Operating Systems

Mutual Exclusion in Distributed Systems

Lecturer:
William Fornaciari
Politecnico di Milano
fornacia@elet.polimi.it
www.elet.polimi.it/~fornacia

© 2003 William Fornaciari

DISTRIBUTED MUTUAL EXCLUSION (DME)

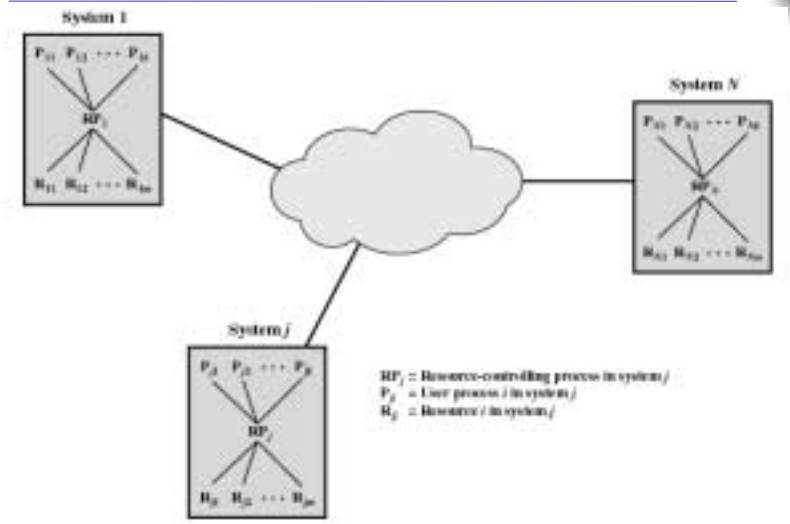
- Assumptions
  - The system consists of *n* processes; each process P_i resides at a different processor
  - Each process has a critical section that requires mutual exclusion
- Basic Requirement
  - If P_i is executing in its critical section, then no other process P_j is executing in its critical section
- The presented algorithms ensure the mutual exclusion execution of processes in their critical sections

DME: GENERAL REQUIREMENTS

- Mutual exclusion must be enforced: only one process at a time is allowed in its critical section
- A process that hales in its noncritical section must do so without interfering with other processes
- It must not be possible for a process requiring access to a critical section to be delayed indefinitely: No deadlock or starvation
- When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay
- No assumptions are made about relative process speeds or number of processors
- A process remains inside its critical section for a finite time only

 © 2003 William Fornaciari

THE MODEL FOR DME



 © 2003 William Fornaciari

DME : CENTRALIZED APPROACH (1)

- One of the processes in the system is chosen to coordinate the entry to the critical section (control access to shared objects)
- 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

 © 2003 William Fornaciari

DME : CENTRALIZED APPROACH(2)

- This scheme requires three messages per critical-section entry:
  - Request, reply, release
- Limits of the Centralized approach
  - if the coordinator process crashes, the system becomes useless
  - the several messages towards the coordinator process can create a bottleneck
- Benefits of the Centralized approach
  - Ensure mutual esclusion
  - Ordering of the access requests
  - No starvation
  - Use of only three messages

 © 2003 William Fornaciari

GENERAL REQUIREMENTS OF DISTRIBUTED ALGORITHMS

- All nodes have equal amount of information, on average
- Each node has only a partial picture of the total system and must make decisions based on this information
- All nodes bear equal responsibility for the final decision
- All nodes expend equal effort, on average, in effecting a final decision
- Failure of a node, in general, does not result in a total system collapse
- There exits no systemwide common clock with which to regulate the time of events

 © 2003 William Fornaciari

DME : TIME STAMPING (1)

- Since in a distributed system the clocks are not synchronized, we have to be able to order events
- Implementation of happened-before relation
  - Associate a timestamp with each system event. Require that for every pair of events *A* and *B*, if *A is happened before B*, then the timestamp of *A* is less than the timestamp of *B*
  - Within *each* process P_i a *logical clock*, LC_i is associated. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process.

 © 2003 William Fornaciari

DME : TIME STAMPING (2)

- A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock (when a message is received, the receiving system sets its counter to one more than the maximum of its current value and the incoming time-stamp (counter))
- If the timestamps of two events *A* and *B* are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering

 © 2003 William Fornaciari

DME : TIME STAMPING (3)

- For this method to work, each message is sent from one process to all other processes
  - ensures all sites have same ordering of messages
  - for mutual exclusion and deadlock all processes must be aware of the situation

 © 2003 William Fornaciari

DME : FULLY DISTRIBUTED (1)

1 When process P_i wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request* (P_i, *TS*) to all other processes in the system

