CSE 421/521 - Operating Systems
Fall 201

Lecture - XXV

Distributed Systems - II

Tevfik Koşar

University at Buffalo
December 3rd, 2013
Distributed Mutual Exclusion (DME)

• Assumptions
  - The system consists of \( n \) processes; each process \( P_i \) resides at a different processor
  - Each process has a critical section that requires mutual exclusion

• Requirement
  - If \( P_i \) is executing in its critical section, then no other process \( P_j \) is executing in its critical section

• We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections
DME: Centralized Approach

• One of the processes in the system is chosen to coordinate the entry to the critical section
• A process that wants to enter its critical section sends a request message to the coordinator
• The coordinator decides which process can enter the critical section next, and its sends that process a reply message
• When the process receives a reply message from the coordinator, it enters its critical section
• After exiting its critical section, the process sends a release message to the coordinator and proceeds with its execution
• This scheme requires three messages per critical-section entry:
  - request
  - reply
  - release
DME: Fully Distributed Approach

- When process $P_i$ wants to enter its critical section, it generates a new timestamp, $TS$, and sends the message \textit{request} $(P_i, TS)$ to all processes in the system.
- When process $P_j$ receives a \textit{request} message, it may reply immediately or it may defer sending a reply back.
- When process $P_i$ receives a \textit{reply} message from all other processes in the system, it can enter its critical section.
- After exiting its critical section, the process sends \textit{reply} messages to all its deferred requests.
DME: Fully Distributed Approach (Cont.)

- The decision whether process $P_j$ replies immediately to a request($P_i$, $TS$) message or defers its reply is based on three factors:
  - If $P_j$ is in its critical section, then it defers its reply to $P_i$
  - If $P_j$ does not want to enter its critical section, then it sends a reply immediately to $P_i$
  - If $P_j$ wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp $TS$
    - If its own request timestamp is greater than $TS$, then it sends a reply immediately to $P_i$ ($P_i$ asked first)
    - Otherwise, the reply is deferred

- **Example:** $P_1$ sends a request to $P_2$ and $P_3$ (timestamp=10)
  - $P_3$ sends a request to $P_1$ and $P_2$ (timestamp=4)
Undesirable Consequences

• The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex

• If one of the processes fails, then the entire scheme collapses
  - This can be dealt with by continuously monitoring the state of all the processes in the system, and notifying all processes if a process fails
Token-Passing Approach

• Circulate a token among processes in system
  - **Token** is special type of message
  - Possession of token entitles holder to enter critical section

• Processes *logically* organized in a **ring structure**

• Unidirectional ring guarantees freedom from starvation

• Two types of failures
  - Lost token - election must be called
  - Failed processes - new logical ring established
Election Algorithms

- Determine where a new copy of the coordinator should be restarted
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process $P_i$ is $i$
- Assume a one-to-one correspondence between processes and sites
- The coordinator is always the process with the highest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures
Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system.

- If process $P_i$ sends a request that is not answered by the coordinator within a time interval $T$, assume that the coordinator has failed; $P_i$ tries to elect itself as the new coordinator.

- $P_i$ sends an election message to every process with a higher priority number, $P_i$ then waits for any of these processes to answer within $T$. 
Bully Algorithm (Cont.)

- If no response within $T$, assume that all processes with numbers greater than $i$ have failed; $P_i$ elects itself the new coordinator.

- If answer is received, $P_i$ begins time interval $T'$, waiting to receive a message that a process with a higher priority number has been elected.

- If no message is sent within $T'$, assume the process with a higher number has failed; $P_i$ should restart the algorithm.
Bully Algorithm (Cont.)

- If \( P_i \) is not the coordinator, then, at any time during execution, \( P_i \) may receive one of the following two messages from process \( P_j \)
  - \( P_j \) is the new coordinator \((j > i)\). \( P_i \), in turn, records this information
  - \( P_j \) started an election \((j < i)\). \( P_i \), sends a response to \( P_j \) and begins its own election algorithm, provided that \( P_i \) has not already initiated such an election

- After a failed process recovers, it immediately begins execution of the same algorithm

- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number
Ring Algorithm

- Applicable to systems organized as a ring (logically or physically)

- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors

- Each process maintains an active list, consisting of all the priority numbers of all active processes in the system when the algorithm ends

- If process Pi detects a coordinator failure, it creates a new active list that is initially empty. It then sends a message elect(i) to its right neighbor, and adds the number i to its active list
Ring Algorithm (Cont.)

