Distributed Operating Systems - Introduction

Prof. Nalini Venkatasubramanian

(includes slides from Prof. Petru Eles and Profs. textbook slides by Kshemkalyani/Singhal)

What does an OS do?

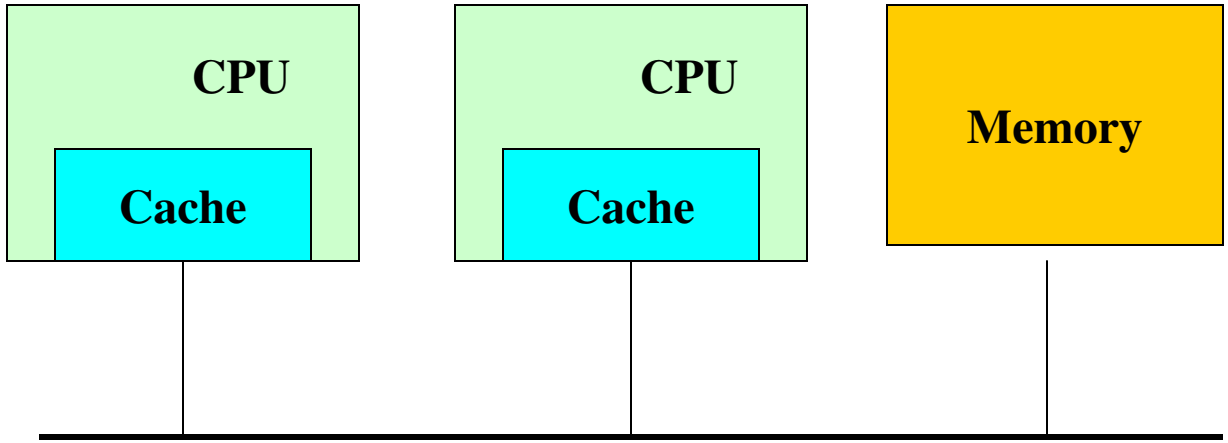
- Process/Thread Management
  - Scheduling
  - Communication
  - Synchronization
- Memory Management
- Storage Management
- FileSystems Management
- Protection and Security
- Networking

Distributed Operating System

- Manages a collection of independent computers and makes them appear to the users of the system as if it were a single computer.

Hardware Architectures

- Multiprocessors
  - Tightly coupled
  - Shared memory

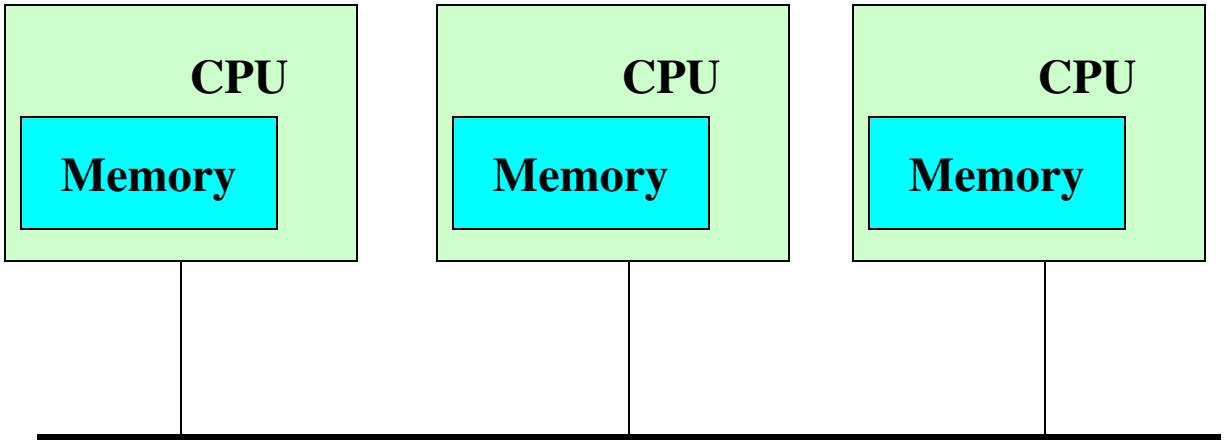


Parallel Architecture

Hardware Architectures

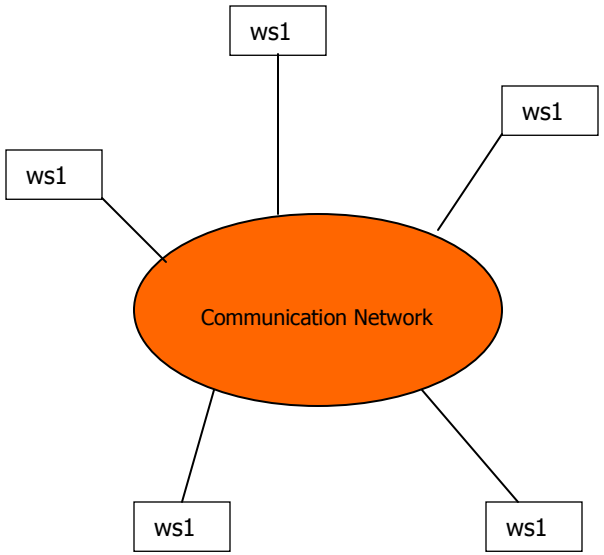
- Multicomputers
  - Loosely coupled
  - Private memory
  - Autonomous

Distributed Architecture



Workstation Model: Issues

- How to find an idle workstation?
- How is a process transferred from one workstation to another?
- What happens to a remote process if a user logs onto a workstation that was idle, but is no longer idle now?
- Other models - processor pool, workstation server...



