wait for a child process to complete.
2. **File Management**
    - Create, delete, read, write, open, and close files.
    - Examples:
        - `open()` – Open a file.
        - `read()` – Read data from a file.
        - `write()` – Write data to a file.
        - `close()` – Close a file.
        - `unlink()` – Delete a file.
3. **Device Management**
    - Request and release access to hardware devices.
    - Examples:
        - `ioctl()` – Configure device parameters.
        - `read()` – Read from a device.
        - `write()` – Write to a device.
4. **Information Maintenance**
    - Get and set system information, such as time and system limits.
    - Examples:

        - `getpid()` – Get the process ID.
        - `gettimeofday()` – Get system time.
5. **Communication (Inter-Process Communication, IPC)**
    - Exchange data between processes.
    - Examples:
        - `pipe()` – Create a communication channel.
        - `shmget()` – Allocate shared memory.
        - `send()` / `recv()` – Send/receive messages in a network.

System Call Execution Process

1. A **user process** requests an OS service using a **library function** (e.g., `printf()`, which internally calls `write()`).
2. The **library function invokes a system call** (e.g., `sys_write()`).
3. The **CPU switches from User Mode to Kernel Mode**.
4. The **Kernel processes the request**.
5. The CPU **switches back to User Mode**, and execution continues.

---

2. System Programs

System Programs are **utility programs** that provide an interface for users to interact with the operating system. These programs use **System Calls** internally.

Types of System Programs

1. **File Management Programs**
    - Example: `cp`, `mv`, `rm`, `ls` (in Linux), `explorer.exe` (in Windows).
2. **Status Information Programs**
    - Display system information.
    - Example: `top`, `ps`, `tasklist`, `uptime`.
3. **Process Management Programs**
    - Example: `kill`, `nice`, `taskmgr.exe`.
4. **Device Management Programs**
    - Example: `Device Manager` (Windows), `lspci`, `lsusb` (Linux).
5. **Communication Programs**
    - Example: `ping`, `telnet`, `ssh`.
6. **Programming Language Support**
    - Example: Compilers (`gcc`, `javac`), Debuggers (`gdb`).

Comparison: System Calls vs. System Programs

Feature	System Calls	System Programs
Definition	Interface to request OS services	Utility programs that provide system access
Mode of Execution	Kernel Mode	User Mode
Usage	Direct interaction with OS	Uses system calls internally
Examples	`open()`, `fork()`, `read()`	`ls`, `cp`, `top`, `ping`

Conclusion

- **System Calls** are the fundamental interface for programs to request OS services.
- **System Programs** provide **user-friendly utilities** that internally use system calls.
- Together, they help users and applications interact with the **OS efficiently**.

System View of the Process and Resources in an Operating System

In an **Operating System (OS)**, processes and resources are fundamental concepts that define how the system executes tasks and manages hardware/software components. The system view provides insights into how processes interact with system resources.

---

1. Process: System View

A **process** is a **program in execution** that requires various system resources such as CPU, memory, files, and I/O devices to function.

Process Components

A process consists of:

- **Code (Text Section)** – The program's executable instructions.
- **Program Counter (PC)** – Keeps track of the next instruction to execute.
- **Stack** – Stores function calls, local variables.
- **Heap** – Stores dynamically allocated memory.
- **Data Section** – Stores global variables.

Process Lifecycle (States)

A process transitions through various states:

1. **New** – Process is created.
2. **Ready** – Waiting for CPU scheduling.
3. **Running** – Currently executing on the CPU.
4. **Blocked (Waiting)** – Waiting for an I/O operation or event.
5. **Terminated** – Process has finished execution.

🔷 **State Transition Example:**
A process moves from **Ready** → **Running** → **Waiting** → **Ready** → **Running** → **Terminated** based on scheduling decisions.

---

2. System View of Resources

A **resource** is any hardware or software component required for process execution. The OS is responsible for allocating and managing these resources.

Types of Resources

1. **CPU** – The primary processing unit executing process instructions.
2. **Memory (RAM, Virtual Memory)** – Stores process instructions and data.
3. **Storage (Hard Disk, SSD)** – Stores files, executable programs.
4. **I/O Devices (Printers, Network, Keyboard, Mouse)** – Facilitates user and process interaction.
5. **Files & Databases** – Provides data storage and retrieval.

Resource Allocation & Management

- **Resource Allocation**: The OS assigns resources to processes based on availability and scheduling policies.
- **Resource Deallocation**: The OS reclaims resources when a process terminates.
- **Deadlock Handling**: The OS manages resource deadlocks using prevention, avoidance, detection, and recovery techniques.

---

3. Process-Resource Interaction

Processes interact with resources in the following ways:

- **Resource Request**: A process requests a resource (e.g., opening a file).
- **Resource Use**: The process utilizes the allocated resource (e.g., reading/writing data).
- **Resource Release**: The process releases the resource when done.

Process-Resource Model

- A **Process Control Block (PCB)** stores process information (state, allocated resources).
- The **OS scheduler** manages CPU time among processes.
- **System Calls** allow processes to request resources from the OS.

Conclusion

The **System View of a Process and Resources** highlights how the OS manages **process execution and resource allocation** efficiently. Proper scheduling and resource management ensure **smooth multitasking, optimal performance, and system stability**.

Process Abstraction in Operating Systems

1. What is Process Abstraction?

Process abstraction is an **operating system concept** that simplifies **process management** by **hiding the complexities** of hardware execution and providing a logical representation of running programs. It enables **multiprogramming and multitasking**, allowing multiple processes to run concurrently.

---

2. Key Aspects of Process Abstraction

(a) Process as an Abstraction of Execution

- A **process** is an **abstract entity** that represents a **running program**.
- The OS provides an **illusion** that each process runs independently, even though they share CPU and memory.
- The OS **manages scheduling and context switching** to enable multitasking.

(b) Virtualization of CPU (Time-Sharing)

- The OS **divides CPU time** among multiple processes using **scheduling algorithms** (e.g., Round Robin, Priority Scheduling).
- Each process thinks it has the CPU to itself due to **context switching**.
- Example: Multiple applications running on a computer appear to be executing simultaneously.

(c) Virtualization of Memory (Address Space Abstraction)

- Each process gets a **virtual address space**, isolating it from others.
- The OS **maps virtual addresses to physical memory** using **paging or segmentation**.
- This abstraction provides **security** and prevents processes from interfering with each other.

(d) Process Control Block (PCB) – The OS's View of a Process

- The OS keeps track of each process using a **Process Control Block (PCB)**, which stores:
    - **Process ID (PID)**
    - **Process state** (New, Ready, Running, Waiting, Terminated)
    - **Program Counter (PC)**
    - **Registers, memory limits, I/O information**

---

3. Advantages of Process Abstraction

✅ **Simplifies application execution** – Applications don't need to manage hardware directly.
✅ **Supports multitasking and concurrency** – Multiple processes can run efficiently.
✅ **Improves system security** – Isolates processes, preventing unauthorized access.
✅ **Enhances portability** – Programs can run on different hardware without modification.

---

4. Process vs. Thread Abstraction

Aspect	Process Abstraction	Thread Abstraction
Definition	Virtualized execution of a program	Lightweight execution unit within a process
Resource Isolation	Has its own memory space	Shares memory with other threads in the same process
Context Switching	Slower (more overhead)	Faster (less overhead)

Conclusion

Process abstraction is **a fundamental OS feature** that simplifies execution by **hiding hardware details**, enabling **efficient process management** and **resource isolation**. It allows modern systems to **support multitasking, process scheduling, and memory management** seamlessly.

Process Hierarchy in Operating Systems

1. What is Process Hierarchy?

Process hierarchy refers to the **parent-child relationship** among processes in an operating system. When a process creates another process, a **hierarchical (tree-like) structure** is formed, where:

- The **creator** is called the **parent process**.
- The **newly created process** is called the **child process**.

This hierarchical relationship helps the OS **manage process execution, termination, and resource allocation** efficiently.

---

2. How a Process Creates Another Process?

A new process is created using **system calls** like:

- **fork()** → Creates a child process (in UNIX/Linux).
- **CreateProcess()** → Used in Windows.

Each child process can create its own child processes, forming a **tree structure** known as the **process tree**.

🔷 Example:

- **Parent:** `init` (PID 1) process in UNIX spawns system processes.
- **Child:** Shell (`bash`) creates user processes like `ls`, `gcc`, `vim`.

---

3. Process Tree Structure

A process hierarchy can be represented as a **tree**, where:

- The **root process** is the first process started by the OS (e.g., `init` in UNIX/Linux).
- Each **parent process** can have multiple child processes.
- Child processes can create their own child processes.

🔷 Example: Process Tree in UNIX/Linux

```
      init (PID 1)
     /     |     \
   bash   sshd    apache
   /  \
 vim   gcc
```

---

4. Characteristics of Process Hierarchy

1. **Parent-Child Relationship**
    - A parent **creates and controls** its child processes.
    - The child inherits **some attributes** (e.g., environment variables, file descriptors).
2. **Resource Sharing**
    - Parent and child processes **may share** memory and files.
    - In some cases, child processes execute independently.
3. **Process Termination**
    - If a **parent terminates**, child processes **may also be terminated** (depends on OS policy).
    - A **child can terminate independently**, but the parent may still keep track of its exit status.
4. **Zombie and Orphan Processes**
    - **Zombie Process**: A child process that **completes execution** but its parent hasn't read its exit status.
    - **Orphan Process**: A child process that **continues execution** after its parent terminates (adopted by `init`).

---

5. Process Hierarchy in Different OS

Operating System	Process Hierarchy Support
UNIX/Linux	Uses a **tree-like hierarchy**, where every process (except `init`) has a parent.
Windows	Uses **process groups**, but processes are not strictly hierarchical.

---

6. Importance of Process Hierarchy

☑ **Efficient process management** – Helps OS track and control processes.
☑ **Resource allocation** – Parent controls resource limits for child processes.
☑ **Security** – Ensures controlled access between parent and child processes.
☑ **Parallel processing** – Enables multitasking through multiple child processes.

Conclusion

Process hierarchy is a key concept in OS that defines how **processes are structured, created, and managed**. It helps in **process tracking, resource sharing, and system stability**, forming the foundation for **multitasking and concurrency**.

Threads in Operating Systems

1. What is a Thread?

A **thread** is the **smallest unit of execution** within a process. It represents an **individual sequence of execution** that runs within a process and shares resources like memory, files, and data with other threads of the same process.

- A **process** can have **multiple threads**, making it **multithreaded**.
- Threads within a process execute **independently** but share the **same process resources**.

---

2. Differences Between Process and Thread

Feature	Process	Thread
Definition	An independent execution unit with its own memory space.	A lightweight execution unit within a process.
Memory	Each process has a **separate address space**.	Threads **share memory** within a process.
Communication	Requires **Inter-Process Communication (IPC)**.	Threads communicate using **shared memory**.
Creation Overhead	High (requires more OS resources).	Low (faster creation and management).
Context Switching	Expensive and slow.	Faster due to shared memory.
Example	Running `Chrome` and `Notepad` as separate processes.	Multiple tabs running inside `Chrome` as separate threads.

---

3. Types of Threads

(a) User-Level Threads

- Managed by **user-level thread libraries** (e.g., POSIX Threads, Java Threads).
- **Faster** and do not require **kernel intervention**.
- If one thread blocks, all threads in the process are blocked (no direct kernel support).
- Example: **Java Multithreading** (using `Thread` class).

(b) Kernel-Level Threads

- Managed **directly by the OS kernel**.
- More **resource-intensive** but provides **true parallelism** on multi-core CPUs.
- If one thread blocks, others can continue running.
- Example: **Windows threads, Linux kernel threads (`kthread`)**.

---

4. Multithreading Models

(a) Many-to-One Model

- **Multiple user threads** are mapped to a **single kernel thread**.
- Disadvantage: If one thread **blocks**, the entire process blocks.
- Example: Green threads in older Java versions.

(b) One-to-One Model

- **Each user thread** is mapped to a **separate kernel thread**.
- Provides **true concurrency** but consumes more OS resources.
- Example: Linux, Windows threading model.

(c) Many-to-Many Model

- **Multiple user threads** are mapped to **fewer or equal kernel threads**.
- Balances efficiency and concurrency.
- Example: Solaris threads.

---

5. Thread Operations

Common thread operations include:

1. **Thread Creation** → `pthread_create()`, `new Thread()` (Java).
2. **Thread Execution** → `start()`, `run()`.
3. **Thread Synchronization** → Prevents race conditions using locks (`mutex`, `semaphore`).
4. **Thread Termination** → `pthread_exit()`, `join()`.

---

6. Advantages of Threads

✅ **Faster execution** – Lightweight, less overhead than processes.
✅ **Efficient resource sharing** – Threads within a process share memory.
✅ **Parallelism** – Utilizes multi-core processors efficiently.

✅ **Responsive applications** – GUI applications remain interactive (e.g., background tasks in web browsers).

---

7. Disadvantages of Threads

❌ **Synchronization issues** – Race conditions and deadlocks.
❌ **Difficult debugging** – Complex execution flow.
❌ **No memory isolation** – A faulty thread can crash the entire process.

---

8. Real-World Examples of Threads

- **Web Browsers** → Each tab runs as a thread inside a browser process.
- **Multimedia Applications** → Audio, video, and UI run in separate threads.
- **Game Development** → AI, rendering, and physics run on different threads.
- **Servers** → Handle multiple client requests using multi-threading (e.g., Apache Web Server).

Conclusion

Threads enable **efficient multitasking** and **faster execution** by sharing process resources. While they improve performance and responsiveness, they require **proper synchronization mechanisms** to avoid issues like race conditions and deadlocks.

Threading Issues in Operating Systems

When multiple threads execute concurrently, various **synchronization and concurrency issues** arise. These issues can lead to **inconsistent results, race conditions, and system crashes** if not handled properly.

---

1. Common Threading Issues

(a) Race Conditions

- Occurs when **multiple threads access shared data simultaneously** and the final outcome depends on the execution order.
- Example: Two threads updating a shared counter without proper synchronization.
- **Solution**: Use **locks, mutexes, and atomic operations**.

🔷 **Example of a Race Condition (Incorrect Output Due to Uncontrolled Access)**

```c
#include <stdio.h>
#include <pthread.h>

int counter = 0; // Shared resource

void* incrementCounter() {
    for (int i = 0; i < 1000000; i++)
        counter++; // No synchronization
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, incrementCounter, NULL);
    pthread_create(&t2, NULL, incrementCounter, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final Counter Value: %d\n", counter); // May not be 2000000 due
to race condition
    return 0;
}
```

(b) Deadlocks

- Occurs when two or more threads are **waiting for resources locked by each other**, resulting in an infinite wait.
- Example: Thread 1 locks **Resource A**, Thread 2 locks **Resource B**, and both wait for each other's resources.
- **Solution**:
    - Avoid circular waiting (resource allocation ordering).
    - Use **deadlock detection and prevention mechanisms**.

🔷 Example of Deadlock

```c
pthread_mutex_t lock1, lock2;

void* thread1() {
    pthread_mutex_lock(&lock1);
    sleep(1);  // Simulating delay
    pthread_mutex_lock(&lock2); // Deadlock occurs here
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}

void* thread2() {
    pthread_mutex_lock(&lock2);
    sleep(1);
    pthread_mutex_lock(&lock1); // Deadlock occurs here
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
}
```

🔷 **Fix:** Maintain **consistent lock ordering** (e.g., always lock `lock1` before `lock2`).

(c) Starvation

- Occurs when **low-priority threads never get CPU time** because higher-priority threads always execute first.
- Example: A background process never gets CPU time due to high-priority interactive tasks.
- **Solution**: Implement **priority aging**, where long-waiting threads gradually increase in priority.

(d) Livelock

- Similar to a deadlock, but threads **keep changing states** in response to each other without making progress.
- Example: Two threads repeatedly **releasing and reacquiring locks**, thinking they are resolving a deadlock.
- **Solution**: Introduce **random backoff or exponential retry delays**.

(e) Thread Synchronization Issues

- When multiple threads access shared resources **without synchronization**, leading to **unexpected behavior**.
- Example: A **producer thread** writes data while a **consumer thread** reads partial/incomplete data.
- **Solution**: Use **mutex, semaphores, and condition variables** to coordinate thread execution.

🔷 Fix Race Conditions Using Mutex

```
pthread_mutex_t lock;
void* safeIncrement() {
    pthread_mutex_lock(&lock);
    for (int i = 0; i < 1000000; i++)
        counter++;
    pthread_mutex_unlock(&lock);
}
```

---

2. Solutions to Threading Issues

✅ **Mutex Locks** – Prevents simultaneous access to shared resources.
✅ **Semaphores** – Used for managing resource availability (e.g., controlling multiple readers/writers).
✅ **Condition Variables** – Synchronizes threads based on a condition (e.g., producer-consumer

problems).
✅ **Priority Scheduling & Aging** – Prevents starvation.
✅ **Deadlock Avoidance (Resource Ordering)** – Ensures threads acquire locks in a defined order.

Conclusion

Threading issues such as **race conditions, deadlocks, starvation, and livelocks** can significantly affect system reliability. Using **proper synchronization techniques**, OS mechanisms, and best practices ensures **safe and efficient multithreading**.

Thread Libraries in Operating Systems

1. What is a Thread Library?

A **thread library** provides an API for creating, managing, and synchronizing threads. It allows developers to implement **multithreading** in applications without directly dealing with low-level OS details.

Thread libraries abstract thread **creation, termination, synchronization, and scheduling** mechanisms.

---

2. Types of Thread Libraries

Thread libraries can be categorized based on where they operate:

1. **User-Level Thread Libraries**
    - Implemented entirely in **user space** without kernel support.
    - Faster but may suffer from **blocking issues**.
    - Example: **POSIX Threads (pthreads)**, **Java Threads**.
2. **Kernel-Level Thread Libraries**
    - Managed directly by the **OS kernel**.
    - Slower but provide **true parallelism** and efficient scheduling.
    - Example: **Windows Threads**, **Linux Kernel Threads**.

---

3. Popular Thread Libraries

Thread Library	Platform	Type	Description
POSIX Threads	UNIX/Linux	User-Level	Standardized API for multithreading in C/C++.

Thread Library	Platform	Type	Description
(pthreads)			
Windows Threads	Windows	Kernel-Level	OS-supported threading model with Win32 API.
Java Threads	Cross-platform	User-Level	Part of Java API (`Thread` class, `Runnable` interface).
OpenMP	Cross-platform	Hybrid	Used for parallel programming in C, C++.
Grand Central Dispatch (GCD)	macOS, iOS	Kernel-Level	Apple's concurrency framework for managing tasks efficiently.

---

4. Key Features of Thread Libraries

✅ **Thread Creation** → Create and manage threads.
✅ **Synchronization Mechanisms** → Mutexes, semaphores, condition variables.
✅ **Thread Scheduling** → Defines priority and execution order.
✅ **Thread Termination** → Safely stopping or joining threads.
✅ **Portability** → Allows multithreading across different OS platforms.

---

5. Example Implementations

(a) POSIX Threads (pthreads) in C

```
#include <pthread.h>
#include <stdio.h>

void* printMessage(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, printMessage, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

🔷 **Explanation**:

- `pthread_create()` → Creates a new thread.
- `pthread_join()` → Waits for the thread to finish execution.

(b) Java Threads using Runnable

```
class MyThread implements Runnable {
    public void run() {
        System.out.println("Thread is running...");
    }

    public static void main(String[] args) {
        Thread t = new Thread(new MyThread());
        t.start(); // Starts the thread execution
    }
}
```

🔷 **Explanation**:

- Implements `Runnable` interface.
- `start()` → Begins execution of the thread.

---

6. Why Use Thread Libraries?

✅ **Simplifies multithreading** → Abstracts low-level OS details.
✅ **Improves performance** → Optimized for efficient execution.
✅ **Portable code** → Allows writing multithreaded programs across platforms.
✅ **Efficient synchronization** → Prevents race conditions and deadlocks.

Conclusion

Thread libraries like **pthreads, Windows Threads, Java Threads, and OpenMP** provide a standard way to implement **multithreading** in applications. Choosing the right library depends on **OS, performance needs, and programming language**.

Process Scheduling in Operating Systems

1. What is Process Scheduling?

Process scheduling is the mechanism by which the **Operating System (OS) selects and manages processes** for execution on the CPU. Since multiple processes compete for CPU time, the OS ensures **efficient execution and resource allocation** using scheduling algorithms.

---

2. Process Scheduling Queues

(a) Job Queue

- Contains **all processes in the system**, including those waiting to be executed.

(b) Ready Queue

- Contains processes **ready to execute** but waiting for CPU time.
- The **CPU scheduler** selects a process from this queue for execution.

(c) Waiting (Blocked) Queue

- Contains processes **waiting for I/O operations** (e.g., disk read/write, user input).
- Moves back to the **ready queue** after I/O completion.

---

3. Process Scheduling Types

(a) Long-Term Scheduling (Job Scheduling)

- Decides **which processes enter the system** (from the job queue to ready queue).
- Controls **degree of multiprogramming** (number of processes in memory).
- **Example**: Batch processing systems.

(b) Short-Term Scheduling (CPU Scheduling)

- Selects a process from the **ready queue** to execute on the CPU.
- **Fast and frequent** decision-making.
- **Example**: Selecting a process for execution in a time-sharing system.

(c) Medium-Term Scheduling (Swapping)

- Moves processes **to and from main memory** to **optimize CPU usage**.
- **Example**: Swapping out a process to disk (virtual memory).

---

4. CPU Scheduling Criteria

Criteria	Description
CPU Utilization	Maximizes CPU usage (avoids idle CPU).
Throughput	Maximizes the number of processes executed per unit time.

Criteria	Description
Turnaround Time	Minimizes total time from process submission to completion.
Waiting Time	Minimizes time spent in the ready queue.
Response Time	Minimizes time for the system to start responding to a process request.
Fairness	Ensures equal CPU time distribution among processes.

---

5. CPU Scheduling Algorithms

(a) First-Come, First-Served (FCFS)

- **Non-preemptive** (process runs until completion).
- **Processes execute in arrival order**.
- **Disadvantage**: **Convoy effect** (long processes delay short ones).
- **Example**: Batch systems.

🔷 Gantt Chart Example

```
Process:   | P1 | P2 | P3 |
Arrival:   | 0  | 1  | 2  |
Burst:     | 5  | 3  | 4  |
Execution: [P1]---->[P2]--->[P3]
```

Waiting Time = (Start time – Arrival time)

(b) Shortest Job Next (SJN or SJF)

- Selects the process with the **smallest burst time** first.
- **Can be preemptive or non-preemptive**.
- **Disadvantage**: **Starvation** (long processes may wait indefinitely).

(c) Round Robin (RR)

- Each process gets a **fixed time slice (time quantum)**.
- **Preemptive** (switches between processes after time quantum).
- **Fair but increases context switching overhead**.
- **Example**: Time-sharing systems.

🔷 Example with Time Quantum = 2

```
Process:   | P1 | P2 | P3 |
Burst:     | 5  | 3  | 4  |
Execution: [P1-2] → [P2-2] → [P3-2] → [P1-3] → [P3-2]
```

(d) Priority Scheduling

- Each process has a **priority** (higher priority executes first).
- **Can be preemptive or non-preemptive**.
- **Disadvantage**: **Starvation** (low-priority processes may never execute).
- **Solution**: Use **aging** (increase priority over time).

(e) Multilevel Queue Scheduling

- **Different queues for different types of processes** (e.g., foreground, background).
- **Queues have their own scheduling algorithms**.
- Example: Interactive processes may use **Round Robin**, while batch jobs use **FCFS**.

---

6. Real-World Examples of Process Scheduling

✅ **Operating Systems** → Windows, Linux, and macOS use **multilevel queue scheduling**.
✅ **Web Browsers** → Chrome assigns tabs as **separate processes**, scheduling them efficiently.
✅ **Multitasking Systems** → Use **Round Robin** for fair execution of multiple applications.

Conclusion

Process scheduling is crucial for **efficient CPU usage, multitasking, and responsiveness**. The choice of scheduling algorithm depends on **system requirements**, such as **real-time processing, fairness, and performance optimization**.

Preemptive vs. Non-Preemptive Scheduling Algorithms

1. Introduction

CPU scheduling is classified into **Preemptive** and **Non-Preemptive** scheduling, based on whether a process can be interrupted while executing.

Type	Description
Preemptive Scheduling	The CPU **can be taken** from a process before it finishes.
Non-Preemptive Scheduling	Once a process gets the CPU, it **runs until completion**.

---

2. Non-Preemptive Scheduling Algorithms

In **Non-Preemptive Scheduling**, the **CPU is assigned to a process until it completes or voluntarily releases the CPU**.

(a) First-Come, First-Served (FCFS)

✅ **Simple and easy to implement**
❌ **Convoy Effect**: Shorter processes wait for longer ones

🔷 **Example Gantt Chart (FCFS Scheduling)**

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

Execution Order:

```
[P1 - 5] → [P2 - 3] → [P3 - 4]
```

(b) Shortest Job First (SJF) - Non-Preemptive

✅ **Minimizes waiting time**
❌ **Starvation**: Longer processes may wait indefinitely

🔷 **Example (SJF Scheduling - Non-Preemptive)**

Process	Arrival Time	Burst Time
P1	0	6
P2	2	2
P3	4	4

Execution Order:

```
[P1 - 6] → [P2 - 2] → [P3 - 4]
```

(c) Priority Scheduling - Non-Preemptive

✅ **High-priority processes execute first**
❌ **Starvation**: Low-priority processes may never execute

🔷 **Example (Priority Scheduling - Non-Preemptive)**

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

Execution Order:

```
[P2 (Priority 1)] → [P1 (Priority 2)] → [P3 (Priority 3)]
```

---

3. Preemptive Scheduling Algorithms

In **Preemptive Scheduling**, the CPU **can be taken away** from a process before it finishes execution.

(a) Round Robin (RR)

✅ **Fair scheduling, avoids starvation**
❌ **High context switching overhead**

🔷 **Example (Round Robin, Time Quantum = 2 ms)**

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

Execution Order (Time Quantum = 2ms):

