Operating Systems & Computer Architecture

- **CT049-3-1-OS&CA**
- **Ver: VE**
- **CPU Scheduling & Process Control Management**

1. Learning Outcomes

At the end of this section, **YOU** should be able to:

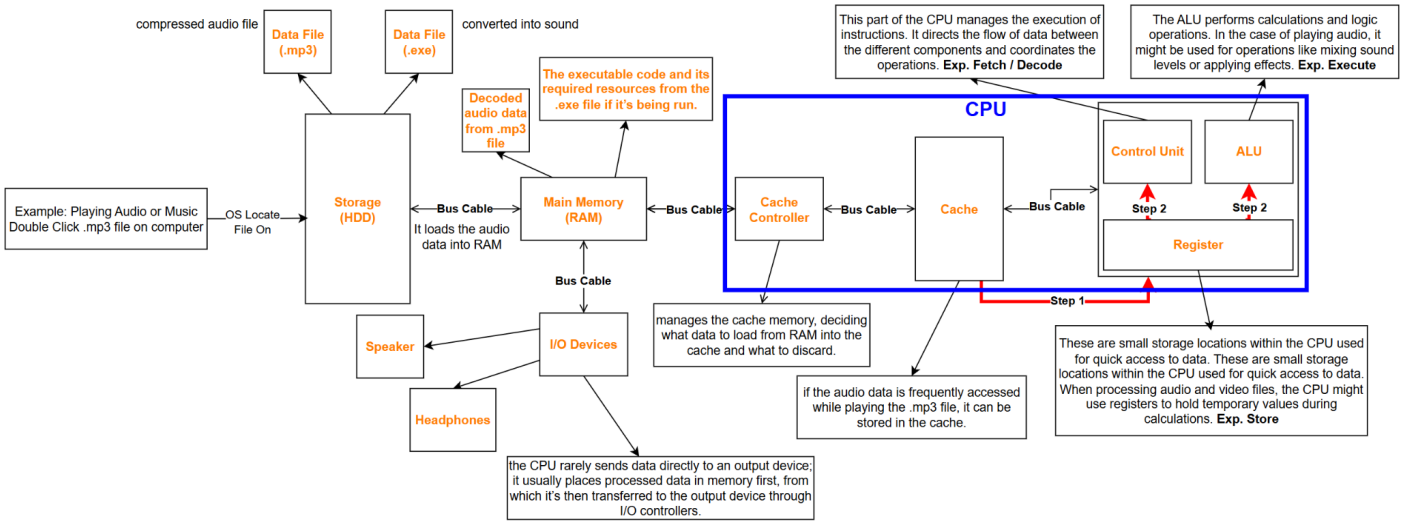
- Define Processes and Threads
- Draw and explain; the process state diagram, the process scheduling diagram and PCB
- State the Aims of CPU Scheduling
- Perform calculations using the various scheduling Algorithms available

2. Topics We Will Cover

- Process and Thread Concepts
- Process States
- Process Scheduling
- CPU scheduling
- Preemptive Scheduling Algorithms
- Non-preemptive Scheduling Algorithms
- Calculation using Preemptive and Non-preemptive Scheduling Algorithms

Table Of Contents

Introduction



To understand how playing a video and handling audio files relate to different components of a computer system—specifically storage, memory, I/O devices, and the CPU—let's break down the process step by step.

1. Storage (HDD)

- **Data Files:**
  - **.mp3 File:** This is a compressed audio file that resides on the hard disk drive (HDD). It's stored as a sequence of bits, which can be read by software when needed.
  - **.exe File (converted into sound):** This executable file might contain code that can generate sound or manipulate audio data. When converted into sound, it will be read and processed similarly to the .mp3 file, but its content is initially a program rather than audio.

2. Main Memory (RAM)

- **Loading Files:** When you play a video or audio file, the operating system loads the necessary data from the HDD into the main memory (RAM). This allows faster access for processing, as RAM is much quicker than HDD.
- **Temporary Storage:** The data in RAM can include:
  - The decoded audio data from the .mp3 file.
  - The executable code and its required resources from the .exe file if it's being run.

3. I/O Devices

- **Output Devices:**
  - **Speakers/Headphones:** These devices receive the audio signals that are output from the CPU after processing the audio files.
  - **Monitor:** For video playback, the monitor displays the visual components of the video file.

4. CPU Components

- **Control Unit:** This part of the CPU manages the execution of instructions. It directs the flow of data between the different components and coordinates the operations.

- **Arithmetic Logic Unit (ALU):**
  - The ALU performs calculations and logic operations. In the case of playing audio, it might be used for operations like mixing sound levels or applying effects.

- **Registers:**
  - These are small storage locations within the CPU used for quick access to data. When processing audio and video files, the CPU might use registers to hold temporary values during calculations.

- **Cache Controller and Cache:**
  - **Cache:** This is a smaller, faster type of volatile memory located within or near the CPU. It stores copies of frequently accessed data from the main memory to speed up processing. For example, if the audio data is frequently accessed while playing the .mp3 file, it can be stored in the cache.
  - **Cache Controller:** This component manages the cache memory, deciding what data to load from RAM into the cache and what to discard.

5. Summary of the Process:

- **File Access:** When you play an audio file, the OS retrieves the file from the HDD.
- **Loading into Memory:** The audio data is loaded into RAM for quick access.
- **Data Processing:**
  - The control unit of the CPU fetches instructions and data.
  - The ALU may perform any necessary computations, such as mixing sounds.
  - Registers hold temporary data for processing.
- **Output Generation:** The processed audio data is sent to the output devices (speakers) for playback.

6. Example Workflow:

- You double-click an .mp3 file on your computer:
    - The operating system locates the file on the HDD.
    - It loads the audio data into RAM.
    - The CPU retrieves the instructions for decoding the audio file, processes it, and sends the output to the speakers.

This interconnection between the HDD, main memory, I/O devices, and various CPU components illustrates the cohesive functioning of a computer system during multimedia playback.

Process

A process is indeed a ***program in execution***, meaning it's an active instance of a program loaded into memory and being managed by the operating system (OS) as it progresses in a **sequence**. Let's dive into the details to give a complete understanding.

1. Key Characteristics of a Process

- **Execution and Sequential Progression**:
  - A process represents a sequence of executed instructions, progressing in an ordered manner. For instance, when you open a web browser, the program transitions into a process that the OS actively manages.

- **Uniquely Identified**:
  - Each process has a *unique Process ID (PID)* that helps the OS keep track of it, differentiate it from others, and manage resource allocation.

- **Resource Requirements**:
  - For its execution, a process needs resources such as:
    - **Memory (RAM)**: To store its code and data.
    - **CPU Time**: To execute its instructions.
    - **Files or I/O Devices**: For reading and writing files, and possibly interacting with peripherals.
  - The OS allocates these resources when the process starts or as required during its execution.

- **Active vs. Passive Entities**:
  - A *program* is a passive file (simply code stored on a disk), while a *process* is an active entity with a current state, resource usage, and progression.

- **Lifecycle Management**:
  - The OS is responsible for:
    - *Creating processes*: For instance, when you run a new application.
    - *Terminating processes*: When processes complete or if they crash.
    - *Switching between processes*: Ensuring efficient CPU usage by cycling between active processes.

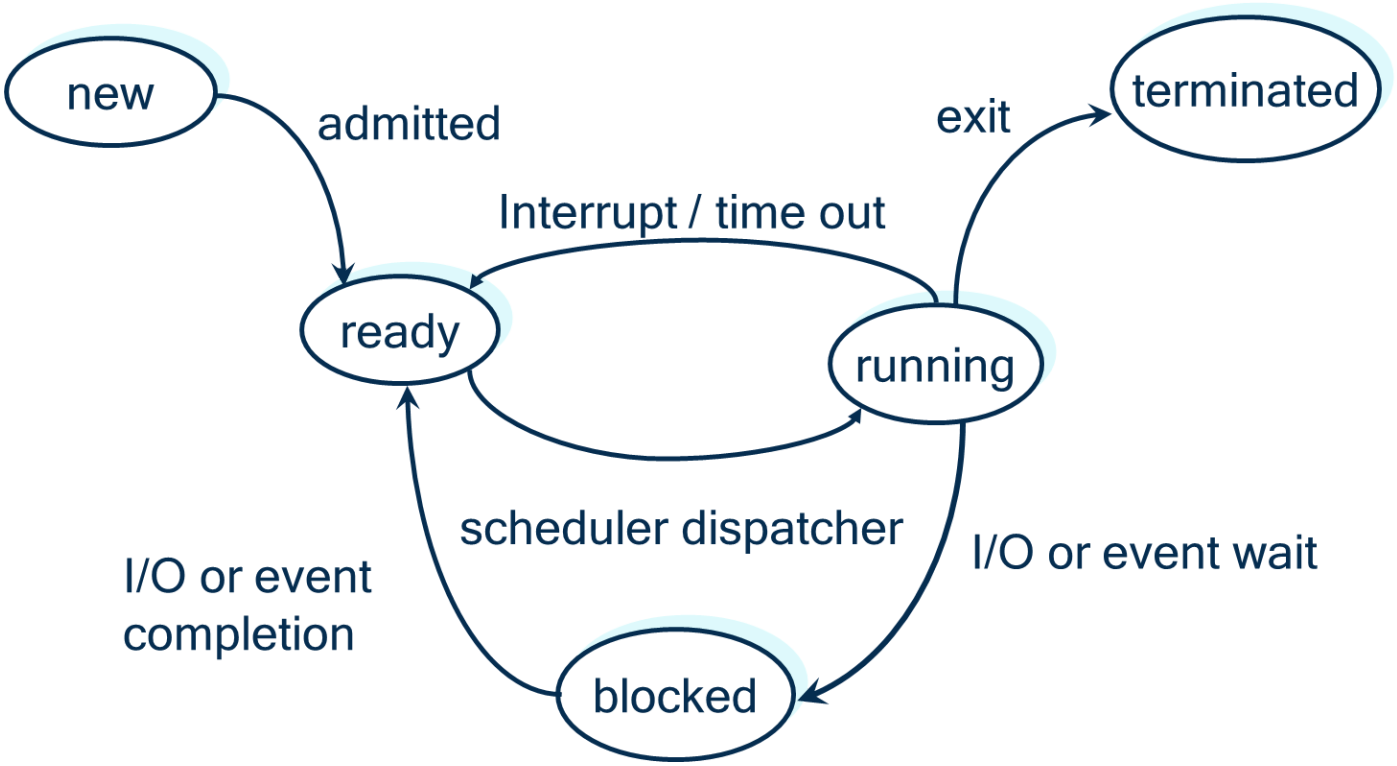
- **Process Table**:
    - The OS uses a *process table* (a data structure) to keep records of all active processes. Each entry typically includes details like the PID, process state (running, ready, waiting), CPU registers, memory locations, and resource ownership.

2. Example to Clarify: Opening a Text Editor

- When you open a text editor like Notepad, the OS:
    - Creates a process from the Notepad program.
    - Assigns it a unique PID.
    - Allocates required resources (memory for code, CPU cycles).
- While Notepad is open, it's considered an active process with its own set of resources.
- When you close it, the OS frees the resources and deletes the process entry from the process table.

This is the basic idea behind a process in an operating system, central to how OSs manage multitasking and resource allocation!

Process States



In an operating system, a process changes states as it progresses from creation to termination. These states are essential for managing resources efficiently and ensuring the system can handle multiple processes effectively. Let's go through each state and the transitions involved.

1. Key Process States and Transitions

- **New**:
  - When a process is initially created, it is in the *New* state.
  - **Reasons for process creation**:
    - ***New batch job***: A batch process is scheduled to run.
    - ***Interactive logon***: A user logs on and starts an application.
  - The process in the ***New*** state will soon be moved to the ***Ready*** state.

- **Ready**:
  - In the *Ready* state, a process is waiting to use the CPU. It has all the required resources except the CPU, meaning it's prepared to execute but waiting for processor time.
  - The OS maintains a list of *Ready* processes, known as the *ready queue*.
  - **Transition to Running**:
    - When the CPU becomes available, a process in the *Ready* state is dispatched to the CPU to begin execution.

- **Running**:
  - A *Running* process is actively using the CPU to execute its instructions.
  - The number of *Running* processes is typically limited by the number of CPU cores available.
  - **Transitions from Running**:
    - ***Blocked***: If the process requests an I/O operation or waits for an external event (like file input), it moves to the *Blocked* state, pausing its execution until the event completes.
    - ***Ready***: The OS may interrupt a *Running* process and move it back to *Ready* if:
      - The process has used its maximum allowed time slice (in systems with time-sharing).
      - The process needs a resource that is currently unavailable.
    - ***Terminated***: When a process completes or encounters a fatal error, it exits to the *Terminated* state.

- **Blocked** (or *Waiting*):
  - A *Blocked* process is paused, waiting for an external event, such as the completion of an I/O operation or the availability of a specific resource.
  - **Transition to Ready**:
    - When the awaited event occurs, the *Blocked* process moves back to the *Ready* state, awaiting CPU allocation again.

- **Terminated**:
  - Once a process finishes execution or is aborted, it transitions to the *Terminated* state.
  - The OS frees all resources associated with the process and removes its entry from the process table.
  - **Reasons for process termination**:
    - *Normal completion*: The process has finished all its tasks.
    - *Errors*: Issues like invalid instructions or memory allocation failure lead to termination.

2. Example to Illustrate Process State Transitions

Imagine opening a document in a text editor:

- The editor's process is created (*New*) and moves to *Ready* when the OS prepares it for execution.
- When the CPU becomes available, the process transitions to *Running*.
- As the editor waits for the document to load from disk, it becomes *Blocked*, waiting on the I/O operation.
- Once the document loads, the process returns to *Ready* and soon moves to *Running*.
- Finally, when the user closes the editor, the process transitions to *Terminated*, freeing resources.

These states allow the OS to control multiple processes efficiently, making sure resources are used effectively and preventing any single process from monopolizing CPU time.

