A.O.S. Question Bank

❖ 2 Marks Questions

Q.1.) What is CPU Scheduling Algorithms?

Ans :- CPU Scheduling Algorithms are strategies used by the operating system to determine the order in which processes are executed by the CPU. These algorithms help in maximizing CPU utilization, reducing waiting time, and improving system performance.

Types of CPU Scheduling Algorithms:

1. **First-Come, First-Served (FCFS)** – Processes are executed in the order they arrive.
2. **Shortest Job Next (SJN)** – The process with the shortest execution time is scheduled first.
3. **Round Robin (RR)** – Each process gets a fixed time slice (time quantum) in a cyclic order.
4. **Priority Scheduling** – Processes are scheduled based on priority levels.
5. **Multilevel Queue Scheduling** – Processes are divided into queues based on categories, each with its own scheduling.

Q.2.) Define Arrival Time of Process.

Ans :- **Arrival Time** is the time at which a process enters the **ready queue** for the first time and becomes available for CPU execution. It is measured from the start of system execution (i.e., time = 0).

Key Points:

- It indicates when a process is ready for scheduling.
- It plays a crucial role in CPU scheduling algorithms like **FCFS, SJF, and Round Robin**.

- A process with an earlier arrival time gets a chance to execute sooner in some scheduling methods.
- It affects **waiting time** and **turnaround time** of processes.

Q.3.) How to calculate Turn Around Time?

Ans :-

Turnaround Time (TAT) Calculation

Turnaround Time (TAT) is the total time taken by a process from its arrival in the system to its completion. It includes both the execution time and the waiting time.

Formula:

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

Where:

- **Completion Time (CT)** = The time when the process finishes execution.
- **Arrival Time (AT)** = The time when the process enters the ready queue.

Example Calculation:

Process	Arrival Time (AT)	Completion Time (CT)	Turnaround Time (TAT)
P1	2 ms	10 ms	$10 - 2 = 8$ ms
P2	5 ms	15 ms	$15 - 5 = 10$ ms

Q.4.) What is the representation of Waiting Time?

Ans :- **Representation of Waiting Time (WT)**

Waiting Time (WT) is the total time a process spends in the **ready queue** before getting CPU execution. It is the time a process waits **excluding its execution time**.

Formula:

$$\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time}$$

Where:

- **Turnaround Time (TAT)** = Total time from arrival to completion.
- **Burst Time (BT)** = Time required by the process for execution.

Example Calculation:

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	2 ms	4 ms	10 ms	$10 - 2 = 8$ ms	$8 - 4 = 4$ ms
P2	5 ms	5 ms	15 ms	$15 - 5 = 10$ ms	$10 - 5 = 5$ ms

Q.5.) What is Reader and Writer Problem?

Ans :- The **Reader-Writer Problem** is a classic synchronization problem in operating systems that deals with processes that read from and write to a shared resource (such as a database or file). It ensures **data consistency** when multiple processes access shared data.

Problem Statement:

- **Readers**: Can read the shared resource **simultaneously** without conflicts.
- **Writers**: Must have **exclusive access** to prevent data inconsistency.
- **Issue**: If a writer is modifying data, no reader or another writer should access it.

Types of Reader-Writer Problems:

1. **First Readers-Preference**:
    - Readers have priority over writers.
    - Writers may starve if readers keep arriving.
2. **Second Writers-Preference**:
    - Writers have priority over readers.

    - Readers may starve if writers keep arriving.
3. **Third (Fair) Solution**:
    - Ensures **no starvation** by handling requests in **order of arrival**.

Solution Using Synchronization:

- **Semaphores** or **mutex locks** are used to control access.
- Ensures **mutual exclusion** when writing and **concurrent reads**.

Q.6.) How to ignore deadlock?

Ans :- A **deadlock** occurs when a set of processes are stuck in a circular wait, each waiting for a resource held by another. To **avoid deadlocks**, we can use several techniques:

1. Deadlock Prevention (Avoiding One of the Four Conditions)

Deadlocks occur when the following four conditions hold simultaneously. Preventing at least one of them avoids deadlock:

- **Mutual Exclusion** – Use **spooling** or share resources where possible.
- **Hold and Wait** – Require processes to request all needed resources at once.
- **No Preemption** – Allow the system to forcibly take a resource from a process.
- **Circular Wait** – Impose an ordering on resource allocation (e.g., always request resources in a fixed order).

