OS Module 2

Process Concept and Scheduling

Basic Terminologies & Definitions

1) Task

A task is a unit of work managed by an OS, which may involve one or more processes, such as downloading a file or baking a cake.

2) Program

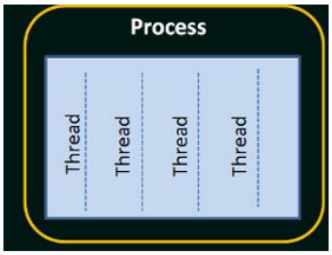
A program is a set of instructions that does nothing until run, like a recipe in a book or a web app. It may involve one or more processes to complete a task

3) Job

A job is a unit of work submitted by a user to the system, which can be interactive or batch and may involve multiple processes, like sending a print job or scheduling a file backup.

Process and Thread

- A process is an instance of a program in execution.
- It is a basic unit of work that can be scheduled and executed by the operating system.
- A thread is the unit of execution within a process.
- A process can have anywhere from just one thread to many threads.



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Process Mode

A process can run in two different modes: **user mode** and **kernel mode**. The operating system switches between these modes based on the nature of the code being executed and the operations being performed.

- User Mode: The process runs with limited privileges, typically to prevent it from performing harmful operations.
- Kernel Mode: The process runs with full access to all system resources and hardware, necessary for performing critical tasks.

Process Context

When a process is temporarily removed from the processor, its current state must be saved so that it can resume execution later without losing its place. This saved state is known as the **process context** and includes several key components:

1)Address Space

2)Stack Space

3)Virtual Address Space

4)Register Set Image: Registers are small, fast storage locations within the CPU. They hold important data needed for execution, including:

Program Counter (PC): The address of the next instruction to execute.

Stack Pointer (SP): The address of the top of the stack.

Program Status Word (PSW): Contains flags and status information about the process's state.

Instruction Register (IR): Holds the current instruction being executed.

General Processor Registers: Various registers used for general-purpose operations.

5)Current State of the Process

Event

In the context of operating systems, an **event** is an activity or occurrence that either happens or is anticipated to happen.

The events may cause the processes to change their state.

E.g. mouse click, file lock reset, etc.

Process Concept

1) Process Priority

In a multiprogramming system, each process is assigned a numerical priority value. This value indicates the process's relative importance, urgency, or value compared to other processes in the system.

2) Preemptive Scheduling

Preemptive scheduling allows the operating system to interrupt and take over a currently executing process in favor of another process, typically one with higher priority.

3) Non-Preemptive Scheduling

In non-preemptive scheduling, once a process starts executing, it cannot be interrupted or taken over by another process until it completes its execution or voluntarily yields control.

Process Control Block (PCB)

The Process Control Block (PCB) is has a detailed record for each process in an operating system. It stores all the information needed to manage and track a process. Here's what's inside a PCB and what each part does:

Identifiers

- **Process Identifier/Number**: This is a unique number given to each process. It helps the operating system recognize and keep track of different processes.

2. Processor State Information

- **Process State**: Shows what the process is currently doing. For example, it could be running (actively using the CPU), waiting (waiting for something like I/O to complete), or ready (waiting to be assigned to the CPU).
- **Program Counter (PC)**: This tells the operating system where the process left off by storing the address of the next instruction to be executed. It's like a bookmark for where the process was in its task.
- **CPU Registers**: These are small, fast storage locations in the CPU that hold temporary data for the process. Each process has its own set of these registers, and they store things like values being used by the process. The number and type of registers depend on the computer's architecture.

3. CPU Scheduling Information

- **Process Priorities**: Indicates how important the process is compared to others. Higher priority processes are usually executed first.
- **Scheduling Queue Pointers**: Points to where the process is in the scheduling queue, which helps the operating system decide when the process should run.
- **Other Scheduling Parameters**: Any additional details needed to manage the process's execution.

4. Memory-Management Information

- **Memory Allocated**: Shows how much memory has been set aside for the process.
- **Base and Limit Registers**: These registers help keep track of where the process's memory starts and ends, ensuring it doesn't access memory that doesn't belong to it.
- **Page Tables or Segment Tables**: These tables help manage how memory is organized and accessed by the process.

5. Accounting Information

- **CPU Used**: Tracks how much CPU time the process has used.
- **Clock Time Elapsed**: Records how long the process has been running since it started.
- **Time Limits**: Shows any maximum time limits set for the process.
- **Job or Process Numbers**: Keeps track of various job or process numbers for reference.

6. I/O Status Information

- **List of I/O Devices**: Shows which input/output devices (like printers or disk drives) are currently being used by the process.
- **List of Open Files**: Tracks which files are open and being used by the process.

Process State

As a process runs, it moves through various states. Each state represents a different stage in the lifecycle of a process. Here's a simple breakdown of each state:

1. **New**
    - This is the initial state of a process when it is being created. At this stage, the process is not yet ready to run but is being set up by the operating system.
2. **Ready**
    - In this state, the process has been loaded into the main memory and is waiting for the CPU to be allocated to it. It's like waiting in line at a service counter; the process is prepared to run but is waiting for its turn.

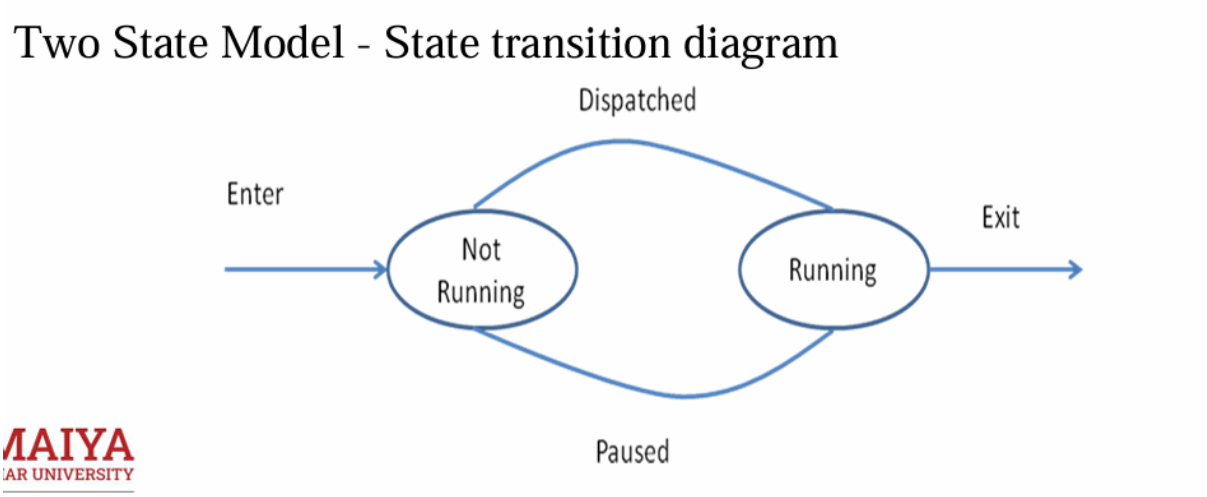
3. **Running**
    - This is the state where the process is actively executing instructions. The CPU is currently working on this process, performing the tasks it needs to complete.
4. **Waiting**
    - The process moves to this state when it is waiting for an external event to complete, such as an I/O operation (e.g., reading from a disk or waiting for user input). It's paused until the event occurs, after which it can continue running.
5. **Terminated**
    - This state indicates that the process has finished execution. The process is no longer active, and its resources are being cleaned up by the operating system.