Distributed Operating System (DOS)

- Distributed Computing Systems commonly use two types of Operating Systems.
  - Network Operating Systems
  - Distributed Operating System
- Differences between the two types
  - System Image
  - Autonomy
  - Fault Tolerance Capability

Operating System Types

- Multiprocessor OS
  - Looks like a virtual uniprocessor, contains only one copy of the OS, communicates via shared memory, single run queue
- Network OS
  - Does not look like a virtual uniprocessor, contains n copies of the OS, communicates via shared files, n run queues
- Distributed OS
  - Looks like a virtual uniprocessor (more or less), contains n copies of the OS, communicates via messages, n run queues

Design Issues

- Transparency
- Performance
- Scalability
- Reliability
- Flexibility (Micro-kernel architecture)
  - IPC mechanisms, memory management, Process management/scheduling, low level I/O
- Heterogeneity
- Security

Transparency

- Location transparency
  - processes, cpu's and other devices, files
- Replication transparency (of files)
- Concurrency transparency
  - (user unaware of the existence of others)
- Parallelism
  - User writes serial program, compiler and OS do the rest

Performance

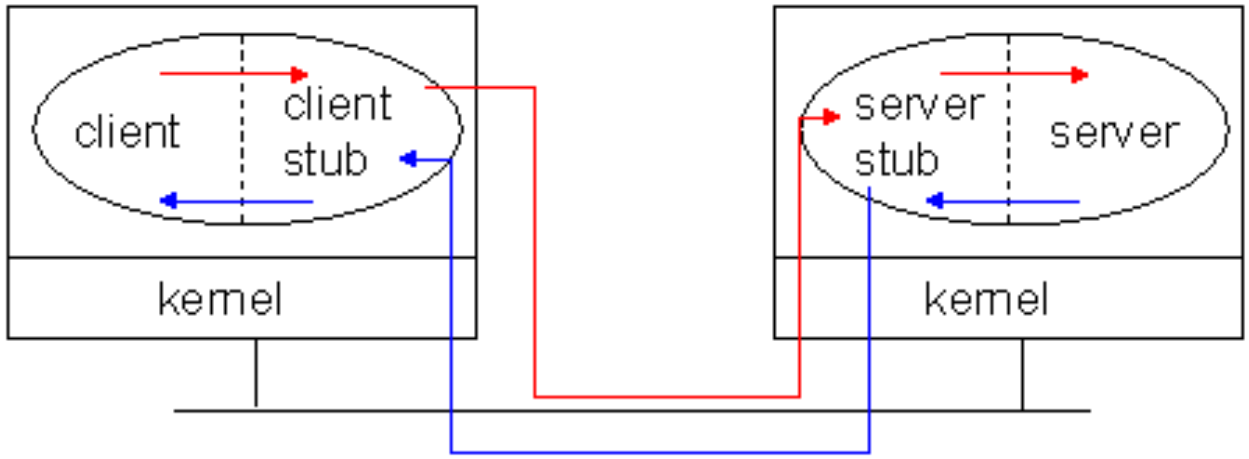
- Throughput - response time
- Load Balancing (static, dynamic)
- Communication is slow compared to computation speed
  - fine grain, coarse grain parallelism

Design Elements

- Process Management
  - Task Partitioning, allocation, load balancing, migration
- Communication
  - Two basic IPC paradigms used in DOS
    - Message Passing (RPC) and Shared Memory
  - synchronous, asynchronous
- FileSystems
  - Naming of files/directories
  - File sharing semantics
  - Caching/update/replication

