

## DEPARTMENT OF COMPUTER APPLICATIONS

### MODULE 2 (PART 2)

## Multi-threaded Programming

### Definition

A **thread** is the smallest unit of CPU utilization that forms the basic unit of execution within a process.

**Multithreading** refers to the ability of a CPU or an operating system to execute multiple threads concurrently within a single process.

Each thread shares the same code, data, and resources of the process but maintains its own register set and stack.

This allows multiple tasks to execute independently yet cooperate through shared memory.

### Introduction

In traditional single-threaded systems, a process performs only one task at a time.

Modern applications, however, demand parallel execution to improve responsiveness and performance.

For example, in a web browser, one thread may handle user input while another downloads data and a third renders graphics.

Multithreading enables such concurrency within a single process, utilizing CPU resources more efficiently.

Threads provide a cost-effective means of parallelism since they share memory and system resources of their parent process.

Context switching between threads is faster than switching between processes.

Operating systems such as Windows, Linux, and macOS support multithreading at both user and kernel levels.

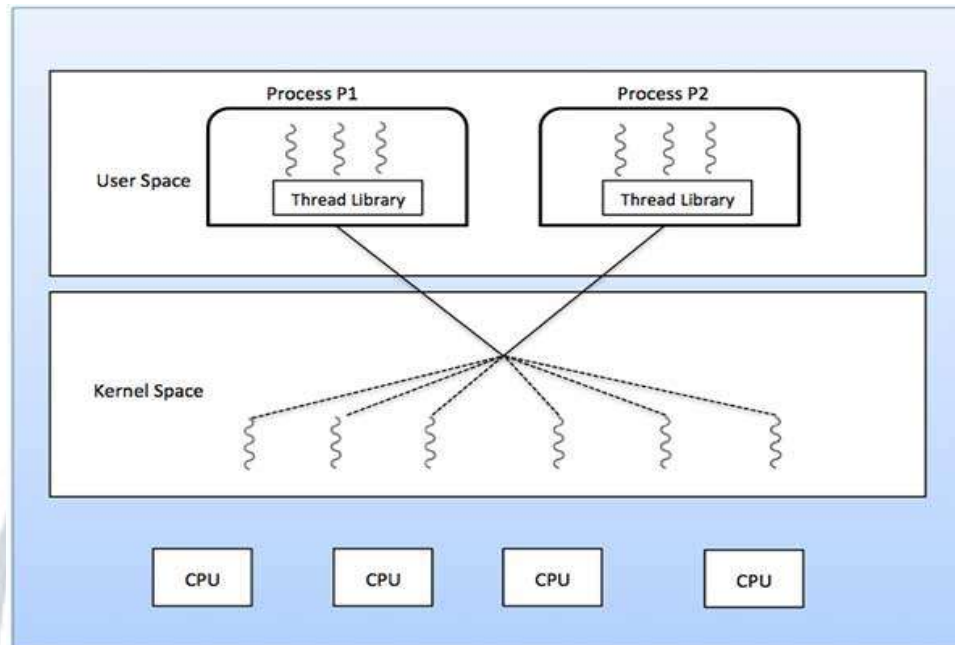
Threads can be created, managed, and synchronized through thread libraries or APIs.

Multithreading improves resource sharing, performance, and program structure while

## DEPARTMENT OF COMPUTER APPLICATIONS

minimizing idle CPU time.

However, it also introduces challenges like synchronization, race conditions, and deadlocks.



Each thread shares the same process memory but has its own execution stack and registers.

### Explanation

A multithreaded process contains several threads running concurrently within the same address space.

Threads allow parallelism at a finer level than processes and are used in both single- and multiprocessor environments.

### Advantages of Multithreading:

- **Responsiveness:** A program remains responsive even if part of it is blocked.
- **Resource Sharing:** Threads share code, data, and files, reducing overhead.
- **Economy:** Thread creation and switching are cheaper than process creation.
- **Scalability:** Multithreading exploits multiprocessor architectures efficiently.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Types of Threads:

1. **User Threads:** Managed by user-level libraries without kernel involvement.
2. **Kernel Threads:** Managed directly by the operating system kernel.