Quick Review Questions

Question 1:

For each of the following transitions between process states, indicate whether the transition is **possible** or **not possible?**

ready ⟶ running

ready ⟶ new

blocked ⟶ ready

Answer:

Let's go through each transition:

- **Ready ⟶ Running**
  - **Possible**: This transition happens when a process in the *Ready* state is allocated the CPU and begins execution.

- **Ready ⟶ New**
  - **Not Possible**: Once a process is in the *Ready* state, it cannot go back to the *New* state. A process only enters the *New* state upon creation.

- **Blocked ⟶ Ready**
  - **Possible**: This transition occurs when a *Blocked* process has been waiting for an external event (like an I/O operation) to complete, and the event has now completed, allowing it to move back to *Ready*.

Question 2:

On a system with **n** number of CPU's what is the **minimum number of processes** that can be in the ready, running and blocked state?

Answer:

Minimum Number of Processes in Each State on a System with n CPUs

- **Running**: The minimum number of processes in the *Running* state is **0** (when no processes are being executed, for example, if the system is idle). The maximum is n, as the number of *Running* processes is limited by the number of CPUs.

- **Ready**: The minimum number of processes in the *Ready* state is **0** (when no processes are waiting for the CPU). However, typically, there is at least 1 process in the *Ready* state, awaiting its turn.

- **Blocked**: The minimum number of processes in the *Blocked* state is also **0** (when no processes are waiting for an external event).

So, in summary:

- *Running*: 0 to n
- *Ready*: 0 or more
- *Blocked*: 0 or more

Process Control Block (PCB)

A **Process Control Block (PCB)** is a crucial data structure used by the operating system (OS) to manage processes. Each process has its own PCB, which contains essential **information** needed to **keep track** of the process's state and manage its execution. Here's a breakdown of what the PCB includes and why each part is important:

1. Key Components of a PCB

- **Pointer to Parent Process**:
  - **Description**: This pointer links each process to its *parent process*, the one that created it.
  - **Purpose**: Enables the operating system to manage the hierarchy and relationships between processes, allowing a parent to track its children.

- **Pointer Area to Child Process**:
  - **Description**: This area contains pointers to any *child processes* that a process has spawned.
  - **Purpose**: Allows the OS to maintain control over process trees, where a parent process may need to coordinate, monitor, or terminate its children.

- **Process State**:
  - **Description**: Indicates the current *status* of the process, such as *New*, *Ready*, *Running*, *Blocked*, or *Terminated*.
  - **Purpose**: Helps the OS manage CPU time allocation by understanding whether the process is ready to run, executing, waiting, or finished.

- **Program Counter**:
  - **Description**: A **register** that holds the address of the *next instruction* to execute within the process.
  - **Purpose**: Ensures the process continues from where it last left off after a pause or context switch, enabling continuity of execution.

- **Register Save Area**:
  - **Description**: Stores the values of the *CPU registers* specific to each process.
  - **Purpose**: Allows the OS to save and restore the CPU state during context switches, ensuring each process resumes with its data intact.

- **Memory Limits**:
  - **Description**: Defines the *bounds* of memory allocated to the process, such as the base and limit registers.
  - **Purpose**: Helps protect a process's memory space, preventing it from accessing or modifying memory allocated to other processes.

- **Priority Information**:
  - **Description**: Information on the *priority level* of the process, which can influence scheduling.
  - **Purpose**: Assists the scheduler in allocating CPU time based on the process's priority, with higher-priority processes generally receiving more CPU time.

- **Accounting Information**:
  - **Description**: Contains *statistics* and tracking data such as CPU time used, memory consumed, and the process's ID.
  - **Purpose**: Used for resource tracking, billing, and monitoring, helping the OS manage resources fairly and efficiently.

- **Pointer to Files and Other I/O Resources**:
  - **Description**: Stores pointers to *open files* and other I/O resources the process is using.
  - **Purpose**: Allows the OS to manage I/O operations, track files in use, and ensure the process has the necessary permissions and access to resources.

These components are crucial for the OS to manage processes effectively, maintain system stability, and ensure efficient resource allocation and security.

2. Purpose and Importance of the PCB

The PCB serves as a **repository of information** that varies between processes, allowing the OS to switch between them efficiently. When a process is interrupted (say by an I/O request or a higher-priority process), the OS saves its context (i.e., the PCB), allowing the process to resume smoothly when it's scheduled again.

3. Example: PCB in Context Switching

Imagine two processes, A and B:

1. Process A is running, but it's interrupted because Process B needs CPU time.
2. The OS saves Process A's state and program counter in its PCB.
3. The OS then loads Process B's information from its PCB and begins its execution.
4. Later, when A is rescheduled, the OS uses its PCB to restore its state, allowing it to resume exactly where it left off.

PCBs are central to process management and scheduling, enabling the OS to maintain efficient and organized multitasking, even in complex environments.

Process Scheduling

Process Scheduling is the method by which the operating system (OS) manages the execution of multiple processes by allocating CPU resources. This scheduling is essential for multitasking, allowing the OS to control when each process executes, how long it runs, and how resources are shared.

1. Key Components and Stages in Process Scheduling

1A. Job Queue:

- When a new process enters the system, it is placed in the *job queue*, which holds all processes that await CPU time.
- The OS schedules each process based on policies (e.g., priority, shortest job first) to determine which will move next to the *ready queue*.

1B. Ready Queue:

- Once a process is prepared for execution (it has necessary resources, excluding the CPU), it moves to the *ready queue*.
- The OS selects processes from this queue and dispatches them to the CPU based on the chosen scheduling algorithm (e.g., round-robin, priority scheduling).

1C. Dispatching and Execution:

- When a process is dispatched from the *ready queue*, it is given CPU resources and enters the *running* state.

- The CPU executes the process's instructions, and during this time, several events may occur:
    - **I/O Request**: If the process requires an I/O operation, it is moved from the *running* state to a *device queue* (or *blocked state*) until the I/O operation is complete.

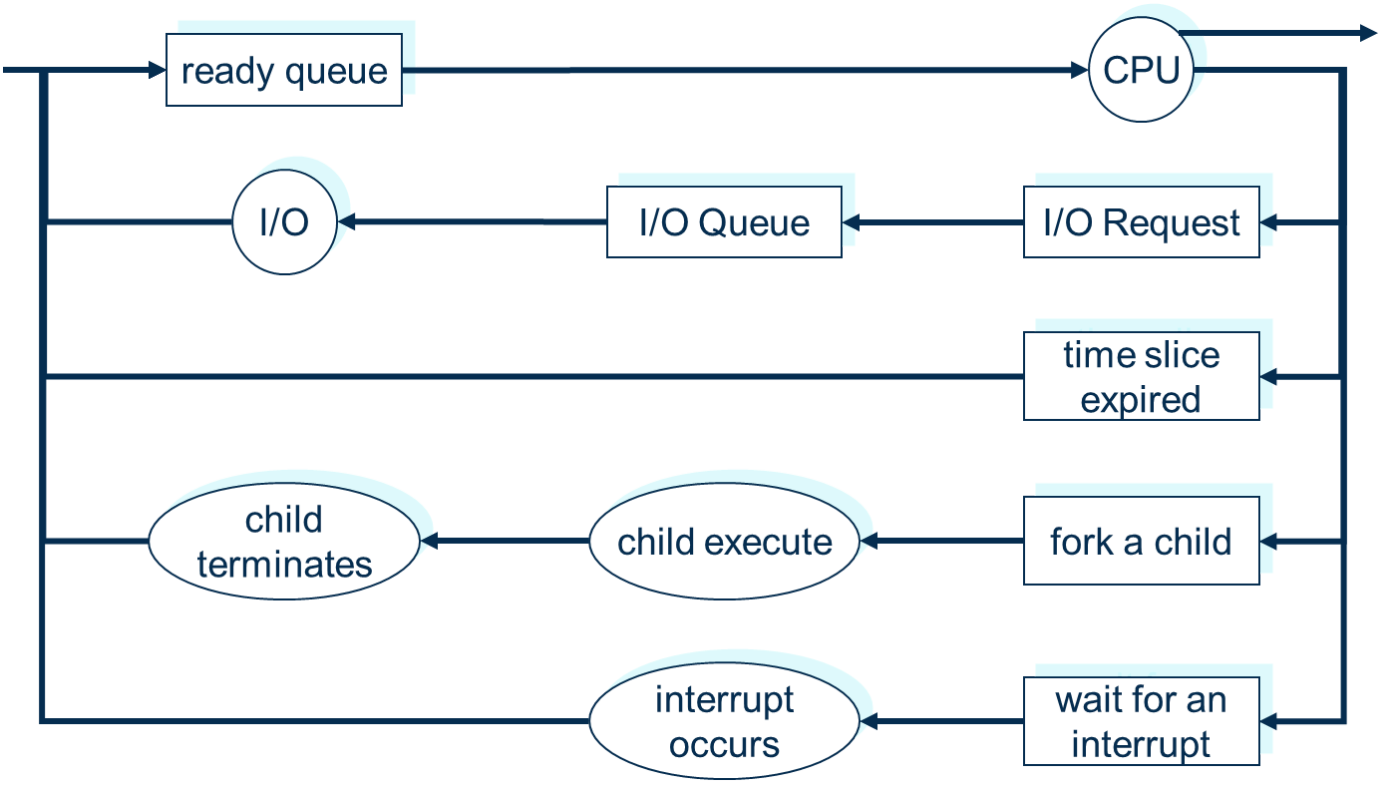
    - **Creation of a Sub-process**: The running process may create a new sub-process. In this case, the new sub-process is placed on the *ready queue*, awaiting CPU time.

    - **Forced Removal (Preemption)**: The OS may forcibly remove the process from the CPU to give other processes a turn. This happens in preemptive multitasking environments, often because the process has reached its time slice limit or a higher-priority process needs CPU time.

3. Example: Process Scheduling Flow

Let's say a user opens a video editing application:

- The OS places the application's process in the *job queue*.
- Once resources are ready, the process moves to the *ready queue*.
- When the CPU becomes available, the OS dispatches the process, and it starts executing instructions.
- During editing, the process might request to save a video, which initiates an I/O operation, placing the process in a *device queue* while the data is saved.
- After the I/O completes, the process returns to the *ready queue* and waits to resume execution.
- The OS continues to dispatch and manage processes in this manner, balancing CPU access between multiple tasks.



4. Importance of Process Scheduling

Process scheduling is vital for efficient multitasking and resource sharing. It ensures:

- **Maximization of CPU Utilization**: By cycling between processes quickly, the CPU remains busy, improving overall performance.

- **Fair Allocation of CPU Time**: Processes receive fair CPU access based on scheduling policies.

- **Balanced I/O and Processing**: By organizing device queues, the OS avoids bottlenecks, allowing processes to run smoothly and respond to user interactions efficiently.

Scheduling enables the OS to handle many processes concurrently, optimizing system performance and user experience.

Concurrent Processes

Concurrent Processes are processes that exist and execute simultaneously within a system. They can be either independent or cooperating, depending on their need (or lack of need) to interact with other processes.

1. Types of Concurrent Processes

- **Independent Processes**:
  - These are processes that operate separately **without any need for interaction** or shared resources with other processes.
  - They execute independently, meaning their actions do not affect other processes, and they are not influenced by other processes.
  - Example: A media player running while a separate file download process occurs in the background. Each operates independently, with no direct impact on the other.

- **Cooperating Processes**:
  - These are processes that *work together*, meaning they need to communicate or share resources to perform tasks effectively.
  - *Cooperating processes* can affect and be affected by other processes. They often share data or coordinate tasks for complex operations.
  - Example: In a web browser, multiple processes may work together to load, render, and display pages, with each part relying on shared data and communication with the others.

2. Why Cooperation Matters in Concurrent Processing

Cooperating processes can improve system efficiency and enable complex task management, such as parallel data processing or distributed computations. However, cooperation introduces challenges like synchronization (ensuring that processes access shared data in the correct sequence) and resource sharing (preventing conflicts over shared resources).

3. Real-World Example of Cooperation

Consider a database management system:

- A *database client* process reads and writes data.
- A *logging process* records changes.
- Both processes cooperate, sharing access to data but requiring synchronization to prevent issues like inconsistent reads or writes.

Understanding the difference between independent and cooperating processes is crucial for effective concurrent process management, as it guides decisions on process isolation, synchronization mechanisms, and resource allocation strategies.

4. Why should an operating system allow for cooperating processes?

An operating system (OS) allows for cooperating processes because they bring significant advantages in terms of information sharing, resource access, and improved computation speed. Here's why these benefits are important:

4A. Reasons for Allowing Cooperating Processes

- **Information Sharing**:
    - Many applications need to share information between processes. For instance, a word processor and a spelling checker can work together, with each process accessing the same document.
    - By allowing cooperation, the OS enables processes to exchange data efficiently, enhancing functionality and enabling complex tasks that would be difficult to achieve with independent processes alone.

- **Efficient Resource Utilization**:
    - Cooperating processes can share resources (like memory, files, and devices) without duplicating them, which conserves system resources.
    - The *Process Control Block (PCB)* plays a crucial role here, as it keeps track of each process's resource usage and permissions, enabling safe and organized access to shared resources.

- **Increased Computation Speed**:
    - Cooperation allows processes to split tasks into smaller, parallel operations, which can be performed simultaneously. This parallelism can significantly speed up computation by distributing workload across multiple processes or processors.
    - For example, in multimedia applications, decoding, rendering, and displaying frames can happen concurrently, leading to faster and smoother output.

- **Modularity and Simplified Design**:
    - Cooperation makes it easier to design complex systems by breaking them into smaller, specialized processes that interact. Each process can handle a specific aspect of a task, making the system more modular and easier to develop and maintain.
    - For instance, in a web server, separate processes may handle request parsing, database access, and response generation. This separation of concerns improves system reliability and makes debugging easier.

- **Improved Responsiveness and Scalability**:
    - Cooperating processes can be structured to enhance the responsiveness of applications, as separate processes can handle user input, background tasks, and other activities simultaneously.
    - This setup allows the system to scale by adding more cooperating processes, making it possible to handle higher workloads without re-engineering the application.