```
[P1 - 2] → [P2 - 2] → [P3 - 2] → [P1 - 2] → [P3 - 2] → [P1 - 1]
```

(b) Shortest Remaining Time First (SRTF) - Preemptive SJF

✅ **Minimizes waiting time**
❌ **Starvation for longer jobs**

🔷 **Example (SRTF Scheduling)**

Process	Arrival Time	Burst Time
P1	0	6
P2	2	2
P3	4	4

Execution Order:

```
[P1 - 2] → [P2 - 2] → [P3 - 4] → [P1 - 4]
```

(c) Priority Scheduling - Preemptive

✅ **Higher-priority processes execute first**
❌ **Starvation for low-priority processes** (fixed using **Aging**)

🔷 Example (Preemptive Priority Scheduling)

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

Execution Order:

```
[P2 (Priority 1)] → [P1 (Priority 2)] → [P3 (Priority 3)]
```

---

4. Comparison of Preemptive vs. Non-Preemptive Scheduling

Factor	Preemptive Scheduling	Non-Preemptive Scheduling
Interrupts Process?	☑ Yes (Process can be interrupted)	✕ No (Runs until completion)
Context Switching Overhead	☑ High (Frequent switching)	✕ Low (No switching during execution)
Response Time	☑ Faster (Suitable for real-time systems)	✕ Slower (Longer processes delay shorter ones)
CPU Utilization	☑ Better CPU utilization	✕ May cause CPU idling
Starvation Risk	☑ Possible, but mitigated by priority aging	✕ High for long processes

---

5. Conclusion

- **Non-Preemptive Scheduling** is **simpler and has lower overhead**, but may cause **long waiting times** for short processes.
- **Preemptive Scheduling** improves **response time and CPU efficiency**, but **increases context switching overhead**.
- **Choosing the right scheduling algorithm** depends on **system requirements** (e.g., fairness, efficiency, or real-time constraints).

UNIT III

Process Management in Operating Systems

1. Introduction to Process Management

Process management is a fundamental function of an **Operating System (OS)** that deals with the **creation, execution, scheduling, and termination** of processes. The OS is responsible for **efficient resource allocation, synchronization, and communication** between processes.

---

2. Process Concept

What is a Process?

A **process** is a program in execution. It includes:

- **Program Code** (Instructions)
- **Program Counter** (Current execution point)
- **Registers** (CPU state)
- **Memory (Data, Stack, Heap)**
- **I/O Information** (Open files, devices)

Process States

A process moves through different states during execution:

State	Description
New	Process is created but not yet ready to execute.
Ready	Process is waiting in the ready queue for CPU time.
Running	Process is actively executing instructions on the CPU.
Waiting	Process is waiting for an I/O operation or event to complete.
Terminated	Process has finished execution or is forcefully stopped.

🔷 **State Transition Diagram:**

```
New → Ready → Running → Terminated
        ↓        ↑
      Waiting
```

---

3. Process Control Block (PCB)

The **Process Control Block (PCB)** is a data structure maintained by the OS to store **process-related information**.

PCB Structure

Field	Description
Process ID (PID)	Unique identifier for the process.
Program Counter	Stores the address of the next instruction to execute.
Registers	Stores the CPU state (accumulator, stack pointer, etc.).
Process State	Current execution state (New, Ready, Running, Waiting, Terminated).
Memory Management Info	Includes base & limit registers, page tables, etc.
I/O Information	Open files, I/O devices allocated to the process.
Scheduling Info	Priority, scheduling queue pointers, etc.

---

4. Process Scheduling

The OS schedules processes to efficiently use CPU time.

Scheduling Queues

1. **Job Queue** → Stores all processes in the system.
2. **Ready Queue** → Stores processes ready for execution.
3. **Waiting Queue** → Stores processes waiting for I/O operations.

Types of Schedulers

Scheduler	Function
Long-Term Scheduler (Job Scheduler)	Controls which processes enter the system for execution.
Short-Term Scheduler (CPU Scheduler)	Selects processes from the ready queue to execute on the CPU.

Scheduler	Function
Medium-Term Scheduler	Swaps processes in and out of memory to optimize CPU usage.

Scheduling Algorithms

- **FCFS (First-Come, First-Served)**
- **SJF (Shortest Job First)**
- **Round Robin (RR)**
- **Priority Scheduling**
- **Multilevel Queue Scheduling**

---

5. Process Creation and Termination

Process Creation

A process creates a child process using **system calls** like `fork()` in Unix/Linux.

🔷 **Parent-Child Relationship (Process Hierarchy)**

```
Parent Process
    ├── Child Process 1
    ├── Child Process 2
```

🔷 **Steps in Process Creation:**

1. Allocate **PCB**.
2. Assign **unique Process ID (PID)**.
3. Load **program code into memory**.
4. Initialize CPU **registers and stack**.
5. Add process to **ready queue**.

Process Termination

A process is terminated when:

- It **completes execution** (`exit()` system call).
- It is **killed by another process** (`kill()` command).
- It encounters **errors** (e.g., divide by zero).

---

6. Inter-Process Communication (IPC)

Processes need to communicate to share data or coordinate execution.

Types of IPC

IPC Mechanism	Description
Shared Memory	Processes share a **common memory segment**.
Message Passing	Processes communicate via **send()** and **receive()** system calls.
Pipes	**Unidirectional** communication between processes.
Sockets	**Network-based IPC** between remote systems.
Signals	Used to notify a process of an event (e.g., `SIGKILL`, `SIGINT`).

---

7. Threads & Multithreading

A **thread** is a lightweight process that runs within a process.

Types of Threads

Type	Description
User-Level Threads	Managed by user libraries, not the OS.
Kernel-Level Threads	Managed by the OS kernel.

Advantages of Multithreading

✅ Faster execution & responsiveness
✅ Efficient CPU utilization
✅ Parallel processing

---

8. Process Synchronization

Processes must be synchronized to avoid conflicts when accessing shared resources.

Synchronization Issues

- **Race Condition**: Two processes modify shared data simultaneously.
- **Critical Section Problem**: A section of code where shared data is accessed.

🔷 **Synchronization Solutions:**

1. **Locks & Mutexes**
2. **Semaphores**
3. **Monitors**

---

9. Deadlocks

A **deadlock** occurs when processes wait indefinitely for resources held by each other.

Deadlock Prevention & Recovery

- **Avoidance**: Use algorithms like **Banker's Algorithm**.
- **Detection & Recovery**: Identify deadlocks and **terminate** a process.

---

10. Conclusion

Process Management in an OS ensures **efficient execution, resource allocation, and process coordination**. Key functions include **process creation, scheduling, synchronization, and IPC**, which are essential for **multitasking and parallel processing**.

Deadlock in Operating System

1. Introduction to Deadlock

A **deadlock** is a situation in which a set of processes get stuck because **each process is waiting for a resource held by another process**, forming a circular waiting condition.

Example of Deadlock

- **Process P1** holds **Resource R1** and requests **Resource R2**.
- **Process P2** holds **Resource R2** and requests **Resource R1**.
- Both processes **wait indefinitely** since each is holding a resource the other needs.

---

2. Necessary Conditions for Deadlock

A deadlock occurs if **all** of the following **four conditions** hold simultaneously:

Condition	Description

Condition	Description
Mutual Exclusion	Only **one process** can use a resource at a time.
Hold and Wait	A process holding at least one resource **waits** for more resources.
No Preemption	A resource cannot be forcibly taken from a process.
Circular Wait	A set of processes form a circular chain where **each process waits for a resource held by the next process**.

🔷 Example Circular Wait:

```
P1 → R1 → P2 → R2 → P3 → R3 → P1
```

Here, **P1 waits for R3**, which is held by **P3**, which in turn **waits for R1**, held by **P1**.

---

3. Resource Allocation Graph (RAG)

A **Resource Allocation Graph (RAG)** is used to represent the **state of resource allocation** in a system.

Components of RAG

- **Processes (P1, P2, ...)** → Represented as **circles**.
- **Resources (R1, R2, ...)** → Represented as **rectangles**.
- **Edges:**
  - **Request Edge (P → R)**: A process **requests** a resource.
  - **Assignment Edge (R → P)**: A resource is **allocated** to a process.

Deadlock Detection in RAG

- If the **graph contains a cycle**, **deadlock may exist**.
- If **each resource has only one instance**, then **a cycle indicates a deadlock**.

---

4. Deadlock Handling Strategies

There are **four approaches** to handle deadlocks:

Method	Description
1. Deadlock Prevention	Ensure that at least one of the **four necessary conditions does not occur**.
2. Deadlock Avoidance	Dynamically check for safe resource allocation using **Banker's Algorithm**.
3. Deadlock Detection & Recovery	Detect a deadlock when it occurs and recover by **terminating processes or preempting resources**.
4. Ignore the Problem	Used in systems like **Windows, Linux, UNIX**, assuming deadlocks **occur rarely**.

---

5. Deadlock Prevention

To **prevent** a deadlock, break at least **one** of the **four necessary conditions**:

Condition	Prevention Strategy
Mutual Exclusion	Allow resource **sharing** where possible (e.g., read-only files).
Hold and Wait	Require a process to **request all resources at once** before execution.
No Preemption	Allow the OS to **preempt resources** from a process when needed.
Circular Wait	Impose an **ordering on resource requests** (e.g., always request in a predefined order).

---

6. Deadlock Avoidance

Deadlock can be **avoided** by ensuring that the system is always in a **safe state**.

Safe State vs. Unsafe State

- **Safe State:** There exists a safe sequence where all processes can complete execution **without deadlock**.
- **Unsafe State:** No guarantee that processes will complete **without deadlock**.

Banker's Algorithm (for Deadlock Avoidance)

Used to decide whether a process should **wait or proceed** based on available resources.

🔷 **Steps:**

1. Calculate the **Need Matrix** (`Need = Max - Allocation`).
2. Check if resources are **available** to fulfill a process's needs.
3. If a safe sequence exists, allocate resources; otherwise, make the process wait.

---

7. Deadlock Detection & Recovery

If deadlocks **cannot be prevented or avoided**, the OS must **detect** and **recover** from them.

Deadlock Detection

- **Method:** Check for a **cycle** in the **Resource Allocation Graph (RAG)**.
- **For multiple resource instances:** Use a **deadlock detection algorithm** similar to **Banker's Algorithm**.

Deadlock Recovery

Once a deadlock is detected, the system can recover in two ways:

Recovery Method	Description
Process Termination	Kill one or more processes in the deadlock until the cycle is broken.
Resource Preemption	Temporarily take resources away from processes and reallocate them.

---

8. Conclusion

- **Deadlocks occur** when processes wait indefinitely for resources.
- **Four necessary conditions** must be met for a deadlock to happen.
- Deadlocks can be **prevented, avoided, detected, or ignored** based on system requirements.
- **Banker's Algorithm** helps **avoid** deadlocks by checking safe resource allocation.
- **Recovery methods** include **process termination** and **resource preemption**.

Deadlock Characterization in Operating System

1. Introduction

A **deadlock** is a situation where a set of processes become permanently blocked because **each process is waiting for a resource that another process is holding**.

To understand deadlocks, we characterize them based on the **necessary conditions, resource allocation models, and detection methods**.

---

2. Necessary Conditions for Deadlock

A **deadlock occurs** if all **four** of the following conditions hold **simultaneously**:

Condition	Description
Mutual Exclusion	At least one resource is held in a **non-shareable** mode (only one process can use it at a time).
Hold and Wait	A process **holds at least one resource** and **waits for additional resources** held by other processes.
No Preemption	Resources **cannot be forcibly taken** from a process; they must be released voluntarily.
Circular Wait	A set of processes form a circular chain where **each process is waiting for a resource held by the next process** in the chain.

Example of Circular Wait

```
P1 → R1 → P2 → R2 → P3 → R3 → P1
```

Here, **P1 waits for R3**, which is held by **P3**, which **waits for R2**, held by **P2**, and so on, forming a **deadlock cycle**.

---

3. Resource Allocation Graph (RAG)

A **Resource Allocation Graph (RAG)** is used to model resource allocation and detect deadlocks.

Components of RAG

- **Processes (P1, P2, ...)** → Represented as **circles**.

- **Resources (R1, R2, ...)** → Represented as **rectangles**.
- **Edges:**
  - **Request Edge (P → R)**: Process **requests** a resource.
  - **Assignment Edge (R → P)**: Resource **allocated** to a process.

Deadlock Detection in RAG

- **No Cycle** → No deadlock.
- **Cycle Exists**:
  - If **each resource has only one instance**, **deadlock exists**.
  - If **resources have multiple instances**, a cycle **may or may not** indicate deadlock.

---

4. Methods to Handle Deadlocks

Deadlocks can be handled using the following approaches:

Method	Description
1. Deadlock Prevention	Ensure that at least one of the **four necessary conditions does not occur**.
2. Deadlock Avoidance	Dynamically check for safe resource allocation using **Banker's Algorithm**.
3. Deadlock Detection & Recovery	Detect a deadlock when it occurs and recover by **terminating processes or preempting resources**.
4. Ignore the Problem	Used in systems like **Windows, Linux, UNIX**, assuming deadlocks **occur rarely**.

---

5. Conclusion

- **Deadlock characterization** helps identify and manage deadlocks efficiently.
- The **four necessary conditions** must all be present for a deadlock to occur.
- **Resource Allocation Graphs (RAGs)** help in visualizing deadlocks.
- Deadlocks can be **prevented, avoided, detected, or ignored** based on system requirements.

Necessary and Sufficient Conditions for Deadlock

1. Introduction

A **deadlock** is a situation in which a set of processes are **permanently blocked**, each waiting for a resource that another process holds.

For a **deadlock to occur**, four **necessary conditions** must **simultaneously** hold.

---

2. Necessary Conditions for Deadlock

A deadlock occurs only if **all four** of the following conditions hold **simultaneously**:

Condition	Description
1. Mutual Exclusion	A resource **can be used by only one process at a time** (it cannot be shared).
2. Hold and Wait	A process **holding at least one resource is waiting** for additional resources held by other processes.
3. No Preemption	A resource **cannot be forcibly taken** from a process; it must be released voluntarily.
4. Circular Wait	A set of processes form a **circular chain** where each process waits for a resource held by the next process.

Example of Circular Wait

```
P1 → R1 → P2 → R2 → P3 → R3 → P1
```

- **P1 waits for R3** held by **P3**.
- **P3 waits for R2** held by **P2**.
- **P2 waits for R1** held by **P1**.

Since this forms a **circular dependency**, **deadlock occurs**.

---

3. Sufficient Condition for Deadlock

The **sufficient condition** for deadlock is the **simultaneous presence of all four necessary conditions**.

Key Points:

- If **any one** of the four conditions is **prevented**, **deadlock will not occur**.
- If **all four** conditions hold, **deadlock is guaranteed**.

---

4. How to Prevent Deadlock?

To **prevent** deadlock, at least **one** of the necessary conditions must be **broken**:

Condition	Prevention Strategy
Mutual Exclusion	Allow **resource sharing** where possible (e.g., read-only access).
Hold and Wait	Require a process to **request all resources at once** before execution.
No Preemption	Allow the OS to **preempt resources** from a process when needed.
Circular Wait	Impose a **strict ordering** on resource requests to prevent cycles.

---

5. Conclusion

- **All four necessary conditions must hold simultaneously** for a deadlock to occur.
- Breaking any **one** condition **prevents** deadlock.
- Understanding these conditions helps in **designing better deadlock-handling techniques** in operating systems.

Deadlock Handling Approaches in Operating System

1. Introduction

A **deadlock** occurs when a set of processes become **permanently blocked**, each waiting for a resource held by another process.

There are four main approaches to **handle deadlocks**:

Approach	Description
1. Deadlock Prevention	Ensure that at least one of the **necessary conditions for deadlock** does not occur.

Approach	Description
2. Deadlock Avoidance	Dynamically check for safe resource allocation to **avoid** deadlocks.
3. Deadlock Detection & Recovery	Detect a deadlock after it occurs and take action to recover.
4. Ignore the Problem	Assume deadlocks are **rare** and do nothing (used in many general-purpose systems).

---

2. Deadlock Prevention

Deadlock prevention ensures that at least **one of the four necessary conditions** does **not hold**, thus preventing deadlock.

Condition	Prevention Strategy
Mutual Exclusion	Allow **resource sharing** where possible (e.g., read-only files).
Hold and Wait	Require a process to **request all resources at once** before execution.
No Preemption	Allow the OS to **forcibly take resources** from a process if needed.
Circular Wait	Impose a **strict ordering** on resource requests (e.g., always request resources in a predefined order).

Drawback:

- **Inefficient resource utilization** (e.g., forcing a process to request all resources at once may lead to underutilization).

---

3. Deadlock Avoidance

Deadlock avoidance ensures the system **never enters an unsafe state**, where deadlock is possible.

🔷 Common Algorithm: Banker's Algorithm

- The OS **analyzes resource allocation requests** before granting them.

- It only grants a request **if it ensures a safe state** (i.e., a sequence of process execution exists that avoids deadlock).

Safe vs. Unsafe State

- **Safe State:** A sequence exists where all processes can complete **without deadlock**.
- **Unsafe State:** No guarantee that processes will complete **without deadlock**.

Drawbacks:

- **Complex implementation** with frequent resource allocation checks.
- **Requires prior knowledge** of maximum resource needs for each process.

---

4. Deadlock Detection & Recovery

If **deadlock prevention or avoidance is not implemented**, the OS must **detect and recover** from deadlocks.

Deadlock Detection

- The OS **monitors resource allocation** and **checks for cycles in the Resource Allocation Graph (RAG)**.
- If a **cycle is found**, deadlock exists.

Deadlock Recovery

Once a deadlock is detected, the system can recover using:

Recovery Method	Description
Process Termination	Kill one or more processes to break the deadlock.
Resource Preemption	Temporarily take resources from a process and reallocate them.

Drawbacks:

- **Process termination can lead to data loss**.
- **Resource preemption may require rollback mechanisms** to restore a process's previous state.

---

5. Ignore the Problem (Ostrich Algorithm)

Some systems (e.g., **Windows, Linux, UNIX**) **ignore deadlocks**, assuming they are rare.

Why?

- **Deadlocks are rare** in most systems.
- **Detecting and preventing deadlocks adds overhead** and complexity.
- The system may **rely on process restarts** or **user intervention** to resolve deadlocks.

Drawback:

- If a deadlock occurs, **the system may freeze**, requiring manual intervention.

---

6. Comparison of Deadlock Handling Approaches

Approach	Prevents Deadlock?	Overhead	Used In
Prevention	✅ Yes	🔺 High	Real-time & safety-critical systems
Avoidance	✅ Yes	🔺 High	Banking, databases, critical systems
Detection & Recovery	❌ No (Detects & Recovers)	🔺 High	General-purpose OS (if implemented)
Ignore the Problem	❌ No	🔽 Low	Windows, Linux, UNIX

---

7. Conclusion

- **Deadlocks can be prevented, avoided, detected, or ignored**, depending on system needs.
- **Prevention & avoidance guarantee deadlock-free execution**, but may reduce resource utilization.
- **Detection & recovery are useful when deadlocks are rare and difficult to prevent.**
- Many systems **ignore deadlocks**, as they are uncommon and difficult to manage efficiently.

Deadlock Prevention in Operating System

1. Introduction

Deadlock prevention ensures that the system never enters a deadlock state by eliminating **one or more** of the **necessary conditions** for deadlock.

A deadlock occurs if all four **necessary conditions** hold **simultaneously**:

1. **Mutual Exclusion**
2. **Hold and Wait**
3. **No Preemption**
4. **Circular Wait**

Goal of Deadlock Prevention

To **prevent deadlock**, at least **one** of these conditions must be **eliminated**.

---

2. Deadlock Prevention Techniques

Deadlock prevention can be achieved by breaking any one of the four necessary conditions.

Condition	Prevention Strategy	Explanation
Mutual Exclusion	Allow resource sharing	If possible, make resources **sharable** (e.g., read-only files can be accessed by multiple processes).
Hold and Wait	Request all resources at once	A process must request **all required resources at the beginning** and wait if any resource is unavailable.
No Preemption	Allow forced resource preemption	The OS can **forcibly take resources** from a process and allocate them to another.
Circular Wait	Impose a strict resource ordering	Define a **fixed order** for resource requests and ensure that processes request resources in this order.

---

3. Eliminating Each Condition in Detail

1. Breaking Mutual Exclusion

🔷 **Strategy**: Convert non-shareable resources into **sharable** ones.
🔷 **Example**:

- Read-only files **do not require mutual exclusion** and can be accessed by multiple processes.

🚨 **Drawback**:

- Some resources (e.g., printers, memory locks) **cannot be shared**, so this method is **not always feasible**.

---

2. Breaking Hold and Wait

🔷 **Strategy**: A process must **request all resources at the start** and wait until all are available.

🔷 **Example**:

- A process requesting CPU, printer, and scanner must **request all at once** before execution starts.

🚨 **Drawbacks**:

- **Resource underutilization**: A process may **hold resources it does not need immediately**.
- **Starvation**: If a process requires many resources, it may wait indefinitely.

✅ **Solution**: Implement **resource request timeouts** to prevent starvation.

---

3. Breaking No Preemption

🔷 **Strategy**: If a process holding a resource is waiting for another, the OS can **forcefully take away the resource** and allocate it to another process.

🔷 **Example**:

- If **Process P1** holds a printer but needs a scanner, and **Process P2** holds the scanner but needs a printer, the OS can **preempt** one of these resources and reassign it.

🚨 **Drawbacks**:

- **Data loss**: If a resource is taken away forcefully, the process may need to **restart**.

- **Not suitable for all resources** (e.g., files being written to cannot be preempted easily).

✅ **Solution**: Use **rollback mechanisms** to store process progress before preemption.

---

4. Breaking Circular Wait

🔷 **Strategy**: Impose a **total ordering** on resource requests. Processes **must request resources in a predefined order**.

🔷 **Example**:

- Assign a **unique number** to each resource (e.g., **R1, R2, R3**).
- A process **must request resources in increasing order** (e.g., a process holding R1 can request R2 but not R3 first).

🚨 **Drawbacks**:

- **Difficult to enforce in dynamic systems** where processes do not know in advance which resources they will need.

✅ **Solution**: Use **hierarchical resource allocation** for efficient ordering.

---

4. Advantages & Disadvantages of Deadlock Prevention

Advantages	Disadvantages
Ensures deadlock never occurs	**Can lead to resource underutilization** (e.g., Hold and Wait prevention).
Simple to implement in some cases (e.g., ordering resources)	**Not always feasible** (e.g., Mutual Exclusion for non-shareable resources).
Works well for critical systems (e.g., real-time OS)	**Can cause starvation** (e.g., large requests may be delayed indefinitely).

---

5. Conclusion

- **Deadlock prevention eliminates at least one of the four necessary conditions** to ensure deadlock never occurs.

- Each technique has **trade-offs**, including resource underutilization and potential starvation.
- **Choosing the right method** depends on the **system type** and **resource characteristics**.

Deadlock Avoidance and Deadlock Detection & Recovery in OS

1. Introduction

Deadlock occurs when processes are indefinitely waiting for resources held by each other. There are two ways to handle deadlocks:

1. **Deadlock Avoidance** – Ensure the system **never enters a deadlock state**.
2. **Deadlock Detection & Recovery** – Allow deadlocks to occur but detect and recover from them.

---

2. Deadlock Avoidance

Deadlock avoidance ensures that the system **never enters an unsafe state** where deadlock is possible. The OS dynamically **analyzes each resource request** before granting it.

Safe and Unsafe States

- **Safe State** – There exists an execution sequence where all processes complete **without deadlock**.
- **Unsafe State** – A sequence exists where deadlock **may** occur.

🔷 **Deadlock Avoidance Goal:** Always remain in a **safe state**.

Techniques for Deadlock Avoidance

Technique	Description
Banker's Algorithm	Ensures safe resource allocation by **granting resources only if they lead to a safe state**.
Resource Allocation Graph (RAG) with Cycle Detection	A **graph-based** approach that checks for cycles before granting resources.

---

1. Banker's Algorithm (Most Common)

- Works **like a bank** deciding loans based on available funds.
- Every process must **declare maximum resource requirements** before execution.
- The OS checks whether granting a request keeps the system in a **safe state**.

Steps of Banker's Algorithm

1. Check if the **requested resources** are within the process's **maximum declared need**.
2. Check if the **resources are available**.
3. **Temporarily allocate** resources and check for a **safe sequence** of execution.
4. If the system **remains safe**, grant the request. Otherwise, **make the process wait**.

Example

Process	Max Need	Allocated	Available: (A=3, B=3, C=2)
P1	(7,5,3)	(0,1,0)	
P2	(3,2,2)	(2,0,0)	
P3	(9,0,2)	(3,0,2)	
P4	(2,2,2)	(2,1,1)	

- If **P1 requests (1,0,2)**, the OS checks if it **leads to a safe sequence**.
- If yes, resources are allocated. Otherwise, P1 must wait.

🚨 **Drawbacks of Banker's Algorithm**:
✅ Prevents deadlocks **before they happen**, but:

- Requires **prior knowledge** of resource needs.
- Can cause **starvation** if a process waits too long.
- **Not practical** for dynamic systems with unknown resource requirements.

---

2. Resource Allocation Graph (RAG) with Cycle Detection

- A **graph** is maintained with **processes and resources as nodes**.
- If a **cycle is detected**, a deadlock **might occur**.

Safe State Example (No Cycle)

```
P1 → R1 → P2 → R2
```

- No **circular dependency**, so no deadlock.

Unsafe State Example (Cycle Present)

