**UNIT-3:**

**INTER-PROCESS COMMUNICATION AND DEADLOCK:**

1. **Deadlock:**

* A deadlock is a situation in an operating system where two or more processes are waiting indefinitely for resources that are being held by each other — so none of them can proceed.
* A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process.
* **Real-life Analogy:**
* Imagine two people trying to cross a narrow bridge from opposite directions. Neither can move until the other moves — they are stuck forever → Deadlock.
* **Conditions for Deadlock**
* According to **Coffman’s conditions, four conditions** must hold simultaneously for a deadlock to occur:
* **Deadlock can occur only if all four conditions hold simultaneously.**

1. Mutual Exclusion

* **Only one process can use a resource at any given time.**
* Some resources are **non-shareable** — for example, printers, CPU, or file locks.
* If another process requests the same resource, it must wait until it’s released.
* Example:
* Process P1 is using a printer.
* P2 wants the printer but can’t access it until P1 releases it.
* This **exclusive control** is essential in many cases, but also **creates potential for deadlock**.

### **Hold and Wait**

* **A process is holding at least one resource and is waiting to acquire additional resources** that are currently being held by other processes.
* The process does **not release the resources** it’s already holding while waiting.
* Example:
* P1 holds Resource R1 and needs R2.
* P2 holds R2 and needs R1.
* Both hold one resource and **wait** for the other → deadlock possibility.
* This condition creates a “waiting chain”.

### **No Preemption**

* **Resources cannot be forcibly taken away** from a process; they must be released voluntarily.
* If a process is holding a resource, the system **cannot force it to release it**.
* Example:
* If a process holding a file lock does not release it, the OS cannot preempt that file from it.
* This allows the process to **block others** even when it is waiting.
* If preemption were allowed, deadlock could be resolved by taking away resources.

1. **Circular Wait**

* **A set of processes {P1, P2, ..., Pn} exists such that each process is waiting for a resource held by the next one**, forming a circular chain.
* Each process in the cycle **waits** for a resource **held by another process** in the same cycle.
* Example:

P1 → waiting for R2 (held by P2)

P2 → waiting for R3 (held by P3)

P3 → waiting for R1 (held by P1)

* This forms a **circular wait** → no one can proceed → **deadlock**.

## Example in Real Life:

### Example: Dining Philosophers

* 5 philosophers, 5 forks
* Each needs **two forks** to eat
* Each picks up one fork and waits for the other
* **All four conditions are met** → deadlock possible

**Dining philosophers:**

* It is a **synchronization and concurrency** problem proposed by **Edsger Dijkstra** to illustrate the challenges of allocating limited resources among multiple processes **without causing deadlock or starvation**.
* **Problem Setup**
* Imagine **5 philosophers** sitting around a circular table.
* Each philosopher alternates between two states:
  + **Thinking**
  + **Eating**
* There is **1 fork between each pair** of philosophers (so 5 forks in total).
* To **eat**, a philosopher needs **two forks** — one on the left and one on the right.

## The Problem

If all philosophers pick up the **left fork first**, they will all be **waiting for the right fork**, which is **already taken** by their neighbor.

This leads to a **deadlock**, where:

* No philosopher can eat.
* No philosopher can release their fork.
* Everyone is stuck forever.

## Issues to Solve

| Problem | Description |
| --- | --- |
| **Deadlock** | All philosophers are waiting indefinitely for a resource. |
| **Starvation** | A philosopher may **never get a chance to eat**. |
| **Concurrency** | Multiple philosophers may try to eat at the same time. |

## Possible Solutions to the Problem

### **Resource Hierarchy (Numbering Forks)**

* Number forks from 1 to 5.
* Philosophers must pick up the **lower-numbered fork first**.
* This breaks the **circular wait condition** and **prevents deadlock**.

### **Allow Maximum 4 Philosophers to Sit**

* If only **4 out of 5** philosophers are allowed to sit at the table at the same time, at least one fork will always be free.
* This avoids deadlock.

