Explain about Lamport’s algorithm for distributed mutual exclusion.

Ss

Discuss about Ricart and Agrawala’s algorithm for mutual exclusion

with a suitable example Diagram and how it is different from Lamport’s algorithm

Ricart and Agrawala Algorithm (12 Marks)

The Ricart-Agrawala Algorithm is a distributed algorithm used to achieve mutual exclusion in distributed systems. It ensures that only one process can enter its critical section (CS) at a time, and it assumes that the communication channels are FIFO (First In, First Out).

Key Concepts:

1. Two Types of Messages:

- REQUEST: A message sent by a process to all other processes requesting permission to enter the CS.

- REPLY: A message sent to a process granting it permission to enter the CS.

2. Timestamping:

- The algorithm uses Lamport logical clocks to timestamp each request. The timestamps help in determining the priority of requests when two or more processes request the CS simultaneously.

- The process with the smaller timestamp gets higher priority.

3. Requesting the Critical Section:

- A process sends a REQUEST message

to all other processes when it wants to enter the CS.

- Upon receiving a REQUEST, the receiving process sends a REPLY if it is not currently in the CS or does not have a higher-priority request.

- If the receiving process has a higher-priority request, it defers the REPLY and will send it only after exiting the CS.

4. Request Deferral:

- Each process maintains a Request-Deferred (RD) array, where each entry corresponds to another process. If a request is deferred, the array entry is set to 1, and when a REPLY is sent, the entry is reset to 0.

5. Executing the Critical Section:

- A process enters the CS only after receiving REPLY messages from all other

processes to whom it sent REQUEST messages.

6. Releasing the Critical Section:

- Once a process exits the CS, it sends all deferred REPLY messages to other processes.

Algorithm Steps:

1. Requesting the CS:

- A process broadcasts a timestamped REQUEST message to all other processes.

- If a process receives a REQUEST and it is not requesting or using the CS, or the requesting process has a smaller timestamp, it sends a REPLY. Otherwise, it defers the REPLY and updates its RD array.

2. Entering the CS:

- A process enters the CS after receiving

REPLY messages from all other processes.

3. Exiting the CS:

- A process sends deferred REPLY messages and resets the corresponding entries in the RD array.

Example:

- Scenario: Two processes, S1 and S2, want to enter the CS. S1 has a timestamp (2,1) and S2 has a timestamp (1,2). S2 has the smaller timestamp, so it gets to enter the CS first.

- Process Flow:

- S2 sends REQUEST messages to other processes and collects REPLYs.

- After exiting, S2 sends a REPLY to S1, allowing S1 to enter the CS.

Advantages:

- Fairness: The use of timestamps

ensures that the process with the earliest request enters the CS first.

- Efficiency: Only two messages (REQUEST and REPLY) are exchanged between processes, reducing message complexity.

---diagram

Difference between Ricart-Agrawala Algorithm and Lamport's Algorithm (4 Marks)

| Aspect | Ricart-Agrawala Algorithm | Lamport’s Algorithm |

|-------------------------------|---------------------------------------------------------------------------|----------------------------------------------------------------------------|

| Number of Messages | 2(n-1)

messages: one REQUEST and one REPLY per process | 3(n-1) messages: REQUEST, REPLY, and RELEASE messages per process |

| Message Types | Only REQUEST and REPLY messages | REQUEST, REPLY, and RELEASE messages |

| Entry to Critical Section | Requires receiving REPLY messages from all other processes | Requires receiving REPLY messages from all processes and RELEASE after execution |

| Blocking | A process is blocked only by higher-priority requests | A process may be blocked even by processes with equal priority timestamps |

| Priority | Based on timestamps; smaller timestamp gets priority | Also based on

timestamps, but requires explicit RELEASE messages to free resources |

| Message Complexity | Lower message complexity, as RELEASE messages are not required | Higher message complexity due to additional RELEASE messages |

Explain the Suzuki–Kasami’s broadcast algorithm with an example

Suzuki-Kasami Algorithm

The Suzuki-Kasami Algorithm is a token-based algorithm for achieving distributed mutual exclusion in a system where multiple processes share resources. The key component is the token. If a site has the token, it can enter the Critical Section (CS). If a site

doesn't have the token and wants to enter the CS, it requests the token from the other sites. The algorithm ensures that only one site at a time holds the token, guaranteeing mutual exclusion.

Key Features:

1. Token-based: There is a single token that is circulated among processes.

2. Request Broadcast: If a process doesn't have the token, it broadcasts a REQUEST message to all other processes.

3. Token Queue: The token maintains a queue of requests (sites waiting to enter the CS).

4. Efficient Messaging: The algorithm uses minimal messages, only N messages (where N is the number of processes) to request and obtain the token.