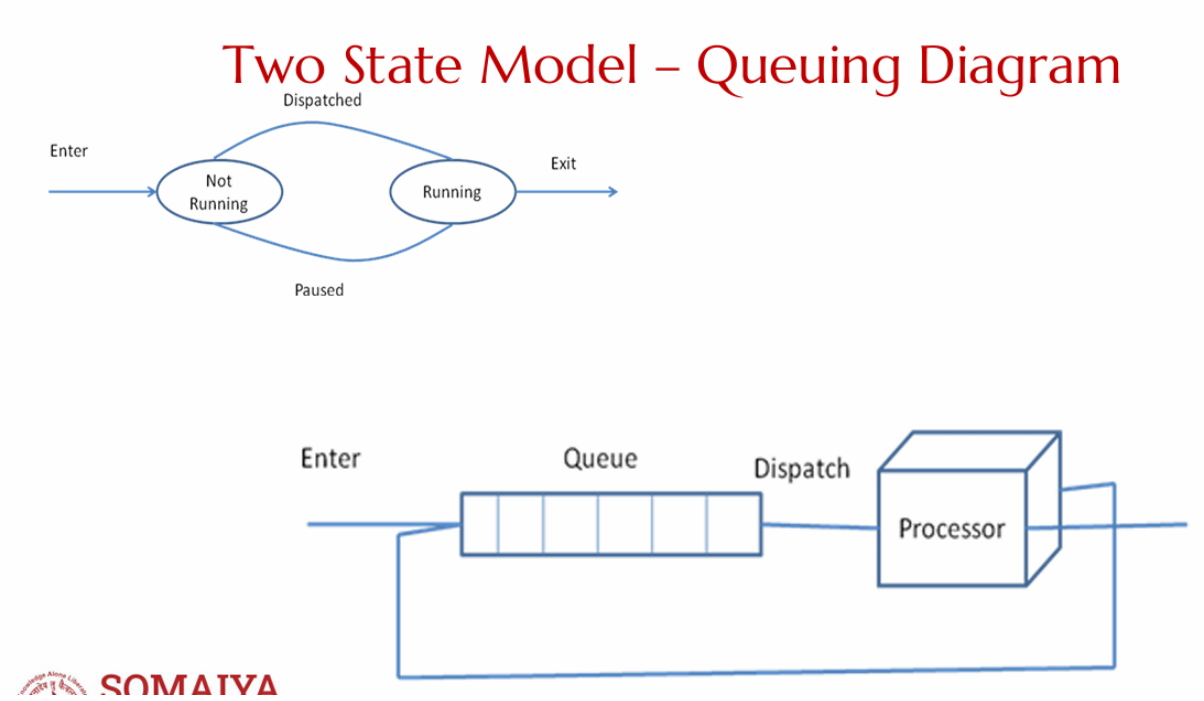
Summary

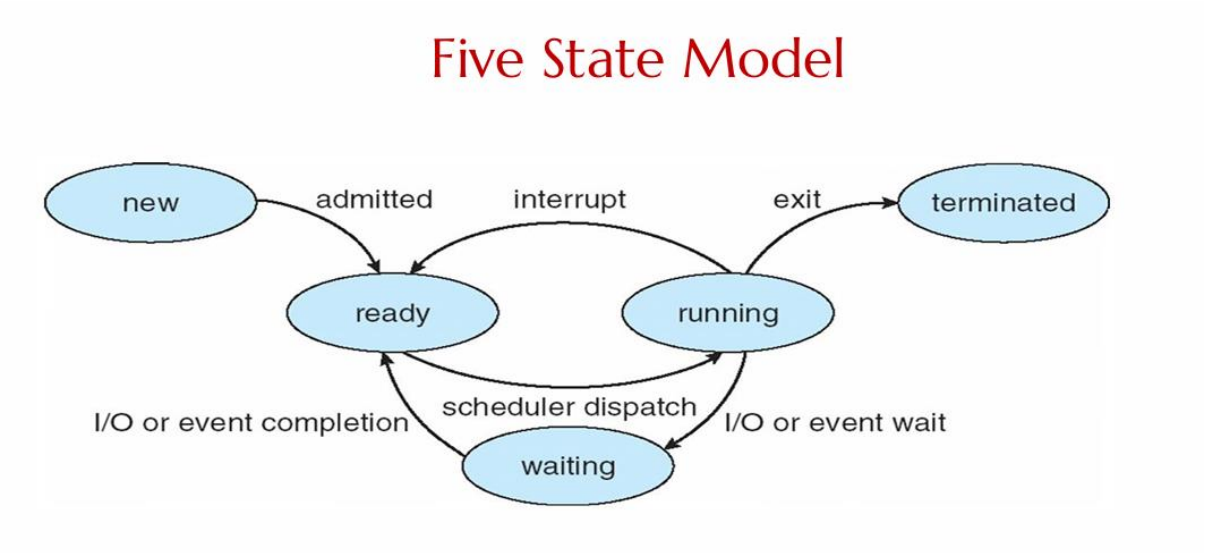
In summary, the states a process goes through are:

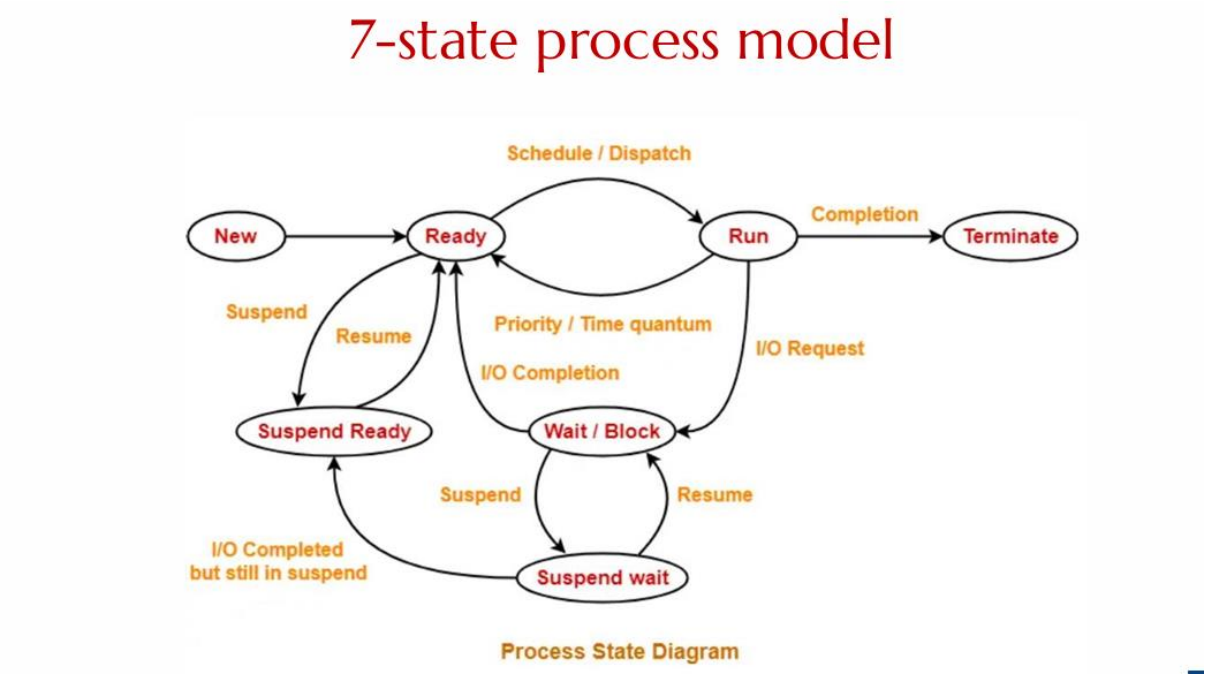
- **New**: Process is being created.
- **Ready**: Process is loaded and waiting for CPU time.
- **Running**: Process is currently executing.
- **Waiting**: Process is waiting for an external event.
- **Terminated**: Process has completed its execution.

Process State Transition









Causes of Process Initiation

Processes in an operating system can be initiated through various mechanisms. Here's a breakdown of the common causes for process creation:

1. **Interactive Logon**
    - **Description**: When a user logs into the system, the operating system creates a new process to handle the user's session. This

process manages user interactions, including handling commands and running applications on behalf of the user.

    - **Example**: Logging into your computer or terminal starts a new process that manages your session and any applications you open.

2. **Created by OS to Provide Some Service**
    - **Description**: The operating system may create processes to provide specific services requested by the user or other processes. This is done to perform tasks without making the user wait or affecting the performance of existing processes.
    - **Example**: When a user requests to print a document, the OS creates a print spooler process to manage the printing task.
3. **Spawned by an Existing Process**
    - **Description**: To support modularity and parallelism, an existing process can create new processes. This allows a program to divide tasks among multiple processes, enabling concurrent execution and improving efficiency.
    - **Example**: A web server process may spawn multiple child processes to handle multiple client requests simultaneously, improving response time and resource utilization.
4. **Program Execution**
    - **Description**: Whenever an application is opened or executed, the operating system creates a new process to run that application. This process is responsible for managing the execution of the program and its resources.
    - **Example**: Opening a web browser or a text editor results in the OS creating a process specifically for running that application.

Summary

Processes can be initiated by:

- **Interactive Logon**: Starting a user session.
- **OS Service Creation**: Handling specific tasks or services.
- **Existing Process Creation**: Supporting modularity or parallel tasks.

- **Program Execution**: Running new applications.

How the OS creates a process?

1. Create a process

2. Assign a unique process ID to newly created process

3. Allocate the memory and create its process image

4. Initialize process control block

5. Set the appropriate linkages to the different data structures such as ready queue etc.

6. Create or expand the other data structures if required

Causes of Process Blocking

A process becomes blocked when it cannot continue executing because it is waiting for some condition to be fulfilled or some resource to become available. Here are common causes of process blocking:

1. **Process Requests an I/O Operation**
    - **Description**: When a process requests input/output operations, such as reading from or writing to a disk, it may need to wait until the I/O operation is completed. During this time, the process is blocked and cannot proceed until the I/O device is ready or the data transfer is finished.
    - **Example**: A process that reads data from a file will be blocked until the data is read from the disk.
2. **Process Requests Memory or Some Other Resource**
    - **Description**: If a process requests memory allocation or another system resource that is currently unavailable, it will be blocked until the requested resources are allocated. This might occur when memory is insufficient or when the resource is held by another process.

- **Example**: A process requesting more memory than is currently available in the system will be blocked until enough memory is freed up.

