Operating System (OS)

PGTRB Computer Science - Latest Study Materials-2025 – D. Sundaravel M.Sc.B.Ed(CS) -9751894315

Title: SYSTEM SOFTWARE AND OPERATING SYSTEM

1. **Introduction to System Software and Operating Systems** **2**
2. **System Software** **6**
    - Machine, Assembly, and High-Level Languages
    - Compilers and Interpreters
    - Loading, Linking, and Relocation
    - Macros
    - Debuggers
3. **Basics of Operating Systems** **10**
    - Operating System Structure
    - Operations and Services
    - System Calls
    - Operating System Design and Implementation
    - System Boot
4. **Process Management** **12**
    - Process Scheduling and Operations
    - Inter-process Communication
    - Communication in Client-Server Systems
    - Process Synchronization
    - Critical-Section Problem
    - Peterson's Solution
    - Semaphores
    - Synchronization
5. **Threads** **39**
    - Multicore Programming
    - Multithreading Models
    - Thread Libraries
    - Implicit Threading
    - Threading Issues
6. **CPU Scheduling** **41**
    - Scheduling Criteria and Algorithms
    - Thread Scheduling
    - Multiple Processor Scheduling
    - Real-Time Scheduling
7. **Deadlocks** **49**
    - Deadlock Characterization
    - Methods for Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance and Detection
    - Recovery from Deadlock
8. **Memory Management** **56**
    - Contiguous Memory Allocation
    - Swapping
    - Paging
    - Segmentation
    - Demand Paging
    - Page Replacement
    - Allocation of Frames
    - Thrashing

Padasalai

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- Memory-Mapped Files

Introduction to Operating Systems

An **operating system (OS)** acts as an intermediary between the user and the computer hardware, essentially serving as the interface between the two. Its primary purpose is to provide an environment in which the user can execute programs efficiently and conveniently.

- An operating system is software that manages both computer hardware and software. It ensures that hardware resources are used appropriately and prevents user programs from interfering with the system's operation.
- The operating system runs continuously on the computer, typically referred to as the **kernel**, while other programs (including applications) are separate from it.
- It is responsible for allocating resources such as memory, processors, and input/output devices among various programs.

History of Operating Systems

The history of operating systems has evolved significantly over time. Here is a snapshot of some key developments:

Era	Key Developments	Examples
1950s-1960s	The first operating system, **GM-NAA I/O**, was created in 1956 by General Motors.	GM-NAA I/O (1956)
1960s	IBM developed the time-sharing system, TSS/360, leading to early multi-user OSs.	OS/360, DOS/360, TSS/360
1970s	**Unix** popularized simplicity and multitasking; personal computers began to emerge.	Unix (1971), CP/M (1974)
1980s	**Graphical User Interfaces (GUIs)** gained traction, and networking features became standard.	Apple Macintosh (1984), Windows (1985)
1990s	Open-source **Linux** emerged; **Windows** and **Mac OS** improved their GUIs.	Linux (1991), Windows 95 (1995)
2000s-Present	Mobile OSs became dominant, and **cloud computing** and **virtualization** advanced.	iOS (2007), Android (2008)

Characteristics of Operating Systems

Operating systems have several important characteristics that define their functionality:

- **Device Management:** The OS tracks and controls all devices, often referred to as the **Input/Output controller**. It determines which process gets access to a device and for how long.
- **File Management:** The OS allocates and de-allocates resources, managing file systems and ensuring that resources are allocated properly.
- **Job Accounting:** The OS tracks the time and resources used by various jobs or users.
- **Error Detection:** It includes debugging methods, such as dumps, traces, and error messages, to identify and manage errors.
- **Memory Management:** The OS manages the primary memory, keeping track of memory usage and allocating memory to processes as needed.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Operating System (OS)

- **Processor Management:** The OS allocates the CPU to processes and de-allocates it when no longer needed.
- **Performance Control:** The OS monitors and manages the performance of the system, tracking delays and system responsiveness.
- **Security:** The OS ensures that unauthorized users cannot access programs or data by using various protection methods, such as passwords.
- **Convenience and Efficiency:** The OS makes the system user-friendly and ensures the efficient use of system resources.
- **Evolution:** The OS is designed to evolve, allowing new features and functions to be added without disrupting existing services.
- **Throughput:** The OS maximizes throughput by ensuring that processes are handled efficiently and without excessive delays.

Types of Operating Systems

There are various types of operating systems, each with unique features suited to different purposes:

- **Windows OS:**
  - **Developer:** Microsoft
  - **Key Features:** User-friendly interface, wide software and hardware support, strong gaming support.
  - **Advantages:** Easy to use, broad support from third-party apps, regular updates.
  - **Typical Use Cases:** Personal computing, business environments, gaming.
- **macOS:**
  - **Developer:** Apple
  - **Key Features:** Sleek, intuitive user interface, strong integration with Apple products, robust security.
  - **Advantages:** Optimized for Apple hardware, seamless experience across the Apple ecosystem.
  - **Typical Use Cases:** Creative industries (e.g., design, video editing), personal computing, professional environments.
- **Linux:**
  - **Developer:** Community-driven (various distributions)
  - **Key Features:** Open-source, highly customizable, lightweight, robust security.
  - **Advantages:** Free, strong community support, suitable for servers and development.
  - **Typical Use Cases:** Servers, programming, tech enthusiasts.
- **Unix:**
  - **Developer:** Originally AT&T Bell Labs, with various commercial and open-source versions.
  - **Key Features:** Multi-user, multitasking, strong security, and portability.
  - **Advantages:** Reliable, robust, suited for high-performance computing and servers.
  - **Typical Use Cases:** Servers, workstations, academic and research environments.

Functionalities of an Operating System

The OS is responsible for several essential functions:

1. **Resource Management:** The OS acts as a resource manager, ensuring that hardware is allocated efficiently and that multiple users or processes do not overload the system.
2. **Process Management:** The OS manages the execution of processes, including scheduling, execution, and termination.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

3. **Storage Management:** The OS uses file systems to manage storage devices like hard disks, handling data organization, retrieval, and storage.
4. **Memory Management:** The OS tracks memory usage, allocating and deallocating space for processes as required.
5. **Security and Privacy Management:** The OS protects data from unauthorized access, ensuring privacy and security.

The Role of the OS in the User Interface

The OS functions as the interface between the user and the hardware, with the following layers:

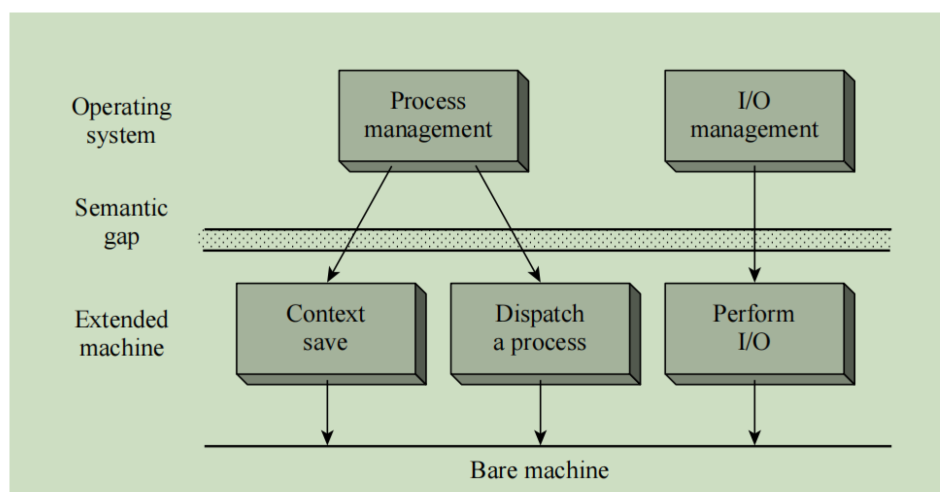
1. User
2. Application Programs/System Programs
3. Operating System
4. Hardware

Conceptual View of the Computer System

An OS coordinates hardware resources, allowing system and application programs to function properly. It creates an environment where applications can execute efficiently by managing resources like the CPU, memory, and storage.

Layered Design of Operating System

The layered OS design divides the operating system into different layers to separate concerns and simplify testing and debugging. The **extended machine layer** manages operations like context switching, swapping, and I/O initiation, while the **OS layer** sits above it, providing services like process scheduling and memory management.



Purposes and Tasks of Operating Systems

- **Purpose:** The OS controls resource allocation, provides an interface for coding and debugging, and ensures efficient use of computing resources.
- **Tasks:** The OS performs several tasks, including providing access to editors, compilers, and loaders for translating programs, and managing I/O operations.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

www.Padasalai.Net www.TrbTnpsc.com

I/O System Management

The I/O subsystem manages device status and includes buffering, caching, and spooling mechanisms to handle data flow efficiently.

Drivers for Specific Hardware Devices

The OS relies on drivers for specific hardware devices (e.g., printers, network cards). These drivers ensure proper communication between hardware and software.

Components of an Operating System

The OS consists of two primary components:

- **Shell:** The user interface that interacts with the user, interpreting commands and providing feedback.
- **Kernel:** The core part of the OS, responsible for managing system resources and acting as an intermediary between the hardware and software.

Types of Kernels

1. **Monolithic Kernel**
2. **Microkernel**
3. **Hybrid Kernel**
4. **Exokernel**

Padasalai

Difference Between 32-Bit and 64-Bit Operating Systems

32-Bit OS	64-Bit OS
Suitable for 32-bit processors	Can run on both 32-bit and 64-bit processors.
Less efficient performance	More efficient performance
Can address 2^32 bytes of RAM	Can address 2^64 bytes of RAM

Fundamental Goals of Operating Systems

1. **Efficient Use:** Ensuring efficient use of computer resources.
2. **User Convenience:** Providing user-friendly methods for interacting with the system.
3. **Non-Interference:** Preventing interference between users or processes.

Advantages and Disadvantages of Operating Systems

- **Advantages:**
  - Efficient memory management
  - Better hardware utilization
  - Enhanced security and user control
  - Multi-tasking and resource allocation

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **Disadvantages:**
  - Some OS can be difficult to use for beginners
  - High maintenance and cost
  - Vulnerability to security threats

System Software

Introduction

System software refers to a collection of low-level programs that manage and control a computer's hardware and provide essential services for higher-level application software. It serves as an intermediary between hardware components and users, ensuring efficient system operation.

Types of Software

Software is broadly categorized into:

1. **System Software** – Software that directly manages and controls hardware, including the operating system, utility programs, device drivers, firmware, and programming language translators.
2. **Application Software** – Software designed for end-user tasks, such as word processors, media players, and business applications.

What is System Software?

Padasalai

System software provides a platform for other software to function effectively. Examples include:

- **Operating systems (OS):** Windows, Linux, macOS, etc.
- **Antivirus software:** Protects the system from malware.
- **Disk formatting software:** Prepares storage devices for use.
- **Language translators:** Compilers, assemblers, and interpreters.

System software is typically developed using low-level programming languages to interact closely with hardware components. It is often pre-installed by computer manufacturers and enables the smooth execution of application programs.

Functions of System Software

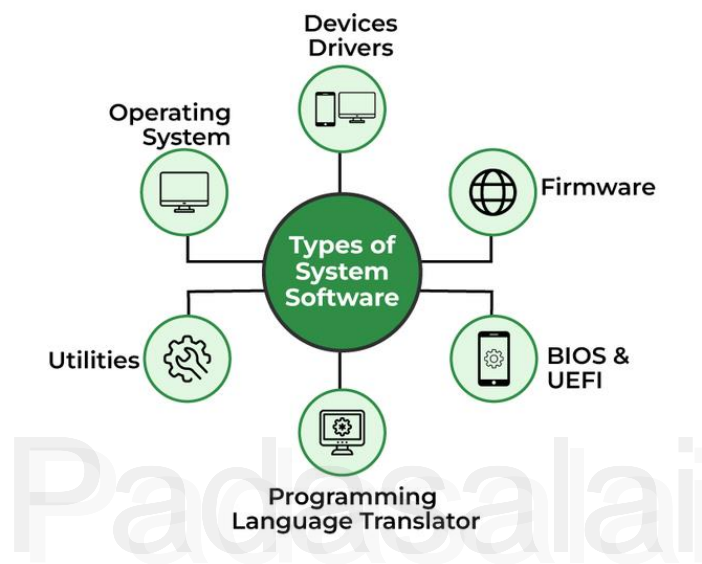
1. **Hardware Communication** – Acts as an interface between hardware and software components, enabling efficient interaction.
2. **Resource Management** – Manages memory, CPU, and other resources to optimize performance.
3. **Security** – Implements firewalls, encryption, and antivirus measures to protect data and system integrity.
4. **User Interface** – Provides graphical or command-line interfaces for user interaction.
5. **Application Support** – Facilitates the installation and execution of applications.
6. **Customization** – Allows users to configure settings for enhanced functionality and user experience.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Key Features of System Software

- **Memory Management:** Allocates and deallocates memory as needed.
- **Processor Management:** Schedules and prioritizes processes.
- **File Management:** Organizes and maintains files and directories.
- **Security:** Implements authentication and access controls.
- **Error Handling:** Detects and logs errors to enhance system reliability.
- **Process Scheduling:** Uses algorithms to allocate resources efficiently.

Types of System Software



1. Operating System (OS)

An operating system is the most crucial system software that manages hardware resources and provides essential services to applications. Common functions include:

- **Resource Management** – Allocates memory, CPU time, and storage.
- **Process Management** – Controls and schedules tasks.
- **Memory Management** – Optimizes RAM usage.
- **Security** – Implements access control and encryption.
- **File Management** – Handles file storage and retrieval.
- **Device Management** – Controls input/output devices through drivers.

2. Machine, Assembly, and High-Level Languages

Programming languages are classified into three major categories based on their abstraction level:

- **Machine Language:** The lowest-level language, consisting of binary code (0s and 1s), directly executed by the CPU.
- **Assembly Language:** A low-level language that uses mnemonic codes (e.g., ADD, MOV) instead of binary, requiring an assembler for translation.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **High-Level Language:** More abstract and user-friendly, using English-like syntax (e.g., Python, Java, C++) that requires a compiler or interpreter for execution.

3. Programming Language Translators

These programs convert high-level code into machine-readable formats. Types include:

- **Compiler:** Converts entire code into machine language before execution (e.g., GCC, Javac).
- **Interpreter:** Translates code line-by-line and executes immediately (e.g., Python, PHP).
- **Assembler:** Converts assembly language into machine code.

4. Compilers and Interpreters

- **Compiler:** Scans the entire code, translates it into machine code, and executes it as a whole. Faster execution but requires complete compilation before running.
- **Interpreter:** Translates and executes code line-by-line, making debugging easier but slower than a compiler.

5. Loading, Linking, and Relocation

- **Loading:** The process of placing a program into memory for execution.
- **Linking:** Combines multiple code modules and libraries into a single executable.
- **Relocation:** Adjusts memory addresses so programs can execute correctly regardless of where they are loaded.

Padasalai

6. Macros

A macro is a set of instructions grouped under a name, allowing repeated execution with a single command. Macros are commonly used in assembly language to reduce redundancy and improve efficiency.

7. Debuggers

A debugger is a tool used to test and debug programs by identifying and fixing errors. Features include:

- **Breakpoints:** Pause execution at specific points.
- **Step Execution:** Run code line by line for detailed analysis.
- **Variable Inspection:** Examine variable values during execution.
- **Error Tracing:** Detect and report runtime errors.

8. Device Drivers

Device drivers enable the OS to communicate with hardware components such as printers, graphics cards, and external storage devices.

9. Firmware

Firmware is a low-level software embedded in hardware components, allowing them to function correctly. Examples include BIOS and UEFI chips, which initialize hardware during startup.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

10. Utility Software

Utility software enhances system performance and maintenance. Examples include:

- **Antivirus Software:** Protects against malware.
- **Disk Defragmentation Tools:** Optimizes storage organization.
- **File Compression Tools:** Reduces file size (e.g., WinRAR, WinZip).
- **Backup Software:** Ensures data recovery in case of failures.

Uses of System Software

- **Operating Systems:** Manage computing devices from personal computers to supercomputers.
- **Device Drivers:** Facilitate hardware-software communication.
- **Firmware:** Controls embedded system behavior.
- **System Utilities:** Perform maintenance tasks such as data backup and disk cleanup.
- **Programming Tools:** Aid in software development, debugging, and performance optimization.

Challenges in System Software

1. **Compatibility Issues:** Some software may not support specific hardware components.
2. **Security Risks:** Vulnerabilities in system software can be exploited by malware.
3. **Performance Concerns:** High resource consumption may slow down the system.
4. **Update Problems:** New updates may introduce bugs or compatibility conflicts.
5. **Licensing Restrictions:** Certain system software requires paid licensing.
6. **User Interface Complexity:** Some system software may be difficult to navigate.

Padasalai

Advantages of System Software

- **Efficient Resource Management** – Enhances CPU, memory, and device performance.
- **Improved Security** – Protects against cyber threats.
- **User-Friendly Interfaces** – Enhances accessibility.
- **Reliability** – Ensures stable system operation.
- **Interoperability** – Supports diverse software and hardware.