```
P1 → R1 → P2 → R2 → P1 (Cycle detected!)
```

- The system is **unsafe**, and deadlock may occur.

🚨 **Drawback:**

- Works well for **single-instance** resources but **fails for multiple instances**.

---

3. Deadlock Detection and Recovery

Instead of **preventing** deadlock, some systems allow deadlocks to occur and then **detect and recover** from them.

3.1 Deadlock Detection

🔷 Used when **deadlock prevention or avoidance is not implemented**.
🔷 The OS **periodically checks** for deadlocks.
🔷 Deadlock is detected using **two techniques**:

Technique	Description
Wait-for Graph (WFG)	Detects cycles in resource allocation when each process waits for another.
Detection Algorithm for Multiple Resources	Uses a **matrix-based** approach similar to the Banker's Algorithm.

---

1. Wait-for Graph (WFG)

- A **simplified Resource Allocation Graph (RAG)** where **resources are removed**, and only processes remain.
- If a **cycle** is detected, **deadlock is confirmed**.

Example

```
P1 → P2 → P3 → P1   (Cycle detected: Deadlock exists!)
```

🚨 **Drawback**:

- **Only works for single-instance resources**.

---

2. Detection Algorithm for Multiple Resources

- Uses **a resource allocation matrix** similar to the **Banker's Algorithm**.

- **Periodically checks if any process is stuck** waiting for unavailable resources.
- If **no progress is possible**, the system **declares a deadlock**.

🚨 **Drawback**:

- **Requires frequent checking**, increasing CPU overhead.

3.2 Deadlock Recovery

Once a deadlock is detected, the OS must **recover** the system. There are **two main approaches**:

Recovery Method	Description
Process Termination	Kill one or more processes to break the deadlock.
Resource Preemption	Take resources from some processes and give them to others.

---

1. Process Termination

- **Terminate all deadlocked processes**.
- **Or terminate one process at a time** until the deadlock breaks.

🚨 **Drawbacks**:

- **Data loss** (if an important process is killed).
- **Difficult to decide which process to terminate**.

---

2. Resource Preemption

- **Temporarily take away resources** and reallocate them.
- **Rollback** the process to a previous safe state.

🚨 **Drawbacks**:

- **Complex implementation** (must store process states).
- **Starvation risk** (same processes may always get preempted).

---

4. Comparison of Deadlock Handling Approaches

Approach	Prevents Deadlock?	Overhead	Used In
Avoidance (Banker's Algorithm)	☑ Yes	🔺 High	Banking, real-time systems
Detection & Recovery	✕ No (Detects & Recovers)	🔺 High	General-purpose OS (if implemented)
Prevention	☑ Yes	🔺 High	Critical systems
Ignore the Problem	✕ No	🔽 Low	Windows, Linux, UNIX

---

5. Conclusion

- **Deadlock avoidance (Banker's Algorithm) ensures the system never enters an unsafe state** but requires **prior knowledge of resource needs**.
- **Deadlock detection** allows deadlocks to occur but periodically checks for them.
- **Recovery methods like termination and preemption help resolve deadlocks but can cause data loss and starvation.**
- **Many operating systems (like Linux, Windows) ignore deadlocks, assuming they are rare.**

Critical Section in Operating System

1. Introduction

In **multitasking and multiprocessing environments**, multiple processes or threads **share resources** like memory, files, and variables. To avoid **race conditions** (where multiple processes modify shared data inconsistently), we need a mechanism to **synchronize access**.

A **Critical Section** is a section of code where a process **accesses shared resources**. It must be **executed by only one process at a time** to ensure data consistency.

---

2. Critical Section Problem

The **Critical Section Problem** arises when multiple processes **simultaneously access shared resources**, leading to:
🔷 **Data inconsistency**

- 🔷 **Race conditions**
- 🔷 **Unexpected behavior**

Example of a Race Condition

```
Process 1:          Process 2:
Read(x);            Read(x);
x = x + 1;         x = x + 1;
Write(x);          Write(x);
```

- If **both processes read x = 5**, they update it to **6 separately** instead of **7**.
- **The final value is incorrect** due to unsynchronized access.

---

3. Conditions for a Solution

A good solution to the Critical Section Problem must satisfy **three conditions**:

Condition	Description
Mutual Exclusion	Only one process should execute the **critical section at a time**.
Progress	If no process is in the critical section, **other processes should not be blocked**.
Bounded Waiting	Every process must get a **fair chance** to enter the critical section (no starvation).

---

4. Critical Section Handling Mechanisms

Various **synchronization mechanisms** are used to solve the **Critical Section Problem**:

1. Software Solutions (Algorithmic Approaches)

- **Peterson's Algorithm**
- **Dekker's Algorithm**

2. Hardware Solutions

- **Test-and-Set Lock (TSL)**
- **Compare-and-Swap (CAS)**
- **Disabling Interrupts**

3. OS-Based Solutions

- **Mutex Locks**

- **Semaphores**
- **Monitors**

---

5. Critical Section Solutions in Detail

1. Peterson's Algorithm (Software Solution)

A classic **two-process solution** using a **flag and turn variable** to ensure mutual exclusion.

Algorithm

- Each process **sets its flag to true** before entering the critical section.
- It **checks the other process's flag** and only enters if it's safe.
- **Turn variable** ensures fairness.

Code (Peterson's Algorithm)

```
int flag[2] = {false, false};
int turn;

void process_0() {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);  // Wait if process 1 is in CS
    // Critical Section
    flag[0] = false;
}
```

✅ Ensures **Mutual Exclusion, Progress, and Bounded Waiting**
🚨 Works only for **two processes**

---

2. Test-and-Set (Hardware Solution)

A **hardware-based atomic instruction** that locks access to the critical section.

Algorithm

1. A **shared lock variable** is initialized to `false`.
2. **Before entering the CS**, a process executes `Test-and-Set(lock)`.
3. If **lock is false**, the process enters; otherwise, it waits.

Code (Test-and-Set)

```
boolean TestAndSet(boolean *lock) {
    boolean old = *lock;
    *lock = true;
    return old;
}
```

✅ **Efficient for multi-processor systems**
🚨 May lead to **busy waiting (CPU waste)**

---

3. Mutex Locks (OS-Based)

- **Mutual Exclusion Locks (Mutexes)** provide an OS-level solution.
- A process **locks** before entering the critical section and **unlocks** after.

Code (Mutex in C)

```
pthread_mutex_t lock;

void function() {
    pthread_mutex_lock(&lock);
    // Critical Section
    pthread_mutex_unlock(&lock);
}
```

✅ **Efficient & widely used in OS kernels**
🚨 **Blocking-based**, so processes may be put to sleep.

---

4. Semaphores (OS-Based)

- Semaphores are **integer variables** used for synchronization.
- **Two operations**:
  - `wait(S)` (decrease semaphore)
  - `signal(S)` (increase semaphore)

Code (Semaphore Solution)

```
sem_t S;

void process() {
    sem_wait(&S);  // Wait operation
    // Critical Section
    sem_post(&S);  // Signal operation
}
```

✅ **Can synchronize multiple processes**
🚨 **Complex implementation & may cause deadlocks**

---

5. Monitors (High-Level Synchronization)

- A **monitor is an abstract data type** that ensures only **one process executes at a time**.

- It uses **condition variables** for synchronization.

Code (Monitor)

```
class SharedData {
    synchronized void criticalSection() {
        // Only one process can execute this method at a time
    }
}
```

✅ **Prevents race conditions automatically**
🚨 **Limited to high-level programming languages like Java**

---

6. Comparison of Synchronization Techniques

Method	Type	Pros	Cons
Peterson's Algorithm	Software	Simple, no special hardware needed	Only for **two processes**
Test-and-Set	Hardware	Fast, atomic operations	Busy waiting (CPU waste)
Mutex Locks	OS-based	Simple, avoids busy waiting	Can lead to **priority inversion**
Semaphores	OS-based	Works for **multiple processes**	Complex, may cause **deadlocks**
Monitors	High-Level	Automatic synchronization	Not available in all languages

---

7. Conclusion

- The **Critical Section Problem** must be solved to prevent **race conditions** and **inconsistent data**.
- **Software-based** solutions work for **small-scale** systems but may be inefficient.
- **Hardware-based** solutions (Test-and-Set) are **faster** but can cause **busy waiting**.
- **OS-based solutions (Mutex, Semaphores, Monitors)** are **widely used in modern operating systems**.

Semaphores in Operating System

1. Introduction

A **semaphore** is a synchronization primitive used to control **access to shared resources** in a multi-process or multi-threaded environment. It helps prevent issues like **race conditions, deadlocks, and starvation** by ensuring proper coordination among processes.

Semaphores were introduced by **Edsger Dijkstra** and are widely used in **process synchronization and inter-process communication (IPC).**

---

2. Types of Semaphores

There are two main types of semaphores:

Type	Description
Binary Semaphore	Takes only two values: **0 (locked)** and **1 (unlocked)**. Functions like a **mutex**.
Counting Semaphore	Can take values **greater than 1** to allow multiple processes to access resources concurrently.

🔷 **Binary semaphores** are often used for **mutual exclusion (mutex-like behavior).**
🔷 **Counting semaphores** are used when multiple instances of a resource are available (e.g., **a pool of printers**).

---

3. Semaphore Operations (Wait & Signal)

Semaphores work using two atomic operations:

1 `wait(S)` (also called `P(S)`)

- Decrements the semaphore (`S = S - 1`)
- If `S < 0`, the process is **blocked** (waiting for the resource).

2 `signal(S)` (also called `V(S)`)

- Increments the semaphore (`S = S + 1`)
- If `S ≤ 0`, a waiting process is **unblocked**.

Example (Pseudo Code)

```
wait(S) {
    while (S <= 0);  // Wait (Busy Waiting)
    S = S - 1;
}

signal(S) {
    S = S + 1;
}
```

🚨 **Problem:** The above approach causes **busy waiting**, wasting CPU cycles. Instead, we use **blocking semaphores**.

---

4. Example: Mutual Exclusion using Semaphores

A semaphore can be used to **control access** to a shared resource.

C Program (Using Binary Semaphore for Mutual Exclusion)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex; // Declare semaphore

void *critical_section(void *arg) {
    sem_wait(&mutex); // Wait (lock)
    printf("Process %d is in the critical section\n", *(int *)arg);
    sleep(1);
    printf("Process %d is leaving the critical section\n", *(int *)arg);
    sem_post(&mutex); // Signal (unlock)
}

int main() {
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    sem_init(&mutex, 0, 1); // Initialize semaphore to 1

    pthread_create(&t1, NULL, critical_section, &id1);
    pthread_create(&t2, NULL, critical_section, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    sem_destroy(&mutex); // Destroy semaphore
    return 0;
}
```

✅ **Ensures mutual exclusion**
🚀 **Prevents race conditions**

5. Example: Producer-Consumer Problem using Semaphores

The **Producer-Consumer Problem** is a classic synchronization problem where:
🔷 The **Producer** produces data and puts it in a buffer.
🔷 The **Consumer** removes data from the buffer.
🔷 The buffer has a limited size.

Solution using Semaphores

- `mutex` ensures **mutual exclusion** in buffer access.
- `empty` ensures the buffer **doesn't overflow**.
- `full` ensures the consumer **doesn't consume from an empty buffer**.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

sem_t empty, full, mutex; // Semaphores
int buffer[BUFFER_SIZE];
int count = 0;

void *producer(void *arg) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&empty);  // Wait for empty space
        sem_wait(&mutex);  // Lock buffer access

        buffer[count++] = i; // Produce item
        printf("Produced: %d\n", i);

        sem_post(&mutex);  // Unlock buffer
        sem_post(&full);   // Signal that buffer has data
        sleep(1);
    }
}

void *consumer(void *arg) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);  // Wait for data
        sem_wait(&mutex); // Lock buffer access

        int item = buffer[--count]; // Consume item
        printf("Consumed: %d\n", item);

        sem_post(&mutex); // Unlock buffer
        sem_post(&empty); // Signal that space is available
        sleep(1);
    }
}
```

```
int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

✅ **Solves Producer-Consumer Problem**
🚀 **Ensures synchronized buffer access**

---

6. Classical Problems Solved Using Semaphores

Semaphores help solve common **process synchronization problems**, such as:

Problem	Description
Producer-Consumer	Ensures a producer does not overwrite data and a consumer does not consume an empty buffer.
Readers-Writers	Synchronizes multiple readers and writers accessing shared data.
Dining Philosophers	Prevents deadlock and starvation in processes competing for resources.

---

7. Advantages & Disadvantages of Semaphores

Advantages

✓ **Efficient** in solving synchronization problems.
✓ **Can handle multiple processes** (unlike mutex, which is limited to two).
✓ **Allows both mutual exclusion & signaling mechanisms**.

❌ Disadvantages

🚨 **Busy Waiting in some cases** – Inefficient CPU utilization.
🚨 **Risk of Deadlock** – If semaphores are not properly used.
🚨 **Complex Implementation** – Requires careful handling to avoid errors like **priority inversion** and **starvation**.

---

8. Deadlocks and Starvation with Semaphores

1. Deadlock Condition

- If two processes hold semaphores and wait for each other to release them, a **deadlock** occurs.
- Example:
  - Process 1: `wait(A); wait(B);`
  - Process 2: `wait(B); wait(A);`
- Both processes **wait indefinitely** (deadlock).

2. Starvation Condition

- If a process **never gets a chance** to acquire the semaphore (because other processes keep holding it), **starvation** occurs.
- Example:
  - A low-priority process is waiting for a semaphore **while high-priority processes keep executing**.

✅ **Solution:** Use **priority scheduling** or **Fair Semaphores (FIFO Queue)**.

---

9. Conclusion

- **Semaphores are crucial for synchronization** in multi-threaded and multi-process environments.
- They help prevent **race conditions, deadlocks, and inconsistent data access**.
- **Binary semaphores** are used for **mutual exclusion**, while **counting semaphores** are used for **resource management**.
- Proper handling is required to **avoid deadlocks and starvation**.

Methods for Inter-Process Communication (IPC) in OS

1. Introduction to IPC

Inter-Process Communication (IPC) refers to the mechanisms that allow processes to **exchange data** and **coordinate actions** in an operating system. Since processes run **independently**, IPC enables them to **share data, synchronize operations, and communicate efficiently**.

📌 **Why is IPC needed?**

- Allows **data exchange** between processes.
- Enables **synchronization** (e.g., producer-consumer problem).
- Prevents **race conditions** in shared memory.
- Helps in **distributed computing**.

---

2. Methods of IPC

IPC mechanisms are classified into two categories:
🔷 **Direct IPC:** Processes communicate directly with each other.
🔷 **Indirect IPC:** Uses a mediator (e.g., shared memory or message queues).

IPC Method	Description	Uses
Pipes	Unidirectional, uses file descriptors.	Parent-child communication.
Named Pipes (FIFO)	Bidirectional, works between unrelated processes.	Used in shell scripting.
Message Queues	Stores messages in a queue for asynchronous communication.	Used in client-server applications.
Shared Memory	Fastest IPC, allows processes to read/write shared memory.	High-speed data exchange.
Semaphores	Synchronization mechanism to prevent race conditions.	Process and thread synchronization.
Sockets	Enables communication over a network.	Used in client-server networking.
Signals	Sends notifications between processes.	Used for process termination and event handling.

---

3. IPC Mechanisms in Detail

1️⃣ Pipes (Unnamed Pipes)

🔷 **Unidirectional** communication channel.
🔷 Works **only between related processes** (parent-child).
🔷 Uses **file descriptors** (read/write ends).

🖊 Example (C Program)

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2]; // File descriptors for pipe
    char buffer[20];

    pipe(fd); // Create a pipe

    write(fd[1], "Hello", 6); // Write to pipe
    read(fd[0], buffer, sizeof(buffer)); // Read from pipe

    printf("Received: %s\n", buffer);
    return 0;
}
```

✅ **Fast and simple**
🚨 **Limited to parent-child processes**

2️⃣ Named Pipes (FIFO)

🔷 **Bidirectional** communication.
🔷 Works between **unrelated processes**.
🔷 Uses a **file in the filesystem**.

🖊 Example

```
mkfifo mypipe  # Create a named pipe
echo "Hello" > mypipe  # Write to pipe
cat < mypipe  # Read from pipe
```

✅ **Works across processes**
🚨 **Slower than unnamed pipes**

3️⃣ Message Queues

🔷 Stores **messages in a queue**.
🔷 Allows **asynchronous** communication.
🔷 Uses **msgget(), msgsnd(), msgrcv()** system calls.

🔧 Example (C Program)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct message {
    long type;
    char text[100];
};

int main() {
    int msgid = msgget(1234, 0666 | IPC_CREAT);
    struct message msg;

    msg.type = 1;
    sprintf(msg.text, "Hello from message queue!");
    msgsnd(msgid, &msg, sizeof(msg), 0);

    msgrcv(msgid, &msg, sizeof(msg), 1, 0);
    printf("Received: %s\n", msg.text);

    return 0;
}
```

✅ **Supports multiple processes**
🚨 **Limited message size**

4️⃣ Shared Memory

🔷 **Fastest IPC method**.
🔷 Allows **multiple processes to access the same memory region**.
🔷 Uses **shmget(), shmat(), shmdt(), shmctl()** system calls.

🔧 Example (C Program)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    int shmid = shmget(1234, 1024, 0666 | IPC_CREAT);
    char *shm = (char *)shmat(shmid, NULL, 0);
```

```
    strcpy(shm, "Hello from Shared Memory!");
    printf("Shared Memory contains: %s\n", shm);

    shmdt(shm); // Detach
    return 0;
}
```

✅ **Fastest IPC method**

Requires synchronization (use semaphores)

---

5 Semaphores

🔷 **Used for synchronization** between processes.
🔷 Prevents **race conditions**.
🔷 Uses **semget(), semop(), semctl()** system calls.

🔧 Example (C Program)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main() {
    int semid = semget(1234, 1, 0666 | IPC_CREAT);

    struct sembuf wait = {0, -1, 0}; // P operation
    struct sembuf signal = {0, 1, 0}; // V operation

    semop(semid, &wait, 1); // Wait
    printf("Critical Section\n");
    semop(semid, &signal, 1); // Signal

    return 0;
}
```

✅ **Ensures synchronization**

More complex than mutexes

---

6 Sockets

🔷 **Allows communication over a network**.
🔷 Supports **local and remote processes**.
🔷 Uses **socket(), bind(), listen(), accept(), send(), recv()** system calls.

🔗 Example (Python Server & Client)

Server

```
import socket

s = socket.socket()
s.bind(('localhost', 12345))
s.listen(5)

conn, addr = s.accept()
print("Connected by", addr)
print(conn.recv(1024).decode())

conn.close()
s.close()
```

Client

```
import socket

s = socket.socket()
s.connect(('localhost', 12345))
s.send(b"Hello from client!")

s.close()
```

Works across networks

Slower than shared memory

---

7 Signals

🔷 **Sends simple notifications between processes**.

🔷 Example signals:

- **SIGKILL** (Terminate process)
- **SIGSTOP** (Stop process)
- **SIGCONT** (Resume process)

🔗 Example (C Program)

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handler(int sig) {
    printf("Received signal %d\n", sig);
}

int main() {
    signal(SIGINT, handler);
    while (1) sleep(1);
}
```

✅ **Lightweight notification system**
🚨 **Limited communication capability**

---

4. Comparison of IPC Methods

Method	Speed	Complexity	Synchronization	Used For
Pipes	Fast	Simple	No	Parent-child processes
Named Pipes	Moderate	Medium	No	Unrelated processes
Message Queues	Moderate	Medium	Yes	Asynchronous messaging
Shared Memory	Fastest	Complex	Yes	High-speed data exchange
Semaphores	Slow	Complex	Yes	Synchronization
Sockets	Slowest	High	No	Network communication
Signals	Fast	Simple	No	Process notification

---

5. Conclusion

IPC is **essential for communication** between processes in an OS.
🔷 **For fast IPC:** Use **Shared Memory**.
🔷 **For simple parent-child communication:** Use **Pipes**.
🔷 **For synchronization:** Use **Semaphores**.
🔷 **For networking:** Use **Sockets**.

Process Synchronization in OS

1. Introduction to Process Synchronization

Process Synchronization ensures that multiple processes or threads **execute concurrently without conflicts**, especially when they share resources. It prevents **race conditions**, maintains **data consistency**, and ensures correct execution in **multithreaded and multiprocess environments**.

🔷 **Why is synchronization needed?**

- Prevents **race conditions** (when multiple processes modify shared data simultaneously).

- Ensures **data consistency**.
- Avoids **deadlocks and starvation**.
- Manages **critical sections** properly.

---

2. The Critical Section Problem

A **critical section** is a portion of code where a **shared resource** (e.g., a variable, file, or database) is accessed.

Solution Requirements (Critical Section Problem Criteria - "Mutual Exclusion")

🔷 **Mutual Exclusion**: Only **one** process can execute in the critical section at a time.
🔷 **Progress**: If no process is in the critical section, then another process **must be allowed to enter**.
🔷 **Bounded Waiting**: Every process should get a chance to enter the critical section **within a finite time** (prevents starvation).

---

3. Classical Synchronization Mechanisms

Mechanism	Description	Example Usage
Peterson's Algorithm	Software-based mutual exclusion	Used in theoretical studies
Locks (Mutex)	Prevents multiple processes from entering the critical section	Used in databases, OS kernel
Semaphores	Synchronization tool using counters	Used in process management
Monitors	High-level synchronization construct	Used in Java, C++ (OOP-based)
Message Passing	Processes send messages instead of sharing memory	Used in distributed systems

---

4. Process Synchronization Approaches

1️⃣ Peterson's Algorithm (Software-Based)

🔷 **Solves critical section problem for two processes**.
🔷 Uses **flags and a turn variable** to ensure mutual exclusion.

🖊 Algorithm:

```
int flag[2] = {0, 0}; // Flags for processes
int turn = 0;  // Whose turn is it?

void process_0() {
    flag[0] = 1; // Indicate process 0 wants to enter
    turn = 1; // Give turn to process 1

    while (flag[1] == 1 && turn == 1); // Wait if process 1 is in critical
section

    // Critical Section
    flag[0] = 0; // Exit section
}
```

✅ **Simple and effective**
🚨 **Only works for two processes**

2️⃣ Locks (Mutex)

🔷 **Mutex (Mutual Exclusion Lock)** allows only one process/thread to access a resource.
🔷 A process **locks** before entering the critical section and **unlocks** after exiting.

🖊 Example (C using Pthread)

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t lock;

void* critical_section(void* arg) {
    pthread_mutex_lock(&lock);  // Lock
    printf("Thread %ld in critical section\n", (long)arg);
    pthread_mutex_unlock(&lock);  // Unlock
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);

    pthread_create(&t1, NULL, critical_section, (void*)1);
    pthread_create(&t2, NULL, critical_section, (void*)2);
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);
    return 0;
}
```

✅ **Efficient and simple**

🚨 **Leads to deadlock if not handled properly**

3️⃣ Semaphores

🔷 A **semaphore** is a counter used for synchronization.

🔷 Types:

- **Binary Semaphore** (0 or 1) → Like Mutex.
- **Counting Semaphore** (>1) → Allows multiple processes.

🔧 Example (C using Semaphores)

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>

sem_t semaphore;

void* critical_section(void* arg) {
    sem_wait(&semaphore); // Wait (P operation)
    printf("Thread %ld in critical section\n", (long)arg);
    sem_post(&semaphore); // Signal (V operation)
    return NULL;
}

int main() {
    pthread_t t1, t2;
    sem_init(&semaphore, 0, 1);

    pthread_create(&t1, NULL, critical_section, (void*)1);
    pthread_create(&t2, NULL, critical_section, (void*)2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    sem_destroy(&semaphore);
    return 0;
}
```

✅ **Used in producer-consumer problems**

🚨 **Complex and requires correct handling**

4️⃣ Monitors (High-Level Synchronization)

🔷 **Encapsulates synchronization** in an **object-oriented way**.
🔷 Used in **Java (synchronized keyword)** and C++ (mutex in classes).

🔧 Example (Java)

```
class Shared {
    synchronized void criticalSection() {
        System.out.println(Thread.currentThread().getName() + " in critical 
section");
    }
}

class MyThread extends Thread {
    Shared shared;

    MyThread(Shared obj) { shared = obj; }

    public void run() {
        shared.criticalSection();
    }
}

public class MonitorExample {
    public static void main(String[] args) {
        Shared obj = new Shared();

        MyThread t1 = new MyThread(obj);
        MyThread t2 = new MyThread(obj);

        t1.start();
        t2.start();
    }
}
```

✅ **Encapsulates synchronization logic**
🚨 **Limited to object-oriented languages**

5️⃣ Message Passing

🔷 Used in **distributed systems** where shared memory is not available.
🔷 Two operations: **send()** and **receive()**.

🔧 Example (Python using MPI)

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    comm.send("Hello from process 0", dest=1)
elif rank == 1:
    msg = comm.recv(source=0)
    print(msg)
```

✅ **Works across networked systems**
🚨 **Slower than shared memory**

---

5. Deadlocks and Starvation in Synchronization

🔷 **Deadlock**: Two or more processes **wait indefinitely** for each other.
🔷 **Starvation**: A process **waits indefinitely** due to scheduling unfairness.

🖊 Deadlock Example

```
pthread_mutex_lock(&A);
pthread_mutex_lock(&B);
// Deadlock occurs if another process locks B first and then A
```

🛠 **Solution:** Use **Deadlock Prevention & Avoidance** techniques like:
✅ **Ordered Locking** (always lock in the same order)
✅ **Deadlock Detection & Recovery**

---

6. Comparison of Synchronization Mechanisms

Method	Speed	Complexity	Best Used For
Mutex	Fast	Simple	Thread synchronization
Semaphore	Moderate	Medium	Process synchronization
Monitor	Moderate	High	Object-oriented synchronization
Peterson's Algorithm	Slow	High	Two-process synchronization
Message Passing	Slowest	High	Distributed systems

---

7. Conclusion

- **For simple synchronization → Use Mutex.**
- **For process synchronization → Use Semaphores.**
- **For object-oriented systems → Use Monitors.**
- **For distributed systems → Use Message Passing.**

Classical Process Synchronization Problems in OS

1. Introduction

Classical Process Synchronization Problems are real-world scenarios that demonstrate **race conditions** and **resource-sharing challenges** in concurrent systems. These problems help in understanding synchronization techniques like **semaphores, monitors, and mutex locks**.