3. **Process Wishes to Wait for a Specific Interval of Time**
    - **Description**: A process may intentionally block itself to wait for a specific period. This could be part of its execution logic, where it needs to delay its operations or wait for a timer to expire.
    - **Example**: A process that needs to perform an operation at regular intervals might block itself to wait for the next interval.
4. **Process Waits for a Message from Another Process**
    - **Description**: Processes often need to communicate with each other through messages. A process may block while waiting to receive a message from another process before it can proceed with its execution.
    - **Example**: In a client-server model, a server process might block while waiting for a request from a client.
5. **Process Wishes to Wait for Some Action to be Performed by Another Process**
    - **Description**: Sometimes, a process needs to wait for another process to complete a specific action or provide some result before it can continue. This type of blocking ensures that dependent actions occur in the correct order.
    - **Example**: A process might block until another process finishes writing data to a shared resource before it can read or use that data.

Summary

Processes can become blocked due to various reasons:

- **I/O Operation**: Waiting for input/output operations to complete.
- **Memory or Resources**: Waiting for memory allocation or other system resources.
- **Time Interval**: Waiting for a specific period of time.

- **Message**: Waiting to receive a message from another process.
- **Action by Another Process**: Waiting for an action or result from another process.

Causes of Process Termination

A process may be terminated for various reasons, each involving different conditions or system requirements. Here's a detailed look at the common causes of process termination:

1. **Normal Completion**
    - **Description**: When a process completes its intended task successfully, it executes a system call to notify the operating system that it has finished its execution. This is a standard and orderly way for a process to end.
    - **Example**: A word processor saving a document and then closing, signaling that it has finished its job.
2. **Self-Termination**
    - **Description**: A process may terminate itself due to encountering issues that prevent it from continuing, such as incorrect file access permissions or encountering inconsistent or invalid data. This is a self-initiated termination where the process recognizes it cannot proceed.
    - **Example**: An application might terminate itself if it tries to access a file it doesn't have permission to read or write.
3. **Termination by the Parent Process**
    - **Description**: A parent process can terminate a child process when it is no longer needed or if the child process is causing issues. This is often done using a system call where the parent process instructs the OS to kill the child process.
    - **Example**: A program that launches multiple helper processes might terminate one of them if it completes its task or if the parent decides it is no longer necessary.

4. **Exceeding Resource Utilization**
    - **Description**: If a process uses more resources than it is allowed or consumes resources inappropriately, the operating system may terminate it. This can help manage resource allocation and prevent system overloads. It is also part of deadlock recovery procedures to free up resources.
    - **Example**: A process that continuously allocates memory without releasing it might be terminated to prevent a system crash.
5. **Abnormal Conditions During Execution**
    - **Description**: The operating system may terminate a process if it encounters abnormal conditions during execution, such as memory protection violations or arithmetic overflows. These conditions indicate that the process is behaving incorrectly or dangerously.
    - **Example**: A process trying to access memory outside its allocated space or performing division by zero might be terminated to protect system stability.
6. **Deadlock Detection and Recovery**
    - **Description**: In the case of deadlocks, where two or more processes are stuck waiting for each other to release resources, the OS might terminate one or more processes to resolve the deadlock and recover the system's operational state.
    - **Example**: If two processes are holding resources and waiting for each other indefinitely, the OS might terminate one process to break the deadlock and free up resources.

Summary

Processes can be terminated due to:

- **Normal Completion**: Successfully finishing its task and signaling completion.
- **Self-Termination**: Ending itself due to internal issues or invalid conditions.

- **Parent Process Termination**: The parent process instructs the OS to terminate its child process.
- **Exceeding Resource Utilization**: Consuming more resources than allowed or part of deadlock recovery.
- **Abnormal Conditions**: Encountering errors like memory violations or arithmetic issues.
- **Deadlock Detection and Recovery**: Resolving deadlocks by terminating processes.

Process Creation in UNIX

One process can create another process, perhaps to do some work for it. • The original process is called the parent

• The new process is called the child

• The child is an (almost) identical copy of parent (same code, same data, etc.)

• The parent can either wait for the child to complete, or continue executing in parallel (concurrently) with the child

In UNIX, a process creates a child process using the system call fork( )

• Negative: A child process could not be successfully created if the fork() returns a negative value.

• Zero: A new child process is successfully created if the fork() returns a zero.

• Positive: The positive value is the process ID of a child's process to the parent. The process ID is the type of pid_t that is defined in OS or sys/types.h.

• Child often uses exec( ) to start another completely different program

System Calls for Process Management

Sr No	System Call	Description
1	fork()	This system calls creates a new process.
2	exec()	This call is used to execute a new program on a process.
3	wait()	This call makes a process wait until some event occurs.
4	exit()	This call makes a process to terminate
5	getpid()	This system call helps to get the identifier associated with the process.
6	getppid()	This system call helps to get the identifier associated with the parent process.
7	nice()	The current process priority can be changed with execution of this system call.
8	brk()	This call helps to increase or decrease the data segment size of the process.
9	Kill()	The forced termination of any process can be executed with this system call.
10	Signal()	This system call is invoked for sending and receiving software interrupts

Context Switch

1. **What It Is**: A context switch happens when the operating system (OS) pauses one process to start another. This allows multiple processes to share the CPU and run efficiently.
2. **Process Control Block (PCB)**:
    - Each process has a **Process Control Block (PCB)**. This is like a storage box for all the important information about the process, including:
        - **Hardware Registers**: These are special registers like the Program Counter (PC) and Stack Pointer (SP) that keep track of where the process is in its execution.
        - **State Information**: This includes other data the OS needs to remember about the process's current state.
3. **Stopping a Process**: When the OS decides to stop a running process:
    - It saves the values of the hardware registers (like where the process was in its code and what it was doing).
    - It also saves any additional state information in the process's PCB.

4. **Starting a New Process**: When the OS is ready to run a different (waiting) process:
    - It retrieves the saved values from that process's PCB.
    - It loads those values back into the hardware registers, allowing the new process to continue running right where it left off.

<u>Process context switch Vs mode switch</u>

Process Context Switch

1. **What It Is**: A context switch occurs when the OS stops one process to switch to another. This typically happens when the OS needs to respond to an interrupt or when one process needs to yield control.
2. **Saving and Loading**:
    - The OS saves the current process's information (its **Process Image**) into its PCB.
    - It then loads the information of the new process from its PCB so that it can start executing.
3. **Changing Status**: During a context switch, processes change their status between **Running** (actively executing) and **Not Running** (paused or waiting).

Mode Switch

1. **What It Is**: A mode switch happens when a process changes its execution privilege level, switching between **user mode** (low privilege) and **kernel mode** (high privilege).
2. **Privileged Operations**:
    - In **user mode**, the process can run applications and perform basic tasks, but it can't directly access hardware or execute sensitive operations.
    - In **kernel mode**, the process has full access to the system's resources and can perform any operation, including managing hardware.

3. **Execution Continues**: Unlike a context switch, when a mode switch occurs, the process continues executing. It does not stop or switch to another process; it simply changes its privilege level.

Summary

- **Context Switch**: Switching between different processes; requires saving/loading process states.
- **Mode Switch**: Switching between privilege levels within the same process; the process continues to run.

Fundamental Kernel Functions of Process Control

1. **Context Save**:
    - This function is responsible for saving all the important information about a process when it is temporarily paused or suspended.
    - This includes things like the process state, program counter, stack pointer, and any other necessary data. It ensures that the process can resume later without losing any progress.
2. **Scheduling**:
    - Once a process is suspended, the kernel must decide which process to run next. This is done through a function called scheduling.
    - The scheduler picks the next process based on a set policy (like priority, fairness, or round-robin) to make sure that all processes get a chance to run efficiently.
3. **Dispatching**:
    - After the scheduler selects a process, the dispatching function prepares the CPU to execute the chosen process.
    - This involves loading the process's saved information from its PCB (like its context) into the CPU so that it can start running.

How They Work Together

- When an event occurs (like an interrupt or a process yielding), the **context save** function kicks in to save the current process's state.
- This may lead to starting new processes, which prompts the **scheduling** function to choose the next process to run.
- Finally, the **dispatching** function takes control and transfers execution to the new process on the CPU.

Summary

These functions work together to manage processes effectively:

- **Context save** ensures processes can be paused and resumed.
- **Scheduling** decides which process runs next.
- **Dispatching** sets up the CPU for the selected process.

Control/Data Structures Maintained by the OS

What Are Control/Data Structures?

Control and data structures are like organizational tools that the operating system (OS) uses to manage and keep track of everything it needs to run smoothly, especially regarding processes, memory, and resources. Here's how they work:

1. **Memory Tables**:
    - **Purpose**: Track how both main (RAM) and secondary (like hard drives) memory is used.
    - **Key Functions**:
        - Keep track of which active processes are in main memory.
        - Move processes to secondary memory when needed (this is called **swapping**).
        - Store information about:
            - Memory allocated to each process in main memory.

        - Memory allocated to processes in secondary storage.
        - Shared memory areas and their attributes.
        - Miscellaneous details for managing virtual memory.

