**Operating Systems**

Unit-3

1. *Describe in brief the different IPC mechanisms*

Ans.

**Inter-Process Communication (IPC):**

Inter-Process Communication (IPC) is a mechanism that allows processes to communicate and share data. It is essential for coordination in multitasking systems. Below are the major IPC mechanisms, explained in detail:

**1. Pipes:**

* Pipes provide a unidirectional communication channel between two processes.
* One process writes to the pipe (producer), and another reads from it (consumer).
* They are commonly used in Unix/Linux systems to chain commands.
* **Example:** ls | grep connects the output of ls to the input of grep.

**Advantages:** Simple and efficient for parent-child process communication.  
**Limitations:** Works only for processes with a shared ancestry.

**2. Message Queues:**

* A message queue allows processes to send and retrieve messages asynchronously.
* It ensures reliable communication as messages persist until they are read.
* Suitable for systems where processes need to exchange structured data.

**Advantages:** Messages are queued, ensuring no data is lost if the recipient is busy.  
**Limitations:** Requires additional overhead to manage queues.

**3. Shared Memory:**

* Shared memory is the fastest IPC method, where multiple processes access a common memory region.
* It avoids the need to copy data between processes.
* Synchronization is critical to prevent race conditions (handled using semaphores or mutex locks).

**Advantages:** High-speed data sharing.  
**Limitations:** Complex to implement due to synchronization requirements.

**4. Semaphores:**

* A semaphore is a signaling mechanism to control access to shared resources.
* It helps manage resource contention and prevents race conditions.
* **Example:** In a multi-threaded application, semaphores can ensure only one thread writes to a shared file at a time.

**Advantages:** Provides synchronization for shared resources.  
**Limitations:** Risk of deadlocks if not implemented carefully.

**5. Sockets:**

* Sockets allow communication between processes over a network or on the same system.
* They are commonly used in client-server architecture.
* **Example:** Web browsers (clients) communicate with web servers using sockets.

**Advantages:** Supports bidirectional communication over long distances.  
**Limitations:** Requires more overhead for setup and data transfer.

**6. Signals:**

* Signals are software interrupts sent to a process to notify it of events (e.g., termination, alarms).
* They are simple and lightweight but provide limited data.
* **Example:** SIGKILL is a signal used to terminate a process.

**Advantages:** Lightweight and effective for notification.  
**Limitations:** Limited data exchange and harder to handle complex events.

**Conclusion:**  
Each IPC mechanism is suited for specific scenarios:

* **Pipes** and **message queues** for simple communication.
* **Shared memory** and **semaphores** for speed and synchronization.
* **Sockets** for network communication.
* **Signals** for event-based notifications.

Proper selection of IPC depends on the system's requirements, like speed, complexity, and reliability.

This expanded explanation includes definitions, use cases, examples, and advantages/limitations to justify a 9-mark answer.

1. *What is deadlock? Explain the necessary and sufficient conditions for the occurrence of a deadlock.*

Ans. **What is Deadlock?**

A **deadlock** occurs in a system when a set of processes becomes stuck because each process is waiting for a resource that another process in the set is holding. In this situation, no process can proceed, and the system remains in a blocked state.

**Example:**

Imagine two people trying to cross a narrow bridge from opposite sides. Each one waits for the other to move back, but neither does, causing a stalemate.

**Necessary and Sufficient Conditions for Deadlock**

For a deadlock to occur, the following **four conditions must hold simultaneously**. These are also known as **Coffman’s Conditions:**

1. **Mutual Exclusion:**
   * At least one resource must be held in a non-shareable mode.
   * Only one process can use the resource at a time.
   * Example: A printer that cannot be used by more than one process simultaneously.
2. **Hold and Wait:**
   * A process is holding at least one resource and is waiting to acquire additional resources held by other processes.
   * Example: Process A holds a printer and waits for a scanner, while Process B holds the scanner and waits for the printer.
3. **No Preemption:**
   * Resources cannot be forcibly taken away from a process. They must be released voluntarily by the process holding them.
   * Example: A process holding a printer will not release it until its task is completed.
4. **Circular Wait:**
   * A set of processes forms a circular chain, where each process is waiting for a resource held by the next process in the chain.
   * Example: Process A waits for a resource held by Process B, Process B waits for a resource held by Process C, and Process C waits for a resource held by Process A.