Remote Procedure Call

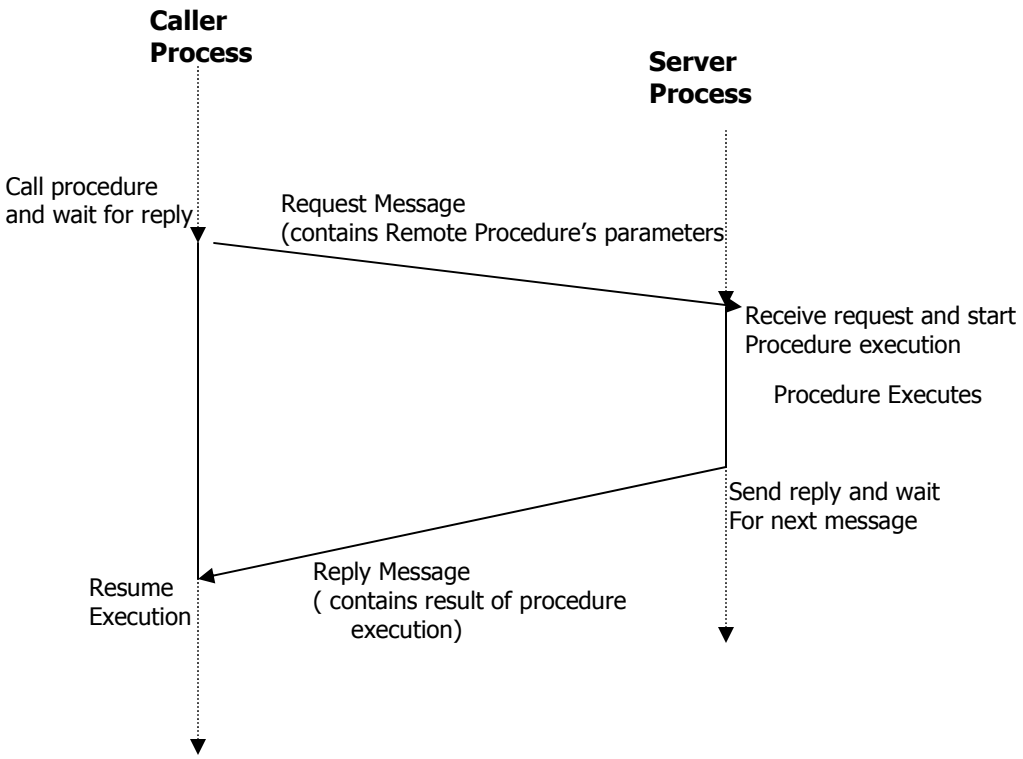
A convenient way to construct a client-server connection without explicitly writing send/ receive type programs (helps maintain transparency).



Remote Procedure Calls (RPC)

- General message passing model. Provides programmers with a familiar mechanism for building distributed applications/systems
- Familiar semantics (similar to LPC)
  - Simple syntax, well defined interface, ease of use, generality and IPC between processes on same/different machines.
- It is generally synchronous
- Can be made asynchronous by using multi-threading

A typical model for RPC



RPC continued…

- Transparency of RPC
  - Syntactic Transparency
  - Semantic Transparency

- Unfortunately achieving exactly the same semantics for RPCs and LPCs is close to impossible

  - Disjoint address spaces
  - More vulnerable to failure
  - Consume more time (mostly due to communication delays)

Implementing RPC Mechanism

- Uses the concept of stubs; A perfectly normal LPC abstraction by concealing from programs the interface to the underlying RPC
- Involves the following elements
  - The client
  - The client stub
  - The RPC runtime
  - The server stub
  - The server

Remote Procedure Call (cont.)

- Client procedure **calls** the client stub in a normal way
- Client stub **builds** a message and **traps** to the kernel
- Kernel **sends** the message to remote kernel
- Remote kernel **gives** the message to server stub
- Server stub **unpacks** parameters and **calls** the server
- Server **computes** results and **returns** it to server stub
- Server stub **packs** results in a message and **traps** to kernel
- Remote kernel **sends** message to client kernel
- Client kernel **gives** message to client stub
- Client stub **unpacks** results and **returns** to client

RPC servers and protocols…

- RPC Messages (call and reply messages)
- Server Implementation
  - Stateful servers
  - Stateless servers
- Communication Protocols
  - Request(R)Protocol
  - Request/Reply(RR) Protocol
  - Request/Reply/Ack(RRA) Protocol