Disadvantages of System Software

- **Complexity:** Can be difficult for non-technical users.
- **High Cost:** Some software requires expensive licenses.
- **System Overhead:** May consume excessive resources.
- **Security Risks:** Vulnerabilities can be exploited by hackers.
- **Upgrade Challenges:** New versions may disrupt existing systems.
- **Limited Customization:** Some system software has fixed settings.
- **Dependency:** Essential for application functionality, making replacement difficult.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Operating System (OS)

Difference Between System Software and Application Software

Feature	System Software	Application Software
Purpose	Manages hardware and system resources	Performs specific user tasks
Language	Written in low-level languages	Written in high-level languages
Generalization	General-purpose	Task-specific
Execution	Runs when the system is powered on	Runs when initiated by the user
Example	Operating systems, drivers	Photoshop, media players

3. Basics of Operating Systems

An operating system (OS) consists of various components that work together to manage hardware and software resources. The main structures include:

- **Monolithic Kernel**: A large single-layer kernel handling all OS functions, offering high performance but reduced modularity.
- **Microkernel**: A minimal kernel managing essential processes, with additional services running in user space, leading to better modularity and security.
- **Layered Structure**: OS functionalities are divided into layers, with each layer interacting only with adjacent layers, enhancing maintainability and organization.
- **Modular Approach**: A flexible system where functionalities are separate modules that can be loaded dynamically, allowing for easier updates and scalability.
- **Hybrid Kernel**: A combination of monolithic and microkernel designs, balancing performance and modularity (e.g., Windows NT and macOS).

Operations and Services

The OS provides key operations and services, including:

- **Process Management**: Creating, scheduling, and terminating processes; implementing multitasking and context switching.
- **Memory Management**: Allocating and managing system memory efficiently through paging and segmentation.
- **File System Management**: Handling file storage, access, security, and directory structures.
- **Device Management**: Controlling communication between hardware and software via device drivers.
- **User Interface**: Providing command-line (CLI) or graphical user interfaces (GUI) for user interaction.
- **Security and Access Control**: Implementing authentication, encryption, and user privilege management.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

System Calls

System calls are essential for communication between application programs and the OS. They are classified as:

- **Process Control**: Creating and terminating processes (e.g., fork(), exit()).
- **File Management**: Creating, deleting, and manipulating files (e.g., open(), read(), write()).
- **Device Management**: Requesting device access and performing I/O operations (e.g., ioctl(), read(), write()).
- **Information Maintenance**: Retrieving system data and setting system time (e.g., getpid(), gettimeofday()).
- **Communication**: Enabling interprocess communication (e.g., pipe(), socket(), send(), recv()).
- **Security & Protection**: Managing access control and permissions (e.g., chmod(), chown()).

Operating System Design and Implementation

OS design involves architectural decisions, including:

- **Performance Optimization**: Efficient resource utilization and process scheduling techniques such as round-robin, priority scheduling, and multilevel queues.
- **Security and Protection**: Implementing authentication, encryption, and intrusion detection systems.
- **Portability**: Ensuring compatibility with different hardware architectures and platforms.
- **Scalability**: Supporting multiple users, multitasking, and cloud computing environments.
- **User-Centric Design**: Enhancing usability, customization, and accessibility features.
- **Reliability and Fault Tolerance**: Implementing error detection, logging, and recovery mechanisms.

Padasalai

System Boot

The boot process initializes the OS when a computer starts. Key steps include:

1. **Power-On Self-Test (POST)**: Checking hardware components, including memory and storage devices.
2. **Loading Firmware (BIOS/UEFI)**: Initializing the hardware, performing diagnostic tests, and locating the OS bootloader.
3. **Bootloader Execution**: Loading the OS kernel into memory (e.g., GRUB for Linux, Windows Boot Manager for Windows).
4. **Kernel Initialization**: Setting up system processes, memory management, and hardware drivers.
5. **System Services Startup**: Launching background services and daemons.
6. **User Login and Session Start**: Allowing user interaction with the system via a login prompt or GUI.

Process Management

Process Scheduling and Operations
Inter-process Communication
Communication in Client-Server Systems
Process Synchronization
Critical-Section Problem
Peterson’s Solution
Semaphores & Synchronization

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

4.Introduction to Process Management

Process management in an operating system (OS) is responsible for handling multiple processes efficiently. In a single-tasking or batch processing system, process management is relatively straightforward as only one process is active at a time. However, in multiprogramming or multitasking systems, multiple processes run concurrently, making process management more complex. The operating system must allocate CPU time effectively among these processes while ensuring smooth execution and coordination.

Key Advantages of Multiprogramming:

- **System Responsiveness:** The system remains responsive even when multiple processes are running.
- **Better CPU Utilization:** The CPU is efficiently utilized by switching between processes when one is waiting for I/O.
- **Interleaved Execution:** When a process is waiting for an I/O operation, another process can use the CPU.

CPU-Bound vs I/O-Bound Processes

A process can be categorized based on its resource requirements:

- **CPU-Bound Process:** Requires more CPU time and spends most of its time in the **running state**.
- **I/O-Bound Process:** Requires more I/O operations and spends most of its time in the **waiting state**.

Process Scheduling

Process scheduling determines which process should run next to improve system performance. The main objectives of process scheduling are:

- **Maximizing CPU utilization**
- **Minimizing throughput time**
- **Improving system response time**

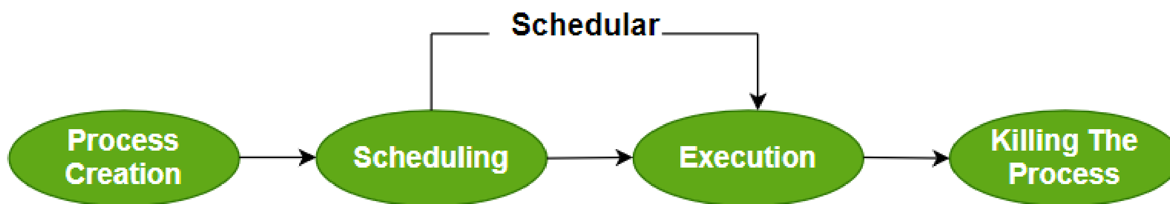
Process Management Tasks

Process management in a multiprogramming or multitasking environment involves several tasks:

1. **Process Creation and Termination:**
    - When a new process is created, it is assigned a Process ID (PID) and a Process Control Block (PCB).
    - Process termination occurs when a process finishes execution or is forcibly terminated by the OS or parent process.
2. **CPU Scheduling:**
    - The OS ensures fair and efficient execution of multiple processes by allocating CPU time to different processes.
3. **Deadlock Handling:**
    - Prevents scenarios where two or more processes are stuck indefinitely, waiting for each other's resources.
4. **Inter-Process Communication (IPC):**
    - Allows processes to communicate via **shared memory** or **message passing**.
5. **Process Synchronization:**
    - Ensures that multiple processes access shared resources safely to prevent race conditions.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Process Operations



Processes undergo various state transitions before termination. The primary operations involved in process management are:

1. Process Creation:

- A new process is created as an instance of a program that runs independently.
- The OS assigns a unique PID and allocates required resources.

2. Scheduling:

- The process moves to the **ready queue** after creation.
- The scheduler selects a process from the queue for execution.

3. Execution:

- The CPU starts working on the process.
- During execution, a process may:
    - Move to the **waiting queue** for I/O operations.
    - Get **blocked** if a higher-priority process requires CPU time.

4. Killing the Process:

- Once the process completes its tasks, the OS terminates it and removes its PCB.

Context Switching

Context switching occurs when the CPU switches from executing one process to another. This involves saving the current process state and loading the new process state.

When Does Context Switching Occur?

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- A **higher-priority process** arrives.
- An **interrupt** occurs.
- A **user-mode to kernel-mode switch** is triggered.
- **Preemptive scheduling** is used.

Context Switch vs Mode Switch

- **Context Switch:** Involves switching from one process to another by saving and restoring process states.
- **Mode Switch:** Occurs when the CPU changes privilege levels (e.g., when handling a system call). A mode switch may lead to a context switch but does not always require one.

Process Scheduling Algorithms

The OS employs different scheduling algorithms to manage process execution efficiently.

1. **First-Come, First-Served (FCFS):**
   - Processes are executed in the order they arrive.
   - **Non-preemptive**, meaning a process runs until it completes or waits for I/O.
2. **Shortest Job First (SJF):**
   - Selects the process with the **shortest burst time**.
   - Minimizes **average waiting time**.
3. **Round Robin (RR):**
   - **Preemptive algorithm** where each process gets a fixed time slot.
   - If a process does not finish in its time slice, it moves to the end of the queue.
   - Ensures **fair CPU distribution** and prevents **starvation**.
4. **Priority Scheduling:**
   - Assigns priority to processes; higher-priority processes execute first.
   - Can be **preemptive** or **non-preemptive**.
5. **Multilevel Queue Scheduling:**
   - Divides the ready queue into multiple priority-based queues.
   - Each queue follows a different scheduling algorithm.

Advantages of Process Management

1. **Supports Multiple Programs:** Enables users to run multiple applications simultaneously.
2. **Ensures Process Isolation:** Prevents one process from interfering with another.
3. **Efficient Resource Allocation:** Distributes CPU and memory fairly among processes.
4. **Smooth Process Switching:** Minimizes delays by handling switching efficiently.

Disadvantages of Process Management

1. **System Overhead:** Managing multiple processes consumes CPU and memory resources.
2. **Increased Complexity:** Implementing scheduling algorithms and resource allocation is challenging.
3. **Risk of Deadlocks:** Improper resource handling can lead to processes being stuck indefinitely.
4. **Frequent Context Switching:** Excessive switching between processes slows down system performance.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Process Scheduling in Operating Systems

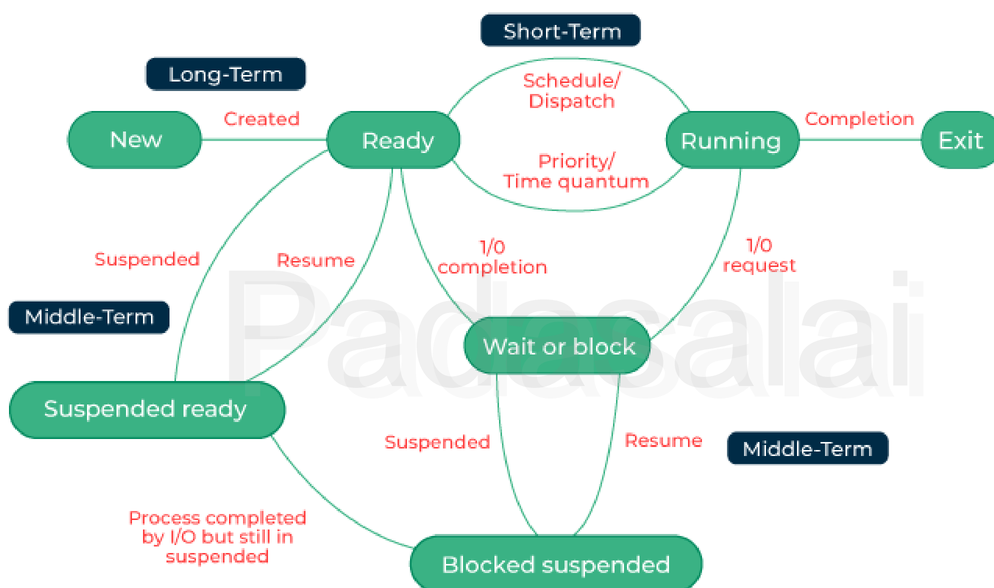
Introduction

A process is the instance of a computer program in execution.

Scheduling is crucial in operating systems with multiprogramming, as multiple processes might be eligible to run simultaneously. One of the key responsibilities of an Operating System (OS) is to decide which programs will execute on the CPU.

Process Schedulers are fundamental components of operating systems responsible for determining the order in which processes are executed by the CPU. They manage how the CPU allocates its time among multiple tasks or processes competing for attention.

What is Process Scheduling?



Process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a particular strategy. Throughout its lifetime, a process moves between various scheduling queues, such as:

- **Ready Queue**
- **Waiting Queue**
- **Device Queue**

Process Scheduler

Process scheduling falls into one of two categories:

Non-Preemptive Scheduling

- A process's resource cannot be taken before it has finished execution.
- When a running process completes or transitions to a waiting state, resources are switched.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Preemptive Scheduling

- The OS can switch a process from the running state to the ready state.
- Switching happens when the CPU gives priority to other processes, replacing the currently active process with a higher-priority process.

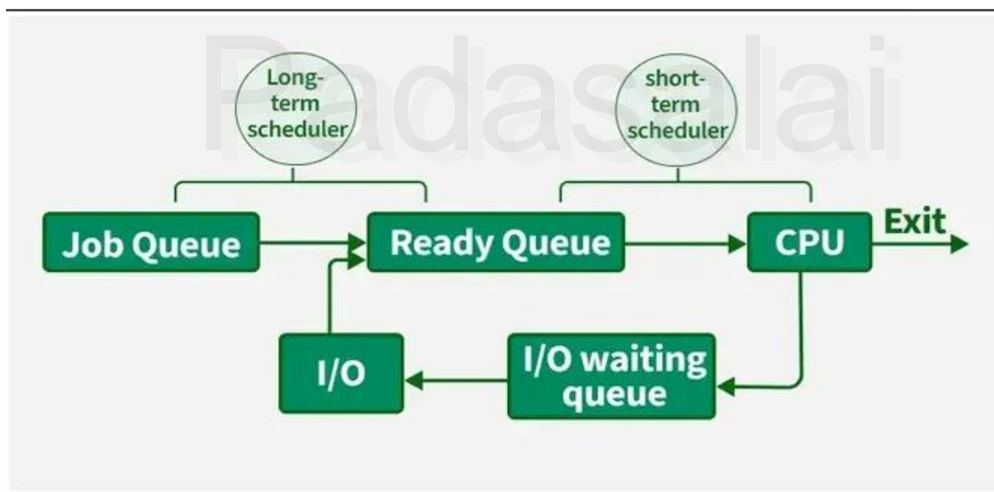
Types of Process Schedulers

There are three types of process schedulers:

1. Long-Term (Job) Scheduler

- Loads a process from disk to main memory for execution.
- Moves processes from **Job Queue** to **Ready Queue**.
- Controls the **Degree of Multiprogramming** (i.e., the number of processes present in the main memory at any point in time).
- Balances between **CPU-bound** and **I/O-bound** processes to maintain system efficiency.
- In some systems, the long-term scheduler may not exist (e.g., time-sharing systems like Microsoft Windows, where every new process is directly added to memory).
- **Slowest** among the three schedulers.

3. Short-Term (CPU) Scheduler



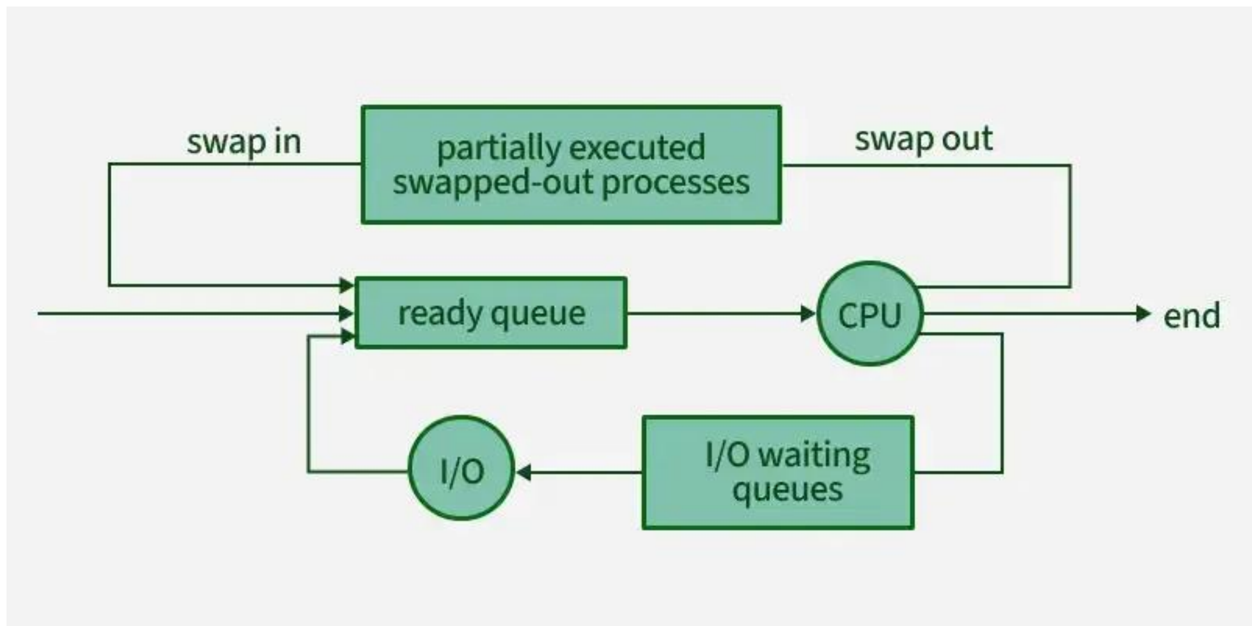
- Selects a process from the **Ready Queue** for execution.
- Must frequently select new processes for CPU allocation to prevent **starvation**.
- Uses different scheduling algorithms to allocate CPU time efficiently.
- Calls the **dispatcher** to load the selected process onto the CPU.
- **Fastest** among the three schedulers.

Dispatcher Responsibilities:

- Saving the context (Process Control Block) of the previously running process if not finished.
- Switching the system mode to **User Mode**.
- Jumping to the proper location in the newly loaded program.
- Time taken by the dispatcher is called **dispatch latency** or **context switch time**.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

3. Medium-Term Scheduler



- Moves a process from **memory to disk** (swapping).
- Reduces the **degree of multiprogramming**.
- Suspended processes are moved to secondary storage when they are waiting for an I/O operation.
- Brings the process back into memory when needed, allowing execution to resume from where it left off.
- **Faster than long-term** but **slower than short-term** scheduling.