**Explanation of Conditions**

* **Mutual Exclusion:** Ensures exclusive access to a resource but creates dependency.
* **Hold and Wait:** Leads to contention for resources.
* **No Preemption:** Prevents the system from resolving conflicts by reallocating resources.
* **Circular Wait:** Forms a loop of dependencies, making recovery impossible without breaking the cycle.

**3)** *Explain the following with example*

*i) Mutual Exclusion   
ii) Synchronization  
iii) Race condition*

Ans. i) Mutual Exclusion

Definition:

Mutual exclusion ensures that only one process or thread accesses a critical section (shared resource or code block) at any given time. This prevents conflicts and maintains data consistency when multiple processes attempt to use the same resource concurrently.

Detailed Example:

Imagine a bank system where two customers are trying to withdraw money from the same account simultaneously. If both processes access the account balance at the same time, the withdrawal operations might lead to an inconsistent balance.

Scenario Without Mutual Exclusion:

Customer A checks the balance: $1000.

Customer B also checks the balance: $1000.

Both withdraw $500 simultaneously.

Final balance: $500 instead of the correct $0, as the operations overlap.

Solution:

Mutual exclusion can be implemented using locks (e.g., mutex locks) to ensure that only one customer can access the account at a time. When one process is in the critical section (checking and updating balance), others must wait.

ii) Synchronization

Definition:

Synchronization coordinates the execution of processes or threads to ensure correct program behavior. It is used to manage dependencies between processes and to enforce the correct order of operations when shared resources are involved.

Detailed Example:

Consider a producer-consumer problem, where one process (producer) creates items and places them in a buffer, and another process (consumer) removes and processes them.

Scenario Without Synchronization:

The producer adds items to a buffer, but if the consumer accesses the buffer while it’s empty, it might result in an error or undefined behavior.

Conversely, if the producer adds items when the buffer is already full, the system might overwrite data or crash.

Solution:

Synchronization ensures the consumer only removes items if the buffer is not empty and the producer only adds items if there’s space in the buffer.

Tools like semaphores and condition variables are commonly used. A semaphore can signal the state of the buffer (e.g., "full" or "empty") to ensure proper coordination.

iii) Race Condition

Definition:

A race condition occurs when two or more processes or threads access and modify shared data concurrently, and the final result depends on the unpredictable timing of their execution. Without proper control mechanisms, this can lead to inconsistent or unexpected outcomes.

Detailed Example:

Suppose two threads are trying to increment a shared counter variable:

// Shared counter

int counter = 0;

// Thread 1

counter++;

// Thread 2

counter++;

Scenario Without Proper Synchronization:

Thread 1 reads counter = 0.

Thread 2 also reads counter = 0 (before Thread 1 writes back).

Both increment the counter to 1 independently.

The final value of counter is 1, even though it should be 2.

Why This Happens:

The counter++ operation is not atomic. It involves three steps:

Read the current value of counter.

Increment the value.

Write the incremented value back to memory.

If two threads perform these steps simultaneously without synchronization, they may read the same value, leading to incorrect results.

Solution:

Using a lock or atomic operation ensures that only one thread can increment the counter at a time, preventing the race condition.

Key Differences Between These Concepts:

Mutual Exclusion prevents multiple processes from accessing a critical section simultaneously.

Synchronization ensures correct sequencing and coordination between processes or threads.

Race Conditions arise due to lack of proper synchronization, leading to inconsistent outcomes.

Together, mutual exclusion and synchronization are the tools to prevent race conditions and ensure consistent, predictable program behavior.

1. *What is Critical Section Problem? Explain readers-writers problem.*

Ans. **What is the Critical Section Problem?**

The **Critical Section Problem** is a fundamental issue in concurrent programming. It arises when multiple processes or threads need to access and modify shared resources (like memory or files) simultaneously. To prevent data corruption or inconsistencies, only one process should execute its critical section at a time.