2. **I/O Tables**:
    - **Purpose**: Manage information about input/output devices and their usage.
    - **Key Functions**:
        - Track I/O devices and channels in the system.
        - Determine whether devices are currently available or allocated to a process.
        - Help ensure that processes can access the devices they need.
3. **File Tables**:
    - **Purpose**: Keep track of files and their usage in the system.
    - **Key Functions**:
        - Maintain a list of all files and their locations on secondary storage.
        - Track the current status of each file (open, closed, etc.).
        - Store attributes like security settings and sharing permissions.
        - This information is usually managed by a **File Management System**.
4. **Process Table**:
    - **Purpose**: Manage all processes currently running in the system.
    - **Key Functions**:
        - Keep information about each process, including:
            - Process ID and status (running, waiting, etc.).

        - References to child processes (processes created by this one).
        - Resources allocated to each process.
        - Context information needed to resume execution.
        - Information necessary for synchronizing processes.
    - All this information is stored in **process images**.

Summary

- **Memory Tables**: Track memory usage for processes and manage swapping.
- **I/O Tables**: Manage I/O device allocation and availability.
- **File Tables**: Keep information about files and their attributes.
- **Process Table**: Maintain details about all processes and their statuses.

Thread

What is a Thread?

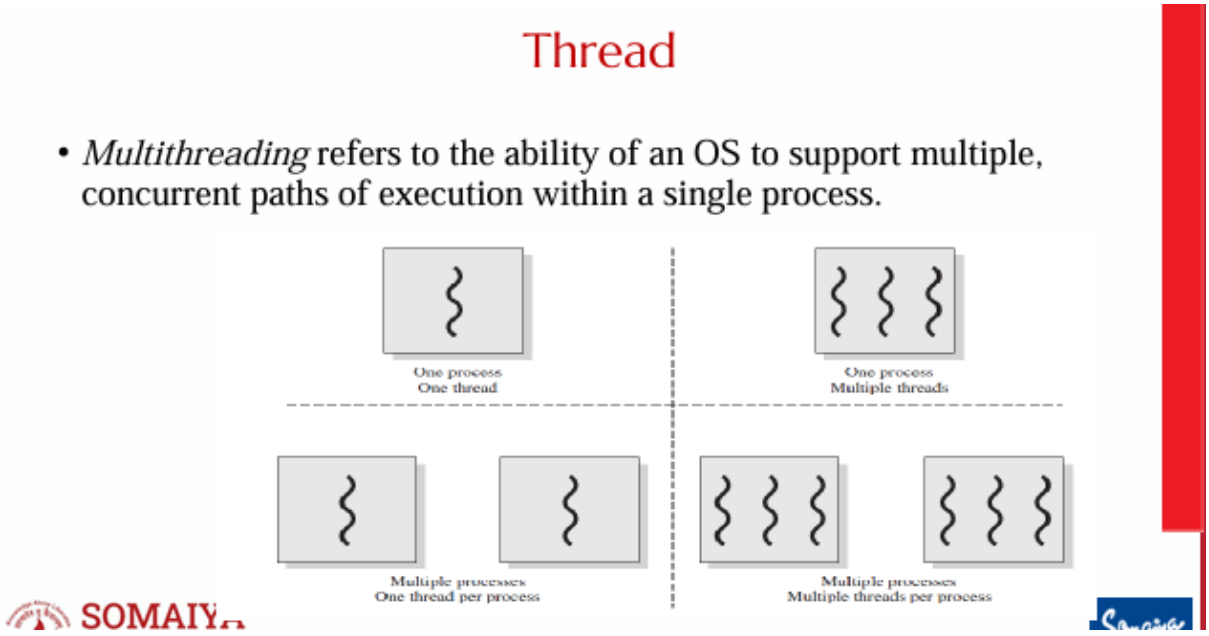
1. **Parallelism**:
    - Threads allow a program to perform multiple tasks at the same time. This is called **parallelism**.
2. **Execution**:
    - Each thread runs in a sequence (one after another) but can be paused so the CPU can switch to another thread. This allows for efficient use of the CPU.
3. **Lightweight**:
    - Threads are considered **lightweight** because they don't need to save all the context of a whole process to run. They only need:
        - A **program counter** (to know where it is in the code).

- A **stack pointer** (to keep track of its own stack space).
- Its own data area for storing variables.

4. **Multiple Executions**:
    - Threads enable multiple tasks to run at the same time within a single application. For example, an app can send notifications in the background while you are using it.
5. **Dividing a Process**:
    - The main idea of using threads is to split a process into smaller parts (threads) that can run simultaneously. This makes programs faster and more responsive.

Summary

- **Threads** are like mini-processes that allow for multitasking within a single application.
- They are lightweight, support parallel execution, and help improve performance by running multiple tasks at the same time.



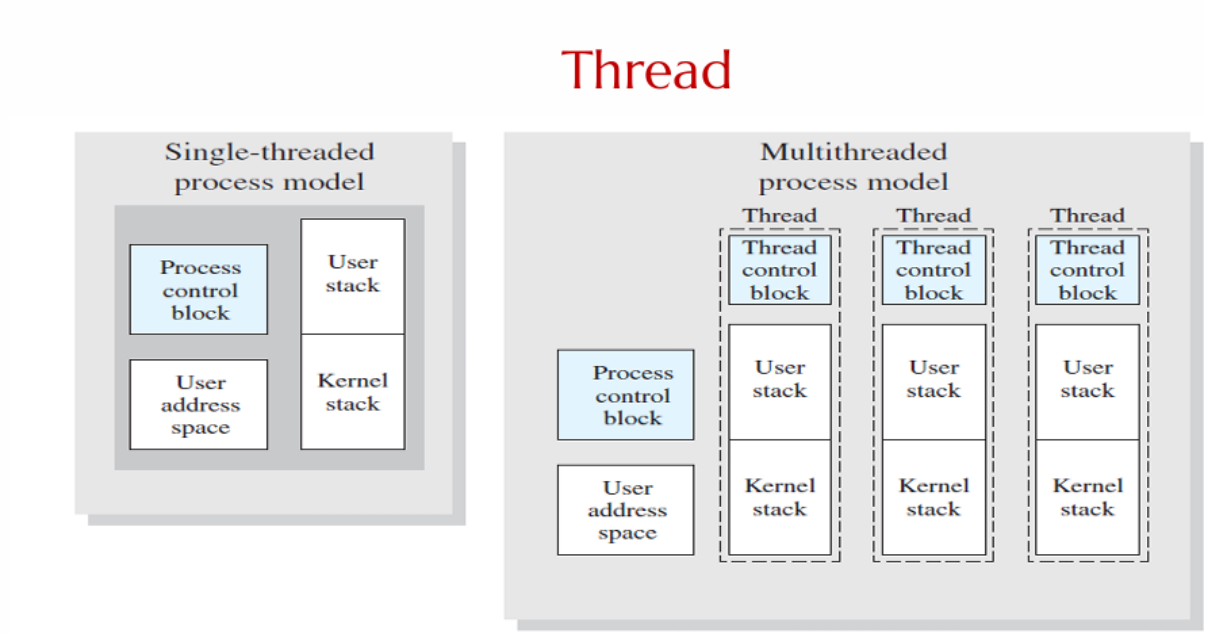


Figure 4.2 **Single Threaded and Multithreaded Process Models**

Benefits of threads

• It takes far less time to create a new thread in an existing process than to create a brand-new process.

• It takes less time to terminate a thread than a process.

Types of threads

• Kernel level threads : managed by kernel

• User level threads : managed by user with support from programming languages and libraries

• Hybrid threads

Process Scheduling

• The operating system is responsible for managing the scheduling activities.

• A uniprocessor system can have only one running process at a time

• The main memory cannot always accommodate all processes at run-time

• The operating system will need to decide on which process to execute next (CPU scheduling), and which processes will be brought to the main memory (job scheduling)

Process Scheduling Queues

1. **Job Queue**:
    - **What it is**: This is a collection of all processes that are currently in the system.
    - **Purpose**: It includes processes that are waiting to be loaded into memory for execution.
2. **Ready Queue**:
    - **What it is**: This is the collection of processes that are in main memory and are ready to run but are waiting for CPU time.
    - **Purpose**: These processes are prepared to execute as soon as the CPU is available.
3. **Device Queues**:
    - **What it is**: These are lists of processes that are waiting for access to an I/O device (like a printer or disk).
    - **Purpose**: Processes in this queue are not able to proceed until the device they need becomes available.
4. **Process Migration**:
    - **What it means**: Processes can move between these queues. For example:
        - A process might move from the job queue to the ready queue when it is loaded into memory.

        - It might move from the ready queue to a device queue if it needs to wait for an I/O operation.

Summary

- **Job Queue**: All processes in the system waiting to be executed.
- **Ready Queue**: Processes ready to run, waiting for CPU time.
- **Device Queues**: Processes waiting for I/O devices.
- **Migration**: Processes can move between these queues based on their state and needs.

