**Module 2**

**Question 1:**

To solve Question 1 from the image, we need to consider the possible values of the shared variable `B` after executing the processes P10 and P20.

### Process Definitions:

- \*\*P10:\*\*

- \( C = B - 1 \)

- \( B = 2 \* C \)

- \*\*P20:\*\*

- \( D = 2 \* B \)

- \( B = D - 1 \)

### Initial Conditions:

- The initial value of \( B \) is 2.

### Steps for Analysis:

We need to consider different execution sequences for P10 and P20 to calculate the potential final values of \( B \). There are two key execution scenarios: sequential (P10 followed by P20 or vice versa) and interleaved (partially preempting P10 with P20 or vice versa).

#### Scenario 1: P10 followed by P20

1. \*\*P10\*\* execution:

- Initial \( B = 2 \)

- Compute \( C = B - 1 = 2 - 1 = 1 \)

- Update \( B = 2 \* C = 2 \* 1 = 2 \)

- \( B \) remains 2.

2. \*\*P20\*\* execution:

- Compute \( D = 2 \* B = 2 \* 2 = 4 \)

- Update \( B = D - 1 = 4 - 1 = 3 \)

- Final \( B = 3 \).

#### Scenario 2: P20 followed by P10

1. \*\*P20\*\* execution:

- Initial \( B = 2 \)

- Compute \( D = 2 \* B = 2 \* 2 = 4 \)

- Update \( B = D - 1 = 4 - 1 = 3 \)

- \( B \) becomes 3.

2. \*\*P10\*\* execution:

- Compute \( C = B - 1 = 3 - 1 = 2 \)

- Update \( B = 2 \* C = 2 \* 2 = 4 \)

- Final \( B = 4 \).

#### Scenario 3: Interleaved Execution (P10 preempted by P20)

1. Start \*\*P10\*\* and execute:

- Initial \( B = 2 \)

- Compute \( C = B - 1 = 1 \) (but don't update \( B \) yet).

2. \*\*P20\*\* executes:

- At this point, \( B \) is still 2.

- Compute \( D = 2 \* B = 4 \)

- Update \( B = D - 1 = 3 \)

3. Resume \*\*P10\*\* and complete:

- Update \( B = 2 \* C = 2 \* 1 = 2 \)

- Final \( B = 2 \).

#### Scenario 4: Interleaved Execution (P20 preempted by P10)

1. Start \*\*P20\*\* and execute:

- Initial \( B = 2 \)

- Compute \( D = 2 \* B = 4 \) (but don't update \( B \) yet).

2. \*\*P10\*\* executes:

- At this point, \( B \) is still 2.

- Compute \( C = B - 1 = 1 \)

- Update \( B = 2 \* C = 2 \)

3. Resume \*\*P20\*\* and complete:

- Update \( B = D - 1 = 4 - 1 = 3 \)

- Final \( B = 3 \).

### Conclusion:

The possible final values of \( B \) are \*\*2, 3, and 4\*\* depending on the order and interleaving of execution between processes P10 and P20.

**Question 2**

### Answer to Question 2:

#### Given:

- Shared variable `x`, initialized to 0.

- 4 concurrent processes: W, X, Y, Z.

Each process:

- W, X: Read `x` from memory, increment by 1, store to memory, and terminate.

- Y, Z: Read `x` from memory, decrement by 2, store to memory, and terminate.

Semaphore S initialized to 2.

#### Process Details:

1. \*\*W and X\*\*: Increment `x` by 1.

2. \*\*Y and Z\*\*: Decrement `x` by 2.

3. Semaphore S ensures only two processes can modify `x` at a time.

#### Execution:

- Initial `x = 0`.

- Maximum possible value of `x`:

- W and X increment by 1 each: \( x = 0 + 1 + 1 = 2 \)

- Y and Z decrement by 2 each: \( x = 2 - 2 - 2 = -2 \)

Thus, the maximum possible value of `x` after all processes complete is \*\*2\*\*.

**Question 3**No, the presence of a cycle in a \*\*resource allocation graph (RAG)\*\* does \*\*not necessarily\*\* create a deadlock. Here's why:

- A \*\*resource allocation graph\*\* represents the allocation of resources to processes in a system. A \*\*cycle\*\* in this graph indicates a situation where a set of processes are each waiting for resources held by the other processes in the cycle.

- \*\*Deadlock\*\* happens when these processes are stuck waiting for resources indefinitely, and none of them can proceed because they're each waiting for another process in the cycle to release a resource.