The **critical section** is the portion of code where shared resources are accessed or modified. The main goal of solving this problem is to ensure **mutual exclusion**, **progress**, and **bounded waiting**.

**Requirements for a Solution:**

1. **Mutual Exclusion:**
   * Only one process can execute its critical section at a time.
2. **Progress:**
   * If no process is in the critical section, the selection of the next process should not be indefinitely delayed.
3. **Bounded Waiting:**
   * A process should not have to wait indefinitely to enter its critical section.

**Readers-Writers Problem**

The **Readers-Writers Problem** is a classic synchronization problem that deals with processes accessing a shared resource, typically a database. It ensures consistency by coordinating how **readers** and **writers** access the resource.

**Definitions:**

* **Reader:** A process that only reads the shared resource and does not modify it. Multiple readers can access the resource simultaneously.
* **Writer:** A process that modifies the shared resource. Only one writer can access the resource at a time, and no readers are allowed during this period.

**Problem Statement:**

Design a solution to manage reader and writer access such that:

1. Multiple readers can read simultaneously if no writer is writing.
2. Writers have exclusive access when writing.
3. Avoid starvation of readers or writers.

**Solutions to the Readers-Writers Problem**

**1. First Readers-Writers Problem (No Reader Starvation):**

* **Constraint:** If a writer is waiting, no new reader can access the resource.
* **Goal:** Writers get priority over readers.

**Algorithm:**

1. Use a shared variable read\_count to track the number of active readers.
2. When the first reader enters, it locks the resource for readers.
3. When the last reader exits, it unlocks the resource.
4. Writers must acquire an exclusive lock before accessing the resource.

**Drawback:** Readers may starve if writers continuously arrive.

**2. Second Readers-Writers Problem (No Writer Starvation):**

* **Constraint:** Readers should not be kept waiting indefinitely if the resource is available.
* **Goal:** Give priority to readers.

**Algorithm:**

1. Readers can access the resource immediately if no writer is currently writing.
2. Writers must wait until all readers have finished.
3. Writers can only proceed when there are no active or incoming readers.

**Drawback:** Writers may starve if there is a continuous stream of readers.

**Example Pseudocode for the Readers-Writers Problem**

**Shared Variables:**

int read\_count = 0; // Number of active readers

Semaphore mutex = 1; // Mutex to protect read\_count

Semaphore rw\_mutex = 1; // Ensures mutual exclusion for writers

**Reader Code:**

wait(mutex); // Lock to update read\_count

read\_count++;

if (read\_count == 1) // First reader locks the resource

wait(rw\_mutex);

signal(mutex); // Unlock to allow other readers

// Reading the resource

// ...

wait(mutex); // Lock to update read\_count

read\_count--;

if (read\_count == 0) // Last reader unlocks the resource

signal(rw\_mutex);

signal(mutex); // Unlock

**Writer Code:**

wait(rw\_mutex); // Lock the resource exclusively

// Writing to the resource

// ...

signal(rw\_mutex); // Unlock the resource

**Conclusion:**

The **Critical Section Problem** focuses on ensuring safe access to shared resources. The **Readers-Writers Problem** is a specific example of this, where careful synchronization ensures that readers and writers do not interfere with each other. Solutions to the Readers-Writers Problem use semaphores or locks to balance the needs of readers and writers while avoiding starvation and ensuring system fairness.

5) *What is semaphore and mutex? Explain with the help of pseudocode, how semaphore is used to solve producer consumer problem?*

Ans. **Semaphore and Mutex**

**Mutex:**

* A **mutex (mutual exclusion)** is a synchronization primitive that provides **exclusive access** to a shared resource. It ensures that only one thread can access the critical section at a time.
* **Key Properties**:
  + Binary in nature: locked (1) or unlocked (0).
  + Ownership: The thread that locks it must unlock it.

**Semaphore:**

* A **semaphore** is a synchronization primitive used to control access to a shared resource by multiple threads.
* **Key Properties**:
  + Can have a value greater than 1, unlike a mutex.
  + Two types:
    1. **Binary Semaphore**: Acts like a mutex but without ownership.
    2. **Counting Semaphore**: Allows more than one thread to access the resource (e.g., bounded buffer size).