CPU and I/O Bursts

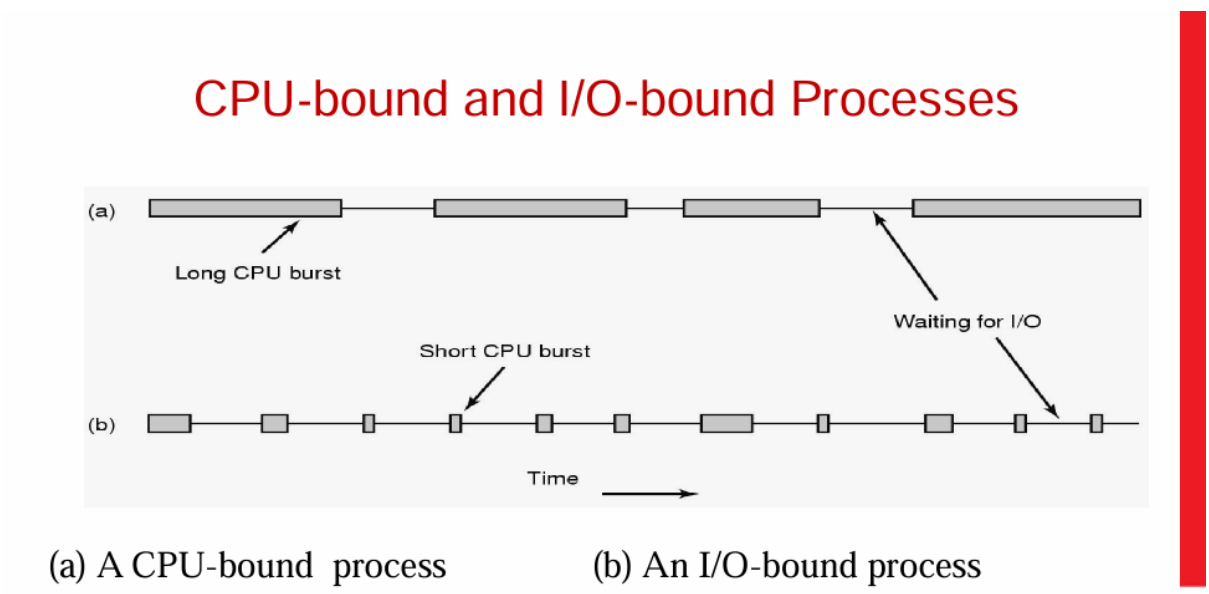
The topic of **CPU and I/O bursts** relates to how processes execute on a computer. It describes the patterns of how processes use the CPU and wait for input/output (I/O) operations.

1. **CPU–I/O Burst Cycle**:
    - **What it is**: The execution of a process alternates between two phases:
        - **CPU Burst**: The time the process spends doing calculations or processing data.
        - **I/O Burst**: The time the process spends waiting for input/output operations to complete (like reading from a disk or sending data to a printer).
2. **I/O-bound Process**:
    - **What it is**: This type of process spends more time on I/O operations than on computations.
    - **Characteristics**:
        - It has many short CPU bursts because it frequently waits for I/O to complete.
        - Example: A web browser loading data from the internet.

3. **CPU-bound Process**:
    - **What it is**: This type of process spends more time on computations than on I/O operations.
    - **Characteristics**:
        - It has fewer but longer CPU bursts because it primarily focuses on processing tasks.
        - Example: A video rendering application that requires extensive calculations.

Summary

- **CPU–I/O Burst Cycle**: Processes alternate between processing (CPU burst) and waiting (I/O burst).
- **I/O-bound Processes**: Spend more time on I/O, have many short CPU bursts.
- **CPU-bound Processes**: Spend more time on computations, have fewer but longer CPU bursts.



Schedulers

Schedulers are key parts of an operating system that help manage how processes (programs in execution) run. They decide which processes run, when they run, and how resources like the CPU are shared.

Key Points

1. **Spooling**:
    - Processes can be temporarily stored on a hard drive before they run. This is called **spooling**. It helps the system manage multiple processes that aren't currently active.
2. **Long-term Scheduler (Job Scheduler)**:
    - **What it does**: Chooses processes from the storage (spool) and loads them into memory so they can run.
    - **Why it matters**: Controls how many processes can be in memory at once, which helps manage system performance.
    - **How often it runs**: This scheduler works less frequently, only when processes need to be loaded.
3. **Short-term Scheduler (CPU Scheduler)**:
    - **What it does**: Picks from the list of processes that are ready to run (in the **ready queue**) and gives one of them CPU time.
    - **Why it matters**: Ensures the CPU is used efficiently by quickly deciding which process should run next.
    - **How often it runs**: This scheduler works very often, sometimes multiple times a second, to respond quickly to changes in process states.

Comparison of Schedulers

Parameters	Long-Term	Short-Term	Medium-Term
Type of Scheduler	It is a type of job scheduler.	It is a type of CPU scheduler.	It is a type of process swapping scheduler.
Speed	Its speed is comparatively less than that of the Short-Term scheduler.	It is the fastest among the other two.	Its speed is in between both Long and Short-Term schedulers.
Minimal time-sharing system	Almost absent	Minimal	Present
Purpose	A Long-Term Scheduler helps in controlling the overall degree of multiprogramming.	The Short-Term Scheduler provides much less control over the degree of multiprogramming.	Medium-Term reduces the overall degree of multiprogramming.
Function	Selects processes from the pool and then loads them into the memory for execution.	Selects all those processes that are ready to be executed.	Can re-introduce the given process into memory. The execution can then be continued.

When to Schedule

In an operating system, the CPU scheduler determines when to allocate CPU time to processes based on their state changes. Here are the key points where the scheduler could be invoked:

Points for CPU Scheduling

1. **When a process switches from the New state to the Ready state**:
    - This happens when a new process is created and is ready to start running but is waiting for CPU time.

2. **When a process switches from the Running state to the Waiting state**:
    - This occurs when a running process needs to wait for an I/O operation to complete (like reading from a disk) and can't continue executing.
3. **When a process switches from the Running state to the Ready state**:
    - This happens when a running process is preempted (paused) so that the CPU can be given to another process, often due to scheduling policies.
4. **When a process switches from the Waiting state to the Ready state**:
    - This occurs when an I/O operation that a waiting process was waiting for has completed, making the process ready to run again.
5. **When a process terminates**:
    - This happens when a process finishes its execution, freeing up resources and allowing the scheduler to allocate the CPU to another process.

Non-preemptive vs. Preemptive Scheduling

In process scheduling, there are two main approaches: **non-preemptive** and **preemptive** scheduling. Each approach has its own method for managing how processes use the CPU.

Non-preemptive Scheduling

- **What it is**: Once a process starts running, it keeps the CPU until it finishes its execution or switches to a waiting (blocked) state.
- **How it works**:
    - The running process cannot be interrupted. It will either:
        - Complete its task, or
        - Wait for I/O operations to finish.
- **Pros**: Simpler to implement and can be more predictable.
- **Cons**: Can lead to longer wait times for other processes, especially if a long process is running.

Preemptive Scheduling

- **What it is**: A running process can be forced to release the CPU before it finishes or switches to a waiting state.
- **How it works**:
    - **Time-slicing**: In time-sharing systems, each process is given a limited time slice (quantum). When the time slice expires, the scheduler can interrupt the running process and switch to another process.
    - **Changes in the ready queue**: Whenever there is a change in the ready queue (like a new process becoming ready), the scheduler can also decide to preempt the current process.
- **Pros**: More responsive and fairer, as it allows the system to allocate CPU time to multiple processes effectively.
- **Cons**: More complex to implement and can lead to overhead from frequent context switching.

Scheduling Criteria

When evaluating the performance of scheduling algorithms in an operating system, several criteria can be used. These criteria help determine how effective a scheduling method is in managing processes. Here's a breakdown of the key criteria:

Key Scheduling Criteria

1. **CPU Utilization**:
    - **What it is**: Measures how effectively the CPU is being used.
    - **Goal**: Keep the CPU as busy as possible to maximize efficiency.
2. **Throughput**:
    - **What it is**: The number of processes that complete execution in a given time period.

- **Goal**: Increase throughput to ensure more processes are finished quickly.

3. **Turnaround Time**:
   - **What it is**: The total time taken to execute a particular process, from submission to completion.
   - **Goal**: Minimize turnaround time so processes are completed faster.
4. **Waiting Time**:
   - **What it is**: The total time a process spends waiting in the ready queue before getting CPU time.
   - **Goal**: Reduce waiting time to improve process responsiveness.
5. **Response Time**:
   - **What it is**: The time from when a request is submitted until the first response is produced (not the complete output).
   - **Goal**: Minimize response time to enhance user experience, especially in interactive systems.
6. **Fairness**:
   - **What it is**: Ensuring that all processes receive an equitable share of CPU time.
   - **Goal**: Prevent any single process from monopolizing CPU resources, avoiding issues like starvation.
7. **Meeting Deadlines (for Real-time Systems)**:
   - **What it is**: The ability to complete tasks within specified time constraints.
   - **Goal**: Ensure that critical processes meet their deadlines, which is crucial for real-time applications.

