DC TUTORIAL 3

Fabin Chandi
S8 CSEB
Roll No.16

1. A system has 5 processes P1, P2, P3, P4, and P5. Each process has its own local clock, and all processes are running Lamport's mutual exclusion algorithm. Process P1 sends a request to all other processes at time 10. The process receives replies from the others at times 12, 15, 18, and 20. What will be the timestamp of the critical section request for P1?

Ans)Lamport's mutual exclusion algorithm ensures a logical order of events in a distributed system by assigning timestamps to requests based on the logical clock of the requesting process. The timestamp is determined at the moment the request is made.

In this scenario:

- Process P1 sends a request at its local time 10 to all other processes.
- The other processes respond at times 12, 15, 18, and 20.

Since Lamport's algorithm assigns timestamps based on the sender's clock at the moment the request is made, the timestamp for P1's request will be 10.

This ensures that all processes can order the request correctly in relation to others, resolving conflicts when multiple processes request the critical section simultaneously.

2. In a distributed system using the Ricart-Agrawala algorithm, Process P1 wants to enter the critical section and sends a request to processes P2, P3, and P4. Process P3 is already in its critical section and cannot reply. Process P2 replies with a timestamp 8, and process P4 replies with timestamp 6. What will be the earliest time at which P1 can enter the critical section?

Ans)The Ricart-Agrawala algorithm is a timestamp-based request-reply algorithm for mutual exclusion. A process can enter the critical section only after receiving replies from all other processes that are not currently executing the critical section.

Given the situation:

- P1 sends a request to P2, P3, and P4.
- P3 is already in the critical section, so it does not send a reply.
- P2 replies with a timestamp of 8, meaning P2 is free and acknowledges P1's request.
- P4 replies with a timestamp of 6, indicating P4 is also free.

Since P3 is currently inside the critical section, P1 must wait until P3 exits and sends a reply before it can enter. The earliest time at which P1 can enter is when P3 finishes its execution and sends the reply.

Without knowing exactly when P3 exits, we can only say that P1 must wait for P3 before proceeding.

3. Consider a system with 4 processes: P1, P2, P3, and P4. In a quorum-based mutual exclusion algorithm with voting, each process is part of 3 quorums, and each quorum consists of 3 processes. If P1 wants to enter the critical section, how many processes must it contact for permission to enter?

Ans)In quorum-based mutual exclusion, each process is part of multiple quorums, and every quorum consists of a subset of processes. A process must receive permission from a majority (or a certain required number) of processes in at least one quorum before entering the critical section.

Given:

- There are 4 processes (P1, P2, P3, P4).
- Each process is part of 3 quorums.
- Each quorum consists of 3 processes.

To ensure mutual exclusion, each quorum must overlap with others in a way that prevents two processes from accessing the critical section at the same time. If P1 wants to enter, it must obtain permission from a quorum, which consists of 3 processes (including itself).

Thus, P1 must contact 2 other processes in one of its quorums.

4. In a Maekawa's quorum-based algorithm for mutual exclusion, process P1 is part of quorums Q1, Q2, and Q3, and each quorum consists of 3 processes. If process P2 is part of quorums Q2, Q3, and Q4, and process P3 is part of Q1, Q2, and Q4, what can be inferred about the relationship between the quorums to guarantee mutual exclusion?

Ans)Maekawa's quorum-based algorithm is designed so that:

1. Every process belongs to at least one quorum.
2. Every quorum overlaps with others to prevent two processes from accessing the critical section simultaneously.

Given the quorum memberships:

- P1 belongs to Q1, Q2, and Q3.
- P2 belongs to Q2, Q3, and Q4.
- P3 belongs to Q1, Q2, and Q4.

We can analyze their relationships:

- Q1 and Q2 share P1 and P3.
- Q2 and Q3 share P1 and P2.
- Q2 and Q4 share P2 and P3.

Since there is at least one common process in every intersection, mutual exclusion is guaranteed. This ensures that if one process is in the critical section, another cannot enter unless it also gets permission from the shared process.