6. Example: Web Browsing Application

Consider a web browser where:

- One process downloads webpage data.
- Another renders and displays the content.
- A third handles user input, like scrolling or clicking links.

These cooperating processes share information and resources (like memory and network access), making the browser faster, more responsive, and modular.

In summary, by supporting cooperating processes, an OS optimizes for performance, resource use, and flexibility, making it essential for modern multitasking environments.

Thread

A **Thread** is a **lightweight**, **smaller unit** within a **process** that can **run independently** from other parts of the same process. Threads allow a single process to handle multiple tasks concurrently, which enhances application performance and responsiveness, particularly in multi-core processors.

1. Key Characteristics of Threads

- Threads share the same resources and memory space of their parent process, making them quicker and more efficient to create and manage than creating entirely new processes.
- Each thread has its own program counter, stack, and register set but shares the process's memory and resources with other threads of the same process.

2. Types of Threads

- **User Threads**:
  - Managed by user-level libraries within applications without kernel support, so the OS does not directly manage these threads.
  - They are generally faster to create and manage than kernel threads because they don't require kernel intervention.
  - Example: Thread libraries like POSIX Threads (Pthreads) in Unix systems allow applications to create multiple user threads within a process.

- **System (or Kernel) Threads**:
  - Managed directly by the operating system's kernel, which schedules and manages them like traditional processes.
  - Kernel threads can communicate directly with the OS, allowing better integration with hardware and more control over resources but also incurring a bit more overhead.
  - Example: Many operating systems (like Windows and Linux) natively support kernel threads, using them to handle system-level tasks and operations.

3. Thread Creation: Spawning and Forking

- **Spawning** or **Forking** refers to creating a new process (or *child process*) from an existing one (the *parent process*). In many systems, the fork() system call is used to duplicate a process, with the child process inheriting the parent's memory and resources.

- In multi-threaded applications, creating a new thread within the same process is often preferred over creating a new process. This is because threads share resources, are faster to create, and reduce the memory overhead associated with creating a new process.

4. Example of Thread Usage

In a web server:

- One thread might handle client requests.
- Another might manage database interactions.
- A third might log events.

Using threads, the server can process multiple tasks concurrently, maximizing resource use and responsiveness.

In summary, threads allow finer granularity in task management within a process, making applications faster and more efficient by leveraging concurrent execution.

Quick Review Question

Question 1. What is a Process Control Block (PCB)?

A **Process Control Block (PCB)** is a data structure used by the operating system to store and manage information about a specific process. Each process in the system has its own PCB, which contains essential details that allow the OS to control and coordinate the process. The PCB includes:

- **Process State**: Indicates the current state of the process (e.g., ready, running, blocked).
- **Program Counter**: Holds the address of the next instruction to execute.
- **CPU Scheduling Information**: Includes process priority and pointers for scheduling queues.
- **Memory Management Information**: Tracks memory locations allocated to the process.
- **Accounting Information**: Records CPU usage, process IDs, etc.
- **I/O Status Information**: Lists I/O devices allocated to the process.

The PCB serves as the process's "identity" in the OS, helping to manage resources, scheduling, and context switching.

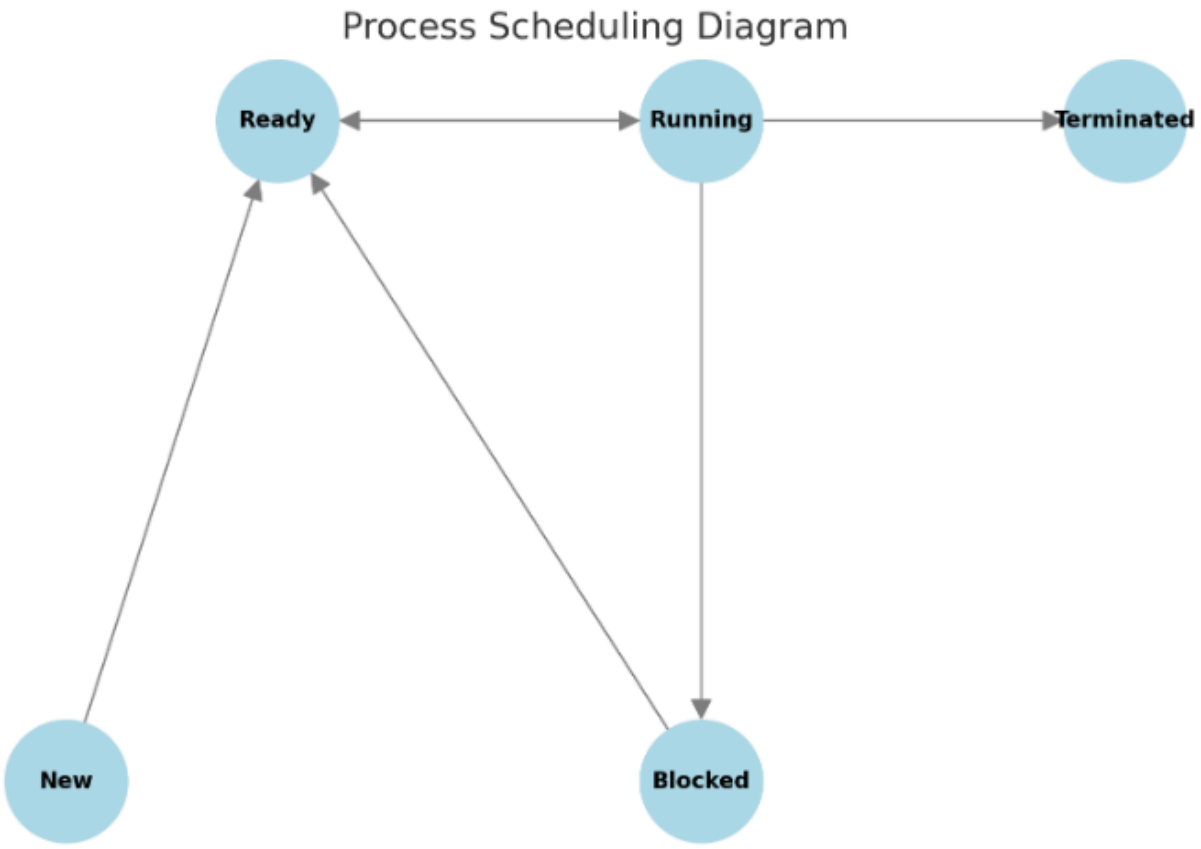
Question 2. Process Scheduling Diagram

Below is an explanation of the **process scheduling diagram**, showing the possible states a process can move through:

- **New**: The process is created and added to the job queue.
- **Ready**: The process is prepared to execute but is waiting for CPU time in the ready queue.
- **Running**: The process is currently executing on the CPU.
- **Blocked (or Waiting)**: The process is paused, waiting for an external event like I/O completion.
- **Terminated**: The process has completed execution and is removed from the system.

Transitions between states:

- *New to Ready*: A newly created process moves to the ready queue.
- *Ready to Running*: The OS schedules the process to run on the CPU.
- *Running to Blocked*: The process waits for I/O or an external event.
- *Blocked to Ready*: The event completes, and the process returns to the ready queue.
- *Running to Ready*: The OS may interrupt the process (e.g., time slice ends).
- *Running to Terminated*: The process completes or encounters an error.



Here's the **Process Scheduling Diagram**:

- **New** → **Ready**: When a process is created, it moves to the *Ready* queue.
- **Ready** → **Running**: The OS assigns the CPU to a process from the *Ready* queue.
- **Running** → **Blocked**: The process requests I/O or an external event, pausing execution.
- **Blocked** → **Ready**: The external event completes, and the process returns to the *Ready* queue.
- **Running** → **Ready**: The OS preempts the process (e.g., due to time slicing).
- **Running** → **Terminated**: The process completes execution or encounters an error.

This diagram visually represents the transitions that manage process scheduling and state changes within the OS.

Follow Up Assignment

Question1. What are the differences and similarities between a process and a thread?

Differences:

- **Independence**:
  - **Process**: A process is an independent program in execution, with its own memory space and resources allocated by the OS.
  - **Thread**: A thread is a smaller unit within a process and shares the process's memory and resources, making it "lighter" in resource usage.

- **Memory and Resource Allocation**:
  - **Process**: Each process has its own memory space (address space), which includes the code, data, and stack sections.
  - **Thread**: Threads within the same process share the same address space, meaning they have access to the same data and resources as the parent process.

- **Creation Overhead**:
  - **Process**: Creating a new process (e.g., through forking) requires more time and resources since it involves setting up a separate memory space and additional management structures.
  - **Thread**: Creating a thread is faster and requires less overhead, as it only needs a unique stack and program counter within the same process space.

- **Communication**:
  - **Process**: Inter-process communication (IPC) is required for processes to share data, as they are isolated from one another.
  - **Thread**: Threads within the same process can communicate directly by accessing shared memory, making data exchange faster.

Similarities:

1. **Execution**: Both processes and threads represent a sequence of instructions that the CPU executes, allowing multitasking in an OS.

2. **States**: Both processes and threads go through states like *Ready*, *Running*, *Blocked*, and *Terminated*.

3. **Scheduling**: Both are managed by the OS scheduler, which assigns CPU time based on scheduling policies.

4. **Resource Usage**: Both require CPU time and memory for execution, though threads use fewer resources by sharing within their parent process.

Question 2. Draw and explain the process state diagram.

Process State Diagram

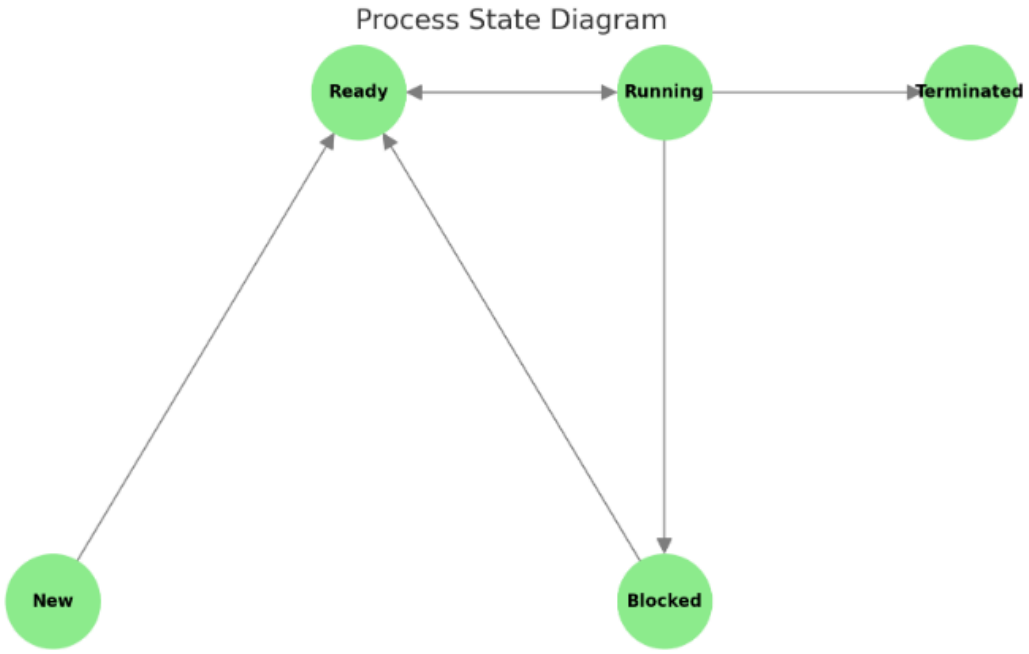
The **Process State Diagram** illustrates the various states a process goes through in an OS. Here's an explanation of each state and its transitions:

1. **New**: The process is created and initialized.
2. **Ready**: The process is ready to execute but is waiting for CPU allocation.
3. **Running**: The process is currently executing on the CPU.
4. **Blocked (or Waiting)**: The process is waiting for an event or I/O operation to complete.
5. **Terminated**: The process has completed execution and exits.

Transitions between states:

- **New to Ready**: The OS prepares the process for execution.
- **Ready to Running**: The scheduler allocates CPU time to the process.
- **Running to Blocked**: The process is paused, awaiting an I/O or event.
- **Blocked to Ready**: The awaited event occurs, and the process returns to the ready state.
- **Running to Ready**: The OS preempts the process (e.g., due to time slicing).
- **Running to Terminated**: The process finishes execution or encounters an error.

I'll create a diagram to represent these states visually.



This **Process State Diagram** illustrates the transitions between different process states:

- **New** → **Ready**: Process is created and added to the ready queue.
- **Ready** → **Running**: CPU is assigned to the process, allowing it to execute.
- **Running** → **Blocked**: Process is waiting for an I/O operation or event.
- **Blocked** → **Ready**: The awaited event completes, and the process re-enters the ready queue.
- **Running** → **Ready**: The OS preempts the process (e.g., due to a time slice).
- **Running** → **Terminated**: The process completes execution or encounters a fatal error.

Question 3: What does a process table store?

The **Process Table** is a data structure maintained by the operating system that stores information about all active processes. Each entry in the process table corresponds to a **Process Control Block (PCB)** and includes:

- **Process ID (PID)**: A unique identifier for each process.
- **Process State**: The current state of each process (e.g., ready, running, blocked).
- **Program Counter**: Indicates the next instruction to execute for each process.
- **CPU Registers**: Stores the CPU state when a process is not running.
- **Memory Management Information**: Information on allocated memory spaces and access permissions.
- **I/O Status Information**: Details on I/O devices and files in use by each process.
- **CPU Scheduling Information**: Priority, scheduling queue pointers, and other details influencing CPU allocation.
- **Accounting Information**: Tracks CPU usage, execution time, and resource usage.

The process table is critical for efficient process management, scheduling, and resource tracking.

CPU Scheduling

CPU scheduling is a critical function of the operating system (OS) that ensures optimal CPU utilization and system responsiveness. Let's break down the key components and purposes of CPU scheduling.

