

MODULE 2

What is CPU Scheduling?

- **CPU (Central Processing Unit):** It's the "brain" of the computer that performs tasks or calculations. A computer often runs **many tasks at the same time** (multitasking).
- **CPU Scheduling:** Since the CPU can only do one task at a time, CPU scheduling is the process the operating system uses to decide which task (or process) gets to use the CPU at any given moment. This is a way of **managing multiple tasks** and ensuring that each gets a **fair share of CPU time**.

Why is CPU Scheduling important?

When a computer has multiple tasks (programs, processes, or threads) to run, we need a system to manage **which one should run next**. If the CPU were to only run one task until it's done, other tasks would have to wait too long. By quickly switching between tasks, the operating system ensures that the system is more efficient and can handle multiple things at once (even though the CPU is only running one task at a time).

Processes vs. Threads

- **Processes:** These are the main tasks or programs running on the computer. Each process can consist of **multiple smaller units of execution**.
- **Threads:** Threads are like **smaller tasks** within a process. Think of a process like a big project, and threads as the different steps or parts of that project.

On modern operating systems, the CPU doesn't just schedule processes but also schedules the smaller **threads** inside those processes.

Types of Scheduling

There are different **scheduling algorithms** (rules) used by operating systems to decide which task should run next. These algorithms are important in real-time systems (systems where tasks need to be done within specific time limits) and general systems. Some algorithms **prioritize speed**, while others might **prioritize fairness or efficiency**.

Key Terms

- **Thread Scheduling:** This specifically refers to scheduling threads inside a process, as threads are the ones that actually **get to use the CPU**.
- **Process Scheduling:** This term is more general and refers to scheduling the whole process, but it can sometimes also refer to scheduling the threads within that process.
- **Core:** A core is like a separate mini-CPU inside a CPU. Modern CPUs often have **multiple cores**, which means they can handle more tasks at once.

Example of How It Works

Imagine your computer is running 3 programs (A, B, and C). The CPU will switch between these programs so that they all seem to be running at the same time. The operating system decides which program to run based on different rules (scheduling algorithms).

Conclusion

In simple terms, CPU scheduling is like a traffic controller deciding which car (task) gets to go next on the road (CPU). Since the CPU can only handle one car at a time, scheduling helps make sure that all cars (tasks) are given their time to use the road (CPU) without crashing into each other.

CPU – I/O Burst Cycle

What is a "burst" in this context?

- A **CPU burst** is a period when a process is actively using the CPU to do some work (like calculations).
- An **I/O burst** is when the process is waiting for something like data from a file, keyboard input, or a network response. While waiting, the CPU can be used by other processes.

What is "burst duration"?

- **Burst duration** refers to how long a process spends in one of these bursts — either a CPU burst or an I/O burst.

What is "frequency"?

- **Frequency** simply means how often something happens. In this case, it refers to how often bursts of a certain length (**short or long**) occur.

Explaining Burst Duration Frequency:

1. **CPU burst durations** usually follow a pattern:
 - **Short bursts** happen **more frequently**. This means most processes spend a small amount of time using the CPU.
 - **Long bursts** happen **less frequently**. Some processes may occasionally need the CPU for a long time.

2. Frequency Curve:

- Imagine you made a chart showing how many times a CPU burst lasts a certain amount of time (e.g., 1 second, 5 seconds, 10 seconds). The chart would likely show:
 - **Many short bursts**, like 1-second or 2-second bursts.
- **Fewer long bursts**, like 10-second or longer bursts.
- This is a **frequency curve**: It shows how often different lengths of bursts occur. In many cases, this curve looks like an **exponential curve**, where short bursts are much more common than long ones.

3. Why does this matter?

- If most processes have **short CPU bursts**, the system can quickly **switch** between many different processes, keeping the CPU busy.
- If a process has a **long CPU burst**, it might keep the CPU busy for a while, meaning **other processes have to wait**.

Example:

- **I/O-bound process**: Think of a process that frequently waits for files or user input. It uses the CPU only briefly, so it has **many short bursts** (like 1-2 seconds each).
- **CPU-bound process**: Imagine a process that does a lot of heavy calculations. It might need the CPU for a **long burst** (like 10 or 20 seconds), only pausing occasionally to wait for I/O.

Summary:

- **Burst duration** tells us how long each burst (CPU or I/O) lasts.
- **Frequency** tells us how often each burst happens.
- For most processes, there are **many short CPU bursts** and **a few long ones**.