### **Odd-Even Solution (Asymmetric)**

* Odd-numbered philosophers pick up **left then right**.
* Even-numbered philosophers pick up **right then left**.
* This **breaks symmetry**, avoiding circular wait.

### **Using Semaphores (Code Implementation)**

semaphore fork[5] = {1, 1, 1, 1, 1}; // all forks available

semaphore mutex = 1;

function philosopher(i):

while (true) {

think();

wait(mutex);

wait(fork[i]); // take left fork

wait(fork[(i+1)%5]); // take right fork

signal(mutex);

eat();

signal(fork[i]); // put down left fork

signal(fork[(i+1)%5]); // put down right fork

}

This code prevents **race conditions** but needs more logic to avoid **deadlock/starvation**.

## Real-Life Analogy

You and 4 friends are eating pizza. Each needs **two plates** to eat (1 for pizza, 1 for sauce). But there are only 5 plates total. If all of you grab one plate and wait for a second, nobody eats → **deadlock**.

* **Deadlock Characterization**
* **Deadlock Characterization** refers to the **conditions and models** used to identify and describe deadlocks in an operating system.
* It is mainly based on:

1. **Coffman’s four necessary conditions**
2. **Resource Allocation Graph (RAG)**

## **Coffman’s Four Necessary Conditions**

As explained before, for a deadlock to occur, **all four conditions must hold simultaneously**:

| Condition | Meaning |
| --- | --- |
| **Mutual Exclusion** | A resource can be used by only one process at a time |
| **Hold and Wait** | A process holding one resource is waiting for another |
| **No Preemption** | Resources can't be forcibly taken away |
| **Circular Wait** | A closed chain of processes exists, each waiting for the next |

If even **one condition is broken**, **deadlock will not occur**.

## **Resource Allocation Graph (RAG)**

A **graphical way to represent processes and resources** in a system.

### Components:

* **Process (P)**: Represented by a **circle**.
* **Resource (R)**: Represented by a **rectangle or square**, with **dots** for multiple instances.

### Edges:

* **Request Edge (P → R)**: Process is **waiting** for a resource.
* **Assignment Edge (R → P)**: Resource is **allocated** to a process.

### Example 1: No Deadlock

P1 → R1 → P2

(No cycle → No deadlock)

### Example 2: Deadlock

P1 → R1 → P2 → R2 → P1

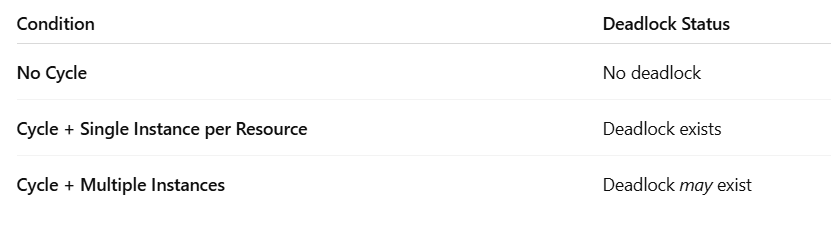
(Cycle exists → Deadlock possible)

### ➕ RAG with Multiple Instances

When resources have multiple instances:

* **Cycle ≠ Deadlock**, but **Cycle + Unavailability = Potential Deadlock**

## 🔁 Cycle in RAG



## Example Scenario

**Processes:** P1, P2  
**Resources:** R1, R2

* P1 holds R1 and requests R2
* P2 holds R2 and requests R1

Graph:

P1 → R2

R1 → P1

P2 → R1

R2 → P2

🔁 Forms a **cycle** → **Deadlock**.

## Real-Life Analogy (RAG)

Imagine 3 people holding 3 different pens and each asking for the next person's pen. No one can proceed → **circular wait** → **deadlock**.