Design Issues:

1. Outdated REQUEST messages: A process might receive a REQUEST message after the request has already been satisfied. To avoid sending tokens unnecessarily, the algorithm tracks the sequence number of each request.

2. Determining Outstanding Requests: After a site finishes using the CS, it needs to decide which site should receive the token next. This is determined by checking which sites have outstanding requests for the token.

Data Structures:

- REQUEST(j, n): A request message from site j for its nth critical section execution.

- RNi[1..N]: Array that stores the largest sequence number of requests received from each site.

- Token Queue (Q): A queue in the token that stores the list of sites waiting to enter the CS.

- LN[1..N]: An array that stores the sequence number of the last request executed by each site.

Algorithm Steps:

1. Requesting the Critical Section:

- If a site Si wants to enter the CS but doesn't have the token, it increments its sequence number RNi[i] and broadcasts a REQUEST(i, sn) to all other sites.

- When a site Sj receives this REQUEST message, it updates its array RNj[i] with the maximum sequence number and, if Sj has the idle token, it sends the token to Si.

2. Executing the Critical Section:

- Si executes the CS after receiving the token.

3. Releasing the Critical Section:

- Once Si finishes using the CS, it updates the token array LN[i] to reflect the sequence number of its last request.

- For each site Sj that has an outstanding request, Si adds Sj to the token queue if it hasn’t already.

- If the token queue is non-empty, Si sends the token to the site at the front of the queue.

Example:

Suppose there are 3 sites S1, S2, and S3. Initially, S1 has the token. The process proceeds as follows:

1. S2 wants to enter the CS but does not have the token. It increments its sequence number and broadcasts a REQUEST(2, 1) to S1 and S3.

2. S1 receives the request. Since it has the token and isn't using the CS, it sends the token to S2.

3. S2 receives the token and enters the CS.

4. Once S2 finishes, it checks if any other site has requested the token. If S3 has made a request (say REQUEST(3, 1)), S2 sends the token to S3.

Performance:

- No message overhead if the site already holds the token.

- N messages are required if the site needs to request the token.

- Synchronization delay is either 0 (if

the token is already held) or T (the time taken to receive the token after broadcasting REQUEST).

Advantages:

- Simple and efficient for systems with a low number of requests.

- Ensures mutual exclusion and fairness.

Disadvantages:

- If a site that holds the token fails, the system becomes blocked.

- The overhead increases with the number of sites due to token management.

Discuss the issues of Deadlock detection in detail

Issues in Deadlock Detection

Deadlock handling using the approach of

deadlock detection entails addressing two basic

issues:

1. Detection of existing deadlocks

2. Resolution of detected deadlocks.

1. Detection of Deadlocks

Sure! Here’s an easy-to-understand explanation of deadlock detection and resolution issues:

1. Detection of Deadlocks

Deadlock detection is about figuring out if processes in a system are stuck waiting for each other. Here are the key points:

- Wait-For Graph (WFG): This is a

diagram that shows which process is waiting for which resource. Each process points to the resources it needs.

- Finding Cycles: If there's a cycle in the WFG (like a loop where Process A waits for B, B waits for C, and C waits for A), it means there's a deadlock. In distributed systems, this can be tricky because the WFG may be spread across different sites.

- Types of Algorithms: There are different ways to detect deadlocks:

- Centralized: One central site checks for deadlocks.

- Distributed: Each site checks its own WFG and shares information with others.

- Hierarchical: A mix of centralized and distributed methods.

Correctness Criteria for Deadlock

Detection Algorithms

1. Progress: The algorithm must find deadlocks within a finite time. It should keep checking until it finds any deadlock.

2. Safety: The algorithm should not mistakenly report a deadlock that doesn’t exist (known as a false deadlock). In distributed systems, outdated information can lead to incorrect conclusions about deadlocks.

2. Resolution of Detected Deadlocks

Once a deadlock is detected, it must be resolved. Here’s how it works:

- Breaking Dependencies: This means that the system has to break the cycle of processes waiting on each other.

- Rolling Back: The system might need to roll back one or more processes to free up resources.

- Cleaning Up: After breaking a dependency, the system should promptly remove any outdated information about the deadlock to prevent confusion. If this isn’t done properly, it might lead to false deadlocks in future detections.

In summary, deadlock detection is about identifying if processes are stuck and ensuring that the methods used to do this are accurate and efficient. Resolving deadlocks involves breaking the cycles and managing resources carefully to keep the system running smoothly.

Ss

Ss

Mitchell - mam sent