By understanding these concepts, an operating system can decide **how to schedule processes efficiently** to keep the CPU busy and avoid wasting time.

Preemptive vs Nonpreemptive Scheduling

1. Nonpreemptive Scheduling:

- **What it means**: Once a process starts using the CPU, **it keeps using it until it finishes** or needs to wait (for example, waiting for data from a file).
- **Example**: Imagine you're reading a book. You don't stop until you finish a chapter (even if someone else is waiting to read). You finish your task before letting someone else use the book.
- **When it's used**: Old operating systems or simpler systems might use this.

2. Preemptive Scheduling:

- **What it means:** The operating system can **interrupt a process** at any time to give the CPU to another process.
- **Example:** Imagine you're reading a book, but every 5 minutes, someone tells you to stop and give the book to someone else. After a few minutes, you can start reading again when it's your turn.
- **When it's used:** Modern operating systems like Windows, macOS, and Linux use this, making sure the system is always running efficiently.

The Role of the Dispatcher

- The **Dispatcher** is like a **traffic controller** for processes. It decides which process gets the CPU.
- **What it does:**
 - It **stops** one process, **saves** its work, and then **starts** another process.
 - This switch happens quickly, so everything feels smooth on the computer.

Context Switches

- A **context switch** happens when the dispatcher switches from one process to another. The computer **saves** the work of the current process and **loads** the work of the **new one**.
- **Why it happens:** To make sure the CPU is always doing something useful, the computer keeps switching between processes.

Voluntary vs Nonvoluntary Context Switches:

- **Voluntary:** A process **chooses** to stop using the CPU because it's waiting for something, like data from a file.
- **Nonvoluntary:** The operating system **forces** a process to stop using the CPU, like when its time is up.

Summary:

- **Nonpreemptive:** A process uses the CPU until it's done or waiting.
- **Preemptive:** The CPU can be taken away from a process at any time to make the system faster.
- **Dispatcher:** The part of the operating system that switches between processes.
- **Context switch:** The action of saving one process's work and starting another.

This is how modern operating systems manage different processes efficiently

What is Scheduling Criteria?

When we choose a CPU scheduling algorithm (the method for deciding which process gets the CPU next), we need to measure how well the algorithm works. Different algorithms have different strengths and weaknesses. These **strengths and weaknesses are measured** using **criteria**, which help us judge which algorithm is best for the system.

Important Criteria to Consider:

1. CPU Utilization:

- **What it means:** We want to keep the CPU **busy** as much as possible, so the system is efficient.
- **How we measure it:** The percentage of time the CPU is **actively doing work**.
- **Goal:** The CPU should be busy between **40% and 90%** of the time in a real system.
- **Example:** If the CPU is idle too often, it's a waste; if it's always running, that's ideal.

2. Throughput:

- **What it means:** How many **processes** the system can complete in a certain amount of time.
- **How we measure it:** The **number of processes** finished per second or minute.
- **Goal:** More processes completed = better throughput.
- **Example:** If your system can complete 10 tasks per second, it's doing well.
- **Streaming Services (e.g., Netflix, YouTube):**
 - **Throughput importance:** The system must process and deliver video data continuously to ensure smooth playback. Higher throughput allows for uninterrupted streaming, even at higher video quality (e.g., 4K resolution).
 - **Why throughput matters:** If a system can handle a large number of video streams simultaneously without buffering, the user experience is significantly better.

3. Turnaround Time:

- **What it means:** The total time it takes to **complete a process** after it starts.
- **How we measure it:** The time from when a process is **submitted** until it's **finished**.
- **Goal:** Minimize the time spent on the process.
- **Example:** If you start a process at 1:00 PM and finish it at 1:10 PM, the turnaround time is 10 minutes.
- **IT Support/Troubleshooting:** If a user submits a request for technical support, the turnaround time would be the time between the request submission and the issue being resolved.
- **Goal:** Minimize the turnaround time to keep systems running smoothly and ensure quick responses for users in need of help.

Mathematically:

$$\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time}$$

The lower the turnaround time, the better the system's performance in terms of process management

4. Waiting Time:

- **What it means:** The amount of time a process spends **waiting** in the **ready queue** before getting the CPU.
- **How we measure it:** The time spent **waiting** (not doing actual work).
- **Goal:** Minimize waiting time to make the system faster.
- **Example:** If a process has to wait for 3 minutes before it gets CPU time, that's its waiting time.