## **Strategies to Handle Deadlock**

**There are 4 methods to deal with deadlocks:**

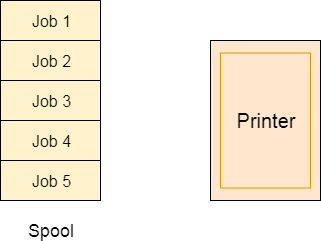
## **Deadlock Ignorance**

* Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses.
* In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.
* The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

## **Deadlock prevention**

* Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously.
* If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.
* The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.
* How prevent-
* Mutual Exclusion
* Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.
* Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

* Hold and Wait
* Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.
* However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.
* **!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)**
* Challenges:
* A process cannot be aware of all necessary resources prior to execution as it carries out instructions one at a time.
* Another issue with releasing all of a process's resources at once is that they might not be useful to other processes and are relinquished needlessly.
* For instance, Resource3 is released needlessly when Process1 releases both Resource2 and Resource3, as Process2 does not require it.

## No Preemption

* Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.
* This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.
* Challenges:
* These methods are troublesome since the process may be actively utilizing these resources, and stopping it via preempting might lead to inconsistent results.
* For instance, a file will remain useless and inconsistent if a process writing to it has its access terminated before it has finished updating the file.
* It is ineffective and time-consuming to submit requests for all resources repeatedly.

## Circular Wait

* To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.
* Challenges:
* Because multiple processes may value the same resource differently, it is challenging to give a relative priority to them.
* For instance, a printer may receive a greater priority from a document processor than from a media player. Depending on the circumstance and use case, resources are prioritized differently.

## **Deadlock avoidance**

* In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.
* In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.
* **Banker’s Algorithm** is used**.**

**Banker algorithm**

* The **Banker's Algorithm** is a **deadlock avoidance algorithm** developed by **Edsger Dijkstra**.
* It is named so because it works similar to how a banker allocates cash to customers without running out of reserves.
* **Goal**: Ensure that the system never enters an **unsafe state** that could lead to deadlock.
* Basic Idea
* Each process must **declare its maximum resource needs** in advance.
* The OS allocates resources **only if** the system remains in a **safe state**.
* If granting a resource request leads to an **unsafe state**, the request is denied **even if resources are available**.
* Formula:



* Steps in Banker’s Algorithm

### A. **Safety Algorithm** (to check if the system is in a safe state)

1. **Initialize**:
   * Work = Available
   * Finish[i] = false for all i
2. **Find a process** i such that:
   * Finish[i] == false
   * Need[i] <= Work
3. If such i is found:
   * Work = Work + Allocation[i]
   * Finish[i] = true
   * Repeat step 2
4. If all Finish[i] = true → system is in a **safe state**  
   Else → **unsafe**

### B. **Resource-Request Algorithm** (when a process makes a request)

Let process Pi make a request Request[i]

1. If Request[i] <= Need[i] → OK  
   Else → Error (asking more than declared)
2. If Request[i] <= Available → Temporarily allocate resources:
   * Available = Available – Request[i]
   * Allocation[i] = Allocation[i] + Request[i]
   * Need[i] = Need[i] – Request[i]
3. Check safety using the **Safety Algorithm**
   * If safe → allocation granted
   * If unsafe → rollback and deny request

## Numerical Example

## Advantages

* Guarantees system will never enter deadlock.
* Safer than deadlock detection & recovery.

## Disadvantages

* Requires processes to declare max resource need in advance.
* High overhead for large number of processes/resources.
* Not practical for real-time or modern complex systems.

## **Deadlock detection and recovery**

* This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

## Deadlock vs Starvation

| **S.No.** | **Deadlock** | **Starvation** |
| --- | --- | --- |
| 1 | All processes involved are **blocked permanently**. | A process is **indefinitely delayed** but not blocked permanently. |
| 2 | Caused due to **circular wait and resource holding**. | Caused due to **unfair resource allocation**. |
| 3 | No process proceeds; system is **stuck**. | Some processes may proceed, but **one or more wait endlessly**. |
| 4 | All four Coffman conditions must hold: Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait. | Occurs when a process **never gets a chance** due to priority or scheduling issues. |
| 5 | Example: P1 waits for R2, P2 waits for R1 → cycle → deadlock. | Example: Low-priority process always preempted by high-priority ones. |
| 6 | Deadlock affects **all involved processes**. | Starvation may affect **only specific processes**. |
| 7 | Can be detected and **recovered** using algorithms (like Banker’s Algorithm, Wait-for Graph). | Solved by using **fair scheduling algorithms** (e.g., aging technique). |
| 8 | More severe and **less frequent** in practice. | Less severe but can be **more frequent**. |
| 9 | Needs **resource management solutions**. | Needs **scheduler-level solutions**. |
| 10 | System must **break at least one** deadlock condition to prevent it. | Can be resolved by **adjusting priorities dynamically**. |