**Producer-Consumer Problem Using Semaphore**

The **Producer-Consumer problem** involves synchronization between producers (which add items to a buffer) and consumers (which remove items from the buffer).

The challenge is to ensure:

1. Producers don't add to a full buffer.
2. Consumers don't remove from an empty buffer.

**Semaphores Used:**

1. **Empty**: Tracks the number of empty slots in the buffer.
2. **Full**: Tracks the number of filled slots in the buffer.
3. **Mutex**: Ensures mutual exclusion when accessing the buffer.

**Pseudocode for Producer-Consumer Problem**

# Initialize semaphores and shared resources

semaphore empty = BUFFER\_SIZE # Count of empty slots

semaphore full = 0 # Count of full slots

mutex = 1 # Mutual exclusion for critical section

# Buffer is a shared resource

BUFFER = []

# Producer Process

function producer():

while True:

item = produce\_item() # Produce an item

wait(empty) # Decrement empty slots

wait(mutex) # Enter critical section

BUFFER.append(item) # Add item to the buffer

print("Produced:", item)

signal(mutex) # Leave critical section

signal(full) # Increment full slots

# Consumer Process

function consumer():

while True:

wait(full) # Decrement full slots

wait(mutex) # Enter critical section

item = BUFFER.pop(0) # Remove item from the buffer

print("Consumed:", item)

signal(mutex) # Leave critical section

signal(empty) # Increment empty slots

consume\_item(item) # Process the consumed item

# Helper functions for semaphore operations

function wait(semaphore):

while semaphore <= 0:

pass # Busy-wait

semaphore -= 1

function signal(semaphore):

semaphore += 1

# Produce and consume logic (for demonstration purposes)

function produce\_item():

return random.randint(1, 100)

function consume\_item(item):

# Placeholder for consuming logic

pass

**Explanation of the Workflow:**

1. **Producer**:
   * Produces an item and waits if the buffer is full (wait(empty)).
   * Acquires the mutex (wait(mutex)), adds the item, then releases the mutex (signal(mutex)).
   * Signals that a slot in the buffer is now full (signal(full)).
2. **Consumer**:
   * Waits if the buffer is empty (wait(full)).
   * Acquires the mutex (wait(mutex)), removes an item, then releases the mutex (signal(mutex)).
   * Signals that a slot in the buffer is now empty (signal(empty)).

This ensures synchronization and prevents race conditions between producers and consumers.

6) *What are the four necessary conditions for deadlock? How is a deadlock detected in a system with resources having single instances? Explain with an example.*

Ans. **Four Necessary Conditions for Deadlock**

A deadlock occurs in a system when a group of processes becomes stuck, waiting indefinitely for resources. The **necessary conditions** for deadlock are:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode, meaning only one process can use the resource at a time.
2. **Hold and Wait**: A process is holding at least one resource and is waiting to acquire additional resources currently held by other processes.
3. **No Preemption**: Resources cannot be forcibly taken from a process; they can only be released voluntarily by the holding process.
4. **Circular Wait**: A set of processes exists such that each process is waiting for a resource held by the next process in the chain, forming a circular chain.

**Deadlock Detection in a System with Single-Instance Resources**

For a system with single-instance resources, deadlock detection is performed using a **resource allocation graph**. If the graph contains a **cycle**, then a deadlock exists.

**Resource Allocation Graph**

* **Nodes**: Represent processes (P1, P2, ...) and resources (R1, R2, ...).
* **Edges**:
  + Request Edge: A directed edge from a process PiP\_i to a resource RjR\_j, denoted as Pi→RjP\_i \rightarrow R\_j (process PiP\_i is requesting RjR\_j).
  + Allocation Edge: A directed edge from a resource RjR\_j to a process PiP\_i, denoted as Rj→PiR\_j \rightarrow P\_i (resource RjR\_j is allocated to PiP\_i).

**Example**

**Initial System State**

* **Processes**: P1, P2, P3
* **Resources**: R1, R2, R3 (all single instances)
* **Current Allocation**:
  + P1 holds R1.
  + P2 holds R2.
  + P3 holds R3.