5. Response Time:

- **What it means:** How long it takes for a process to **start responding** after it's been submitted. This is especially important for interactive systems (like when you click a button and want to see a result).
- **How we measure it:** The time it takes from when you **request** something until the first part of the result is shown.
- **Goal:** Minimize the time it takes to **start responding**.
- **Example:** If you click "play" in a game and it takes 2 seconds to start, the response time is 2 seconds.

$$\text{Response Time} = \text{First Response Time} - \text{Arrival Time}$$

What do we want to optimize?

- **Maximize:** CPU utilization and throughput (we want the CPU to work as much as possible and complete as many processes as it can).
- **Minimize:** Turnaround time, waiting time, and response time (we want to finish processes quickly, avoid long wait times, and respond fast).

Which is more important: Average or Extreme Values?

- In most cases, we want to **optimize averages**. For example, average waiting time or average response time.
- However, sometimes it's better to look at the **worst-case scenario**. For example, we might want to make sure that no process experiences an **extremely long wait** (minimize maximum response time).

Interactive Systems:

- For **interactive systems** (like your desktop or phone), it's often **more important to have predictable** response times rather than just the fastest average response time. People

generally prefer a system that responds in a fairly predictable way, even if it's not the fastest in every case.

Summary in Simple Terms:

- **CPU Utilization:** Keep the CPU working as much as possible.
- **Throughput:** Complete as many processes as possible.
- **Turnaround Time:** Make processes finish as quickly as possible.
- **Waiting Time:** Minimize how long a process waits for CPU time.
- **Response Time:** In interactive systems, make sure the system starts responding quickly.

Why is this important?

- Different scheduling algorithms focus on different criteria. So, depending on what the system needs (whether it's better throughput, faster response time, or less waiting), we might choose a different scheduling method.

Scheduling Algorithms

First-Come, First-Served Scheduling(FCFS)

What is FCFS?

The First-Come First-Serve (FCFS) scheduling algorithm is the simplest CPU scheduling algorithm. It works just like a line at a checkout counter in a store: the first person to arrive gets served first. In the case of a computer, this means the **first process to request CPU time will get the CPU first**, and it will continue running until it's done. After that, the next process in line will get the CPU, and so on.

Visualizing with a Gantt Chart:

You can introduce the **Gantt chart**, which is a graphical way of showing the scheduling of processes over time.

Pros and Cons of FCFS:

Pros:

- It's very simple and easy to implement.
- It's fair because processes are handled in the order they arrive.

Cons:

- The **average waiting time** can be high, especially if there's a process that requires a lot of CPU time at the front of the queue. This is called the **convoy effect**: long processes at the beginning can make all subsequent processes wait a long time.
- It doesn't consider the length of each process's burst time. So, if a short process arrives after a long one, it has to wait, even though it could have finished quickly.

$$\text{TAT} = \text{CT} - \text{AT}$$

$$\text{WT} = \text{TAT} - \text{BT}$$

$$\text{RT} = \text{LHS}(\text{GC}) - \text{AT}$$

Process Synchronization

1. Serial vs Parallel Execution

Serial Execution: Processes are executed one after another.

Parallel Execution: Multiple processes are executed simultaneously.

2. Cooperative vs Independent Processes

Cooperative Processes:

- Processes share some common resources, such as variables or memory.
- One process can affect the execution of another.

Example: Multiple processes accessing a shared variable or hardware (like a printer or scanner). Synchronization is necessary to prevent conflicts (deadlock, race conditions).

Independent Processes:

Processes do not share resources and do not affect each other.

Example: Different applications running on your computer without interacting with each other.

3. Synchronization in Processes

In cooperative processes, synchronization is crucial. If processes do not synchronize correctly, it can lead to inconsistent results or system crashes.

Synchronization Issues:

Deadlock: Processes wait indefinitely for resources held by each other.

Race Conditions: Multiple processes access shared data simultaneously, leading to unpredictable results.

Example: If two processes access a shared variable at the same time, the result may be inconsistent if not properly synchronized.

4. Types of Synchronization Methods

Mutexes: Locking mechanisms to ensure that only one process can access a critical section at a time.

Semaphores: Variables used to control access to shared resources.

Atomic Operations: Ensures that operations on shared resources are completed without interference.

Shared Variables in Parallel Execution

In parallel execution, when multiple processes access the same variable, synchronization must be ensured.

Example:

Suppose two processes modify a shared variable x.

Without synchronization, one process may overwrite the changes of the other, leading to errors.

