# CIS 505: Software Systems
# Lecture Note on Synchronization

Instructor: Insup Lee
Department of Computer and Information Science
University of Pennsylvania

CIS 505, Spring 2007

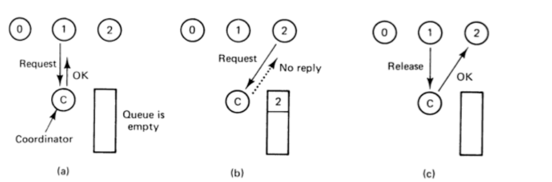
## Mutual Exclusion and Synchronization

- To solve synchronization problems in a distributed system, we need to provide distributed semaphores.
- Schemes for implementation :
  1. A Centralized Algorithm
  2. A Distributed Algorithm
  3. A Token Ring Algorithm



## A Centralized Algorithm

- Use a coordinator which enforces mutual exclusion.
- Two operations: request and release.
  - Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.
  - Process 2 then asks permission to enter the same critical region. The coordinator des not reply.
  - When process 1 exists the critical region, it tells the coordinator, which then replies to 2.






## A Centralized Algorithm (continued)

- Coordinator
```
loop
  receive(msg);
    case msg of
      REQUEST: if nobody in CS
               then reply GRANTED
               else queue the REQ;
                    reply DENIED
      RELEASE: if queue not empty then
                remove 1st on the queue
                reply GRANTED
    end case
end loop
```
- Client
```
send(REQUEST);
receive(msg);
if msg != GRANTED then receive(msg);
enter CS;
send(RELEASE)
```

## A Centralized Algorithm

- Algorithm properties
  - guarantees mutual exclusion
  - fair (if First Come First Served)
  - a single point of failure (Coordinator)
  - if no explicit DENIED message, then cannot distinguish permission denied from a dead coordinator



## A Decentralized Algorithm

Decision making is distributed across the entire system

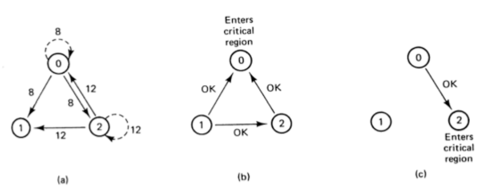
- Two processes want to enter the same critical region at the same moment.
- Both send request messages to all processes
- All events are time-stamped by the global ordering algorithm
- The process whose request event has smaller time-stamp wins
- Every process must respond to request messages



## A Decentralized Algorithm

Decision making is distributed across the entire system

- Two processes want to enter the same critical region at the same moment.
- Process 0 has the lowest timestamp, so it wins.
- When process 0 is done, it sends an OK also; so, 2 can now enter the critical region.






## Decentralized Algorithm (continued)

1 When a process P wants to enter its critical section, it generates a new time stamp, TS, and sends the msg request (P,TS) to all other processes in the system (recall algorithm for global ordering of events)

2 A process, which receives reply msgs from all other processes, can enter its critical section.

3 When a process receives a request message,
   (A) if it is in CS, defers its answer;
   (B) if it does not want to enter its CS, reply immediately;
   (C) if it also wants to enter its CS, it maintains a queue of requests (including its own request) and sends a reply to the request with the minimum time-stamp

## Correctness

**Theorem.** The Algorithm achieves mutual exclusion.

**Proof:**

By contradiction.

Suppose two processes Pi and Pj are in CS concurrently. WLOG, assume that Pi's request has earlier timestamp than Pj. That is, Pi received Pj's request after Pi made its own request.

Thus, if Pj can concurrently execute the CS with Pi, then Pi must returned a REPLY to Pj before Pi exited the CS.

But, this is impossible since Pj has a later timestamp than Pi.