2. Deadlock Avoidance (Using Resource Allocation Strategies)

- **Banker's Algorithm** – A process must declare its maximum resource needs in advance, and the system grants resources **only if it keeps the system in a safe state**.
- **Resource Allocation Graph (RAG)** – Detects circular waits before granting resources.

3. Deadlock Detection & Recovery

- Use algorithms to periodically check for deadlocks.
- **Recovery Methods**:
    - **Process Termination** – Kill one or more processes to break the cycle.
    - **Resource Preemption** – Take resources from one process and give them to another.

4. Ignoring Deadlock (Ostrich Algorithm)

- Some systems (like UNIX & Windows) **ignore** deadlocks, assuming they are rare.
- This is practical when **the cost of handling deadlocks is higher than the cost of restarting the system**.

Q.7.) What is safe and unsafe state?

Ans :- In **deadlock avoidance**, a system is in either a **safe state** or an **unsafe state** based on whether it can allocate resources without leading to a deadlock.

1. Safe State

A system is in a **safe state** if **there exists a sequence of process execution (safe sequence) that allows all processes to complete without causing a deadlock**.

- Resources are allocated in a way that ensures at least one safe execution sequence.
- Example: If a process requests resources, the system checks if granting them keeps the system in a safe state. If yes, resources are allocated.

✅ **No deadlock is possible in a safe state.**

2. Unsafe State

A system is in an **unsafe state** if **no safe sequence exists, meaning that deadlock is possible but not guaranteed**.

- The system may not be able to allocate resources in a way that ensures all processes complete.
- Being in an unsafe state **does not** mean deadlock has occurred, but it increases the risk.

❌ **Deadlock may happen in an unsafe state.**

Example:

Consider a system with:

- **Total Resources** = 10
- **Allocated Resources** = 7
- **Maximum Demand** of processes = 8
- If the system can still allocate resources without leading to a deadlock, it is **safe**.
- If allocating resources results in a situation where no process can complete, the system is **unsafe**.

Q.8.) Define the Resource Allocation Graph.

Ans :- A **Resource Allocation Graph (RAG)** is a graphical representation used in **deadlock detection and avoidance**. It shows the allocation of resources to processes in a system.

Components of RAG:

1. **Processes (P1, P2, ... Pn)** → Represented by **circles** (●).
2. **Resources (R1, R2, ... Rm)** → Represented by **squares** (■).
3. **Assignment Edge (→)** → A directed edge from a **resource to a process (R → P)** indicates that the resource is allocated to the process.
4. **Request Edge (→)** → A directed edge from a **process to a resource (P → R)** indicates that the process is requesting the resource.

Deadlock Detection Using RAG:

- **If the graph has a cycle**, then a **deadlock may exist**.
- **If no cycle exists**, the system is in a safe state.

- If each resource has **only one instance**, a cycle in the graph **confirms** deadlock.
- If a resource has **multiple instances**, a cycle **may or may not** indicate deadlock.

Example:

1. **Safe State (No Cycle)**

P1 → R1 → P2 → R2 (No cycle, so no deadlock)

2. **Deadlock (Cycle Present)**

P1 → R1 → P2 → R2 → P1 (Cycle detected, possible deadlock)

Q.9.) How to recover Deadlock?

Ans :- Deadlock recovery involves breaking the cycle of waiting processes and returning the system to normal operation. There are two main methods to recover from a deadlock: **Process Termination** and **Resource Preemption**.

1. Process Termination:

- **Terminate All Deadlocked Processes:** Kill all processes involved in the deadlock. This is a drastic method but ensures recovery. However, it may result in data loss.
- **Terminate One Process at a Time:**
    - Processes can be terminated selectively to break the deadlock cycle.
    - The system can choose which process to terminate based on criteria like **priority**, **cost**, **age**, or the **number of resources held**.
    - Terminating processes that can be rolled back is preferred to minimize data loss.

2. Resource Preemption:

- **Preempt Resources from Processes:** Temporarily take resources away from processes involved in the deadlock and assign them to other processes that can finish and release resources.
- **Rollback:** If a process is capable of rolling back to a safe state, it can be moved to a point where it does not hold any resources, breaking the deadlock.