Optimization Criteria

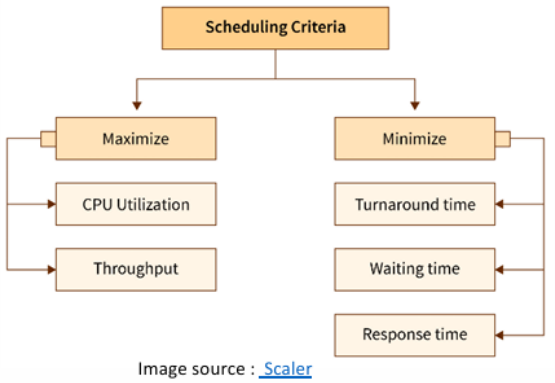


Image source : Scaler

- In the examples, we will assume
  - average waiting time is the performance measure
  - only one CPU burst (in milliseconds) per process

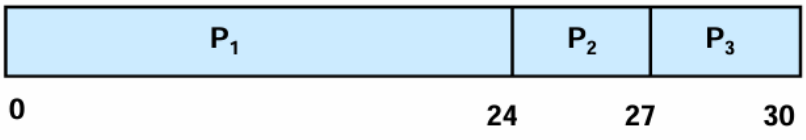
First-Come, First-Served (FCFS) Scheduling

- Single FIFO ready queue
- No-preemptive
  - Not suitable for timesharing systems
- Simple to implement and understand
- Average waiting time dependent on the order processes enter the system
- Consider processes arrive at time 0

Turnaround Time = Completion Time – Arrival Time

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- The *Gantt Chart* for the schedule:



- Turnaround Time P_1 = 24; P_2 = 27; P_3= 30
- Average turnaround time: (24+27+30)/3 = 27ms

1)Considering Process arrive P1,P2,P3

Calculating Turnaround Time

Turn around time of P1 is 24

Turnaround time of P1 = Completion Time of P1 – Arrival Time of P1

= 24-0

=24

Turn around time of P2 is 27

Turnaround time of P2 = Completion Time of P2 – Arrival Time of P2

= 27-0

=27

Turn around time of P3 is 30

Turnaround time of P3 = Completion Time of P3 – Arrival Time of P3

= 30-0

=30

Arrival Time of all Processes is at 0.

Avg turnaround time = (24+27+30)/3

=27ms

Now Let's Calculate Waiting Time

We are still assuming Process arrive P1,P2,P3

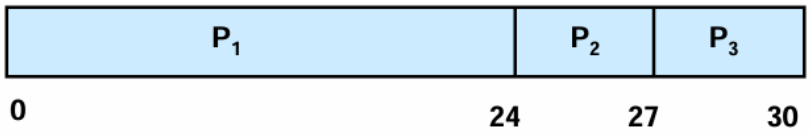
First-Come, First-Served (FCFS) Scheduling

- Consider processes arrive at time 0

  Waiting Time = Turnaround Time – Burst Time

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 ,
- The *Gantt Chart* for the schedule:



0 24 27 30

- Turnaround Time P_1 = 24; P_2 = 27; P_3= 30
- Waiting time for P_1 = 0; P_2 = 24; P_3= 27
- Average waiting time: (0+24+27)/3 = 17ms

Turnaround Time P1=24, P2=27, P3=30

Waiting time of P1 is 0

Waiting time of P1 = Turnaround Time of P1 – Burst Time of P1

= 24-24

=0

Waiting time of P2 is 24

Waiting time of P2 = Turnaround Time of P2 – Burst Time of P2

= 27-3

=24

Waiting time of P3 is 27

Waiting time of P3 = Turnaround Time of P3 – Burst Time of P3

= 30-3

=27

What if we change the order of Process?

Now we will assume process comes in order P2, P1,P3

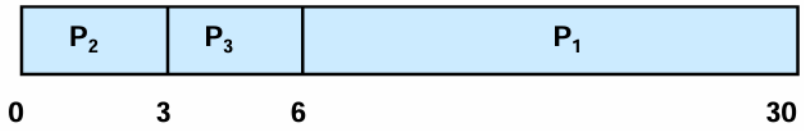
FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order P_2 , P_3 , P_1

Turnaround Time = Completion Time – Arrival Time

- The Gantt chart for the schedule:



- Turnaround Time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Average Turnaround time: (30+3+6)/3 = 13ms
- *Problems:*
  - *Convoy effect* (short processes behind long processes)
  - Non-preemptive -- not suitable for time-sharing systems

FCFS is one of the simplest scheduling algorithms, where processes are served in the order they arrive. However, it has several drawbacks that can impact system performance:

Key Problems

1. **Convoy Effect**:
   - **What it is**: This occurs when short processes get stuck waiting behind long processes in the queue.
   - **Impact**: As a result, the overall waiting time increases, leading to poor responsiveness for short tasks. For example, if a long process is running, all subsequent shorter processes must wait, causing delays.
2. **Non-preemptive**:
   - **What it means**: Once a process starts executing, it cannot be interrupted until it finishes or moves to a waiting state.

- **Impact**: This is not suitable for time-sharing systems where responsiveness is critical. If a long-running process occupies the CPU, other processes may have to wait a long time, leading to inefficiency and poor user experience.

Summary

While FCFS is straightforward and easy to implement, its main issues, such as the convoy effect and being non-preemptive, can lead to increased waiting times and reduced responsiveness. This makes it less ideal for systems where quick and fair access to the CPU is important.

Now Lets Try, Change the order again.

Now we will assume process comes in order P2, P3, P1

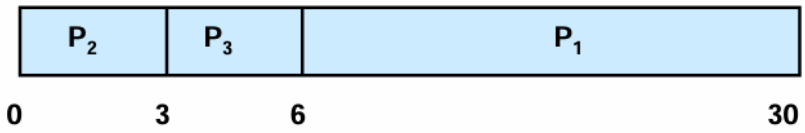
FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order P_2 , P_3 ,

Waiting Time = Turnaround Time – Burst Time

- The Gantt chart for the schedule:



- Turnaround Time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (30+3+6)/3 = 13ms

- *Problems:*
  - *Convoy effect* (short processes behind long processes)
  - Non-preemptive -- not suitable for time-sharing systems

We are facing the same Problem Again.

Shortest-Job-First (SJF) Scheduling

Shortest-Job-First (SJF) scheduling is a popular algorithm used to determine which process gets access to the CPU based on the length of its next CPU burst. The goal is to minimize waiting time and improve efficiency.

There are two types of SJF

- **Non-preemptive SJF**:
  - Once a process starts executing, it cannot be interrupted until it completes.
  - This approach is optimal when all processes are ready at the same time, as it minimizes the average waiting time for that set of processes.
- **Preemptive SJF (Shortest-Remaining-Time-First, SRTF)**:
  - In this variant, if a new process arrives with a shorter burst time than the remaining time of the currently running process, the CPU is preempted and given to the new process.
  - This allows for better responsiveness and shorter average waiting times in scenarios where processes can arrive at different times.

Key Points

1. **Process Burst Length**:
   - Each process is associated with the length of its next CPU burst. The CPU is assigned to the process that has the shortest burst time.

Summary

- **SJF** is an effective scheduling algorithm that prioritizes processes with shorter CPU bursts, leading to reduced average waiting time.
- It can be implemented in two ways: non-preemptive (optimal for simultaneous arrivals) and preemptive (SRTF), which enhances responsiveness in dynamic environments.

Let's Learn Non-Preemptive SJF

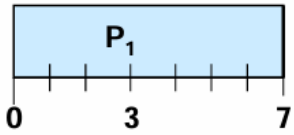
In SJF we give CPU to the process with the lowest Burst Time

But also in Non- Preemptive SJF if a process once starts running, it keeps the CPU until it finishes.
If a process is already running and if a process with lower Burst Time comes we cannot switch to the process with lower burst time.
We Complete the process which is already running and then after completion of the process we take the process with the lowest Burst Time.

Example for Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- At time 0, P_1 is the only process, so it gets the CPU and runs to completion

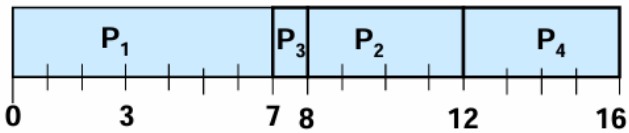


Example for Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Turnaround Time = Completion Time – Arrival Time

- Once P_1 has completed the queue now holds P_2, P_3 and P_4



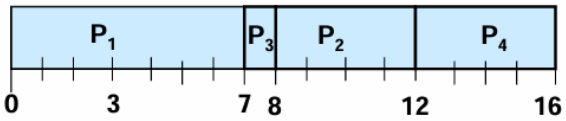
- P_3 gets the CPU first since it is the shortest. P_2 then P_4 get the CPU in turn (based on arrival time)
- Turnaround Time for process p1= 7, p2= 10. p3=4, p4=11
- Average Turnaround time : (7+10+4+11)/4 = 8ms

Example for Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Waiting Time = Turnaround Time – Burst Time

- Once P_1 has completed the queue now holds P_2, P_3 and P_4



- Turnaround Time for process p1= 7, p2= 10. p3=4, p4=11
- Waiting Time for process p1 = 0, p2=6,p3=3,p4=7

Problems with Shortest-Job-First (SJF) Scheduling

While Shortest-Job-First (SJF) scheduling is effective for minimizing waiting time, it has some challenges, particularly in predicting CPU burst lengths.