Example Problem: Shared Variable Issue

P1

int s = 5

int x=s;

x++;

sleep(1)

s=x;

p2

int x=s;

x++;

sleep(1)

s=x;

- If two processes perform these operations simultaneously without synchronization, the final values of x and y may be incorrect.
- Proper Synchronization is needed to avoid race conditions.

Conclusion: Importance of Synchronization

Why synchronization is important:

- Without proper synchronization, processes that share resources can lead to errors, inconsistencies, or system crashes.
- Synchronization ensures that processes work in harmony, providing accurate results and efficient system operation.

Common Synchronization Problems:

Deadlock: Processes are blocked because they are waiting for each other to release resources.

Race Conditions: Unpredictable results due to unsynchronized access to shared data.

Methods to Synchronize:

Mutexes, Semaphores, and Atomic Operations are used to ensure only one process accesses shared resources at a time.

Summary

1. Cooperative processes share resources and require synchronization to avoid errors like race conditions.
2. Independent processes do not interact and do not require synchronization.
3. Synchronization is vital in parallel execution to avoid deadlock and ensure the integrity of data.
4. Proper use of synchronization tools like mutexes, semaphores, and locks is essential for maintaining system stability and performance.

Critical-Section Problem

Imagine you have multiple processes running on a computer. These processes may need to access and update shared data (e.g., files, memory) while they're running. But if multiple processes try to do this at the same time, problems arise — data could get corrupted or incorrect results might happen.

Critical Section (CS):

Each process has a special piece of code called a **critical section**, where it accesses and modifies shared data.

Key Rule:

Only one process can execute in its critical section at any time. If one process is working with shared data, no other process should be allowed to do the same at the same time.

The Three Requirements for Solving the Problem

A good solution to the critical-section problem must meet **three important rules**:

- **Mutual Exclusion:** If one process is in its critical section, no other process can be in its critical section at the same time. This ensures that only one process accesses shared data at a time.
- **Progress:** If no process is in the critical section and multiple processes want to enter, the system must decide quickly and fairly which process will enter next. The decision should not be postponed indefinitely.

```
while (true)
```

```
{
```

```
    entry section
```

```
        critical section
```

```
    exit section
```

```
    remainder section
```

```
}
```

- **Bounded Waiting:** Once a process asks to enter the critical section, there should be a limit on how long it will have to wait before it gets in. This prevents a process from waiting forever if other processes keep entering their critical sections.

3. Example of a Race Condition

Race condition is when multiple processes try to update shared data at the same time, and the order of execution leads to an unexpected result.

- **Example with PID (Process Identifier):**
 - When a process creates a new child process, it needs a unique process ID (PID).
 - If two processes ask for the next available PID at the same time, there's a risk they'll both receive the same PID, which is a **race condition**.
 - Without synchronization, this could lead to **duplicate process IDs**, causing confusion and errors.

LOCK

```
do {
    acquire lock

    Critical Section

    Release lock
}
```

1. While (Lock == 1);
 2. Lock = 1
 3. **Critical Section**
 4. Lock = 0
- ENTRY CODE
- EXIT CODE
- Execute in user mode
 - Multi-process solution
 - No mutual exclusion (guaranteed).

Semaphore

Introduction

A **semaphore** is a synchronization tool used to control access to shared resources in a concurrent system such as a multitasking operating system. It helps prevent **race conditions**, where multiple processes try to access and modify shared resources simultaneously, leading to unpredictable behavior.

Definition of Semaphore

A **semaphore** is an **integer variable** used by multiple concurrent cooperative processes in a **mutual exclusive manner** to achieve synchronization.

Why Semaphores?

In a multi-processing environment, several processes may require access to shared resources. Without synchronization, **race conditions** can occur, leading to issues like:

- **Data inconsistency**
- **Deadlocks**
- **Starvation**

Semaphores provide a mechanism to ensure that only a limited number of processes can access the critical section at a time.

There are two main types of semaphores:

1. Counting Semaphore

- Can take any integer value from $-\infty$ to $+\infty$.
- Used when multiple instances of a resource are available.

For example, if a printer can only serve 3 jobs at once, the counting semaphore might be initialized to 3. Every time a process uses the printer, the semaphore is decremented. When the printer is free again, the semaphore is incremented.

Binary Semaphore (Mutex)

- Can take only two values: **0 and 1**.
- Works as a **lock (mutual exclusion)**.
- Used to allow or block a single process at a time.

Critical Section and Process Synchronization

A **critical section** is a portion of the code where a process accesses shared resources. To avoid conflicts, a process must follow these steps:

1. **Entry Section** – Request permission to enter the critical section.
2. **Critical Section** – Execute the shared code.
3. **Exit Section** – Release the resource after completion.
4. **Remainder Section** – Continue execution after leaving the critical section.