## Properties

1. mutual exclusion is guaranteed
2. deadlock free
3. no starvation, assuming total ordering on msgs
4. 2(N-1) msgs: (N-1) request and (N-1) reply msgs
5. N points of failure (i.e., each process becomes a point of failure) can use explicit ack and timeout to detect failed processes
6. each process needs to maintain group membership; (i.e., IDs of all active processes) non-trivial for large and/or dynamically changing memberships
7. N bottlenecks since all processes involved in all decisions
8. Could use majority votes to improve the performance



## A Token Passing Algorithm

- A token is circulated in a logical ring.
- A process enters its CS if it has the token.
- Issues:
  - If the token is lost, it needs to be regenerated.
  - Detection of the lost token is difficult since there is no bound on how long a process should wait for the token.
  - If a process can fail, it needs to be detected and then by-passed.
  - When nobody wants to enter, processes keep on exchanging messages to circulate the token



## Comparison

- A comparison of three mutual exclusion algorithms

| Algorithm | Messages per entry/exit | Delay before entry (in message 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 |

## Leader Election

- In many distributed applications, particularly the centralized solutions, some process needs to be declared the central coordinator
- Electing the leader also may be necessary when the central coordinator crashes
- Election algorithms allow processes to elect a unique leader in a decentralized manner



## Bully Algorithm

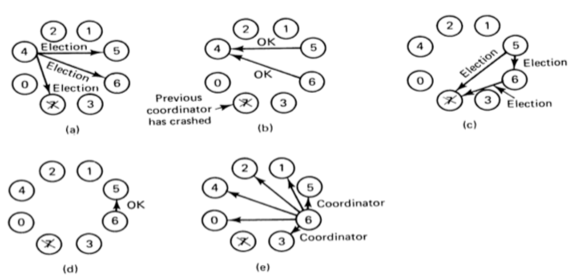
Goal: Determine who is the active process with max ID

- Suppose a process P detects a failure of the current leader
  - P sends an "election" message to all processes with higher ID
  - If nobody responds within interval T, sends "coordinator" message to all processes with lower IDs
  - If someone responds with "OK" message, P waits for a "coordinator" message (if not received, restart the algorithm)
- If P receives a message "election" from a process with lower ID, responds with "OK" message, and starts its own leader election algorithm (as in step 1)
- If P receives "coordinator" message, record the ID of the leader



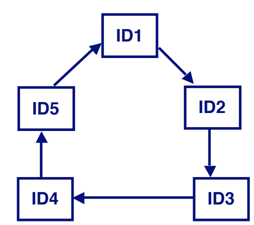
## Bully Algorithm

(a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.



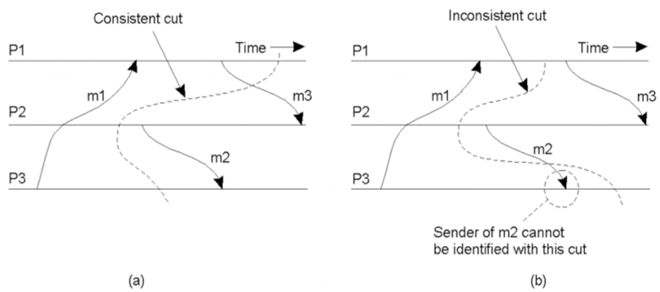



## Leader Election in a Ring




- Each process has unique ID; can receive messages from left, and send messages to the right
- Goal: agree on who is the leader (initially everyone knows only its own ID)
- Idea:
  - initially send your own ID to the right. When you receive an ID from left, if it is higher than what you have seen so far, send it to right.
  - If your own ID is received from left, you have the highest ID and are the leader

## Global State




a) A consistent cut
b) An inconsistent cut



## Distributed Deadlock

- A deadlock occurs when a set of processes in a system is blocked waiting for requests that can never be satisfied.
- Approaches:
  - Detection (& Recovery)
  - Prevention
  - Avoidance - not practical in distributed setting
- Difficulties:
  - resource allocation information is distributed
  - gathering information requires messages. Since messages have non-zero delays, it is difficult to have an accurate and current view of resource allocation.



## Deadlock Detection Recall

- Suppose the following information is available, for each process:
  - the resources it currently holds, and
  - the request that it is waiting for.
- Then, one can check if the current system state is deadlocked, or not.
- In single-processor systems, OS can maintain this information, and periodically execute deadlock detection algorithm
- What to do if a deadlock is detected?
  - Kill a process involved in the deadlocked set
  - Inform the users, etc.