RPC NG: DCOM & CORBA

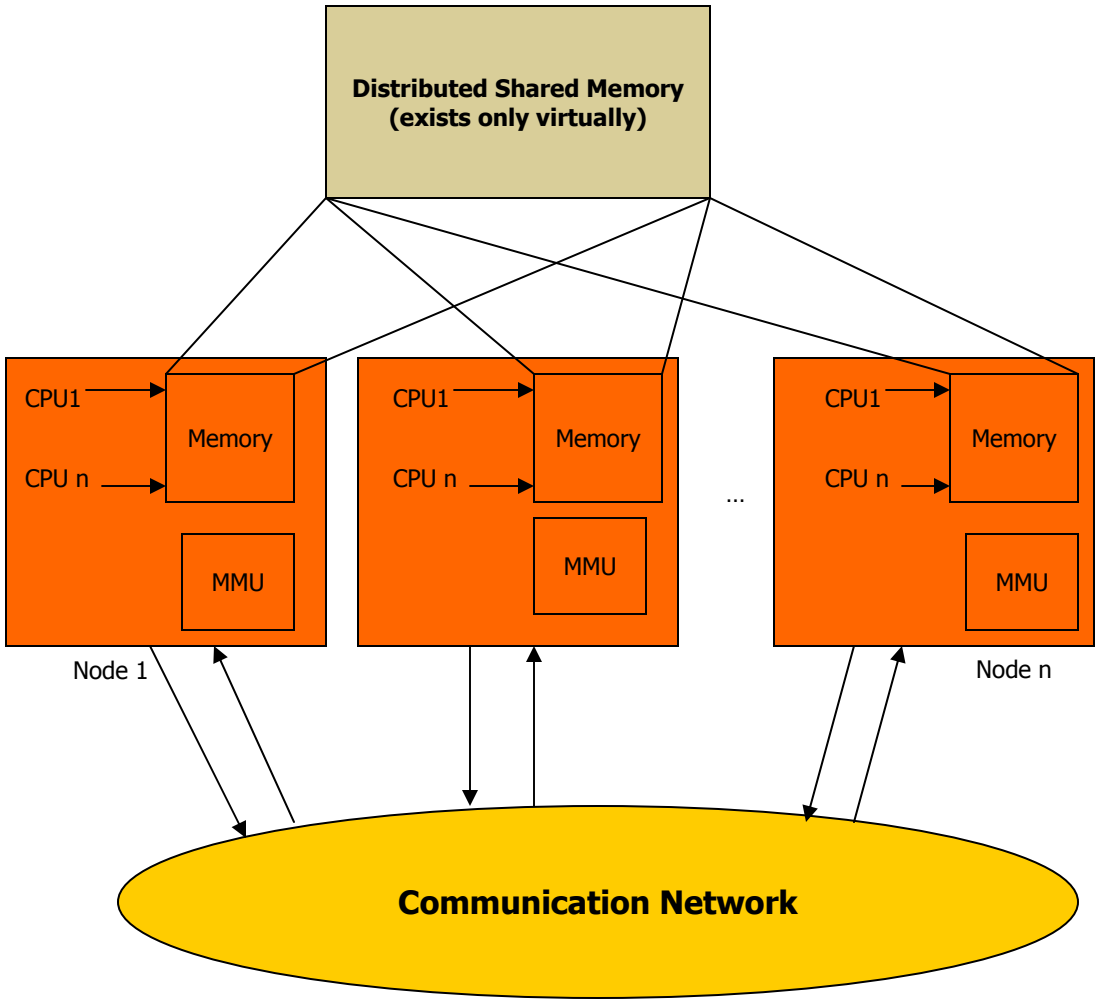
- Object models allow services and functionality to be called from distinct processes
- DCOM/COM+(Win2000) and CORBA IIOP extend this to allow calling services and objects on different machines
- More OS features (authentication,resource management,process creation,…) are being moved to distributed objects.

Distributed Shared Memory (DSM)

- Two basic IPC paradigms used in DOS
  - Message Passing (RPC)
  - Shared Memory
- Use of shared memory for IPC is natural for tightly coupled systems

- DSM is a middleware solution, which provides a shared-memory abstraction in the loosely coupled distributed-memory processors.

General Architecture of DSM



Issues in designing DSM

- Granularity of the block size
- Synchronization
- Memory Coherence (Consistency models)
- Data Location and Access
- Replacement Strategies
- Thrashing
- Heterogeneity

Synchronization

- Inevitable in Distributed Systems where distinct processes are running concurrently and sharing resources.
- Synchronization related issues
  - Clock synchronization/Event Ordering (recall happened before relation)
  - Mutual exclusion
  - Deadlocks
  - Election Algorithms

Distributed Mutual Exclusion

- Mutual exclusion
  - ensures that concurrent processes have serialized access to shared resources - the ***critical section problem***.
  - At any point in time, only one process can be executing in its critical section.
- Shared variables (semaphores) cannot be used in a distributed system
  - Mutual exclusion must be based on message passing, in the context of unpredictable delays and incomplete knowledge
  - In some applications (e.g. transaction processing) the resource is managed by a server which implements its own lock along with mechanisms to synchronize access to the resource.

Approaches to Distributed Mutual Exclusion

- Central coordinator based approach
  - A centralized coordinator determines who enters the CS
- Distributed approaches to mutual exclusion
  - Token based approach
    - A unique token is shared among the sites. A site is allowed to enter its CS if it possesses the token.
    - Mutual exclusion is ensured because the token is unique.
  - Non-token based approach
    - Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
  - Quorum based approach
    - Each site requests permission to execute the CS from a subset of sites (called a quorum).
    - Any two quorums contain a common site. This common site is responsible to make sure that only one request executes the CS at any time.

System Model for Distributed Mutual Exclusion Algorithms

- The system consists of N sites, S1, S2, ..., SN.
- We assume that a single process is running on each site. The process at site Si is denoted by pi .
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
  - In the 'requesting the CS' state, the site is blocked and can not make further requests for the CS. In the 'idle' state, the site is executing outside the CS.
- In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called the idle token state).
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

Requirements/Conditions

- Safety Property (*Mutual Exclusion*)
  - At any instant, only one process can execute the critical section.
- Liveness Property (*Progress*)
  - This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
- Fairness (*Bounded Waiting*)
  - Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

Performance Metrics for Mutual Exclusion Algorithms

- Message complexity
  - The number of messages required per CS execution by a site.
- Synchronization delay
  - After a site leaves the CS, it is the time required and before the next site enters the CS
- Response time
  - The time interval a request waits for its CS execution to be over after its request messages have been sent out
- System throughput
  - The rate at which the system executes requests for the CS.

$$\text{System throughput} = 1/(SD+E)$$

where SD is the synchronization delay and E is the average critical section execution time

Mutual Exclusion (cont'd)

- Often there is no synchronization built in which implicitely protects the resource (files, display windows, peripheral devices, etc.).