3. Combination of Both:

- A mix of process termination and resource preemption can be used in some cases for more efficient deadlock recovery.

Q.10.) What is Gantt Chart?

Ans :- A **Gantt Chart** is a visual representation used to plan and track the progress of tasks or processes over time. It is a bar chart that displays the start and finish dates of elements within a project or a set of tasks, showing their duration and sequencing.

Key Features:

1. **Horizontal Bars:** Represent tasks or processes. The length and position of each bar show the **duration** and **timing** of the task.
2. **Time Axis:** The **x-axis** represents time, typically in units like **days, weeks, or months**.
3. **Task List:** The **y-axis** lists the tasks or processes, typically arranged in order.
4. **Dependencies:** Shows how tasks are related and may depend on the completion of others.

Uses:

- **Project Management:** Gantt charts are extensively used to schedule and track the progress of projects.
- **CPU Scheduling:** In operating systems, Gantt charts visualize the order and duration of processes scheduled by algorithms like **FCFS** or **Round Robin**.

Example:

For a set of processes:

Process	Burst Time
P1	4 ms
P2	3 ms
P3	2 ms

Gantt Chart Example:

lua Copy Edit

```
|----/----/----/----/
P1   P1   P1   P1
       P2   P2   P2
            P3   P3
```

❖ 10 Marks Questions

Q.1.) Consider the following table of arrival time and burst time for three processes P1, P2 and P3.

Process	Burst Time	Arrival Time
P1	6 ms	0 ms
P2	8 ms	1 ms
P3	5 ms	2 ms

Calculate the Average Turn Around Time & Average Waiting Time using SRTF Scheduling Algorithm.

Ans :-

The **Shortest Remaining Time First (SRTF)** scheduling algorithm is a preemptive version of **Shortest Job Next (SJN)**. It always selects the process with the shortest remaining burst time for execution.

Given Data:

Process	Burst Time	Arrival Time
P1	6 ms	0 ms
P2	8 ms	1 ms
P3	5 ms	2 ms

Step 1: Construct the Gantt Chart

We track execution based on the shortest remaining time at each step:

1. **At time 0**: Only **P1** is available → **P1 starts execution**.
2. **At time 1**: **P2 arrives** (P1 still has 5 ms left, P2 has 8 ms). **P1 continues**.
3. **At time 2**: **P3 arrives** (P1 = 4 ms left, P2 = 8 ms, P3 = 5 ms). **P3 has the shortest burst time, so switch to P3**.
4. **At time 3 - 6**: **P3 runs until completion**.
5. **At time 7**: **P1 resumes execution (since it has the shortest remaining time)**.
6. **At time 10**: **P1 finishes**.
7. **At time 10 - 18**: **P2 is the only process left, so it runs till completion**.

Gantt Chart

```
0    2    7    10    18
| P1 | P3 | P1 | P2   |
```

Step 2: Calculate Completion, Turnaround & Waiting Times

Completion Time (CT)

- P1 finishes at **10 ms**.
- P2 finishes at **18 ms**.
- P3 finishes at **7 ms**.

Turnaround Time (TAT)

$$TAT = CT - Arrival\ Time$$

- **P1** = 10 - 0 = **10 ms**.
- **P2** = 18 - 1 = **17 ms**.
- **P3** = 7 - 2 = **5 ms**.

Waiting Time (WT)

$$WT = TAT - Burst\ Time$$

- **P1** = 10 - 6 = **4 ms**.
- **P2** = 17 - 8 = **9 ms**.
- **P3** = 5 - 5 = **0 ms**.

Step 3: Calculate Averages

$$\text{Average Turnaround Time} = \frac{10 + 17 + 5}{3} = \frac{32}{3} = 10.67 \text{ ms}$$

$$\text{Average Waiting Time} = \frac{4 + 9 + 0}{3} = \frac{13}{3} = 4.33 \text{ ms}$$

Final Answer:

- **Average Turnaround Time = 10.67 ms.**
- **Average Waiting Time = 4.33 ms.**

Q.2.) Explain the example of Semaphore.

Ans :- A **semaphore** is a synchronization mechanism used in concurrent programming to control access to shared resources by multiple processes or threads. It is commonly used in operating systems, multithreading, and distributed computing.

