CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania
Fall 2003
Lecture Note: Distributed Systems

Introduction to Distributed Systems

- Why do we develop distributed systems?
  - availability of powerful yet cheap microprocessors (PCs, workstations), continuing advances in communication technology,

- **What is a distributed system?**
  - A distributed system is a collection of independent computers that appear to the users of the system as a single system.

- Examples:
  - Network of workstations
  - Distributed manufacturing system (e.g., automated assembly line)
  - Network of branch office computers

Distributed Systems

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

Comparison of three kinds of multiple CPU systems

Advantages of Distributed Systems over Centralized Systems

- **Economics**: a collection of microprocessors offer a better price/performance than mainframes. Low price/performance ratio: cost effective way to increase computing power.
- **Speed**: a distributed system may have more total computing power than a mainframe. Ex. 10,000 CPU chips, each running at 50 MIPS. Not possible to build 500,000 MIPS single processor since it would require 0.002 nsec instruction cycle. Enhanced performance through load distributing.
- **Inherent distribution**: Some applications are inherently distributed. Ex. a supermarket chain.
- **Reliability**: If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.
- **Incremental growth**: Computing power can be added in small increments. Modular expandability
- **Another deriving force**: the existence of large number of personal computers, the need for people to collaborate and share information.

Advantages of Distributed Systems over Independent PCs

- Data sharing: allow many users to access to a common data base
- Resource Sharing: expensive peripherals like color printers
- Communication: enhance human-to-human communication, e.g., email, chat
- Flexibility: spread the workload over the available machines

Disadvantages of Distributed Systems

- Software: difficult to develop software for distributed systems
- Network: saturation, lossy transmissions
- Security: easy access also applies to secrete data

Software Concepts

- **Software more important for users**
- **Two types:**
    1. **Network Operating Systems**
    2. **(True) Distributed Systems**

Network Operating Systems

- loosely-coupled software on loosely-coupled hardware
- A network of workstations connected by LAN
- each machine has a high degree of autonomy
  - rlogin machine
  - rcp machine1:file1 machine2:file2
- Files servers: client and server model
- Clients mount directories on file servers
- Best known network OS:
  - Sun's NFS (network file servers) for shared file systems (Fig. 9-11)
- a few system-wide requirements: format and meaning of all the messages exchanged

NFS (Network File System)

- NFS Architecture
  - Server exports directories
  - Clients mount exported directories
- NSF Protocols
  - For handling mounting
  - For read/write: no open/close, stateless
- NSF Implementation

(True) Distributed Systems

- tightly-coupled software on loosely-coupled hardware
- provide a single-system image or a virtual uniprocessor
- a single, global interprocess communication mechanism, process management, file system; the same system call interface everywhere
- <u>Ideal definition</u>:

  " A distributed system runs on a collection of computers that do not have shared memory, yet looks like a single computer to its users."

Design Issues of Distributed Systems

- Transparency
- Flexibility
- Reliability
- Performance
- Scalability

1. Transparency

- How to achieve the single-system image, i.e., how to make a collection of computers appear as a single computer.
- Hiding all the distribution from the users as well as the application programs can be achieved at two levels:
  1) hide the distribution from users
  2) at a lower level, make the system look transparent to programs.

  1) and 2) requires uniform interfaces such as access to files, communication.

2. Flexibility

- Make it easier to change
- Monolithic Kernel: systems calls are trapped and executed by the kernel. All system calls are served by the kernel, e.g., UNIX.
- Microkernel: provides minimal services.
    - IPC
    - some memory management
    - some low-level process management and scheduling
    - low-level i/o (E.g., Mach can support multiple file systems, multiple system interfaces.)

3. Reliability

- Distributed system should be more reliable than single system. Example: 3 machines with .95 probability of being up. 1-.05**3 probability of being up.
  - Availability: fraction of time the system is usable. Redundancy improves it.
  - Need to maintain consistency
  - Need to be secure
  - Fault tolerance: need to mask failures, recover from errors.

4. Performance

- Without gain on this, why bother with distributed systems.
- Performance loss due to communication delays:
  - fine-grain parallelism: high degree of interaction
  - coarse-grain parallelism
- Performance loss due to making the system fault tolerant.