**Resource Requests**

* P1 requests R2.
* P2 requests R3.
* P3 requests R1.

**Resource Allocation Graph:**

1. R1→P1R1 \rightarrow P1 (P1 holds R1).
2. P1→R2P1 \rightarrow R2 (P1 requests R2).
3. R2→P2R2 \rightarrow P2 (P2 holds R2).
4. P2→R3P2 \rightarrow R3 (P2 requests R3).
5. R3→P3R3 \rightarrow P3 (P3 holds R3).
6. P3→R1P3 \rightarrow R1 (P3 requests R1).

**Graph Analysis**

* There is a cycle in the graph: P1→R2→P2→R3→P3→R1→P1P1 \rightarrow R2 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R1 \rightarrow P1.

Since there is a cycle, the system is in a deadlock state.

**Deadlock Detection Algorithm for Single-Instance Resources**

1. **Construct the Resource Allocation Graph** based on current resource allocation and requests.
2. **Search for a Cycle**:
   * Use graph traversal techniques like Depth-First Search (DFS) to detect cycles.
3. If a cycle exists, the processes involved in the cycle are in a deadlock.

**Resolution**

* Deadlocks can be resolved by:
  1. Preempting resources (breaking the "no preemption" condition).
  2. Terminating one or more processes involved in the cycle to break the circular wait.

7) *What is Bankers safe sequence algorithm? Apply it for finding safe sequence of execution of 5 processes in a system having Snapshot at time T0: [12] Also determine whether following requests can be granted or not: i) Request for process P2: - 3 0 0 and ii) Request for process P3: - 0 0 1*

Ans. **Given Data**

**Snapshot at Time T0:**

| **Process** | **Allocation** | **Max** | **Need = Max - Allocation** |
| --- | --- | --- | --- |
|  | A B C | A B C | A B C |
| P0 | 0 1 0 | 7 5 3 | 7 4 3 |
| P1 | 2 0 0 | 3 2 2 | 1 2 2 |
| P2 | 3 0 2 | 9 0 2 | 6 0 0 |
| P3 | 2 1 1 | 2 2 2 | 0 1 1 |
| P4 | 0 0 2 | 4 3 3 | 4 3 1 |

**Available Matrix**: ABC=[3 3 2]

**Step 1: Calculate Need Matrix**

The Need Matrix is calculated as Need = Max - Allocation. It has already been provided above.

**Step 2: Apply Banker's Algorithm to Find Safe Sequence**

1. **Initialization**:
   * Work = [3 \ 3 \ 2]
   * Finish = [False \ False \ False \ False \ False]
   * Safe Sequence = []
2. **Iteration 1**:
   * Find a process P such that Need[P] ≤ Work:
     + P0: [7 4 3] > [3 3 2] (not satisfied)
     + P1: [1 2 2] ≤ [3 3 2] (satisfied)
   * Execute P1: Work = Work + Allocation[P1] = [3 3 2] + [2 0 0] = [5 3 2]
   * Mark Finish[P1] = True and add P1 to Safe Sequence: Safe Sequence = [P1]
3. **Iteration 2**:
   * P0: [7 4 3] > [5 3 2] (not satisfied)
   * P2: [6 0 0] > [5 3 2] (not satisfied)
   * P3: [0 1 1] ≤ [5 3 2] (satisfied)
   * Execute P3: Work = Work + Allocation[P3] = [5 3 2] + [2 1 1] = [7 4 3]
   * Mark Finish[P3] = True and add P3 to Safe Sequence: Safe Sequence = [P1, P3]
4. **Iteration 3**:
   * P0: [7 4 3] ≤ [7 4 3] (satisfied)
   * Execute P0: Work = Work + Allocation[P0] = [7 4 3] + [0 1 0] = [7 5 3]
   * Mark Finish[P0] = True and add P0 to Safe Sequence: Safe Sequence = [P1, P3, P0]
5. **Iteration 4**:
   * P2: [6 0 0] ≤ [7 5 3] (satisfied)
   * Execute P2: Work = Work + Allocation[P2] = [7 5 3] + [3 0 2] = [10 5 5]
   * Mark Finish[P2] = True and add P2 to Safe Sequence: Safe Sequence = [P1, P3, P0, P2]
6. **Iteration 5**:
   * P4: [4 3 1] ≤ [10 5 5] (satisfied)
   * Execute P4: Work = Work + Allocation[P4] = [10 5 5] + [0 0 2] = [10 5 7]
   * Mark Finish[P4] = True and add P4 to Safe Sequence: Safe Sequence = [P1, P3, P0, P2, P4]