2. List of Classical Synchronization Problems

◈ 1 Bounded Buffer (Producer-Consumer) Problem

◈ 2 Readers-Writers Problem

◈ 3 Dining Philosophers Problem

◈ 4 Sleeping Barber Problem

---

1 Bounded Buffer (Producer-Consumer) Problem

Problem Statement

- A **producer** creates data and stores it in a shared buffer.
- A **consumer** retrieves data from the buffer.
- **Synchronization Issues**:
  - ✅ **Producer must wait** if the buffer is **full**.
  - ✅ **Consumer must wait** if the buffer is **empty**.
  - ✅ **Access to the buffer must be mutually exclusive** (only one process modifies it at a time).

Solution: Using Semaphores

🔧 Implementation (C using Semaphores)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE], count = 0;

sem_t empty, full;
```

```
pthread_mutex_t mutex;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        sem_wait(&empty); // Wait if buffer is full
        pthread_mutex_lock(&mutex);

        buffer[count++] = item;
        printf("Produced: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&full); // Signal consumer
        item++;
    }
}

void *consumer(void *arg) {
    while (1) {
        sem_wait(&full); // Wait if buffer is empty
        pthread_mutex_lock(&mutex);

        int item = buffer[--count];
        printf("Consumed: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty); // Signal producer
    }
}

int main() {
    pthread_t prod, cons;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    return 0;
}
```

✅ **Prevents buffer overflow & underflow**
✅ **Ensures mutual exclusion**

---

2 Readers-Writers Problem

Problem Statement

- Multiple **reader processes** can read a shared file simultaneously.
- Only **one writer process** can write at a time.
- **Synchronization Issues**:
  - ✅ **Multiple readers** should be allowed concurrently.
  - ✅ **Writers must get exclusive access**.
  - ✅ **Avoid reader or writer starvation**.

Solution: Using Read-Write Locks

🔧 Implementation (C using Semaphores)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

sem_t rw_mutex, mutex;
int read_count = 0;

void *reader(void *arg) {
    while (1) {
        sem_wait(&mutex);
        read_count++;
        if (read_count == 1) sem_wait(&rw_mutex); // First reader locks write
        sem_post(&mutex);

        printf("Reader %ld reading\n", (long)arg);

        sem_wait(&mutex);
        read_count--;
        if (read_count == 0) sem_post(&rw_mutex); // Last reader unlocks
write
        sem_post(&mutex);
    }
}

void *writer(void *arg) {
    while (1) {
        sem_wait(&rw_mutex);
        printf("Writer %ld writing\n", (long)arg);
        sem_post(&rw_mutex);
    }
}

int main() {
    pthread_t r1, r2, w1;
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);

    pthread_create(&r1, NULL, reader, (void*)1);
    pthread_create(&r2, NULL, reader, (void*)2);
    pthread_create(&w1, NULL, writer, (void*)1);

    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
```

```
    pthread_join(w1, NULL);

    return 0;
}
```

✅ **Allows multiple readers but ensures single writer access**
✅ **Prevents inconsistent read/write operations**

---

3 Dining Philosophers Problem

Problem Statement

- **Five philosophers** sit at a circular table, each having a plate and a fork on the left and right.
- **To eat, a philosopher must pick up both forks** (one from the left and one from the right).
- **Synchronization Issues**:
  ✅ **Prevent deadlock** (if all philosophers pick up their left fork at the same time, no one can eat).
  ✅ **Prevent starvation** (ensure all philosophers get a chance to eat).

Solution: Using Semaphores

🔧 Implementation (C using Semaphores)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
sem_t forks[N];

void *philosopher(void *num) {
    int i = *(int *)num;

    sem_wait(&forks[i]); // Pick left fork
    sem_wait(&forks[(i+1) % N]); // Pick right fork

    printf("Philosopher %d is eating\n", i);

    sem_post(&forks[i]); // Release left fork
    sem_post(&forks[(i+1) % N]); // Release right fork
}

int main() {
    pthread_t phil[N];
    int ids[N];

    for (int i = 0; i < N; i++) sem_init(&forks[i], 0, 1);

    for (int i = 0; i < N; i++) {
```

```
        ids[i] = i;
        pthread_create(&phil[i], NULL, philosopher, &ids[i]);
    }

    for (int i = 0; i < N; i++) pthread_join(phil[i], NULL);

    return 0;
}
```

✅ **Solves deadlock by ensuring order in picking forks**
✅ **Uses semaphores to manage access**

4️⃣ Sleeping Barber Problem

Problem Statement

- A barber **sleeps** if no customers are in the waiting room.
- If a customer arrives:
  ✅ If a **chair is available**, the customer sits.
  ✅ If **no chairs are available**, the customer leaves.
  ✅ The barber cuts hair **one by one**.

Solution: Using Semaphores

🔧 Implementation (Pseudocode)

```
sem_t customers, barber, mutex;
int waiting = 0;

void *customer() {
    sem_wait(&mutex);
    if (waiting < CHAIRS) {
        waiting++;
        sem_post(&customers);
        sem_post(&mutex);
        sem_wait(&barber); // Wait for the barber
    } else {
        sem_post(&mutex);
    }
}

void *barber() {
    while (1) {
        sem_wait(&customers); // Wait for a customer
        sem_wait(&mutex);
        waiting--;
        sem_post(&barber); // Signal barber is ready
        sem_post(&mutex);
        cut_hair();
```

```
    }
}
```

✅ **Ensures barber sleeps when no customers are present**
✅ **Prevents customers from entering if no chairs are available**

---

5. Conclusion

Problem	Synchronization Tool Used	Main Issue Solved
Producer-Consumer	Semaphores	Buffer overflow & underflow
Readers-Writers	Read-Write Locks	Prevents data inconsistency
Dining Philosophers	Semaphores	Prevents deadlock
Sleeping Barber	Semaphores	Prevents race conditions & starvation

Producer-Consumer Problem in OS

1. Introduction

The **Producer-Consumer Problem** is a classic synchronization problem in Operating Systems where two types of processes—**producers** and **consumers**—share a common buffer.

- The **producer** generates items and places them in the buffer.
- The **consumer** removes items from the buffer for processing.

Synchronization Issues

1. **Buffer Overflow** – If the producer keeps adding items when the buffer is full.
2. **Buffer Underflow** – If the consumer tries to remove an item when the buffer is empty.
3. **Mutual Exclusion** – Only one process should access the buffer at a time to avoid race conditions.

2. Solutions Using Synchronization Techniques

To handle synchronization issues, we use:

- **Semaphores**
- **Mutex Locks**
- **Monitors & Condition Variables**

---

3. Solution Using Semaphores

Semaphores help coordinate producer and consumer operations by signaling when the buffer has space or data.
We define three semaphores:

- **`empty`** – Counts empty slots in the buffer.
- **`full`** – Counts filled slots in the buffer.
- **`mutex`** – Ensures mutual exclusion when accessing the buffer.

🔗 C Implementation Using Semaphores

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE], count = 0;

sem_t empty, full;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int item = 1;
    while (1) {
        sem_wait(&empty); // Wait if buffer is full
        pthread_mutex_lock(&mutex); // Lock buffer

        buffer[count++] = item;
        printf("Produced: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&full); // Signal consumer

        item++;
        sleep(1);
    }
}

void *consumer(void *arg) {
    while (1) {
        sem_wait(&full); // Wait if buffer is empty
        pthread_mutex_lock(&mutex); // Lock buffer

        int item = buffer[--count];
        printf("Consumed: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty); // Signal producer

        sleep(2);
    }
}
```

```
int main() {
    pthread_t prod, cons;
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    return 0;
}
```

◈ Explanation

✅ **Ensures Mutual Exclusion** – Uses `pthread_mutex_lock` for buffer access.
✅ **Prevents Buffer Overflow & Underflow** – Uses `empty` and `full` semaphores.
✅ **Synchronization Maintained** – The consumer waits if the buffer is empty, and the producer waits if it's full.

---

4. Solution Using Monitors (Java Implementation)

Monitors simplify synchronization by providing condition variables.

🔗 Java Implementation Using Monitors

```
class Buffer {
    private final int[] buffer;
    private int count = 0;

    public Buffer(int size) {
        buffer = new int[size];
    }

    public synchronized void produce(int item) throws InterruptedException {
        while (count == buffer.length) {
            wait(); // Wait if buffer is full
        }
        buffer[count++] = item;
        System.out.println("Produced: " + item);
        notify(); // Notify consumers
    }

    public synchronized int consume() throws InterruptedException {
        while (count == 0) {
            wait(); // Wait if buffer is empty
        }
        int item = buffer[--count];
        System.out.println("Consumed: " + item);
        notify(); // Notify producers
```

```
        return item;
    }
}

class Producer extends Thread {
    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        int item = 1;
        while (true) {
            try {
                buffer.produce(item++);
                sleep(1000);
            } catch (InterruptedException e) { }
        }
    }
}

class Consumer extends Thread {
    private final Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        while (true) {
            try {
                buffer.consume();
                sleep(2000);
            } catch (InterruptedException e) { }
        }
    }
}

public class ProducerConsumerMonitor {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(5);
        new Producer(buffer).start();
        new Consumer(buffer).start();
    }
}
```

◈ Explanation

✅ **Monitors use `wait()` and `notify()` instead of semaphores.**
✅ **Simplifies thread synchronization with Java's `synchronized` methods.**
✅ **Ensures only one thread modifies the buffer at a time.**

---

5. Conclusion

Method	Synchronization Tool Used	Key Benefits
Semaphores	`sem_wait()`, `sem_post()`	Ensures mutual exclusion & prevents race conditions
Mutex Locks	`pthread_mutex_lock()`	Prevents data corruption
Monitors	`wait()`, `notify()`	Simplifies synchronization

📌 Real-World Applications

- **CPU Scheduling** – OS uses a queue (buffer) to manage processes.
- **Message Queues in Networking** – Producers generate messages, consumers process them.
- **File System Caching** – Producers write to cache, consumers read from it.

Reader-Writer Problem in OS

1. Introduction

The **Reader-Writer Problem** is a classic process synchronization problem in Operating Systems that deals with multiple **reader** and **writer** processes accessing a shared resource (such as a database or file).

Problem Definition

- **Readers** can read the shared resource **simultaneously** without conflicts.
- **Writers** must have **exclusive access** (only one writer at a time).
- **Readers and Writers should not cause data inconsistency or race conditions**.

Synchronization Issues

1. **Multiple readers should be allowed** at the same time.
2. **Writers must have exclusive access** to avoid inconsistent updates.
3. **Prevent starvation** – Writers should not wait indefinitely if readers keep entering.

---

2. Variants of Reader-Writer Problem

✅ First Reader-Writer Problem (Readers Priority)

- Multiple readers can read simultaneously.
- If a writer wants to write, it **waits until all readers finish**.
- Issue: **Writers may starve** if readers keep arriving continuously.

✅ Second Reader-Writer Problem (Writers Priority)

- If a writer is waiting, **no new readers should be allowed**.
- Readers must wait until the writer finishes.
- Issue: **Readers may starve** if writers keep arriving.

✅ Third Reader-Writer Problem (Fair Approach)

- Uses **FIFO (First-Come, First-Serve) queue**.
- Neither readers nor writers suffer starvation.

---

3. Solution Using Semaphores

We define three semaphores:

- **`mutex`** → Ensures mutual exclusion for reader count updates.
- **`wrt`** → Ensures **exclusive** access for writers.
- **`read_count`** → Tracks the number of active readers.

🔧 C Implementation Using Semaphores

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

sem_t wrt;           // Writer semaphore
pthread_mutex_t mutex; // Mutex for reader count
int read_count = 0; // Number of active readers

void *reader(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        read_count++;
        if (read_count == 1)
            sem_wait(&wrt);  // First reader blocks writers
        pthread_mutex_unlock(&mutex);

        // Reading section
        printf("Reader %ld is reading\n", (long)arg);
        sleep(1);

        pthread_mutex_lock(&mutex);
        read_count--;
        if (read_count == 0)
            sem_post(&wrt);  // Last reader allows writers
        pthread_mutex_unlock(&mutex);

        sleep(1);
    }
```

```
}

void *writer(void *arg) {
    while (1) {
        sem_wait(&wrt);  // Only one writer at a time

        // Writing section
        printf("Writer %ld is writing\n", (long)arg);
        sleep(2);

        sem_post(&wrt);  // Release lock for other writers/readers

        sleep(1);
    }
}

int main() {
    pthread_t r[5], w[2];
    sem_init(&wrt, 0, 1);
    pthread_mutex_init(&mutex, NULL);

    for (long i = 0; i < 5; i++)
        pthread_create(&r[i], NULL, reader, (void *)i);
    for (long i = 0; i < 2; i++)
        pthread_create(&w[i], NULL, writer, (void *)i);

    for (int i = 0; i < 5; i++)
        pthread_join(r[i], NULL);
    for (int i = 0; i < 2; i++)
        pthread_join(w[i], NULL);

    return 0;
}
```

◈ Explanation

✅ **Multiple readers can read together.**
✅ **Only one writer can write at a time.**
✅ **First reader blocks writers; last reader releases writers.**

---

4. Solution Using Monitors (Java)

Monitors use **`wait()` and `notify()`** instead of semaphores.

🔗 Java Implementation Using Monitors

```
class ReaderWriterMonitor {
    private int readCount = 0;
    private boolean writing = false;

    public synchronized void startRead() throws InterruptedException {
        while (writing) {
```

```
            wait();  // Wait if a writer is writing
        }
        readCount++;
    }

    public synchronized void endRead() {
        readCount--;
        if (readCount == 0) {
            notifyAll();  // Notify waiting writers
        }
    }

    public synchronized void startWrite() throws InterruptedException {
        while (writing || readCount > 0) {
            wait();  // Wait if readers or another writer is active
        }
        writing = true;
    }

    public synchronized void endWrite() {
        writing = false;
        notifyAll();  // Notify readers and writers
    }
}

class Reader extends Thread {
    private final ReaderWriterMonitor monitor;

    public Reader(ReaderWriterMonitor monitor) {
        this.monitor = monitor;
    }

    public void run() {
        try {
            while (true) {
                monitor.startRead();
                System.out.println(Thread.currentThread().getName() + " is
reading");
                sleep(1000);
                monitor.endRead();
                sleep(1000);
            }
        } catch (InterruptedException e) { }
    }
}

class Writer extends Thread {
    private final ReaderWriterMonitor monitor;

    public Writer(ReaderWriterMonitor monitor) {
        this.monitor = monitor;
    }

    public void run() {
        try {
            while (true) {
                monitor.startWrite();
```

```
                System.out.println(Thread.currentThread().getName() + " is 
writing");
                sleep(2000);
                monitor.endWrite();
                sleep(1000);
            }
        } catch (InterruptedException e) { }
    }
}

public class ReaderWriter {
    public static void main(String[] args) {
        ReaderWriterMonitor monitor = new ReaderWriterMonitor();