1. Principle of concurrency
2. Synchronization:

* **Synchronization** is the **coordination or orderly execution of multiple processes or threads** that share common resources, to **prevent conflicts**, **inconsistencies**, or **data corruption**.
* **Synchronization** refers to the technique of ensuring that two or more concurrent processes or threads **do not execute specific parts of a program at the same time**, especially when they access **shared resources**.
* Why is Synchronization Needed?

When multiple processes:

* Access shared memory or variables
* Use shared files, buffers, or devices

Without synchronization:

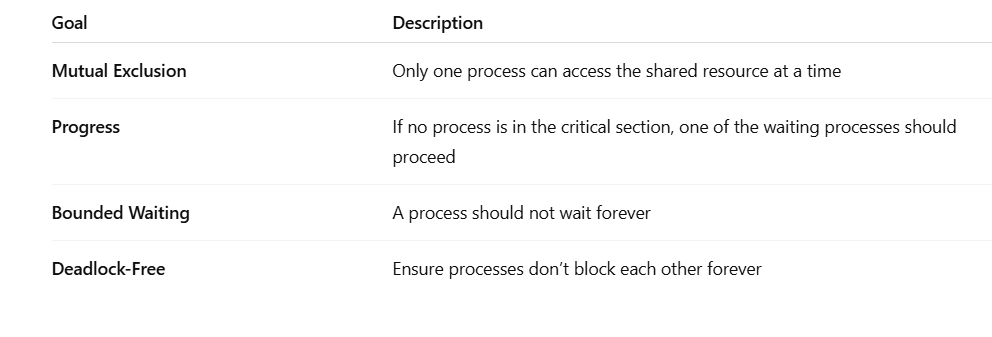
* One process might **change data** while another is reading it → **Race Condition**
* Multiple processes might **enter the critical section at the same time** → **Data inconsistency**
* May lead to **deadlocks or starvation**

## Real-life Analogy

Imagine a **shared bathroom** in a hostel:

* Only **one person** should be inside at a time (mutual exclusion).
* If many try to enter at once without coordination, **conflict arises**.
* A **lock** (mutex) ensures only one person uses it at a time.

## Goals of Synchronization



## Key Concepts

### **Critical Section**

* A block of code that accesses shared resources.
* Only one process should execute it at a time.

### **Entry and Exit Section**

* Code executed before/after entering the critical section.
* Used to **request or release** control of the critical section.

### **Race Condition**

* When multiple threads access and change shared data concurrently and the result depends on the timing.
* Synchronization Mechanisms

| Mechanism | Description |
| --- | --- |
| **Mutex (Mutual Exclusion Object)** | Allows only one thread to access a section |
| **Semaphore** | Integer variable to control access (binary or counting) |
| **Monitor** | High-level abstraction with mutual exclusion + condition variables |
| **Spinlock** | Busy-wait lock for short critical sections |

## Synchronization Problems

## **Producer–Consumer Problem**

* One or more **producers** generate data and place it in a **bounded buffer**.
* One or more **consumers** take data from the buffer.
* If buffer is **full**, producer waits.
* If buffer is **empty**, consumer waits.
* Solved using **semaphores**: mutex, empty, full.
* **Solved by**: Mutual exclusion, counting semaphores.