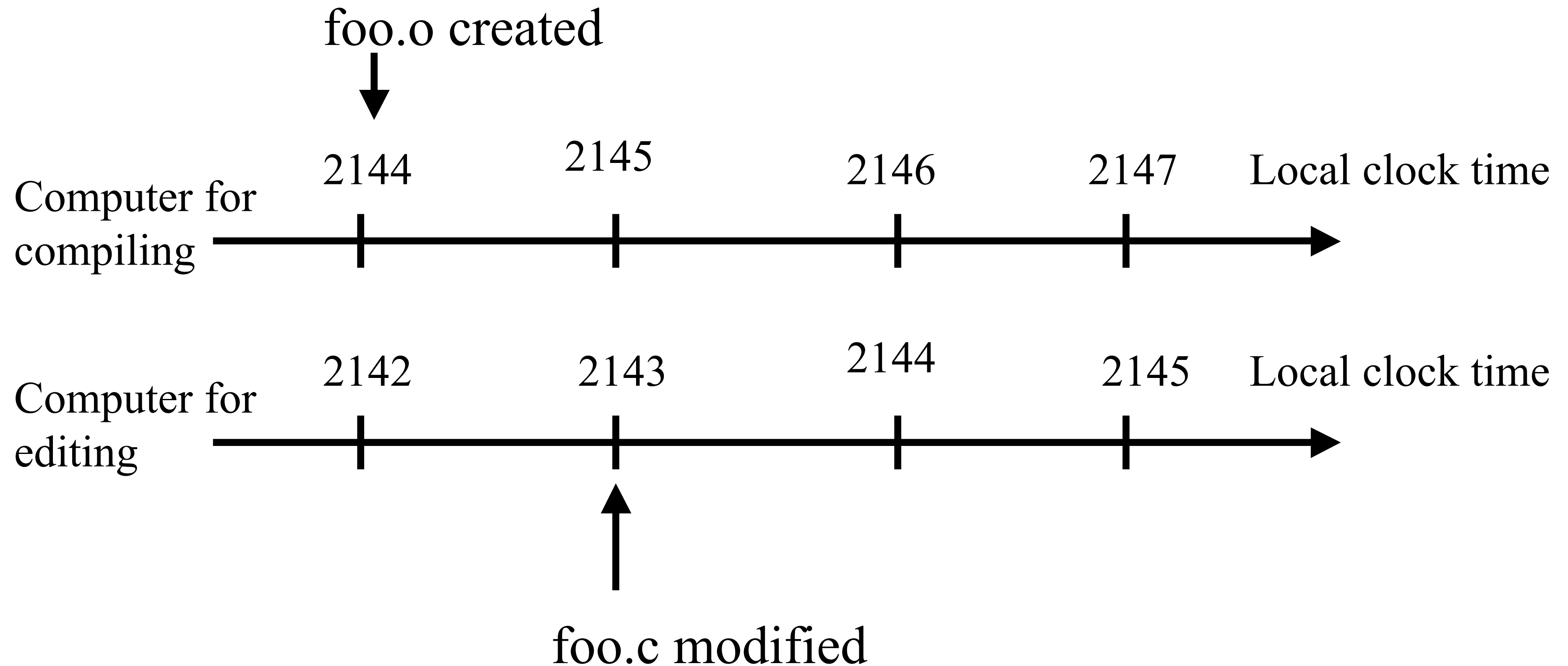
5. Scalability

- Systems grow with time or become obsolete.
- Techniques that require resources linearly in terms of the size of the system are not scalable. (e.g., broadcast based query won't work for large distributed systems.)
- Examples of bottlenecks
  - o Centralized components: a single mail server
  - o Centralized tables: a single URL address book
  - o Centralized algorithms: routing based on complete information

Distributed Coordination

- Communication between processes in a distributed system can have unpredictable delays, processes can fail, messages may be lost
- Synchronization in distributed systems is harder than in centralized systems because the need for distributed algorithms.
- Properties of distributed algorithms:
  1. The relevant information is scattered among multiple machines.
  2. Processes make decisions based only on locally available information.
  3. A single point of failure in the system should be avoided.
  4. No common clock or other precise global time source exists.
- Challenge: How to design schemes so that multiple systems can coordinate/synchronize to solve problems efficiently?

Why need to synchronize clocks?



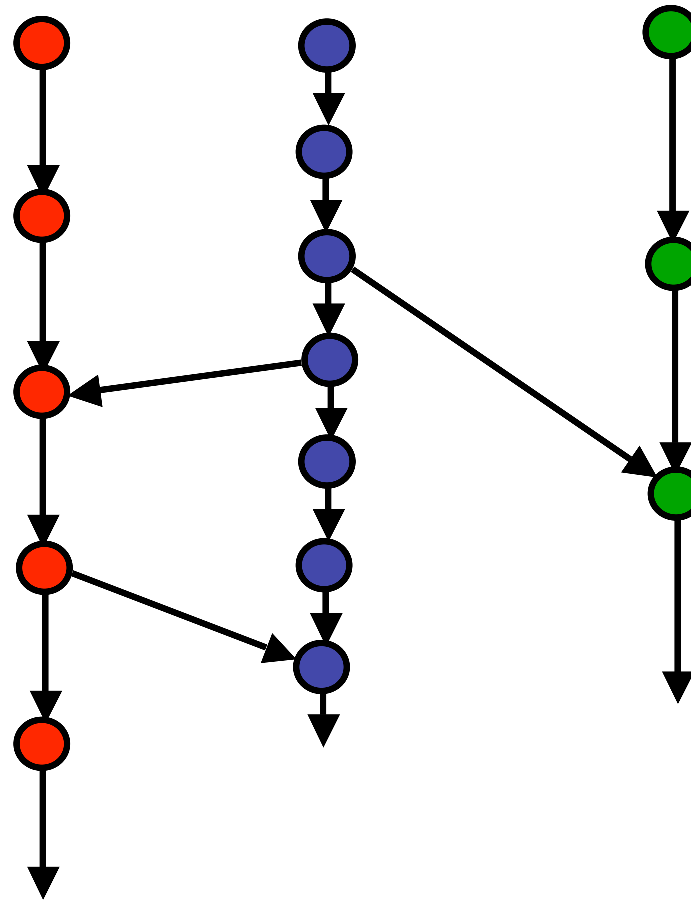
Logical and physical clocks

- How a computer timer works?
  - A counter register and a holding register.
  - The counter is decremented by a quartz crystals oscillator.
    When it reaches zero, an interrupted is generated and the counter is reloaded from the holding register.
  - E.g, interrupt 60 times per second.
- clock skew problem
- logical clocks -- to provide consistent event ordering
- physical clocks -- clocks whose values must not deviate from the real time by more than a certain amount.

Event Ordering

- Since there is no common memory or clock, it is sometimes impossible to say which of two events occurred first.
- The *happened-before* relation is a partial ordering of events in distributed systems such that
  1. If A and B are events in the same process, and A was executed before B, then $A \Rightarrow B$.
  2. If A is the event of sending a message by one process and B is the event of receiving that by another process, then $A \Rightarrow B$.
  3. If $A \Rightarrow B$ and $B \Rightarrow C$, then $A \Rightarrow C$.
- If two events A and B are not related by the $\Rightarrow$ relation, then they are executed concurrently (no causal relationship)
- To obtain a global ordering of all the events, each event can be *time stamped* satisfying the requirement: for every pair of events A and B, if $A \Rightarrow B$ then the time stamp of A is less than the time stamp of B. (Note that the converse need not be true.)