Other Schedulers

I/O Scheduler

- Manages the execution of **I/O operations** (e.g., reading and writing to disks or networks).
- Uses algorithms like **First-Come, First-Served (FCFS)** or **Round Robin (RR)**.

Real-Time Scheduler

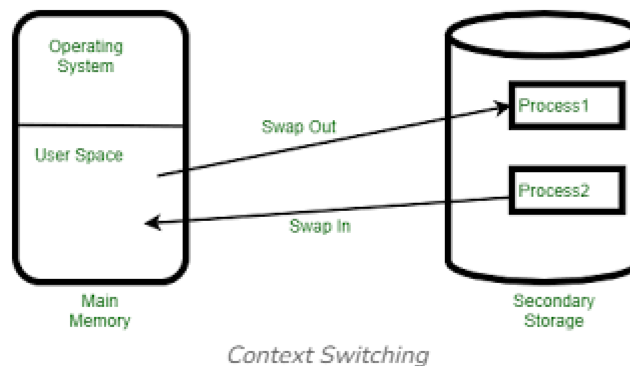
- Ensures that critical tasks complete within a specified time frame in **real-time systems**.
- Uses algorithms like **Earliest Deadline First (EDF)** or **Rate Monotonic (RM)**.

Comparison of Schedulers

Feature	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
Type	Job Scheduler	CPU Scheduler	Process-Swapping Scheduler
Speed	Slowest	Fastest	Intermediate
Controls Multiprogramming	Yes	Limited	Reduces Multiprogramming
Presence in Time-Sharing	Often absent	Minimal presence	Present
Process Re-entry	Yes	No	Yes
Function	Selects processes for ready queue	Selects process for CPU execution	Swaps process in and out of memory

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Context Switching



Context Switching

Context switching allows a process to resume execution from the same point at a later time. It enables **multiple processes to share a single CPU** and is an essential feature of multitasking OS.

Steps in Context Switching:

1. Save the state of the currently running process into its **Process Control Block (PCB)**.
2. Load the state from the PCB of the next process.
3. Update system settings (registers, program counter, etc.).
4. Start executing the next process.

Information Stored in Context Switching:

- **Program Counter**
- **Scheduling Information**
- **Base and Limit Register Values**
- **Currently Used Registers**
- **Changed State**
- **I/O State Information**
- **Accounting Information**

This structured document provides a comprehensive overview of **Process Scheduling**, ensuring clarity while retaining all relevant details.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Inter-Process Communication (IPC)

Processes often need to communicate with each other in various scenarios. For instance, when counting occurrences of a word in a text file, the output of the grep command needs to be passed to the wc command, as in:

```
grep -o -i <word> <file> | wc -l
```

Inter-Process Communication (IPC) is a mechanism that enables processes to communicate, synchronize activities, share information, and prevent conflicts while accessing shared resources.

Types of Processes

Processes can be categorized into two types based on their interaction:

- **Independent Process:** These processes are not affected by the execution of other processes and do not share data or resources. As a result, no inter-process communication is required.
- **Cooperating Process:** These processes interact with each other and share data or resources. They can be influenced by other executing processes. IPC mechanisms allow such processes to communicate and synchronize their actions effectively.

Inter-Process Communication (IPC)

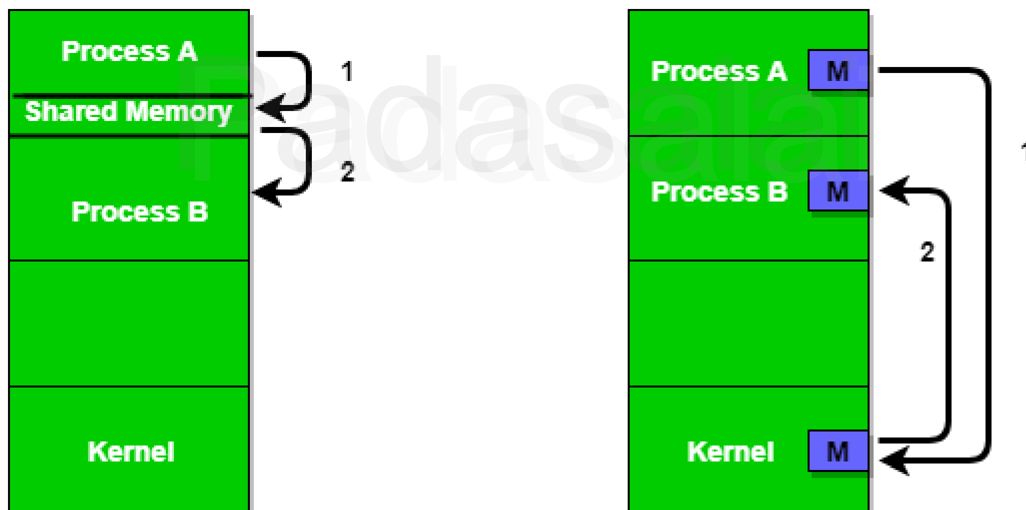


Figure 1 - Shared Memory and Message Passing

IPC enables different programs or processes running on a computer to share information using various techniques, such as shared memory, message passing, or file-based communication. It ensures that cooperating processes can exchange data without interfering with one another.

The two fundamental models of IPC are:

1. **Shared Memory**
2. **Message Passing**

An operating system can implement both methods. First, we will discuss shared memory, followed by message passing.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Shared Memory Communication

In shared memory IPC, processes share a specific memory region where they can read and write data. The implementation of shared memory communication depends on the programmer.

For example, consider two processes, Process1 and Process2, running simultaneously and sharing certain resources or information. Process1 generates data related to computations or resource usage and stores it in shared memory. When Process2 needs to access this information, it reads from the shared memory and takes necessary actions accordingly.

Shared memory facilitates fast communication since processes can directly access common memory areas rather than exchanging messages. However, it requires synchronization mechanisms like semaphores or mutexes to prevent conflicts and ensure consistency.

Message Passing Communication

In message passing, processes communicate by sending and receiving messages through communication channels managed by the operating system. Unlike shared memory, this approach does not require processes to share memory space, making it useful in distributed systems.

Message passing mechanisms include:

- **Pipes**: Unidirectional communication channels between processes.
- **Message Queues**: Data structures maintained by the OS for storing messages until they are retrieved by the recipient process.
- **Sockets**: Allow communication between processes over a network.
- **Remote Procedure Calls (RPCs)**: Enable processes to invoke procedures on remote systems as if they were local function calls.
- **Memory-Mapped Files**: Allow processes to map files into memory and access them as if they were part of the process's address space.

Methods of Inter-Process Communication

Several methods exist for implementing IPC, each suited to different scenarios:

- **Shared Memory**: Fast communication but requires synchronization mechanisms.
- **Message Passing**: Uses system-managed communication channels.
- **Semaphores**: Used to control access to shared resources and prevent race conditions.
- **Signals**: Used for event notification between processes.
- **Pipes and Named Pipes**: Facilitate communication between related or unrelated processes.
- **Memory-Mapped Files**: Allow processes to share data efficiently by mapping files into memory.
- **Remote Procedure Calls (RPCs)**: Allow processes to execute functions on remote systems.
- **Sockets**: Support communication over networks between different systems.

Role of Synchronization in IPC

Synchronization is essential in IPC to control access to shared resources and ensure data consistency. Without proper synchronization, problems such as race conditions can occur, leading to unpredictable results. Common synchronization techniques include:

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **Mutexes (Mutual Exclusion Locks)**: Prevent multiple processes from accessing shared resources simultaneously.
- **Semaphores**: Used for signaling and resource control.
- **Monitors**: High-level synchronization constructs that encapsulate shared resources and operations.
- **Barrier Synchronization**: Ensures that multiple processes reach a certain execution point before proceeding.

Advantages of IPC

- Facilitates communication and resource sharing between processes, improving efficiency.
- Enhances coordination between multiple processes, leading to better system performance.
- Supports the development of distributed systems that span multiple computers or networks.
- Allows implementation of various synchronization mechanisms, such as semaphores, pipes, and sockets.
- Enables modular and scalable system design by allowing processes to operate independently and communicate as needed.

Disadvantages of IPC

- Increases system complexity, making design, implementation, and debugging more challenging.
- May introduce security vulnerabilities if unauthorized processes gain access to shared data.
- Requires careful management of system resources like memory and CPU to prevent performance degradation.
- Can lead to data inconsistencies if multiple processes access or modify shared data simultaneously.
- Message passing mechanisms may introduce overhead due to context switching and data copying.
- Network-based IPC methods like sockets and RPCs can experience latency and reliability issues in distributed systems.

Padasalai

Client-Server Communication in Operating System

Introduction

Client-Server Communication in an Operating System refers to the exchange of data and services between multiple machines or processes. In a Client-Server Communication System, one process or machine acts as a **client**, requesting a service or data, while another process or machine acts as a **server**, providing those services or data to the client. This communication model is widely used in various computing environments, such as **Distributed Systems, Internet Applications, and Networking Applications**. The communication between a server and a client takes place through different **protocols and mechanisms**.

Different Ways of Client-Server Communication

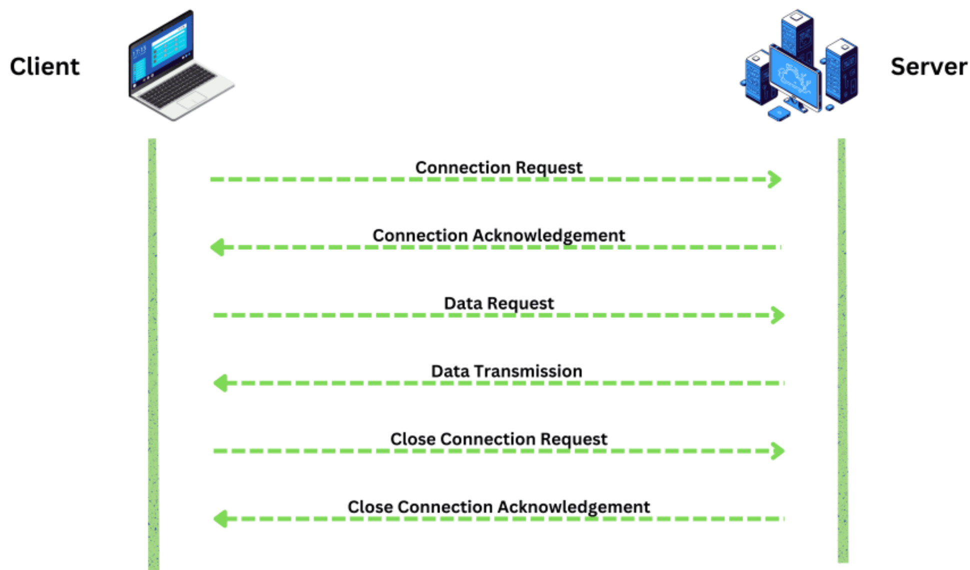
Client-Server Communication can be implemented using different mechanisms:

1. **Sockets Mechanism**
2. **Remote Procedure Call (RPC)**
3. **Message Passing**
4. **Inter-Process Communication (IPC)**
5. **Distributed File Systems**

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

www.Padasalai.Net www.TrbTnpsc.com

1. Sockets Mechanism

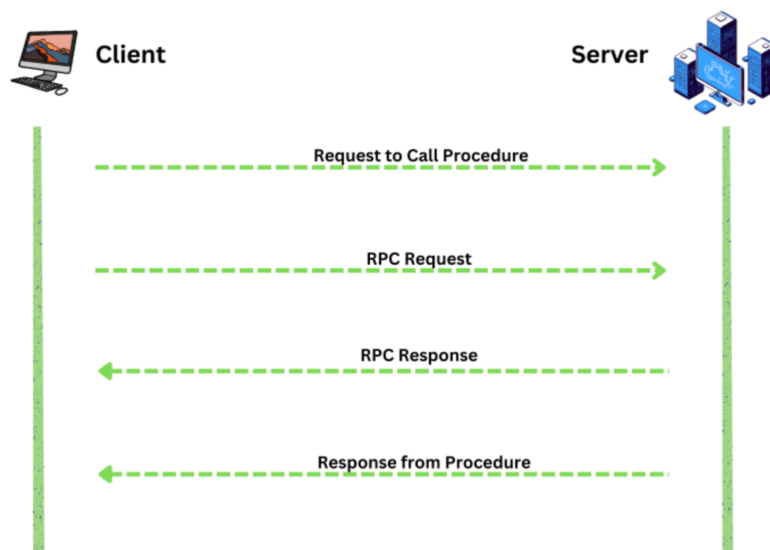


Sockets serve as the endpoints of communication between two machines. They provide a way for processes to communicate with each other, either on the same machine or over a network (such as the Internet). Sockets enable **bidirectional communication** between the client and the server, allowing data to be transferred efficiently.

Client-Server Communication using Sockets:

- The server creates a socket and binds it to a specific address and port.
- The server listens for incoming connections.
- The client establishes a connection to the server's socket.
- The client and server exchange data.
- The connection is closed when the communication is complete.

2. Remote Procedure Call (RPC)



Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

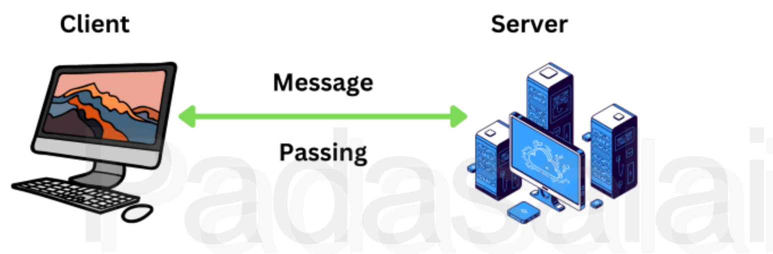
A **Remote Procedure Call (RPC)** is a protocol that allows a client to execute a procedure on a remote server, as if it were a local procedure call. RPC provides a high level of abstraction to the programmer, making distributed computing simpler.

Remote Procedure Call Process:

- The client program issues a procedure call.
- The call is translated into a message and sent over the network to the server.
- The server executes the requested procedure.
- The result is sent back to the client.

RPC is widely used in **client-server architectures** to enable seamless interaction between distributed applications.

3. Message Passing



Message Passing is a communication method in which machines communicate by sending and receiving messages. This approach is commonly used in **Parallel and Distributed Systems**, allowing efficient data exchange.

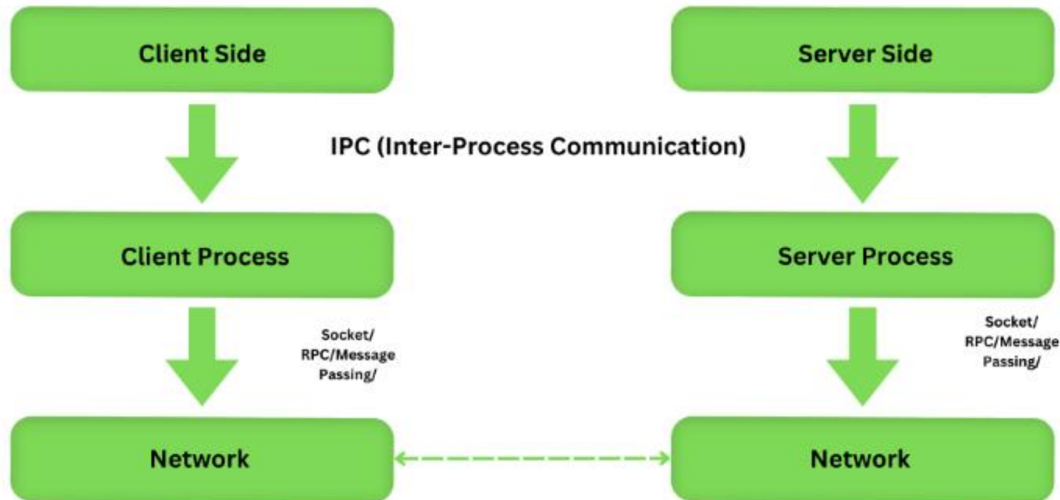
Message Passing Process:

- The sender process creates a message and sends it to the receiver process.
- The receiver process reads and processes the message.
- Message queues may be used to store and manage messages before they are delivered.

This method enables communication **between processes on different machines** and is crucial for achieving parallel execution in distributed systems.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

4. Inter-Process Communication (IPC)



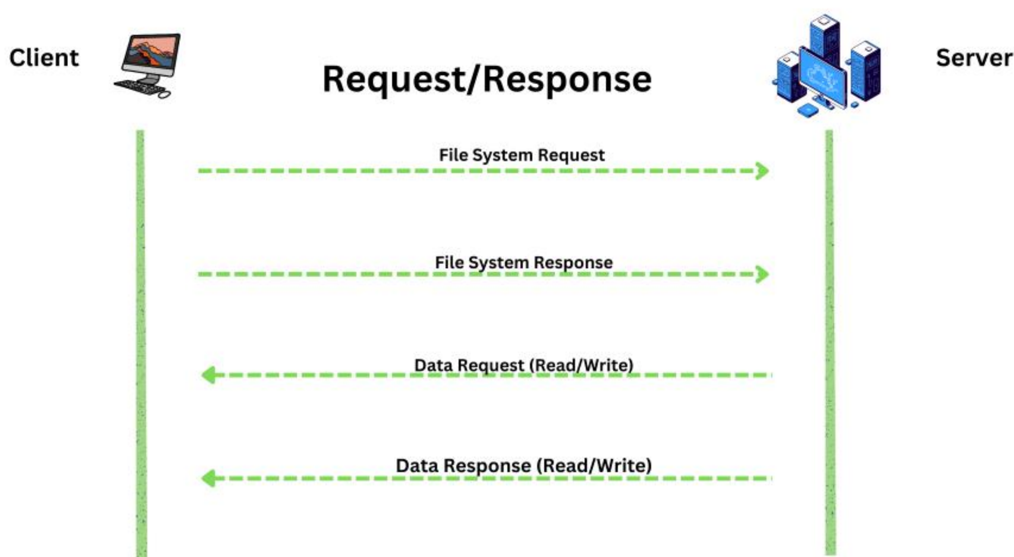
Inter-Process Communication (IPC) allows processes within the same machine to communicate with each other. IPC facilitates **data sharing and synchronization** between different processes running concurrently in an operating system.