2 When process P_j receives a *request* message, it may reply immediately or it may defer sending a reply back

3 When process P_i receives a *reply* message from all other processes in the system, it can enter its critical section

4 After exiting its critical section, the process sends *reply* messages to all its deferred requests

 © 2003 William Fornaciari

DME : FULLY DISTRIBUTED (2)

- 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

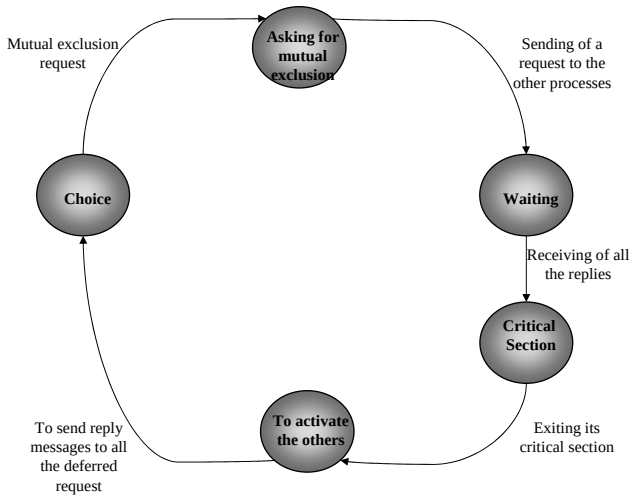
 © 2003 William Fornaciari

DME : FULLY DISTRIBUTED (3)

- Freedom from Deadlock is ensured
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first served order
- The number of messages per critical-section entry is $2 \times (n - 1) \rightarrow (n - 1)$ requests and $(n - 1)$ replies. This is the number of required messages per critical-section entry when processes act independently and concurrently

 © 2003 William Fornaciari

STATES OF THE FULLY DISTRIBUTED APPROACH



 © 2003 William Fornaciari

THREE 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
- Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section. This protocol is therefore suited for small, stable sets of cooperating processes

 © 2003 William Fornaciari

TOKEN-PASSING APPROACH (1)

- It exists a logical ring where each process has an assigned position and know who is next
- The token is an entity that at any time is held by one process
- The process holding the token may enter its critical section without asking permission. Only this which has got the token can enter its critical section
- When a process leaves its critical section, it passes the token to another process
- When a process wants enter its critical section and it hasn't got the token, it sends request messages to all the other processes, waiting since the arrival of the token

 © 2003 William Fornaciari

TOKEN-PASSING ALGORITHM (1)

- How to manage the use of the critical section

```
if not token_present then begin        clock := clock + 1;
                                       broadcast(Request, clock I);
                                       wait(access, token);
                                       token_present := True;
                              end
endif;
token_held := True:

<critical section>

token(i) := clock;
token_held := False;
for j := i + 1 to n, 1 to i - 1 do
  if (request(j) > token(J)) [Symbol]^token_present
    then begin
      token_present := False;
      send(access, token(j))
    end
  endif;
```

 © 2003 William Fornaciari

TOKEN-PASSING ALGORITHM (2)

- How to do when a process receives a request

```
when received (Request, k, j) do
  request(j) := max(request(j), k);
  if token_present[Symbol] ^not token_held then
    token(i) := clock;
    token_held := False;
    for j := i + 1 to n, 1 to i - 1 do
      if (request(j) > token(J)) [Symbol]^token_present
        then begin
          token_present := False;
          send(access, token(j))
        end
      endif;
  endif
enddo;

Notation :
send(j, access, token)          send message of type access, with token, by process j
broadcast(request, clock, i)    send message from process i of type request, with timestamp clock, to all
                                other processes
received(request, t, j)         receive message from process j of type request,with timestamp t
```

 © 2003 William Fornaciari

TOKEN-PASSING APPROACH (2)

- Troubles
  - Difficult detection/regeneration of lost token
  - Process crash
    - Difficult detection
    - Easy recover: if required ack from process when the token is received, dead processes are detected by neighbor which can skip it over (need to know the entire ring configuration)
- Benefits
  - Mutual esclusion
  - No starvation
  - At worst a process have to wait for every other process to enter and leave one critical region

 © 2003 William Fornaciari

DME: COMPARISON

Algorithm	Messages per entry/exit	Delay before entry (in msg times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n-1)	2 (n-1)	Crash of any process
Token ring	1 to ∞	0 to (n-1)	Lost token, process crash

- None is robust to system crashes

 © 2003 William Fornaciari

REFERENCES

- Stallings W. **Sistemi Operativi** Jackson Libri
- Silberschatz A. **Sistemi Operativi** Addison-Wesley
- Tanenbaum A. **I Moderni Sistemi Operativi** Jackson Libri
- Web Material (see links)

 © 2003 William Fornaciari