### Multithreading Models:

1. **Many-to-One Model:** Many user threads map to one kernel thread. Efficient but cannot exploit multicore systems.
2. **One-to-One Model:** Each user thread maps to a kernel thread; allows concurrency but increases overhead.
3. **Many-to-Many Model:** Many user threads map to multiple kernel threads; combines flexibility and efficiency.
4. **Two-Level Model:** Variation of many-to-many where user threads can be bound to specific kernel threads.

### Thread Libraries:

Common thread libraries include **POSIX Pthreads (UNIX/Linux)**, **Win32 threads (Windows)**, and **Java threads**.

These libraries provide APIs for creating, synchronizing, and managing threads.

### Threading Issues:

Multithreading introduces complexity such as:

- **Fork and Exec system calls** (how threads behave during process creation)
- **Signal handling** (which thread handles asynchronous events)
- **Thread cancellation** (stopping threads safely)
- **Thread-local storage** (variables private to each thread)
- **Scheduler activations** (communication between user and kernel thread schedulers).

Proper synchronization mechanisms like mutexes and semaphores are essential to prevent race conditions and inconsistent states.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Example

A **web server** uses multithreading to handle multiple client requests simultaneously.

Each client connection is served by a separate thread sharing common resources like log files and cache.

This allows one thread to read from a client socket while another writes data or processes HTTP responses.

In contrast, a single-threaded server would queue connections and respond sequentially, reducing performance.

### Conclusion

Multithreading enhances performance by allowing concurrent execution within a single process.

It reduces overhead, improves responsiveness, and utilizes CPU resources efficiently.

Despite its benefits, it requires careful design to manage synchronization and avoid race conditions.

Most modern operating systems and applications rely heavily on multithreading for responsiveness and scalability.



## DEPARTMENT OF COMPUTER APPLICATIONS

### Process Scheduling

#### Definition

**Process scheduling** is the mechanism by which the operating system decides the order in which processes are executed by the CPU.

It ensures maximum CPU utilization, fairness, and responsiveness in multiprogrammed systems.

The component responsible for this function is called the **CPU Scheduler**.

Scheduling determines which ready process will run next and for how long.

#### Introduction

In a multiprogramming environment, several processes compete for the CPU simultaneously.

Since there is usually only one CPU, the OS must schedule these processes efficiently.

The **scheduler** selects processes from the ready queue and allocates CPU time according to predefined policies.

The goals of scheduling are to maximize throughput, minimize turnaround and waiting times, and ensure fairness among processes.

Scheduling can be **preemptive** (CPU can be taken away) or **non-preemptive** (once given, process runs to completion).

Different scheduling algorithms are used depending on system requirements, such as batch, interactive, or real-time systems.

Efficient scheduling increases CPU utilization and enhances user experience.

In multiprocessor systems, scheduling extends to load balancing and thread-level parallelism.

Thread scheduling further refines control by selecting which thread within a process executes next.

Proper scheduling policies ensure the optimal use of system resources and responsiveness.

#### Explanation

The **process scheduler** operates at three levels:

## DEPARTMENT OF COMPUTER APPLICATIONS

### 1. Long-Term Scheduler (Job Scheduler):

- Decides which processes are admitted to the system for execution.
- Controls the degree of multiprogramming.

### 2. Short-Term Scheduler (CPU Scheduler):

- Selects from the ready queue which process to execute next.
- Must be fast because it runs frequently.

### 3. Medium-Term Scheduler:

- Handles process suspension and swapping between memory and disk to improve performance.

### Scheduling Criteria:

- CPU Utilization (maximize)
- Throughput (maximize)
- Turnaround Time (minimize)
- Waiting Time (minimize)
- Response Time (minimize)
- Fairness (avoid starvation)

### Scheduling Algorithms:

#### 1. First-Come, First-Served (FCFS):

- Processes executed in the order they arrive.
- Simple but may cause the “convoy effect.”

#### 2. Shortest-Job-First (SJF):

- Executes the process with the smallest CPU burst next.
- Can be preemptive (Shortest Remaining Time First).

#### 3. Priority Scheduling:

- Each process assigned a priority; highest priority runs first.
- Risk of starvation unless aging is used.