Example of Semaphore

Let's consider a real-life example:

Traffic Light System (Analogy)

Imagine a single-lane bridge where only one vehicle can pass at a time. If multiple vehicles try to cross at the same time, a collision might happen. To manage this, we can use a traffic light (semaphore) that allows only a certain number of vehicles at a time.

- **Green Light (Permit Available):** Vehicles can pass.
- **Red Light (Permit Not Available):** Vehicles must wait until the bridge is free.

This ensures that only a limited number of vehicles (say one or two) cross at a time, preventing congestion.

Programming Example: Semaphore in C (Using POSIX)

Let's implement a simple program where multiple threads try to access a shared resource, but only two can access it at a time using a semaphore.

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t semaphore;

void* task(void* arg) {
    int thread_num = *(int*)arg;

    printf("Thread %d waiting...\n", thread_num);
    sem_wait(&semaphore);  // Decrease the semaphore count (entering critical section)

    printf("Thread %d is in the critical section\n", thread_num);
    sleep(2); // Simulating work

    printf("Thread %d leaving critical section\n", thread_num);
    sem_post(&semaphore);  // Increase the semaphore count (exiting critical section)


    return NULL;
}

int main() {
    pthread_t threads[5];
    int thread_num[5];

    // Initialize semaphore with a count of 2 (allowing two threads at a time)
    sem_init(&semaphore, 0, 2);

    for (int i = 0; i < 5; i++) {
        thread_num[i] = i + 1;
        pthread_create(&threads[i], NULL, task, &thread_num[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore); // Clean up
    return 0;
}
```

Explanation

1. We initialize a semaphore with a value of 2, meaning two threads can access the critical section at a time.
2. Each thread tries to enter the critical section:
    - If the semaphore count is greater than 0, it decrements the count and enters.
    - If the semaphore count is 0, it waits until another thread releases it.

3. The thread performs some work (sleep(2)) and then releases the semaphore (sem_post), allowing another thread to enter.

Output (Example)

```
Thread 1 waiting...
Thread 1 is in the critical section
Thread 2 waiting...
Thread 2 is in the critical section
Thread 3 waiting...
Thread 1 leaving critical section
Thread 3 is in the critical section
Thread 2 leaving critical section
Thread 4 waiting...
Thread 4 is in the critical section
Thread 3 leaving critical section
Thread 5 waiting...
Thread 5 is in the critical section
Thread 4 leaving critical section
Thread 5 leaving critical section
```

Q.3.) Explain the solution of Bounded Buffer Problem?

Ans :- **Bounded Buffer Problem (Producer-Consumer Problem)**

The **Bounded Buffer Problem** is a classic synchronization problem that deals with a **fixed-size buffer** shared between a **producer** (which produces data) and a **consumer** (which consumes data). The challenge is to ensure that:

1. The producer does not add data if the buffer is full.
2. The consumer does not remove data if the buffer is empty.
3. Synchronization is maintained to avoid race conditions.

Solution Using Semaphores

To solve the **Bounded Buffer Problem**, we use **three semaphores**:

1. **empty** – Counts the number of empty slots in the buffer.
2. **full** – Counts the number of filled slots in the buffer.
3. **mutex** – Ensures mutual exclusion while accessing the buffer.

Algorithm

1. **Producer:**
    - Waits for an empty slot (empty semaphore).
    - Acquires the mutex to safely insert an item into the buffer.
    - Releases the mutex and signals the full semaphore.
2. **Consumer:**
    - Waits for a filled slot (full semaphore).
    - Acquires the mutex to safely remove an item from the buffer.
    - Releases the mutex and signals the empty semaphore.

Implementation in C (Using POSIX Threads and Semaphores)

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5  // Size of the bounded buffer

int buffer[BUFFER_SIZE];
int in = 0, out = 0;  // Indices for producer and consumer

sem_t empty, full, mutex;  // Semaphores for synchronization

void* producer(void* arg) {
    int item = 1;
    while (1) {
        sleep(1);  // Simulate time taken to produce item
        sem_wait(&empty);  // Wait for empty slot
        sem_wait(&mutex);  // Acquire mutex lock
```

```
        // Produce item
        buffer[in] = item;
        printf("Producer produced: %d at %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;
        item++;  // Increment item value

        sem_post(&mutex);  // Release mutex lock
        sem_post(&full);   // Signal that buffer has data
    }
}

void* consumer(void* arg) {
    while (1) {
        sleep(2);  // Simulate time taken to consume item
        sem_wait(&full);   // Wait for filled slot
        sem_wait(&mutex);  // Acquire mutex lock

        // Consume item
        int item = buffer[out];
        printf("Consumer consumed: %d from %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);  // Release mutex lock
        sem_post(&empty);  // Signal that buffer has space
    }
}
```

```c
int main() {
    pthread_t prod, cons;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);  // Initially, all slots are empty
    sem_init(&full, 0, 0);             // Initially, no filled slots
    sem_init(&mutex, 0, 1);            // Mutex initialized to 1

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for threads to finish (they run indefinitely)
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Destroy semaphores (never reached in this infinite loop example)
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

Explanation

1. **Producer Thread:**
    - Waits if the buffer is full (sem_wait(&empty)).
    - Acquires the lock (sem_wait(&mutex)) to safely insert an item.
    - Releases the lock and signals that the buffer has data (sem_post(&full)).
2. **Consumer Thread:**
    - Waits if the buffer is empty (sem_wait(&full)).
    - Acquires the lock (sem_wait(&mutex)) to safely remove an item.
    - Releases the lock and signals that the buffer has space (sem_post(&empty)).
3. **Synchronization Ensured Using Semaphores:**
    - **empty** keeps track of available slots.
    - **full** keeps track of filled slots.

- **mutex** ensures only one thread accesses the buffer at a time.

Example Output (May Vary)