**Safe Sequence**

The system is in a safe state, and the safe sequence is: **P1 → P3 → P0 → P2 → P4**

**Step 3: Check Resource Requests**

**(i) Request by P2: [3 0 0]**

* **Check 1**: Request ≤ Need:
  + Request[P2] = [3 0 0] and Need[P2] = [6 0 0] → Satisfied.
* **Check 2**: Request ≤ Available:
  + Request[P2] = [3 0 0] and Available = [3 3 2] → Satisfied.
* **Grant Request**:
  + Update matrices:
    - Available = Available - Request = [3 3 2] - [3 0 0] = [0 3 2]
    - Allocation[P2] = Allocation[P2] + Request = [3 0 2] + [3 0 0] = [6 0 2]
    - Need[P2] = Need[P2] - Request = [6 0 0] - [3 0 0] = [3 0 0]
  + Re-run Banker's Algorithm to confirm safety. The system remains **safe**.

**(ii) Request by P3: [0 0 1]**

* **Check 1**: Request ≤ Need:
  + Request[P3] = [0 0 1] and Need[P3] = [0 1 1] → Satisfied.
* **Check 2**: Request ≤ Available:
  + Request[P3] = [0 0 1] and Available = [3 3 2] → Satisfied.
* **Grant Request**:
  + Update matrices:
    - Available = Available - Request = [3 3 2] - [0 0 1] = [3 3 1]
    - Allocation[P3] = Allocation[P3] + Request = [2 1 1] + [0 0 1] = [2 1 2]
    - Need[P3] = Need[P3] - Request = [0 1 1] - [0 0 1] = [0 1 0]
  + Re-run Banker's Algorithm to confirm safety. The system remains **safe**.

**Conclusion**

* **Safe Sequence**: P1 → P3 → P0 → P2 → P4
* Both requests (P2: [3 0 0] and P3: [0 0 1]) can be safely granted.

8) *Differentiate between named pipe and unnamed pipe ?*

Ans. **Difference Between Named and Unnamed Pipes**

| **Feature** | **Unnamed Pipe** | **Named Pipe (FIFO)** |
| --- | --- | --- |
| **Definition** | A pipe with no explicit name, used for communication between related processes (parent-child). | A pipe with a name in the filesystem, used for communication between unrelated or related processes. |
| **Identification** | Identified by a file descriptor, not visible in the filesystem. | Identified by a name, visible in the filesystem. |
| **Scope** | Limited to parent-child processes. | Can be used by any processes with access to the pipe. |
| **Persistence** | Exists only during the lifetime of the process. | Exists in the filesystem until explicitly deleted. |
| **Creation** | Created using pipe() system call. | Created using mkfifo() system call or mknod(). |
| **Example Use Case** | A parent process sending data to its child process. | Communication between two unrelated programs, e.g., a producer and a consumer. |
| **Example Code** | Uses pipe() and fork(). | Uses mkfifo() to create a FIFO file. |

9) *What conditions are generally associated with readers-writers problem? Write its pseudo code*

Ans. **Readers-Writers Problem**

The **Readers-Writers Problem** is a classic synchronization problem in which multiple processes either read from or write to a shared resource. The goal is to ensure data consistency while maximizing concurrency.

**Conditions Associated with the Problem:**

1. **Mutual Exclusion**:
   * Writers must have exclusive access to the shared resource. Only one writer can write at a time.
   * Readers can access the resource concurrently if no writer is writing.
2. **No Starvation**:
   * Writers should not be starved by continuous reader access.
   * Readers should not be starved by continuous writer access.
3. **Synchronization**:
   * Readers and writers must coordinate to prevent conflicts.