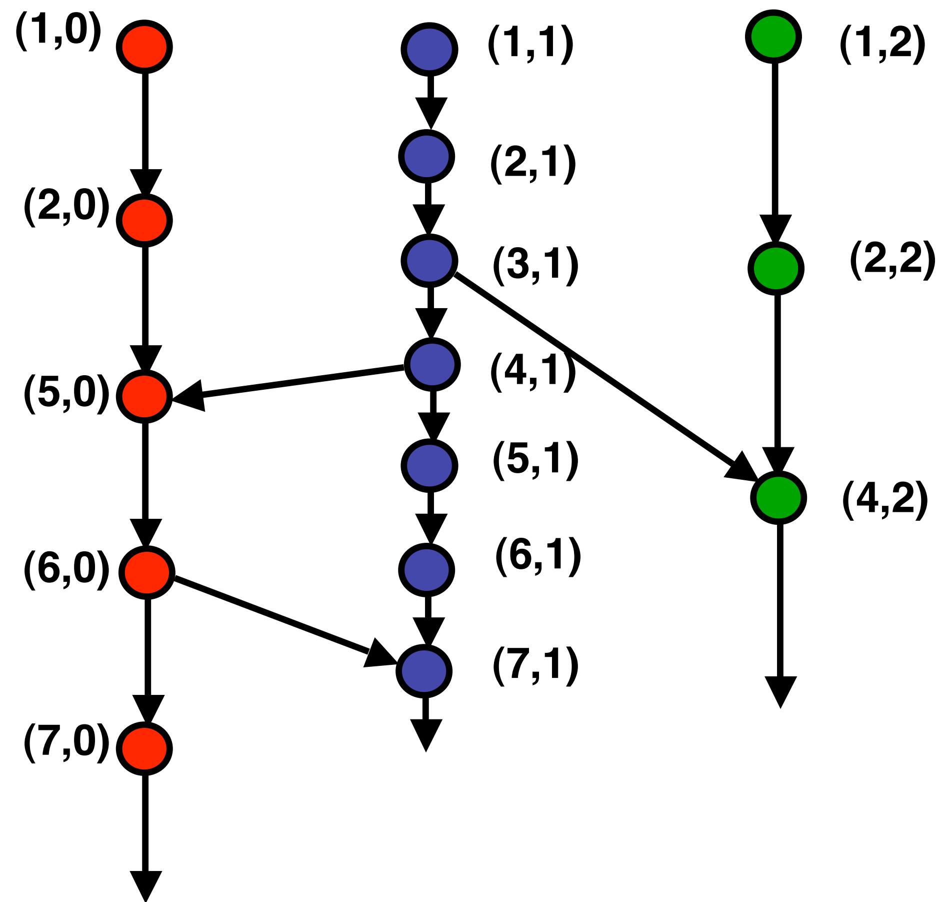
Example of Event Ordering



Global ordering

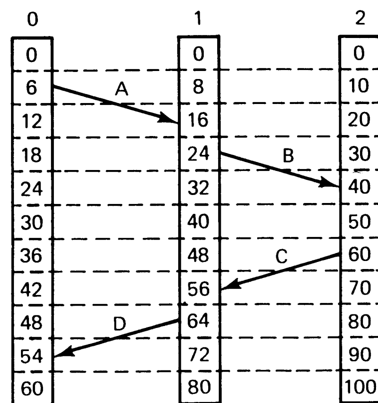
- How do we enforce the global ordering requirement in a distributed environment (without a common clock)?
  1. For each process P_i , a logical clock LC_i assign a unique value to every event in that process.
  2. If process P_i receives a message (event B) with time stamp t and $LC_i(B) < t$, then advance its clock so that $LC_i(B) = t+1$.
  3. Use processor ids to break ties to create a total ordering.

Example of Global Timestamps

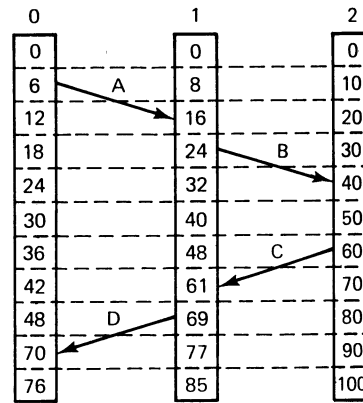


Example: Lamport's Algorithm

- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's Algorithm corrects the clock.



(a)

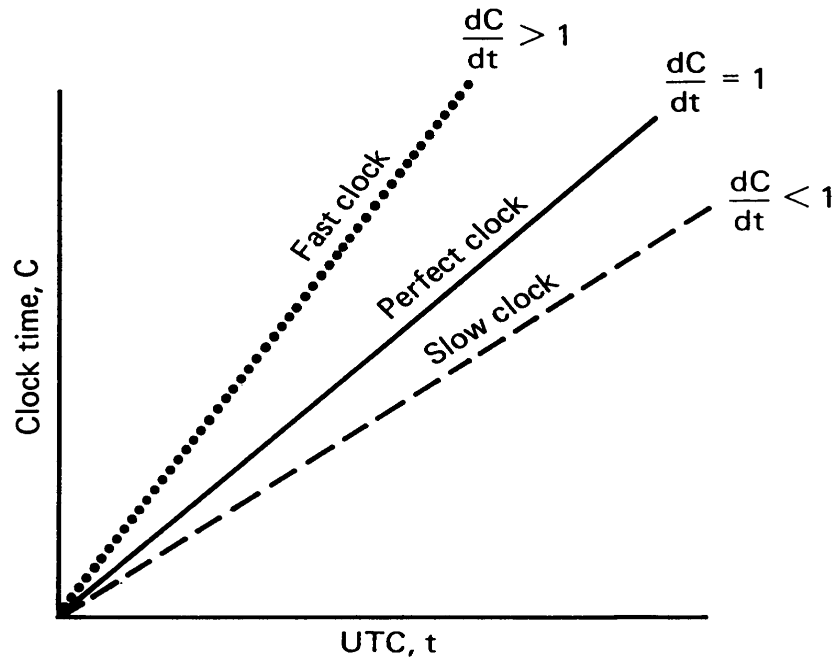


(b)

- Note: ts(A) < ts(B) does not imply A happened before B.