        for (int i = 0; i < 5; i++) new Reader(monitor).start();
        for (int i = 0; i < 2; i++) new Writer(monitor).start();
    }
}
```

🔷 Explanation

✅ **Uses `wait()` and `notifyAll()`** to manage synchronization.
✅ **Ensures mutual exclusion for writers**.
✅ **Prevents race conditions and starvation**.

---

5. Comparison of Approaches

Approach	Tool Used	Key Features
Semaphores	`sem_wait(), sem_post()`	More control but harder to implement correctly
Mutex Locks	`pthread_mutex_lock()`	Simple but does not solve starvation
Monitors (Java)	`wait(), notifyAll()`	Easier to use and manage synchronization

---

6. Real-World Applications

📌 **Database Management Systems (DBMS)** – Multiple users (readers) querying the database while transactions (writers) update it.
📌 **File System Access** – Multiple processes reading logs while a single process updates them.
📌 **Networking Protocols** – Clients (readers) retrieving data while servers (writers) update records.

🚀 **Understanding Reader-Writer problems is crucial for designing multi-threaded applications and database management systems!**

UNIT – IV

Memory Management in OS

1. Introduction

Memory management in an operating system (OS) is responsible for **allocating, managing, and deallocating memory** efficiently for processes. It ensures that multiple programs can run **concurrently** without interference.

Goals of Memory Management

✓ **Efficient Resource Utilization** – Ensure memory is used optimally.
✓ **Process Isolation** – Prevent one process from interfering with another.
✓ **Fast Access** – Minimize memory access delays.
✓ **Multiprogramming Support** – Allow multiple processes to execute simultaneously.

---

2. Types of Memory in OS

Memory Type	Description
Primary Memory (RAM)	Fast, volatile memory used for executing processes.
Cache Memory	High-speed memory between CPU and RAM to speed up access.
Virtual Memory	Extends RAM using disk storage (swap space).
Secondary Storage (HDD/SSD)	Used for permanent data storage.

---

3. Memory Management Techniques

1 Contiguous Memory Allocation

- Processes are assigned **continuous** blocks of memory.
- **Two types:**
    1. **Fixed Partitioning** – Memory is divided into fixed-sized partitions.
    2. **Variable Partitioning** – Partitions are created dynamically as needed.

Advantages

✓ Simple to implement.
✓ Efficient for small systems.

Disadvantages

❌ **Fragmentation Issues**

- **Internal Fragmentation** – Wasted memory inside a partition.
- **External Fragmentation** – Free memory scattered across different locations.

---

2 Paging

- Memory is divided into **fixed-size blocks** called **pages** (process side) and **frames** (RAM side).
- **No external fragmentation**, but **internal fragmentation may occur**.

Paging Table

- Maintains a **mapping** between **logical pages** and **physical frames**.
- **Page Table Entry (PTE)** contains:
  - Page Number
  - Frame Number
  - Valid/Invalid Bit

Advantages

✓ Eliminates **external fragmentation**.
✓ Supports **virtual memory**.
✓ Efficient memory utilization.

Disadvantages

❌ **Page Table Overhead** – Requires additional memory for the page table.
❌ **TLB Misses** – Translation Lookaside Buffer (TLB) is used to speed up address translation.

---

3 Segmentation

- Memory is divided into **variable-sized segments** based on **logical divisions** of a program (code, stack, heap).
- Each segment has a **base address** and **limit** (size).

Advantages

✓ Supports **modular programming**.
✓ Reduces **internal fragmentation**.

Disadvantages

❌ Causes **external fragmentation**.
❌ Segment table overhead.

---

4 Virtual Memory

- Uses **disk space** as an extension of **RAM**.
- **Swapping**: Inactive pages are moved to disk, and active pages are brought into RAM.
- Implemented using **Demand Paging** and **Page Replacement Algorithms**.

Advantages

✓ Allows running **large processes** with limited RAM.
✓ Enables **multiprogramming**.

Disadvantages

❌ Slower than physical RAM (Disk I/O is slow).
❌ **Thrashing** – Excessive swapping reduces performance.

---

4. Page Replacement Algorithms

When a new page is needed, the OS decides **which page to replace** in RAM.

Algorithm	Description
FIFO (First-In, First-Out)	Replaces the oldest page. Simple but inefficient.
LRU (Least Recently Used)	Replaces the least recently used page. Better than FIFO.
Optimal Page Replacement	Replaces the page that will not be used for the longest time. **Ideal but impossible to implement.**

Algorithm	Description
Clock Algorithm	Uses a reference bit to approximate LRU.

---

5. Memory Allocation Strategies

1 Single Contiguous Allocation

- Only **one** process runs at a time.
- **Example:** Early DOS systems.

2 Multiple Partition Allocation

- Divides memory into **fixed or dynamic** partitions.
- Uses **allocation strategies** like:
  - **First Fit** – Allocates first available block.
  - **Best Fit** – Allocates the smallest suitable block.
  - **Worst Fit** – Allocates the largest available block.

---

6. Swapping

- Process **moves in and out of RAM** between execution.
- **Swap Space** is used on disk.
- Helps in **multiprogramming**.

---

7. Thrashing (Performance Issue in Virtual Memory)

- Occurs when **excessive page swapping** reduces CPU efficiency.
- **Causes:**
  1. **High multiprogramming** (too many processes).
  2. **Insufficient RAM**.
- **Solutions:**
  ✓ Reduce multiprogramming.
  ✓ Improve **page replacement algorithms**.
  ✓ Increase **physical memory**.

---

8. Real-World Applications of Memory Management

📌 **Operating Systems (Windows, Linux, macOS)** – Uses paging and virtual memory.
📌 **Databases (MySQL, Oracle)** – Implements memory caching and page replacement.
📌 **Embedded Systems (IoT, Smartphones)** – Uses efficient memory allocation due to hardware constraints.

🚀 **Understanding memory management helps in OS optimization and system design!**

Physical and Virtual Address Space in OS

1. Introduction

In an operating system, memory is accessed using two different types of addresses:
✔ **Physical Address** – Actual location in RAM.
✔ **Virtual Address** – Logical address used by a process.

The OS maps virtual addresses to physical addresses using **Memory Management Unit (MMU)**.

---

2. Physical Address Space

- The **actual memory locations** in RAM where data and instructions reside.
- Generated after address translation by the MMU.
- Used by the hardware (CPU, RAM).

Example:

If a process needs to access **memory location 2000**, the CPU accesses the **physical memory at address 2000**.

---

3. Virtual Address Space

- A **logical address space** used by processes.
- Each process gets its own **isolated** virtual address space.
- Helps in **security, isolation, and efficient memory management**.

Example:

If a process tries to access memory **location 2000**, it refers to a **virtual address (e.g., 2000 in its own space)**, which is then mapped to an actual **physical address** (e.g., 5000 in RAM).

---

4. Key Differences Between Physical and Virtual Address Space

Feature	Physical Address	Virtual Address
Definition	Actual location in RAM	Logical address used by processes
Visibility	Used by hardware	Used by the OS and user programs
Isolation	Shared among all processes	Each process has its own virtual space
Security	No protection	Provides process isolation
Mapping	Directly used by CPU	Translated via MMU to physical address
Flexibility	Limited to available RAM	Can be larger than RAM (uses disk for virtual memory)

---

5. Address Translation (Virtual → Physical)

- The **Memory Management Unit (MMU)** converts **virtual addresses** into **physical addresses**.
- Used in **paging and segmentation**.
- Helps in **address protection and security**.

Translation Process Example:

1. **Process requests memory** at virtual address `VA = 2000`.
2. **MMU checks page table** and maps `VA = 2000` to `PA = 5000`.
3. **CPU accesses** physical memory at `PA = 5000`.

---

6. Virtual Addressing and Paging

To manage virtual memory, the OS uses **paging**, which divides memory into:
✓ **Pages** (Virtual Memory Side)
✓ **Frames** (Physical Memory Side)

- A **page table** stores the mapping of **virtual pages** to **physical frames**.
- The **Translation Lookaside Buffer (TLB)** speeds up address translation.

---

7. Benefits of Virtual Address Space

✅ **Process Isolation** – Prevents one process from accessing another's memory.
✅ **Efficient Memory Utilization** – Supports virtual memory, allowing processes to use more memory than physically available.
✅ **Security** – Protects memory from unauthorized access.
✅ **Flexibility** – Enables processes to run without worrying about physical memory locations.

---

8. Real-World Applications

📌 **Multitasking OS (Windows, Linux, macOS)** – Uses virtual memory for running multiple processes.
📌 **Cloud Computing** – Uses address virtualization for efficient resource allocation.
📌 **Databases (MySQL, Oracle)** – Uses memory mapping for fast access.

🚀 **Understanding Physical vs. Virtual Address Space is crucial for memory management in modern operating systems!**

Memory Allocation Strategies in OS

1. Introduction

Memory allocation strategies determine how the OS assigns memory blocks to processes efficiently. The primary goal is to **maximize memory utilization** and **reduce fragmentation** while ensuring smooth execution of processes.

---

2. Types of Memory Allocation

1️⃣ Contiguous Memory Allocation

- Memory is allocated in **a single continuous block** for each process.
- Used in early OS and simple memory models.
- Can lead to **fragmentation** (wasted memory).

Types of Contiguous Allocation:

✔ **Fixed Partitioning:** Memory is divided into fixed-sized blocks.
✔ **Variable Partitioning:** Partitions are created dynamically based on process size.

Advantages:

✓ Simple to implement.
✓ Efficient for small systems.

Disadvantages:

✗ Causes **fragmentation** (internal/external).
✗ Limited support for multiprogramming.

---

2 Non-Contiguous Memory Allocation

- Memory is allocated **in separate blocks**, which may not be adjacent.
- Supports **paging and segmentation**.
- Reduces fragmentation and allows **better memory utilization**.

Advantages:

✓ Supports **virtual memory**.
✓ Avoids **external fragmentation**.

Disadvantages:

✗ **Complex address translation** required.
✗ Requires **page tables** or **segment tables**.

---

3. Dynamic Memory Allocation Strategies

When allocating memory to processes, the OS follows different strategies to find a suitable block in free memory.

1 First Fit

- Allocates the **first available** memory block that is large enough.
- **Fast but may cause fragmentation.**

✓ **Advantage:** Simple and fast.
✗ **Disadvantage:** Causes **external fragmentation**.

---

2 Best Fit

- Allocates the **smallest available block** that fits the process.
- Reduces **internal fragmentation**, but may leave **small unusable gaps**.

✓ **Advantage:** Less wasted memory.
✗ **Disadvantage:** Slower than First Fit.

---

3 Worst Fit

- Allocates the **largest available block**, leaving the maximum remaining free space.
- Tries to **minimize fragmentation**, but is generally inefficient.

✓ **Advantage:** Leaves larger free blocks for future allocation.
✗ **Disadvantage:** Leads to **fragmentation** over time.

---

4 Next Fit

- Similar to **First Fit**, but starts searching **from the last allocated block** instead of the beginning.
- Improves efficiency in some scenarios.

✓ **Advantage:** Avoids scanning from the start every time.
✗ **Disadvantage:** Can still cause fragmentation.

---

4. Fragmentation in Memory Allocation

1 Internal Fragmentation

- Occurs when allocated memory **is larger** than the required memory.
- **Example:** A process needs **8 KB**, but is assigned a **12 KB block**.

2 External Fragmentation

- Occurs when **free memory blocks are scattered**, but none are large enough for a new process.
- **Example:** Several **small free spaces** exist, but none can fit a large process.

Solution: Compaction

- The OS **shifts processes** to create **a larger continuous free block**.
- Works only if memory is **relocatable**.

---

5. Real-World Applications of Memory Allocation Strategies

📌 **Operating Systems (Windows, Linux, macOS)** – Uses dynamic allocation strategies.
📌 **Databases (MySQL, Oracle)** – Allocates memory dynamically for caching.
📌 **Embedded Systems (IoT, Smartphones)** – Requires efficient memory allocation due to hardware constraints.

🚀 **Choosing the right allocation strategy is key to optimizing memory utilization in an OS!**

Fixed and Variable Partitions in OS

1. Introduction

In **contiguous memory allocation**, memory is divided into **partitions** to allocate space for processes. Two common partitioning methods are:

✓ **Fixed Partitioning** – Memory is divided into fixed-sized blocks.
✓ **Variable Partitioning** – Partitions are created dynamically based on process size.

---

2. Fixed Partitioning

- Also known as **static partitioning**.
- The **total memory is divided into fixed-size partitions** at system startup.
- Each partition **can hold only one process**.

Example:

If memory is **16 MB** and divided into **four 4 MB partitions**, a process requiring **5 MB** cannot be allocated.

Advantages:

✓ **Simple and easy to implement**.
✓ **Fast allocation** since partitions are predefined.

Disadvantages:

❌ **Internal Fragmentation** – If a process requires **2 MB** but is allocated **4 MB**, the extra **2 MB is wasted**.

❌ **Limited process flexibility** – Large processes may not fit in any partition.
❌ **Inefficient memory utilization** – Some partitions may remain **empty**.

---

3. Variable Partitioning

- Also called **dynamic partitioning**.
- Memory is divided into **partitions of variable size** based on process requirements.
- **No predefined partitions** – partitions are created dynamically when processes arrive.

Example:

If memory is **16 MB**, and a process requires **6 MB**, it gets exactly **6 MB**, leaving **10 MB free** for other processes.

Advantages:

✔ **No Internal Fragmentation** – Memory is allocated **exactly** as needed.
✔ **Better memory utilization** – Efficiently allocates memory to fit processes.
✔ **Supports large processes** since partitions are flexible.

Disadvantages:

❌ **External Fragmentation** – Free memory gets **scattered**, making it hard to allocate large processes.
❌ **Memory Compaction Needed** – To merge free spaces, the OS must **shift processes** (which is time-consuming).

---

4. Key Differences Between Fixed and Variable Partitioning

Feature	Fixed Partitioning	Variable Partitioning
Definition	Memory is divided into fixed-size partitions at startup.	Partitions are created dynamically based on process size.
Flexibility	Not flexible – partition sizes cannot change.	Flexible – partitions adjust dynamically.
Fragmentation	Causes **internal fragmentation**.	Causes **external fragmentation**.
Memory Utilization	Less efficient due to unused space.	More efficient but may require

Feature	Fixed Partitioning	Variable Partitioning
		compaction.
Process Size Handling	Large processes **may not fit** if no large partition exists.	Large processes can be allocated if enough free memory exists.
Implementation Complexity	Simple to implement.	More complex due to dynamic allocation.

---

5. Real-World Applications

📌 **Fixed Partitioning** – Used in early OS like **DOS** and simple embedded systems.
📌 **Variable Partitioning** – Used in **modern OS** (Windows, Linux) to optimize memory allocation.

🚀 **Variable partitioning is widely used today due to better memory efficiency and flexibility!**

Paging in OS

1. Introduction

Paging is a **non-contiguous memory allocation technique** used by operating systems to manage memory efficiently. It allows processes to be stored in **separate, non-contiguous blocks**, eliminating **external fragmentation**.

---

2. Key Concepts in Paging

🔷 Page and Page Frame

- **Page:** A fixed-size block of a process's virtual address space.
- **Page Frame:** A fixed-size block in **physical memory (RAM)**.

🔷 Page Table

- A data structure that maps **virtual page numbers** to **physical frame numbers**.
- Stored in **main memory** and managed by the **Memory Management Unit (MMU)**.

---

3. How Paging Works?

Step-by-Step Process:

1 **Divide** process memory into **pages** (e.g., 4 KB each).
2 **Divide** physical memory into **frames** of the **same size**.
3 **Page Table** maps **pages** to **frames** in RAM.
4 **When a process accesses a page**, the MMU translates the virtual address to a physical address.

Example:

- **Process Size:** 10 KB
- **Page Size:** 4 KB → Process needs **3 pages**.
- **Frames:** Can be anywhere in RAM, e.g., Page 0 → Frame 5, Page 1 → Frame 2, Page 2 → Frame 9.

---

4. Address Translation in Paging

A **virtual address** is divided into:
✓ **Page Number (PN)** – Index in the **page table**.
✓ **Page Offset (PO)** – Specifies the exact location within a page.

Formula:

Physical Address = (Frame Number × Page Size) + Page Offset

Example Calculation:

- **Virtual Address:** `VA = 2200`
- **Page Size:** `1024 Bytes (1 KB)`
- **Page Number (PN) = 2200 ÷ 1024 = 2 (Quotient)`**
- **Offset = 2200 % 1024 = 152 (Remainder)`**
- **If Page 2 is in Frame 7, Physical Address = (7 × 1024) + 152 = 7416**

---

5. Types of Paging

1 Simple Paging

- Each process gets **a page table**.
- Uses **one-level** mapping.

2 Multi-Level Paging

- **Hierarchical paging** (e.g., 2-level or 3-level) to handle large address spaces efficiently.
- Example: Used in **64-bit systems** where single-level paging would require **huge page tables**.

3 Inverted Paging

- Instead of **one page table per process**, maintains **one global page table**.
- **Reduces memory overhead** but is **slower** in lookups.

---

6. Advantages & Disadvantages of Paging

✅ Advantages:

✓ **Eliminates External Fragmentation** – No need for memory compaction.
✓ **Efficient Memory Utilization** – Supports **multiprogramming**.
✓ **Allows Virtual Memory** – Uses disk as an extension of RAM.

❌ Disadvantages:

❌ **Internal Fragmentation** – Last page may have unused space.
❌ **Page Table Overhead** – Page tables require memory space.
❌ **Translation Overhead** – Accessing page tables adds an extra step in memory access.

---

7. Paging vs. Segmentation

Feature	Paging	Segmentation
Memory Division	Fixed-size pages	Variable-size segments
Fragmentation	Internal fragmentation	External fragmentation
Logical Division	No logical grouping	Groups related data together
Used In	Most modern OS	Used with paging in some OS (e.g., Linux)

---

8. Real-World Applications of Paging

📌 **Windows, Linux, macOS** – Use paging for **virtual memory management**.
📌 **Embedded Systems** – Use paging to optimize **limited memory**.
📌 **Cloud Computing** – Paging helps in **efficient VM memory allocation**.

🚀 **Paging is widely used in modern operating systems to improve memory management and eliminate external fragmentation!**

Segmentation in OS

1. Introduction

Segmentation is a **memory management technique** that divides a process into **variable-sized segments** based on logical divisions, such as functions, arrays, or data structures. Unlike paging (which divides memory into **fixed-size** pages), segmentation ensures that related data is stored together, reducing unnecessary memory usage.

---

2. Key Concepts in Segmentation

🔷 Segment and Segment Table

- **Segment:** A logically divided portion of a program, such as code, stack, heap, or data.
- **Segment Table:** A table that stores the **base address** and **limit (size)** of each segment.

---

3. How Segmentation Works?

Step-by-Step Process:

1️⃣ A process is divided into **multiple segments** (e.g., Code, Stack, Heap).
2️⃣ The **Segment Table** stores the **base address** and **limit** for each segment.
3️⃣ A process requests memory using **segment number & offset**.
4️⃣ The OS translates the **logical address** (segment number + offset) into a **physical address**.

---

4. Address Translation in Segmentation

A **logical address** in segmentation consists of:
✔ **Segment Number (SN)** – Identifies the segment.
✔ **Offset (O)** – Specifies the exact location within the segment.

Formula:

Physical Address = Base Address (from Segment Table) + Offset

Example Calculation:

- **Segment Table:**
  - Segment 0 (Code): **Base = 4000, Limit = 1000**
  - Segment 1 (Stack): **Base = 8000, Limit = 2000**
- **Logical Address:** (Segment = 1, Offset = 500)
- **Physical Address:** `8000 + 500 = 8500`

---

5. Types of Segmentation

1 Simple Segmentation

- Each segment is a **variable size**.
- Used in basic **memory management** systems.

2 Segmentation with Paging (Modern OS)

- **Combines segmentation & paging** to reduce fragmentation.
- Each segment is divided into **pages** for efficient memory allocation.
- Example: **Linux** uses **segmentation for logical division** and **paging for physical memory allocation**.

---

6. Advantages & Disadvantages of Segmentation

✅ Advantages:

✔ **Logical Organization** – Groups related data logically (code, stack, heap).
✔ **Efficient Memory Usage** – No **internal fragmentation** (unlike paging).
✔ **Easier Sharing & Protection** – Allows segment-based access control.

❌ Disadvantages:

❌ **External Fragmentation** – Free memory gets scattered.
❌ **Compaction Required** – OS must rearrange segments to free up memory.
❌ **Variable Segment Sizes** – Complex memory management.

---

7. Paging vs. Segmentation

Feature	Paging	Segmentation
Memory Division	Fixed-size pages	Variable-size segments
Fragmentation	Internal fragmentation	External fragmentation
Logical Grouping	No logical meaning	Groups related data
Address Mapping	Uses **page table**	Uses **segment table**
Used In	Most modern OS	Used with paging in some OS (e.g., Linux)

---

8. Real-World Applications of Segmentation

📌 **Unix & Linux** – Use segmentation **for logical division** and **paging for memory allocation**.
📌 **Multitasking Systems** – Segmentation helps in **efficient process management**.
📌 **Database Systems** – Stores **large datasets** logically.

🚀 **Segmentation helps in organizing memory efficiently by grouping logically related components!**

Virtual Memory in OS

1. Introduction

Virtual Memory is a memory management technique that allows a computer to execute processes that require more memory than is physically available. It creates an **illusion of a larger memory** by using **secondary storage (e.g., Hard Disk, SSD) as an extension of RAM**.

---

2. Key Concepts of Virtual Memory

🔷 Logical vs. Physical Address

- **Logical Address (Virtual Address):** The address generated by the CPU.
- **Physical Address:** The actual address in RAM where data is stored.

🔷 Demand Paging

- Only the required **pages** of a process are loaded into RAM.
- The rest of the process remains in **secondary storage** until needed.

🔷 Page Replacement

- If RAM is full, the OS **removes an old page** to load a new page.
- **Algorithms used:** FIFO, LRU, Optimal Page Replacement.

🔷 Thrashing

- If page swapping happens **too frequently**, the system slows down.
- Caused by **insufficient RAM** or too many processes.

---

3. How Virtual Memory Works?

Step-by-Step Process:

1 The **CPU generates a logical address**.
2 The **MMU (Memory Management Unit)** translates the logical address to a **physical address** using the **page table**.
3 If the required page is **in RAM**, it is accessed.
4 If the required page is **not in RAM (page fault)**, the OS loads it from the **hard disk** (swap space).

Example:

- **Process Size:** 8 MB
- **RAM Available:** 4 MB
- The OS loads only the required **pages** of the process into RAM, keeping the rest in **secondary storage**.

---

4. Advantages & Disadvantages of Virtual Memory

✅ Advantages:

✔ **Allows Running Large Programs** – Even if RAM is small.
✔ **Efficient Memory Utilization** – Only necessary pages are loaded.
✔ **Multi-Tasking Support** – Multiple processes can run simultaneously.

❌ Disadvantages:

❌ **Slower Performance** – Accessing disk is slower than RAM.
❌ **Page Faults Increase Latency** – Frequent page swapping slows the system.
❌ **Thrashing** – Too much swapping causes performance issues.

---

5. Virtual Memory vs. Physical Memory

Feature	Virtual Memory	Physical Memory (RAM)
Location	Uses both RAM & disk	Only in RAM
Size	Larger than physical memory	Limited by hardware
Access Speed	Slower (due to disk)	Faster
Used When	RAM is insufficient	Directly used by CPU

---

6. Real-World Applications of Virtual Memory

📌 **Operating Systems (Windows, Linux, macOS)** – Use virtual memory to handle large processes.
📌 **Cloud Computing** – Virtual memory optimizes **resource allocation** in VMs.
📌 **Gaming & High-Performance Apps** – Uses **swap files/page files** to handle memory-intensive tasks.

🚀 **Virtual Memory is essential for running large applications without requiring large RAM!**

UNIT V

File and I/O Management & OS Security in OS

1. File Management in OS

File Management refers to the system used by the OS to store, organize, retrieve, and manage files efficiently.

◈ Key Concepts in File Management

- **File** – A collection of related data stored on secondary storage.
- **Directory** – A logical structure to organize files.
- **File System** – Defines how files are stored and retrieved (e.g., FAT32, NTFS, ext4).

◈ File Operations

- **Create** – Making a new file.
- **Read** – Accessing file data.
- **Write** – Modifying or adding data.
- **Delete** – Removing files.
- **Rename** – Changing the file name.

◈ File Access Methods

1 **Sequential Access** – Files are read in order (e.g., text files).
2 **Direct Access** – Can access any part of the file directly (e.g., databases).

◈ File Allocation Methods

- **Contiguous Allocation** – Stores file in consecutive blocks.
- **Linked Allocation** – Stores blocks in a linked list.
- **Indexed Allocation** – Uses an index table to store block addresses.

---

2. I/O Management in OS

I/O Management controls communication between the OS and **input/output devices** (e.g., keyboard, disk, network).

🔷 I/O System Components

✓ **Device Drivers** – Software that enables OS to communicate with hardware.
✓ **Interrupts** – Signals that notify the CPU about I/O events.
✓ **Buffers** – Temporary storage for data transfer.
✓ **DMA (Direct Memory Access)** – Transfers data without CPU intervention.

🔷 Types of I/O Operations

1️⃣ **Programmed I/O** – CPU waits for I/O operations.
2️⃣ **Interrupt-Driven I/O** – CPU gets interrupted when I/O is ready.
3️⃣ **DMA (Direct Memory Access)** – Transfers data directly between devices and memory.

---

3. OS Security

OS Security protects the **system, data, and processes** from threats like unauthorized access, malware, and data breaches.

🔷 Security Threats in OS

❌ **Unauthorized Access** – Hackers accessing sensitive data.
❌ **Malware & Viruses** – Malicious programs affecting system operations.
❌ **Data Breach** – Exposure of confidential data.
❌ **Denial of Service (DoS) Attacks** – Overloading a system to make it unavailable.

🔷 OS Security Mechanisms

✓ **Authentication** – Verifies user identity (e.g., passwords, biometrics).
✓ **Access Control** – Restricts unauthorized access (e.g., file permissions).
✓ **Encryption** – Converts data into unreadable form for security.
✓ **Firewall & Antivirus** – Protects against malware and network threats.

🔷 User & Process Security

- **User Privileges** – Admin vs. regular user restrictions.
- **Process Isolation** – Prevents one process from interfering with another.
- **Security Patches** – Regular updates to fix vulnerabilities.

🚀 **OS Security ensures data safety, system integrity, and reliable performance!**

Directory Structure in OS

1. Introduction

A **directory** is a system used by an Operating System (OS) to organize and manage files in a structured way. It allows users to store, retrieve, and access files efficiently.

---

2. Types of Directory Structures

1️⃣ Single-Level Directory

📌 **Structure:** All files are stored in a **single** directory.
📌 **Advantages:**
✔ Simple and easy to manage.
✔ Fast file access.
📌 **Disadvantages:**
❌ Name conflicts if multiple users exist.
❌ Difficult to organize files.

2️⃣ Two-Level Directory

📌 **Structure:** Each user has a **separate** directory under a common root.
📌 **Advantages:**
✔ Avoids file name conflicts.
✔ User-specific directories improve security.
📌 **Disadvantages:**
❌ Cannot share files between users easily.
❌ More complex than a single-level directory.

3️⃣ Tree Directory

📌 **Structure:** Files and directories form a **hierarchical tree** structure.
📌 **Advantages:**
✔ Supports efficient **file grouping and organization**.
✔ Allows **file sharing** using links.
📌 **Disadvantages:**
❌ Requires more management (e.g., permission control).

4 Acyclic Graph Directory

📌 **Structure:** Similar to a tree but **allows multiple parent directories** using links (symbolic or hard links).
📌 **Advantages:**
✓ Supports **file sharing** efficiently.
✓ Reduces storage duplication.
📌 **Disadvantages:**
✕ Managing links can be complex.
✕ Deleting a file may create **dangling references**.

5 General Graph Directory

📌 **Structure:** A more flexible version of the acyclic graph, allowing **cyclic paths** (links can create loops).
📌 **Advantages:**
✓ Most **flexible directory structure**.
✓ Supports **advanced file sharing mechanisms**.
📌 **Disadvantages:**
✕ **Cycle detection required** to prevent infinite loops.
✕ **Harder to manage permissions & access control**.

---

3. Directory Operations

✓ **Create a directory** – Make a new folder.
✓ **Delete a directory** – Remove an existing directory.
✓ **Rename a directory** – Change the directory name.
✓ **List files** – Display contents of a directory.
✓ **Search for files** – Locate files in the directory structure.

---

4. Real-World Examples of Directory Structures

📌 **Windows OS** – Uses a **tree-structured directory** (`C:\Users\Documents`).
📌 **Linux/Unix OS** – Uses a **hierarchical structure** (`/home/user/docs`).
📌 **Cloud Storage (Google Drive, OneDrive)** – Uses **graph-based structures** for file sharing.

🚀 **A well-organized directory structure improves file management, security, and accessibility!**

File Operations in OS

1. Introduction

A **file** is a collection of data stored on secondary storage. The **Operating System (OS)** provides various file operations to manage files effectively.

---

2. Basic File Operations

◈ 1. Create a File

- Generates a new file in the file system.
- The OS assigns a **name** and allocates space.
- Example command:
  - **Windows:** `type nul > filename.txt`
  - **Linux:** `touch filename.txt`

◈ 2. Open a File

- Loads the file into memory for reading or writing.
- The OS checks **permissions** and **file attributes**.
- Example in C:
- `FILE *fp;`
- `fp = fopen("file.txt", "r");  // Opens file in read mode`

◈ 3. Read from a File

- Retrieves data from the file.
- Example in Python:
- `with open("file.txt", "r") as f:`
- `    data = f.read()`
- `    print(data)`

◈ 4. Write to a File

- Modifies or adds data to the file.
- Example in Python:
- `with open("file.txt", "w") as f:`
- `    f.write("Hello, World!")`

◈ 5. Append Data to a File

- Adds new data **without overwriting existing content**.
- Example in C:
- `FILE *fp = fopen("file.txt", "a");`
- `fprintf(fp, "New data");`
- `fclose(fp);`

🔹 6. Close a File

- Saves changes and releases memory.
- Example in C:
- `fclose(fp);`

🔹 7. Delete a File

- Removes the file from the storage system.
- Example command:
    - **Windows:** `del filename.txt`
    - **Linux:** `rm filename.txt`

---

3. Advanced File Operations

🔹 8. Rename a File

- Changes the file name.
- Example in Linux:
- `mv oldname.txt newname.txt`

🔹 9. Copy a File

- Creates a duplicate of an existing file.
- Example command:
    - **Windows:** `copy source.txt destination.txt`
    - **Linux:** `cp source.txt destination.txt`

🔹 10. Move a File

- Transfers a file to a different directory.
- Example command:
    - **Windows:** `move file.txt C:\new_folder`
    - **Linux:** `mv file.txt /new_directory/`

---

4. File Operations in Real-world OS

📌 **Windows OS** – GUI-based file operations (drag & drop, right-click options).
📌 **Linux OS** – Command-line and shell scripts for automation.
📌 **Databases & Cloud Storage** – Automatic file handling using APIs.

🚀 **Efficient file operations ensure data integrity, security, and performance in any OS!**

File Allocation Methods in OS

1. Introduction

File allocation refers to how disk space is assigned to files. The **Operating System (OS)** uses different allocation methods to store and retrieve files efficiently while minimizing fragmentation.

---

2. Types of File Allocation Methods

1 Contiguous Allocation

📌 **Definition:**

- Each file is stored in **consecutive memory blocks** on the disk.
- The OS assigns **a fixed set of contiguous blocks** to a file when it is created.

📌 **Example:**

File	Start Block	Length (Blocks)
A	5	3
B	8	2

📌 **Advantages:**
✓ **Fast access** – Easy to read sequential files.
✓ **Simple implementation** – Only starting block and length are needed.

📌 **Disadvantages:**
✕ **External fragmentation** – Hard to allocate space for new files.
✕ **File size must be known in advance** – Difficult for growing files.

---

2 Linked Allocation

📌 **Definition:**

- Each file is a **linked list of disk blocks**, where each block contains a **pointer** to the next block.
- The OS maintains a **File Allocation Table (FAT)** to track block locations.

📌 **Example:**

File	Block Sequence
A	5 → 12 → 9
B	8 → 14 → 20

📌 **Advantages:**
✓ **No external fragmentation** – Files can grow dynamically.
✓ **Efficient disk space utilization**.

📌 **Disadvantages:**
❌ **Slow access speed** – Must follow pointers to find data.
❌ **Extra space needed for pointers** – Reduces usable storage.
❌ **Not suitable for random access** – Must traverse the linked list.

---

3 Indexed Allocation

📌 **Definition:**

- The OS creates an **index block** for each file that stores **pointers to disk blocks**.
- Allows **direct access** to file blocks.

📌 **Example:**

File	Index Block	Data Blocks
A	7	5, 12, 9
B	10	8, 14, 20

📌 **Advantages:**
✓ **Supports direct access** – Faster than linked allocation.
✓ **Efficient for both small and large files**.

📌 **Disadvantages:**
❌ **Index block overhead** – Wastes space if files are small.
❌ **Limited block entries per index block** – May require multi-level indexing for large files.

---

3. Comparison of File Allocation Methods

Method	Speed	Fragmentation	Suitable for	Random Access
Contiguous	Fastest	External	Static files (e.g., OS files)	✅ Yes
Linked	Slow	No	Dynamic files (e.g., logs)	❌ No
Indexed	Moderate	No	Large and dynamic files	✅ Yes

---

4. Real-World Usage of File Allocation Methods

📌 **Contiguous Allocation** – Used in **CD-ROMs, DVDs** for fast sequential access.
📌 **Linked Allocation** – Used in **FAT (File Allocation Table)** file systems (e.g., older Windows systems).
📌 **Indexed Allocation** – Used in **NTFS, ext4, Unix-based file systems**.

🚀 **Choosing the right file allocation method ensures efficient disk space utilization and fast file access!**

Device Management in OS

1. Introduction

📌 **Device Management** is a crucial function of an **Operating System (OS)** that handles **input/output (I/O) devices** like keyboards, printers, hard drives, and USBs.
📌 The OS ensures **efficient communication** between hardware and software through **device drivers and I/O control mechanisms**.

---

2. Functions of Device Management

1️⃣ **Device Detection** – Identifies connected devices.
2️⃣ **Device Communication** – Enables interaction between OS and devices.
3️⃣ **Device Allocation** – Assigns devices to processes.
4️⃣ **Device Scheduling** – Prioritizes device requests.
5️⃣ **Device Protection & Security** – Prevents unauthorized access.
6️⃣ **Error Handling** – Detects and corrects device failures.

---

3. Device Management Components

1️⃣ I/O Devices

- **Input Devices**: Keyboard, Mouse, Scanner, etc.
- **Output Devices**: Monitor, Printer, Speakers, etc.
- **Storage Devices**: Hard Drives, SSDs, USBs.

2️⃣ Device Drivers

- A **software program** that helps the OS communicate with hardware.
- Example: Printer driver translates print commands for a printer.

3️⃣ I/O Controller

- A **hardware component** that manages data transfer between devices and CPU.
- Example: **Disk Controller** handles HDD/SSD operations.

---

4. I/O Management Techniques

1️⃣ Programmed I/O (Polling)

✔ The CPU repeatedly checks the device status.
✔ **Simple but inefficient** (wastes CPU time).
✔ Used in **basic systems** like embedded devices.

2️⃣ Interrupt-Driven I/O

✔ Devices send **interrupts** to notify the CPU when they need attention.
✔ **Efficient** – CPU doesn't waste time checking devices.
✔ Used in **modern OS like Windows & Linux**.

3️⃣ Direct Memory Access (DMA)

✔ **High-speed data transfer** between devices & memory **without CPU intervention**.
✔ Improves system performance.
✔ Used in **HDDs, GPUs, and network cards**.

---

5. Device Scheduling Algorithms

Used for **disk and I/O request management**.

Algorithm	Description	Use Case
FCFS (First-Come, First-Served)	Processes I/O requests in order received	Simple, but slow
SSTF (Shortest Seek Time First)	Picks request closest to current disk position	Fast but may starve long requests
SCAN (Elevator Algorithm)	Moves in one direction, then reverses	Efficient for large loads
C-SCAN (Circular SCAN)	Moves in one direction, resets to start	Avoids starvation
LOOK & C-LOOK	Like SCAN & C-SCAN but stops at last request	Optimized disk scheduling

---

6. Device Management in Real-world OS

📌 **Windows** – Uses **Plug and Play (PnP)** for automatic device recognition.
📌 **Linux** – Uses `/dev` directory to represent devices as files.
📌 **MacOS** – Uses **I/O Kit** for device communication.

🚀 **Effective device management ensures smooth hardware-software interaction, improving system performance and efficiency!**

Pipes in Operating System

1. Introduction

📌 **Pipes** are an **Inter-Process Communication (IPC)** mechanism that allows **data transfer between processes** in an operating system.
📌 They enable **one process to send data to another** through a communication channel.
📌 Commonly used in **Linux/Unix shells** for chaining commands.

---

2. Types of Pipes

1 Anonymous Pipes

✓ **Used for communication between related processes (Parent-Child)**.
✓ **Unidirectional** – Data flows in only **one direction**.
✓ Exists **only during process execution**.
✓ Used in **shell commands** like:

```
ls | wc -l
```

- The `ls` command lists files.
- The `wc -l` command counts the number of lines.

✓ **Example in C (Anonymous Pipe)**

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2]; // File descriptors for pipe
    pipe(fd);
    if (fork() == 0) {  // Child process
        close(fd[0]);   // Close read end
        write(fd[1], "Hello", 5);
        close(fd[1]);
    } else {  // Parent process
        close(fd[1]);  // Close write end
        char buffer[10];
        read(fd[0], buffer, 5);
        printf("Received: %s\n", buffer);
        close(fd[0]);
    }
    return 0;
}
```

---

2 Named Pipes (FIFOs - First In First Out)

✓ **Allows communication between unrelated (independent) processes**.
✓ **Bidirectional** – Can be used for **both reading and writing**.
✓ Exists **beyond process execution** (remains in file system).
✓ Created using the `mkfifo` command in Linux.