1. CPU Scheduling Purpose

- **Efficient CPU Utilization:** CPU scheduling ensures that the CPU is constantly working on tasks, reducing idle time. By quickly switching between processes, it maximizes the usage of CPU resources.
- **Multitasking Support:** In a multitasking system, multiple processes reside in the system memory and are ready for execution. CPU scheduling chooses which process to run next, enabling the system to handle multiple tasks simultaneously.
- **Improved Responsiveness and Throughput:** Proper scheduling makes systems more responsive for users and increases the number of tasks completed within a certain period (throughput).

2. Components of CPU Scheduling

- **CPU Scheduler:** This component is responsible for deciding which process from the **ready queue** gets to use the CPU. The scheduler runs whenever the CPU becomes idle, selecting the next process for execution based on the scheduling algorithm in place.
- **Ready Queue:** This queue holds all processes that are ready to execute but are currently waiting for CPU time. Processes are added to the ready queue after creation and once they complete I/O or wait operations.
- **Scheduling Algorithm:** The CPU scheduler relies on this algorithm to make decisions about which process should be executed next.

3. Types of CPU Scheduling Algorithms

Scheduling algorithms vary in complexity and goals. Here are a few common algorithms:

- **First-Come, First-Served (FCFS):**
  - This is the simplest scheduling algorithm. Processes are executed in the order they arrive in the ready queue.
  - Example: Imagine three processes arriving in the following order:
    - Process A takes 5 seconds, B takes 3 seconds, and C takes 2 seconds. They will run in sequence: A, then B, then C.

- **Shortest Job Next (SJN):**
  - This algorithm selects the process with the shortest execution time.
  - Example: If Process A needs 4 seconds, Process B needs 3 seconds, and Process C needs 1 second, the CPU will schedule C first, then B, and finally A.

- **Round Robin (RR):**
  - Round Robin assigns a fixed time slice (quantum) to each process in the queue, cycling through them in a circular order.
  - Example: With a time quantum of 2 seconds, processes will get 2 seconds each in turn. After 2 seconds, the CPU moves to the next process in line.

- **Priority Scheduling:**
  - This approach assigns a priority to each process, and the CPU selects the process with the highest priority (smallest numerical value).
  - Example: If processes have priorities 1, 3, and 2, the scheduler will execute the process with priority 1 first, then 2, then 3.

4. Context Switching

When the CPU scheduler decides to switch from one process to another, a **context switch** occurs. This means the OS saves the current process's state and loads the next process's state. Context switching allows multitasking but can introduce overhead, as switching takes time and resources.

Summary

CPU scheduling ensures efficient CPU use, enhances system performance, and maintains responsiveness. The choice of scheduling algorithm impacts system performance, with different algorithms suited to different types of workloads.

Aims of Scheduling

CPU scheduling aims to balance multiple goals to create an efficient, fair, and responsive computing environment. Each aim helps address specific system performance metrics and user experience. Here's a closer look at each aim:

1. Fairness

- **Objective**: Ensure all processes get a **fair share** of CPU time, so no process dominates the **CPU** while others are left waiting excessively.
- **Details**:
  - A fair scheduling system avoids **starvation**—a situation where lower-priority processes or those that require fewer resources do not get access to the CPU.
  - Example: A Round Robin algorithm with equal time slices helps distribute CPU time fairly among all processes.

2. Efficiency

- **Objective**: **Maximize CPU utilisation**, aiming to keep the CPU busy close to 100% of the time, ensuring minimal idle time.
- **Details**:
  - Efficient scheduling reduces the periods when the CPU is idle, meaning the OS effectively leverages hardware resources.
  - Example: Multi-level feedback queues and priority scheduling algorithms aim to maximize efficiency by prioritizing shorter or high-priority tasks.

3. Response Time

- **Objective**: Ensure **consistent response times** and **minimize the time** between a process's request for CPU time and its first execution.
- **Details**:
  - For interactive systems, quick and predictable response times are essential to improve user experience.
  - Example: Round Robin scheduling, with a small time quantum, can provide better response times for interactive processes.

4. Turnaround Time

- **Objective**: **Minimize the total time** between job submission and job completion, ensuring tasks finish quickly.
- **Details**:
    - This includes reducing waiting time in the ready queue and execution time on the CPU, which is especially crucial for batch processing.
    - Example: Shortest Job Next (SJN) is effective for minimizing turnaround time, as it prioritizes shorter tasks, helping them finish sooner.

5. Throughput

- **Objective**: **Maximize** the number of **processes completed** within a given time period, ensuring the system handles a high volume of work efficiently.
- **Details**:
    - Higher throughput indicates a productive and efficient system. Achieving this requires balancing fast and slow processes to avoid long delays.
    - Example: High-throughput algorithms prioritize process completion, such as Priority Scheduling, which can complete tasks based on priority and complexity.

Summary

By balancing these aims, the scheduling system strives to meet user expectations, optimize hardware usage, and handle various types of workloads effectively. Different scheduling algorithms emphasize different aims, making the choice of algorithm essential for specific OS requirements.

Preemptive Scheduling

Preemptive scheduling is a method where the operating system (OS) **can interrupt a currently running process, suspending its execution to give the CPU to another process**. This approach is ideal for multitasking environments because it ensures that high-priority or time-sensitive tasks can access the CPU as needed, improving system responsiveness and balancing workload. Here's a detailed look at preemptive scheduling and the specific algorithms that use it:

1. Characteristics of Preemptive Scheduling

- **Temporary Suspension**: A running process can be paused and moved back to the ready queue before it completes, allowing another process to use the CPU.
- **Responsive CPU Allocation**: The scheduler can reclaim the CPU from a process if a higher-priority process or an interactive task requires immediate attention.
- **Control Mechanism**: Processes release the CPU when they receive a command from the OS (such as reaching the time quantum in Round Robin or when a higher-priority process arrives).

2. Preemptive Scheduling Algorithms

Several scheduling algorithms rely on preemption to balance efficiency, responsiveness, and fairness. Here are three common ones:

- **Round Robin (RR)**
  - **How It Works**: In Round Robin scheduling, each process receives a fixed time slice (quantum) to execute. If a process doesn't complete within its allocated time, it is preempted and moved to the back of the ready queue, allowing the next process to run.
  - **Advantages**: Round Robin is fair, as each process gets an equal opportunity to use the CPU. It also reduces response time for interactive systems, as processes cycle through in short intervals.
  - **Example**: If there are three processes (A, B, C) and the time quantum is 2 seconds, each process runs for 2 seconds in turn. If Process A is not complete after 2 seconds, it's preempted, allowing Process B to start.

- **Multilevel Queue (MLQ)**
  - **How It Works**: In Multilevel Queue scheduling, processes are divided into different queues based on priority or process type (e.g., foreground vs. background). Each queue can have its own scheduling algorithm, and the CPU is assigned based on the priority of each queue. Processes in higher-priority queues preempt those in lower-priority queues.
  - **Advantages**: It efficiently handles processes with distinct requirements by isolating them in different queues, allowing fine-grained control over CPU allocation.
  - **Example**: A system might have three queues—high priority, medium priority, and low priority. High-priority tasks (such as real-time processes) can preempt tasks in lower-priority queues.

- **Multilevel Feedback Queue (MLFQ)**
    - **How It Works**: The Multilevel Feedback Queue scheduling algorithm allows processes to move between different priority queues based on their behavior and requirements. A new process starts in a higher-priority queue and, if it does not finish within a set time, is moved to a lower-priority queue. Processes in higher-priority queues preempt those in lower-priority ones.
    - **Advantages**: MLFQ adapts to the needs of processes by dynamically adjusting their priority. Interactive or shorter tasks get more CPU time, while CPU-bound tasks gradually receive less priority.
    - **Example**: A process that completes quickly remains in a high-priority queue for quick turnaround. In contrast, a long-running process is gradually demoted, which balances CPU usage between shorter and longer tasks.

3. Benefits of Preemptive Scheduling

- **Improved Responsiveness**: Preemptive scheduling is ideal for systems requiring quick response times, such as interactive environments or real-time systems.
- **Efficient Multitasking**: By allowing the OS to interrupt processes, preemptive scheduling manages CPU time effectively across multiple tasks.
- **Fair Resource Distribution**: Since processes cannot monopolize the CPU indefinitely, preemptive scheduling promotes fairness, avoiding issues like starvation.

Summary

Preemptive scheduling is key to managing multitasking effectively, particularly in systems requiring high responsiveness. Algorithms like Round Robin, Multilevel Queue, and Multilevel Feedback Queue leverage preemption to handle various workloads, balancing efficiency, fairness, and responsiveness.

Non-Preemptive Scheduling

Non-preemptive scheduling is a CPU scheduling approach where **a process retains the CPU until it voluntarily releases it upon completion**. In this approach, once a process begins executing, it is not interrupted, even if other processes arrive in the ready queue with higher priority or shorter execution time. Non-preemptive scheduling ensures each process runs to completion but may sometimes lead to inefficiencies in multitasking environments.

1. Characteristics of Non-Preemptive Scheduling

- **CPU Control**: Once a process gains control of the CPU, it keeps it until it finishes. The operating system will not interrupt a running process to assign the CPU to another waiting process.
- **Voluntary CPU Release**: Processes release the CPU only when they complete or need I/O. They are not preempted by other processes.
- **Simpler Scheduling**: Non-preemptive scheduling is less complex than preemptive scheduling since there are no context switches caused by process interruptions.

2. Non-Preemptive Scheduling Algorithms

Here are three common non-preemptive scheduling algorithms:

- **First-Come, First-Served (FCFS) or First-In, First-Out (FIFO)**
    - **How It Works**: Processes are scheduled in the order they arrive in the ready queue. The first process in the queue is given the CPU and allowed to run until completion before the next process begins.
    - **Advantages**: FCFS is simple and easy to implement, providing fairness by serving processes in the order of arrival.
    - **Disadvantages**: Long processes can cause delays for shorter ones, leading to the "convoy effect," where many short jobs wait behind a long-running job.
    - **Example**: If Process A arrives first and takes 5 seconds, Process B arrives second and takes 3 seconds, and Process C arrives third and takes 2 seconds, the order of execution will be A → B → C.

- **Shortest Job First (SJF)**
    - **How It Works**: SJF selects the process with the shortest execution time from the ready queue. Once selected, the process runs to completion before the CPU is assigned to the next shortest job.
    - **Advantages**: SJF minimizes average waiting time, as shorter jobs are given priority, making it efficient for batch processing.
    - **Disadvantages**: SJF can cause **starvation** for longer jobs if there is a continuous stream of shorter jobs arriving in the queue.
    - **Example**: If Process A requires 6 seconds, Process B requires 2 seconds, and Process C requires 3 seconds, the CPU will execute B → C → A based on their job durations.

- **Priority Scheduling**
    - **How It Works**: Each process is assigned a priority, and the CPU is allocated to the process with the highest priority (usually the smallest numerical value). In a non-preemptive priority algorithm, once a process starts, it will complete before the CPU moves to another process, even if a higher-priority process arrives.
    - **Advantages**: Priority scheduling allows urgent tasks to be prioritized, which can be useful for applications requiring specific execution orders.
    - **Disadvantages**: Lower-priority processes may suffer from **starvation** if higher-priority processes keep arriving. Aging techniques can be used to gradually increase the priority of waiting processes, reducing starvation.
    - **Example**: Suppose Process A has priority 2, Process B has priority 1, and Process C has priority 3. The order of execution will be B → A → C.

3. Benefits of Non-Preemptive Scheduling

- **Predictability**: Non-preemptive scheduling provides predictable execution, as once a process starts, it will complete without interruption.
- **Simplicity and Low Overhead**: Non-preemptive scheduling is easier to implement than preemptive scheduling and involves less context-switching overhead.
- **Useful for Certain Applications**: It's suitable for batch processing and applications where interruptions would disrupt processing, such as in embedded systems.

4. Drawbacks of Non-Preemptive Scheduling

- **Potential Delays**: Non-preemptive scheduling can lead to long waiting times, especially if long or low-priority tasks dominate the CPU.
- **Inefficiency in Multitasking Environments**: Non-preemptive scheduling may not handle interactive tasks well, as it lacks the flexibility to switch to time-sensitive or higher-priority tasks promptly.

Summary

Non-preemptive scheduling suits environments where simplicity and predictability are essential. However, its limitations in handling interactive tasks and balancing short and long processes make it less ideal for modern multitasking systems.

Quick Review Questions

Question 1: Why is CPU scheduling important?

Answer: CPU scheduling is crucial because it optimizes CPU utilization by keeping the CPU busy and minimizing idle time. This helps manage multitasking, balancing workload among processes, reducing response time, and ensuring fair resource allocation. Effective CPU scheduling improves system efficiency, responsiveness, and user experience.

Question 2: What is the difference between preemptive and non-preemptive algorithms?

Answer:

Preemptive Algorithms: These algorithms allow a running process to be interrupted and temporarily suspended so that the CPU can switch to another process, especially if it has higher priority or needs immediate attention.

Non-Preemptive Algorithms: In non-preemptive algorithms, a process keeps the CPU until it either completes or voluntarily releases it. Once a process starts, it is not interrupted by other processes.

Question 3: Give 2 examples each for a preemptive and non-preemptive algorithms.

Answer:

Preemptive Algorithms:

- *Round Robin (RR)*: Processes receive a fixed time slice in turn, and the CPU switches between them in a cyclic order.
- *Multilevel Feedback Queue (MLFQ)*: Processes are assigned to different priority queues and can move between queues based on behavior and needs, allowing for dynamic preemption.

Non-Preemptive Algorithms:

- *First-Come, First-Served (FCFS)*: Processes are executed in the order they arrive, with no interruptions.
- *Shortest Job First (SJF)*: The process with the shortest execution time is selected and runs to completion without preemption.