#### 4. Round Robin (RR):

## DEPARTMENT OF COMPUTER APPLICATIONS

- Each process gets a fixed time quantum.
- Suitable for time-sharing systems.

### 5. Multilevel Queue Scheduling:

- Processes divided into queues (e.g., system, interactive, batch).

### 6. Multilevel Feedback Queue:

- Dynamic adjustment of process priorities based on behavior.

### Thread Scheduling:

- In multithreaded systems, scheduling occurs at both **user level** (thread library) and **kernel level** (system scheduler).
- User-level threads may use cooperative scheduling, while kernel threads are preemptively managed.

### Multiple-Processor Scheduling:

- CPUs may share a common ready queue (symmetric multiprocessing) or have dedicated queues (asymmetric).
- Load balancing ensures even distribution of processes.
- Processor affinity keeps threads on the same CPU for cache performance.

### Example

Consider a time-sharing system using **Round Robin scheduling** with a 10ms time quantum. Five processes P1–P5 enter the ready queue sequentially. Each process gets CPU for 10ms before being preempted and placed at the end of the queue. This ensures all processes get fair CPU access and the system remains responsive to user interactions.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Conclusion

Process scheduling is essential for efficient CPU utilization and fair distribution of processing time.

Different scheduling algorithms serve different system objectives.

Proper scheduler design enhances system throughput, minimizes waiting times, and maintains responsiveness.

Thread and multiprocessor scheduling further refine performance in modern multitasking systems.

### Process Concept

#### Definition

A **process** is a program in execution.

It represents an active entity that performs computations using CPU, memory, and I/O devices.

A process includes the program code, current activity, stack, heap, and data section.

In short, it is the unit of work in a modern time-sharing operating system.

#### Introduction

In a computer system, programs exist as passive entities stored on disk.

When a program is loaded into memory and starts execution, it becomes a **process**.

Each process executes in its own address space, isolated from others, ensuring protection and stability.

The OS manages multiple processes simultaneously using **process control blocks (PCBs)** that store execution state, program counter, and register contents.

A process passes through various states: **new, ready, running, waiting, and terminated**.

This lifecycle allows efficient CPU scheduling and resource management.

Processes can be **independent** (no data sharing) or **cooperating** (sharing information with others).

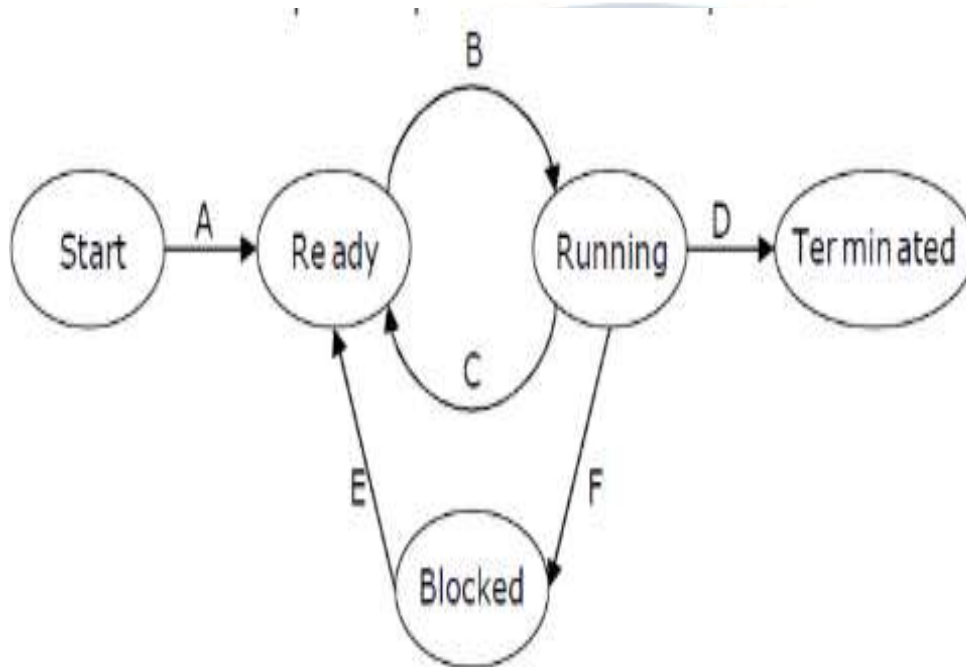


## DEPARTMENT OF COMPUTER APPLICATIONS

Cooperation among processes increases efficiency but requires proper synchronization to prevent race conditions.