✓ **Example in Linux**

```
mkfifo mypipe   # Create a named pipe
echo "Hello" > mypipe  # Write data to the pipe
cat < mypipe  # Read data from the pipe
```

✓ Example in C (Named Pipe - FIFO)

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    char *fifo = "/tmp/myfifo";
    mkfifo(fifo, 0666);

    char message[100];
    int fd = open(fifo, O_WRONLY);
    write(fd, "Hello FIFO", 10);
    close(fd);

    fd = open(fifo, O_RDONLY);
    read(fd, message, 10);
    printf("Received: %s\n", message);
    close(fd);

    return 0;
}
```

---

3. Comparison: Anonymous vs. Named Pipes

Feature	Anonymous Pipe	Named Pipe (FIFO)
Communication	Between **related processes**	Between **unrelated processes**
Direction	**Unidirectional**	**Bidirectional**
Lifetime	Exists **only during execution**	**Persists in file system**
Access	No file system entry	Appears as a **file in the system**

---

4. Advantages of Pipes

✓ **Simple & Efficient** – Easy to use for process communication.
✓ **Fast** – Direct memory transfer between processes.
✓ **Synchronization** – Ensures ordered execution of processes.

5. Disadvantages of Pipes

❌ **Limited Scope** – Cannot be used for remote process communication.
❌ **Unidirectional (Anonymous Pipes)** – Named Pipes solve this issue.
❌ **No Persistence (Anonymous Pipes)** – Named Pipes provide persistence.

🚀 **Pipes are a fundamental IPC mechanism for process communication in Unix/Linux and play a crucial role in scripting, automation, and multi-processing!**

Buffer in Operating System

1. Introduction

📌 A **buffer** is a temporary memory storage area used to hold data while it is being transferred between two locations.
📌 It helps in **handling speed mismatches** between processes, devices, or networks.
📌 Buffers are widely used in **I/O operations, inter-process communication (IPC), and memory management**.

---

2. Purpose of Buffers

✔ **Smoothens data transfer** – Manages differences in processing speed between sender and receiver.
✔ **Handles burst data** – Stores data temporarily to prevent data loss.
✔ **Enhances performance** – Reduces frequent I/O access and CPU wait time.
✔ **Supports multitasking** – Allows processes to work independently.

---

3. Types of Buffers

1 Single Buffer

✔ The OS allocates **one buffer** in main memory for I/O operations.
✔ Data is stored **temporarily** before being processed.

📌 **Example:**

- A **keyboard buffer** stores user keystrokes before they are processed.

📌 **Advantages:**
✔ Simple and easy to implement.

📌 **Disadvantages:**
❌ Inefficient for large data transfers.
❌ CPU must wait if buffer is full.

---

2 Double Buffering (Ping-Pong Buffering)

✓ Uses **two buffers** to allow continuous data transfer.
✓ While one buffer is being **processed**, the other buffer **receives new data**.

📌 **Example:**

- **Video streaming**: One buffer loads the next frame while the current frame is displayed.

📌 **Advantages:**
✓ Increases efficiency by overlapping I/O and processing.
✓ Reduces waiting time for CPU.

📌 **Disadvantages:**
❌ Uses **more memory** than a single buffer.

---

3 Circular Buffer (Ring Buffer)

✓ A **fixed-size** buffer organized as a **circular queue**.
✓ When the buffer is full, **new data overwrites the oldest data**.

📌 **Example:**

- **Audio processing**: Continuous audio recording uses a circular buffer.

📌 **Advantages:**
✓ Prevents data loss in real-time applications.
✓ Efficient memory utilization.

📌 **Disadvantages:**
❌ Requires **complex buffer management**.

---

4 Spooling (Simultaneous Peripheral Operation On-Line)

✓ A special type of buffer used for **storing I/O requests in a queue** before execution.

📌 **Example:**

- **Print spooling**: Documents are stored in a buffer before printing.

📌 **Advantages:**
✓ Allows **multiple processes** to submit print jobs without waiting.

📌 **Disadvantages:**
✗ Requires additional **disk space** for spooling queue.

---

4. Buffering vs Caching

Feature	Buffering	Caching
Purpose	Handles **speed mismatch** between sender and receiver	Stores **frequently accessed** data for quick retrieval
Data Processing	Temporary storage for **incomplete data**	Holds **previously used data** for reuse
Example	Keyboard buffer, Printer spooler	Web cache, CPU cache

---

5. Advantages of Buffering

✓ **Increases system performance** by reducing I/O waits.
✓ **Smoothens data flow** between slow and fast devices.
✓ **Prevents data loss** in real-time applications.
✓ **Improves CPU utilization** by allowing multitasking.

6. Disadvantages of Buffering

✗ **Consumes memory** – Large buffers reduce available RAM.
✗ **Increases system complexity** – Buffer management requires additional processing.
✗ **May cause delays** – If buffer filling takes time, processing is delayed.

Shared Memory in Operating System

1. Introduction

📌 **Shared Memory** is an **Inter-Process Communication (IPC)** mechanism that allows **multiple processes to access a common memory segment** for fast data exchange.
📌 Instead of passing messages, processes **read/write directly into the shared memory**.
📌 **Fastest IPC method** because it avoids kernel intervention after initialization.

---

2. How Shared Memory Works?

✓ The **OS allocates a segment of memory** that multiple processes can access.
✓ One process **writes data** into shared memory.
✓ Other processes **read data** from the same memory.
✓ Requires **synchronization** (like Semaphores) to avoid conflicts.

---

3. Implementation of Shared Memory (Example in C - Linux)

Step 1: Create Shared Memory Segment

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
    key_t key = ftok("shmfile", 65); // Generate unique key
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT); // Create shared memory
    char *data = (char *) shmat(shmid, (void *)0, 0); // Attach to process

    strcpy(data, "Hello, Shared Memory!"); // Write data
    printf("Data written: %s\n", data);

    shmdt(data); // Detach memory
    return 0;
}
```

Step 2: Read Shared Memory in Another Process

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, 1024, 0666);
    char *data = (char *) shmat(shmid, (void *)0, 0);

    printf("Data read: %s\n", data); // Read data
```

```
    shmdt(data);
    shmctl(shmid, IPC_RMID, NULL); // Remove shared memory
    return 0;
}
```

---

4. Advantages of Shared Memory

✓ **Fastest IPC mechanism** – No kernel involvement after setup.
✓ **Efficient for large data transfer** – Avoids message copying overhead.
✓ **Low CPU usage** – Reduces system calls for communication.

5. Disadvantages of Shared Memory

❌ **Requires synchronization** – Need semaphores/mutex to avoid conflicts.
❌ **Security concerns** – Any process with access can modify data.
❌ **Complexity** – Managing multiple processes accessing shared memory is challenging.

🚀 **Shared memory is widely used in databases, multi-threading, and real-time systems for efficient inter-process communication!**

Disk Scheduling Algorithms in OS

1. Introduction

📌 Disk Scheduling is the process of **determining the order of disk I/O requests** to improve access time and throughput.
📌 The OS manages disk scheduling to **reduce seek time** (the time taken for the disk arm to move to the requested track).
📌 **Seek Time >> Rotational Latency >> Transfer Time**, so reducing seek time is the primary goal.

---

2. Disk Scheduling Algorithms

1️⃣ First-Come, First-Served (FCFS)

✓ **Requests are served in the order they arrive.**
✓ **Simple but inefficient** as it may cause long seek times.

📌 **Example:**
🔷 **Queue:** 98, 183, 37, 122, 14, 124, 65, 67

🔷 **Head Position:** 53
🔷 **Total Seek Time:** `|53-98| + |98-183| + |183-37| + ...`

📌 **Advantages:**
✓ Simple & Fair (No starvation).
📌 **Disadvantages:**
❌ High seek time (Inefficient).

2️⃣ Shortest Seek Time First (SSTF)

✓ **Processes the request closest to the current head position first.**
✓ **Reduces seek time but may cause starvation.**

📌 **Example:**
🔷 **Queue:** 98, 183, 37, 122, 14, 124, 65, 67
🔷 **Head Position:** 53
🔷 **Closest request:** 37 → 14 → 65 → 67 → …

📌 **Advantages:**
✓ **Less seek time** than FCFS.
📌 **Disadvantages:**
❌ **Starvation** (Far requests may never be served).

3️⃣ SCAN (Elevator Algorithm)

✓ **Moves towards one direction, servicing requests, then reverses direction.**
✓ **Better than SSTF in avoiding starvation.**

📌 **Example:**
🔷 **Queue:** 98, 183, 37, 122, 14, 124, 65, 67
🔷 **Head Position:** 53
🔷 **Direction:** Right
🔷 **Processing Order:** 65 → 67 → 98 → 122 → 124 → 183 → (Reverse) → 37 → 14

📌 **Advantages:**
✓ Reduces long waiting times.
📌 **Disadvantages:**
❌ **More seek time** than LOOK.

---

4 LOOK (Optimized SCAN)

✓ **Same as SCAN but stops at the last request instead of going to the disk end.**
✓ **Improves efficiency over SCAN.**

📌 **Example:**
🔷 **Stops at 183 instead of reaching max cylinder.**

📌 **Advantages:**
✓ Reduces unnecessary movement compared to SCAN.
📌 **Disadvantages:**
❌ May still suffer from **starvation**.

---

5 C-SCAN (Circular SCAN)

✓ **Only moves in one direction and jumps back to the start.**
✓ **Provides uniform waiting time.**

📌 **Example:**
🔷 **Head moves right → reaches max → jumps to the start → continues servicing.**

📌 **Advantages:**
✓ **Fair and prevents starvation.**
📌 **Disadvantages:**
❌ **Jumping back increases seek time.**

---

6 C-LOOK (Optimized C-SCAN)

✓ **Similar to C-SCAN but stops at the last request before jumping back.**

📌 **Advantages:**
✓ More efficient than C-SCAN.
📌 **Disadvantages:**
❌ May still involve long jumps.

---

3. Comparison of Disk Scheduling Algorithms

Algorithm	Seek Time	Starvation	Direction	Use Case
FCFS	**High**	No	No direction	Simple but inefficient
SSTF	**Low**	Yes	No direction	Good for small workloads
SCAN	**Medium**	No	Two directions	Elevator-type workloads
LOOK	**Better than SCAN**	No	Two directions	Optimized version of SCAN
C-SCAN	**Medium**	No	One direction	Fair waiting time
C-LOOK	**Better than C-SCAN**	No	One direction	Optimized version of C-SCAN

🚀 **Choosing the best algorithm depends on workload, fairness, and efficiency needs!**

LAB PROGRAMS

1. Illustrate the LINUX commands

a) pwd

b) mkdir

c) rmdir

d) grep

e) chmod

f) ls

g) rm

h) cp

Here is an illustration and explanation of the given **Linux commands**:

1. `pwd` (Print Working Directory)

- **Usage**: Displays the current working directory.
- **Command**:
- `pwd`
- **Example Output**:
- `/home/user/Documents`

---

2. `mkdir` (Make Directory)

- **Usage**: Creates a new directory.
- **Command**:
- `mkdir new_folder`
- **Example**:
  Creates a directory named `new_folder` in the current location.

---

3. `rmdir` (Remove Directory)

- **Usage**: Deletes an empty directory.
- **Command**:
- `rmdir empty_folder`
- **Example**:
  Removes `empty_folder` if it is empty.

---

4. `grep` (Global Regular Expression Print)

- **Usage**: Searches for a pattern in a file.
- **Command**:
- `grep "hello" file.txt`
- **Example Output** (if found in `file.txt`):
- `hello world`

---

5. `chmod` (Change File Permissions)

- **Usage**: Changes file permissions.
- **Command**:
- `chmod 755 script.sh`
- **Example**:
  Grants read, write, and execute permission to the owner, and read & execute to others.

---

6. `ls` (List Directory Contents)

- **Usage**: Lists files and directories.
- **Command**:
- `ls -l`
- **Example Output**:
- `-rw-r--r--  1 user user  1234 Mar 03 10:00 file.txt`
- `drwxr-xr-x  2 user user  4096 Mar 03 09:30 folder/`

---

7. `rm` (Remove File/Directory)

- **Usage**: Deletes a file or directory.
- **Command**:
- `rm file.txt`
- **To remove a directory and its contents**:
- `rm -r directory_name`

---

8. `cp` (Copy File/Directory)

- **Usage**: Copies files or directories.
- **Command**:
- `cp file1.txt backup/`
- **Example**:
  Copies `file1.txt` to the `backup/` directory.

Let me know if you need further details! 🚀

2. Write a program to calculate average waiting time and turn around time of each process using the following CPU Scheduling algorithm for the given process schedules.

a) FCFS

b) SJF

c) Priority

d) Round Robin

```
import java.util.Arrays;

import java.util.Scanner;


class Process {

   int pid, bt, at, priority, wt, tat, ct;


   Process(int pid, int bt, int at, int priority) {

      this.pid = pid;

      this.bt = bt;

      this.at = at;

      this.priority = priority;

   }

}


public class CPUScheduling {


   static void calculateWTAndTAT(Process[] processes, int n) {

      int totalWT = 0, totalTAT = 0;

      System.out.println("\nProcess\tAT\tBT\tWT\tTAT");
```

```
    for (Process p : processes) {
        p.tat = p.ct - p.at;
        p.wt = p.tat - p.bt;
        totalWT += p.wt;
        totalTAT += p.tat;
        System.out.println(p.pid + "\t" + p.at + "\t" + p.bt + "\t" + p.wt + "\t" + p.tat);
    }
    System.out.println("Average Waiting Time: " + (float) totalWT / n);
    System.out.println("Average Turnaround Time: " + (float) totalTAT / n);
}

static void FCFS(Process[] processes, int n) {
    Arrays.sort(processes, (a, b) -> a.at - b.at);
    int currentTime = 0;
    for (Process p : processes) {
        if (currentTime < p.at)
            currentTime = p.at;
        p.ct = currentTime + p.bt;
        currentTime = p.ct;
    }
    calculateWTAndTAT(processes, n);
}

static void SJF(Process[] processes, int n) {
    Arrays.sort(processes, (a, b) -> a.bt - b.bt);
```

```
    int currentTime = 0;
    for (Process p : processes) {
        if (currentTime < p.at)
            currentTime = p.at;
        p.ct = currentTime + p.bt;
        currentTime = p.ct;
    }
    calculateWTAndTAT(processes, n);
}

static void PriorityScheduling(Process[] processes, int n) {
    Arrays.sort(processes, (a, b) -> a.priority - b.priority);
    int currentTime = 0;
    for (Process p : processes) {
        if (currentTime < p.at)
            currentTime = p.at;
        p.ct = currentTime + p.bt;
        currentTime = p.ct;
    }
    calculateWTAndTAT(processes, n);
}

static void RoundRobin(Process[] processes, int n, int quantum) {
    int[] remainingBT = new int[n];
    for (int i = 0; i < n; i++)
```

```
        remainingBT[i] = processes[i].bt;

    int currentTime = 0, completed = 0;
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (remainingBT[i] > 0) {
                int executeTime = Math.min(remainingBT[i], quantum);
                currentTime += executeTime;
                remainingBT[i] -= executeTime;
                if (remainingBT[i] == 0) {
                    processes[i].ct = currentTime;
                    completed++;
                }
            }
        }
    }
    calculateWTAndTAT(processes, n);
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter number of processes: ");
    int n = sc.nextInt();
    Process[] processes = new Process[n];
```

```
    System.out.println("Enter Process ID, Arrival Time, Burst Time, and Priority:");
    for (int i = 0; i < n; i++) {
        int pid = sc.nextInt();
        int at = sc.nextInt();
        int bt = sc.nextInt();
        int priority = sc.nextInt();
        processes[i] = new Process(pid, bt, at, priority);
    }

    System.out.println("Choose Scheduling Algorithm:\n1. FCFS\n2. SJF\n3. Priority\n4. Round Robin");
    int choice = sc.nextInt();

    switch (choice) {
        case 1 -> FCFS(processes, n);
        case 2 -> SJF(processes, n);
        case 3 -> PriorityScheduling(processes, n);
        case 4 -> {
            System.out.print("Enter Time Quantum: ");
            int quantum = sc.nextInt();
            RoundRobin(processes, n, quantum);
        }
        default -> System.out.println("Invalid choice!");
    }
    sc.close();
}
```

}

Output :-

Enter number of processes: 3

Enter Process ID, Arrival Time, Burst Time, and Priority:

1 0 5 3

2 1 3 1

3 2 8 2

Choose Scheduling Algorithm:

1. FCFS

2. SJF

3. Priority

4. Round Robin

2

Process AT BT WT TAT

2 1 3 0 3

1 0 5 3 8

3 2 8 8 16

Average Waiting Time: 3.67

Average Turnaround Time: 9.00

3.Simulate MVT and MFT memory management techniques

```
import java.util.Scanner;

import java.util.ArrayList;


class Process {

    int id, size, allocatedPartition;


    Process(int id, int size) {

        this.id = id;

        this.size = size;

        this.allocatedPartition = -1;

    }

}


class MemoryPartition {

    int size, allocatedProcess;


    MemoryPartition(int size) {

        this.size = size;

        this.allocatedProcess = -1;

    }

}


public class MemoryManagementSimulation {
```

```
static void MVT(ArrayList<Process> processes, int totalMemory) {
    int usedMemory = 0;

    System.out.println("\nMVT (Multiprogramming with Variable Tasks) Simulation:");
    for (Process p : processes) {
        if (usedMemory + p.size <= totalMemory) {
            System.out.println("Process " + p.id + " allocated " + p.size + " KB");
            usedMemory += p.size;
        } else {
            System.out.println("Process " + p.id + " cannot be allocated due to insufficient memory.");
        }
    }
    System.out.println("Total Memory Used: " + usedMemory + " KB");
    System.out.println("Total Memory Free: " + (totalMemory - usedMemory) + " KB");
}

static void MFT(ArrayList<Process> processes, ArrayList<MemoryPartition> partitions) {
    System.out.println("\nMFT (Multiprogramming with Fixed Tasks) Simulation:");
    for (Process p : processes) {
        boolean allocated = false;
        for (MemoryPartition partition : partitions) {
            if (partition.allocatedProcess == -1 && partition.size >= p.size) {
                partition.allocatedProcess = p.id;
                p.allocatedPartition = partition.size;
                System.out.println("Process " + p.id + " allocated to partition of size " + partition.size + " KB");
```

```
                allocated = true;
                break;
            }
        }
        if (!allocated) {
            System.out.println("Process " + p.id + " cannot be allocated due to insufficient partition size.");
        }
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter total memory size (for MVT): ");
    int totalMemory = sc.nextInt();

    System.out.print("Enter number of processes: ");
    int n = sc.nextInt();

    ArrayList<Process> processes = new ArrayList<>();
    System.out.println("Enter Process ID and Size:");
    for (int i = 0; i < n; i++) {
        int id = sc.nextInt();
        int size = sc.nextInt();
        processes.add(new Process(id, size));
```

```
        }

        System.out.print("Enter number of fixed-size partitions (for MFT): ");
        int m = sc.nextInt();
        ArrayList<MemoryPartition> partitions = new ArrayList<>();

        System.out.println("Enter Partition Sizes:");
        for (int i = 0; i < m; i++) {
            int size = sc.nextInt();
            partitions.add(new MemoryPartition(size));
        }

        MVT(processes, totalMemory);
        MFT(processes, partitions);

        sc.close();
    }
}
```

Output :-

```
Enter total memory size (for MVT): 1000
Enter number of processes: 3
Enter Process ID and Size:
1 200
2 400
3 500
```

Enter number of fixed-size partitions (for MFT): 3

Enter Partition Sizes:

300

400

600

MVT (Multiprogramming with Variable Tasks) Simulation:

Process 1 allocated 200 KB

Process 2 allocated 400 KB

Process 3 cannot be allocated due to insufficient memory.

Total Memory Used: 600 KB

Total Memory Free: 400 KB

MFT (Multiprogramming with Fixed Tasks) Simulation:

Process 1 allocated to partition of size 300 KB

Process 2 allocated to partition of size 400 KB

Process 3 allocated to partition of size 600 KB

4.Write a program for Bankers Algorithm for Dead Lock Avoidance

```
import java.util.Scanner;

public class BankersAlgorithm {
    int n, m; // n = number of processes, m = number of resources
    int[][] max, allocation, need;
    int[] available;

    public BankersAlgorithm(int n, int m) {
        this.n = n;
        this.m = m;
        max = new int[n][m];
        allocation = new int[n][m];
        need = new int[n][m];
        available = new int[m];
    }

    // Input function
    void inputData(Scanner sc) {
        System.out.println("Enter Max matrix:");
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                max[i][j] = sc.nextInt();

        System.out.println("Enter Allocation matrix:");
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            allocation[i][j] = sc.nextInt();

    System.out.println("Enter Available resources:");
    for (int j = 0; j < m; j++)
        available[j] = sc.nextInt();

    // Calculate Need matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}

// Safety Algorithm
boolean isSafe() {
    boolean[] finish = new boolean[n]; // Track completed processes
    int[] work = available.clone(); // Work = Available
    int[] safeSequence = new int[n];
    int count = 0;

    while (count < n) {
        boolean found = false;
        for (int i = 0; i < n; i++) {
            if (!finish[i]) { // If process is not finished
```

```
            boolean canAllocate = true;
            for (int j = 0; j < m; j++)
                if (need[i][j] > work[j]) {
                    canAllocate = false;
                    break;
                }

            if (canAllocate) { // Allocate resources
                for (int j = 0; j < m; j++)
                    work[j] += allocation[i][j];

                safeSequence[count++] = i;
                finish[i] = true;
                found = true;
            }
        }
    }
    if (!found) return false; // If no process can be allocated, return false
}

System.out.print("Safe Sequence: ");
for (int i = 0; i < n; i++)
    System.out.print("P" + safeSequence[i] + " ");
System.out.println("\nSystem is in a Safe State!");
return true;
```

```
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of processes: ");
        int n = sc.nextInt();
        System.out.print("Enter number of resources: ");
        int m = sc.nextInt();

        BankersAlgorithm ba = new BankersAlgorithm(n, m);
        ba.inputData(sc);

        if (!ba.isSafe())
            System.out.println("System is in an Unsafe State! Deadlock may occur.");

        sc.close();
    }
}
```

Output :-

```
Enter number of processes: 5
Enter number of resources: 3
Enter Max matrix:
7 5 3
3 2 2
9 0 2
```

2 2 2

4 3 3

Enter Allocation matrix:

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Enter Available resources:

3 3 2

Safe Sequence: P1 P3 P4 P0 P2

System is in a Safe State!

5.Implement Bankers Algorithm Dead Lock Prevention

```
import java.util.Scanner;

public class BankersAlgorithmPrevention {
    int n, m; // n = processes, m = resources
    int[][] max, allocation, need;
    int[] available;

    public BankersAlgorithmPrevention(int n, int m) {
        this.n = n;
        this.m = m;
        max = new int[n][m];
        allocation = new int[n][m];
        need = new int[n][m];
        available = new int[m];
    }

    // Input function
    void inputData(Scanner sc) {
        System.out.println("Enter Max matrix:");
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                max[i][j] = sc.nextInt();
```

```
    System.out.println("Enter Allocation matrix:");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            allocation[i][j] = sc.nextInt();

    System.out.println("Enter Available resources:");
    for (int j = 0; j < m; j++)
        available[j] = sc.nextInt();

    // Calculate Need matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}

// Deadlock Prevention by Requesting All at Once
boolean requestResources(int processID, int[] request) {
    System.out.println("\nProcess P" + processID + " requesting resources: ");
    for (int i = 0; i < m; i++)
        System.out.print(request[i] + " ");
    System.out.println();

    // Check if request exceeds Need
    for (int i = 0; i < m; i++) {
        if (request[i] > need[processID][i]) {
```

```
            System.out.println("Error: Process P" + processID + " requested more than its Need.");
            return false;
        }
    }

    // Check if request exceeds Available
    for (int i = 0; i < m; i++) {
        if (request[i] > available[i]) {
            System.out.println("Process P" + processID + " must wait. Resources unavailable.");
            return false;
        }
    }

    // Temporarily allocate requested resources
    for (int i = 0; i < m; i++) {
        available[i] -= request[i];
        allocation[processID][i] += request[i];
        need[processID][i] -= request[i];
    }

    // Check if system is still safe
    if (isSafe()) {
        System.out.println("Request granted.");
        return true;
    } else {
```

```
        // Rollback if unsafe
        for (int i = 0; i < m; i++) {
            available[i] += request[i];
            allocation[processID][i] -= request[i];
            need[processID][i] += request[i];
        }
        System.out.println("Request denied: System would be unsafe.");
        return false;
    }
}