A mechanism has to be implemented at the level of the process requesting for access.

- Basic requirements for a mutual exclusion mechanism:
  - **safety**: at most one process may execute in the critical section (CS) at a time;
  - **liveness**: a process requesting entry to the CS is eventually granted it (so long as any process executing the CS eventually leaves it).
    Liveness implies freedom of *deadlock* and *starvation*.

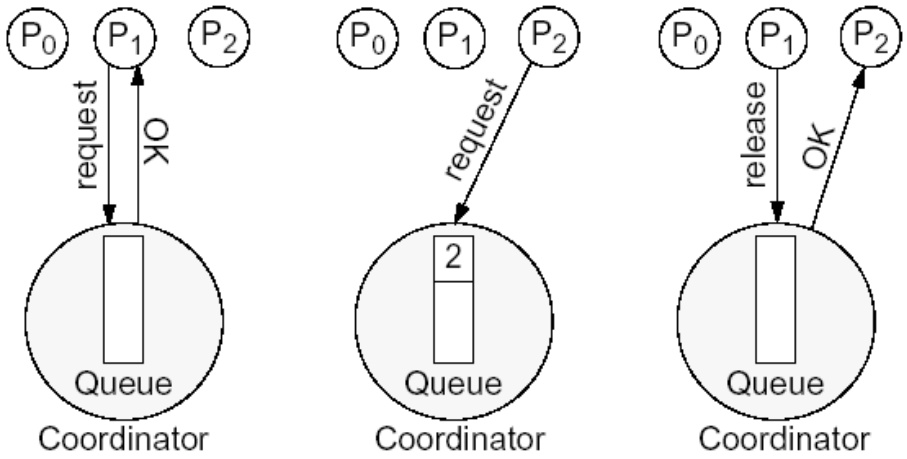
- There are two basic approaches to distributed mutual exclusion:
  1. Non-token-based: each process freely and equally competes for the right to use the shared resource; requests are arbitrated by a central control site or by distributed agreement.
  2. Token-based: a logical token representing the access right to the shared resource is passed in a regulated fashion among the processes; whoever holds the token is allowed to enter the critical section.

Mutual Exclusion Techniques Covered

- Central Coordinator Algorithm
- Non-token based
  - Lamport's Algorithm
  - Ricart-Agrawala Algorithm
- Token Based
  - Ricart-Agrawala Second Algorithm
  - Token Ring Algorithm

Central Coordinator Algorithm

- A central coordinator grants permission to enter a CS.



- To enter a CS, a process sends a request message to the coordinator and then waits for a reply (during this waiting period the process can continue with other work).
- The reply from the coordinator gives the right to enter the CS.
- After finishing work in the CS the process notifies the coordinator with a release message.

Central Coordinator Algorithm (cont'd)

- The scheme is simple and easy to implement.
- The strategy requires only three messages per use of a CS (*request*, *OK*, *release*).

Problems

- The coordinator can become a performance bottleneck.
- The coordinator is a critical point of failure:
  - If the coordinator crashes, a new coordinator must be created.
  - *The coordinator can be one of the processes competing for access*; an *election algorithm* (see later) has to be run in order to choose one and only one new coordinator.

Distributed Algorithms for Mutual Exclusion

- In a distributed environment it seems more natural to implement mutual exclusion, based upon distributed agreement - not on a central coordinator.
  - Shared variables (semaphores) cannot be used in a distributed system
  - Mutual exclusion must be based on message passing, in the context of unpredictable delays and incomplete knowledge
  - In some applications (e.g. transaction processing) the resource is managed by a server which implements its own lock along with mechanisms to synchronize access to the resource.

Lamport's Algorithm

- Basic Idea
  - Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
  - Every site S_i keeps a queue, request queue_i , which contains mutual exclusion requests ordered by their timestamps.
  - This algorithm requires communication channels to deliver messages the FIFO order.

Lamport's Algorithm

- Requesting the critical section
  - When a site Si wants to enter the CS, it broadcasts a REQUEST(ts_i , i) message to all other sites and places the request on request queuei . ((ts_i , i) denotes the timestamp of the request.)
  - When a site Sj receives the REQUEST(ts_i , i) message from site Si ,places site Si 's request on request queuej and it returns a timestamped REPLY message to Si
- Executing the critical section
  - Site Si enters the CS when the following two conditions hold:
    - L1: Si has received a message with timestamp larger than (ts_i , i) from all other sites.
    - L2: Si 's request is at the top of request queue_i .
- Releasing the critical section
  - Site Si , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
  - When a site Sj receives a RELEASE message from site Si , it removes Si 's request from its request queue.
  - When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

Performance – Lamport's Algorithm

- For each CS execution Lamport's algorithm requires
  - (N – 1) REQUEST messages, (N – 1) REPLY messages, and (N – 1) RELEASE messages.
  - Thus, Lamport's algorithm requires 3(N – 1) messages per CS invocation.
- Optimization
  - In Lamport's algorithm, REPLY messages can be omitted in certain situations.
  - For example, if site Sj receives a REQUEST message from site Si after it has sent its own REQUEST message with timestamp higher than the timestamp of site Si 's request, then site Sj need not send a REPLY message to site Si .
  - This is because when site Si receives site Sj 's request with timestamp higher than its own, it can conclude that site Sj does not have any smaller timestamp request which is still pending.
  - With this optimization, Lamport's algorithm requires between 3(N – 1) and 2(N – 1) messages per CS execution.