To implement this, we use **semaphore operations**.

Semaphore Operations

1. P() Operation (Proberen/Wait/Down)

- Decrements the semaphore value.
- If the value becomes negative, the process **waits** (blocks) until it becomes positive.
- Ensures mutual exclusion in critical section entry.

P(S) {

$S = S - 1;$

 if ($S < 0$)

 process waits in the queue;

}

2. V() Operation (Verhogen/Signal/Up)

- Increments the semaphore value.
- If a process is waiting, it is **unblocked**.

V(S) {

$S = S + 1;$

 if ($S \leq 0$)

 wake up a waiting process;

}

Alternative Naming Conventions:

P() Operation	V() Operation
Wait()	Signal()
Down()	Up()
Lock()	Unlock()

Working of Counting Semaphore

A **counting semaphore** allows multiple processes to access a resource based on the semaphore value.

Example: Allowing 3 Processes in Critical Section

- Initial $S = 3$
- **P1 arrives:** $S = S - 1 = 2$
- **P2 arrives:** $S = S - 1 = 1$
- **P3 arrives:** $S = S - 1 = 0$ (Last allowed)
- **P4 arrives:** $S = S - 1 = -1$ (Blocked)

Now, if **P1 exits**:

- $S = S + 1 = 0$
- **P4 is unblocked** and enters.

Example of Binary Semaphore (Mutex)

A binary semaphore works as a **lock mechanism** where only one process can enter at a time.

- Initial $S = 1$ (Unlocked)
- **P1 arrives:** $S = S - 1 = 0$ (Locked)
- **P2 arrives:** $S = S - 1 = -1$ (Blocked)

Now, if **P1 exits**:

- $S = S + 1 = 1$ (Unlocked)
- **P2 is unblocked** and enters.

Example Problem

Given:

- Initial semaphore value $S = 10$
- 6 $P()$ operations
- 4 $V()$ operations

Final value of S? $S = 10$

$S = 10 - 6$ // After $P()$ operations

$S = 4 + 4$ // After $V()$ operations

Final $S = 8$

Practical Applications of Semaphores

- **Process Synchronization:** Ensuring that multiple processes do not access shared resources simultaneously.
- **Producer-Consumer Problem:** Used to manage buffers where producers add items and consumers remove them.
- **Dining Philosophers Problem:** Ensuring that philosophers do not pick up both forks simultaneously, avoiding deadlock.
- **Readers-Writers Problem:** Allowing multiple readers but only one writer at a time

Advantages and Disadvantages of Semaphores

Advantages

Prevents Race Conditions
Ensures Process Synchronization
Efficient Resource Utilization

Disadvantages

Deadlocks & Starvation – If not managed properly.
Busy Waiting – Inefficient CPU usage if not implemented using blocking mechanisms.
Complex to Implement – Requires careful handling to avoid logical errors.

A **deadlock** occurs when two or more processes are **stuck in a cycle**, each waiting for a resource held by another process, and none can proceed.

Real-Life Example: Traffic Jam at an Intersection

Imagine a **four-way intersection** with no traffic lights, where each car (representing a process) must move forward but cannot because another car is blocking the way.

Car A → Wants to go straight but is blocked by **Car B**

Car B → Wants to turn but is blocked by **Car C**

Car C → Wants to move but is blocked by **Car D**

Car D → Wants to move but is blocked by **Car A**

Since **no one can move**, they are stuck in a cycle. This is a **deadlock** in real life!

Deadlock in a Computer System

Imagine two processes:

- **Process 1** has **Resource A** and needs **Resource B** to continue.
- **Process 2** has **Resource B** and needs **Resource A** to continue.

Since both are waiting for each other, **neither can proceed**, leading to a deadlock.

How to Solve Deadlocks?

1. **Avoidance** – Implement rules (like traffic signals) to prevent circular waiting.
2. **Detection & Recovery** – Detect deadlock and force processes to release resources (like removing a car from the intersection).
3. **Prevention** – Assign resources in a specific order to ensure no circular wait.

System Model

Imagine a **library** where multiple students (threads) want to use limited resources (books, computers, or study rooms).

- Each student **requests a resource** (e.g., a book).
- They **use the resource** to study.
- Once done, they **return the resource** for others to use.

This is exactly how a **computer system model** works, where **multiple threads** request **finite resources** like CPU time, files, and I/O devices.