**Types of Readers-Writers Problem**

1. **First Readers-Writers Problem**:
   * Readers have priority over writers.
   * A writer can only write when there are no readers.
2. **Second Readers-Writers Problem**:
   * Writers have priority over readers.
   * Readers must wait until all writers are done.

**Pseudocode for Readers-Writers Problem**

**Initialization**

semaphore mutex = 1 # Protects shared variables

semaphore rw\_mutex = 1 # Ensures mutual exclusion for writers

int read\_count = 0 # Tracks the number of active readers

**Reader Process**

function reader():

while True:

wait(mutex) # Reader acquires lock to update read\_count

read\_count += 1

if read\_count == 1: # First reader locks rw\_mutex

wait(rw\_mutex)

signal(mutex) # Release lock for other readers

# Critical Section: Reading is performed

read\_data()

wait(mutex) # Reader updates read\_count

read\_count -= 1

if read\_count == 0: # Last reader unlocks rw\_mutex

signal(rw\_mutex)

signal(mutex) # Release lock

**Writer Process**

function writer():

while True:

wait(rw\_mutex) # Writer locks rw\_mutex

# Critical Section: Writing is performed

write\_data()

signal(rw\_mutex) # Writer releases rw\_mutex

**Explanation**

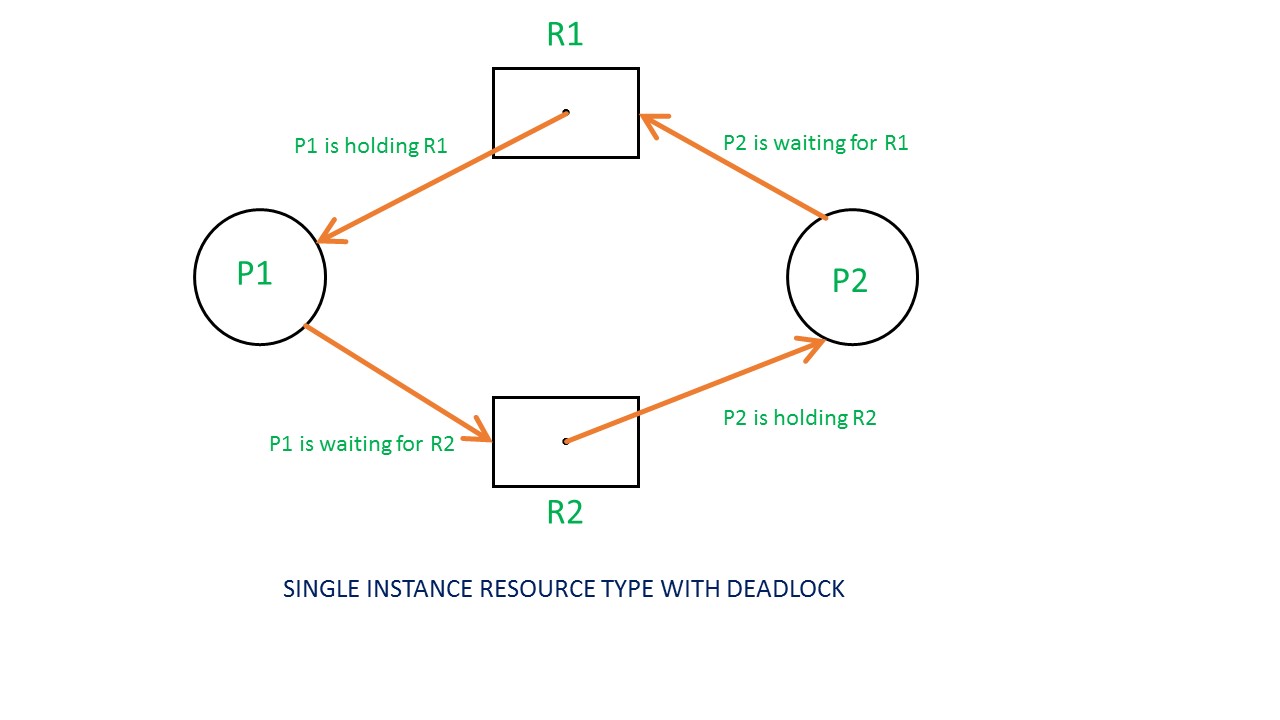
1. **Reader Process**:
   * The first reader locks the rw\_mutex to prevent writers from writing while readers are accessing the resource.
   * Subsequent readers increment the read\_count and proceed without locking rw\_mutex to allow concurrent reading.
   * When the last reader finishes, it unlocks the rw\_mutex to allow writers access.
2. **Writer Process**:
   * Writers lock the rw\_mutex to gain exclusive access to the resource.
   * Only one writer can write at a time, and no readers can access the resource during writing.

