Mutual Exclusion

Tokenless and Token Based Algorithms

Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India

Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Motivation

- Lamport's algorithm required 3(N-1) messages.
- Insight:

Question

Do we really have to send a timestamped reply message?

Motivation

- Lamport's algorithm required 3(N-1) messages.
- Insight:

Question

Do we really have to send a timestamped reply message?

- Solution
  - Lamport's algorithm sent an acknowledgement immediately.
  - Let us hold on to the acknowledgement and piggy back it with a release message.
  - We can reduce the number of messages per critical section to 2(N-1).

Algorithm

Requesting the Lock

- P_i sends a timestamped request message to all other nodes.
- When P_j receives a request, it sends a reply if:
  - P_j is neither holding the lock, not is it interested in acquiring it. **OR**
  - P_i's request timestamp is smaller than P_j's request timestamp, and P_j is not holding the lock. (means that P_i made an earlier request)

Acquiring and Releasing the Lock

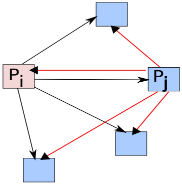
Acquiring the Lock

A process acquires the lock when it has received $N - 1$ replies .

Releasing the Lock

A process replies to all pending requests after it releases the lock.

Proof



Proof

- Assume P_i and P_j both have a lock at the same time.
- Assume that P_i has a lower request timestamp.
- This means that P_i must have gotten P_j's request after its request.
- According to the algorithm, P_i cannot send a reply to P_j.
- Hence, P_j does not have the lock.

Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

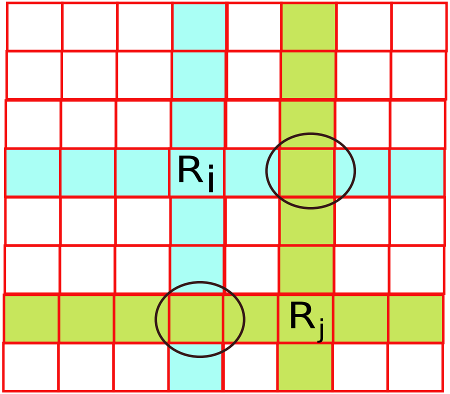
Can we do better?

- The main reason for a linear number of messages is:
  1. We send a message to all the sites.
  2. We also expect replies from all of them.

Can we do better?

- The main reason for a linear number of messages is:
  1. We send a message to all the sites.
  2. We also expect replies from all of them.
- Can we send a message to a subset of sites?
  - Let a set of processes associated with a process be called its request set (R_i).
  - For any two processes, P_i and P_j, we have: $R_i \cap R_j \neq 0$.
  - The minimum value of $\mid R_i \mid$ is $\sqrt{N}$.
  - It is possible to construct this using results from field theory.

Simpler construction $R_i = 2\sqrt{N}$



Acquiring a Lock

- P_i sends a timestamped request message to every node in R_i including itself.
- Upon receiving a request message a node $P_j \in R_i$ marks itself as locked , if it is not already locked. It returns a locked reply to P_i.

Acquiring a Lock

- P_i sends a timestamped request message to every node in R_i including itself.
- Upon receiving a request message a node $P_j \in R_i$ marks itself as locked, if it is not already locked. It returns a locked reply to P_i.
- If P_j is already locked by a request from P_k.
    1. P_j places the request in a wait queue.
    2. If the locking request or any other request in the queue precedes the current request, then send a failed message.
    3. Otherwise, send an inquire message to P_k.

Acquiring a Lock - II

- When P_k receives an inquire message:
  1. If P_k has received a failed message, and knows that it cannot succeed, it sends a relinquish message.
  2. Otherwise, it defers the reply.

Acquiring a Lock - II

- When P_k receives an inquire message:
    1. If P_k has received a failed message, and knows that it cannot succeed, it sends a relinquish message.
    2. Otherwise, it defers the reply.
- When P_j receives the relinquish message:
    1. It locks itself for the earliest message from P_i' in its wait queue (might be P_i).
    2. It sends a locked message to P_i'.
    3. It adds the request from P_k to its wait queue.

Acquiring a Lock - II

- When P_k receives an inquire message:
  1. If P_k has received a failed message, and knows that it cannot succeed, it sends a relinquish message.
  2. Otherwise, it defers the reply.
- When P_j receives the relinquish message:
  1. It locks itself for the earliest message from P_i' in its wait queue (might be P_i).
  2. It sends a locked message to P_i'.
  3. It adds the request from P_k to its wait queue.
- A process acquires the lock when it has received locked messages from its entire request set.