Calculation Keywords

1. CPU Utilization:

- **Definition**: The percentage of time the CPU is actively working on processes.
- **Formula**: CPU Utilization = $(1 - \frac{CPU\ Idle\ Time}{Total\ Time}) \times 100\%$

2. Throughput:

- **Definition**: The number of processes completed per unit of time.
- **Formula**: Throughput = $\frac{Number\ of\ Process\ Completed}{Total\ Time}$

3. Turnaround Time:

- **Definition**: The total time taken from the submission of a process to its completion.
- **Formula**: Turnaround Time = Completion Time − Arrival Time

4. Average Turnaround Time:

- **Definition**: The average of the turnaround times for all processes.
- **Formula**: Average Turnaround Time = $\frac{\sum Turnaround\ Times\ of\ All\ Processes}{Total\ Time}$

5. Waiting Time:

- **Definition**: The total time a process spends in the ready queue waiting for the CPU.
- **Formula**: Waiting Time = Turnaround Time − Burst Time

6. Average Waiting Time:

- **Definition**: The average waiting time for all processes.
- **Formula**: Average Waiting Time = $\frac{\sum Turnaround\ Times\ of\ All\ Processes}{Total\ Number\ of\ Processes}$

7. Response Time:

- **Definition**: The time from when a process is submitted until the first time it starts executing.
- **Formula**: Response Time = First Start Time − Arrival Time

8. Burst Time:

- **Definition**: The actual time required by a process to complete on the CPU (without interruptions).

Round Robin

Round Robin (RR) is a classic CPU scheduling algorithm particularly suited for time-sharing and multitasking systems. It is one of the fairest scheduling algorithms because it gives each process an equal share of CPU time in a cyclic manner. Here are the key features and how it operates:

1. Key Features of Round Robin Scheduling

- **Fair and Simple**:
  - Round Robin is one of the **oldest** and most straightforward scheduling algorithms. Each process receives equal CPU time, making it fair for all tasks in the queue.

- **Designed for Time-Sharing Systems**:
  - It is ideal for time-sharing systems where multiple users or applications share CPU time. This approach ensures a responsive user experience by regularly cycling between processes.

- **Time Quantum (or Time Slice)**:
  - A fixed time quantum (or time slice) is defined, which is the maximum time a process can hold the CPU before it is interrupted. If a process completes within this time, it releases the CPU voluntarily. If it doesn't, it is preempted and placed at the end of the ready queue.

- **Circular Queue (Ready Queue)**:
  - The ready queue operates in a circular manner, treating all processes cyclically. After a process uses its time slice, it goes to the back of the queue, allowing the next process in line to run.

- **FIFO Queue Structure**:
  - The ready queue in Round Robin is organized as a First-In, First-Out (FIFO) queue, where processes are added at the end of the queue as they arrive or complete a time quantum, maintaining the order of execution.

2. How Round Robin Works

- Processes enter the ready queue in the order they arrive.
- The CPU scheduler picks the first process from the queue and assigns it the CPU for a duration equal to the time quantum.
- If the process completes within the time quantum, it releases the CPU, and the scheduler moves to the next process in line.
- If the process does not finish within the time quantum, it is preempted and placed at the end of the queue, allowing the next process to run.
- This cycle continues, with each process receiving a time slice repeatedly until all processes are complete.

3. Example of Round Robin Scheduling

Suppose we have three processes, A, B, and C, with burst times of 4, 5, and 7 seconds, respectively, and a time quantum of 2 seconds.

- **First cycle**: Process A gets the CPU for 2 seconds, then moves to the end of the queue. Remaining burst time for A is 2 seconds.
- **Second cycle**: Process B gets 2 seconds, moves to the end. Remaining burst time for B is 3 seconds.
- **Third cycle**: Process C gets 2 seconds, moves to the end. Remaining burst time for C is 5 seconds.

This cycle repeats until each process completes its required burst time. Round Robin provides fairness and prevents starvation, as each process is guaranteed CPU time at regular intervals.

4. Advantages of Round Robin

- **Fairness**: Each process gets an equal share of CPU time.
- **Reduced Starvation**: Processes cannot monopolize the CPU, preventing longer jobs from blocking shorter ones.
- **Enhanced Responsiveness**: Time-sharing allows interactive tasks to receive frequent CPU time, improving responsiveness.

5. Disadvantages of Round Robin

- **High Context Switching**: Frequent switching between processes increases context-switching overhead.
- **Dependent on Time Quantum**: If the time quantum is too short, excessive context switches occur. If it's too long, responsiveness may suffer, and it starts resembling First-Come, First-Served (FCFS) scheduling.

Round Robin is widely used in environments where multiple users or applications need responsive CPU access, like in multitasking operating systems.

6. Example 1

Time Slice = 2 Milliseconds	
Process	**Burst Time (Milliseconds)**
P1	**10**
P2	**1**
P3	**2**
P4	**1**
P5	**5**

Step 1: Gantt Chart

The Gantt chart shows the order and timing for each process based on a time slice of 2 milliseconds. The sequence is:

- P1 runs for 2 ms
- P2 runs for 1 ms (completes in 1 time slice)
- P3 runs for 2 ms (completes in 1 time slice)
- P4 runs for 1 ms (completes in 1 time slice)
- P5 runs for 2 ms
- Then it cycles back to P1, P5, and P1 again until P1 completes.
- The total time for all processes is 19 ms.

Step 2: Waiting Times

The waiting time for each process (T_w) is calculated as the time spent waiting in the queue before it is completed.

- **T_w(P1) = (0 + 6 + 2 + 1) = 9**
- **T_w(P2) = 2**
- **T_w(P3) = 3**
- **T_w(P4) = 5**
- **T_w(P5) = (6 + 2 + 2) = 10**

Average Waiting Time

The average waiting time is calculated by summing the individual waiting times and dividing by the number of processes.

$$T_w(average) = \frac{9 + 2 + 3 + 5 + 10}{5} = 5.8ms$$

Summary

- **Average Waiting Time** = 5.8 ms
- This example demonstrates how Round Robin scheduling fairly distributes CPU time across processes with minimal waiting time for shorter processes.

Step 3: Calculate turnaround times and average turnaround time

Turnaround Time (T_T)

The turnaround time for each process is calculated as:

$$T_T = T_w + T_B$$

where:

- TWT_WTW is the waiting time.
- TBT_BTB is the burst time (execution time) of the process.

Using the values provided:

- T_T**(P1) = 9 + 10 = 19ms**
- T_T**(P2) = 2 + 1 = 3ms**
- T_T**(P3) = 3 + 2 = 5ms**
- T_T**(P4) = 5 + 1 = 6ms**
- T_T**(P5) = 10 + 5 = 15ms**

Average Turnaround Time

The average turnaround time is calculated by adding up the individual turnaround times and dividing by the number of processes.

$$T_T(average) = \frac{19 + 3 + 5 + 6 + 15}{5} = \frac{48}{5} = 9.6ms$$

Summary

- **Average Turnaround Time** = 9.6 ms
- This calculation confirms that each process's total time in the system, from arrival to completion, is fairly balanced across processes in round-robin scheduling.

In the Round Robin scheduling algorithm, the size of the **time quantum** (or **time slice**) is a critical factor that impacts the overall performance. Here's a breakdown of the key points regarding time quantum and its influence on Round Robin scheduling:

Average Waiting Time:

- In Round Robin, the **average waiting time** can be relatively **long** compared to some other scheduling algorithms, particularly when there are many processes. Since each process must wait for its turn in a circular queue, it might take a while before a process gets another chance to execute, especially in a heavily loaded system.

Time Quantum Size (10 – 100 milliseconds):

- The **time quantum** is typically set between 10 and 100 milliseconds, balancing between too frequent context switches and reasonable responsiveness.
- Choosing the right time quantum is essential: if it's too small, excessive context switching can degrade performance. If it's too large, the algorithm behaves more like **First-Come, First-Served (FCFS)**, reducing interactivity.

Context Switching Overhead:

- Every time the CPU switches between processes, there's a small overhead associated with saving and loading process states, known as **context switching**. During this time, the CPU is not executing any process, which can slightly reduce CPU efficiency.
- Since the time quantum clock continues to run during context switching, frequent switches can lead to reduced CPU time for actual processing.

Time Quantum Too Short:

- When the time quantum is set too short, the CPU will frequently switch between processes, leading to **excessive context switches**.
- This reduces CPU efficiency, as more time is spent on switching than on executing processes.

Time Quantum Too Long:

- If the time quantum is set too long, **response time** for short, interactive processes may suffer.
- In this case, Round Robin scheduling will start to resemble FCFS, where longer processes can monopolize the CPU, resulting in **delayed responses** for shorter tasks.

Summary

The **performance** of Round Robin is highly sensitive to the time quantum:

- **Optimal Quantum**: Strikes a balance between minimizing context switches and maintaining good response times.
- **Too Short**: Leads to high context-switching overhead, reducing CPU efficiency.
- **Too Long**: Causes poor responsiveness for short, interactive requests, making Round Robin less effective for time-sharing systems.

Selecting the right time quantum requires understanding the workload of the system. Systems with many interactive processes might benefit from a shorter time quantum, while systems with fewer, longer tasks might perform better with a longer one.

Multilevel Queue

The **Multilevel Queue Scheduling** algorithm is a method where processes are classified into different/distinct groups, each with its own queue. This approach is typically used in systems where different types of processes need to be managed separately based on their characteristics or priority. Here's a breakdown of how Multilevel Queue Scheduling works:

1. Key Features of Multilevel Queue Scheduling

- **Process Classification**:
  - Processes are classified into different groups based on characteristics such as **priority**, **process size**, or **type of process** (e.g., system processes, interactive tasks, batch jobs).
  - Each group of processes is assigned to a separate queue, ensuring that similar types of processes are managed together.

- **Separate Ready Queues**:
  - Instead of a single ready queue, the system has multiple queues, each handling a different group of processes.
  - For example:
    - **Foreground queue** for interactive processes
    - **Background queue** for batch processes
    - **System queue** for system-level tasks

- **Permanent Queue Assignment**:
  - Once a process is classified and assigned to a queue, it remains in that queue permanently. This assignment is typically based on the process's **initial characteristics**.
  - This fixed assignment ensures that processes are consistently managed according to their specific needs (e.g., higher priority for system processes).

- **Distinct Scheduling Algorithms**:
  - Each queue can have its own **scheduling algorithm** tailored to the type of processes it handles.
  - For example:
    - The foreground queue might use **Round Robin** for fast, responsive scheduling.
    - The background queue might use **First-Come, First-Served (FCFS)** or **Shortest Job First (SJF)** to prioritize efficiency over response time.

- **Priority Levels Among Queues**:
  - Queues are organized with a priority hierarchy. **Higher-priority queues** have precedence over **lower-priority queues**.
  - This means that processes in higher-priority queues (e.g., system processes) are given CPU time before those in lower-priority queues (e.g., batch jobs).

- **Time Slice Allocation**:
    - To manage CPU access among different queues, **time slices** can be allocated to each queue based on their priority.
    - For instance, a high-priority queue might receive more CPU time, while a lower-priority queue receives less time, possibly in proportion to the queue's importance.

2. Example of Multilevel Queue Scheduling

Imagine a system with three types of queues:

- **System Queue** (highest priority) for OS and system-level tasks, scheduled with FCFS.
- **Interactive Queue** for user tasks, scheduled with Round Robin.
- **Batch Queue** (lowest priority) for background jobs, scheduled with FCFS.
- **Execution Order**:
    - Processes in the System Queue are given first access to the CPU.
    - If there are no processes in the System Queue, the CPU moves to the Interactive Queue.
    - Finally, if both the System and Interactive Queues are empty, the CPU will execute processes from the Batch Queue.

3. Advantages of Multilevel Queue Scheduling

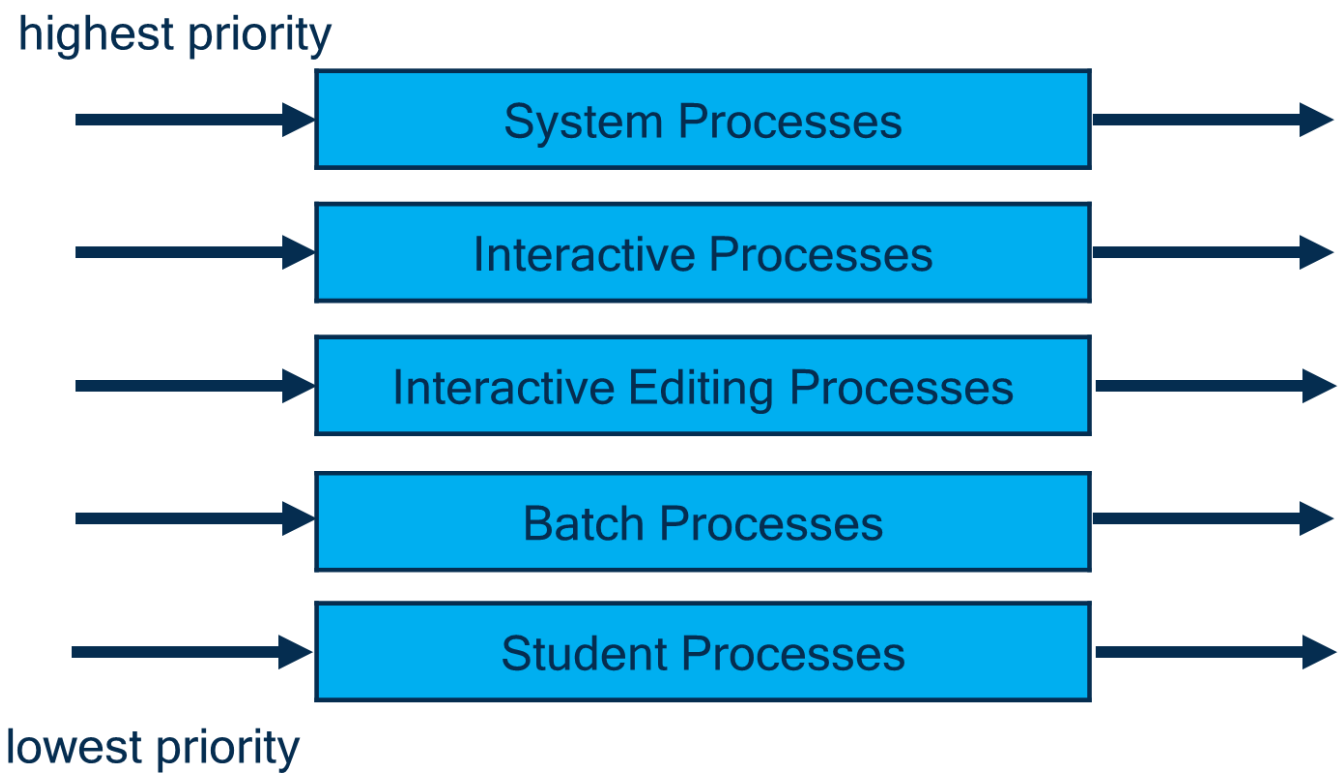
- **Process Separation**: Different types of processes are managed separately, making it easier to handle specific requirements for each group.
- **Priority Management**: System-critical processes or interactive tasks get CPU time sooner, ensuring higher responsiveness for essential tasks.
- **Custom Scheduling**: Each queue can use a scheduling algorithm that best suits its processes, allowing a mix of scheduling approaches.