The OS provides system calls like `fork()`, `exec()`, and `wait()` to manage processes.

Understanding the concept of processes is fundamental to designing multitasking and multiprogramming systems.



### Explanation

A process includes several components:

1. **Text Section:** Program code.
2. **Data Section:** Global and static variables.
3. **Heap:** Dynamically allocated memory.
4. **Stack:** Function parameters, local variables, and return addresses.

### Process States:

- **New:** Process is being created.

## DEPARTMENT OF COMPUTER APPLICATIONS

- **Ready:** Waiting for CPU allocation.
- **Running:** Currently executing.
- **Waiting:** Blocked for an event (like I/O completion).
- **Terminated:** Execution completed.

### Process Control Block (PCB):

Maintains information about each process, including process ID, state, program counter, CPU registers, memory limits, and accounting information.

### Context Switch:

When the CPU switches from one process to another, the OS saves the current process state in its PCB and loads the next process's state.

### Process Types:

- **System processes:** Background tasks handled by the OS.
- **User processes:** Tasks initiated by users or applications.

The OS uses scheduling queues such as the **ready queue**, **waiting queue**, and **job queue** to manage processes.

### Example

When you open a text editor like Notepad, the program file on disk is loaded into memory. The OS creates a PCB for it, allocates CPU time, and transitions it through the states until you close it, at which point it's terminated.

### Conclusion

The process concept forms the foundation of CPU scheduling and system concurrency. By maintaining separate states and PCBs, the OS ensures efficient and controlled execution of multiple programs simultaneously.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Process Scheduling

#### Definition

Process scheduling is the mechanism by which the OS decides which process should execute next on the CPU.

It maximizes CPU utilization, ensures fairness, and improves overall system performance.

The CPU scheduler selects from the ready queue the next process to run.

This process is then dispatched by the OS to begin or resume execution.

#### Introduction

In a multiprogramming environment, multiple processes compete for CPU time.

The scheduling function is vital to ensure fair and efficient allocation of CPU resources.

Different types of schedulers (long-term, short-term, and medium-term) operate at various levels of process management.

Scheduling decisions can be **preemptive** or **non-preemptive**.

The scheduler uses algorithms like FCFS, SJF, RR, and Priority Scheduling to select processes.

Scheduling criteria help evaluate performance—these include CPU utilization, throughput, turnaround time, waiting time, and response time.

Efficient process scheduling ensures that no single process monopolizes CPU time and that all tasks make progress.

Thread scheduling extends this idea further in multithreaded environments.

Proper scheduling improves responsiveness, particularly in interactive and real-time systems.

It directly impacts the performance of multitasking operating systems.

#### Explanation

##### 1. Schedulers in OS:

- **Long-Term Scheduler:** Controls job admission and system load.
- **Short-Term Scheduler:** Decides which ready process runs next.

## DEPARTMENT OF COMPUTER APPLICATIONS

- **Medium-Term Scheduler:** Manages process swapping and suspension.
- 2. **Scheduling Criteria:**
  - **CPU Utilization:** Keep the CPU as busy as possible.
  - **Throughput:** Maximize number of completed processes.
  - **Turnaround Time:** Minimize total time taken to execute a process.
  - **Waiting Time:** Reduce idle time in queues.
  - **Response Time:** Optimize time for first output response in interactive systems.
- 3. **Scheduling Algorithms:**
  - **FCFS (First Come, First Served):** Non-preemptive and simple.
  - **SJF (Shortest Job First):** Minimizes average waiting time.
  - **Priority Scheduling:** Executes highest-priority process first.
  - **Round Robin:** Allocates equal time quantum to each process.
  - **Multilevel Queue Scheduling:** Divides processes into queues by type or priority.
- 4. **Thread Scheduling:**
  - In user-level threads, scheduling is managed by the thread library.
  - In kernel-level threads, the OS decides scheduling at the system level.
- 5. **Multiple-Processor Scheduling:**
  - In symmetric multiprocessing (SMP), all CPUs share the ready queue.
  - Load balancing distributes processes evenly across CPUs.
  - Processor affinity improves cache performance by keeping threads on the same CPU.