Physical clock synchronization algorithms

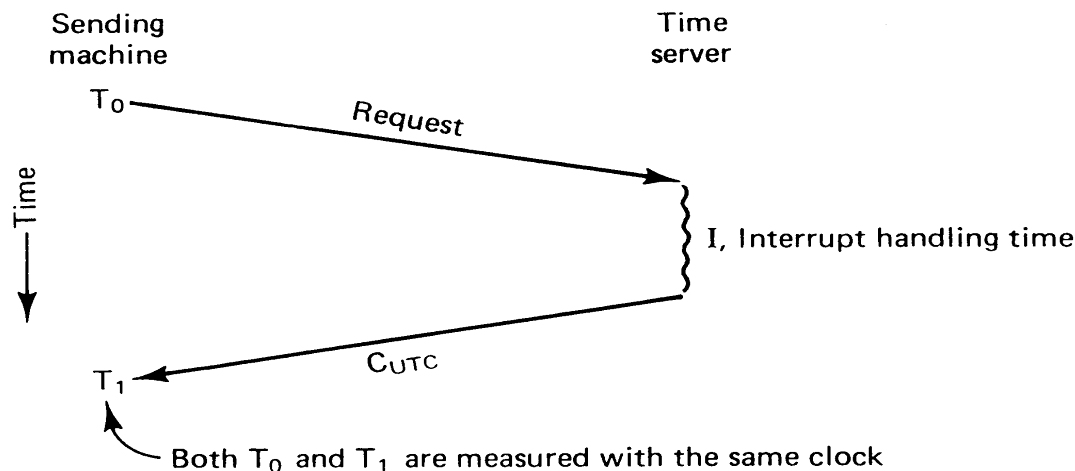
- Maximum drift rate
  - One can determine how often they should be synchronized



Not all clock's tick precisely at the current rate.

Physical clock synchronization algorithms

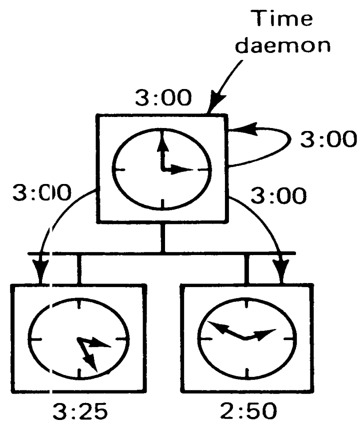
- Cristian's algorithm
  - need to change time gradually
  - need to consider msg delays, subtract (T1 - T0 - I)/2



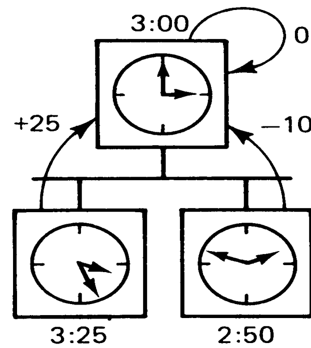
- Getting the current time from a time server

Physical clock synchronization algorithms

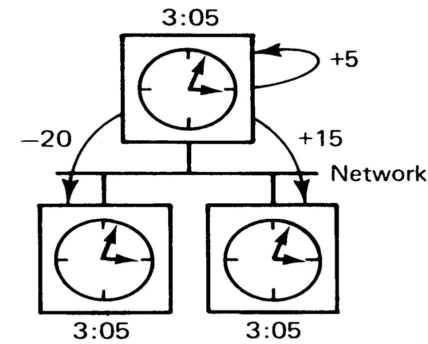
- The Berkeley algorithm
- Averaging algorithm
  - The time daemon asks all the other machines for their clock values.
  - The machines answer.
  - The Time daemon tells everyone how to adjust their clock.



(a)



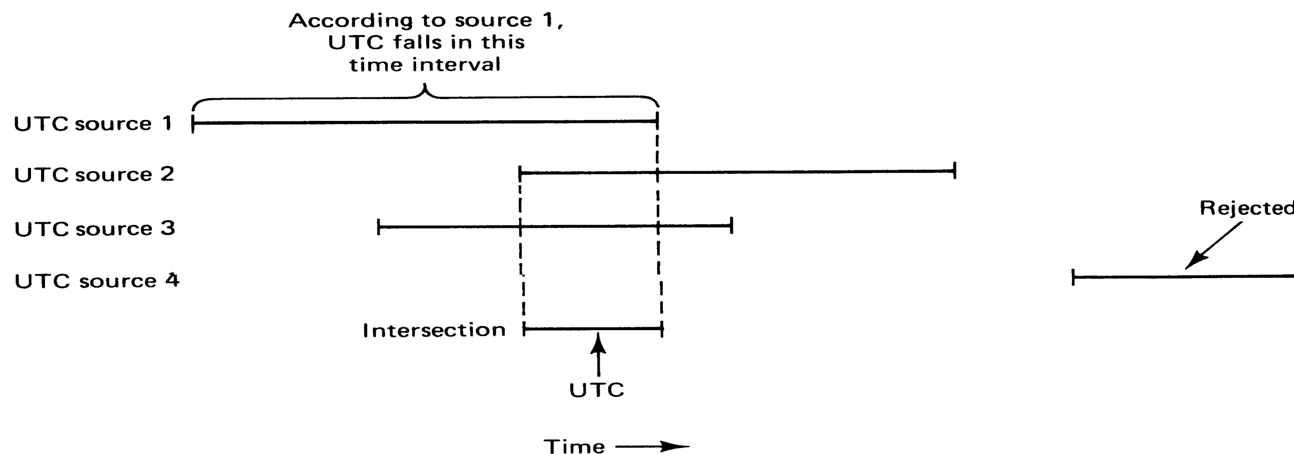
(b)