IPC Mechanisms:

- **Shared Memory:** Processes share a common memory space to exchange data efficiently.
- **Message Queues:** Processes communicate by sending and receiving messages through system-maintained queues.
- **Semaphores:** Used for synchronization and avoiding race conditions in concurrent processing.
- **Pipes:** Enable unidirectional or bidirectional communication between processes.

IPC is widely used for managing process interactions within an operating system.

5. Distributed File Systems (DFS)



Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

A **Distributed File System (DFS)** enables access to files stored on remote machines in a network. Clients can **access, manipulate, and manage files** as if they were stored locally, even though they reside on a remote server.

Examples of Distributed File Systems:

- **Network File System (NFS):** Allows remote file access using a standard interface.
- **Server Message Block (SMB):** Used primarily in Windows-based systems for shared file access.

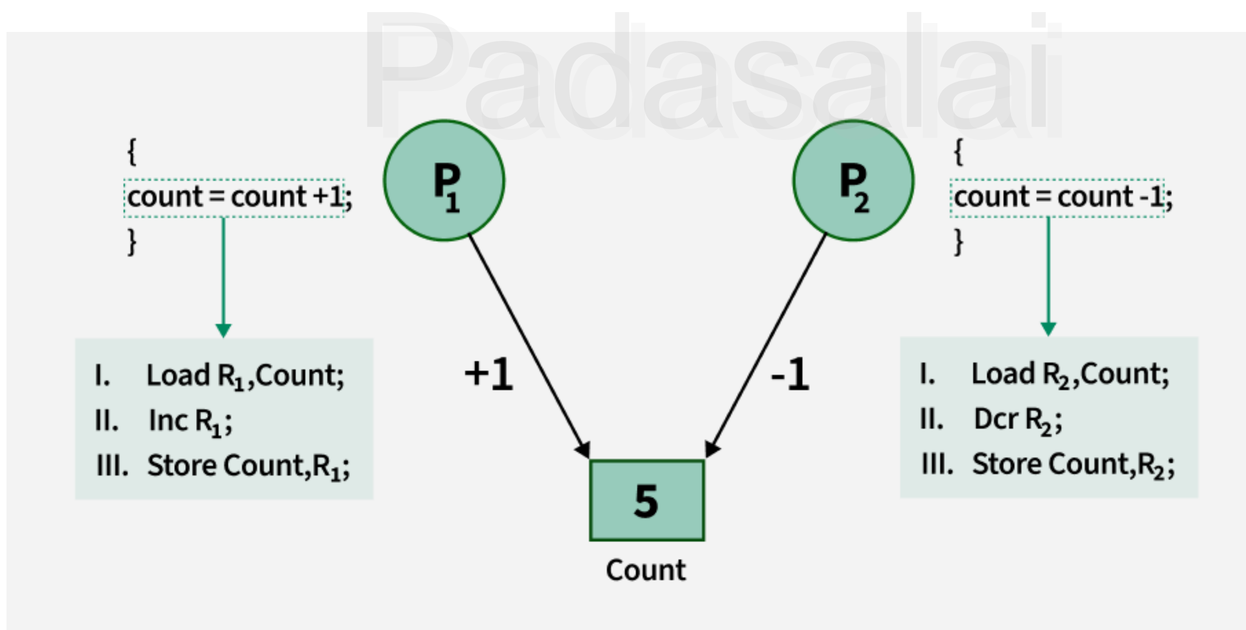
DFS ensures **data consistency, availability, and scalability** in networked environments, making it a critical component of modern distributed computing.

Process Synchronization

Introduction to Process Synchronization

Process synchronization ensures that multiple processes or threads can run concurrently without interfering with each other. The primary goal is to manage shared resources efficiently while preventing inconsistencies in data due to concurrent access. Synchronization techniques such as semaphores, monitors, and critical sections help achieve this goal.

In a multi-process system, synchronization is crucial for maintaining data integrity, avoiding deadlocks, and ensuring predictable execution. Process synchronization plays a vital role in modern operating systems by coordinating process execution.



Types of Processes Based on Synchronization

Processes can be classified into two categories:

1. **Independent Process**: Execution of one process does not affect the execution of other processes.
2. **Cooperative Process**: Execution of one process can affect or be affected by other processes in the system.
    - Synchronization issues mainly arise in cooperative processes due to shared resource access.

Process Synchronization Challenges

Process synchronization addresses several issues arising in a concurrent system:

Problems Due to Lack of Synchronization

1. **Inconsistency**: Multiple processes accessing shared data simultaneously may lead to conflicting updates, resulting in unreliable and incorrect data.
2. **Loss of Data**: If multiple processes write or modify the same resource without coordination, crucial information may be lost, leading to incomplete or corrupted data.
3. **Deadlock**: Processes can get stuck waiting for each other to release resources, rendering the system unresponsive.

Types of Process Synchronization

1. **Competitive Synchronization**: Multiple processes compete for access to a shared resource. Lack of synchronization can lead to inconsistency and data loss.
2. **Cooperative Synchronization**: Processes affect each other, meaning execution of one process influences others. Lack of synchronization can lead to deadlocks.

Example of Process Synchronization in Linux

Consider the Linux command:

```
ps | grep "chrome" | wc
```

- `ps` lists processes running in Linux.
- `grep` filters processes containing "chrome" from `ps` output.
- `wc` counts words in `grep` output.

Here, `ps`, `grep`, and `wc` function as cooperative processes, where one process's output is used as another's input. The operating system manages their synchronization to ensure correct execution.

Conditions Requiring Process Synchronization

1. **Critical Section**: A program segment where shared resources are accessed. Only one process should execute a critical section at a time to prevent data inconsistency.
2. **Race Condition**: A scenario where multiple processes attempt to execute a critical section, and the outcome depends on their execution order.
3. **Preemption**: The operating system suspends a running process to allocate CPU time to another. If the preempted process was updating shared data, inconsistency can arise.

Race Condition Example

- Shared variable `balance = 100`
- Two processes: `deposit(10)` and `withdraw(10)`
  - `deposit`: `balance = balance + 10`
  - `withdraw`: `balance = balance - 10`
- Possible interleaving:
  - `deposit()` fetches `balance` as `100` but is preempted.
  - `withdraw()` runs, making `balance = 90`.
  - `deposit()` resumes, setting `balance = 110` (incorrect final value).

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Preventing Race Conditions

- Treat the critical section as an atomic operation.
- Use synchronization mechanisms like locks or atomic variables.

Critical Section Problem

The **Critical Section Problem** is about designing a mechanism for cooperative processes to access shared resources without data inconsistencies. Solutions must satisfy:

1. **Mutual Exclusion**: Only one process executes in the critical section at a time.
2. **Progress**: If no process is in the critical section, a waiting process should be able to proceed without indefinite delays.
3. **Bounded Waiting**: A process waiting for the critical section should be allowed to enter within a limited number of executions.

Classical Inter-Process Communication (IPC) Problems

1. Producer-Consumer Problem

- **Producer** generates data and places it in a shared buffer.
- **Consumer** retrieves and processes data from the buffer.
- Challenges:
  - **Buffer Overflow**: The producer generates data when the buffer is full.
  - **Buffer Underflow**: The consumer attempts to retrieve data when the buffer is empty.
- **Solution**: Use semaphores to synchronize producer and consumer actions.

Padasalai

2. Readers-Writers Problem

- **Readers** read shared data.
- **Writers** modify shared data.
- Challenges:
  - Allow multiple readers to access data simultaneously.
  - Ensure only one writer accesses the data at a time.
- **Solutions**:
  - **Readers Preference Solution**: Prioritizes readers over writers.
  - **Writers Preference Solution**: Gives writers exclusive access before readers.

3. Dining Philosophers Problem

- Five philosophers sit around a table, each requiring two forks to eat.
- If all pick one fork at the same time, they can enter deadlock.
- **Solution**: Use semaphores to manage fork access.

Advantages and Disadvantages of Process Synchronization

Advantages

- Ensures data consistency and integrity.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- Prevents race conditions and data corruption.
- Facilitates efficient use of shared resources.

Disadvantages

- Adds system overhead.
- May degrade performance due to synchronization mechanisms.
- Increases system complexity.
- Can lead to deadlocks if not implemented properly.

Process synchronization is essential in modern operating systems to ensure efficient and correct execution of multiple processes while avoiding race conditions, deadlocks, and inconsistencies.

Critical-Section Problem

Critical Section in Synchronization

A critical section is a part of a program where shared resources like memory or files are accessed by multiple processes or threads. To avoid issues like data inconsistency or race conditions, synchronization techniques ensure that only one process or thread uses the critical section at a time.

- The critical section contains shared variables or resources that need to be synchronized to maintain the consistency of data variables.
- In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically, such as accessing a resource (file, input/output port, global data, etc.). In concurrent programming, if one process tries to change the value of shared data at the same time as another thread tries to read the value (i.e., data race across threads), the result is unpredictable. The access to such shared variables (shared memory, shared files, shared port, etc.) is to be synchronized.
- Few programming languages have built-in support for synchronization. It is critical to understand the importance of race conditions while writing kernel-mode programming (a device driver, kernel thread, etc.), since the programmer can directly access and modify kernel data structures.

Properties of Critical Section

1. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Approaches to Handle Critical Sections

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

1. **Preemptive Kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
2. **Non-preemptive Kernels:** A non-preemptive kernel does not allow a process running in kernel mode to be preempted. A kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. A non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Critical Section Problem

The use of critical sections in a program can cause several issues, including:

- **Deadlock:** When two or more threads or processes wait for each other to release a critical section, it can result in a deadlock situation in which none of the threads or processes can move. Deadlocks can be difficult to detect and resolve, and they can significantly impact a program's performance and reliability.
- **Starvation:** When a thread or process is repeatedly prevented from entering a critical section, it can result in starvation, in which the thread or process is unable to progress. This can happen if the critical section is held for an unusually long period or if a high-priority thread or process is always given priority when entering the critical section.
- **Overhead:** When using critical sections, threads or processes must acquire and release locks or semaphores, which can take time and resources. This may reduce the program's overall performance.

Critical Section Implementation

Critical Section Implementation

Example Pseudo-code:

```
do{
   flag=1;
   while(flag); // (entry section)
       // critical section
   if (!flag)
       // remainder section
} while(true);
```

Solution to the Critical Section Problem

A simple solution to the critical section can be implemented as shown below:

```
acquireLock();
Process Critical Section;
releaseLock();
```

A thread must acquire a lock before executing a critical section. The lock can be acquired by only one thread. Various methods can be used to implement locks, such as mutexes, semaphores, and monitors.

Strategies for Avoiding Problems

- **Using timeouts** to prevent deadlocks.
- **Priority inheritance** to prevent priority inversion and starvation.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Padasalai

- **Optimizing lock implementations** to reduce overhead.

Real-World Examples of Critical Sections

- **Database Management Systems:** Transactions often require locking mechanisms to ensure data consistency when multiple users access the same records.
- **Web Servers:** Multiple client requests accessing shared resources (e.g., caches, log files) require synchronization.
- **Operating Systems:** Synchronization mechanisms are used in kernel operations, memory management, and process scheduling.

Impact on Scalability

The use of critical sections can impact the scalability of a program, particularly in distributed systems where multiple nodes access shared resources. If not properly managed, frequent access to critical sections can create bottlenecks, reducing the system's overall performance.

Synchronization Mechanisms

1. **Semaphores:** Used to indicate whether a resource is available or not.
2. **Mutexes:** Provide mutual exclusion to shared resources.
3. **Monitors:** High-level synchronization construct that combines mutual exclusion and condition synchronization.

Advantages of Critical Section in Process Synchronization

Padasalai

1. **Prevents race conditions:** Ensures consistency of shared data.
2. **Provides mutual exclusion:** Prevents multiple processes from accessing the same resource simultaneously.
3. **Reduces CPU utilization:** Allows processes to wait without wasting CPU cycles.
4. **Simplifies synchronization:** Ensures only one process accesses a resource at a time.

Disadvantages of Critical Section in Process Synchronization

1. **Overhead:** Synchronization mechanisms introduce additional processing time.
2. **Deadlocks:** Poorly implemented critical sections can lead to deadlocks.
3. **Limits parallelism:** Large or frequently executed critical sections can reduce concurrency.
4. **Contention:** Multiple processes frequently accessing the same critical section can reduce performance.

Important Points to Remember

1. Understanding the concept of the critical section and its role in synchronization.
2. Familiarity with synchronization mechanisms like semaphores, mutexes, and monitors.
3. Knowledge of common synchronization problems such as race conditions, deadlocks, and livelocks.
4. Best practices for designing and implementing critical sections efficiently.
5. Awareness of the impact of critical sections on system performance and scalability.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Proper use of critical sections and process synchronization mechanisms is essential in concurrent programming to ensure synchronization of shared resources and to prevent race conditions, deadlocks, and other synchronization-related issues.

Peterson's Algorithm in Process Synchronization

Peterson's Algorithm is a classic solution to the critical section problem in process synchronization. It ensures **mutual exclusion**, meaning only one process can access the critical section at a time, thus avoiding race conditions. The algorithm uses two shared variables to manage the turn-taking mechanism between two processes, ensuring both follow a fair order of execution. It is simple yet effective for solving synchronization issues in two-process scenarios.

In this article, we will explore **Peterson's Algorithm, its working principle, and practical examples** to understand its significance in process synchronization.

What is Peterson's Algorithm?

Peterson's Algorithm is a well-known solution for ensuring **mutual exclusion** in process synchronization. It is designed to manage access to shared resources between two processes, preventing conflicts or data corruption. The algorithm ensures that only one process can enter the critical section at any given time while the other process waits its turn.

Peterson's Algorithm relies on **two shared variables**:

1. **A turn variable** – Indicates whose turn it is to access the critical section.
2. **A flag array** – Indicates whether a process intends to enter the critical section.

This method is often used in scenarios where **two processes need to share resources or data without interfering with each other**. It is simple, easy to understand, and serves as a foundational concept in **process synchronization and concurrent programming**.

Peterson's Algorithm Explanation

Peterson's Algorithm ensures that two processes do not enter the critical section simultaneously by using two key components:

- **The turn variable** – An integer that indicates **which process gets priority** in accessing the critical section.
- **The flag array** – A Boolean array where each entry corresponds to a process, indicating whether that process **wants to enter** the critical section.

Step-by-Step Working of Peterson's Algorithm:

1. **Initial Setup:**
   - Both processes set their respective flag values to **false**, meaning neither wants to enter the critical section initially.
   - The **turn variable** is set to either process **0 or 1**, indicating whose turn it is to enter.
2. **Intention to Enter:**

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- When a process wants to enter the critical section, it **sets its flag value to true**, signaling its intent.

3. **Set the Turn:**
    - The process then **sets the turn variable to its own ID**, indicating that it is ready to enter the critical section.
4. **Waiting Loop (Busy Waiting):**
    - Before entering, the process continuously **checks the flag of the other process and the turn variable**:
        - If the **other process also wants to enter** (i.e., flag[1 - processID] == true), and
        - If it is currently the **other process's turn** (i.e., turn == 1 - processID),
        - The process **waits**, allowing the other process to proceed first.
    - This loop ensures that only **one process enters the critical section at a time**, effectively preventing a race condition.
5. **Critical Section Execution:**
    - Once the process successfully exits the waiting loop, it enters the **critical section**, where it can safely access or modify the shared resource **without interference** from the other process.
6. **Exiting the Critical Section:**
    - After completing its task in the critical section, the process resets its **flag to false**, signaling that it no longer needs access.
    - This allows the **other process to proceed**, ensuring fair alternation.

Properties of Peterson's Algorithm

Peterson's Algorithm satisfies three crucial properties of process synchronization:

1. **Mutual Exclusion:**
    - Ensures that **only one process** accesses the critical section at any given time.
2. **Progress:**
    - If no process is in the critical section, and **one or more processes want to enter**, one of them is **guaranteed to proceed**.
3. **Bounded Waiting:**
    - Ensures that **no process waits indefinitely**, as access is granted in a fair order.

Limitations of Peterson's Algorithm

While Peterson's Algorithm is an elegant solution for two-process synchronization, it has some **limitations**:

- **Only works for two processes** – The algorithm is not easily scalable for multiple processes.
- **Relies on busy waiting** – The **waiting loop** continuously checks conditions, leading to CPU wastage.
- **Not ideal for modern multi-core processors** – Due to optimizations like instruction reordering and cache coherence mechanisms, Peterson's Algorithm may **not work reliably** on modern architectures **without additional memory barriers**.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Example of Peterson's Algorithm

Peterson's solution is often used as a simple example of mutual exclusion in concurrent programming. Here are a few scenarios where it can be applied:

- **Accessing a shared printer**: Peterson's solution ensures that only one process can access the printer at a time when two processes are trying to print documents.
- **Reading and writing to a shared file**: It can be used when two processes need to read from and write to the same file, preventing concurrent access issues.
- **Competing for a shared resource**: When two processes are competing for a limited resource, such as a network connection or critical hardware, Peterson's solution ensures mutual exclusion to avoid conflicts.