Key Problem

1. **Predicting CPU Burst Lengths**:
    - **Challenge**: It's difficult to know the exact length of the next CPU burst for a process. Without accurate prediction, the SJF algorithm cannot function optimally.
    - **Impact**: If the predicted burst time is incorrect, it can lead to suboptimal scheduling decisions, affecting overall system performance.

Solution: Predicting Burst Times

To address this issue, we can make predictions based on past CPU bursts using a technique called **exponential averaging**.

Exponential Averaging

- **Formula**: The next CPU burst time can be estimated using the formula:

$$T_{n+1} = \alpha T_n + (1 - \alpha) T_{avg}$$

where:

- T_{n+1} is the predicted length of the next CPU burst.
- T_n is the most recent CPU burst length.
- T_{avg} is the average of all past bursts.
- α is a weighting factor between 0 and 1.

- **How It Works:**
  - A higher value of α (close to 1) gives more weight to the most recent burst, making the prediction more reactive to recent trends.
  - A lower value (close to 0) gives more weight to the average, smoothing out variations.

Shortest Remaining Time First (SRTF) [Preemptive SJF]

• Shortest Remaining Time First (SRTF) scheduling algorithm is basically a preemptive mode of the Shortest Job First (SJF) algorithm in which jobs are scheduled according to the shortest remaining time.

• In this scheduling technique, the process with the shortest burst time is executed first by the CPU, but the arrival time of all processes need not be the same.

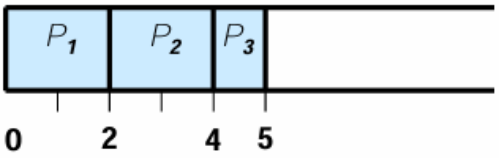
• If another process with the shortest burst time arrives, then the current process will be preempted, and a newer ready job will be executed first.

• Also called as Shortest Remaining Time Next (SRTN)

Example for Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

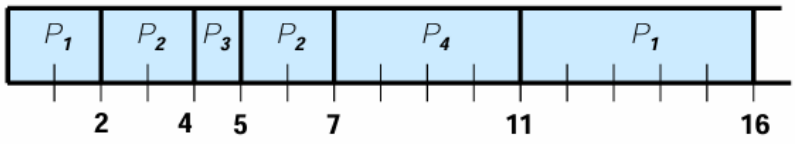
- Time 0 – P_1 gets the CPU Ready = [(P_1,7)]
- Time 2 – P_2 arrives – CPU has P_1 with time=5, Ready = [(P_2,4)] – P_2 gets the CPU
- Time 4 – P_3 arrives – CPU has P_2 with time = 2, Ready = [(P_1,5),(P_3,1)] – P_3 gets the CPU



SOMAIYA VIDYAVIHAR UNIVERSITY

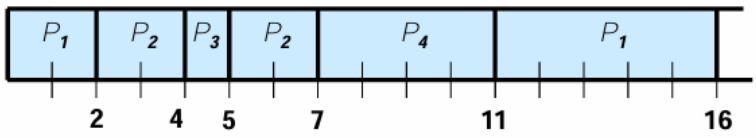
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- Time 5 – P_3 completes and P_4 arrives - Ready = [(P_1,5),(P_2,2),(P_4,4)] – P_2 gets the CPU
- Time 7 – P_2 completes – Ready = [(P_1,5),(P_4,4)] – P_4 gets the CPU
- Time 11 – P_4 completes, P_1 gets the CPU



Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

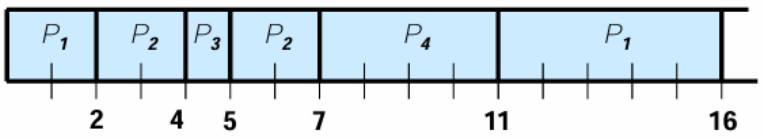
Turnaround Time = Completion Time – Arrival Time



- Turnaround Time p1=16 ,p2=5,p3=1,p4=6
- Average Turnaround time = (16 + 5 + 1 +6)/4 = 7ms

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Waiting Time = Turnaround Time – Burst Time



- Turnaround Time p1=16 ,p2=5,p3=1,p4=6
- Waiting Time p1=9 ,p2=1,p3=0,p4=2
- Average waiting time time = (9 + 1 + 0 +2)/4 = 3ms

Priority-Based Scheduling

Priority-based scheduling is a method used by operating systems to manage process execution based on priority levels. Each process is assigned a priority number, and the CPU is allocated to processes according to their priority.

Key Points

1. **Priority Assignment**:
    - Each process is associated with a priority number (usually an integer).
    - The CPU is allocated to the process with the highest priority (the smallest integer represents the highest priority).
2. **Process Arrival**:

- When a new process arrives in the ready queue, its priority is compared with that of the currently running process.

3. **Types of Priority Scheduling**:
    - **Preemptive Priority Scheduling**:
        - If a newly arrived process has a higher priority than the currently running process, the CPU is preempted (the current process is paused).
        - The currently running process is moved back to the ready queue, and the higher-priority process is scheduled for execution.
    - **Non-Preemptive Priority Scheduling**:
        - If a new process arrives with a higher priority, it is placed at the end of the ready queue.
        - The currently running process continues until it completes. Afterward, the scheduler picks the next process from the ready queue.

Special Case: SJF as Priority Scheduling

- **Relationship to SJF**:
    - Shortest Job First (SJF) can be seen as a special case of priority scheduling:
        - In this context, the process priority is defined as the inverse of the remaining CPU time. This means:
            - The shorter the remaining CPU burst time, the higher the priority.
            - Conversely, longer CPU bursts have lower priority.
    - If two processes have equal priority, they can be scheduled using First-Come, First-Served (FCFS) order.

Summary

Priority-based scheduling efficiently allocates CPU time based on process importance. It can be preemptive or non-preemptive, allowing for flexible

management of process execution. Additionally, SJF is a specific example of priority scheduling, focusing on burst lengths.

THIS EXAMPLE IS OF NON-PREMITIVE

Example for Priority-based Scheduling

- Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds.

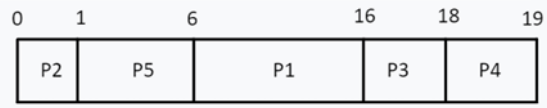
Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- Low number represents the high priority.

Turnaround Time = Completion Time – Arrival Time

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- Gantt Chart

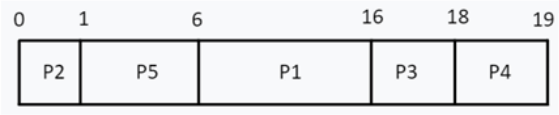


- Turnaround Time p1=16,p2=1,p3=18,p4=19,p5=6
- Average Turnaround Time = (16+1+18+19+6)/5=12ms

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Waiting Time = Turnaround Time – Burst Time

- Gantt Chart



- Turnaround Time p1=16,p2=1,p3=18,p4=19,p5=6
- Waiting Time p1=6,p2=0,p3=16,p4=18,p5=1
- Average Turnaround Time = (6+0+16+18+1)/5=8.2ms

EXAMPLE OF PRIMITIVE IS NOT GIVEN

Problems with Priority-Based Scheduling

Key Problem: Indefinite Blocking (Starvation)

1. **Indefinite Blocking (Starvation)**:
    - **What it is**: Low-priority processes may never get a chance to execute if higher-priority processes continuously arrive and are scheduled for execution.
    - **Impact**: This can lead to some processes being perpetually delayed, which is particularly problematic for critical tasks that might need to run.

Solution: Aging

- **Aging**:
    - **What it is**: A technique to prevent starvation by gradually increasing the priority of processes that have been waiting for a long time.

- **How it works**: As time progresses, the system increases the priority of waiting processes, making it more likely that they will eventually be scheduled for execution.
- **Goal**: Ensure that no process remains indefinitely blocked, promoting fairness in process scheduling.

Round Robin Scheduling

Round Robin (RR) scheduling is a CPU scheduling method that improves on First-Come, First-Served (FCFS) by allowing tasks to be interrupted. This makes it better for multitasking because it ensures that all tasks get a fair share of CPU time, improving responsiveness.

Key Features

1. **Time Quantum (Time Slice)**:
   - A fixed time period, usually between 10 to 100 milliseconds, is defined as the time quantum.
   - Each process in the ready queue is assigned this time quantum for execution.
2. **Execution Process**:
   - The CPU scheduler allocates the CPU to each process for a maximum of one time quantum.
   - After the time quantum expires, the current process is preempted (paused) and moved to the end of the ready queue.
   - The CPU then allocates time to the next process in the queue.
3. **Handling New Processes**:
   - Newly arriving processes and those that complete their I/O operations are added to the end of the ready queue.
   - This ensures that all processes have an opportunity to execute in a fair manner.

4. **Waiting Time**:
    - If there are n processes in the ready queue and the time quantum is q , then no process will wait more than (n−1)×q time units before getting CPU time again.
    - This ensures that processes are serviced within a reasonable time frame, enhancing responsiveness.