Effective scheduling ensures fair CPU allocation and system responsiveness while avoiding starvation.



## DEPARTMENT OF COMPUTER APPLICATIONS

### Conclusion

Process scheduling determines system performance, efficiency, and user satisfaction.

By selecting the right algorithms and criteria, the OS can balance CPU load, reduce waiting time, and maintain fairness across all processes.

### Scheduling Criteria

#### Definition

Scheduling criteria are performance measures used to evaluate and compare CPU scheduling algorithms.

They help determine how well a scheduling policy meets system goals.

Common criteria include CPU utilization, throughput, turnaround time, waiting time, and response time.

Each criterion balances efficiency and fairness in process execution.

#### Introduction

The performance of scheduling algorithms depends on how they manage CPU and process interaction.

Different systems (batch, interactive, real-time) prioritize different criteria.

Batch systems may focus on throughput and turnaround time, while interactive systems value response time.

The OS designer must balance these metrics for optimal performance.

Some criteria aim to maximize efficiency, while others aim to ensure fairness.

CPU-bound and I/O-bound processes must be scheduled carefully to maintain equilibrium.

Each scheduling algorithm optimizes one or more criteria at the cost of others.

Choosing the right algorithm depends on workload type and system design.

Proper evaluation ensures predictable performance under varying loads.

Hence, scheduling criteria form the basis of comparing and improving scheduling techniques.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Explanation

### Major Scheduling Criteria:

#### 1. CPU Utilization:

- Keep CPU as busy as possible (ranges from 0%–100%).
- High utilization means efficient resource use.

#### 2. Throughput:

- Number of processes completed per unit time.
- Higher throughput improves system productivity.

#### 3. Turnaround Time:

- Total time taken from process submission to completion.
- Includes waiting, execution, and I/O time.

#### 4. Waiting Time:

- Time a process spends in the ready queue.
- Lower waiting time leads to faster processing.

#### 5. Response Time:

- Time from process submission to first output response.
- Important in time-sharing and interactive systems.

### Trade-offs:

- Minimizing one metric may increase another.
- Example: Minimizing response time may reduce throughput.
- Fairness ensures all processes get reasonable CPU access.

Schedulers often use weighted combinations of these criteria to make balanced decisions.

### Example

An interactive OS may prioritize **response time**, while a batch-processing system prioritizes **throughput**.

## DEPARTMENT OF COMPUTER APPLICATIONS

For instance, Linux uses a dynamic scheduler that balances fairness and performance using CPU time slices.

### Conclusion

Scheduling criteria define the effectiveness of an algorithm.

By measuring parameters like CPU utilization and waiting time, the OS ensures optimized performance and fairness.

### Scheduling Algorithms

#### Definition

Scheduling algorithms determine the order in which processes access the CPU.

They define how long a process runs and when it is interrupted.

Different algorithms suit different system goals.

Common types include FCFS, SJF, RR, and Priority Scheduling.

#### Introduction

Each scheduling algorithm aims to optimize CPU performance and meet specific system needs.

Batch systems require algorithms that maximize throughput, while interactive systems prioritize responsiveness.

Algorithms can be **preemptive** or **non-preemptive** based on whether a running process can be interrupted.

Choosing the right algorithm depends on workload type and scheduling criteria.

Many systems use hybrid approaches combining several algorithms.

Understanding their working and behavior helps analyze trade-offs between fairness and efficiency.

Scheduling algorithms are core to process management and performance optimization.

They directly influence waiting time, turnaround, and response times.

## DEPARTMENT OF COMPUTER APPLICATIONS

An ideal scheduler maximizes CPU utilization while preventing starvation.

Thus, studying them is essential for understanding OS behavior.

### Explanation

#### 1. First-Come, First-Served (FCFS):

- Simple, non-preemptive algorithm.
- Processes executed in order of arrival.
- Drawback: Convoy effect (slow process delays others).