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 2; // Number of threads (producer and consumer)

std::vector<bool> flag(N, false); // Flags to indicate readiness
int turn = 0; // Variable to indicate turn

void producer(int j) {
    do {
        flag[j] = true; // Producer j is ready to produce
        turn = 1 - j;   // Allow consumer to consume
        while (flag[1 - j] && turn == 1 - j) {
            // Wait for consumer to finish
            // Producer waits if consumer is ready and it's consumer's turn
        }

        // Critical Section: Producer produces an item and puts it into the buffer

        flag[j] = false; // Producer is out of the critical section

        // Remainder Section: Additional actions after critical section
    } while (true); // Continue indefinitely
}

void consumer(int i) {
    do {
        flag[i] = true; // Consumer i is ready to consume
        turn = i;       // Allow producer to produce
        while (flag[1 - i] && turn == i) {
            // Wait for producer to finish
            // Consumer waits if producer is ready and it's producer's turn
        }

        // Critical Section: Consumer consumes an item from the buffer

        flag[i] = false; // Consumer is out of the critical section

        // Remainder Section: Additional actions after critical section
    } while (true); // Continue indefinitely
}

int main() {
    std::thread producerThread(producer, 0); // Create producer thread
    std::thread consumerThread(consumer, 1); // Create consumer thread

    producerThread.join(); // Wait for producer thread to finish
    consumerThread.join(); // Wait for consumer thread to finish

    return 0;
}
```

Padasalai

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Explanation of the above code:

This code shows a simple version of Peterson's Algorithm for synchronizing two processes: a producer and a consumer. The goal is to make sure that both processes don't interfere with each other while accessing a shared resource, which is a buffer in this case. The producer creates items, and the consumer takes them.

Here's a detailed explanation of the code:

Producer (j)

Producer is ready to produce:

flag[j] = true;: This indicates that the producer (process j) is ready to produce an item. The flag array holds the intentions of both processes (producer and consumer). When the producer sets its flag to true, it means it's willing to access the shared resource (the buffer).

Set the turn for consumer:

turn = i;: The turn variable is used to decide whose turn it is to enter the critical section (where the shared resource is accessed). Here, the producer is setting the turn to the consumer (process i), meaning the producer is willing to wait if the consumer wants to consume an item.

Wait until the consumer is done:

while (flag[i] == true && turn == i) : This is the crucial part of the algorithm. The producer checks if the consumer has indicated that it is ready (flag[i] == true) and whether it's the consumer's turn (turn == i). If both conditions are true the producer must wait (it cannot enter the critical section). This ensures that the consumer gets a chance to consume before the producer produces a new item.

Producer produces an item:

Once the condition in the while loop is no longer true (meaning either the consumer is not ready or it is the producer's turn), the producer can safely enter the critical section, produce an item, and place it into the buffer.

Exit the critical section:

flag[j] = false;: After the producer finishes its work in the critical section, it sets its flag to false, indicating that it is done and no longer wants to produce. This allows the consumer to have the opportunity to consume the item next.

Consumer (i)

Consumer is ready to consume:

flag[i] = true;: The consumer sets its flag to true, indicating that it is ready to consume an item from the buffer.

Set the turn for producer:

turn = j;: The consumer sets the turn variable to the producer's process ID (j). This indicates that it is the producer's turn to produce if the consumer is done consuming.

Wait until the producer is done:

while (flag[j] == true && turn == j) : This is similar to the producer's waiting condition. The consumer checks if the producer is ready to produce (flag[j] == true) and whether it's the producer's turn (turn == j). If both conditions are true, the consumer must wait, allowing the producer to produce an item.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Consumer consumes an item:

Once the while loop exits (meaning the consumer is allowed to consume), the consumer enters the critical section and consumes an item from the buffer.

Exit the critical section:

flag[i] = false;: After consuming the item, the consumer sets its flag to false, indicating it no longer wants to consume. This allows the producer to have the opportunity to produce the next item.

Advantages of Peterson's Algorithm

- Ensures **mutual exclusion**, preventing conflicts over shared resources.
- **Fairness**: Both processes get a chance to execute.
- **Software-based solution**, independent of hardware.
- Avoids **deadlocks**, ensuring progress.

Disadvantages of Peterson's Algorithm

- **Busy waiting**: A process continuously checks conditions, leading to CPU inefficiency.
- **Not suitable for multiprocessor systems**, as memory updates may not be immediately visible to all processors.

Peterson's Algorithm remains a fundamental concept in understanding process synchronization, particularly in single-processor environments.

Padasalai

Semaphores in Process Synchronization

Semaphores are a tool used in operating systems to help manage how different processes (or programs) share resources, like memory or data, without causing conflicts. A semaphore is a special kind of synchronization primitive that ensures safe access to shared resources in a concurrent system.

Semaphores are particularly useful in preventing race conditions, which occur when multiple processes access shared data simultaneously, leading to unpredictable behavior. They help enforce critical sections—regions of code that must be executed by only one process at a time—by restricting access based on a controlled counter mechanism.

What is a Semaphore?

A semaphore is a synchronization tool used in concurrent programming to manage access to shared resources. It is a lock-based mechanism designed to achieve process synchronization, built on top of basic locking techniques.

Semaphores use a counter to control access, allowing synchronization for multiple instances of a resource. Processes can attempt to access one instance, and if it is not available, they can try other instances. Unlike basic locks, which allow only one process to access one instance of a resource, semaphores can handle more complex synchronization scenarios, involving multiple processes or threads.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

They help prevent problems like race conditions by controlling when and how processes access shared data.

Operations on Semaphores

The process of using semaphores involves two fundamental operations:

- **Wait (P operation):** This operation decrements the value of the semaphore. If the value is zero, the process is blocked until another process increments it.
- **Signal (V operation):** This operation increments the value of the semaphore. If there are processes waiting, one of them is unblocked.

Behavior of Semaphores

1. When a process performs a **wait (P) operation**, it checks whether the semaphore's value is greater than 0:
   - If **yes**, it decrements the value and continues execution.
   - If **no (value is 0)**, the process is blocked until another process performs a signal operation.
2. When a process performs a **signal (V) operation**, it:
   - Activates a waiting process if one exists.
   - Otherwise, increments the semaphore value by 1.

Due to these semantics, semaphores are also known as **counting semaphores**. The initial value of a semaphore determines how many processes can get past the wait operation.

Padasalai

Types of Semaphores

Semaphores are classified into two types:

1. Binary Semaphore

- Also known as a **mutex lock** (mutual exclusion).
- Can have only two values: **0 (locked)** and **1 (unlocked)**.
- Used to implement solutions for **critical section problems** with multiple processes and a single resource.

2. Counting Semaphore

- Used when multiple instances of a resource are available.
- The semaphore value is initialized to the number of available resources.
- It allows multiple processes to access resources **up to a specified limit**.

Working of a Semaphore

A semaphore maintains a counter that controls access to a specific resource, ensuring that **no more than the allowed number of processes access the resource simultaneously**.

- **Wait (P operation):**

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- If the semaphore value is greater than 0, the process is allowed to proceed, and the counter is decremented.
- If the value is 0, the process is blocked until another process releases the resource.

- **Signal (V operation):**
    - The semaphore value is incremented, potentially unblocking a waiting process.

Example: Mutual Exclusion using a Binary Semaphore

Consider two processes **P1** and **P2**, and a semaphore **S** initialized to 1.

1. **P1 enters the critical section:** The value of **S** becomes 0.
2. **P2 wants to enter the critical section:** Since **S = 0**, it must wait.
3. **P1 finishes and performs a signal operation:** The value of **S** becomes 1.
4. **P2 can now enter the critical section.**

This ensures **mutual exclusion** and prevents conflicts.

Example: Resource Allocation using a Counting Semaphore

Consider a system with **4 identical printers** and a semaphore initialized to **4**.

1. When a process needs a printer, it performs a **wait operation**, reducing the count.
2. When done, it performs a **signal operation**, increasing the count.
3. If all printers are in use (**S = 0**), a new process must wait until a printer is released.

Limitations of Semaphores

Padasalai

Despite their usefulness, semaphores have certain drawbacks:

1. **Priority Inversions:** A higher-priority process may be blocked by a lower-priority process holding a semaphore.
2. **Deadlocks:** If a process waits for a signal from another process that never arrives, it can result in a **deadlock**.
3. **Busy Waiting:** Traditional semaphores use **spinlocks**, where a process continuously checks for availability, wasting CPU cycles.
4. **Complexity:** Managing semaphores properly requires careful implementation to avoid synchronization bugs.

To avoid busy waiting, a more efficient implementation uses **waiting queues** instead of spinlocks. Here:

- A process that cannot proceed is **blocked** instead of busy waiting.
- When a signal operation occurs, a blocked process is **woken up** and allowed to proceed.

Uses of Semaphores

Semaphores are used for:

1. **Mutual Exclusion:** Ensures only one process accesses a critical section at a time.
2. **Process Synchronization:** Coordinates execution order among processes.
3. **Resource Management:** Controls access to limited resources like printers, databases, and files.
4. **Avoiding Deadlocks:** Regulates access to prevent circular wait conditions.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

5. **Solving Classical Synchronization Problems:** Such as producer-consumer, dining philosophers, and readers-writers problems.

Advantages of Semaphores

- **Simple and effective** mechanism for synchronization.
- **Flexible** in controlling access to shared resources.
- **Prevents race conditions** by restricting concurrent access.
- **Can be used for both mutual exclusion and coordination.**

Disadvantages of Semaphores

- **Deadlocks** can occur if semaphores are not managed properly.
- **Priority inversion** can degrade performance.
- **Difficult to debug and maintain** in large systems.
- **Overhead** in managing wait and signal operations.

Classical Synchronization Problems using Semaphores

1. Producer-Consumer Problem

- Producers generate items and place them in a buffer.
- Consumers remove items from the buffer.
- **Semaphores Used:**
  - `empty` (counts available slots in the buffer).
  - `full` (counts occupied slots).
  - `mutex` (ensures mutual exclusion).

Padasalai

2. Traffic Light Control

- Semaphores regulate green, yellow, and red signals.
- Only one direction has a green light at a time.

3. Bank Transaction Processing

- Only one transaction can access an account at a time using a semaphore.

4. Print Queue Management

- Only one print job can access a printer at a time.

5. Railway Track Management

- Ensures only one train enters a track at a time.

6. Dining Philosopher's Problem

- Philosophers must acquire **two forks (semaphores)** to eat.
- Prevents **deadlock** and **starvation**.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

7. Reader-Writer Problem

- Multiple readers can read simultaneously.
- Writers require **exclusive** access.

5.Threads and Multithreading – Detailed Notes

1. Threads

A **thread** is the smallest unit of execution within a process. It is a lightweight process that runs within the same address space as other threads in the process.

Characteristics of Threads:

- A process can have **multiple threads**, each with its own execution path.
- Threads **share memory and resources** (such as files, variables, and open sockets).
- They provide **faster context switching** compared to processes.

Types of Threads:

1. **User Threads:**
    - Managed by user-level libraries, without kernel intervention.
    - Faster as switching between threads does not require kernel mode transition.
    - Example: POSIX Threads (Pthreads).
2. **Kernel Threads:**
    - Managed directly by the operating system.
    - More powerful but slower than user threads.
    - Example: Windows and Linux kernel threads.

Padasalai

---

2. Multicore Programming

With the rise of **multicore processors**, single-threaded applications **cannot fully utilize** processing power. **Multicore programming** enables software to use multiple cores efficiently.

Parallelism vs. Concurrency:

- **Parallelism:**
    - Tasks execute **simultaneously** on multiple cores.
    - Requires multiple threads or processes.
- **Concurrency:**
    - Tasks appear to be executed simultaneously, but the CPU **switches** between them.

Benefits of Multicore Programming:

✅ Improved **performance** due to multiple cores executing tasks.
✅ **Better resource utilization** by distributing tasks.
✅ **Reduced power consumption per task** due to efficient execution.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

3. Multithreading Models

A **multithreading model** defines how **user threads** are mapped to **kernel threads** for execution.

Types of Multithreading Models:

1. **Many-to-One Model:**
    - Multiple user threads map to **a single kernel thread**.
    - **Disadvantage:** No true parallelism, as only one thread executes at a time.
    - Used in some **old operating systems**.
2. **One-to-One Model:**
    - Each **user thread** is mapped to a **kernel thread**.
    - **Advantage:** Better concurrency and responsiveness.
    - **Disadvantage:** High overhead due to thread creation.
    - Example: Windows Threads, Linux.
3. **Many-to-Many Model:**
    - Many user threads dynamically map to a **smaller or equal number** of kernel threads.
    - **Advantage:** Combines efficiency with flexibility.
    - **Used in modern operating systems** such as Solaris.

---

4. Thread Libraries

Thread libraries provide **APIs** to create, manage, and synchronize threads.

Examples of Thread Libraries:

Padasalai

1. **POSIX Threads (Pthreads):** Used in Unix/Linux.
2. **Windows Threads:** Managed using the Windows API.
3. **Java Threads:** Uses the **Java concurrency package** for multithreading.

---

5. Implicit Threading

Implicit threading allows the **OS or runtime environment** to **handle thread management automatically** instead of manual thread creation.

Approaches to Implicit Threading:

1. **Thread Pools:**
    - Pre-created **threads** wait for tasks.
    - Efficient as it avoids the overhead of creating/destroying threads repeatedly.
2. **Fork-Join Model:**
    - A **task is divided** (forked) into smaller subtasks.
    - Subtasks **execute in parallel** and then combine (join) the results.
    - Used in **Java Fork-Join Framework**.
3. **OpenMP:**
    - A compiler-based approach for parallelism in **C, C++, and Fortran**.
    - Uses **directives** to tell the compiler how to parallelize tasks.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

6. Threading Issues

1. Race Conditions:

- Occurs when multiple threads **access shared data** unpredictably.
- Solution: Use **locks, mutexes, or atomic operations**.

2. Deadlocks:

- Occurs when two or more threads **are waiting for each other** to release resources, causing a permanent block.
- Solution: Implement **deadlock prevention techniques** like resource ordering or Banker's Algorithm.

3. Livelock:

- Threads keep **changing state without making progress**.
- Solution: Implement **randomized backoff** strategies.

4. Starvation:

- A thread **never gets CPU time** because other threads keep executing.
- Solution: Implement **fair scheduling algorithms**.

5. Context Switching Overhead:

- **Switching between threads** frequently can reduce performance.
- Solution: Minimize context switches by **using efficient scheduling strategies**.

Padasalai

CPU Scheduling in Operating Systems

Introduction

CPU scheduling is a process used by the operating system to decide which task or process gets to use the CPU at a particular time. Since a CPU can only handle one task at a time, scheduling ensures efficient execution of multiple processes. The primary goals of CPU scheduling are:

- **Maximizing CPU Utilization:** Keeping the CPU busy as much as possible.
- **Minimizing Response and Waiting Time:** Ensuring that processes spend minimal time waiting in the queue.

Need for a CPU Scheduling Algorithm

CPU scheduling determines which process will use the CPU while another process is suspended. The main function of CPU scheduling is to ensure that whenever the CPU is idle, the OS selects a process from the ready queue for execution.

In multiprogramming environments, if the long-term scheduler selects multiple I/O-bound processes, the CPU may remain idle most of the time. An effective CPU scheduling algorithm improves resource utilization and system performance.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Terminologies Used in CPU Scheduling

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Burst Time:** The amount of time required by a process for CPU execution.
- **Turnaround Time (TAT):** The total time taken by a process from arrival to completion.
  - *Formula:* Turnaround Time = Completion Time – Arrival Time
- **Waiting Time (WT):** The total time a process spends waiting in the ready queue.
  - *Formula:* Waiting Time = Turnaround Time – Burst Time
- **Response Time (RT):** The time taken from process submission to the first response generated.

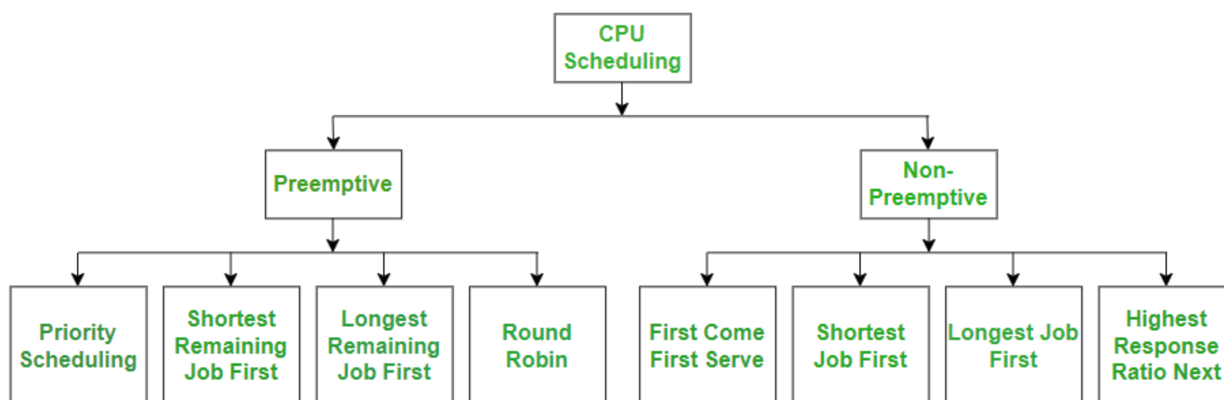
Factors to Consider When Designing a CPU Scheduling Algorithm

Different CPU scheduling algorithms vary based on multiple factors. The selection of an algorithm depends on the following considerations:

- **CPU Utilization:** The algorithm should keep the CPU as busy as possible. Theoretically, CPU utilization can range from 0% to 100%, but in real systems, it typically varies from 40% to 90% depending on the load.
- **Throughput:** The number of processes executed per unit time. It depends on the process length and scheduling algorithm.
- **Turnaround Time:** The total time a process spends from arrival to completion, including waiting, execution, and I/O operations.
- **Waiting Time:** The amount of time a process spends in the ready queue before execution.
- **Response Time:** The time taken from process arrival to the first response generated, which is critical in interactive systems.

Padasalai

Types of CPU Scheduling Algorithms



There are two main types of CPU scheduling methods:

- **Preemptive Scheduling:** The CPU can be taken away from a running process before it completes execution. Examples include:
  - Shortest Remaining Time First (SRTF)
  - Round Robin (RR)
  - Priority Scheduling (Preemptive)

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **Non-Preemptive Scheduling:** Once a process is given the CPU, it cannot be taken away until it completes execution or enters a waiting state. Examples include:
  - First Come, First Serve (FCFS)
  - Shortest Job First (SJF)
  - Priority Scheduling (Non-Preemptive)

CPU Scheduling Algorithms

Here is an overview of major CPU scheduling algorithms:

1. **FCFS (First Come, First Serve):**
   - Allocation: Based on process arrival time.
   - Complexity: Simple and easy to implement.
   - Average Waiting Time (AWT): High, as processes must wait for previous processes to finish.
   - Preemption: No.
   - Starvation: No.
   - Performance: Poor for short processes; suffers from the "convoy effect."
2. **SJF (Shortest Job First):**
   - Allocation: Based on the shortest CPU burst time.
   - Complexity: More complex than FCFS.
   - AWT: Lower than FCFS.
   - Preemption: No.
   - Starvation: Yes, long processes may never get CPU time.
   - Performance: Minimum average waiting time but can lead to starvation.
3. **SRTF (Shortest Remaining Time First):**
   - Allocation: Similar to SJF but preemptive.
   - Complexity: Higher than FCFS.
   - AWT: Depends on arrival time and process size.
   - Preemption: Yes.
   - Starvation: Yes, as shorter processes keep getting preference.
   - Performance: Gives priority to short jobs, improving responsiveness.
4. **Round Robin (RR):**
   - Allocation: Processes execute in a cyclic order with a fixed time quantum.
   - Complexity: Depends on the time quantum size.
   - AWT: Larger than SJF and Priority Scheduling.
   - Preemption: Yes.
   - Starvation: No.
   - Performance: Ensures fair execution but increases context switching overhead.
5. **Priority Scheduling:**
   - Allocation: CPU is assigned based on process priority.
   - Complexity: Less complex than SJF.
   - AWT: Lower than FCFS.
   - Preemption: Yes (Preemptive) / No (Non-Preemptive).
   - Starvation: Yes (higher priority processes dominate lower ones).
   - Performance: Good, but starvation can be an issue.
6. **HRRN (Highest Response Ratio Next):**
   - Allocation: Based on response ratio.
   - Complexity: Higher than FCFS but balances waiting and execution.
   - AWT: Smaller than FCFS and Priority Scheduling.
   - Preemption: No.
   - Starvation: No.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

   - Performance: Balances short and long processes effectively.
7. **Multilevel Queue Scheduling (MLQ):**
   - Allocation: Processes assigned to different priority queues.
   - Complexity: More complex than priority scheduling.
   - AWT: Lower than FCFS.
   - Preemption: No.
   - Starvation: Yes.
   - Performance: Efficient but prone to starvation.
8. **Multilevel Feedback Queue Scheduling (MFLQ):**
   - Allocation: Processes move between multiple queues based on execution behavior.
   - Complexity: Most complex but highly adaptable.
   - AWT: Lower than all other scheduling types in many cases.
   - Preemption: Yes.
   - Starvation: No.
   - Performance: High, efficiently handles all types of processes.

Thread Scheduling

Thread scheduling in Java determines which thread should execute or get a resource in the operating system. Thread scheduling involves two boundary levels:

1. **Scheduling of User-Level Threads (ULT) to Kernel-Level Threads (KLT)** via Lightweight Process (LWP) by the application developer.
2. **Scheduling of Kernel-Level Threads by the System Scheduler** to perform various OS functions.

Lightweight Process (LWP)

Padasalai

Lightweight processes act as an interface for user-level threads to access physical CPU resources. The thread library schedules which thread of a process runs on which LWP and for how long. The number of LWPs created depends on the application type:

- **I/O-bound applications:** The number of LWPs is equal to the number of ULTs because when an LWP is blocked on I/O, a new LWP is created and scheduled for another ULT.
- **CPU-bound applications:** The number of LWPs depends on the application's requirements.

Each LWP is attached to a separate kernel-level thread.

Thread Scheduling Controls

Thread scheduling involves two controls: **Contention Scope** and **Allocation Domain**.

Contention Scope

Contention refers to the competition among user-level threads for kernel resources. The extent of contention is classified into:

1. **Process Contention Scope (PCS):** Contention occurs among threads within the same process. The thread library schedules high-priority PCS threads via available LWPs.
2. **System Contention Scope (SCS):** Contention occurs among all system threads. Each SCS thread is associated with an LWP and scheduled by the system scheduler.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

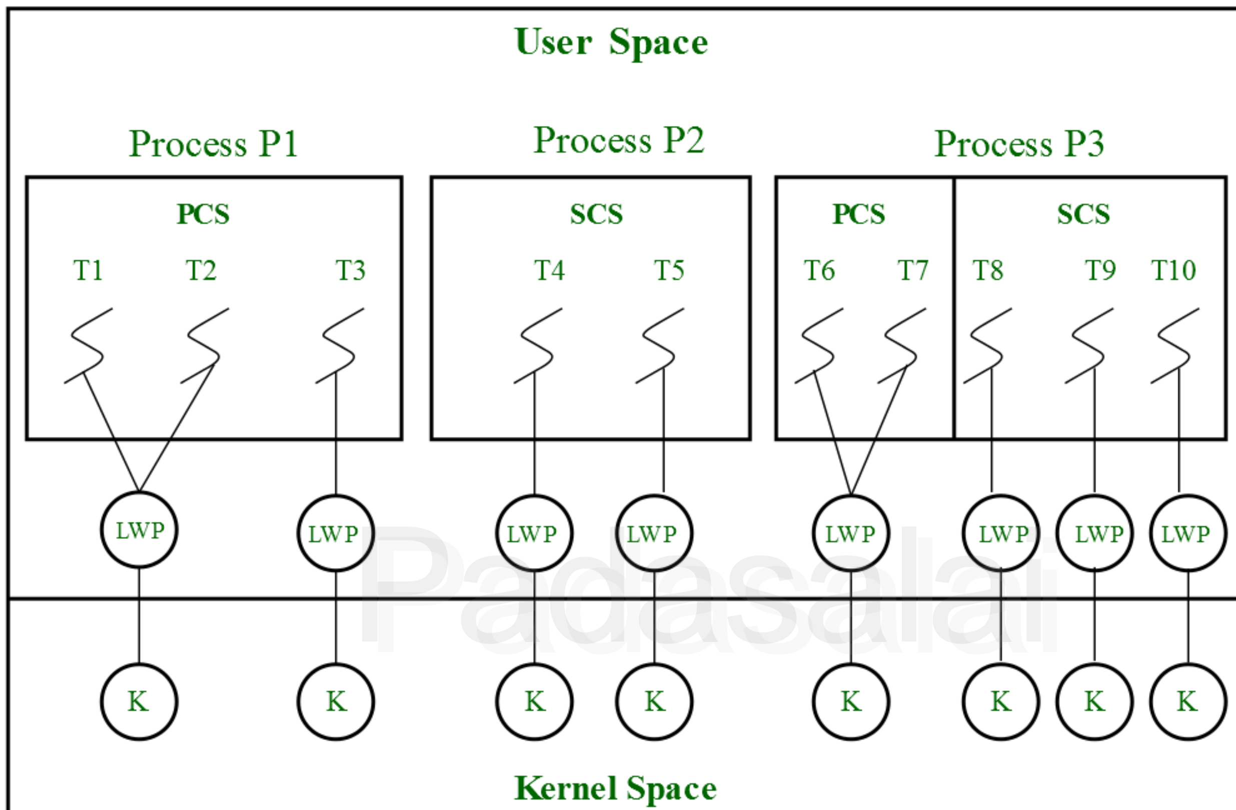
The **POSIX Pthread library** provides a function to define contention scope:

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope)
```