However, the presence of a cycle only guarantees deadlock in specific scenarios:

1. \*\*Single Instance of Each Resource Type\*\*:

- If each resource in the system has \*\*only one instance\*\*, then the presence of a cycle in the resource allocation graph \*\*definitely indicates a deadlock\*\*. This is because no process can release the required resource, so all processes in the cycle are stuck waiting indefinitely.

2. \*\*Multiple Instances of Resource Types\*\*:

- If a resource can have \*\*multiple instances\*\* (i.e., several units of a resource type), then the presence of a cycle does \*\*not necessarily\*\* imply a deadlock. In such a case, processes could still be able to acquire and release resources in such a way that the cycle is eventually broken, and deadlock does not occur.

### Example:

- If a cycle exists in a system with multiple instances of resources, one process in the cycle may still complete, releasing enough resources for other processes in the cycle to proceed, thus avoiding deadlock.

### Conclusion:

- \*\*With a single instance of each resource type\*\*: Cycle = Deadlock.

- \*\*With multiple instances of resource types\*\*: Cycle ≠ Deadlock necessarily. It depends on how resources are allocated and released.

Would you like further clarification or examples for this?

**Question 4**

To prove that deadlock prevention mechanisms actually prevent deadlock, we need to show how they avoid the \*\*four necessary conditions\*\* for deadlock, known as the \*\*Coffman conditions\*\*:

1. \*\*Mutual Exclusion\*\*: At least one resource must be held in a non-sharable mode (i.e., only one process can use the resource at a time).

- \*\*Deadlock Prevention\*\*: Deadlock prevention avoids mutual exclusion for some resources by making them sharable (e.g., read-only resources like files). If no resource requires mutual exclusion, deadlock cannot occur.

2. \*\*Hold and Wait\*\*: A process must be holding at least one resource and waiting for additional resources that are being held by other processes.

- \*\*Deadlock Prevention\*\*: The system can either:

- Require a process to request all its resources at once, ensuring it never waits while holding.

- Or require processes to release all held resources before requesting new ones. This prevents the "hold and wait" condition, since no process is holding resources while waiting for others.

3. \*\*No Preemption\*\*: Resources cannot be forcibly taken away from processes holding them.

- \*\*Deadlock Prevention\*\*: Preemption is introduced by allowing the system to forcibly take away resources from a process if needed. For example, if a process holding some resources requests additional resources that are not available, the held resources can be released and reallocated. This breaks potential deadlocks.

4. \*\*Circular Wait\*\*: A circular chain of processes exists, where each process holds resources needed by the next process in the chain.

- \*\*Deadlock Prevention\*\*: The system imposes an ordering on resource types and requires each process to request resources in a predefined order (i.e., a process can only request a higher-ranked resource after obtaining lower-ranked ones). This ensures no circular wait condition can occur.

By preventing at least one of these conditions, deadlock is effectively avoided.

**Question 5**Let me organize the solution with clear tables and step-by-step analysis.

1. First, let's show our input tables:

Current Allocation Matrix:

| **Process** | **R1** | **R2** | **R3** | **R4** |
| --- | --- | --- | --- | --- |
| P1 | 0 | 0 | 1 | 2 |
| P2 | 2 | 0 | 0 | 0 |
| P3 | 0 | 0 | 3 | 4 |
| P4 | 2 | 3 | 5 | 4 |
| P5 | 0 | 3 | 3 | 2 |

Maximum Demand Matrix:

| **Process** | **R1** | **R2** | **R3** | **R4** |
| --- | --- | --- | --- | --- |
| P1 | 0 | 0 | 1 | 2 |
| P2 | 2 | 7 | 5 | 0 |
| P3 | 6 | 6 | 5 | 6 |
| P4 | 4 | 3 | 5 | 6 |
| P5 | 0 | 6 | 5 | 2 |

Available Resources:

| **R1** | **R2** | **R3** | **R4** |
| --- | --- | --- | --- |
| 1 | 5 | 2 | 0 |

1. Calculate Need Matrix (Maximum Demand - Current Allocation):

Need Matrix:

| **Process** | **R1** | **R2** | **R3** | **R4** |
| --- | --- | --- | --- | --- |
| P1 | 0 | 0 | 0 | 0 |
| P2 | 0 | 7 | 5 | 0 |
| P3 | 6 | 6 | 2 | 2 |
| P4 | 2 | 0 | 0 | 2 |
| P5 | 0 | 3 | 2 | 0 |

1. Let's find a safe sequence:

Step 1: Initial Available = (1, 5, 2, 0)

* P1 can run (needs ≤ available)
  + After P1: Available = (1, 5, 3, 2)

Step 2: Available = (1, 5, 3, 2)

* P5 can run (needs ≤ available)
  + After P5: Available = (1, 8, 6, 4)