Ricart-Agrawala Algorithm

- It is assumed that all processes keep a (Lamport's) logical clock which is updated according to the clock rules.
  - The algorithm requires a total ordering of requests. Requests are ordered according to their global logical timestamps; if timestamps are equal, process identifiers are compared to order them.
- The process that requires entry to a CS multicasts the request message to all other processes competing for the same resource.
  - Process is allowed to enter the CS when all processes have replied to this message.
  - The request message consists of the requesting process' timestamp (logical clock) and its identifier.
- Each process keeps its state with respect to the CS: released, requested, or held.

Ricart-Agrawala Algorithm (cont'd)

The Algorithm

- Rule for process initialization

  /* performed by each process P_i at initialization */

[RI1]: $state_{Pi}$:= RELEASED.

- Rule for access request to CS

  /* performed whenever process P_i requests an access to the CS */

[RA1]: $state_{Pi}$:= REQUESTED.
T_{Pi} := the value of the local logical clock corresponding to this request.

[RA2]: P_i sends a request message to all processes; the message is of the form (T_{Pi}, i), where i is an identifier of P_i.

[RA3]: P_i waits until it has received replies from all other n-1 processes.

- Rule for executing the CS

  /* performed by P_i after it received the n-1 replies */

[RE1]: $state_{Pi}$:= HELD.
P_i enters the CS.

Ricart-Agrawala Algorithm (cont'd)

- Rule for handling incoming requests

  /* performed by P_i whenever it received a request (T_{Pj}, j) from P_j */

[RH1]: **if** $state_{Pi}$ = HELD **or** (($state_{Pi}$ = REQUESTED) **and** ((T_{Pi}, i) < (T_{Pj}, j))) **then**

Queue the request from P_j without replying.

else

Reply immediately to P_j.

end if.

- Rule for releasing a CS

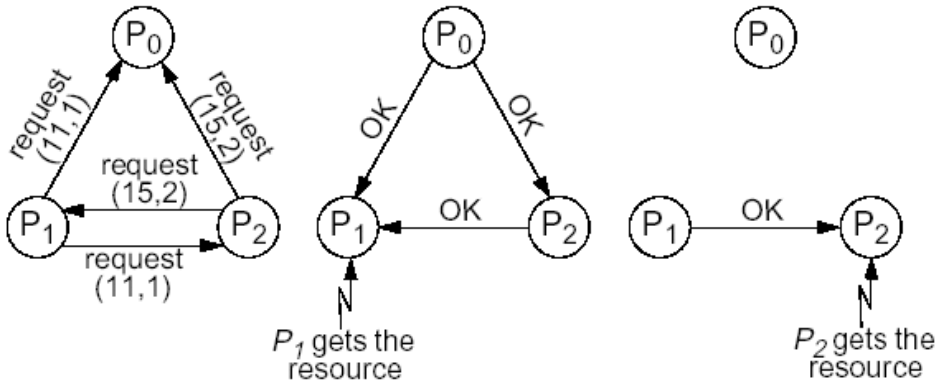
  /* performed by P_i after it finished work in a CS */

[RR1]: $state_{Pi}$:= RELEASED.

P_i replies to all queued requests.

Ricart-Agrawala Algorithm (cont'd)

- A request issued by a process P_j is blocked by another process P_i only if P_i is holding the resource or if it is requesting the resource with a higher priority (this means a smaller timestamp) then P_j.



Problems

- The algorithm is expensive in terms of message traffic; it requires 2(n-1) messages for entering a CS: (n-1) requests and (n-1) replies.
- The failure of any process involved makes progress impossible if no special recovery measures are taken.

Token-Based Mutual Exclusion

- Ricart-Agrawala Second Algorithm

- Token Ring Algorithm

Ricart-Agrawala Second Algorithm

- A process is allowed to enter the critical section when it gets the token.
  - Initially the token is assigned arbitrarily to one of the processes.
- In order to get the token it sends a request to all other processes competing for the same resource.
  - The request message consists of the requesting process' timestamp (logical clock) and its identifier.
- When a process Pi leaves a critical section
  - it passes the token to one of the processes which are waiting for it; this will be the first process Pj, where j is searched in order [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] for which there is a pending request.
  - If no process is waiting, Pi retains the token (and is allowed to enter the CS if it needs); it will pass over the token as result of an incoming request.
- How does Pi find out if there is a pending request?
  - Each process Pi records the timestamp corresponding to the last request it got from process Pj, in requestPi[j]. In the token itself, token[j] records the timestamp (logical clock) of Pj's last holding of the token. If requestPi[j] > token[j] then Pj has a pending request.

Ricart-Agrawala Second Algorithm (cont'd)

The Algorithm

- Rule for process initialization

/* performed at initialization */

[RI1]: $state_{Pi}$:= NO-TOKEN for all processes P_i, except *one single process* P_X for which $state_{PX}$:= TOKEN-PRESENT.