- `PTHREAD_SCOPE_SYSTEM` (for SCS)
- `PTHREAD_SCOPE_PROCESS` (for PCS)

If the scope is unsupported, the function returns `ENOTSUP`.

Allocation Domain



An allocation domain consists of one or more cores competing for resources. A ULT can be part of multiple allocation domains. In a multicore system, scheduling depends on contention scope, scheduling policy, and priority.

Example Scenario

Consider an OS with **3 processes (P1, P2, P3)** and **10 user-level threads (T1 to T10)** sharing a single allocation domain.

- **Process P1:** PCS threads (T1, T2, T3) compete internally. T1 and T2 share an LWP, while T3 has a separate LWP. Preemptive priority scheduling applies within P1.
- **Process P2:** SCS threads (T4, T5) compete with processes P1 and P3 (T8, T9, T10). System scheduler handles scheduling.
- **Process P3:** Combination of PCS (T6, T7) and SCS (T8, T9, T10) threads. CPU resources are split (e.g., 50% to P3, with 25% allocated for PCS and 25% for SCS threads).

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Kernel-Level Threads (KLT) Calculation

- **Number of SCS threads:** 5
- **Number of LWP for PCS:** 3
- **Number of LWP for SCS:** 5
- **Total LWPs:** 8 (5+3)
- **Total Kernel-Level Threads:** 8

Advantages of PCS over SCS

1. **Efficiency:** Context switching, synchronization, and scheduling occur within userspace, reducing system calls.
2. **Lower Overhead:** PCS threads are cheaper than SCS threads.
3. **Resource Sharing:** PCS threads share LWPs, while SCS threads require a separate LWP for each.
4. **Kernel Complexity:** More SCS threads increase scheduling complexity, requiring SCS threads to be fewer than PCS threads.
5. **Scalability:** Systems with multiple allocation domains require careful handling of SCS threads to avoid synchronization issues.
6. **Performance Optimization:** PCS threads allow greater control over scheduling policies, improving performance in high-demand applications.
7. **Flexibility:** Developers can optimize resource utilization by configuring PCS threads based on specific application needs.

Second Boundary: CPU Scheduling by System Scheduler

The system scheduler treats each kernel-level thread as a separate process and allocates kernel resources accordingly.

Padasalai

This detailed explanation ensures clarity on thread scheduling while retaining all critical information.

Multiple-Processor Scheduling in Operating System

In multiple-processor scheduling, multiple CPUs are available, making load sharing possible. However, multiple-processor scheduling is more complex compared to single-processor scheduling. When the processors are identical (HOMOGENEOUS) in terms of functionality, any available processor can be used to run any process in the queue.

What is CPU Scheduling?

CPU scheduling is the mechanism through which an operating system selects tasks or processes for execution by the CPU at any instant in time. The primary goal is to keep the CPU busy by efficiently assigning processing time to various processes, optimizing system performance. Several scheduling algorithms, such as Round Robin or Priority Scheduling, govern task scheduling.

What is Multiple-Processor Scheduling?

In systems with more than one processor, multiple-processor scheduling allocates tasks to multiple CPUs. This results in higher throughput since multiple tasks can be executed concurrently across different processors. Scheduling must determine which CPU will handle a specific task and balance loads efficiently.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Approaches to Multiple-Processor Scheduling

1. **Asymmetric Multiprocessing (AMP):**
   - A single processor makes all scheduling decisions and handles I/O processing, while other processors execute user code.
   - This reduces the need for data sharing and simplifies scheduling.
2. **Symmetric Multiprocessing (SMP):**
   - Each processor is self-scheduling.
   - Processes can either be placed in a common ready queue or each processor may maintain its own private queue.
   - The scheduler for each processor selects a process from the ready queue for execution.

1. Processor Affinity

Processor Affinity refers to a process's preference for the processor on which it is currently running. When a process runs on a particular processor, its data gets cached, improving performance. If the process moves to another processor, cache memory must be invalidated and repopulated, leading to performance overhead. SMP systems try to minimize process migration to avoid this issue.

Types of Processor Affinity:

- **Soft Affinity:** The OS attempts to keep a process running on the same processor but does not guarantee it.
- **Hard Affinity:** The OS allows a process to specify a subset of processors on which it can run. Some systems, like Linux, implement soft affinity but also provide system calls like `sched_setaffinity()` for hard affinity.

Padasalai

2. Load Balancing

Load balancing ensures that the workload is evenly distributed across all processors in an SMP system. It is essential when each processor has its own private queue of ready processes. Otherwise, an idle processor will automatically pick a process from a common run queue.

Load Balancing Approaches:

- **Push Migration:** A dedicated task monitors processor loads and redistributes tasks from overloaded to idle or less-busy processors.
- **Pull Migration:** An idle processor pulls a waiting task from a busy processor for execution.

3. Multicore Processors

In multicore processors, multiple processor cores reside on the same physical chip. Each core has its own register set and appears to the OS as an independent processor. SMP systems using multicore processors are faster and consume less power than systems with separate physical chips. However, multicore processors introduce additional scheduling challenges.

Memory Stall:

When a processor accesses memory, it often spends time waiting for data, known as a **memory stall**. This occurs due to cache misses, where the requested data is not available in the cache. To mitigate this, modern hardware uses multithreaded processor cores.

Types of Multithreading:

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **Coarse-Grained Multithreading:**
    - A thread executes until a long-latency event (e.g., memory stall) occurs.
    - The processor switches to another thread, but the switching cost is high due to pipeline termination.
- **Fine-Grained Multithreading:**
    - Thread switching occurs at instruction cycle boundaries.
    - The processor contains logic to enable fast thread switching, reducing switching overhead.

4. Virtualization and Threading

In virtualized systems, even a single CPU system can simulate multiple processors. Virtualization creates one or more virtual CPUs (vCPUs) per virtual machine (VM), scheduling their execution on physical CPUs.

- The host OS manages VMs, each of which runs a guest OS and applications.
- Virtualized OSs may experience poor scheduling performance since the perceived CPU time differs from actual physical CPU availability.
- Time-sharing systems that expect consistent time slices may be negatively impacted by virtualization delays.
- Time-of-day clocks in VMs can become inaccurate due to differences in CPU cycle allocations.

Real-Time Scheduling

Real-time scheduling ensures that time-critical tasks meet their deadlines, making it essential for systems where timing constraints are crucial, such as embedded systems, medical devices, and industrial automation.

Padasalai

There are two types of real-time scheduling:

1. **Hard Real-Time Scheduling**:
    - Guarantees task completion within strict deadlines.
    - Missing a deadline can lead to catastrophic failure (e.g., airbag deployment in vehicles, medical monitoring systems).
    - Used in safety-critical applications.
2. **Soft Real-Time Scheduling**:
    - Prioritizes meeting deadlines but does not guarantee them.
    - Occasional deadline misses are tolerable but can degrade system performance.
    - Used in multimedia streaming, online transaction processing, and gaming systems.

Common Real-Time Scheduling Algorithms

- **Rate-Monotonic Scheduling (RMS)**:
    - A fixed-priority algorithm where shorter-period tasks get higher priority.
    - Assumes tasks are periodic, independent, and preemptive.
    - Utilizes worst-case response time analysis to determine schedulability.
- **Earliest Deadline First (EDF)**:
    - A dynamic-priority algorithm that assigns priorities based on the closest deadline.
    - More flexible than RMS but requires higher computational overhead.
    - Maximizes CPU utilization but may face challenges under high system load.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Operating System (OS)

Virtualization and Threading

Virtualization allows multiple operating systems to run concurrently on a single physical machine by creating virtual instances of hardware resources. Even on a single CPU system, virtualization can simulate multiple processors, enabling the execution of multiple virtual CPUs (vCPUs).

Virtual Machine (VM) and vCPU Scheduling

- **Host OS Role**: The host OS (or hypervisor) is responsible for managing VMs, allocating resources, and scheduling vCPUs on physical CPUs.
- **Guest OS and Applications**: Each VM runs a guest OS and its own applications, unaware of the underlying virtualized environment.

Challenges of Virtualization in Scheduling

1. **Scheduling Overhead**:
   - Virtualization introduces additional scheduling complexity, as multiple vCPUs compete for limited physical CPU resources.
   - The hypervisor must balance the execution of multiple VMs, leading to potential inefficiencies.
2. **Performance Degradation**:
   - Time-sharing systems that expect consistent time slices may be negatively impacted due to virtualization delays.
   - Guest OSs may perceive CPU availability differently from the actual allocation, leading to unpredictable performance.
3. **Time Synchronization Issues**:
   - Time-of-day clocks in VMs can become inaccurate due to differences in CPU cycle allocations.
   - Variations in CPU scheduling can lead to clock drift, affecting applications that rely on precise timing (e.g., financial transactions, distributed systems).
   - Solutions include using time synchronization protocols like NTP (Network Time Protocol) or hypervisor-assisted clock adjustments.