(c)

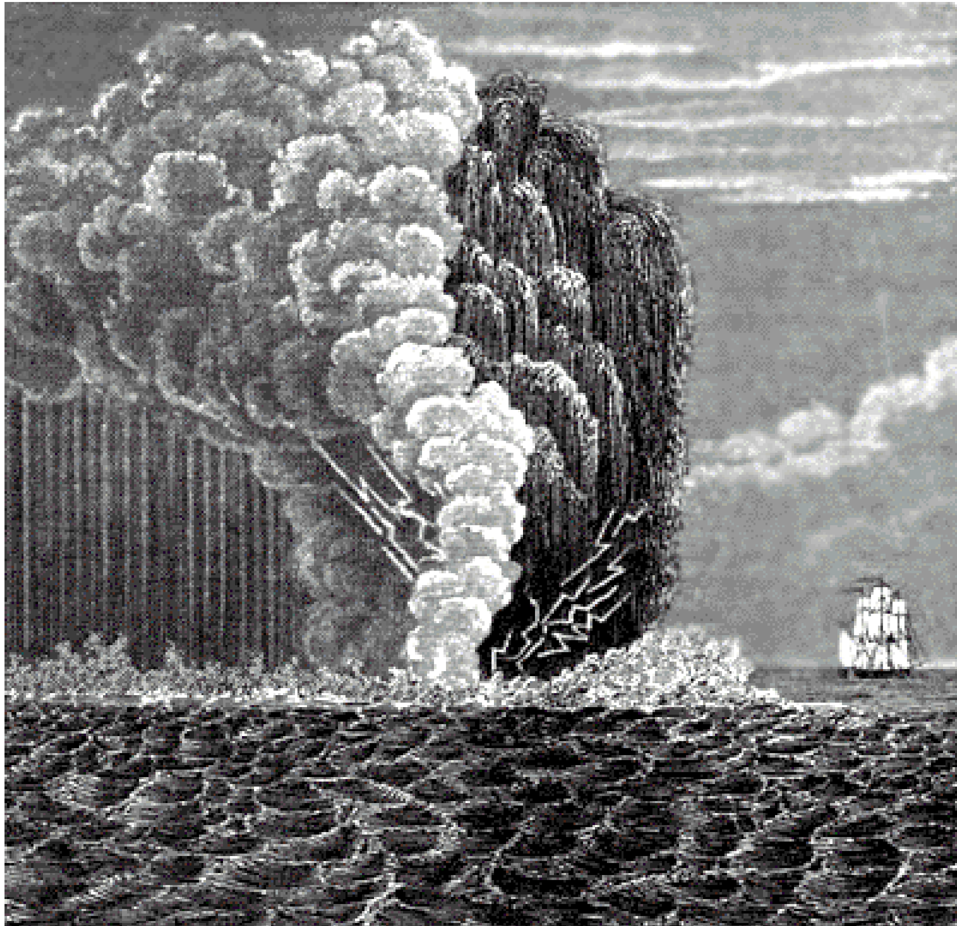
Physical clock synchronization algorithms

- Multiple external time sources
  - UTC (Universal Coordinated Time)
  - NIST broadcasts WWV signal at every UTC sec from CO.



- Computing UTC from multiple time sources, each of which gives a time interval in which UTC falls.

Unreliable communication



Reaching Agreement

- How can processes reach consensus in a distributed system
  - Messages can be delayed
  - Messages can be lost
  - Processes can fail (or even malignant)
  - Messages can be corrupted
- Each process starts with a bit (0 or 1) and Non-faulty processes should eventually agree on common value
  - No solution is possible
  - Note: solutions such as computing majority do not work. Why?
- Two generals problem (unreliable communications)
- Byzantine generals problem (faulty processes)

Two generals' problem

- Two generals on opposite sides of a valley have to agree on whether to attack or not (at a pre-agreed time)
- Goal: Each must be sure that the other one has made the same decision
- Communicate by sending messenger who may get captured
- Can never be sure whether the last messenger reached the other side (every message needs an ack), so no perfect solution
- Impossibility of consensus is as fundamental as undecidability of the halting problem !
- In practice: probability of losing a repeatedly sent message decreases (so agreement with high probability possible)

Impossibility Proof

Theorem. If any message can be lost, it is not possible for two processes to agree on non-trivial outcome using only messages for communication.

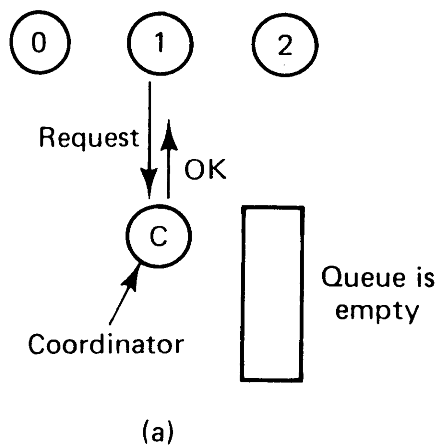
Proof. Suppose it is possible. Let m[1],…,m[k] be a finite sequence of messages that allowed them to decide. Furthermore, let's assume that it is a minimal sequence, that is, it has the least number of messages among all such sequences. However, since any message can be lost, the last message m[k] could have been lost. So, the sender of m[k] must be able to decide without having to send it (since the sender knows that it may not be delivered) and the receiver of m[k] must be able to decide without receiving it. That is, m[k] is not necessary for reaching agreement. That is, m[1],…,m[k-1] should have been enough for the agreement. This is a contradiction to that the sequence m[1],…,m[k] was minimum.