- If $P_i$ receives a message $\text{elect}(j)$ from the process on the left, it must respond in one of three ways:

  ✦ If this is the first $\text{elect}$ message it has seen or sent, $P_i$ creates a new active list with the numbers $i$ and $j$
    - It then sends the message $\text{elect}(i)$, followed by the message $\text{elect}(j)$
  ✦ If $i \neq j$, the message does not contain $P_i$
    - $P_i$ adds $j$ to its active list and forwards the message to the right
  ✦ If $i = j$, then $P_i$ receives the message $\text{elect}(i)$
    - The active list for $P_i$ contains all the active processes in the system
    - $P_i$ can now determine the largest number in the active list to identify the new coordinator process
Distributed Deadlock Handling

• Resource-ordering deadlock-prevention
  => define a *global* ordering among the system resources
  - Assign a unique number to all system resources
  - A process may request a resource with unique number \( i \) only if it is not holding a resource with a unique number greater than \( i \)
  - Simple to implement; requires little overhead

• Timestamp-ordering deadlock-prevention
  => unique Timestamp assigned when each process is created

  1. wait-die scheme -- non-reememptive
  2. wound-wait scheme -- preemptive
Prevention: Wait-Die Scheme

- non-preemptive approach
- If $P_i$ requests a resource currently held by $P_j$, $P_i$ is allowed to wait only if it has a smaller timestamp than does $P_j$ ($P_i$ is older than $P_j$)
  - Otherwise, $P_i$ is rolled back (dies - releases resources)

- Example: Suppose that processes $P_1$, $P_2$, and $P_3$ have timestamps 5, 10, and 15 respectively
  - if $P_1$ request a resource held by $P_2$, then $P_1$ will wait
  - If $P_3$ requests a resource held by $P_2$, then $P_3$ will be rolled back

- The older the process gets, the more waits
Prevention: Wound-Wait Scheme

• Preemptive approach, counterpart to the wait-die

• If $P_i$ requests a resource currently held by $P_j$, $P_i$ is allowed to wait only if it has a larger timestamp than does $P_j$ ($P_i$ is younger than $P_j$). Otherwise $P_j$ is rolled back ($P_j$ is wounded by $P_i$)

• Example: Suppose that processes $P_1$, $P_2$, and $P_3$ have timestamps 5, 10, and 15 respectively
  - If $P_1$ requests a resource held by $P_2$, then the resource will be preempted from $P_2$ and $P_2$ will be rolled back
  - If $P_3$ requests a resource held by $P_2$, then $P_3$ will wait

• The rolled-back process eventually gets the smallest timestamp.
Comparison

• Both avoid starvation, provided that when a process is rolled back, it is not assigned a new timestamp

• In wait-die, older process must wait for the younger one to release its resources. In wound-wait, an older process never waits for a younger process.

• There are fewer roll-backs in wound-wait.
  - Pi->Pj; Pi dies, requests the same resources; Pi dies again…
  - Pj->Pi; Pi wounded. requests the same resources; Pi waits..
Distributed Deadlock Detection

Two Local Wait-For Graphs

site $S_1$

site $S_2$
Deadlock Detection - Centralized Approach

- Each site keeps a local wait-for graph
  - The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
- A global wait-for graph is maintained in a single coordination process; this graph is the union of all local wait-for graphs
- There are three different options (points in time) when the wait-for graph may be constructed:
  1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
  2. Periodically, when a number of changes have occurred in a wait-for graph
  3. Whenever the coordinator needs to invoke the cycle-detection algorithm
- Option1: unnecessary rollbacks may occur as a result of false cycles
Local and Global Wait-For Graphs

site $S_1$

site $S_2$

coordinator
Fully Distributed Approach

- All controllers share equally the responsibility for detecting deadlock
- Every site constructs a wait-for graph that represents a part of the total graph
- We add one additional node $P_{ex}$ to each local wait-for graph
  - $P_i \rightarrow P_{ex}$ exists if $P_i$ is waiting for a data item at another site being held by any process
- If a local wait-for graph contains a cycle that does not involve node $P_{ex}$, then the system is in a deadlock state
- A cycle involving $P_{ex}$ implies the possibility of a deadlock
  - To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked
Augmented Local Wait-For Graphs

Site $S_1$

- $P_1$
- $P_2$
- $P_3$
- $P_5$
- $P_{ex}$

Site $S_2$

- $P_{ex}$
- $P_2$
- $P_3$
- $P_4$
Augmented Local Wait-For Graph in Site S2
Any Questions?

Hmm.
Acknowledgements

• “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne

• “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings

• “Modern Operating Systems” book and supplementary material by A. Tanenbaum

• R. Doursat and M. Yuksel from UNR