Example 1)

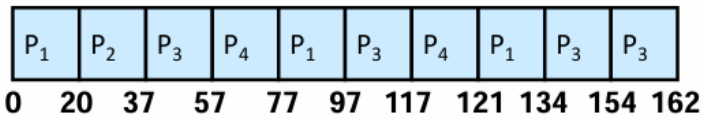
Example for Round-Robin

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- Time Quantum given as 20

Turnaround Time = Completion Time – Arrival Time

- The Gantt chart:



- Turn around time = 134+37+162+121=454/4=113.5

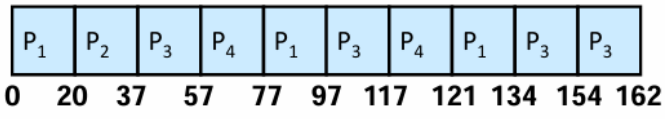
Example for Round-Robin

Process	Burst Time	Turnaround Time
P_1	53	134
P_2	17	37
P_3	68	162
P_4	24	121

- Time Quantum given as 20

Waiting Time = Turnaround Time – Burst Time

- The Gantt chart:



- Average wait time = (81+20+94+97)/4 = 73
- Typically, higher average turnaround time (amount of time to execute a particular process) than SJF, but better *response time* (amount of time it takes from when a request was submitted until the first response is produced)

Example 2)

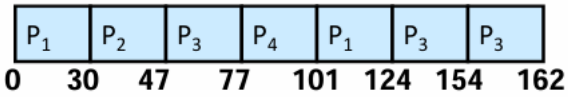
Example for Round-Robin

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- Time Quantum = 30

Turnaround Time = Completion Time – Arrival Time

- The Gantt chart

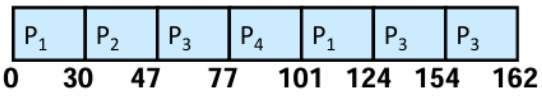


- Turn around Time = (124+47+162+101)=434/4 = 108.5

Example for Round-Robin

Process	Burst Time	Turnaround Time
P_1	53	124
P_2	17	47
P_3	68	162
P_4	24	101

The Gantt chart: (Time Quantum = 30)



Waiting Time = Turnaround Time – Burst Time

- Average wait time = (71+30+94+77)/4 = 68

Effect of Time Quanta in Round Robin Scheduling

The length of the time quantum (or time slice) in Round Robin scheduling significantly impacts system performance and responsiveness. Here's how it affects the scheduling process:

Key Effects of Time Quantum Length

1. **Short Time Quanta**:
    - **Benefits**:
        - Short processes can complete more quickly, leading to faster turnaround times.

        - Improved responsiveness for interactive applications since processes are frequently switched.
    - **Drawbacks**:
        - Increased overhead due to more frequent context switches (the process of saving and loading process states).
        - Excessive context switching can reduce overall CPU efficiency, as time is spent on management rather than execution.
        - **Conclusion**: Very short time quanta should be avoided to prevent overwhelming the system with overhead.
2. **Long Time Quanta**:
    - **Benefits**:
        - Fewer context switches, which can improve CPU efficiency, especially for CPU-bound processes that require more continuous execution time.
        - Less overhead from scheduling and dispatching, allowing more time for actual processing.
    - **Drawbacks**:
        - Increased response time for interactive processes, as they might have to wait longer before getting CPU time.
        - If the time quantum exceeds typical interaction times, users may experience delays, leading to a poorer user experience.

Multilevel Queue Scheduling

Multilevel Queue Scheduling is a way to manage processes in an operating system by dividing them into different groups (queues) based on their needs. Here's a simple breakdown:

Key Features

1. **Separate Queues**:
    - The ready queue is split into different queues:

        - **Foreground (Interactive) Queue**: For processes that need quick responses, like user applications.
        - **Background (Batch) Queue**: For processes that can run without immediate user interaction, like long calculations.
2. **Different Scheduling Methods**:
    - Each queue can use its own scheduling method. Examples include:
        - **Round Robin**: Each process gets a small amount of CPU time in turns (great for interactive tasks).
        - **First-Come, First-Served (FCFS)**: Processes are handled in the order they arrive (suitable for background tasks).
3. **Priority Handling**:
    - Queues can be served based on priority. The highest priority queue gets to run first.
    - This ensures that important tasks are completed quickly.

Challenges

1. **Time Allocation**:
    - Each queue is given a specific portion of CPU time. For example:
        - The highest priority queue might get 50% of the CPU time.
        - The next queue might get 20%, and so on.
    - Finding the right balance is crucial to avoid starving lower-priority tasks.
2. **Queue Assignment**:
    - When a new process arrives, it must be placed in the right queue. Clear rules are needed for this assignment to keep everything running smoothly.

Summary

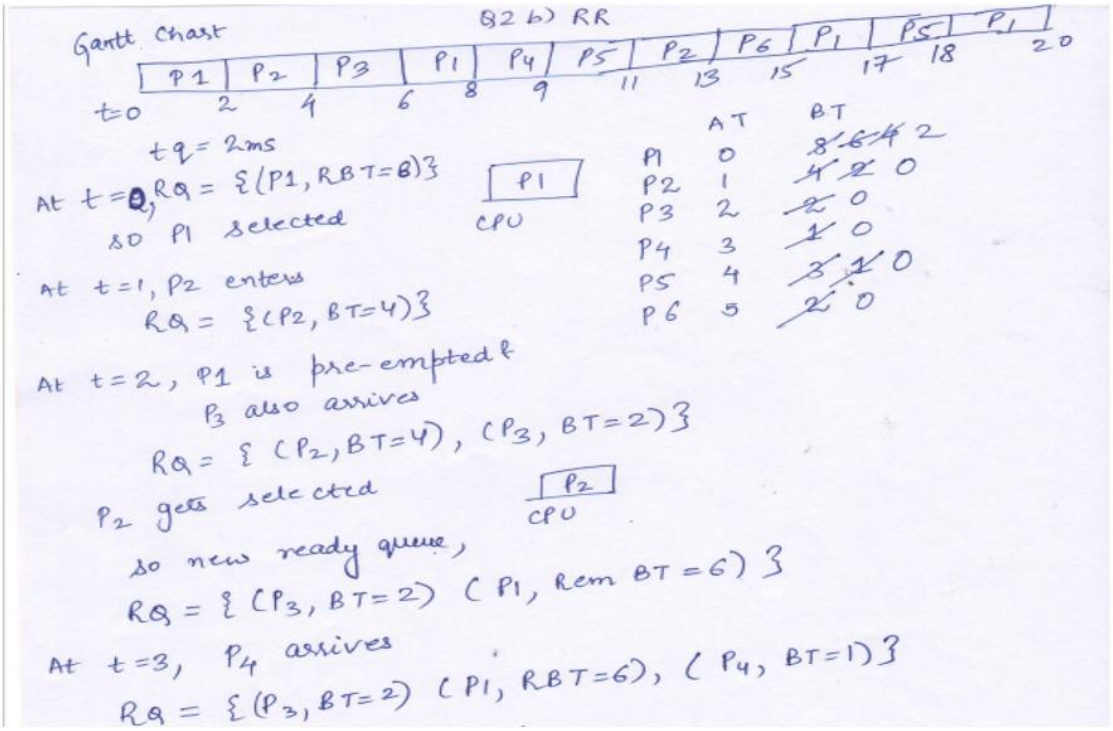
Multilevel Queue Scheduling helps the operating system efficiently manage different types of processes by using separate queues and tailored scheduling methods. This way, it ensures that both interactive and background tasks are handled appropriately

Consider a set of 6 processes whose arrival time, CPU time needed are given below:

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	8
P2	1	4
P3	2	2
P4	3	1
P5	4	3
P6	5	2

If the CPU scheduling policy is Round Robin. Illustrate the scheduling policy with the help of Gantt chart.

What will be the Average Waiting Time and Average Turnaround time if the scheduling policy is Round Robin. Assume quantum time for Round-Robin as 2 ms.



Linux Scheduling:

1. **Definition and Purpose**:
    - Linux scheduling refers to the process by which the kernel manages the execution of processes (tasks) in a system. It ensures efficient CPU time allocation to various processes, providing multitasking and responsiveness.
2. **Scheduler Type**:
    - Linux uses the **Completely Fair Scheduler (CFS)** for general-purpose tasks. CFS aims to allocate CPU time fairly among all runnable processes by maintaining a virtual runtime for each task.
3. **Scheduling Classes**:
    - There are different scheduling classes, such as:
        - **CFS**: For normal tasks (user processes).
        - **Real-time**: For time-sensitive tasks, ensuring high priority.
        - **Idle**: For tasks running when no other task is ready to run.
4. **Time Slices and Priorities**:
    - Each process is assigned a time slice, or quantum, during which it can run. Priority determines how soon a process gets scheduled for execution. Higher priority tasks preempt lower priority ones.
5. **Preemption**:
    - The Linux scheduler supports **preemption**, meaning a running process can be interrupted and replaced with another higher-priority process, ensuring responsiveness, especially in real-time systems.