Releasing the Lock

- A process sends released messages to all the processes in its request set.
- A process in the request set, locks itself for the earliest request in the wait queue. It sends it a locked message.
- If there is no such request, then it marks its status as unlocked .

Proof

Mutual Exclusion

Assume two processes P_i and P_j have the lock simultaneously.

- There must be a node P_k that must have given locked messages to both the processes.
- This is not possible .

Deadlock

Not possible because because we order the requests by their timestamp.

Starvation

Ultimately, a request will become the earliest message in the system.

Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Main Idea

- A site can access the lock(critical section) if it has a token.
- Every process maintains a sequence number (request id)
  - A request is of the form (i, m) . This means that P_i wants its m^{th} access to the lock.
  - P_i keeps an array $seq_i[1 \ldots N]$.
  - $seq_i[j]$ is the largest sequence number received from j.
  - When P_i receives (j, m) , it sets

$$seq_i[j] = max(seq_i[j], m)$$

- Token
  1. A queue(Q) of requesting sites.
  2. An array of sequence numbers C.
  3. $C[i]$ is the sequence number of the latest request that P_i executed.

Requesting the Lock

Requesting the Lock

- P_i: $seq_i[i] ++$, $val \leftarrow seq_i[i]$
- Sends (i, val) to all sites
- When P_j receives (i, val)
    - $seq_j[i] \leftarrow max(seq_j[i], val)$
    - If the token is idle and with P_j then it sends it to P_i, if $(seq_j[i] = C[i] + 1)$.
- P_i enters the critical section when it has the token.

Releasing the Lock

Releasing the Lock

P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q , and sends the token to P_k

Releasing the Lock

Releasing the Lock

P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q , and sends the token to P_k

Message Overhead: 0 or N

Proof

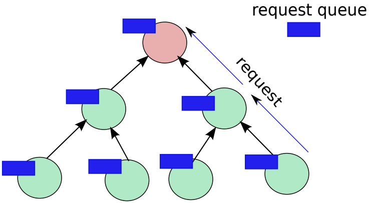
A requesting process gets the lock in finite time.

- The request will reach all the processes in finite time.
- By induction, one of these processes will have the token in finite time.
- Thus the current request will get added to Q .
- There can at most be $N - 1$ messages before it.

Outline

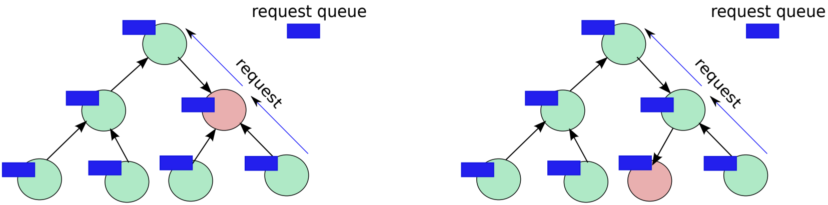
1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Main Idea



- Nodes are arranged as a tree. Every node has a parent pointer.
- Each node has a FIFO queue of requests.
- There is one token with the root and the owner of the token can enter the critical section.
- Message Complexity: approximately $O(log(N))$ for trees with high fan-out

Dynamic Nature of the Tree



- As the token moves across nodes, the parent pointers change.
- They always point towards the holder of the token.
- It is thus possible to reach the token by following parent pointers.

Requesting for the Token

Requesting a Token

- The node adds self in its request queue.
- Forwards the request to the parent.
- The parents adds the request to its request queue.
- If the parent does not hold the token and it has not sent any requests to get the token, it sends a request to its parent for the request.
- This process continues till we reach the root (holder of the token).

Releasing a Token

Releasing a Token

- Ultimately a request will reach the token holder.
- The token holder will wait till it is done with the critical section.
- It will forward the token to the node at the head of its request queue.
  - It removes the entry.
  - It updates its parent pointer.
- Any subsequent node will do the following:
  - Dequeue the head of the queue.
  - If self was at the head of its request queue, then it will enter the critical section.
  - Otherwise, it forwards the token to the dequeued entry.
- After forwarding the entry, a process needs to make a fresh request for the token , if it has outstanding entries in its request queue.

Correctness

Mutual Exclusion

1. No two nodes can have a token at the same time, and thus cannot be in the CS at one time.

Deadlock

1. Circular wait cannot occur because all the nodes wait on the node that holds the token .
2. Messages cannot get lost because all the time our parent pointers ensure that we have a rooted tree.

Starvation

1. Ultimately a starved process's request will come to the front of all the request queues.
2. At this point it will have the highest priority, and the token will have to flow back to the starved process.

A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems by Mamoru Maekawa, ACM Transactions on Computer Systems, 1985