#### 2. Shortest Job First (SJF):

- Selects the process with the smallest CPU burst next.
- Optimal for minimizing average waiting time.
- Can be preemptive (SRTF) or non-preemptive.
- Requires prediction of CPU burst time.

### [ Problems discussed in class ]

### Conclusion

Scheduling algorithms define how efficiently the CPU is used.

Each algorithm has advantages and trade-offs depending on system goals.

Combining strategies allows flexibility and balanced performance.



## DEPARTMENT OF COMPUTER APPLICATIONS

### Thread Scheduling

#### Definition

Thread scheduling determines which thread within a process executes next.

It applies to systems supporting multithreading at user and kernel levels.

Thread scheduling enhances concurrency and responsiveness.

It is an extension of process scheduling principles.

#### Introduction

In multithreaded environments, scheduling occurs at both the user and kernel levels.

The system must decide which thread runs when multiple threads are ready.

User-level threads are managed by thread libraries, while kernel threads are managed by the OS scheduler.

Scheduling may differ depending on whether a process uses many-to-one, one-to-one, or many-to-many models.

Thread scheduling improves performance in applications like web servers and databases.

Each thread represents an independent path of execution.

Efficient thread scheduling ensures proper CPU sharing among threads.

It minimizes blocking and increases system responsiveness.

Synchronization mechanisms prevent race conditions among threads.

Thread scheduling is critical for systems relying heavily on parallelism.

#### Explanation

##### User-Level Thread Scheduling:

- Managed by thread libraries like Pthreads.
- The kernel is unaware of these threads.
- Fast and flexible but cannot utilize multiple cores simultaneously.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Kernel-Level Thread Scheduling:

- Managed directly by the OS scheduler.
- Each thread is treated as a separate schedulable entity.
- Allows true parallel execution on multiprocessor systems.

### Hybrid Scheduling:

- Combines benefits of both user and kernel scheduling (many-to-many model).
- Threads can be bound to specific kernel threads for performance.

### Thread Scheduling Policies:

- **Preemptive:** The OS interrupts threads to give others a chance to run.
- **Non-preemptive:** A thread runs until it voluntarily yields.

### Issues:

- Synchronization between threads.
- Load balancing among CPUs.
- Thread priority management.

Proper thread scheduling enhances concurrency and minimizes idle CPU cycles.

### Example

A web server creates multiple threads to handle multiple client requests.

Each thread manages one connection, allowing simultaneous serving of many clients without blocking.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Conclusion

Thread scheduling extends CPU scheduling to finer execution levels.

It ensures efficient use of resources and balanced thread execution.

Modern operating systems depend on robust thread scheduling for parallel processing.

### Multiple-Processor Scheduling

#### Definition

Multiple-processor scheduling involves assigning processes or threads to multiple CPUs.

It ensures balanced load and efficient use of system resources.

Used in systems with two or more processors (SMP, AMP).

Aims to maximize throughput and minimize idle CPU time.

#### Introduction

As hardware evolves, multiprocessor systems have become standard.

The operating system must schedule processes across several CPUs effectively.

Scheduling on multiple processors adds complexity due to shared memory and synchronization requirements.

Two major approaches are **asymmetric multiprocessing (AMP)** and **symmetric multiprocessing (SMP)**.

In AMP, one CPU handles scheduling and others execute tasks; in SMP, all CPUs are peers.

Load balancing ensures even workload distribution among processors.

Processor affinity improves performance by keeping a process on the same CPU.

Thread scheduling further complicates multi-CPU scheduling.

Modern OSs like Linux and Windows use SMP with dynamic load balancing.

This improves performance, scalability, and fault tolerance.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Explanation

#### Types of Multiprocessor Scheduling:

##### 1. Asymmetric Multiprocessing:

- One processor is master and handles all scheduling decisions.
- Simplifies design but underutilizes other processors.

##### 2. Symmetric Multiprocessing (SMP):

- Each processor is self-scheduling.
- All processors share the ready queue or maintain private queues.

#### Key Concepts:

- **Load Balancing:**

Distributes processes evenly among CPUs.

Achieved via **push migration** (one CPU pushes tasks) or **pull migration** (idle CPU pulls tasks).

- **Processor Affinity:**

Keeps processes on the same CPU to improve cache performance.