**Key Features**

* **Mutual Exclusion**:
  + Writers and readers use rw\_mutex to ensure data integrity.
* **Fairness**:
  + Priority can be adjusted by modifying the synchronization logic (e.g., using FIFO for fair scheduling).

This pseudocode ensures synchronization and avoids race conditions while allowing as much concurrency as possible between readers.

10) *Describe resource allocation graph in detail.*

Ans. 

**Resource Allocation Graph (RAG)**

A **Resource Allocation Graph (RAG)** is a directed graph used to model the allocation of resources to processes in a system. It is a visual representation that helps detect potential deadlocks in a system.

**Components of RAG**

1. **Nodes**:
   * **Processes (P)**: Represented as circles, each node corresponds to a process.
   * **Resources (R)**: Represented as rectangles, each node corresponds to a resource type.
2. **Edges**:
   * **Request Edge (P → R)**:
     + A directed edge from a process PiP\_i to a resource RjR\_j indicates that process PiP\_i is requesting an instance of resource RjR\_j.
   * **Allocation Edge (R → P)**:
     + A directed edge from a resource RjR\_j to a process PiP\_i indicates that an instance of RjR\_j is allocated to PiP\_i.

**Structure of RAG**

* A process node can have outgoing edges to resource nodes (requests).
* A resource node can have multiple outgoing edges to processes (allocation of instances).

**Deadlock Detection Using RAG**

1. **Single-Instance Resources**:
   * A **cycle** in the RAG indicates a deadlock.
   * For example, if P1→R1→P2→R2→P1P1 → R1 → P2 → R2 → P1, the system is in a deadlock because each process is waiting for a resource held by the other.
2. **Multiple-Instance Resources**:
   * A cycle in the RAG does **not necessarily** indicate a deadlock.
   * Additional checks are needed to determine if the processes in the cycle can proceed based on available resources.

**Example of RAG**

**Scenario**:

* Processes: P1,P2,P3
* Resources: R1,R2
* Resource Instances:
  + R1: 1 instance
  + R2: 2 instances

**Graph Representation**:

 P1 → R1: Process P1 requests R1.

 R1 → P2: Resource R1 is allocated to P2.

 P2 → R2: Process P2 requests R2.

 R2 → P3: Resource R2 is allocated to P3.

 P3 → R1: Process P3 requests R1.

**Graph Analysis**:

* + **** Processes: P1, P2, P3
  +  Resources: R1, R2
  +  Edges: P1 → R1, R1 → P2, P2 → R2, R2 → P3, P3 → R1
  +  Cycle: P1 → R1 → P2 → R2 → P3 → R1

Since there is a cycle, the system is in a deadlock.

**Applications of RAG**

1. **Deadlock Detection**:
   * In single-instance systems, detecting a cycle in the RAG is sufficient to identify deadlock.
   * In multiple-instance systems, additional checks are required to confirm deadlock.
2. **Visualization**:
   * Helps visualize resource allocation and potential conflicts between processes.
3. **Analysis**:
   * Assists in studying resource utilization and improving system efficiency.

**Advantages**

* Simplifies understanding of resource allocation in a system.
* Provides an intuitive way to detect potential deadlocks.
* Serves as a foundation for more advanced deadlock detection and avoidance techniques.

**Limitations**

* Limited practicality in large, dynamic systems due to the complexity of maintaining and analyzing the graph.
* In multiple-instance resource systems, detecting deadlocks requires more than just cycle detection.