```
Producer produced: 1 at 0
Consumer consumed: 1 from 0
Producer produced: 2 at 1
Producer produced: 3 at 2
Consumer consumed: 2 from 1
Producer produced: 4 at 3
Consumer consumed: 3 from 2
...
```

Q.4.) Explain the Dining-Philosophers Problem with the solution.

Ans :- The **Dining Philosophers Problem** is a classic synchronization problem in computer science. It demonstrates the challenges of resource sharing and avoiding **deadlock, starvation, and race conditions** in concurrent programming.

Problem Statement

Imagine **five philosophers** sitting around a circular dining table. Each philosopher has a **plate** of food and a **chopstick** on their left and right sides. Since a philosopher needs **two chopsticks** to eat, they must pick up both the left and right chopsticks.

The main challenges are:

- **Deadlock:** If every philosopher picks up their left chopstick at the same time, no one can pick up their right chopstick, and they all wait indefinitely.
- **Starvation:** Some philosophers might never get a chance to eat if others always acquire the chopsticks first.

Solution Using Semaphores

To solve this problem, we use **sem**aphores:

- **One semaphore per chopstick:** Controls access so that only one philosopher can hold a chopstick at a time.
- **A mutex semaphore:** Ensures mutual exclusion.

Implementation in C (Using POSIX Threads and Semaphores)

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5  // Number of philosophers

sem_t chopstick[N];  // Semaphores for chopsticks

void* philosopher(void* arg) {
    int id = *(int*)arg;

    while (1) {
        // Thinking
        printf("Philosopher %d is thinking...\n", id);
        sleep(1); // Simulate thinking

        // Picking up chopsticks
        sem_wait(&chopstick[id]);                // Pick left chopstick
        sem_wait(&chopstick[(id + 1) % N]);      // Pick right chopstick

        // Eating
        printf("Philosopher %d is eating...\n", id);
        sleep(2); // Simulate eating

        // Putting down chopsticks
        sem_post(&chopstick[id]);                // Release left chopstick
        sem_post(&chopstick[(id + 1) % N]);      // Release right chopstick

        // Back to thinking
    }
}
```

```c
int main() {
    pthread_t philosophers[N];
    int ids[N];

    // Initialize semaphores (one per chopstick)
    for (int i = 0; i < N; i++) {
        sem_init(&chopstick[i], 0, 1);
    }

    // Create philosopher threads
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }

    // Join threads (infinite loop)
    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Destroy semaphores (not reached in this example)
    for (int i = 0; i < N; i++) {
        sem_destroy(&chopstick[i]);
    }

    return 0;
}
```

Explanation

1. **Philosopher Thinking** 🧠
    - **A philosopher spends some time thinking before attempting to eat.**
2. **Picking Up Chopsticks** 🥢
    - **The philosopher waits (sem_wait) until both the left and right chopsticks are available.**
3. **Eating** 🍽️
    - **The philosopher eats for some time.**
4. **Putting Down Chopsticks** 🔄
    - **The philosopher releases (sem_post) both chopsticks.**
5. **Preventing Deadlock**

- **The implementation ensures that a philosopher can eat only if both chopsticks are available.**

Potential Issues & Solutions

1. Deadlock

Problem:
If all philosophers pick their left chopstick at the same time, none can pick their right chopstick, leading to deadlock.