- **Soft Affinity:** Preference-based.
- **Hard Affinity:** Fixed CPU assignment.

#### Challenges:

- Maintaining fairness among CPUs.
- Synchronization across processors.
- Minimizing overhead in queue management.

Multiprocessor scheduling improves parallel execution and system responsiveness.



## DEPARTMENT OF COMPUTER APPLICATIONS

### Example

In an SMP Linux system with 4 CPUs, each maintains its own run queue.

Idle CPUs pull processes from overloaded CPUs to maintain balance and maximize throughput.

### Conclusion

Multiple-processor scheduling enables true parallel execution.

It enhances performance and system throughput but requires careful coordination.

Load balancing and processor affinity ensure efficient and fair use of CPU resources.

## Interprocess Communication (IPC)

### Definition

**Interprocess Communication (IPC)** is a mechanism that allows processes to exchange data and synchronize their actions.

It enables processes to cooperate and coordinate their execution in a multitasking system.

IPC can be achieved through **shared memory** or **message passing** models.

It is essential for modular, distributed, and concurrent applications.

### Introduction

In modern operating systems, multiple processes often execute simultaneously.

These processes may need to share information or coordinate their actions to perform a task efficiently.

For example, in a client-server model, the server process communicates with many client processes.

To facilitate this, the OS provides IPC mechanisms that ensure reliable and secure data exchange between processes.

IPC ensures that processes can communicate even if they are running on different CPUs or in distributed environments.

## DEPARTMENT OF COMPUTER APPLICATIONS

The two fundamental methods of IPC are **shared memory** (fast, needs synchronization) and **message passing** (safe, needs kernel mediation).

Shared memory allows direct data access, while message passing relies on send/receive primitives.

IPC helps prevent race conditions, maintains data consistency, and supports process cooperation.

Operating systems like UNIX, Linux, and Windows provide various IPC system calls and APIs.

These mechanisms are vital for building multitasking and distributed software systems.

### Explanation

IPC allows processes to work together by either **sharing data** or **sending messages**.

The two primary models are:

#### 1. Shared Memory Model

- A region of memory is established that can be accessed by multiple processes.
- Once created, processes can read and write to the shared segment directly.
- The operating system provides system calls to create (`shmget()`), attach (`shmat()`), and detach (`shmdt()`) shared memory.
- Synchronization mechanisms (like semaphores or monitors) are required to avoid concurrent access problems such as race conditions.
- Shared memory is fast since data does not pass through the kernel.
- Commonly used in producer–consumer models.

#### 2. Message Passing Model

- Processes exchange information using messages rather than shared memory.
- The OS provides primitives:
  - `send(destination, message)`
  - `receive(source, message)`

## DEPARTMENT OF COMPUTER APPLICATIONS

- Communication can be **direct** (processes name each other) or **indirect** (through mailboxes or message queues).
- Message passing can be **blocking** (synchronous) or **non-blocking** (asynchronous).
- It is easier to implement in distributed systems because processes need not share address space.
- Examples include **pipes, message queues, sockets, and remote procedure calls (RPC)**.

### Key IPC Mechanisms in Operating Systems:

- **Pipes:** Enable one-way communication between related processes (parent–child).
- **Named Pipes (FIFOs):** Support communication between unrelated processes.
- **Message Queues:** Allow asynchronous communication using queued messages.
- **Shared Memory:** Fastest communication but needs synchronization tools.
- **Sockets:** Used for communication across networks (e.g., client–server).
- **Signals:** Used for asynchronous notifications.
- **Semaphores and Monitors:** Used for process synchronization in shared memory.

IPC also requires **synchronization** to ensure that processes read and write shared data in a consistent order.

Improper synchronization can cause data inconsistency, deadlocks, or starvation.

Hence, IPC is closely tied to synchronization mechanisms like semaphores, mutexes, and condition variables.

### Conclusion

Interprocess Communication is essential for process cooperation and coordination.

It allows safe and efficient data exchange between concurrent processes.

Shared memory offers speed, while message passing ensures safety and modularity.

Together, they enable scalable multitasking and distributed computing across modern operating systems.

## DEPARTMENT OF COMPUTER APPLICATIONS