By understanding real-time scheduling and virtualization effects, system designers can optimize scheduling policies for better performance and reliability.

7.Introduction to Deadlock in Operating System

A deadlock is a situation where a set of processes is blocked because each process is holding a resource and waiting for another resource acquired by some other process.

- Deadlock occurs when two or more processes are unable to proceed because each is waiting for the other to release resources.
- Key concepts include mutual exclusion, resource holding, circular wait, and no preemption.

Example of Deadlock

Consider an example where two trains are coming toward each other on the same track. Since there is only one track, neither train can move once they are in front of each other. This is a practical example of deadlock.

How Does Deadlock Occur in the Operating System?

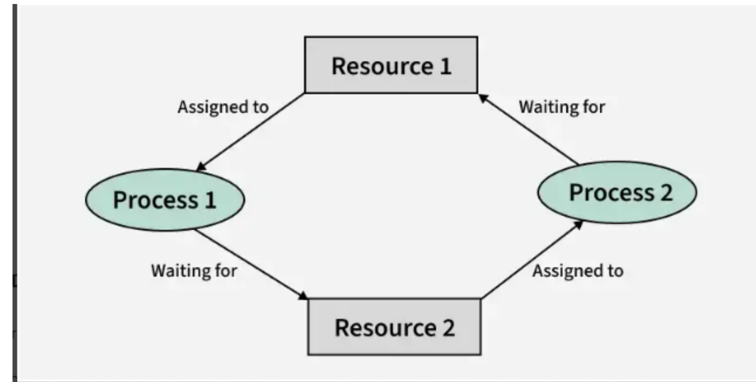
A process in an operating system uses resources in the following way:

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- Requests a resource
- Uses the resource
- Releases the resource

A deadlock occurs when multiple processes hold some resources and wait for resources held by others. For example, Process 1 holds Resource 1 and waits for Resource 2, while Process 2 holds Resource 2 and waits for Resource 1.

Examples of Deadlock



1. The system has two tape drives. P0 and P1 each hold one tape drive and need another.
2. Semaphores A and B are initialized to 1:
   - P0 executes wait(A) and preempts.
   - P1 executes wait(B).
   - P0 then executes wait(B), and P1 executes wait(A), causing deadlock.
3. Assume the system has 200KB available. The following sequence occurs:
   - P0 requests 80KB.
   - P1 requests 70KB.
   - P0 requests another 60KB, while P1 requests 80KB.
   - Deadlock occurs as both wait for additional memory.

Necessary Conditions for Deadlock

A deadlock arises if the following four conditions hold simultaneously:

1. **Mutual Exclusion:** Only one process can use a resource at a time.
2. **Hold and Wait:** A process holds at least one resource while waiting for another.
3. **No Preemption:** A resource cannot be taken from a process unless it releases it.
4. **Circular Wait:** A circular chain of processes exists, each waiting for a resource held by the next.

Methods of Handling Deadlocks

There are three ways to handle deadlocks:

1. **Deadlock Prevention or Avoidance**
2. **Deadlock Detection and Recovery**
3. **Deadlock Ignorance**

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Deadlock Prevention

Deadlock prevention aims to ensure that at least one of the necessary conditions for deadlock is not met:

- **Mutual Exclusion:** Only apply locks to non-shareable resources.
- **Hold and Wait:** Allocate all resources to a process before execution.
- **No Preemption:** If a process requests an unavailable resource, release all held resources.
- **Circular Wait:** Impose an ordering on resource requests.

Deadlock Avoidance

Deadlock avoidance ensures that the system remains in a safe state. The **Banker's Algorithm** is used to avoid deadlocks by pre-checking whether granting a request will lead to an unsafe state.

Deadlock Detection and Recovery

When deadlock prevention or avoidance is not applied, the system must detect and recover from deadlocks:

1. **Deadlock Detection:** Uses algorithms like the Resource Allocation Graph and Banker's Algorithm to detect deadlocks.
2. **Deadlock Recovery:** Involves strategies like:
   - **Manual Intervention**: The operator manually resolves deadlocks.
   - **Automatic Recovery**: The system automatically breaks deadlock cycles.
   - **Process Termination**: Aborting all or some deadlocked processes.
   - **Resource Preemption**: Selectively preempting resources to break deadlocks.

Padasalai

Deadlock Ignorance

Some operating systems, like Windows and UNIX, ignore deadlocks if they are rare. This approach, called the **Ostrich Algorithm**, lets deadlocks occur and resolves them by restarting the system.

Safe State

A **safe state** is one where no deadlock exists and processes can complete execution without waiting indefinitely. If no safe sequence exists, the system is in an unsafe state, potentially leading to deadlock.

Difference between Starvation and Deadlocks

Aspect	Deadlock	Starvation
Definition	A condition where processes are blocked indefinitely, each waiting for a resource held by another.	A condition where a process is denied resources indefinitely despite availability.
Resource Availability	Resources are held by processes involved in the deadlock.	Resources are available but continuously allocated to other processes.
Cause	Circular dependency between processes.	Continuous preference given to other processes.
Resolution	Requires intervention like aborting processes or preempting resources.	Adjusting scheduling policies to ensure fairness.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

www.Padasalai.Net www.TrbTnpsc.com

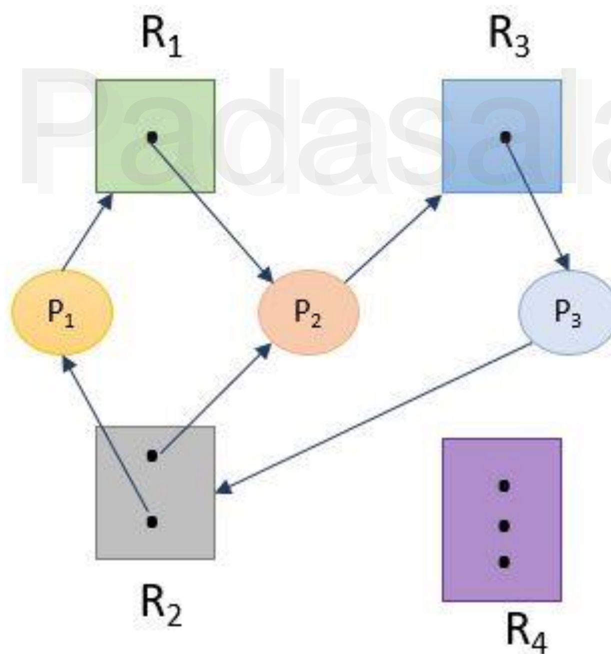
Deadlock Characterization

In an operating system, a **deadlock** occurs when multiple processes are indefinitely stuck, each waiting for a resource held by another, creating a circular dependency. Deadlocks arise when **all four necessary conditions—mutual exclusion, hold and wait, no preemption, and circular wait**—are present simultaneously.

Key Conditions for Deadlock:

1. **Mutual Exclusion**:
   - A resource can be held by only one process at a time.
   - If a process is using a resource, no other process can access it until it is released.
2. **Hold and Wait**:
   - A process holding one or more resources can continue to wait for additional resources held by another process.
3. **No Preemption**:
   - A resource cannot be forcibly taken from a process; it must be released voluntarily.
4. **Circular Wait**:
   - A circular chain of processes exists where each process is waiting for a resource held by the next process in the chain.

Example of Deadlock:



Resource Allocation Graph with Deadlock

Consider two processes, **A** and **B**:

- **Process A** needs access to a **printer** to complete its task.
- **Process B** needs access to a **file** held by A.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **A** holds the file and is waiting for the printer.
- **B** holds the printer and is waiting for the file.
- This circular dependency causes a deadlock, as neither process can proceed.

Deadlock Management Strategies:

1. **Deadlock Prevention:**
   - The system is designed to **ensure at least one of the four necessary conditions never occurs**.
   - Example techniques:
     - Eliminating hold and wait by requiring a process to request all resources at once.
     - Breaking circular wait by enforcing an ordering on resource allocation.
2. **Deadlock Avoidance:**
   - Algorithms monitor resource allocation to **ensure the system never enters an unsafe state**.
   - Example: **Banker's Algorithm**, which ensures processes only proceed if the system remains in a safe state.
3. **Deadlock Detection and Recovery:**
   - The system **periodically checks for deadlocks** using resource allocation graphs.
   - If a deadlock is detected, it is resolved by terminating processes or preempting resources.

Deadlock Prevention and Avoidance

Deadlock prevention and avoidance are strategies used in computer systems to ensure that different processes can run smoothly without getting stuck waiting for each other indefinitely. Think of it like a traffic system where cars (processes) must move through intersections (resources) without getting into a gridlock.

Necessary Conditions for Deadlock

Padasalai

A deadlock occurs when the following four conditions hold simultaneously:

1. **Mutual Exclusion**: Only one process can use a resource at a time.
2. **Hold and Wait**: A process holding at least one resource is waiting to acquire 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 exist where each process is waiting for a resource held by the next process in the chain, forming a cycle.

Deadlock Prevention

Deadlock prevention aims to eliminate one or more of the necessary conditions for deadlock.

Eliminating Mutual Exclusion

Since some resources, such as printers and tape drives, are inherently non-shareable, mutual exclusion cannot always be eliminated. However, in cases where sharing is possible, techniques like **Spooling (Simultaneous Peripheral Operations Online)** can be used.

- **Example**: For a printer, spooling allows multiple processes to send print jobs to a queue. The printer processes jobs sequentially, ensuring that no process has to wait indefinitely.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Eliminating Hold and Wait

Hold and wait can be eliminated in two ways:

1. **Eliminating wait**: A process must request and acquire all required resources before execution starts.
   - *Example*: Process P1 declares in advance that it requires both Resource R1 and Resource R2 before execution begins.
2. **Eliminating hold**: A process must release all currently held resources before making new requests.
   - *Example*: Process P1 must release Resource R2 and R3 before requesting Resource R1.

Eliminating No Preemption

Preemption allows resources to be forcibly taken from a process and reassigned to others.

- **Processes must release resources voluntarily**: A process should release resources only when it has completed its task.
- **Avoid partial allocation**: Allocate all required resources at once before a process starts execution. If not all resources are available, the process must wait.

Eliminating Circular Wait

Circular wait can be prevented by enforcing an ordering rule on resource allocation.

- **Method**:
  1. Assign a unique numerical ID to each resource.
  2. Require processes to request resources in increasing order of their assigned IDs.
  3. No process can request a resource with a lower ID than what it currently holds.
- *Example*: If a printer has ID 1 and a scanner has ID 2, a process holding the printer cannot request the scanner unless it releases the printer first.

Padasalai

Deadlock Detection and Recovery

If deadlocks are not prevented or avoided, they must be detected and resolved.

- **Detection**: The system runs a deadlock detection algorithm periodically.
- **Recovery Methods**:
  - Terminate one or more deadlocked processes.
  - Preempt resources from deadlocked processes and allocate them to others.

Deadlock Avoidance

Deadlock avoidance ensures that resource requests are only granted if they do not lead to deadlock. This requires knowledge of future resource needs.

Banker's Algorithm

The **Banker's Algorithm** is a resource allocation and deadlock avoidance algorithm that tests all resource requests before granting them.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Inputs:

1. **Maximum needs** of resources by each process.
2. **Currently allocated** resources to each process.
3. **Available system resources**.

Conditions for Granting a Request:

- The request made by a process must be ≤ its **maximum need**.
- The request must be ≤ the **available** resources.
- If granted, the system must remain in a **safe state**.

Example of Banker's Algorithm

Given System Resources:

Resource	A	B	C	D
Total	6	5	7	6
Available	3	1	1	2

Processes (Currently Allocated Resources):

Process	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Maximum Resource Needs per Process:

Process	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Calculating Remaining Needs:

Need = Maximum Requirement - Currently Allocated

Process	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

If a request is made, the algorithm checks whether granting it will leave the system in a safe state before proceeding.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Timeouts

To avoid deadlocks caused by indefinite waiting, a **timeout mechanism** can be used to limit how long a process waits for a resource. If the resource is not available within the timeout period, the process releases its resources and retries later.

Summary

Method	Approach
Deadlock Prevention	Prevent at least one of the necessary conditions from occurring.
Deadlock Avoidance	Allow resource requests only if they do not lead to a deadlock.
Detection & Recovery	Detect deadlocks and take corrective actions (e.g., process termination or resource preemption).
Timeouts	Impose time limits on resource requests to prevent indefinite blocking.

By applying these strategies, computer systems can effectively manage resources and prevent deadlocks from hindering performance.

8.Memory Management in Operating Systems

Introduction to Memory Management

Memory management is a critical function of an operating system (OS) responsible for handling the allocation and deallocation of memory to processes. It ensures efficient utilization of memory, system stability, and optimal performance.

Objectives of Memory Management

1. **Efficient Resource Utilization:** Ensures optimal use of available memory by managing allocation and deallocation effectively.
2. **Process Isolation:** Prevents one process from interfering with another, ensuring data integrity.
3. **Multiprogramming Support:** Allows multiple processes to reside in memory simultaneously, maximizing CPU utilization.
4. **Memory Protection:** Implements safeguards to prevent unauthorized access or corruption of memory.
5. **Dynamic Allocation:** Allocates and deallocates memory dynamically as per process requirements.

Memory Hierarchy

Memory in a computer system is structured in a hierarchical manner based on speed, cost, and size:

1. **Registers:**
   - Fastest but smallest storage, located within the CPU.
   - Used for temporary data storage during instruction execution.
2. **Cache Memory:**
   - Provides fast access to frequently used data, reducing access time.
   - L1 (closest to CPU), L2, and L3 caches improve performance.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

3. **Main Memory (RAM):**
    - Primary volatile memory used for executing active processes.
    - Faster than secondary storage but slower than cache.
4. **Secondary Storage (HDD/SSD):**
    - Non-volatile storage for long-term data retention.
    - Used for storing OS, applications, and user data.
5. **Virtual Memory:**
    - Extends RAM using disk space, enabling execution of large programs beyond physical memory capacity.
    - Implemented through paging and swapping mechanisms.

Contiguous Memory Allocation

- Each process is allocated a single contiguous block of memory.
- **Advantages:** Easy to implement and manage.
- **Disadvantages:**
    - Internal fragmentation (due to fixed-sized partitions).
    - External fragmentation (due to small unused memory gaps).

Swapping in Operating System

Swapping is a memory management scheme used in multiprogramming to increase CPU utilization. It involves temporarily moving a process between RAM and disk storage to manage memory space efficiently.

Padasalai

What is Swapping?

- Swapping moves processes between **main memory (RAM) and secondary storage (disk/SSD)**.
- Allows execution of more programs than available RAM.
- Improves multitasking performance at the cost of speed.
- Swapping is also known as **memory compaction**.

Process of Swapping

1. When RAM is full, an inactive process is selected.
2. The process is moved to secondary storage, freeing up RAM.
3. A new process is loaded into the free space.
4. If needed again, the swapped-out process is reloaded.

Swap-in & Swap-out

- **Swap-out:** Moves a process from RAM to disk.
- **Swap-in:** Transfers a process back from disk to RAM.

Example

Imagine a small desk (RAM) holding active documents while storing other papers in a filing cabinet (disk). When needed, papers are swapped between desk and cabinet.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Advantages of Swapping

- **Better CPU Utilization:** Keeps CPU busy by managing processes efficiently.
- **Efficient Memory Management:** Allows better RAM allocation.
- **Supports Multiprogramming:** Runs multiple processes despite memory limitations.
- **Allows Execution of Larger Programs:** Loads only necessary program parts into RAM.
- **Prevents Overload:** Swaps out inactive processes to free resources.

Disadvantages of Swapping

- **Data Loss Risk:** Power failure can lead to data loss during swaps.
- **Page Fault Overhead:** Frequent swapping increases page faults, slowing performance.
- **Performance Decline:** Excessive swapping (thrashing) reduces execution speed.
- **Increased I/O Overhead:** Disk reads/writes slow down system performance.
- **Dependency on Storage:** Requires fast storage to reduce delays.

Swapping in Single vs. Multitasking OS

- **Single-tasking OS:** A single process remains in memory until execution completes.
- **Multitasking OS:** Active processes may be swapped to allow new processes.

For further understanding, refer to:

- **Difference between Paging and Swapping**
- **Difference between Swapping and Context Switching**

Padasalai

Paging

- Divides memory into fixed-size blocks (pages) that map to frames in physical memory.
- **Advantages:** Eliminates external fragmentation.
- **Disadvantages:** Requires a **page table**, adding overhead.

Paging in Operating System

Introduction

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. It retrieves processes in the form of pages from secondary storage into the main memory. The primary objective of paging is to divide each process into fixed-size pages and allocate them to available memory frames efficiently.

The **Memory Management Unit (MMU)** handles the mapping between logical pages and physical page frames using a **Page Table**. The page table maintains the mapping between logical addresses (used by programs) and physical addresses (actual memory locations in RAM).

Why Paging is Used for Memory Management?

Paging is essential for efficient memory management due to the following reasons:

- **Memory is not always available in a single block:** Programs often require more memory than what is available in a single continuous block. Paging breaks memory into smaller, fixed-size pieces, making it easier to allocate scattered free spaces.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **Processes can grow or shrink dynamically:** Since programs do not need contiguous memory, they can expand without requiring relocation.

Key Terminologies in Paging

Logical Address or Virtual Address

The **Logical Address** (or **Virtual Address**) is the address generated by the CPU when a program accesses memory.

Logical Address Space

The **Logical Address Space** is the total range of logical addresses a process can generate during its execution. It is independent of the actual physical memory.