5. In a Suzuki-Kasami token-based algorithm, there are 6 processes in the system, and the token is initially held by P1. If P1 has completed its critical section and wants to pass the token to P2, describe the sequence of events that must occur for P2 to acquire the token.

Ans)The Suzuki-Kasami algorithm is a token-based approach where a unique token circulates among processes, granting access to the critical section.

Given that P1 holds the token and wants to pass it to P2, the following steps occur:

1. P1 finishes its critical section and checks if any other process has requested the token.
2. If P2 has sent a request, P1 updates its request queue.
3. P1 sends the token to P2 using a message over the network.
4. P2 receives the token and enters the critical section.
5. When P2 finishes, it repeats the process by checking the next request and passing the token accordingly.

Since the token is the only way to enter the critical section, this method ensures that no two processes access it at the same time.

6. In a distributed system, there are 4 processes (P1, P2, P3, P4) and 3 resources (R1, R2, R3). At a certain point, the following holds: P1 holds R1 and is waiting for R2. P2 holds R2 and is waiting for R3. P3 holds R3 and is waiting for R1. P4 holds no resources but is waiting for R1. Is this system deadlocked? Explain why or why not.

Ans)A deadlock occurs when a cycle forms in the wait-for graph—a structure where nodes represent processes and edges represent resource dependencies.

Given the scenario:

- P1 holds R1 and waits for R2.
- P2 holds R2 and waits for R3.
- P3 holds R3 and waits for R1.
- P4 waits for R1 but holds no resource.

Analyzing the dependencies:

- P1 → R2 → P2
- P2 → R3 → P3
- P3 → R1 → P1 (Cycle detected)

Since P1, P2, and P3 form a circular wait condition, a deadlock exists. P4 is waiting but not causing a cycle, so it is just blocked, not deadlocked.

7. Given the following deadlock handling strategies in a distributed system: Prevention: Ensure that one of the deadlock conditions is violated. Avoidance: Avoid resource allocation that could lead to deadlock. Detection and Recovery: Periodically check for deadlocks and recover by aborting or preempting processes. Which strategy would be most efficient in a system with low contention for resources and why?

Ans)Three deadlock-handling strategies exist:

1. Prevention – Eliminates one of the four necessary conditions for deadlock (e.g., force processes to request all resources at once).
2. Avoidance – Uses algorithms (like the Banker's algorithm) to ensure resources are only allocated if deadlock can be avoided.
3. Detection & Recovery – Allows deadlocks to occur but detects them and resolves them by preempting processes.

For low contention systems, the best strategy is detection and recovery because:

- Deadlocks are infrequent, making prevention or avoidance unnecessary overhead.
- Instead of restricting resource allocation, we only detect deadlocks when needed.
- Recovery mechanisms (like terminating a process or reclaiming resources) resolve deadlocks efficiently.

Thus, detection and recovery are more practical and efficient in such cases.

8. A system uses a distributed deadlock detection algorithm, where each process sends a message to its neighbors when it holds a resource and waits for another. If a process detects a cycle, it considers itself to be in a deadlock. What are the challenges that arise when implementing deadlock detection in a distributed system?

Ans)Detecting deadlocks in a distributed system is harder than in centralized systems because of:

1. Message Delays – Network delays may cause outdated information, leading to incorrect deadlock detection.
2. Inconsistent States – Processes may update their resource allocation asynchronously, making it difficult to detect cycles in real-time.
3. False Positives – A process might assume a deadlock exists when another process is just slow in responding.
4. Communication Overhead – Checking for deadlocks requires sending many messages, which increases network congestion.

To handle these issues, distributed systems often use:

- Timeout-based detection (assume a process is deadlocked if it doesn't respond in time).
- Probe-based detection (a process sends probes through the network to detect cycles).
- Decentralized monitoring (where multiple nodes work together to detect deadlocks).

These techniques help minimize overhead while ensuring accurate detection.