4. Disadvantages of Multilevel Queue Scheduling

- **Rigidity**: Once assigned to a queue, processes cannot move between queues. This can be inefficient if the process characteristics change during runtime.
- **Starvation**: Lower-priority queues may experience starvation, as high-priority queues are served first and may dominate CPU time.
- **Complexity**: Managing multiple queues with different algorithms and priorities adds complexity to the system.

Multilevel Queue Scheduling is commonly used in environments with well-defined process categories, like operating systems that handle system, interactive, and background tasks separately. This approach balances the needs of different types of processes by ensuring priority handling where needed, but can lead to inefficiencies if not carefully managed.

5. Multilevel Queue Scheduling



The image illustrates a **Multilevel Queue Scheduling** model in an operating system. This scheduling method is used when processes are divided into different categories, each with its own priority and queue. Let's break down the elements:

5A. Explanation of the Queues

- **System Processes**:
  - **Highest Priority**.
  - These processes are essential for the operating system's core functionality, like managing hardware, memory, and critical resources.

- **Interactive Processes**:
  - Second-highest priority.
  - Includes processes that require immediate user interaction, such as command-line inputs or actions taken by system administrators.

- **Interactive Editing Processes**:
  - Medium-high priority.
  - Typically used by applications where users directly interact with content, like text editors and design software, requiring fairly responsive feedback.

- **Batch Processes**:
    - Medium-low priority.
    - Processes that do not require immediate user interaction and can be executed in the background, like scheduled tasks and maintenance jobs.

- **Student Processes**:
    - **Lowest Priority**.
    - Often reserved for less critical tasks, experimental processes, or lower-priority applications.

5B. How Multilevel Queue Scheduling Works

In this model:

- **Separate Queues**: Each process type has its dedicated queue, and the queues are arranged based on priority.
- **Fixed Priority**: Processes in higher-priority queues (like System Processes) are given preference over those in lower-priority queues (like Batch or Student Processes).
- **No Process Migration**: Processes are fixed in their respective queues and are not moved to other queues.
- **Scheduling Within Queues**: Each queue may use its own scheduling algorithm (e.g., Round Robin for Interactive Processes and First-Come-First-Served for Batch Processes).

This type of scheduling is useful for systems with varied types of workloads, allowing for efficient allocation of CPU time according to process needs and priority.

Multilevel Feedback Queue

The **Multilevel Feedback Queue (MLFQ)** scheduling algorithm is an advanced version of multilevel queue scheduling. In MLFQ, processes are allowed to **move between queues** based on their CPU usage and behavior, providing the flexibility that can improve both system responsiveness and fairness. Here's a breakdown of how MLFQ works:

1. Key Features of Multilevel Feedback Queue Scheduling

- **Dynamic Process Movement**:
  - Unlike in a standard multilevel queue where a process is assigned to a specific queue and remains there, **MLFQ allows processes to move between queues**.
  - This movement is based on how much CPU time a process uses, its behavior, and priority adjustments made by the system.

- **Queue Separation by CPU Burst Time**:
  - Queues are typically organized with different priorities and scheduling algorithms based on **CPU burst time** requirements.
  - For example:
    - High-priority queues may be designed for short burst (interactive) processes.
    - Lower-priority queues may handle longer-running (batch) jobs that don't need immediate CPU access.

- **Promotion and Demotion of Processes**:
  - **High CPU Usage → Lower Priority**: If a process uses too much CPU time, it is **moved to a lower-priority queue**. This prevents CPU-bound processes from monopolizing CPU time, allowing interactive or shorter processes to receive faster responses.
  - **Starvation Prevention**: If a process waits too long without CPU access (also known as starvation), it may be **moved to a higher-priority queue**. This ensures fairness, preventing lower-priority processes from being perpetually delayed.

- **Feedback Mechanism**:
  - The "feedback" in MLFQ refers to the system's ability to **adjust the priority of a process based on its behavior**.
  - The scheduling system continuously monitors process behavior and dynamically adjusts queue assignments to balance CPU usage across all processes.

- **Time Quantum**:
  - Each queue can have a different **time quantum**.
  - High-priority queues may have shorter time slices to quickly serve interactive tasks.
  - Lower-priority queues may have longer time slices to handle longer tasks efficiently with fewer context switches.

2. Example of Multilevel Feedback Queue Scheduling

Consider a system with three queues:

- **Queue 1 (highest priority)**: Shortest time quantum for interactive tasks, using Round Robin.
- **Queue 2**: Medium time quantum, using Round Robin for medium-priority tasks.
- **Queue 3 (lowest priority)**: Long time quantum for CPU-bound batch processes, using FCFS.

- A new process begins in **Queue 1** and is given a short time slice.
- If the process does not complete within this slice, it is moved to **Queue 2**, where it receives a longer time slice.
- If it still requires more CPU time after completing Queue 2's time slice, it moves to **Queue 3**.
- **Starvation Prevention**: If a process waits too long in Queue 3, it may be temporarily promoted to a higher-priority queue to ensure it eventually receives CPU time.

3. Advantages of Multilevel Feedback Queue Scheduling

- **Adaptability**: Processes that need more CPU time are moved to lower-priority queues, while interactive or shorter tasks receive higher priority for quicker responses.
- **Fairness**: Processes that experience starvation are promoted to higher-priority queues, ensuring they eventually get CPU access.
- **Balanced Performance**: Combines the benefits of different scheduling algorithms across queues, creating a balance between efficiency and responsiveness.

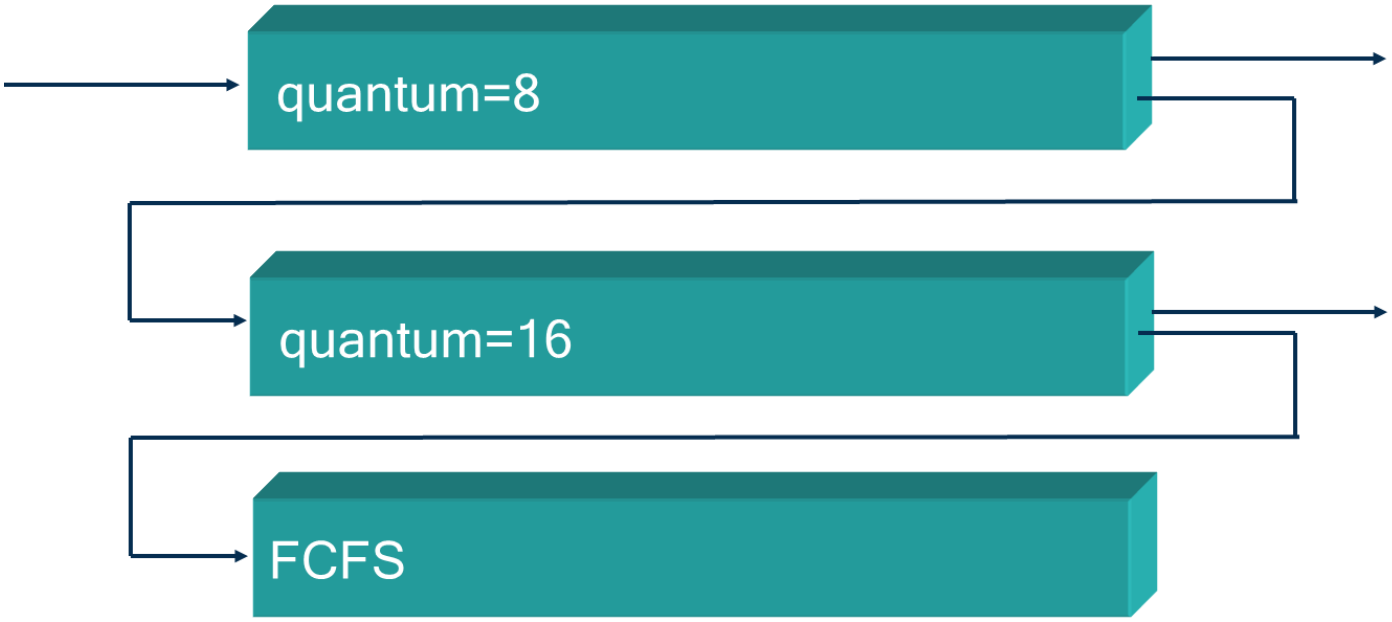
4. Disadvantages of Multilevel Feedback Queue Scheduling

- **Complexity**: MLFQ requires a sophisticated mechanism to monitor processes, adjust priorities, and move processes between queues, making it complex to implement.
- **Tuning Challenges**: Setting optimal parameters (like time quantum and conditions for promotion/demotion) can be challenging, as it depends on workload and system requirements.
- **Potential Overhead**: The feedback mechanism can introduce overhead, as the system constantly monitors process behavior and adjusts priorities.

Summary

The Multilevel Feedback Queue is a **flexible and dynamic scheduling algorithm** that adjusts process priorities based on real-time CPU usage. It's commonly used in systems where a mix of interactive and CPU-bound tasks exists, such as time-sharing environments, because it balances quick response times for interactive tasks with fair CPU access for longer processes.

5. Multilevel Feedback Queue Scheduling



The image illustrates a **Multilevel Feedback Queue Scheduling** system, a scheduling algorithm in operating systems that adjusts the priority of processes based on their behavior and requirements. This system dynamically moves processes between queues to improve responsiveness and efficiency.

5A. Breakdown of the Queues

- **Top Queue (Quantum = 8)**:
  - This queue has a time quantum of 8 units, meaning each process gets a maximum of 8 units of CPU time before being moved to the next queue **if it is not finished**.
  - It's typically used for processes that are expected to complete quickly or are highly interactive.

- **Middle Queue (Quantum = 16)**:
  - Processes that do not complete within 8 units in the first queue are moved to this queue, which has a longer time quantum of 16 units.
  - This queue serves processes that need more CPU time but are still somewhat responsive.

- **Bottom Queue (FCFS - First-Come-First-Served)**:
  - If a process does not finish within the 16-unit quantum in the middle queue, it is moved to the bottom queue.
  - This queue operates on a **First-Come-First-Served (FCFS)** basis, meaning each process runs until completion without any time quantum.
  - The bottom queue is generally for long-running or lower-priority processes, which do not need frequent CPU access.

5B. How Multilevel Feedback Queue Scheduling Works

- **Dynamic Priority Adjustment**: Processes are initially given higher priority and placed in queues with shorter time quantums. If they require more CPU time, they are demoted to lower-priority queues.
- **Response to Process Behavior**: Short, interactive tasks tend to complete in the higher-priority queues, while CPU-bound or long-running tasks gradually move to lower-priority queues.
- **Efficiency**: This system aims to give responsive processes faster access to the CPU while ensuring that longer-running processes still receive CPU time, albeit less frequently.

5C. Benefits of Multilevel Feedback Queue Scheduling

- **Adaptability**: Adjusts priorities based on how much time a process needs, balancing between responsiveness and fairness.
- **Fair CPU Allocation**: Prevents starvation by ensuring all processes eventually reach the FCFS queue, where they will complete without further demotion.

This scheduling method is commonly used in systems with a variety of process types, helping to balance interactive and non-interactive tasks effectively.

Quick Review Questions

Question 1. How does the round robin algorithm work?

Answer:

- **Round Robin** is a scheduling algorithm designed primarily for time-sharing systems. It works by giving each process a **fixed time slice or time quantum** to execute.
- The processes are placed in a **circular queue** (FIFO order), and the CPU cycles through the processes in this queue.
- When a process's time quantum expires, it is **moved to the back of the queue**, allowing the next process to use the CPU.
- If a process completes within its allocated time slice, it is removed from the queue. Otherwise, it waits for the next round.
- This approach ensures **fair CPU access** among processes, especially for interactive tasks, but the performance depends on the size of the time quantum.
- **Example**: If there are three processes (P1, P2, and P3) with a time quantum of 2 milliseconds, the CPU will give each process 2 milliseconds in a cyclic order until they all complete.

Question 2. What is the difference between the multilevel queue and multilevel feedback queue?

Answer:

- **Multilevel Queue Scheduling**: In this method, processes are assigned permanently to different **queues** based on characteristics like priority, process type, or memory needs. Each queue has its own scheduling algorithm, and processes do not move between queues. Higher-priority queues are serviced before lower-priority ones, which can lead to **starvation** for low-priority processes.
- **Multilevel Feedback Queue Scheduling**: This approach is more flexible. Processes can **move between queues** based on their CPU usage. If a process uses a lot of CPU time, it may be **demoted to a lower-priority queue**. Conversely, if a process waits too long without CPU access (starvation), it may be **promoted to a higher-priority queue**. This feedback mechanism makes it more adaptable and fair, ensuring a balance between responsiveness for short tasks and CPU efficiency for longer tasks.