[RI2]: *token*[*k*] initialized 0 for all elements $k = 1 .. n$. $request_{Pi}[k]$ initialized 0 for all processes P_i and all elements $k = 1 .. n$.

- Rule for access request and execution of the CS

/* performed whenever process P_i requests an access to the CS and when it finally gets it; in particular P_i can already possess the token */

[RA1]: **if** $state_{Pi}$ = NO-TOKEN **then**

P_i sends a request message to all processes; the message is of the form (T_{Pi}, *i*), where $T_{Pi} = C_{Pi}$ is the value of the local logical clock, and *i* is an identifier of P_i.

P_i waits until it receives the token.

end if.

$state_{Pi}$:= TOKEN-HELD.

P_i enters the CS.

Ricart-Agrawala Second Algorithm (cont'd)

☞ Rule for handling incoming requests

/* performed by P_i whenever it received a request (T_{Pj}, j) from P_j */

[RH1]: $request[j]$:= max($request[j]$, T_{Pj}).

[RH2]: **if** $state_{Pi}$ = TOKEN-PRESENT **then**

P_i releases the resource (see rule RR2).

end if.

☞ Rule for releasing a CS

/* performed by P_i after it finished work in a CS or when it holds a token without using it and it got a request */

[RR1]: $state_{Pi}$ = TOKEN-PRESENT.

[RR2]: **for** k = [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] **do**

if request[k] > token[k] **then**

$state_{Pi}$:= NO-TOKEN.

$token[i]$:= C_{Pi}, the value of the local logical clock.

P_i sends the token to P_k.

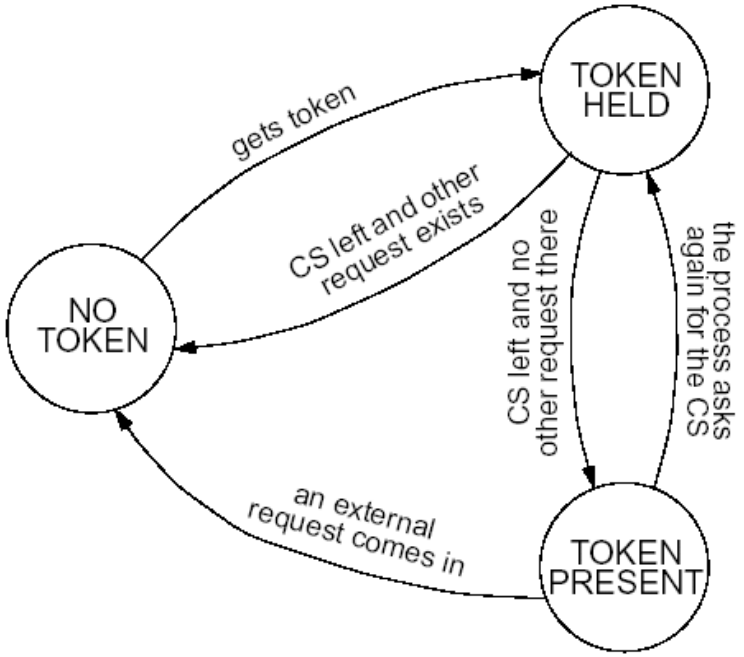
break. /* leave the for loop */

end if.

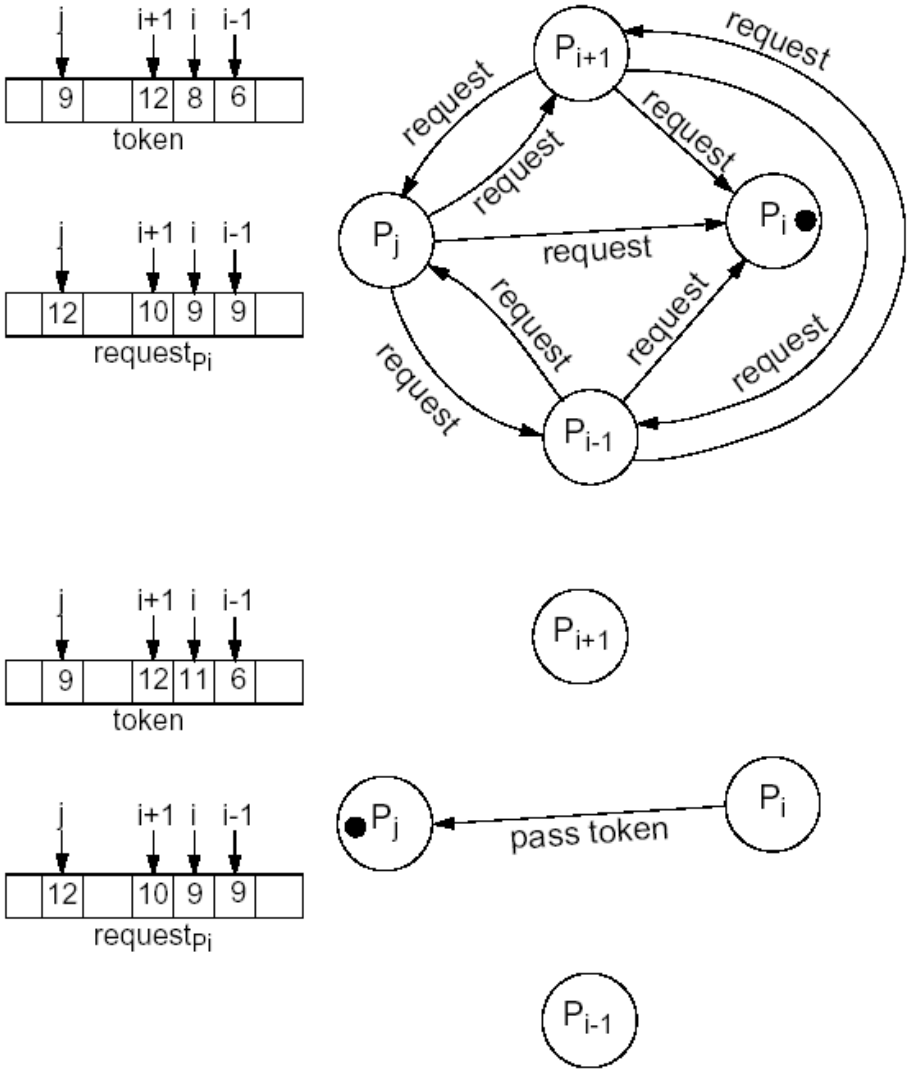
end for.