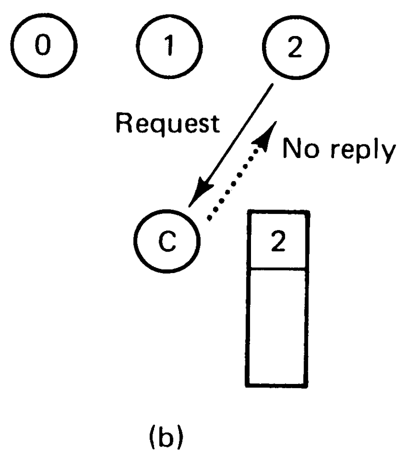
Mutual Exclusion and Synchronization

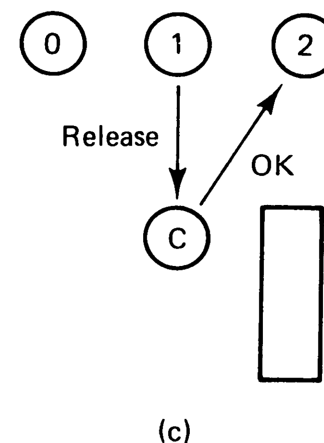
- To solve synchronization problems in a distributed system, we need to provide distributed semaphores.
- Schemes for implementation :
  1. A Centralized Algorithm
  2. A Distributed Algorithm
  3. A Token Ring Algorithm

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 (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

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

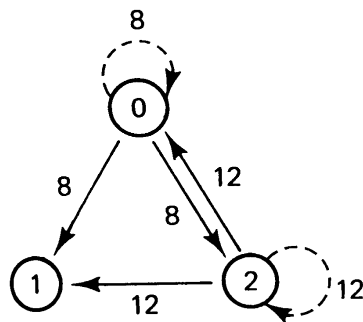
A Decentralized Algorithm

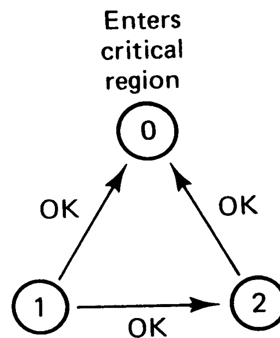
- Decision making is distributed across the entire system

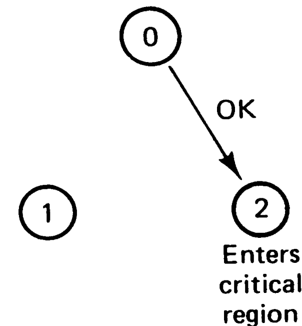
a) Two processes want to enter the same critical region at the same moment.

b) Process 0 has the lowest timestamp, so it wins.

c) 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 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, Pj can concurrently execute the CS with Pi only if Pi returns a REPLY to Pj before Pi exits 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. may 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

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

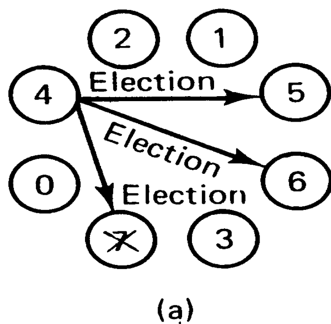
Bully Algorithm

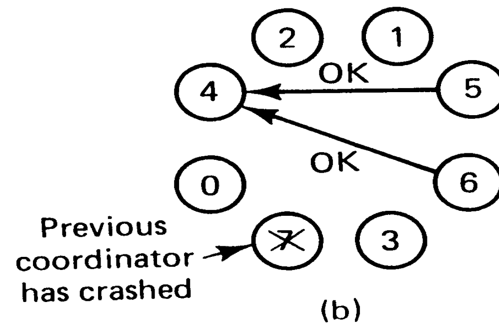
Goal: Figure out the active process with max ID

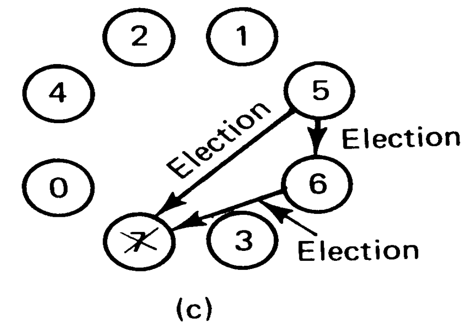
1. Suppose a process P detects a failure of current leader
   - P sends an "election" message to all processes with higher ID
   - If nobody responds within interval T, sends "coordinator" message to all processes with lower IDs
   - If someone responds with "OK" message, P waits for a "coordinator" message (if not received, restart the algorithm)
2. 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)
3. If P receives "coordinator" message, record the ID of leader

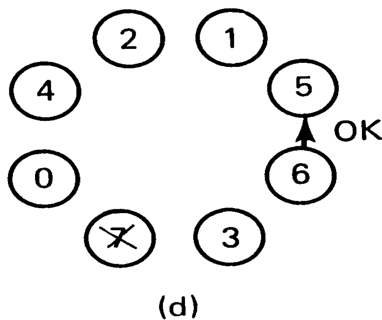
Bully Algorithm

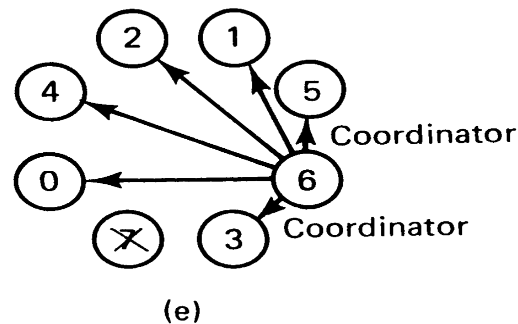
(a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.