Step 3: Available = (1, 8, 6, 4)

* P4 can run (needs ≤ available)
  + After P4: Available = (3, 11, 11, 8)

Step 4: Available = (3, 11, 11, 8)

* P2 can run (needs ≤ available)
  + After P2: Available = (5, 11, 11, 8)

Step 5: Available = (5, 11, 11, 8)

* P3 can run (needs ≤ available)

Safe Sequence Found: P1 → P5 → P4 → P2 → P3

1. Resource Distribution at Each Step:

|  |  |  |
| --- | --- | --- |
| **Step** | **Process Run** | **Available Resources After Step** |
| 0 | Initial | (1, 5, 2, 0) |
| 1 | P1 | (1, 5, 3, 2) |
| 2 | P5 | (1, 8, 6, 4) |
| 3 | P4 | (3, 11, 11, 8) |
| 4 | P2 | (5, 11, 11, 8) |
| 5 | P3 | All resources free |

Conclusion: The system is NOT in deadlock because:

1. We found a safe sequence: P1 → P5 → P4 → P2 → P3
2. All processes can eventually complete their execution
3. Resources are sufficient to meet the needs of processes in this sequence

Therefore, this is a safe state and no deadlock exists in the system.

**Question 6**

### Answer to Question 6:

#### Given:

- Shared Boolean variables \( S1 \) and \( S2 \).

- Two processes P10 and P20 for accessing the critical section.

#### Process logic:

- \*\*P10\*\*:

```

While (S1 == S2);

critical section

S1 = S2;

```

- \*\*P20\*\*:

```

While (S1 != S2);

critical section

S1 = not(S2);

```

#### Which is true?

- \*\*Option (b)\*\*: Mutual exclusion only.

**Case 1: S1=0,S2=0S1=0,S2=0**

* **P1**: while (S1 == S2) evaluates to while (0 == 0) → True. So, P1 waits in the loop and cannot enter the critical section.
* **P2**: while (S1 != S2) evaluates to while (0 != 0) → False. P2 can enter the critical section and then updates S1 = not(S2), so S1 becomes 1 (since S2 is 0).

**Case 2: S1=0,S2=1S1=0,S2=1**

* **P1**: while (S1 == S2) evaluates to while (0 == 1) → False. P1 can enter the critical section and then sets S1 = S2, so S1 remains 1 (as S2 = 1).
* **P2**: while (S1 != S2) evaluates to while (0 != 1) → True. P2 waits in the loop and cannot enter the critical section.

**Case 3: S1=1,S2=0S1=1,S2=0**

* **P1**: while (S1 == S2) evaluates to while (1 == 0) → False. P1 can enter the critical section and then sets S1 = S2, so S1 becomes 0.
* **P2**: while (S1 != S2) evaluates to while (1 != 0) → True. P2 waits in the loop and cannot enter the critical section.

**Case 4: S1=1,S2=1S1=1,S2=1**

* **P1**: while (S1 == S2) evaluates to while (1 == 1) → True. P1 waits in the loop and cannot enter the critical section.
* **P2**: while (S1 != S2) evaluates to while (1 != 1) → False. P2 can enter the critical section and then updates S1 = not(S2), so S1 becomes 0.

**Question 7**### Answer to Question 7:

#### Given:

- Shared Boolean variables \( P \), \( Q \), \( T \), and \( F \).

- Two processes P1 and P2 for accessing the critical section.

#### Process logic:

- \*\*P1\*\*:

```

var P = T;

while(var Q != T);

critical section

var P = F;

```

- \*\*P2\*\*:

```

var Q = T;

while(var P != T);

critical section

var Q = F;

```

#### Which is true?

- \*\*Option (a)\*\*: No Mutual exclusion + No Deadlock.

#### Explanation:

- Both processes set their respective variables \( P \) and \( Q \) to \( T \) before entering the critical section, but there is no proper synchronization between them. This allows both processes to enter the critical section simultaneously, violating mutual exclusion.

- However, there is no circular wait condition, so deadlock does not occur. Hence, there is neither mutual exclusion nor deadlock.

**Question 8**

### Answer to Question 8:

#### Given:

- Semaphore \( S \) initialized to 10.

- 6 `P(Wait)` operations and 4 `V(Signal)` operations are performed on \( S \).

#### Calculation:

1. Initial value of \( S = 10 \).

2. After 6 `P(Wait)` operations:

\[

S = 10 - 6 = 4

\]

3. After 4 `V(Signal)` operations:

\[

S = 4 + 4 = 8

\]

#### Final value of \( S \): \*\*8\*\*.

**Question 9**(same as Question1)