## Wait For Graph (WFG)

- **Definition.** A resource graph is a bipartite directed graph (N,E), where
  - N = P U R,
  - $P = \{p_1, \dots p_n\}$ , $R = \{r_1, \dots r_n\}$
  - $(r_1, \dots r_n)$ available unit vector,
  - An edge $(p_i, r_j)$ a request edge, and
  - An edge $(r_i, p_j)$ an allocation edge.
- **Definition**: Wait For Graph (WFG) is a directed graph, where nodes are processes and a directed edge from $P \rightarrow Q$ represents that $P$ is blocked waiting for $Q$ to release a resource.
- So, there is an edge from process $P$ to process $Q$ if $P$ needs a resource currently held by $Q$.

## Definitions

- **Def:** A node Y is reachable from a node X, X ⇒ Y, if there is a path (i.e., a sequence of directed edges) from node X to node Y.
- **Def:** A cycle in a graph is a path that starts and ends on the same node. If a set C of nodes is a cycle, then for all X in C : X ⇒ X
- **Def:** A knot K in a graph is a non-empty set of nodes such that, for each X in K, all nodes in K and only the nodes in K are reachable from X. That is,
  - (for every X for every Y in K, X ⇒ Y) and
  - (for every X in K, there exists Z s.t. X ⇒ Z implies Z is in K)

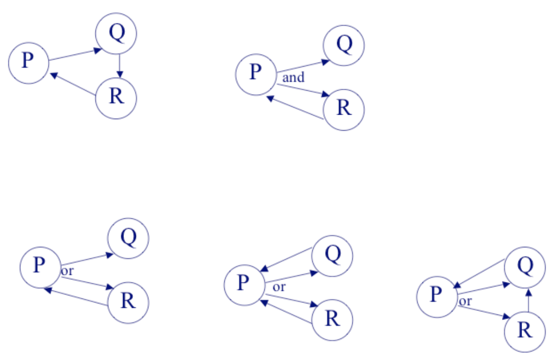


## Sufficient Conditions for Deadlock

- **Resource Model**
  1. reusable resource
  2. exclusive access
- **Three Request Models**
  1. Single-unit request model:
     - a cycle in WFG
  2. AND request model: simultaneous requests
     - blocked until all of them granted
     - a cycle in WFG
     - a process can be in more than one cycle
  3. OR request model: any one, e.g., reading a replicated data object
     - a cycle in WFG not a sufficient condition (but necessary)
     - a knot in WFG is a sufficient condition (but not necessary)



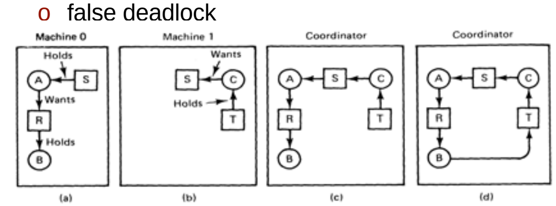
## Examples






## Deadlock Detection Algorithms

- Centralized Deadlock Detection
  - o false deadlock




(a) Initial resource graph for machine 0.

(b) Initial resource graph for machine 1.

(c) The coordinator's view of the world.

(d) The situation after the delayed message.

## Wait-for Graph for Detection

- Assume only one instance of each resource
- Nodes are processes
  - Recall Resource Allocation Graph: it had nodes for resources as well as processes (basically same idea)
- Edges represent waiting: If P is waiting to acquire a resource that is currently held by Q, then there is an edge from P to Q
- A deadlock exists if and only if the global wait-for graph has a cycle
- Each process maintains a local wait-for graph based on the information it has
- Global wait-for graph can be obtained by the union of the edges in all the local copies



## Distributed Cycle Detection

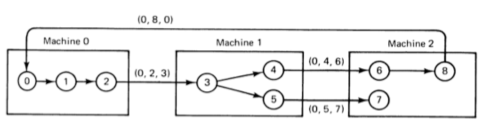
Basic Idea:

- Each site looks for potential cycles
- Suppose site S1 has processes P1, P2, P3, P4.
- S1 knows that P7 (on a different site) is waiting for P1, P1 is waiting for P4, P4 is waiting for P2, and P2 is waiting for P9 (on a different site S3)
- This can be a potential cycle
- S1 sends a message to S3 giving the chain P7, P1, P4, P2, P9
- Site S3 knows the local dependencies, and can extend the chain, and pass it on to a different site
- Eventually, some site will detect a deadlock, or will stop forwarding the chain



## Deadlock Detection Algorithms

- Distributed Deadlock Detection: An Edge-Chasing Algorithm




Chandy, Misra, and Haas distributed deadlock detection algorithm.



## Deadlock Prevention

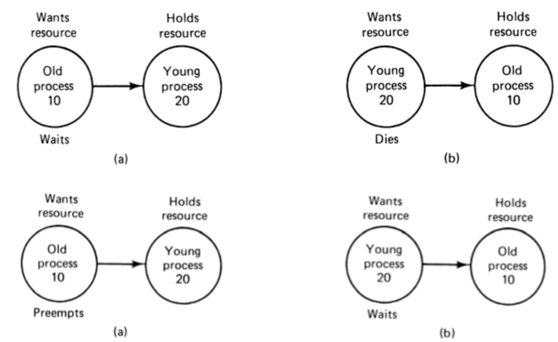
- Hierarchical ordering of resources avoids cycles
- Time-stamp ordering approach:
  Prevent the circular waiting condition by preempting resources if necessary.
  - The basic idea is to assign a unique priority to each process and use these priorities to decide whether process *P* should wait for process *Q*.
  - Let *P* wait for *Q* if *P* has a higher priority than *Q*; Otherwise, *P* is rolled back.
  - This prevents deadlocks since for every edge (*P*,*Q*) in the wait-for graph, *P* has a higher priority than *Q*.
    Thus, a cycle cannot exist.

## Two commonly used schemes

- Wait-Die (WD): Non-preemptive
  - When *P* requests a resource currently held by *Q* , *P* is allowed to wait only if it is older than *Q*.
  - Otherwise, *P* is rolled back (i.e., dies).
- Wound-Wait (WW): Preemptive
  - When *P* requests a resource currently held by *Q* , *P* is allowed to wait only if *P* is younger than *Q*.
  - Otherwise, *Q* is rolled back (releasing its resource). That is, *P* wounds *Q*.
- Note:
  - Both favor old jobs (1) to avoid starvation, and (2) since older jobs might have done more work, expensive to roll back.
  - Unnecessary rollbacks may occur.



## WD versus WW






## Sample Scenario

- Processes P, Q, R are executing at 3 distributed sites
- Suppose the time-stamps assigned to them (at the time of their creation) are 5, 10, 20, respectively
- Q acquires a shared resource
- Later, R requests the same resource (held by Q)
  - WD would roll back R
  - WW would make R wait
- Later, P requests the same resource (held by Q)
  - WD would make P wait
  - WW would roll back Q, and give the resource to P



## Example

Wait-Die (WD):

(1) P requests the resource held by Q. P waits.

(2) R requests the resource held by Q. R rolls back.

Wound-Wait (WW):

(1) P requests the resource held by Q. P gets the resource and Q is rolled back.

(2) R requests the resource held by Q. R waits.

## Differences between WD and WW

- In WD, older waits for younger to release resources.
- In WW, older never waits for younger.
- WD has more roll back than WW.
  In WD, *R* requests and dies because *Q* is older in the above example. If *R* restarts and again asks for the same resource, it rolls back again if *Q* is still using the resource.
  However, in WW, *Q* is rolled back by *P*. If it requests the resource again, it waits for *P* to release it.
- When there are more than one process waiting for a resource held by *P*, which process should be given the resource when *P* finishes?
  In WD, the youngest among waiting ones. In WW, the oldest.