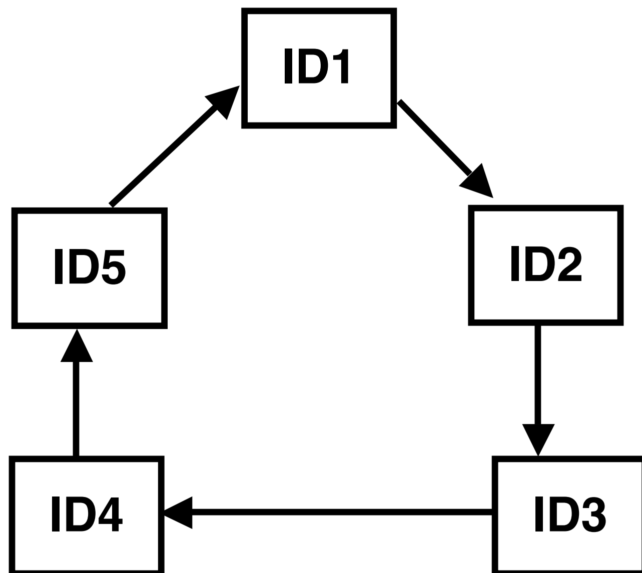








Leader Election in a Ring



- 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

Distributed Deadlock

- A deadlock occurs when a set of processes in a system are 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 following information is available:
  - For each process, the resources it currently holds
  - For each process, 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

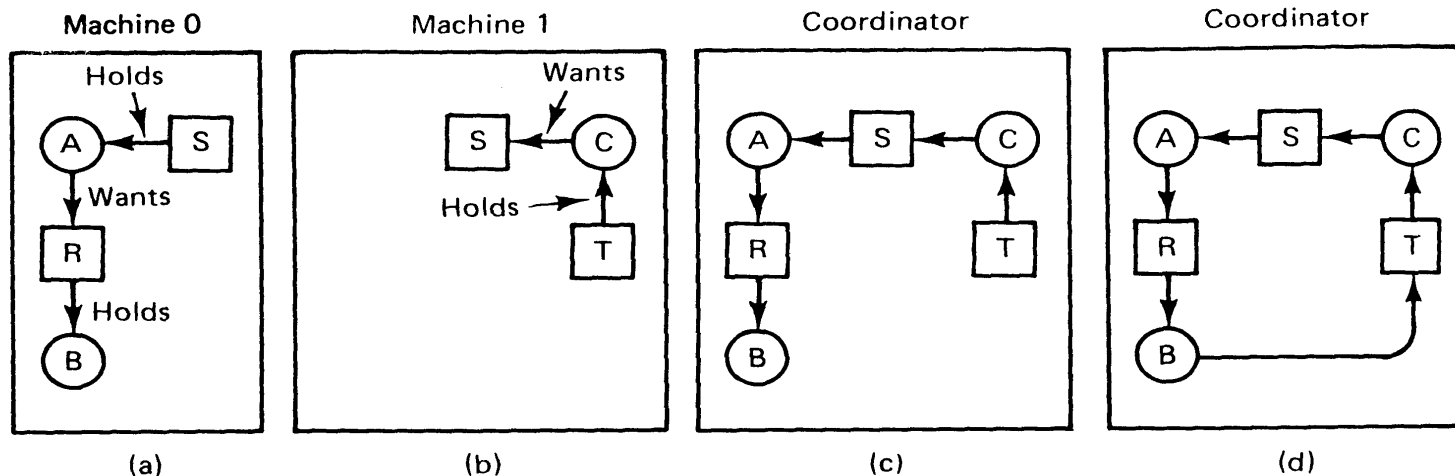
- **Def:** A node Y is reachable from a node X, $X \Rightarrow Y$, if there is a path (i.e., a sequence of directed edges) from node X to node Y.
- **Def:** A cycle in a graph is a path that starts and ends on the same node. If a set C of nodes is a cycle, then for all X in C : $X \Rightarrow X$
- **Def:** 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, $X \Rightarrow Y$) and
  - (for every X in K, there exists Z s.t. $X \Rightarrow Z$ 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)

Deadlock Detection Algorithms

- Centralized Deadlock Detection
  - 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

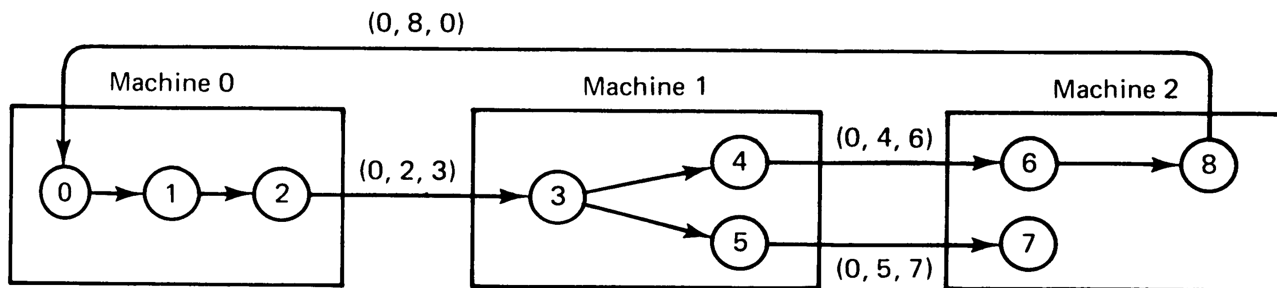
- Assume only one instance of each resource
- Nodes are processes
  - Recall Resource Allocation Graph: it had nodes for resources as well as processes (basically same idea)
- Edges represent waiting: If P is waiting to acquire a resource that is currently held by Q, then there is an edge from P to Q
- 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

- 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).