## **Dining Philosophers Problem**

* 5 philosophers sit at a table with 5 forks.
* Each philosopher needs **2 forks** to eat.
* If all philosophers pick up one fork and wait for the next, it causes **deadlock**.
* Also risks **starvation**.
* **Solved by**: Resource hierarchy, semaphore limiting access to 4 philosophers.

## **Readers–Writers Problem**

* Shared resource like a file or database.
* Multiple **readers** can access simultaneously.
* Only one **writer** can modify at a time — and not while readers are active.
* **Problem**: Prioritize fairness — avoid **reader or writer starvation**.
* **Solved by**:
* Reader-preference
* Writer-preference
* Fair-read-write locking

## **Sleeping Barber Problem**

* A barber sleeps when no customers are in the shop.
* If a customer arrives and the barber is asleep, they wake him up.
* If all waiting chairs are full, the customer leaves.
* Controls waiting, sleeping, and waking using synchronization.
* **Solved by**: Semaphores and condition variables.

## **Bounded Buffer Problem**

* Buffer of fixed size (N) shared by producer and consumer.
* Ensures **buffer doesn’t overflow or underflow**.
* Essentially the same as the Producer–Consumer Problem.

1. Critical section problem
2. **Semaphores**

* A **semaphore** is a **synchronization tool** used to control access to a **shared resource** by multiple processes in a **concurrent system** such as an operating system.
* It helps **prevent race conditions**, **deadlocks**, and ensures **mutual exclusion**.
* A **semaphore** is an **integer variable** that is **accessed and modified using two atomic operations**:
* wait() or P() (proberen = to test)
* signal() or V() (verhogen = to increment)
* **Semaphores are used for:**
* **Mutual exclusion** (only one process enters the critical section at a time)
* **Process synchronization** (coordinate order of execution)
* **Resource counting** (e.g., number of printers or buffer slots)

## **Real-Life Analogy**

Imagine a **parking lot** with 5 spots:

* Only 5 cars can enter.
* Each car entering **decreases the count**.
* If count = 0, cars must **wait** outside.
* When a car leaves, the count is **increased**.

The **count variable** is a **semaphore**.

## **Operations on Semaphore**

### wait(S) (also called P operation)

wait(S):

while (S <= 0)

; // busy wait (or block)

S--;

* Decreases the semaphore value by 1.
* If S > 0, the process continues.
* If S == 0, the process **waits** until S > 0.

### signal(S) (also called V operation)

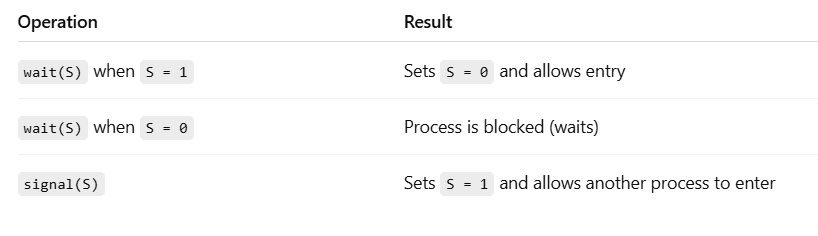
signal(S):

S++;

* Increases the semaphore value by 1.
* If other processes are waiting, **wake one** of them.
* **Types of Semaphores**

1. Binary semaphores

* A binary semaphore can take only two values: 0 and 1.
* It is typically used to enforce mutual exclusion, like a lock.
* Use Case:
* When only one process or thread can enter the critical section at a time.
* Used in situations like:
  + Accessing a file
  + Writing to a database
  + Using a printer
* Working:



* Example Code:

semaphore S = 1;

wait(S); // Lock

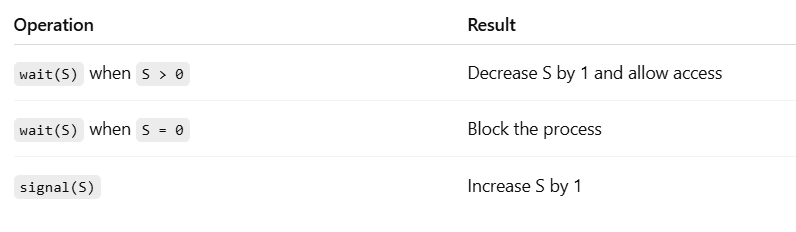
// Critical Section

signal(S); // Unlock

* Characteristics:
* Works like a mutex (mutual exclusion)
* Ensures only one process can access a resource at a time