Solution:

- **One Less Philosopher Rule: Allow only N-1 philosophers to eat at the same time.**
- **Pick Odd-Even Rule:**
  - **Odd-indexed philosophers pick left first.**
  - **Even-indexed philosophers pick right first.**

2. Starvation

Problem:
Some philosophers may never get a chance to eat if others keep acquiring chopsticks first.

Solution:

- **Introduce a fair scheduling mechanism such as assigning a time limit for each philosopher to hold chopsticks.**

Example Output

```
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 2 is eating...
Philosopher 4 is eating...
Philosopher 2 finished eating.
Philosopher 4 finished eating.
...
```

Q.5.) What are the necessary conditions for Deadlock?

Ans :- **Necessary Conditions for Deadlock**

A **deadlock** is a situation in which a set of processes become stuck, waiting for resources that are held by other processes, leading to an indefinite wait.

For a **deadlock to occur**, the following **four conditions (Coffman Conditions)** must hold simultaneously:

1. **Mutual Exclusion** 🛑
    - At least one resource must be held in a **non-shareable** mode.
    - If another process requests that resource, it must **wait** until it is released.
2. **Hold and Wait** ⏳
    - A process **holds** at least one resource and **waits** for additional resources that are currently held by other processes.
3. **No Preemption** 🚫
    - Resources **cannot be forcibly taken** from a process; they can only be **released voluntarily** by the process holding them.

4. **Circular Wait** 🔄
    - A set of two or more processes must exist where **each process is waiting for a resource held by the next process** in the chain, forming a cycle.

Example of Deadlock

Consider two processes (P1 and P2) and two resources (R1 and R2):

1. P1 holds R1 and requests R2.
2. P2 holds R2 and requests R1.
3. Since neither process can release the resource it holds before getting the other resource, they remain **stuck forever**.

```
P1 → holding R1, waiting for R2
P2 → holding R2, waiting for R1
```

This situation satisfies **all four conditions**, leading to **deadlock**.

How to Prevent Deadlock?

1. **Breaking Mutual Exclusion**
    - Use **spooling** (e.g., printers) or **lock-free data structures** to allow shared access.
2. **Avoid Hold and Wait**
    - Require processes to **request all resources upfront** before execution begins.
3. **Allow Preemption**
    - If a process is waiting, forcefully **take resources** away and allocate them to others.
4. **Avoid Circular Wait**
    - Impose an **ordering on resource allocation** so that processes request resources in a predefined order.

Q.6.) What are the Differences between Starvation and Deadlock?

Ans :- **Differences Between Starvation and Deadlock**

Feature	Deadlock 🔄	Starvation ⏳
Definition	A situation where multiple processes **wait indefinitely** because each is holding a resource that the other needs.	A situation where a **low-priority process waits indefinitely** because higher-priority processes keep executing.
Cause	Caused by the four necessary conditions (**Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait**).	Caused by unfair resource allocation, such as **priority-based scheduling**.
Resource Holding	Processes **hold** resources while waiting for others, leading to a cycle of dependency.	A process does **not hold** resources but cannot acquire them due to continuous preemption.
Resolution	Requires **deadlock prevention, avoidance, or detection** mechanisms.	Can be resolved using **aging** (gradually increasing priority of waiting processes).
Example	Two processes `P1` and `P2` hold resources `R1` and `R2`, but each needs the other's resource to proceed.	A low-priority process waits forever because a high-priority process keeps executing.

Example of Deadlock

1. **P1 holds Resource R1 and waits for R2.**
2. **P2 holds Resource R2 and waits for R1.**
3. **Neither can proceed, leading to deadlock.**

Example of Starvation

1. **A CPU scheduling algorithm gives higher priority to some processes.**
2. **A low-priority process waits indefinitely because higher-priority processes keep arriving.**