CSCI 460 — Operating Systems

Lecture 14

Distributed Mutual Exclusion/Deadlock

Textbook: Operating Systems
by William Stallings

1. Distributed Mutual Exclusion Concepts

- Mutual Exclusion Requirements
  - 1. Only one process at one time is allowed to enter the critical region.
  - 2. A process that halts in its non-critical region must not interfere with other processes.
  - 3. The request of a process to enter a critical region must not be delayed indefinitely.
  - 4. When the critical region is free, any other process is permitted to enter it without delay.
  - 5. No assumptions are made about relative process speeds/number of processors.
  - 6. The critical region is of limited time.

2. Centralized algorithm for mutual exclusion

- **Idea:** one node is designated as the control node and controls access to all shared objects.
  - 1. Only the control node makes resource-allocation decisions.
  - 2. The control node keeps the information (identity & status) of each resource.

  Consequence?

  Not very much different from the usual mutual exclusion control.

- **Drawbacks**
  - 1. If the control node fails, then mutual exclusion control breaks down, at least temporarily.
  - 2. The control node might become a bottleneck in system performance.

3. Distributed algorithms for mutual exclusion

- A fully distributed algorithm should have the following properties.
  - 1. All nodes have roughly equal amount of information.
  - 2. Each node has only some local information of the system.
  - 3. All nodes expand roughly equal effort in making a final decision.
  - 4. Failure of a node does not make the system collapse.
  - 5. There is no systemwide common clock for the whole system.

4. How to handle the problem of no systemwide clock?

- Problem: Assume that event a of system i occurred before event b at system j, we want to make sure that this conclusion is consistent among all nodes in the system.

  Why this is a problem?

- **Timestamp:** a method which orders events in a distributed system without using system clocks [Lamport, 1978].
  - 1. Each system i maintains a local counter C_i (which functions like a clock).
  - 2. When a system i transmits a message, it first increments its clock by 1 and sends the message in the form of (m, T_i, i).
  - 3. The receiving system j sets its clock by $C_j \leftarrow 1 + \max[C_j, T_i]$.
  - 4. For message x from system i and message y from system j, x **precedes** y if either $T_i < T_j$ or $T_i = T_j$ and $i < j$.

5. Distributed Queue Solution [Lamport,78]

- **Assumptions**:
  - 1. N nodes, each with a process which is in charge of mutual exclusion requests.
  - 2. Messages are received in the same order as they are sent.
  - 3. All messages are delivered in a finite period of time.
  - 4. A node can send a message to all other nodes.
  - 5. Each node keeps an array (queue) q. At any time $q[j]$ in the local array contains a message from P_j.
- Similar to a centralized system, all of the sites have a copy of the common queue.
- **One more assumption:** before a process makes a decision based on its own queue, it must have received a message from all other sites.

  Can you see why we need this assumption?

- Three types of messages are used in this algorithm:
  - 1. $(request, T_i, i)$: P_i makes a request to access a resource at time T_i.
  - 2. $(reply, T_j, j)$: P_j grants access to a resource under its control.
  - 3. $(release, T_k, k)$: P_k releases a resource previously allocated to it.

- **Algorithm:**
  - 1. When P_i wants to access resource, it sends a message $(request, T_i, i)$ to all other processes and it also stores the message in $q[i]$.
  - 2. When P_j receives $(request, T_i, i)$, it stores the message in its own $q[i]$. If $q[j]$ does not contain a request message then P_j sends $(reply, T_j, j)$ to P_i.
  - 3. P_i can access a resource when both of these conditions hold:
    (a). P_i's own request message (stored in $q[i]$) is the earliest request message in q.
    (b). All other messages in q are later than the message in $q[i]$.
  - 4. When P_i exits from the critical region, it sends $(release, T_i, i)$ to every process.
  - 5. When P_i receives $(release, T_j, j)$, it replaces the current content of $q[j]$ with this message.
  - 6. When P_i receives $(reply, T_j, j)$, it replaces the current content of $q[j]$ with this message.

- **Conclusion for Lamport's Solution:**
  - 1. Mutual exclusion is enforced.
  - 2. The algorithm is fair, i.e., requests are granted according to the timestamp ordering.
  - 3. Deadlock free.
  - 4. Starvation free.
- **Question:** to guarantee mutual exclusion, how many messages are required?

6. Improved Distributed Queue Solution

- **Assumptions**: same as before, except that we do not necessarily require that messages sent from a process are received in the same order.
- **Algorithm:**
  - 1. When P_i wants to access resource, it sends a message $(request, T_i, i)$ to all other processes and it also stores the message in $q[i]$.
  - 2. When P_j receives $(request, T_i, i)$, it does the following:
    (a). If P_j is currently in its critical region, it defers sending a REPLY message.
    (b). If P_j is not waiting to enter its critical region, it sends $(reply, T_j, j)$ to P_i.
    (c). If P_j is waiting to enter its critical region and if the incoming message follows P_j's request, then it stores this message in $q[i]$ and defers sending a REPLY message.
    (d). If P_j is waiting to enter its critical region and if the incoming message precedes P_j's request, then it stores this message in $q[i]$ and sends $(reply, T_j, j)$ to P_i.
  - 3. When P_i receives $(reply, T_j, j)$ for all P_j, it can access a resource.
  - 4. When P_i exits from the critical region, it sends $(reply, T_i, i)$ to all pending processes (i.e. process sends a request message and is waiting).

- **Question:** to guarantee mutual exclusion, how many messages are required with this new solution?

This new solution was proposed by Ricart and Agrawala (1981).

7. A Token-Passing Approach

- **Token**: an entity which is held by one process at any time.

- Whichever process holds the token can enter its critical region (without asking any permission); when it leaves its critical region, it passes the token to another process.
- **Algorithm:**

- **Question:** to guarantee mutual exclusion, how many messages are required with this solution?

This solution was proposed by Suzuki and Kasami (1982).

8. Some Famous Distributed Algorithms

- Leadership Election:
  - **1**. Each process has a unique ID known to all members.
  - **2**. The process with the highest ID is the leader.
  - **3**. Any process may fail at any time.
- Algorithm 1—**The BULLY election algorithm**:
  all process do the following
  - **1**. P notices there is no reply from the coordinator.
  - **2**. P sends an **elect** message to all processes with higher IDs.
  - **3**. If there is any reply then P exits.
  - **4**. If there is no reply then P wins, obtains any state needed to function as a leader, then sends a **coordinator** message to all processes.
  - **5**. On receipt of an **elect** message a process must both reply to the sender and start an election if it is not already holding one.

- Algorithm 2—**A ring-based election algorithm**:
  all process do the following
  - **1**. P notices the coordinator is not functioning.
  - **2**. P sends an **elect** message containing its own ID to the next process in the ring.
  - **3**. On receipt of an **elect** message
    a without the receiver's ID — add this ID and pass on the message.
    b with the receiver's ID (the message has been round the ring)—send a message (**coordinator**, highest ID in the message) around the ring.