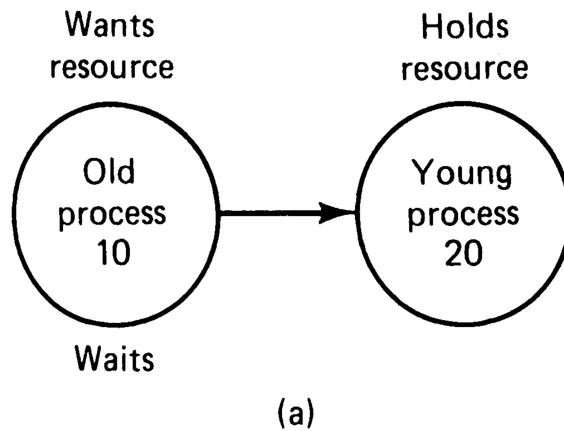
- Wound-Wait (WW): Preemptive
  When *P* requests a resource currently held by *Q* , *P* is allowed to wait only if *P* is younger than *Q*. Otherwise, *Q* is rolled back (releasing its resource). That is, *P* wounds *Q*.

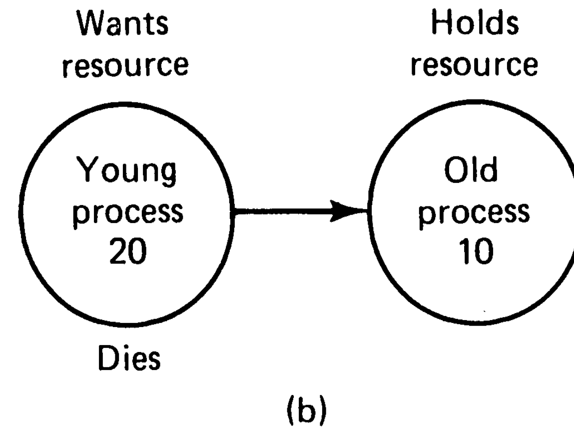
- Note:
  - Both favor old jobs (1) to avoid starvation, and (2) since older jobs might have done more work, expensive to roll back.
  - Unnecessary rollbacks may occur.

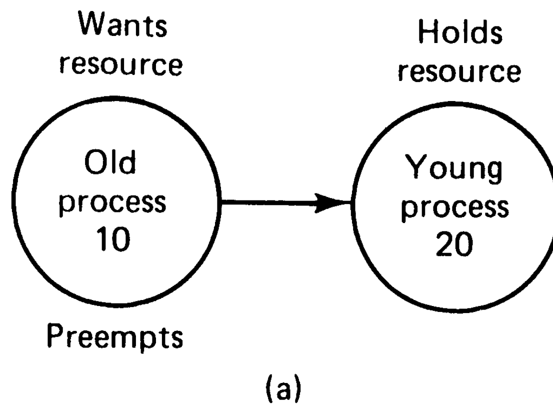
Sample Scenario

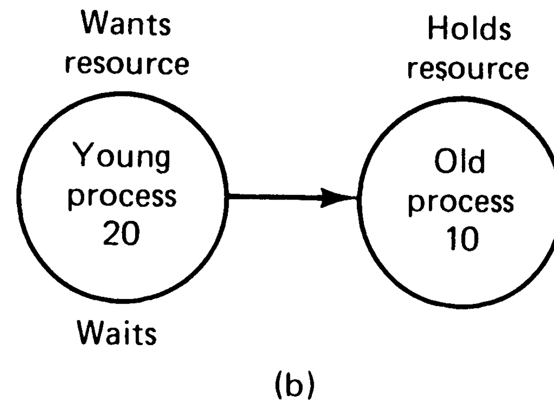
- Processes P, Q, R are executing at 3 distributed sites
- Suppose the time-stamps assigned to them (at the time of their creation) are 5, 10, 20, resp
- Q acquires a shared resource
- Later, R requests the same resource
  - WD would roll back R
  - WW would make R wait
- Later, P requests the same resource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

WD versus WW









Example

- Let P1 (5), P2 (10), P3 (15), and P2 has a resource.

Wait-Die (WD):

(1) P1 requests the resource held by P2. P1 waits.

(2) P3 requests the resource held by P2. P3 rolls back.

Wound-Wait (WW):

(1) P1 requests the resource held by P2. P1 gets the resource and P2 is rolled back.

(2) P3 requests the resource held by P2. P3 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, P_3 requests and dies because P_2 is older in the above example. If P_3 restarts and again asks for the same resource, it rolls back again if P_2 is still using the resource.
  However, in WW, P_2 is rolled back by P_1. If it requests the resource again, it waits for P_1 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.

Layers of distributed systems

- Computer networks
  - Local area networks such as Ethernet
  - Wide area networks such as Internet
- Network services
  - Connection-oriented services
  - Connectionless services
  - Datagrams
- Network protocols
  - Internet Protocol (IP)
  - Transmission Control Protocol (TCP)
- Middleware

Middleware for Distributed Systems

- Middleware is a layer of software between applications and OS that gives a uniform interface
- Central to developing distributed applications
- Different types
  - Document based (world-wide web)
  - File-system based (e.g., NFS)
  - Shared object-based (CORBA)
  - Coordination based (Linda, Publish-subscribe, Jini)

Summary

- Distributed coordination problems
  - Event ordering
  - Agreement
  - Mutual exclusion
  - Leader election
  - Deadlock detection
- Middleware for distributed application support

- Starting next week: Chapter 9 (Security)