1. Counting semaphores

* A counting semaphore is a semaphore whose value can range over an unrestricted domain (usually from 0 to N).
* It is used to control access to a resource that has multiple instances.
* Use Case:
* When you have more than one identical resource:
  + Printers (say 3 printers → S = 3)
  + Buffer slots (like in the producer-consumer problem)
* Working:



* Example Code:

semaphore printers = 3;

wait(printers); // Use one printer

// Printing job

signal(printers); // Release printer

* Characteristics:
* Allows multiple processes to access resources in parallel
* Keeps track of how many units of a resource are available

| **S.No.** | **Feature** | **Binary Semaphore** | **Counting Semaphore** |
| --- | --- | --- | --- |
| 1️⃣ | **Value Range** | Only 0 or 1 | Can be any non-negative integer (e.g., 0 to N) |
| 2️⃣ | **Purpose** | Used for **mutual exclusion** | Used for **resource counting** |
| 3️⃣ | **Alternate Name** | Also known as **Mutex Semaphore** | Also called **General Semaphore** |
| 4️⃣ | **Access Allowed** | Allows **only one** process/thread at a time | Allows **multiple** processes/threads (up to N) |
| 5️⃣ | **Typical Use Case** | Locking critical sections, single resource access | Managing access to a pool of identical resources |
| 6️⃣ | **Initialization** | Initialized with value 1 | Initialized with value = number of resources |
| 7️⃣ | **Concurrency Support** | No parallel access (exclusive use only) | Supports parallel access (up to N processes) |
| 8️⃣ | **Example Scenario** | Writing to a shared file | Allocating multiple printers/buffers |
| 9️⃣ | **Wait Block Condition** | If value is 0, process **waits** | If value is 0, process **waits** |
| 🔟 | **Use in OS Problems** | Used in **mutual exclusion** problems | Used in **Producer–Consumer**, **Readers–Writers** |

## **Example: Producer-Consumer Using Semaphores**

### Semaphores:

* mutex = 1 → for mutual exclusion
* empty = N → number of empty slots in buffer
* full = 0 → number of filled slots in buffer

### Producer Process:

wait(empty); // Decrease empty slot

wait(mutex); // Lock buffer

// Add item to buffer

signal(mutex); // Unlock buffer

signal(full); // Increase full slot

### Consumer Process:

wait(full); // Decrease full slot

wait(mutex); // Lock buffer

// Remove item from buffer

signal(mutex); // Unlock buffer

signal(empty); // Increase empty slot

## **Common Problems with Semaphores**

| Problem | Description |
| --- | --- |
| **Deadlock** | Two processes waiting forever for each other |
| **Starvation** | One process never gets the semaphore |
| **Busy Waiting** | CPU cycles wasted while waiting |

These can be avoided using **blocking semaphores**, **proper order**, and **priority scheduling**.

1. Inter process communication model and schemes

* **Inter-Process Communication (IPC)** is the **mechanism** that allows **processes to communicate and coordinate** with each other **in a system**.
* When **two or more processes** need to **share data** or **coordinate actions**, they use **IPC**.
* **Why is IPC Needed?**
* Share **data/information** between processes
* Coordinate **execution/order** of processes
* Manage **dependencies** between tasks
* Ensure **resource sharing** (files, memory, CPU)
* Prevent **race conditions and deadlocks**

## **IPC Models (Types)**

There are **two main IPC models**:

| IPC Model | Description | Example |
| --- | --- | --- |
| 1️⃣ **Message Passing** | Processes **send and receive messages** to communicate | Client-Server communication |
| 2️⃣ **Shared Memory** | Processes **share a memory segment** and read/write data there | Buffer between producer and consumer |