// Safety Algorithm
boolean isSafe() {
    boolean[] finish = new boolean[n];
    int[] work = available.clone();
    int[] safeSequence = new int[n];
    int count = 0;

    while (count < n) {
        boolean found = false;
        for (int i = 0; i < n; i++) {
            if (!finish[i]) {
                boolean canAllocate = true;
                for (int j = 0; j < m; j++)
                    if (need[i][j] > work[j]) {
```

```
                    canAllocate = false;
                    break;
                }

            if (canAllocate) {
                for (int j = 0; j < m; j++)
                    work[j] += allocation[i][j];

                safeSequence[count++] = i;
                finish[i] = true;
                found = true;
            }
        }
    }
    if (!found) return false;
}

System.out.print("Safe Sequence: ");
for (int i = 0; i < n; i++)
    System.out.print("P" + safeSequence[i] + " ");
System.out.println("\nSystem remains in a Safe State!");
return true;
}

public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of processes: ");
        int n = sc.nextInt();
        System.out.print("Enter number of resources: ");
        int m = sc.nextInt();

        BankersAlgorithmPrevention ba = new BankersAlgorithmPrevention(n, m);
        ba.inputData(sc);

        System.out.print("Enter Process ID requesting resources: ");
        int processID = sc.nextInt();

        System.out.println("Enter request for each resource:");
        int[] request = new int[m];
        for (int i = 0; i < m; i++)
            request[i] = sc.nextInt();

        ba.requestResources(processID, request);

        sc.close();
    }
}
```

Output :-

Enter number of processes: 5
Enter number of resources: 3
Enter Max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Available resources:
3 3 2
Enter Process ID requesting resources: 1
Enter request for each resource:
1 0 2

Process P1 requesting resources:
1 0 2
Safe Sequence: P1 P3 P4 P0 P2
System remains in a Safe State!
Request granted.

Process P1 requesting resources:
1 0 2
Request denied: System would be unsafe.

6.Write a program to simulate Producer-Consumer problem

```
import java.util.LinkedList;

import java.util.Queue;

import java.util.concurrent.Semaphore;


class ProducerConsumer {

    private static final int BUFFER_SIZE = 5;

    private final Queue<Integer> buffer = new LinkedList<>();


    // Semaphores

    private final Semaphore mutex = new Semaphore(1);  // Controls access to buffer

    private final Semaphore empty = new Semaphore(BUFFER_SIZE); // Tracks empty slots

    private final Semaphore full = new Semaphore(0); // Tracks filled slots


    // Producer Thread

    class Producer extends Thread {

        public void run() {

            int item = 0;

            while (true) {

                try {

                    Thread.sleep(1000); // Simulate production time

                    empty.acquire();  // Wait for an empty slot

                    mutex.acquire();  // Lock the buffer


                    buffer.add(++item); // Produce an item
```

```
                System.out.println("Producer produced: " + item);

                mutex.release(); // Unlock the buffer
                full.release();  // Signal a filled slot
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// Consumer Thread
class Consumer extends Thread {
    public void run() {
        while (true) {
            try {
                Thread.sleep(2000); // Simulate consumption time
                full.acquire();  // Wait for an available item
                mutex.acquire(); // Lock the buffer

                int item = buffer.poll(); // Consume item
                System.out.println("Consumer consumed: " + item);

                mutex.release(); // Unlock the buffer
                empty.release(); // Signal an empty slot
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) {
    ProducerConsumer pc = new ProducerConsumer();
    Producer producer = pc.new Producer();
    Consumer consumer = pc.new Consumer();

    producer.start();
    consumer.start();
}
}
```

Output :-

```
Producer produced: 1
Consumer consumed: 1
Producer produced: 2
Producer produced: 3
Consumer consumed: 2
Producer produced: 4
Consumer consumed: 3
Producer produced: 5
Consumer consumed: 4
Producer produced: 6
Consumer consumed: 5
...
```

7.Simulate all Page replacement algorithms.
e) FIFO
f) LRU
g) LFU
h) Optimal

```
import java.util.*;


public class PageReplacementAlgorithms {


    // FIFO Page Replacement Algorithm

    static void fifo(int[] pages, int capacity) {

        Set<Integer> set = new LinkedHashSet<>(capacity);

        int pageFaults = 0;


        System.out.println("\nFIFO Page Replacement:");

        for (int page : pages) {

            if (!set.contains(page)) {

                if (set.size() == capacity) {

                    Iterator<Integer> itr = set.iterator();

                    int first = itr.next();

                    set.remove(first);

                }

                set.add(page);

                pageFaults++;

                System.out.println("Page " + page + " added -> " + set);

            }
```

```java
        }
        System.out.println("Total Page Faults: " + pageFaults);
    }

    // LRU Page Replacement Algorithm
    static void lru(int[] pages, int capacity) {
        Set<Integer> set = new LinkedHashSet<>(capacity);
        Map<Integer, Integer> lruMap = new HashMap<>();
        int pageFaults = 0, time = 0;

        System.out.println("\nLRU Page Replacement:");
        for (int page : pages) {
            if (!set.contains(page)) {
                if (set.size() == capacity) {
                    int lru = Collections.min(lruMap.entrySet(),
Comparator.comparingInt(Map.Entry::getValue)).getKey();
                    set.remove(lru);
                    lruMap.remove(lru);
                }
                set.add(page);
                pageFaults++;
            }
            lruMap.put(page, time++);
            System.out.println("Page " + page + " accessed -> " + set);
        }
        System.out.println("Total Page Faults: " + pageFaults);
```

```
    }

    // LFU Page Replacement Algorithm
    static void lfu(int[] pages, int capacity) {
        Map<Integer, Integer> freqMap = new HashMap<>();
        Set<Integer> set = new LinkedHashSet<>(capacity);
        int pageFaults = 0;

        System.out.println("\nLFU Page Replacement:");
        for (int page : pages) {
            if (!set.contains(page)) {
                if (set.size() == capacity) {
                    int lfu = Collections.min(freqMap.entrySet(),
Comparator.comparingInt(Map.Entry::getValue)).getKey();
                    set.remove(lfu);
                    freqMap.remove(lfu);
                }
                set.add(page);
                pageFaults++;
            }
            freqMap.put(page, freqMap.getOrDefault(page, 0) + 1);
            System.out.println("Page " + page + " accessed -> " + set);
        }
        System.out.println("Total Page Faults: " + pageFaults);
    }
```

```
// Optimal Page Replacement Algorithm
static void optimal(int[] pages, int capacity) {
    Set<Integer> set = new LinkedHashSet<>(capacity);
    int pageFaults = 0;

    System.out.println("\nOptimal Page Replacement:");
    for (int i = 0; i < pages.length; i++) {
        int page = pages[i];

        if (!set.contains(page)) {
            if (set.size() == capacity) {
                int farthest = -1, replace = -1;
                for (int p : set) {
                    int nextUse = Integer.MAX_VALUE;
                    for (int j = i + 1; j < pages.length; j++) {
                        if (pages[j] == p) {
                            nextUse = j;
                            break;
                        }
                    }
                    if (nextUse > farthest) {
                        farthest = nextUse;
                        replace = p;
                    }
                }
```

```
                set.remove(replace);
            }
            set.add(page);
            pageFaults++;
            System.out.println("Page " + page + " added -> " + set);
        }
    }
    System.out.println("Total Page Faults: " + pageFaults);
  }

  public static void main(String[] args) {
    int[] pages = {3, 2, 1, 5, 4, 2, 3, 6, 1, 3};
    int capacity = 3;

    fifo(pages, capacity);
    lru(pages, capacity);
    lfu(pages, capacity);
    optimal(pages, capacity);
  }
}
```

Output :-

```
FIFO Page Replacement:
Page 3 added -> [3]
Page 2 added -> [3, 2]
```

Page 1 added -> [3, 2, 1]

Page 5 added -> [2, 1, 5]

Page 4 added -> [1, 5, 4]

Page 2 added -> [5, 4, 2]

Page 3 added -> [4, 2, 3]

Page 6 added -> [2, 3, 6]

Page 1 added -> [3, 6, 1]

Page 3 added -> [6, 1, 3]

Total Page Faults: 9

LRU Page Replacement:

Page 3 accessed -> [3]

Page 2 accessed -> [3, 2]

Page 1 accessed -> [3, 2, 1]

Page 5 accessed -> [2, 1, 5]

Page 4 accessed -> [1, 5, 4]

Page 2 accessed -> [5, 4, 2]

Page 3 accessed -> [4, 2, 3]

Page 6 accessed -> [2, 3, 6]

Page 1 accessed -> [3, 6, 1]

Page 3 accessed -> [6, 1, 3]

Total Page Faults: 9

LFU Page Replacement:

Page 3 accessed -> [3]

Page 2 accessed -> [3, 2]

Page 1 accessed -> [3, 2, 1]

Page 5 accessed -> [2, 1, 5]

Page 4 accessed -> [1, 5, 4]

Page 2 accessed -> [5, 4, 2]

Page 3 accessed -> [4, 2, 3]

Page 6 accessed -> [2, 3, 6]

Page 1 accessed -> [3, 6, 1]

Page 3 accessed -> [6, 1, 3]

Total Page Faults: 9

Optimal Page Replacement:

Page 3 added -> [3]

Page 2 added -> [3, 2]

Page 1 added -> [3, 2, 1]

Page 5 added -> [2, 1, 5]

Page 4 added -> [1, 5, 4]

Page 2 added -> [5, 4, 2]

Page 3 added -> [4, 2, 3]

Page 6 added -> [2, 3, 6]

Page 1 added -> [3, 6, 1]

Page 3 added -> [6, 1, 3]

Total Page Faults: 8

8.Simulate Paging Techniques of memory management

```
import java.util.Scanner;

import java.util.HashMap;


class PagingSimulation {

    private static final int PAGE_SIZE = 4; // Each page has 4 units (example)

    private static final int FRAME_COUNT = 5; // Total available frames in memory


    private int[] pageTable; // Stores frame mappings

    private boolean[] frames; // Tracks occupied frames


    // Constructor: Initializes memory

    public PagingSimulation(int pageCount) {

        pageTable = new int[pageCount];

        frames = new boolean[FRAME_COUNT]; // Initially all frames are free


        for (int i = 0; i < pageCount; i++) {

            pageTable[i] = -1; // -1 means page is not allocated

        }

    }


    // Function to allocate pages to frames

    public void allocatePages() {

        Scanner scanner = new Scanner(System.in);

        for (int i = 0; i < pageTable.length; i++) {
```

```
            System.out.print("Enter frame number for Page " + i + " (Available: 0 to " + (FRAME_COUNT - 1) +
"): ");
            int frame = scanner.nextInt();

            if (frame >= 0 && frame < FRAME_COUNT && !frames[frame]) {
                pageTable[i] = frame; // Map page to frame
                frames[frame] = true; // Mark frame as occupied
            } else {
                System.out.println("Invalid or occupied frame! Try again.");
                i--; // Retry for the same page
            }
        }
    }

    // Function to translate Logical Address to Physical Address
    public void translateAddress(int logicalAddress) {
        int pageNumber = logicalAddress / PAGE_SIZE;
        int offset = logicalAddress % PAGE_SIZE;

        if (pageNumber >= pageTable.length || pageTable[pageNumber] == -1) {
            System.out.println("Invalid address! Page not allocated.");
            return;
        }

        int frameNumber = pageTable[pageNumber];
        int physicalAddress = frameNumber * PAGE_SIZE + offset;
```

```
        System.out.println("Logical Address (" + logicalAddress + ") -> Physical Address (" + physicalAddress
+ ")");
    }

    // Display Page Table
    public void displayPageTable() {
        System.out.println("\nPage Table:");
        System.out.println("Page No. | Frame No.");
        for (int i = 0; i < pageTable.length; i++) {
            System.out.println("   " + i + "     |    " + (pageTable[i] == -1 ? "Not Allocated" : pageTable[i]));
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Number of pages in process
        System.out.print("Enter number of pages in process: ");
        int pages = scanner.nextInt();

        PagingSimulation paging = new PagingSimulation(pages);

        // Allocate pages
        paging.allocatePages();

        // Display page table
```

```java
        paging.displayPageTable();


        // Translate logical addresses
        System.out.print("\nEnter logical address to translate (or -1 to exit): ");
        int logicalAddress;
        while ((logicalAddress = scanner.nextInt()) != -1) {
            paging.translateAddress(logicalAddress);
            System.out.print("\nEnter next logical address (or -1 to exit): ");
        }


        System.out.println("\nSimulation Ended.");
        scanner.close();
    }
}
```

Output :-

```
Enter number of pages in process: 3
Enter frame number for Page 0 (Available: 0 to 4): 1
Enter frame number for Page 1 (Available: 0 to 4): 3
Enter frame number for Page 2 (Available: 0 to 4): 0

Page Table:
Page No. | Frame No.
   0     |    1
   1     |    3
   2     |    0

Enter logical address to translate (or -1 to exit): 5
Logical Address (5) -> Physical Address (9)

Enter next logical address (or -1 to exit): 9
Logical Address (9) -> Physical Address (13)

Enter next logical address (or -1 to exit): -1
Simulation Ended.
```

9.Simulate the following disk scheduling algorithms

a) FCFS

b) SSTF

c) SCAN

d) CSCAN

```
import java.util.Arrays;

import java.util.Scanner;


class DiskScheduling {


    // First-Come, First-Served (FCFS)

    static int fcfs(int[] requests, int head) {

        int seek = 0, current = head;

        System.out.println("FCFS Order: ");

        for (int req : requests) {

            System.out.print(req + " ");

            seek += Math.abs(req - current);

            current = req;

        }

        System.out.println("\nTotal Seek Time: " + seek);

        return seek;

    }


    // Shortest Seek Time First (SSTF)

    static int sstf(int[] requests, int head) {

        int seek = 0, current = head;
```

```
    boolean[] visited = new boolean[requests.length];
    System.out.println("SSTF Order: ");
    for (int i = 0; i < requests.length; i++) {
        int minIndex = -1, minDistance = Integer.MAX_VALUE;
        for (int j = 0; j < requests.length; j++) {
            if (!visited[j] && Math.abs(requests[j] - current) < minDistance) {
                minDistance = Math.abs(requests[j] - current);
                minIndex = j;
            }
        }
        visited[minIndex] = true;
        System.out.print(requests[minIndex] + " ");
        seek += minDistance;
        current = requests[minIndex];
    }
    System.out.println("\nTotal Seek Time: " + seek);
    return seek;
}

// SCAN (Elevator Algorithm)
static int scan(int[] requests, int head, int diskSize) {
    Arrays.sort(requests);
    int seek = 0, current = head;
    System.out.println("SCAN Order: ");
    for (int req : requests) {
```

```
        if (req >= head) {
            System.out.print(req + " ");
            seek += Math.abs(req - current);
            current = req;
        }
    }
    seek += Math.abs(diskSize - 1 - current);
    System.out.print(diskSize - 1 + " ");
    current = diskSize - 1;
    for (int i = requests.length - 1; i >= 0; i--) {
        if (requests[i] < head) {
            System.out.print(requests[i] + " ");
            seek += Math.abs(requests[i] - current);
            current = requests[i];
        }
    }
    System.out.println("\nTotal Seek Time: " + seek);
    return seek;
}

// C-SCAN (Circular SCAN)
static int cscan(int[] requests, int head, int diskSize) {
    Arrays.sort(requests);
    int seek = 0, current = head;
    System.out.println("C-SCAN Order: ");
```

```
    for (int req : requests) {
        if (req >= head) {
            System.out.print(req + " ");
            seek += Math.abs(req - current);
            current = req;
        }
    }
    seek += Math.abs(diskSize - 1 - current);
    System.out.print(diskSize - 1 + " ");
    seek += diskSize - 1;
    System.out.print("0 ");
    current = 0;
    for (int req : requests) {
        if (req < head) {
            System.out.print(req + " ");
            seek += Math.abs(req - current);
            current = req;
        }
    }
    System.out.println("\nTotal Seek Time: " + seek);
    return seek;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
```

```
// Input disk requests
System.out.print("Enter number of requests: ");
int n = scanner.nextInt();
int[] requests = new int[n];

System.out.print("Enter request sequence: ");
for (int i = 0; i < n; i++) {
    requests[i] = scanner.nextInt();
}

// Input initial head position
System.out.print("Enter initial head position: ");
int head = scanner.nextInt();

// Input disk size (for SCAN & C-SCAN)
System.out.print("Enter disk size: ");
int diskSize = scanner.nextInt();

System.out.println("\n--- Disk Scheduling Results ---");
fcfs(requests, head);
System.out.println();
sstf(requests, head);
System.out.println();
scan(requests, head, diskSize);
```

```java
        System.out.println();

        cscan(requests, head, diskSize);



        scanner.close();

    }

}
```

Output :-

```
Enter number of requests: 5
Enter request sequence: 82 170 43 140 24
Enter initial head position: 50
Enter disk size: 200

--- Disk Scheduling Results ---

FCFS Order:
82 170 43 140 24
Total Seek Time: 302

SSTF Order:
43 24 82 140 170
Total Seek Time: 158

SCAN Order:
82 140 170 199 43 24
Total Seek Time: 295

C-SCAN Order:
82 140 170 199 0 24 43
Total Seek Time: 356
```

VIVA QUESTIONS

Here's a **set of important viva questions and answers** for **Unit-I (Introduction to Operating Systems)** to help you prepare effectively. 🚀

Viva Questions and Answers - UNIT I (Introduction to OS)

1. What is an Operating System (OS)?

Answer:
An **Operating System (OS)** is **system software** that acts as an **interface between hardware and users**. It manages hardware resources and provides essential services for applications.

2. What are the main functions of an OS?

Answer:
The primary functions of an OS include:

1. **Process Management** – Handles process creation, scheduling, and termination.
2. **Memory Management** – Allocates and manages memory space.
3. **File System Management** – Organizes, stores, and retrieves files.
4. **Device Management** – Controls hardware components like disks and printers.
5. **Security & Access Control** – Ensures data security and user authentication.
6. **I/O System Management** – Manages input/output devices efficiently.

3. What are the different types of Operating Systems?

Answer:

1. **Batch Operating System** – Jobs are processed in batches without user interaction.
2. **Time-Sharing OS** – Multiple users share system resources using CPU scheduling.
3. **Multiprogramming OS** – Runs multiple programs simultaneously to maximize CPU utilization.
4. **Distributed OS** – Manages multiple computers working as a single system.
5. **Real-Time OS (RTOS)** – Provides immediate response to time-critical tasks.
6. **Embedded OS** – Designed for specialized embedded devices (e.g., ATM, washing machines).
7. **Network OS** – Supports networking capabilities for multiple devices.

4. What is the difference between Multiprogramming and Multitasking?

Answer:

Feature	Multiprogramming	Multitasking

Feature	Multiprogramming	Multitasking
Definition	Multiple processes in memory at the same time	Rapid switching between processes
Execution	CPU switches between programs	User interacts with multiple tasks simultaneously
User Interaction	No user interaction	Allows user interaction
Example	Batch OS	Windows, Linux

5. Explain the history and evolution of Operating Systems.

Answer:

- **1st Generation (1940s-1950s)** – No OS, manual programming using punched cards.
- **2nd Generation (1950s-1960s)** – Batch processing systems developed.
- **3rd Generation (1960s-1970s)** – Multiprogramming and time-sharing OS introduced.
- **4th Generation (1980s-Present)** – GUI-based OS, real-time OS, and mobile OS.
- **5th Generation (Future)** – AI-powered OS, cloud-based computing.

6. What is a Batch Operating System?

Answer:
A **Batch OS** processes jobs in a queue without user intervention. Jobs are **collected, grouped, and executed sequentially**. Example: **IBM OS/360**.

7. What is a Time-Sharing Operating System?

Answer:
A **Time-Sharing OS** allows multiple users to interact with the system simultaneously. CPU time is divided into **time slices**, and each user gets a share. Example: **UNIX, Windows Server**.

8. What is the difference between a Time-Sharing and Real-Time OS?

Answer:

Feature	Time-Sharing OS	Real-Time OS
Response Time	Delayed	Immediate
Priority	Equal for all processes	Critical tasks get higher priority

Feature	Time-Sharing OS	Real-Time OS
Usage	General-purpose	Industrial, medical, defense
Example	UNIX, Linux	VxWorks, QNX

9. What is a Real-Time Operating System (RTOS)?

Answer:
A **Real-Time OS** is designed to handle tasks **within a fixed time constraint**. It is used in **critical applications** like medical devices and automotive systems.

Types of RTOS:

- **Hard RTOS** – No delay tolerated (e.g., pacemakers, aircraft control).
- **Soft RTOS** – Small delays are acceptable (e.g., online gaming, video streaming).

10. What are Process Control and Embedded Operating Systems?

Answer:

- **Process Control OS**: Used in industries to monitor and control machinery (e.g., SCADA systems).
- **Embedded OS**: Designed for specialized devices like **ATMs, Smart TVs, IoT devices**.

11. What are the different Operating Systems used in various devices?

Answer:

- **Personal Computers** – Windows, Linux, macOS
- **Workstations** – UNIX, Solaris
- **Handheld Devices** – Android, iOS
- **Embedded Systems** – RTOS, VxWorks

12. What is the main purpose of a Kernel in an OS?

Answer:
The **Kernel** is the **core component** of an OS that directly interacts with hardware. It manages CPU, memory, and I/O operations.

Types of Kernels:

1. **Monolithic Kernel** – Large single program (e.g., Linux).
2. **Microkernel** – Small core with minimal services (e.g., QNX).

13. What are System Calls in an OS?

Answer:
System Calls are **interface functions** used by applications to request services from the OS (e.g., file handling, process control).

14. What is the difference between Process and Thread?

Answer:

Feature	Process	Thread
Definition	Independent executing instance	Lightweight sub-task of a process
Memory	Separate memory space	Shares process memory
Speed	Slower	Faster
Example	MS Word	Web browser tabs

15. What are the advantages of a Multiprogramming OS?

Answer:

1. **Increased CPU Utilization** – Reduces idle CPU time.
2. **Faster Execution** – Multiple programs execute in parallel.
3. **Efficient Memory Usage** – Optimized memory allocation.
4. **Better System Throughput** – More jobs completed per unit time.

16. What are the key features of a Distributed Operating System?

Answer:
A **Distributed OS** runs on multiple machines and provides a unified computing experience.
Features:

- **Resource Sharing** – Uses networked resources efficiently.
- **Fault Tolerance** – Continues operation even if one system fails.
- **Scalability** – Can add more systems easily.
- **Examples**: Amoeba, Mach, Windows Server.

Viva Questions and Answers – UNIT-II (Processor & Process Management)

This unit covers **Processor and User Modes, Kernels, System Calls, Process Management, Threads, and Scheduling Algorithms**. Below are **important viva questions and answers** to help you prepare. 🚀

---

1. What are Processor and User Modes?

Answer:
The **Processor operates in two modes**:

1. **User Mode** – Executes user programs with limited access to hardware.
2. **Kernel Mode** – Executes OS-level operations with full access to system resources.

Mode switching occurs when a program requests privileged operations via **system calls**.

---

2. What is a Kernel? What are its types?

Answer:
A **Kernel** is the **core of an OS** that manages CPU, memory, and device operations.

Types of Kernels:

1. **Monolithic Kernel** – All OS services in one layer (e.g., Linux, UNIX).
2. **Microkernel** – Minimal kernel with external services (e.g., QNX, Minix).
3. **Hybrid Kernel** – Combination of monolithic & microkernel (e.g., Windows, macOS).
4. **Exokernel** – Allows direct hardware access (e.g., MIT Exokernel).

---

3. What are System Calls?

Answer:
System Calls are functions used by programs to request services from the OS (e.g., file handling, process management).

Types of System Calls:

1. **Process Control** – `fork()`, `exit()`, `exec()`
2. **File Management** – `open()`, `read()`, `write()`, `close()`
3. **Device Management** – `ioctl()`, `read()`, `write()`
4. **Information Maintenance** – `getpid()`, `gettimeofday()`

5. **Communication** – `pipe()`, `shmget()`, `msgsend()`

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Process ID: %d\n", getpid()); // Using system call getpid()
    return 0;
}
```

---

4. What is a System Program?

Answer:
A **System Program** provides an interface between user applications and the OS. Examples include:

- **Compilers** (e.g., GCC, Java Compiler)
- **Interpreters** (e.g., Python, Bash)
- **File Management Tools** (e.g., `ls`, `cp`, `rm`)
- **Debuggers** (e.g., GDB)

---

5. What is a Process? How is it different from a Program?

Answer:
A **Process** is a running instance of a program.

Feature	Program	Process
Definition	A set of instructions	A program in execution
State	Passive	Active
Memory	Stored on disk	Uses CPU & memory
Example	Notepad.exe file	Running Notepad

---

6. Explain the Process Lifecycle.

Answer:
A process goes through **five states**:

1. **New** – Process is created.
2. **Ready** – Waiting to be assigned CPU.
3. **Running** – Process is executing.
4. **Waiting (Blocked)** – Process waits for an event (I/O).
5. **Terminated** – Process finishes execution.

---

7. What is Process Hierarchy?

Answer:
Processes in an OS follow a **parent-child relationship**, called the **Process Hierarchy**.

- **Parent Process** creates a **Child Process** using `fork()`.
- Child processes can create more processes, forming a **tree structure**.

Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int pid = fork(); // Creates child process
    if (pid == 0)
        printf("Child Process\n");
    else
        printf("Parent Process\n");
    return 0;
}
```

---

8. What are Threads? What are the types?

Answer:
A **Thread** is the smallest unit of execution within a process.

Types of Threads:

1. **User-Level Threads (ULT)** – Managed by user libraries (e.g., Pthreads).
2. **Kernel-Level Threads (KLT)** – Managed by OS.

---

9. What are Threading Issues?

Answer:
Thread management faces **synchronization and execution** challenges:

- **Race Conditions** – Multiple threads access shared data incorrectly.
- **Deadlocks** – Two or more threads wait indefinitely for each other.

- **Starvation** – Low-priority threads may never execute.

---

10. What are Thread Libraries?

Answer:
Thread libraries provide functions for thread management. Examples:

- **Pthreads (POSIX Threads)** – Linux, UNIX
- **Windows Threads** – Windows OS
- **Java Threads** – Java applications

Example (POSIX Thread Creation in C):

```
#include <pthread.h>
#include <stdio.h>
void *threadFunction() {
    printf("Hello from thread!\n");
}
int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, threadFunction, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

---

11. What is CPU Scheduling? Why is it needed?

Answer:
CPU scheduling selects **which process gets CPU time** to improve efficiency.
Goals:

- **Max CPU utilization**
- **Minimize waiting time**
- **Ensure fairness**

---

12. What are the different CPU Scheduling Algorithms?

Non-Preemptive Scheduling

1. **First Come, First Serve (FCFS)** – Executes processes in order of arrival.
2. **Shortest Job Next (SJN)** – Executes shortest process first.
3. **Priority Scheduling** – Executes highest priority process first.

Preemptive Scheduling

1. **Round Robin (RR)** – Each process gets a fixed time slice (time quantum).
2. **Shortest Remaining Time First (SRTF)** – Runs process with the shortest remaining time.
3. **Multilevel Queue Scheduling** – Processes divided into queues based on priority.

---

13. What is the difference between Preemptive and Non-Preemptive Scheduling?

Feature	Preemptive Scheduling	Non-Preemptive Scheduling
Definition	Process can be interrupted	Process runs until completion
CPU Utilization	Higher	Lower
Example	Round Robin, SRTF	FCFS, SJF

---

14. What is Context Switching?

Answer:
Context switching is the process of **saving the state of one process and loading another**. It occurs during:

- **Process switching**
- **Interrupt handling**
- **Multitasking**

---

15. What are the criteria for CPU Scheduling?

Answer:

1. **CPU Utilization** – Maximize CPU usage.
2. **Throughput** – Number of processes completed per second.
3. **Turnaround Time** – Time taken from submission to completion.
4. **Waiting Time** – Time spent in the ready queue.
5. **Response Time** – Time from request submission to first response.

Viva Questions and Answers – UNIT-III (Process Management & Synchronization)

This unit covers **Deadlocks, Process Synchronization, Critical Section, Semaphores, IPC, and Classical Problems like Producer-Consumer & Reader-Writer.** Below are **important viva questions and answers** to help you prepare. 🚀

Deadlock

1. What is a Deadlock?

Answer:
A **deadlock** occurs when a set of processes are waiting for resources that are held by other processes, creating a circular wait and preventing further execution.

---

2. What are the necessary conditions for Deadlock?

Answer:
A deadlock occurs if the following **four conditions hold simultaneously**:

1. **Mutual Exclusion** – A resource can only be held by one process at a time.
2. **Hold and Wait** – A process holding a resource is waiting for additional resources held by other processes.
3. **No Preemption** – A resource cannot be forcibly taken from a process.
4. **Circular Wait** – A set of processes are waiting in a circular chain for each other's resources.

---

3. What are the approaches to handling Deadlocks?

Answer:

1. **Deadlock Prevention** – Ensuring that at least one of the four necessary conditions never occurs.
2. **Deadlock Avoidance** – Using algorithms (like **Banker's Algorithm**) to ensure safe resource allocation.
3. **Deadlock Detection & Recovery** – Detecting deadlocks using **wait-for graphs** and recovering by preempting resources or terminating processes.

---

4. How does the Banker's Algorithm prevent Deadlock?

Answer:
The **Banker's Algorithm** checks if allocating resources will lead to a safe state by:

1. Checking the **maximum resource need** of each process.
2. Ensuring resources are available **before granting a request**.
3. **Simulating allocation** to verify no deadlock occurs.

---

5. What is a Wait-for Graph?

Answer:
A **Wait-for Graph (WFG)** is used for deadlock detection. It is a directed graph where:

- **Nodes represent processes**.
- **Edges (P1 → P2) indicate P1 is waiting for a resource held by P2**.
  A cycle in the graph indicates **deadlock**.

Process Synchronization

6. What is Process Synchronization? Why is it needed?

Answer:
Process synchronization ensures that **multiple processes access shared resources without conflicts**, preventing **race conditions** and **inconsistencies**.

---

7. What is the Critical Section Problem?

Answer:
A **Critical Section** is a part of a program where shared resources are accessed. The problem is ensuring that **only one process enters the critical section at a time**.

Solution requirements (Critical Section Problem Criteria):

1. **Mutual Exclusion** – Only one process is in the critical section at a time.
2. **Progress** – If no process is in the critical section, one must be allowed in.
3. **Bounded Waiting** – A process must not wait indefinitely.

---

8. What are the methods for solving the Critical Section Problem?

Answer:

1. **Software-based solutions** – Peterson's Algorithm, Dekker's Algorithm.
2. **Hardware-based solutions** – Disabling interrupts, Test-and-Set Lock.
3. **Synchronization tools** – Semaphores, Mutex, Monitors.

Semaphores

9. What is a Semaphore? What are its types?

Answer:
A **semaphore** is a synchronization mechanism used to control access to shared resources.

Types:

1. **Binary Semaphore (Mutex)** – Values {0,1}, used for mutual exclusion.
2. **Counting Semaphore** – Values > 1, used for resource management.

Operations:

- **Wait (P operation)**: `wait(S) { while (S <= 0) wait; S--; }`
- **Signal (V operation)**: `signal(S) { S++; }`

---

10. What is a Mutex? How is it different from a Semaphore?

Answer:
A **Mutex (Mutual Exclusion Lock)** is a **binary semaphore** used to protect shared resources.

Feature	Semaphore	Mutex
Value	Can be >1 (Counting Semaphore)	Only 0 or 1
Ownership	Can be signaled by any process	Only released by the owner
Use Case	Resource management	Mutual exclusion

---

11. What are the methods of Inter-Process Communication (IPC)?

Answer:

1. **Shared Memory** – Fast but requires synchronization (e.g., using semaphores).
2. **Message Passing** – Processes communicate via messages (e.g., `pipe()`, `msgsend()`).
3. **Pipes** – Unidirectional communication between processes.
4. **Sockets** – Used for inter-machine communication over a network.

Classical Process Synchronization Problems

12. Explain the Producer-Consumer Problem.

Answer:

- The **Producer** creates items and puts them in a buffer.
- The **Consumer** takes items from the buffer.
- Synchronization ensures **no overflow or underflow** of the buffer.

Solution using Semaphores:

```
Semaphore full = 0, empty = N, mutex = 1;
Producer() {
    wait(empty);
    wait(mutex);
    // Produce item and add to buffer
    signal(mutex);
    signal(full);
}
Consumer() {
    wait(full);
    wait(mutex);
    // Consume item from buffer
    signal(mutex);
    signal(empty);
}
```

---

13. What is the Reader-Writer Problem?

Answer:

- **Multiple Readers can read simultaneously**.
- **Writers need exclusive access**.
- Ensures **no writer-starvation**.

Solution using Semaphores:

```
Semaphore mutex = 1, write = 1;
int readcount = 0;

Reader() {
    wait(mutex);
    readcount++;
    if (readcount == 1) wait(write);
    signal(mutex);

    // Read operation

    wait(mutex);
    readcount--;
```

```
    if (readcount == 0) signal(write);
    signal(mutex);
}

Writer() {
    wait(write);
    // Write operation
    signal(write);
}
```

---

14. What is Starvation? How is it different from Deadlock?

Answer:

- **Starvation** occurs when a **low-priority process never gets CPU time** due to high-priority processes.
- **Deadlock** occurs when processes wait indefinitely due to **circular resource dependency**.

Solution to Starvation:

- **Aging** – Gradually increasing the priority of waiting processes.

---

15. What is the Dining Philosophers Problem?

Answer:

- **Five philosophers sit at a circular table with five chopsticks.**
- They must **pick up two adjacent chopsticks to eat**.
- Ensures **no deadlock and no starvation**.

Solution using Semaphores:

```
Semaphore chopstick[5];

Philosopher(i) {
    wait(chopstick[i]);      // Pick left chopstick
    wait(chopstick[(i+1)%5]); // Pick right chopstick
    // Eat
    signal(chopstick[i]);    // Put left chopstick
    signal(chopstick[(i+1)%5]); // Put right chopstick
}
```

Viva Questions and Answers – UNIT-IV (Memory Management)

This unit covers **Physical & Virtual Memory, Memory Allocation Strategies, Paging, Segmentation, and Virtual Memory.** Below are **important viva questions and answers** to help you prepare. 🚀

---

1. What is Memory Management?

Answer:
Memory Management is a function of the Operating System that handles:

- **Allocation & deallocation** of memory to processes.
- **Tracking free and occupied memory**.
- **Ensuring efficient memory usage**.

Physical and Virtual Address Space

2. What is the difference between Physical and Virtual Address?

Answer:

Aspect	Physical Address	Virtual Address
Definition	Actual address in RAM	Logical address generated by CPU
Accessed by	Memory unit	CPU
Translation	Not required	Requires **MMU (Memory Management Unit)**
Visibility	Hardware-visible	Software-visible

---

3. What is Memory Management Unit (MMU)?

Answer:
The **MMU (Memory Management Unit)** is a hardware component that translates **virtual addresses** into **physical addresses** dynamically.

Steps:

1. CPU generates a virtual address.

2. MMU translates it using page tables.
3. The translated **physical address** is used to fetch data from RAM.

Memory Allocation Strategies

4. What are the different types of Memory Allocation?

Answer:

1. **Contiguous Memory Allocation**
   - Divides memory into **fixed or variable partitions**.
   - Processes are allocated **contiguous** memory blocks.
2. **Non-Contiguous Memory Allocation**
   - Uses **paging and segmentation**.
   - A process can be loaded into **different locations** in memory.

---

5. What is the difference between Fixed and Variable Partitioning?

Answer:

Aspect	Fixed Partitioning	Variable Partitioning
Definition	Memory is divided into fixed-size blocks	Memory is divided dynamically
Fragmentation	Leads to **internal fragmentation**	Leads to **external fragmentation**
Flexibility	Less flexible	More flexible
Utilization	Poor (small processes waste space)	Better (but can cause fragmentation)

---

6. What is Fragmentation? What are its types?

Answer:
Fragmentation occurs when memory is wasted due to inefficient allocation.

Types:

1. **Internal Fragmentation** – Occurs in **Fixed Partitioning** when allocated memory is larger than needed.
2. **External Fragmentation** – Occurs in **Variable Partitioning** when free memory is available but **not in contiguous form**.

Solution:

- **Compaction** – Rearranging memory blocks to create larger contiguous spaces.
- **Paging and Segmentation** – Eliminates the need for contiguous allocation.

Paging

7. What is Paging? Why is it used?

Answer:
Paging is a memory management scheme that eliminates **external fragmentation** by dividing:

- **Physical memory into fixed-size blocks (Frames)**.
- **Process memory into same-size blocks (Pages)**.

A **Page Table** maps **pages** to **frames**, allowing **non-contiguous allocation**.

---

8. What is the role of a Page Table?

Answer:
A **Page Table** stores the mapping of **virtual pages** to **physical frames**.

Each entry contains:

1. **Page Number** – Virtual memory identifier.
2. **Frame Number** – Corresponding physical memory location.

---

9. What is a TLB (Translation Lookaside Buffer)?

Answer:
A **TLB (Translation Lookaside Buffer)** is a **cache** that stores frequently used page table entries to speed up address translation.

Advantages:
✅ Reduces memory access time.
✅ Improves performance.

Segmentation

10. What is Segmentation? How is it different from Paging?

Answer:
Segmentation divides memory based on **logical divisions**, not fixed sizes.

Feature	Paging	Segmentation
Division	Fixed-size pages	Variable-size segments
Fragmentation	No **external fragmentation**	**External fragmentation** occurs
Table Used	Page Table	Segment Table
Usage	OS-based	User-based (e.g., code, data, stack)

---

11. What is the Segment Table?

Answer:
A **Segment Table** maps segment numbers to **physical memory locations**. Each entry contains:

1. **Base Address** – Start location of the segment.
2. **Limit** – Size of the segment.

Logical Address Format:
`<Segment Number, Offset>`

Virtual Memory

12. What is Virtual Memory?

Answer:
Virtual Memory is a technique that allows processes to use **more memory than physically available** by **swapping** data between RAM and disk.

---

13. What are the advantages of Virtual Memory?

Answer:
✅ Supports **large programs**.

✅ Reduces **need for contiguous allocation**.
✅ Enables **multiprogramming**.

---

14. What is Demand Paging?

Answer:
In **Demand Paging**, pages are loaded into memory **only when needed**, reducing memory waste.

Steps:

1. Process requests a page.
2. If not in memory, **Page Fault** occurs.
3. OS loads the required page from disk (swap space).

---

15. What is Thrashing? How to prevent it?

Answer:
Thrashing occurs when excessive page swapping slows down a system.

Viva Questions and Answers – UNIT-V (File & I/O Management, OS Security, and Disk Scheduling)

This unit covers **File Management, I/O Management, OS Security, Device Management, and Disk Scheduling Algorithms.** Below are **important viva questions and answers** to help you prepare. 🚀

File and I/O Management

1. What is a file in an Operating System?

Answer:
A **file** is a collection of related information stored on a disk with a specific name and format. It represents **data, programs, or multimedia**.

---

2. What are the different types of files?

Answer:

1. **Text files** – Contain readable characters (e.g., `.txt`, `.c`).
2. **Binary files** – Store data in binary format (e.g., `.exe`, `.jpg`).
3. **System files** – Used by the OS (e.g., `.dll`, `.sys`).
4. **Special files** – Devices or pipes.

---

3. What are the different File Operations?

Answer:

1. **Create** – Make a new file.
2. **Open** – Load a file for use.
3. **Read** – Retrieve data from a file.
4. **Write** – Add/modify data.
5. **Close** – End file access.
6. **Delete** – Remove file from storage.

---

4. What is a Directory? What are its types?

Answer:
A **directory** is a file system structure that stores references to files.

Types of directory structures:

1. **Single-level directory** – All files in a single directory.
2. **Two-level directory** – Each user gets a separate directory.
3. **Hierarchical (Tree) directory** – Allows subdirectories for organization.
4. **Acyclic Graph directory** – Supports shared files.
5. **General Graph directory** – Allows cyclic links but is complex.

---

5. What are File Allocation Methods?

Answer:
File allocation defines how disk blocks are assigned to files.

Method	Description	Pros	Cons
Contiguous	Files occupy continuous blocks.	Fast access	Causes **external**

Method	Description	Pros	Cons
Allocation			**fragmentation**
Linked Allocation	Each file is a linked list of blocks.	No fragmentation	Slow access (sequential only)
Indexed Allocation	Index table stores all block addresses.	Random access possible	Extra overhead for index

---

OS Security

6. What are the different types of OS security?

Answer:

1. **File Security** – Prevents unauthorized access/modification.
2. **User Authentication** – Uses passwords or biometrics.
3. **Network Security** – Protects from external threats (e.g., firewalls).
4. **Process Security** – Prevents process interference.

---

7. What are Access Control Mechanisms?

Answer:
Access Control restricts **who can access a file** and **what operations** they can perform.

Common mechanisms:

- **Discretionary Access Control (DAC)** – User-based permissions.
- **Mandatory Access Control (MAC)** – Enforced by the OS.
- **Role-Based Access Control (RBAC)** – Access based on job roles.

---

8. What is the difference between Pipes, Buffers, and Shared Memory?

Answer:

Feature	Pipes	Buffers	Shared Memory
Definition	A communication channel between processes	Temporary storage for data transfer	Memory segment shared by multiple processes

Feature	Pipes	Buffers	Shared Memory
Type	Unidirectional or bidirectional	Temporary storage	Memory mapped
Usage	IPC (Inter-Process Communication)	Data transfer between processes	Fast data sharing

---

9. What are the types of Pipes?

Answer:

1. **Named Pipes (FIFOs)** – Exist beyond process execution.
2. **Unnamed Pipes** – Exist only while processes are running.

---

10. What is Device Management in OS?

Answer:
Device Management handles **hardware resources** like printers, hard disks, and USB devices.

Functions:
✅ Allocates & deallocates devices.
✅ Provides drivers for communication.
✅ Schedules I/O requests.

Disk Scheduling Algorithms

11. What is Disk Scheduling? Why is it needed?

Answer:
Disk scheduling determines the order in which disk I/O requests are processed to **minimize seek time and improve performance**.

---

12. What are the different Disk Scheduling Algorithms?

Answer:

Algorithm	Description	Pros	Cons

Algorithm	Description	Pros	Cons
FCFS (First-Come, First-Serve)	Serves requests in arrival order.	Fair and simple	High seek time
SSTF (Shortest Seek Time First)	Processes the request nearest to the current head position.	Reduces seek time	Causes **starvation**
SCAN (Elevator Algorithm)	Moves disk head in one direction, then reverses.	Efficient for high loads	Biased towards middle cylinders
C-SCAN (Circular SCAN)	Similar to SCAN but moves in one direction only.	Reduces wait time for new requests	Longer seek for some requests
LOOK	Like SCAN, but stops at the last request before reversing.	Reduces unnecessary movement	Similar to SCAN
C-LOOK	Like C-SCAN but stops at last request before restarting.	Improves efficiency	-

---

13. Which Disk Scheduling Algorithm is best?

Answer:
It depends on system needs:
✅ **SSTF** – Good for reducing seek time.
✅ **SCAN / C-SCAN** – Good for heavy loads and prevents starvation.
✅ **LOOK / C-LOOK** – Reduces unnecessary movement.

---

14. What is Seek Time, Latency, and Transfer Time?

Answer:

1. **Seek Time** – Time to move the disk head to the correct track.
2. **Rotational Latency** – Time for the correct sector to rotate under the read/write head.
3. **Transfer Time** – Time to transfer data from disk to memory.

IMPORTANT QUESTIONS

UNIT-I: Operating System – 5 Marks & 10 Marks Questions

---

5 Marks Questions (Short Answer Type)

1. Define an Operating System. What are its main functions?
2. Explain the history and evolution of Operating Systems.
3. What are the key functions of an Operating System?
4. What is the difference between Multiprogramming and Multitasking OS?
5. Define Batch Processing System and state its advantages & disadvantages.
6. What is a Time-Sharing Operating System? Give an example.
7. Explain Real-Time Operating System (RTOS) with examples.
8. What are the different types of Operating Systems?
9. Compare Monolithic Kernel and Microkernel.
10. How does a Process Control System differ from a Real-Time System?

---

10 Marks Questions (Long Answer Type)

1. What is an Operating System? Explain its functions in detail.
2. Discuss the evolution of Operating Systems from early systems to modern OS.
3. Compare different types of Operating Systems with examples.
4. Explain Batch Processing, Multiprogramming, and Time-Sharing Operating Systems with examples.
5. What is a Real-Time Operating System (RTOS)? Explain Hard and Soft RTOS with examples.
6. Explain the key features of an Operating System with examples.
7. Differentiate between Personal Computer OS, Workstation OS, and Handheld Device OS.
8. Compare Multiprogramming, Multitasking, Multiprocessing, and Multithreading OS.
9. What is Process Control in an Operating System? How is it used in Real-Time Systems?
10. Explain OS services and system calls with examples.

UNIT-II: Processor & Process Management – 5 Marks & 10 Marks Questions

---

5 Marks Questions (Short Answer Type)

1. What is the difference between Processor Mode and User Mode?
2. Define Kernel. What are its types?
3. Explain System Calls with examples.
4. What is the difference between System Calls and System Programs?
5. What do you mean by Process Abstraction?
6. Define Process Hierarchy with an example.
7. What are Threads? How are they different from Processes?
8. List and explain different Threading Issues.

9. What are Thread Libraries? Name some commonly used ones.
10. Define Process Scheduling and its types.

---

10 Marks Questions (Long Answer Type)

1. Explain Processor Mode and User Mode with examples.
2. What is a Kernel? Compare Monolithic Kernel and Microkernel.
3. Explain System Calls in detail with their types and examples.
4. Discuss the concept of Process Abstraction and Process Hierarchy with a diagram.
5. Explain Threads, their benefits, and differences from Processes.
6. Discuss different Threading Issues and how they affect performance.
7. What are Thread Libraries? Explain POSIX, Windows, and Java Thread Libraries.
8. Describe Process Scheduling in detail. Compare Non-Preemptive and Preemptive Scheduling.
9. Explain various Process Scheduling Algorithms with examples.
10. Compare different types of Scheduling Algorithms and their real-world applications.

UNIT-III: Process Management – 5 Marks & 10 Marks Questions

---

5 Marks Questions (Short Answer Type)

1. Define Deadlock. What are its main causes?
2. Explain Deadlock Characterization.
3. What are the Necessary and Sufficient Conditions for Deadlock?
4. What is Deadlock Prevention? Mention any two techniques.
5. How does Deadlock Avoidance work? Explain briefly.
6. What is the difference between Deadlock Detection and Deadlock Recovery?
7. Define the Critical Section Problem. Why is it important?
8. What are Semaphores? How are they used in Process Synchronization?
9. List and explain different Interprocess Communication (IPC) methods.
10. What is the Producer-Consumer problem? Give an example.

---

10 Marks Questions (Long Answer Type)

1. Explain Deadlock with an example. Discuss its causes and effects.
2. Describe the four necessary and sufficient conditions for Deadlock.
3. Explain Deadlock Prevention techniques in detail.
4. What is Deadlock Avoidance? Explain the Banker's Algorithm with an example.
5. Discuss Deadlock Detection and Recovery methods.
6. What is the Critical Section Problem? Explain the solutions using Semaphores.
7. Explain different methods of Interprocess Communication (IPC) with examples.
8. What is Process Synchronization? Explain the need for synchronization mechanisms.
9. Describe the Producer-Consumer Problem with its implementation using Semaphores.
10. Explain the Reader-Writer Problem. Discuss its solutions using Semaphores.

Here are the **5-mark and 10-mark questions** for **UNIT-IV: Memory Management** in **Operating Systems**.

5-Mark Questions

1. Define **Physical Address Space** and **Virtual Address Space** with an example.
2. Explain **Fixed Partition Memory Allocation** with its advantages and disadvantages.
3. Describe **Variable Partition Memory Allocation** and its benefits over fixed partitions.
4. What is **Paging**? Explain how it helps in memory management.
5. Differentiate between **Paging and Segmentation**.
6. What is **Virtual Memory**? How does it benefit an operating system?
7. Explain **Internal Fragmentation** and **External Fragmentation** with examples.
8. Describe **Page Table** and its role in Paging.
9. What is **Page Replacement**? Name two common **page replacement algorithms**.
10. Explain **Segmentation** and how it differs from Paging.
11. What is **Demand Paging**? How does it improve memory utilization?
12. Discuss the role of **TLB (Translation Lookaside Buffer)** in Paging.
13. Explain the concept of **Thrashing** in Virtual Memory.
14. How does the **FIFO Page Replacement Algorithm** work?
15. What are the **different types of fragmentation** in memory allocation?

10-Mark Questions

1. Explain **Physical and Virtual Address Spaces** with a diagram and examples.
2. Discuss **Fixed and Variable Partitioning** memory allocation strategies with advantages, disadvantages, and examples.
3. Explain **Paging in detail**, including page tables, page frames, and page replacement policies.
4. Describe **Segmentation** in memory management. How is it different from Paging?
5. What is **Virtual Memory**? Explain its working, benefits, and implementation techniques.
6. Discuss **Page Replacement Algorithms** (FIFO, LRU, Optimal) with examples.
7. Explain **Demand Paging and Page Fault Handling** with a detailed example.
8. What is **Thrashing**? Explain its causes and solutions in virtual memory management.
9. Explain the **concept of fragmentation (internal and external)** and how it impacts memory allocation.
10. Describe **Memory Management in Modern Operating Systems**, covering paging, segmentation, and virtual memory.

Here are the **5-mark and 10-mark questions** for **UNIT-V: File and I/O Management, OS Security** in **Operating Systems**.

5-Mark Questions

File and Directory Management

1. What is a **File System**? Explain its importance in an OS.
2. Explain **File Operations** in an operating system.
3. What are the **different types of directory structures** in an OS?
4. Describe **Single-Level and Two-Level Directory Structures** with examples.
5. What is a **Hierarchical Directory Structure**? How does it work?
6. Explain **File Allocation Methods**: Contiguous, Linked, and Indexed.
7. Compare **Contiguous and Linked File Allocation** methods.
8. What is **Indexed File Allocation**? How does it improve file access?
9. Define **File Attributes** and their role in file management.
10. Explain **Access Control Mechanisms** in file security.

I/O Management and OS Security

11. What is **Device Management** in an operating system?
12. Explain the concept of **Buffering** in I/O operations.
13. What is **Spooling**? How does it enhance I/O efficiency?
14. Describe the use of **Pipes** in Interprocess Communication (IPC).
15. What is **Shared Memory**? How is it used for process communication?
16. Define **Disk Scheduling** and explain its need.
17. What are **different types of Disk Scheduling algorithms**?
18. Explain **First-Come, First-Served (FCFS) Disk Scheduling** with an example.
19. What is **Shortest Seek Time First (SSTF) Scheduling**? How does it work?
20. Explain the **SCAN (Elevator) Disk Scheduling Algorithm**.

10-Mark Questions

File and Directory Management

1. Explain **File Operations and File Structure** in an operating system.
2. Discuss **Directory Structures** in detail: Single-Level, Two-Level, Tree, Acyclic, and Graph directories.
3. Compare and contrast **Contiguous, Linked, and Indexed File Allocation** with diagrams.
4. Describe **File Access Methods**: Sequential, Direct, and Indexed Access.
5. What is **File System Security**? Explain file permissions and access control mechanisms.

I/O Management and OS Security

6. Explain **Device Management in Operating Systems** with examples.
7. Describe **Pipes, Buffers, and Shared Memory** in Interprocess Communication (IPC).
8. Explain **Disk Scheduling Algorithms (FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK)** with examples.
9. Discuss **Disk Structure, Disk Formatting, and Disk Management Techniques**.
10. What is **Operating System Security**? Explain threats, authentication, and access control.