Physical Address

The **Physical Address** is the actual location in the computer's physical memory (RAM) where data or instructions are stored.

Physical Address Space

The **Physical Address Space** refers to the total range of addresses available in a computer's RAM.

Important Features of Paging in Memory Management

1. **Logical to Physical Address Mapping**
   - The logical address space of a process is divided into fixed-sized pages.
   - Each page is mapped to a corresponding frame in physical memory, enabling flexible memory allocation.
2. **Fixed Page and Frame Size**
   - Paging uses a fixed page size, which is equal to the frame size in physical memory. This uniformity simplifies memory management.
3. **Page Table Entries (PTEs)**
   - Each logical page has a corresponding page table entry (PTE) that stores details about the physical frame it maps to.
4. **Page Table Storage**
   - The page table is stored in the main memory for efficient access and modification.
   - Large page tables introduce overhead, requiring optimizations such as **Translation Lookaside Buffers (TLB)**.

How Paging Works

Paging divides both logical and physical memory into **fixed-size blocks**:

- **Logical memory** is split into **pages**.
- **Physical memory** is split into **frames**.
- The **page size is equal to the frame size**.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Address Translation in Paging

When a program accesses memory:

1. The **CPU generates a logical address** containing:
   - **Page number (p):** Index in the page table.
   - **Page offset (d):** Position within the page.
2. The **page table maps the page number (p) to a frame number (f)** in physical memory.
3. The **physical address** is calculated as:

   Physical Address = (Frame Number << Frame Offset Bits) + Frame Offset

   (Where "<<" denotes a left bitwise shift operation)

Example:

Assume:

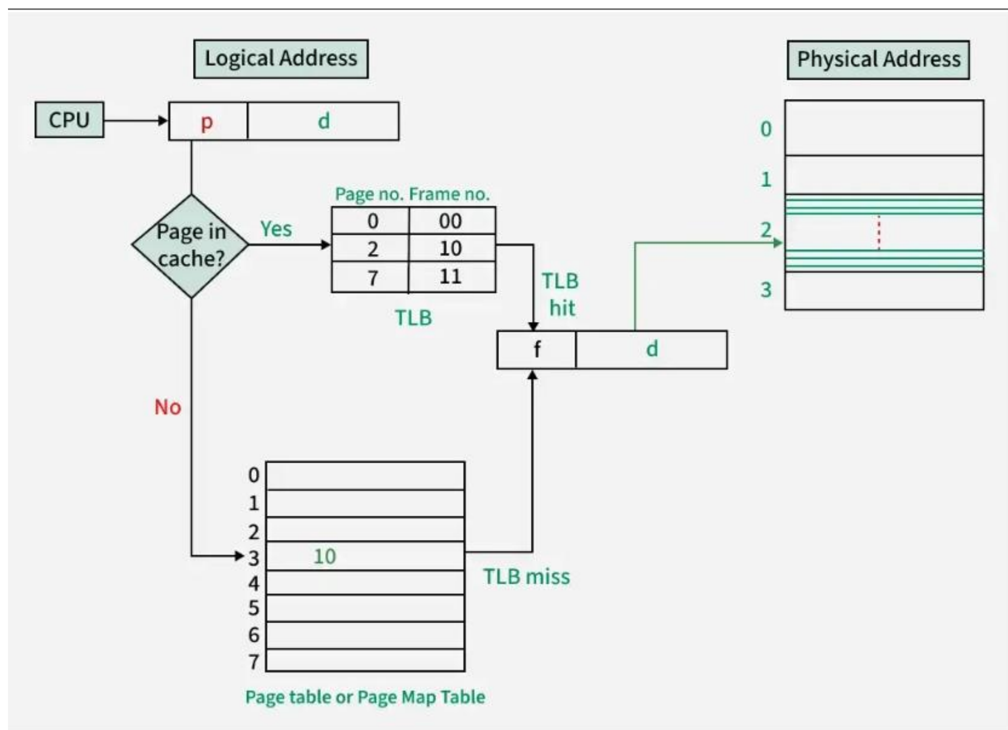
- **Logical Address = 13 bits** → Logical Address Space = 8K words
- **Physical Address = 12 bits** → Physical Address Space = 4K words
- **Page Size = Frame Size = 1K words**

Then:

- **Number of Pages** = Logical Address Space / Page Size = 8K / 1K = 8 (2^3)
- **Number of Frames** = Physical Address Space / Frame Size = 4K / 1K = 4 (2^2)

Hardware Implementation of Paging

Page Table Storage Methods



Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

1. **Dedicated Registers** (For small page tables)
2. **Translation Lookaside Buffer (TLB)** (For large page tables)
    - The **TLB** is an associative, high-speed memory cache storing frequently used page table entries.
    - Each entry consists of:
        - **Tag:** Logical page number.
        - **Value:** Corresponding frame number.
    - TLB lookup reduces address translation time.

Effective Memory Access Time Calculation

If:

- Memory access time = **m**
- Page table is stored in memory,

Then, **Effective Access Time** = **m (for page table lookup)** + **m (for accessing data)** = **2m**

Using **TLB**, access time improves significantly as most lookups avoid main memory.

Advantages of Paging

1. **Eliminates External Fragmentation**
    - Fixed-size pages prevent wasted memory due to fragmentation.
2. **Efficient Memory Utilization**
    - Non-contiguous allocation allows better use of available memory.
3. **Supports Virtual Memory**
    - Enables running programs larger than physical memory.
4. **Ease of Swapping**
    - Pages can be swapped between RAM and disk independently.
5. **Improved Security and Isolation**
    - Each process operates within its own pages, preventing unauthorized access.

Disadvantages of Paging

1. **Internal Fragmentation**
    - If a process does not use all space in a page, memory is wasted.
2. **Increased Overhead**
    - Storing and managing the page table consumes memory.
3. **Page Table Lookup Time**
    - Address translation adds extra time to memory access.
4. **I/O Overhead During Page Faults**
    - Retrieving pages from disk causes delays.
5. **Complexity in Implementation**
    - Requires sophisticated hardware (e.g., MMU, page replacement algorithms).

Conclusion

Paging is an efficient memory management technique that simplifies memory allocation, prevents external fragmentation, and enables virtual memory. However, it introduces overhead due to page table storage and lookup times. Optimization techniques like **TLB** and **hierarchical paging** help mitigate these challenges, improving system performance.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Segmentation

- Divides memory into variable-sized segments.
- Each segment corresponds to logical units such as functions or arrays.
- **Advantages:** More natural program structure.
- **Disadvantages:** Can cause external fragmentation.

Segmentation in Operating System

A process is divided into **segments**. The chunks that a program is divided into, which are not necessarily of the exact same sizes, are called **segments**. Segmentation provides a **user's logical view of the process**, unlike paging, which does not provide this view. In segmentation, the **user's view is mapped to physical memory**.

Types of Segmentation in Operating Systems

1. **Virtual Memory Segmentation**: Each process is divided into multiple segments, but segmentation is not necessarily performed all at once. This segmentation **may or may not occur at runtime**.
2. **Simple Segmentation**: Each process is divided into segments, all of which are **loaded into memory at runtime** but not necessarily in contiguous memory locations.

There is no **direct** relationship between logical addresses and physical addresses in segmentation. Instead, a **Segment Table** stores information about segments.

Padasalai

Segment Table

A **Segment Table** maps a **two-dimensional logical address** into a **one-dimensional physical address**. Each entry in the Segment Table contains:

- **Base Address**: The starting physical address where the segment resides in memory.
- **Segment Limit**: Also known as **segment offset**, specifies the **length of the segment**.

The **address generated by the CPU** is divided into:

- **Segment number (s)**: Number of bits required to represent the segment.
- **Segment offset (d)**: Number of bits required to represent the **position of data within a segment**.

Advantages of Segmentation in Operating Systems

- **Reduced Internal Fragmentation**: Since segments are allocated dynamically according to process needs, internal fragmentation is minimized compared to fixed-size paging.
- **Efficient Memory Utilization**: The **Segment Table is smaller** compared to a Page Table in paging.
- **Improved CPU Utilization**: Since complete modules are loaded at once, CPU utilization improves.
- **Better User Representation**: Segmentation follows a **user's logical program structure**, making it more intuitive.
- **Security and Protection**: Segmentation provides better **data isolation**, enabling controlled access.
- **Flexibility**: Segments can be **variable-sized**, unlike paging, where page size is fixed by the hardware.
- **Memory Sharing**: Segmentation **supports inter-process communication** by allowing shared segments.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

- **Security**: Different segments can have **different access rights**, improving system security.

Disadvantages of Segmentation in Operating Systems

- **External Fragmentation**: Memory **gets fragmented** over time as processes are loaded and removed.
- **Segment Table Overhead**: Maintaining a **segment table for each process** adds overhead.
- **Slower Memory Access**: Since memory access requires two lookups (one in the segment table, one in memory), access time increases.
- **Complex Implementation**: Managing **variable-sized segments** adds complexity to memory allocation algorithms.

Demand Paging in Operating System

Definition: Demand paging is a memory management technique where **only necessary pages** of a program are loaded into memory at runtime. Instead of loading the entire program, pages are loaded **on demand** when required.

Real-Life Analogy:

Imagine reading a large book but only carrying the pages you need instead of the entire book. Similarly, demand paging loads only required **pages of a program** into memory, reducing memory usage.

How Demand Paging Works?

Padasalai

- **A program is divided into pages** (fixed-size memory blocks).
- **Pages are stored in secondary storage (hard disk)**.
- When a page is needed but not in memory, a **page fault occurs**.
- The **OS loads the required page into memory** and updates the page table.
- The program continues executing as if the page was always in memory.

Page Fault

A **page fault** occurs when a program accesses a page **not present in main memory**. The OS then:

- Retrieves the page from secondary storage (hard disk).
- Updates the page table.
- Resumes program execution.

Thrashing

Thrashing occurs when excessive paging activity **overloads the system**, leading to a drastic drop in performance. This happens when:

- **Insufficient physical memory** forces frequent swapping of pages.
- **Poor memory management** results in unnecessary paging.
- **Overloaded processes** require more memory than available.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

To prevent thrashing, the OS can:

- **Limit running processes**.
- **Optimize page replacement strategies**.
- **Increase physical memory (RAM)**.

Pure Demand Paging

In **pure demand paging**, **no pages are loaded initially**; all pages remain in secondary storage until required. This method **minimizes initial memory usage** but may lead to **increased page faults**.

Working of Demand Paging

1. **Program Execution**: The OS **allocates memory** and sets up a page table.
2. **Handling Page Faults**: If a page is not in memory, a **page fault occurs**.
3. **Page Fetch**: The OS **retrieves the required page from disk**.
4. **Page Replacement**: If memory is full, an **existing page is replaced**.
5. **Updating Page Table**: The new page's location is updated in the **page table**.
6. **Program Resumes Execution**.

Common Page Replacement Algorithms

1. **FIFO (First-In-First-Out)**: Replaces the oldest page in memory.
2. **LRU (Least Recently Used)**: Replaces the page **unused for the longest time**.
3. **LFU (Least Frequently Used)**: Replaces the **least accessed** page.
4. **MRU (Most Recently Used)**: Replaces the **most recently accessed** page.
5. **Random**: Replaces a page **randomly**.

Impact of Demand Paging on Virtual Memory Management

- **Reduces memory requirement**, allowing larger programs to run.
- **Improves system performance** by efficiently utilizing RAM.
- **Delays execution if too many page faults occur**.

Demand Paging vs. Pre-Paging

Feature	Demand Paging	Pre-Paging
Pages Loaded	When needed	Before execution
Memory Efficiency	High	Lower (may load unnecessary pages)
Initial Delay	Low	Higher
Performance	Can slow due to page faults	Faster program execution

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Advantages of Demand Paging

- **Efficient Memory Usage**: Loads only necessary pages, minimizing RAM usage.
- **Supports Large Programs**: Enables execution of programs larger than physical memory.
- **Faster Program Start**: Since only part of the program is loaded initially.
- **Optimized Disk I/O**: Reduces unnecessary loading of pages.

Disadvantages of Demand Paging

- **Page Fault Overhead**: Excessive page faults degrade performance.
- **Increased Latency**: Retrieving pages from disk is slow.
- **Fragmentation**: May cause **physical memory fragmentation**.
- **Complex Implementation**: Requires **sophisticated page management** algorithms.

Page Replacement Algorithms in Operating Systems

Introduction

In an operating system that uses paging for memory management, a page replacement algorithm is required to decide which page needs to be replaced when a new page is loaded into memory. Page replacement becomes necessary when a page fault occurs, and there are no free frames available in physical memory.

Page Replacement Algorithms

Page replacement algorithms help manage memory efficiently when virtual memory is full. When a new page needs to be loaded and there is no free space, these algorithms determine which existing page to replace.

Padasalai

The virtual memory manager performs page replacement by:

1. Selecting a page for replacement using a page replacement algorithm.
2. Marking the selected page as "not present" in memory.
3. Initiating a page-out operation if the page is modified (dirty page).

Common Page Replacement Techniques

- **First In First Out (FIFO)**
- **Optimal Page Replacement**
- **Least Recently Used (LRU)**
- **Most Recently Used (MRU)**

First In First Out (FIFO)

- The simplest page replacement algorithm.
- Maintains a queue where the oldest page is at the front.
- When a new page needs to be loaded, the oldest page is removed.

Example: Given a page reference string: `1, 3, 0, 3, 5, 6, 3` with `3` page frames:

- Initially, all slots are empty, so the first three pages cause **3 page faults**.
- When `3` arrives again, it's already in memory (**0 page faults**).
- `5` replaces `1`, `6` replaces `3`, and `3` replaces `0`, each causing **1 page fault**.

Implementation: FIFO can be implemented using a queue.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Optimal Page Replacement

- Replaces the page that will not be used for the longest duration in the future.
- Provides the lowest number of page faults but is impractical since future page references are unknown.

Example: Given a page reference string: `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3` with `4` frames:

- The first four pages cause **4 page faults**.
- Subsequent replacements are based on future use, minimizing page faults.

Optimal Page Replacement serves as a benchmark for other algorithms.

Least Recently Used (LRU)

- Replaces the page that has not been used for the longest time.
- Requires additional data structures (e.g., stack or counters) to track usage.

Example: Given a reference string: `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3` with `4` frames:

- The first four pages cause **4 page faults**.
- Each subsequent replacement is based on past usage, minimizing faults.

Implementation: LRU can be implemented using a stack or a linked list.

Most Recently Used (MRU)

Padasalai

- Replaces the most recently used page.
- Can lead to Belady's anomaly (higher page faults with more frames).

Example: Given a reference string: `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3` with `4` frames:

- The first four pages cause **4 page faults**.
- Each replacement is based on the most recent usage, leading to higher page faults.

Frame Allocation in Operating Systems

Frame allocation determines how many frames are assigned to each process.

Frame Allocation Strategies:

1. **Equal Allocation:**
   - Each process gets an equal number of frames.
   - Works well if all processes are of similar size.
   - **Disadvantage:** Inefficient when processes vary in size.
2. **Proportional Allocation:**
   - Allocates frames based on the size of the process.
   - Ensures fair distribution based on memory needs.

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Global vs. Local Allocation:

1. **Local Replacement:**
   - A process can only replace pages within its allocated frames.
   - **Advantage:** Stable performance for individual processes.
   - **Disadvantage:** Can lead to inefficient memory usage.
2. **Global Replacement:**
   - A process can take frames from other processes.
   - **Advantage:** Maximizes system throughput.
   - **Disadvantage:** Performance of individual processes may suffer.

Techniques to Handle Thrashing

Thrashing occurs when excessive page faults cause a process to spend more time swapping pages than executing.

Causes of Thrashing:

1. High degree of multiprogramming.
2. Insufficient allocated frames.
3. Poor page replacement policies.

Handling Techniques:

1. **Working Set Model:**
   - Defines a *working set* (set of pages actively used).
   - Ensures enough frames are allocated to match the working set.
   - Thrashing occurs when the total demand for frames exceeds available frames.
2. **Page Fault Frequency (PFF):**
   - Monitors page fault rate to adjust frame allocation.
   - High page fault rate → allocate more frames.
   - Low page fault rate → remove frames.

Memory-Mapped Files in OS

Memory mapping allows a file to be mapped into virtual memory, improving I/O performance.

Mechanism:

1. A file is mapped into memory using demand paging.
2. A page-sized portion of the file is read into physical memory.
3. The file is accessed as if it were part of the process's address space.
4. Multiple processes can map the same file to share data.

Types of Memory-Mapped Files:

1. **Persisted:**
   - Backed by a disk file.
   - Data is saved after the process exits.
   - Useful for large files.
   -
2. **Non-Persisted:**
   - Exists only in memory.
   - Data is lost after the process exits.
   - Used for inter-process communication (IPC).

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com

Advantages of Memory-Mapped Files:

1. Faster access than standard read/write system calls.
2. Efficient I/O for large files.
3. Enables memory sharing across processes.
4. Supports copy-on-write for process isolation.

Disadvantages of Memory-Mapped Files

1. Can be slower than standard file I/O in some cases.
2. Requires hardware support (Memory Management Unit - MMU).
3. Difficult to dynamically expand file size.

Conclusion

Page replacement and frame allocation are crucial for efficient memory management in operating systems. Various strategies help optimize performance and minimize thrashing. Memory-mapped files further enhance I/O efficiency, allowing applications to access disk files as part of memory. Understanding these techniques helps improve system performance and resource management.

Padasalai

Kindly Send Me Your Study Materials To Us Email ID: padasalai.net@gmail.com