### **Message Passing Model**

* Processes **communicate by sending messages** using system calls.
* Key Features:
* No shared memory
* Communication is **explicit**
* OS provides support for **send()** and **receive()** operations
* Example:

Process A --> send(message) --> Kernel --> receive() --> Process B

* Used in:
* Microkernels
* Distributed systems
* Client-server apps
* Advantages:
* Simpler, no memory conflict
* Suitable for remote processes (e.g., over network)
* Disadvantages:
* Slower than shared memory
* More overhead in sending/receiving

### **Shared Memory Model**

* Multiple processes **access a common memory space** to exchange data.
* Key Features:
* Communication via a **shared buffer**
* Requires **synchronization** (using semaphores/mutexes)
* Faster than message passing
* Example:

Producer Process --> writes to shared buffer

Consumer Process --> reads from shared buffer

* Used in:
* Producer–Consumer problem
* Thread communication
* Operating systems with tight performance needs
* Advantages:
* Fast (direct memory access)
* Efficient for large data exchange
* Disadvantages:
* More complex to implement
* Needs synchronization to avoid race conditions

## **IPC Mechanisms/Schemes** (Also Called IPC Techniques)

* **IPC (Inter-Process Communication) mechanisms/schemes** are the **tools and techniques** provided by an Operating System to enable processes to **communicate, coordinate**, and **share data** effectively.
* IPC mechanisms are used for both **message passing** and **shared memory** models.
* **Types of IPC Mechanisms/Schemes**

### **Pipes**

* **Unidirectional** communication channel (from one process to another)
* Used for communication between **related processes** (parent-child)
* **Temporary**, created using pipe() system call
* Example:

$ ls | grep "file"

* In C:

int fd[2];

pipe(fd);

* Limitations:
* Only parent-child communication
* One-way communication

### **Named Pipes (FIFOs)**

* Special kind of pipe with a **name in the file system**
* **Persistent** and used between **unrelated processes**
* Created using mkfifo() in UNIX/Linux
* Example:

$ mkfifo mypipe

$ echo "Hello" > mypipe # Writing

$ cat mypipe # Reading

* Advantage:
* More flexible than unnamed pipes

### **Message Queues**

* A **queue structure** maintained by the **kernel** for storing messages
* Allows processes to **send** and **receive** messages asynchronously
* Message IDs and **priorities** can be set
* In C (Linux):

msgget(), msgsnd(), msgrcv()

* Used in:
* Multi-process applications needing ordered message passing

### **Shared Memory**

* Fastest IPC mechanism
* A **common memory area** is mapped between multiple processes
* Processes can **read/write** directly to shared memory
* Requires **synchronization** (semaphore or mutex)
* In C:

shmget(), shmat(), shmdt()

* Used in:
* Producer-consumer, data buffers, games, real-time systems

### **Semaphores**

* Used to **synchronize access** to shared resources
* Prevents **race conditions**
* Two types:
  + **Binary Semaphore** (0/1)
  + **Counting Semaphore** (0 to N)
* In C:

semget(), semop(), semctl()

* Used in:
* Critical sections, mutual exclusion, shared memory protection

### **Sockets**

* Used for **communication between processes over a network**
* Supports **inter-process** or **inter-machine** communication
* Can be **stream-oriented (TCP)** or **datagram-oriented (UDP)**
* In C:

socket(), bind(), listen(), accept(), connect(), send(), recv()

* Used in:

Chat applications, web servers, client-server systems

### **Signals**

* Asynchronous **notification mechanism**
* A signal is **sent to a process** to notify it of an event (like termination, interruption)
* Example signals:

SIGINT, SIGTERM, SIGKILL, SIGSEGV

* In C:

signal(SIGINT, handler\_function);

* Used in:

Interrupt handling, alarms, software timers

### **Memory-Mapped Files**

* Map a **file** to memory so multiple processes can **read/write** data
* Combines benefits of **file I/O** and **shared memory**
* In C:

mmap(), munmap()

* Used in:

Shared configuration files, image/video editing software