In Summary:

- **Multilevel Queue**: Fixed queues, no movement between queues, and rigid prioritization.
- **Multilevel Feedback Queue**: Dynamic queue adjustment based on process behavior, allows movement between queues, and adapts to balance fairness and efficiency.

Follow Up Assignment

Question 1. List down some features of the round robin scheduling algorithm

Answer:

- **Time Quantum**: Each process is allocated a fixed time slice (or quantum) for execution. Once a process's time slice expires, it moves to the back of the queue.
- **Fairness**: It provides equal CPU time to all processes, making it suitable for time-sharing systems and ensuring that no single process monopolizes the CPU.
- **Circular Queue**: The ready queue is organized as a circular queue (FIFO order), cycling through processes repeatedly.
- **Minimizes Starvation**: By giving each process a chance to execute in a cyclic manner, round robin reduces the chance of starvation.
- **Performance Dependent on Time Quantum**: The algorithm's efficiency is highly affected by the size of the time quantum:
    - **Too Small**: Causes frequent context switching, leading to overhead and decreased CPU efficiency.
    - **Too Large**: Approaches First-Come, First-Served (FCFS) behavior, leading to longer wait times for short, interactive processes.
- **Suitable for Interactive Systems**: Round robin is particularly useful in environments where many processes need quick responses, such as in user-facing applications.

Question 2. What is turnaround time?

Answer:

- **Turnaround Time** is the total time taken for a process to complete after it is submitted. It includes the time the process spends waiting in the ready queue, executing on the CPU, and waiting for any I/O operations.
- **Formula**: Turnaround Time = Completion Time − Arrival Time
- Turnaround time is an important metric in CPU scheduling, as it reflects the time users wait for their tasks to finish. A low average turnaround time across processes indicates an efficient scheduling policy.

Question 3. Why are priorities used?

Answer:

- **Priorities** are used to determine the **order of process execution** in systems where different tasks have different levels of importance or urgency.

- **Improves Responsiveness**: Higher-priority tasks (e.g., system-critical or interactive processes) get faster access to the CPU, providing quick responses where needed.

- **Efficient Resource Allocation**: By giving preference to certain tasks, priority scheduling ensures that important jobs are completed in a timely manner.

- **Reduces Starvation for Important Processes**: Assigning higher priority to essential tasks prevents them from being delayed indefinitely by less critical processes.

- **Adaptability in Multilevel Feedback Queue**: In MLFQ, priorities can dynamically adjust based on process behavior, ensuring that both interactive and CPU-bound tasks receive appropriate CPU time.

FIFO / FCFS

- **Simplicity**: FCFS is the simplest CPU scheduling algorithm. It's straightforward to implement, as processes are handled in the exact order they arrive.

- **Scheduling Principle**: The process that requests the CPU first is allocated the CPU first. It follows a **"first in, first out" (FIFO)** approach, meaning that the first process in the queue is the first to execute.

- **Implementation**: The FIFO policy can be managed easily with a **FIFO queue**. As processes arrive, they are added to the back of the queue. The CPU executes each process sequentially from the front of the queue.

- **Average Waiting Time**: One of the downsides of FCFS is that the **average waiting time can be quite long**, especially when shorter processes are waiting behind a long-running process (known as the **convoy effect**). This situation can result in poor CPU and resource utilization.

- **Non-Preemptive Nature**: FCFS is a **non-preemptive algorithm**, meaning that once a process has been allocated the CPU, it keeps the CPU until it either completes execution or voluntarily releases the CPU (such as by making an I/O request).

1. Example of FCFS Scheduling

Consider three processes:

- **P1** with a burst time of 5 ms,
- **P2** with a burst time of 3 ms,
- **P3** with a burst time of 8 ms.

If they arrive in the order P1, P2, P3, FCFS will schedule them in the same order:

- P1 executes first, taking 5 ms.
- After P1 finishes, P2 starts and takes an additional 3 ms (total 8 ms from the start).
- Finally, P3 executes and completes at 16 ms.

2. Advantages of FCFS

- **Simple and Easy to Implement**: It requires minimal overhead, as it only involves managing a queue.
- **Fair**: Every process is given a chance to execute in the order it arrives, making it fair in terms of arrival time.

3. Disadvantages of FCFS

- **Long Average Waiting Time**: Processes can experience long wait times, especially if a shorter process is waiting behind a longer process.
- **Convoy Effect**: The convoy effect occurs when shorter, faster processes are forced to wait for a longer process to finish, leading to inefficient CPU utilization.
- **Poor Performance for Interactive Systems**: FCFS is not ideal for interactive systems where quick response times are required. It's better suited for batch processing systems where tasks are completed in sequence without strict timing constraints.

4. Example 1

Process	Burst Time (Milliseconds)
P1	**10**
P2	**1**
P3	**2**
P4	**1**
P5	**5**

Step 1: Gantt Chart

The Gantt chart visually represents the execution order of processes over time:

- **Processes and their execution times**:
  - P1: 10 ms (from time 0 to 10)
  - P2: 1 ms (from time 10 to 11)
  - P3: 2 ms (from time 11 to 13)
  - P4: 1 ms (from time 13 to 14)
  - P5: 5 ms (from time 14 to 19)

Step 2: Calculate Waiting Times and Average Waiting Time

The **waiting time** for each process is the time it spends waiting before it begins execution.

- T_w(P1) = 0 (P1 starts immediately)
- T_w(P2) = 10 (P2 starts after P1 finishes)
- T_w(P3) = 11 (P3 starts after P2 finishes)
- T_w(P4) = 13 (P4 starts after P3 finishes)
- T_w(P5) = 14 (P5 starts after P4 finishes)

The **average waiting time** is calculated as:

$$T_w(average) = \frac{0 + 10 + 11 + 13 + 14}{5} = 9.6ms$$

Step 3: Calculate Turnaround Times and Average Turnaround Time

The **turnaround time** (TT) for each process is the total time taken from its arrival until completion. It is calculated as:

$$TT = T_w + T_B$$

where T_B is the burst time of the process.

- **For P1**: TT(P1) = T_w(P1) + T_B(P1) = 0 + 10 = 10ms
- **For P2**: TT(P2) = T_w(P2) + T_B(P2) = 10 + 1 = 11ms
- **For P3**: TT(P3) = T_w(P3) + T_B(P3) = 11 + 2 = 13ms
- **For P4**: TT(P4) = T_w(P4) + T_B(P4) = 13 + 1 = 14ms
- **For P5**: TT(P5) = T_w(P5) + T_B(P5) = 14 + 5 = 19ms

The **average turnaround time** can be calculated by averaging the individual turnaround times:

$$TT\ (average) = \frac{10 + 11 + 13 + 14 + 19}{5} = 13.4ms$$

Summary

- **Average Waiting Time**: 9.6 ms
- **Average Turnaround Time**: 13.4 ms

In FCFS scheduling, processes are handled in the order they arrive, which can lead to longer waiting times, especially if a long process arrives first.

Shortest Job First (SJF)

- **Scheduling Based on Burst Time**: The SJF scheduling algorithm selects the process with the **shortest CPU burst** (execution time) to execute next. This helps reduce the overall waiting time, as shorter tasks are prioritized.

- **CPU Assignment**: If the CPU is free, the process with the **smallest CPU burst** in the ready queue is selected for execution. This ensures that shorter jobs finish quickly, minimizing the time other processes spend waiting.

- **Tie-Breaking**: If two or more processes have the **same CPU burst length**, SJF uses **First-Come, First-Served (FIFO)** to break the tie, meaning the process that arrived first among the tied processes will be chosen.

- **Optimal Average Waiting Time**: One of the main advantages of SJF is that it provides the **minimum average waiting time** for a set of processes. This makes it an **optimal algorithm** in terms of minimizing average waiting time, which is beneficial for overall CPU efficiency.

- **Difficulty in Predicting Burst Time**: The main challenge with SJF is accurately predicting the **next CPU burst length** of each process, which can be difficult or impossible in certain scenarios. In real-time systems, this information is often estimated, which may reduce SJF's effectiveness.

- **Non-Preemptive and Preemptive Variants**:
    - **Non-Preemptive SJF**: Once a process starts, it cannot be interrupted until it finishes.
    - **Preemptive SJF (Shortest Remaining Time First - SRTF)**: If a new process arrives with a shorter burst time than the remaining time of the current process, it preempts the current process and starts execution.

1. Example of Non-Preemptive SJF Scheduling

Consider four processes with the following burst times:

- **P1**: 6 ms
- **P2**: 8 ms
- **P3**: 7 ms
- **P4**: 3 ms

If these processes arrive at the same time, SJF would schedule them in the following order:

1. **P4** (3 ms) - shortest burst time.
2. **P1** (6 ms) - next shortest.
3. **P3** (7 ms).
4. **P2** (8 ms).

This ordering minimizes the average waiting time for all processes compared to other scheduling methods.

2. Advantages of SJF

- **Optimal for Waiting Time**: SJF minimizes average waiting time for all processes, making it highly efficient in terms of CPU usage.
- **Ideal for Batch Processing**: SJF works well in batch-processing systems where the burst times of processes are known beforehand.

3. Disadvantages of SJF

- **Difficult to Implement in Real-Time**: Predicting the CPU burst length is challenging, especially in interactive or real-time systems.
- **Starvation of Long Processes**: SJF can lead to **starvation** for longer processes if shorter processes keep arriving, as longer processes may never get CPU time.
- **Not Suitable for Interactive Systems**: Because of the unpredictable nature of interactive tasks, SJF may not provide the responsiveness needed for user-facing applications.

Summary

The **Shortest Job First** algorithm is highly efficient for systems where burst times can be accurately predicted, offering the **minimum average waiting time** among scheduling algorithms. However, it is less suited for environments with unpredictable task durations, as it can lead to starvation and struggles with real-time responsiveness.

4. Example 1

Process Selection:

- In SJF scheduling, the process with the shortest CPU burst time is selected first. This approach minimizes the average waiting time.

Process	Burst Time (Milliseconds)
P1	**10**
P2	**1**
P3	**2**
P4	**1**
P5	**5**

Step 1: Gantt Chart

- A Gantt chart represents the timing of each process.
- In this example, processes P2, P4, P3, P5, and P1 execute sequentially, with P2 starting at time 0 and P1 finishing at time 19.

Step 2: Waiting Calculation

- Waiting time for each process is calculated as the time spent waiting in the queue before its first execution.
- For example:
  - T_w(P1) = 9 milliseconds
  - T_w(P2) = 0 milliseconds
  - T_w(P3) = 2 milliseconds
  - T_w(P4) = 1 millisecond
  - T_w(P5) = 4 milliseconds
- Average waiting time is computed by summing these values and dividing by the number of processes, resulting in 3.2 milliseconds.

Step 3: Turnaround Time Calculation

- Turnaround time (T_T) for each process is calculated as the sum of waiting time and burst time.
- For example:
  - T_T(P1) = 9 + 10 = 19 milliseconds
  - T_T(P2) = 0 + 1 = 1 millisecond
  - T_T(P3) = 2 + 2 = 4 milliseconds
  - T_T(P4) = 1 + 1 = 2 milliseconds
  - T_T(P5) = 4 + 5 = 9 milliseconds
- Average turnaround time is calculated similarly, yielding 7 milliseconds.

SJF Key Points:

- **Optimal Waiting Time**: SJF minimizes average waiting time when compared to other algorithms like FCFS.
- **Non-preemptive**: This example demonstrates non-preemptive SJF, where each process completes once it starts, and no other process interrupts it.
- **Challenge**: SJF requires knowledge of the next CPU burst length, which is often difficult to estimate in real-time systems.

Priority

1. Priority Scheduling Algorithm

- **Priority Assignment**: Each process is assigned a **priority value**. This value determines the order in which processes are scheduled for CPU allocation. Processes with **higher priority** are executed before processes with lower priority.

- **CPU Allocation**: The CPU is allocated to the process with the **highest priority**. If a higher-priority process arrives while the CPU is executing a lower-priority process (in the preemptive version), the current process may be interrupted to allow the higher-priority process to run.

- **Tie-Breaking**: If two or more processes have the **same priority**, **FIFO (First-Come, First-Served)** is used to decide the order among them, giving preference to the process that arrived first.

- **Priority Levels**: Priority levels can vary from high to low. Sometimes, a **lower numerical value indicates a higher priority** (for example, 0 could represent the highest priority).

- **Preemptive and Non-Preemptive Variants**:
    - **Preemptive Priority Scheduling**: The CPU will switch to a higher-priority process if it arrives while a lower-priority process is running.
    - **Non-Preemptive Priority Scheduling**: Once the CPU starts executing a process, it will complete that process even if a higher-priority process arrives.

- **Starvation Issue**: One major issue with priority scheduling is **starvation**, where low-priority processes may wait indefinitely if there is a continuous flow of higher-priority processes.

- **Aging Technique**: **Aging** is a solution to the starvation problem. It gradually increases the priority of a waiting process over time. This way, even a low-priority process will eventually get a chance to execute if it waits long enough.

2. Example of Priority Scheduling (Non-Preemptive)

Consider four processes with the following burst times and priorities:

- **P1**: Burst time = 8 ms, Priority = 2
- **P2**: Burst time = 4 ms, Priority = 0 (highest priority)
- **P3**: Burst time = 9 ms, Priority = 1
- **P4**: Burst time = 5 ms, Priority = 3

Assuming lower values indicate higher priority:

- **P2** runs first (highest priority).
- **P3** runs next (second highest priority).
- **P1** runs after P3.
- **P4** runs last (lowest priority).

3. Advantages of Priority Scheduling

- **Flexible Control Over Process Execution**: Allows critical or time-sensitive processes to be prioritized, ensuring they complete sooner.
- **Efficient for Real-Time Systems**: Processes with real-time needs can be assigned higher priority to ensure they meet timing constraints.

4. Disadvantages of Priority Scheduling

- **Starvation**: Lower-priority processes may never execute if there are always higher-priority processes waiting.
- **Complexity of Managing Priorities**: Requires careful management to avoid issues like starvation and ensure fair CPU usage.
- **Not Ideal for Interactive Systems**: If interactive tasks have lower priority, they may experience delays, affecting responsiveness.