Define: A **system model** consists of:

- **Resources** (e.g., CPU cycles, memory, files, network connections).

- **Threads/Processes** competing for these resources.
- **Rules for accessing resources** to avoid conflicts.

The system **manages** how resources are allocated and released to ensure smooth execution.

Resource Types with Examples

A **resource** can be anything that a thread needs to complete a task. Resources are divided into **types (classes)**, and each type has **multiple instances**.

Resource Type	Example	Number of Instances
CPU	4 Cores	4 Instances
Network Interface	Wi-Fi, Ethernet	2 Instances
Mutex Locks	Lock for a file	1 per file
I/O Devices	Printers, Hard Drives	2 Printers, 1 HDD

Example: If a computer has **four CPU cores**, the **CPU resource type** has **four instances**. If a thread requests CPU time, any of the four cores can be assigned.

Three-Step Process of Resource Allocation

A thread **must** follow this sequence when using a resource:

Request: The thread asks for a resource.

- If the resource is **available**, it is allocated.
- If it's **not available**, the thread **waits**.

Use: The thread performs operations on the resource.

- Example: If it's a **mutex lock**, the thread enters a **critical section**.

Release: The thread **frees** the resource after use.

- This allows **other threads** to use it.

Example:

- A student requests a book ☐ from the library.
- They read the book.
- After finishing, they return the book for others to use.

How the System Keeps Track of Resources

he **operating system maintains a table** to track:

- Which resources are **free**.
- Which are **allocated** and to **which thread**.
- If a resource is requested but not available, the thread **waits in a queue**.

Synchronization Tools & Deadlock Risk

Many resources involve **synchronization mechanisms** like **mutex locks** and **semaphores**. If used incorrectly, they can lead to **deadlocks** where threads **wait indefinitely** for resources.

Example of Deadlock:

- **Thread A** locks **Resource 1** and waits for **Resource 2**.
- **Thread B** locks **Resource 2** and waits for **Resource 1**.
- They are **stuck forever**—deadlock!

To avoid this, resources must be requested and released in a proper order.

Summary

- The **system model** describes how **resources** are **requested, used, and released** by threads.
- Each resource belongs to a **type (class)** and has **multiple instances**.
- The OS **manages resource allocation** using a **table**.
- **Synchronization tools** (mutexes, semaphores) help, but improper use can cause **deadlocks**.

Real-world comparison:

- Library (Students = Threads, Books = Resources).
- Computer (Threads = Programs, Resources = CPU, Memory, I/O devices).

The Dining-Philosophers Problem



- Imagine **5 philosophers** sitting around a circular table.
- Each philosopher thinks and occasionally needs to **eat**.
- To eat, a philosopher needs **two forks** (one from each side).
- There are **5 forks** in total, one between each pair of philosophers.

The Challenge

- Philosophers spend time **thinking** (doing nothing) and **eating**.
- They must **share resources** (forks), but there's a limitation.
- The problem is how to avoid **deadlock** (where no philosopher can eat) and **starvation** (where a philosopher never gets to eat).

Deadlock:

- Occurs when every philosopher is holding one fork and waiting for another, causing a cycle of dependency.

Starvation:

- Happens when a philosopher can't get both forks and thus never gets to eat.

Concurrency:

- Philosophers are **competing for limited resources** (forks) but must avoid blocking each other.

Solution Approaches

1. Mutex/Lock Approach

- Philosophers request a lock for each fork before eating.
- If a philosopher can't acquire both locks, they release the locks and try again later.

2. Resource Hierarchy Solution

- Philosophers pick up the lower-numbered fork first, and then the higher-numbered one.
- This avoids circular wait, a key condition for deadlock.

Philosophical Model

Title: Philosophical Model

Explanation:

- **Think of philosophers** as processes, and **forks** as resources.
- This problem mirrors issues in **multithreading** and **resource allocation** in computer science.

Pseudocode or a flowchart showing how philosophers pick up and release forks.:

```
while (true) {

    pick up left fork;

    pick up right fork;

    eat;

    put down left fork;

    put down right fork;

    think;

}
```

Real-World Applications

The **Dining Philosophers Problem** is used to illustrate **synchronization issues** in:

- Database management
- Operating systems (especially resource allocation)
- Distributed systems

Summary

- The problem demonstrates the challenges of **resource sharing** in a concurrent environment.
- Solutions must handle both **deadlock** and **starvation**.
- Concepts like **locks**, **mutexes**, and **resource hierarchy** are key in resolving concurrency issues.