Summary

The **Priority Scheduling** algorithm allows for flexible and controlled scheduling by assigning priorities to processes. It is well-suited for **real-time and critical applications**, where certain processes must complete quickly. However, **starvation** is a potential issue, which can be mitigated by techniques like **aging** to ensure all processes eventually get CPU time.

5. Example 1

Process Selection:

- In Priority Scheduling, each process is assigned a priority, and the CPU is allocated to the process with the highest priority.
- If two processes have the same priority, a secondary scheduling method, such as First-Come-First-Serve (FIFO), may be used.

Process	Burst Time (Milliseconds)	Priority
P1	**10**	**3**
P2	**1**	**1**
P3	**2**	**3**
P4	**1**	**4**
P5	**5**	**2**

Step 1: Gantt Chart

The Gantt chart shows the sequence of processes based on priority:

- The order is P4, P1, P3, P5, and P2.
- Each process is scheduled according to its priority, with lower numerical values indicating higher priority.
- Highest Priority: 5 > 4 > 3 > 2 > 1
- Lowest Priority: 1 > 2 > 3 > 4 > 5

Step 2: Waiting Time Calculation

- Waiting time (TWT_WTW) for each process is calculated as the time spent in the queue before execution starts.
- For example:
  - T_w(P1) = 1 ms
  - T_w(P2) = 18 ms
  - T_w(P3) = 11 ms
  - T_w(P4) = 0 ms
  - TT_w(P5) = 13 ms
- The average waiting time is calculated by summing these values and dividing by the total number of processes, resulting in an average waiting time of 8.6 milliseconds.

Step 3: Turn Around Time

$$T_T = T_w + T_B$$

where:

- T_T is the turnaround time,
- T_w is the waiting time (time spent in the queue before execution),
- T_B is the burst time (time required for execution).

T_T(P1) = T_w(P1) + T_B(P1) = 1 + 10 = 11 ms

T_T(P2) = T_w(P2) + T_B(P2) = 18 + 1 = 19ms

T_T(P3) = T_w(P3) + T_B(P3) = 11 + 2 = 13ms

T_T(P4) = T_w(P4) + T_B(P4) = 0 + 1 = 1ms

T_T(P5) = T_w(P5) + T_B(P5) = 13 + 5 = 18ms

Step 4: Average TAT

$$Avearge\ T_T = \frac{11 + 19 + 13 + 1 + 18}{5} = \frac{62}{5} = 12.4ms$$

Quick Review Questions

Question 1. Name three preemptive scheduling algorithms.

Answer:

- Round Robin (RR)
- Preemptive Priority Scheduling
- Shortest Remaining Time First (SRTF) (preemptive version of Shortest Job First)

Question 2. Which of the three is the easiest to implement?

Answer:

Round Robin is generally the easiest to implement among the three. It only requires a fixed time quantum and a circular queue to keep track of processes, making it simpler compared to algorithms that need complex calculations or priority management.

Follow Up Assignment

Question 1. FIFO on its own is a scheduling algorithm, however FIFO is also used in other scheduling algorithms. Name these algorithms.

Answer:

- **Shortest Job First (SJF)**: If two processes have the same CPU burst time, FIFO is used to determine which process to schedule first.
- **Priority Scheduling**: If two processes have the same priority, FIFO is used to break the tie and select the process that arrived first.

Question 2. Which non-preemptive algorithm is optimal and why?

Answer:

Shortest Job First (SJF) is considered the **optimal non-preemptive algorithm** because it minimizes the **average waiting time** for a set of processes. By always selecting the process with the shortest CPU burst time, SJF ensures that processes complete quickly, reducing the time that other processes spend waiting in the queue. This leads to an overall more efficient and responsive system.

Summary of Main Teaching Points

- A process is a unit of work in execution which requires resources.
- The state of a processes changes as it executes.
- A process control block represents a process in the operating system.
- Processes can execute concurrently or independently from each other.
- A thread is a basic unit of CPU utilisation.
- CPU scheduling is used to select a process from the ready queue and allocate this process CPU time.
- Scheduling algorithms can be pre-emptive or non-pre-emptive.
- Scheduling algorithms aim to achieve fairness, efficiency, maximising throughput and minimising turnaround time.
- Round robin, multilevel queues and multilevel feedback queues are examples of pre-emptive algorithms.
- Non-preemptive algorithms are FIFO, SJF and priority.
- SJF algorithm has a shorter average waiting time.
- Using priority scheduling may result in some processes experiencing starvation; aging is a method used to overcome this.

Test Yourself

Question 1: A process may be defined as

a) A set of instructions to be executed by a computer

b) A program in execution

c) A piece of hardware that executes a set of instructions

d) The main procedure of a program

Answer: (b) A program in execution - A process is specifically a program that is currently being executed by the CPU, with allocated resources such as memory and CPU time.

Question 2. A starvation-free job scheduling policy guarantees that no job waits indefinitely for service. Which of the following job-scheduling policies is starvation-free

a) Round-robin

b) Priority queuing

c) Shortest job first

d) Youngest job first

e) None of the above

Answer: (a) Round-robin scheduling is starvation-free because each process gets a fair share of CPU time within a defined time quantum, ensuring that every job eventually runs without indefinite waiting.

Question 3: For each of the following transitions between process states, indicate whether the transition is
possible or not possible?

- Ready ⟶ Running
- Ready ⟶ New
- Blocked ⟶ Ready
- Ready ⟶ Blocked

Answer:

Ready ⟶ Running: Possible. This occurs when the scheduler selects a process from the ready queue to execute on the CPU.

Ready ⟶ New: Not possible. The "New" state is for processes that have just been created, not yet admitted to the ready queue.

Blocked ⟶ Ready: Possible. This transition happens when a process completes an I/O operation or waits for a resource, making it eligible to enter the ready queue.

Ready ⟶ Blocked: Not possible. A process must first enter the Running state before it can go to the Blocked state.

Question 4. On a system with n number of CPU's what is the minimum number of processes that can be in the ready, running and blocked state?

Answer:

The minimum number of processes is:

- **Running:** nnn (since each CPU can execute one process at a time)
- **Ready:** 0 (it's possible to have no processes waiting to be scheduled)
- **Blocked:** 0 (it's possible to have no processes waiting on I/O or other resources)

Question 5. What is a process control block?

Answer:

- A Process Control Block (PCB) is a data structure maintained by the operating system to keep track of each process. It contains information like:
    - Process ID
    - Process state (e.g., ready, running, blocked)
    - Program counter
    - CPU registers
    - Memory management information
    - Accounting information
    - I/O status information

The PCB enables the operating system to save and restore the process's state, allowing for process scheduling and switching.

Question 6. Draw and briefly explain the process scheduling diagram.

Answer:

The process scheduling diagram typically involves the following states:

- **New**: A process is created and ready to be admitted to the system.
- **Ready**: The process is waiting to be assigned to the CPU.
- **Running**: The process is currently executing on the CPU.
- **Blocked (Waiting)**: The process is waiting for an I/O operation or other resource.
- **Terminated**: The process has completed its execution.

State Transitions:

- **New** → **Ready**: The process is admitted to the ready queue.
- **Ready** → **Running**: The scheduler selects the process for execution.
- **Running** → **Blocked**: The process requests I/O or waits on a resource.
- **Blocked** → **Ready**: The process completes the I/O operation and returns to the ready queue.
- **Running** → **Ready** (preemptive scheduling): The process is preempted and moved back to the ready queue.
- **Running** → **Terminated**: The process completes execution and is removed from the system.

Question 7. Differentiate between preemptive and non-preemptive scheduling.

Answer:

Preemptive Scheduling: Allows a process to be interrupted and moved back to the ready queue if a higher-priority process arrives or if the running process's time quantum expires (e.g., Round Robin, Preemptive Priority Scheduling). This enables better response time but may lead to more context switching.

Non-Preemptive Scheduling: Once a process starts executing, it cannot be interrupted and will continue until it completes or enters the blocked state (e.g., FCFS, SJF, Non-Preemptive Priority Scheduling). It reduces context switching but may lead to issues like starvation.

Question 8. Consider a workload with 5 jobs that each compute for an identical amount of time, x, and that perform no I/O. To minimize the average waiting time of the jobs, should a FCFS scheduler or a RR scheduler be used?

Answer: **FCFS scheduler** should be used.

With jobs of identical computation time and no I/O, FCFS will minimize the average waiting time since each job will execute sequentially without interruption. Round Robin, on the other hand, would introduce additional context switches, increasing the average waiting time due to time quantum overhead.

Question 9. Consider the following set of processes that arrive at time 0 with the length of CPU-burst time given in milliseconds (ms). Time Slice: 2ms

Process	Burst Time (ms)	Priority
P1	**6**	**3**
P2	**8**	**1**
P3	**7**	**2**
P4	**3**	**4**

i. Draw Gant charts to illustrate the execution of these processes using the following algorithm and find out the average waiting time for each of them.

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

Gantt Chart (Execution Timeline):

Since FCFS runs processes in the order of their arrival, the execution order is P1 ⟶ P2 ⟶ P3 ⟶ P4.

Time	0 - 6	6 - 14	14 - 21	21 - 24
Process	**P1**	**P2**	**P3**	**P4**

Waiting Time Calculation:

Waiting time is the time a process waits in the queue before getting executed.

Waiting Time (WT) For Each Process:

- WT(P1) = 0 ms (since P1 starts immediately)
- WT(P2) = 6 ms (P1 runs for 6 ms, then P2 starts)
- WT(P3) = 14 ms (P1 + P2 = 6 + 8 = 14 ms)
- WT(P4) = 21 ms (P1 + P2 + P3 = 6 + 8 + 7 = 21 ms)

Average Waiting Time (FCFS):

Average Waiting Time = $\frac{0+6+14+21}{4} = 10.25ms$

b. Shortest-Job-First(SJF)

Gantt Chart (Execution Timeline):

The burst times for the processes are:

- P1: 6 ms
- P2: 8 ms
- P3: 7 ms
- P4: 3 ms

Order of execution based on burst times (shortest first): P4 ⟶ P1 ⟶ P3 ⟶ P2.

Time	0 - 3	3 - 9	9 - 16	16 - 24
Process	P4	P1	P3	P2

Waiting Time Calculation:

- WT(P4) = 0 ms (P4 runs first)
- WT(P1) = 3 ms (P4 runs for 3 ms, then P1 starts)
- WT(P3) = 9 ms (P4 + P1 = 3 + 6 = 9 ms)
- WT(P2) = 16 ms (P4 + P1 + P3 = 3 + 6 + 7 = 16 ms)

Average Waiting Time (FCFS):

Average Waiting Time = $\frac{0+3+9+16}{4} = 7ms$

c. Priority (small no high priority)

Process Information (by priority):

P2 (Priority 1) ⟶ P3 (Priority 2) ⟶ P1 (Priority 3) ⟶ P4 (Priority 4)

Gantt Chart (Execution Timeline):

Order of execution based on priority: P2 ⟶ P3 ⟶ P1 ⟶ P4.

Time	0 - 8	8 - 15	15 - 21	21 - 24
Process	P2	P3	P1	P4

Waiting Time Calculation:

- WT(P2) = 0 ms (P2 runs first)
- WT(P3) = 8 ms (P2 runs for 8 ms, then P3 starts)
- WT(P1) = 15 ms (P2 + P3 = 8 + 7 = 15 ms)
- WT(P4) = 21 ms (P2 + P3 + P1 = 8 + 7 + 6 = 21 ms)

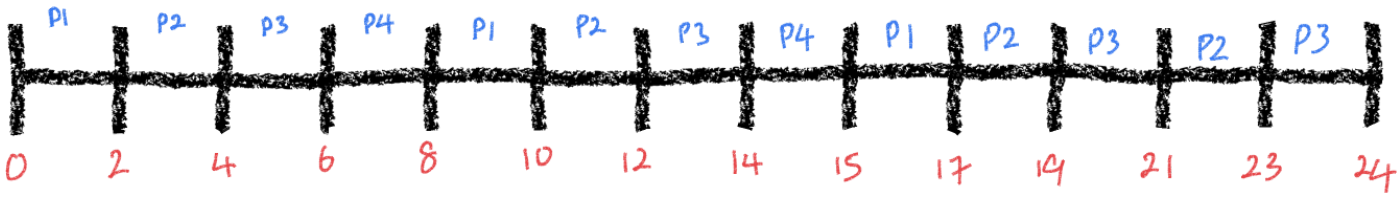
Average Waiting Time (FCFS):

Average Waiting Time = $\frac{0+8+15+21}{4} = 11ms$

d. Round Robin

- Each process is given a fixed time slice (2 ms here), after which it is moved to the back of the queue if not finished.
- Execution continues in a cyclic fashion until all processes are finished.

Gantt Chart (Execution Timeline):



Waiting Time Calculation:

- WT(P1) = 0 + 6 + 5 = 11ms
- WT(P2) = 2 + 6 + 5 +2 = 15 ms
- WT(P3) = 4 + 6 + 5 + 2 = 17ms
- WT(P4) = 6 + 6 = 12ms

Average Waiting Time (FCFS):

Average Waiting Time = $\frac{11+15+17+12}{4} = 13.75ms$

ii. What is the average waiting time if the processes arrive in the order P2,P3,P1,P4 using the FCFS?

Gantt Chart (Execution Timeline):

Since FCFS runs processes in the order of their arrival, the execution order is P1 ⟶ P2 ⟶ P3 ⟶ P4.

Time	0 - 8	8 - 15	15 - 21	21 - 24
Process	P2	P3	P1	P4

Waiting Time Calculation:

Waiting time is the time a process waits in the queue before getting executed.

Waiting Time (WT) For Each Process:

- WT(P1) = 15 ms
- WT(P2) = 0ms
- WT(P3) = 8 ms
- WT(P4) = 21 ms

Average Waiting Time (FCFS):

Average Waiting Time = $\frac{0+8+15+21}{4} = 11ms$