Ricart-Agrawala Second Algorithm (cont'd)

- Each process keeps its state with respect to the token: NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.



Ricart-Agrawala Second Algorithm (cont'd)



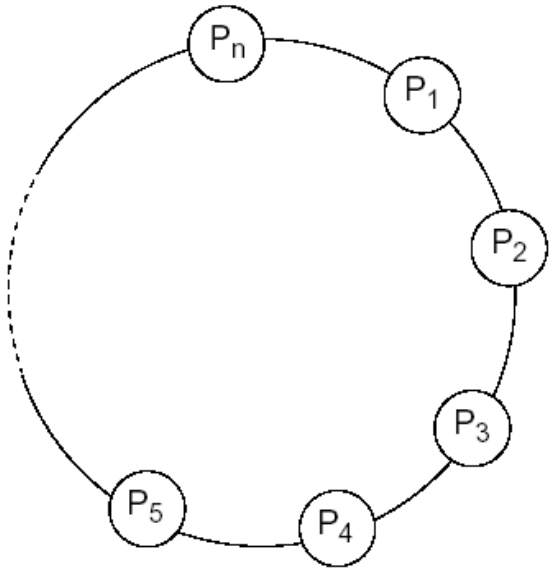
Ricart-Agrawala Second Algorithm (cont'd)

- The complexity is reduced compared to the (first) Ricart-Agrawala algorithm: it requires *n* messages for entering a CS: (*n*-1) requests and one reply.

- The failure of a process, except the one which holds the token, doesn't prevent progress.

Token Ring Algorithm

- A very simple way to solve mutual exclusion ⇒ arrange the *n* processes P_1, P_2, ... P_n in a logical ring.
- The logical ring topology is created by giving each process the address of one other process which is its neighbour in the clockwise direction.
- The logical ring topology is unrelated to the physical interconnections between the computers.



Token Ring Algorithm (cont'd)

The algorithm

- The token is initially given to one process.
- The token is passed from one process to its neighbour round the ring.
- When a process requires to enter the CS, it waits until it receives the token from its left neighbour and then it retains it; after it got the token it enters the CS; after it left the CS it passes the token to its neighbour in clockwise direction.
- When a process receives the token but does not require to enter the critical section, it immediately passes the token over along the ring.

Token Ring Algorithm (cont'd)

- It can take from 1 to *n*-1 messages to obtain a token. Messages are sent around the ring even when no process requires the token ⇒ additional load on the network.

The algorithm works well in heavily loaded situations, when there is a high probability that the process which gets the token wants to enter the CS. It works poorly in lightly loaded cases.

- If a process fails, no progress can be made until a reconfiguration is applied to extract the process from the ring.

- If the process holding the token fails, a unique process has to be picked, which will regenerate the token and pass it along the ring; an *election algorithm* (see later) has to be run for this purpose.

Election Algorithms

- Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role.

- Examples with mutual exclusion
  - Central coordinator algorithm
    - At initialization or whenever the coordinator crashes, a new coordinator has to be elected.
  - Token ring algorithm
    - When the process holding the token fails, a new process has to be elected which generates the new token.

Election Algorithms

- It doesn't matter which process is elected.
  - What is important is that one and only one process is chosen (we call this process the coordinator) and all processes agree on this decision.
- Assume that each process has a unique number (identifier).
  - In general, election algorithms attempt to locate the process with the highest number, among those which currently are up.
- Election is typically started after a failure occurs.
  - The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out → a process that gets no response for a period of time suspects a failure and initiates an election process.
- An election process is typically performed in two phases:
  - Select a leader with the highest priority.
  - Inform all processes about the winner.

The Bully Algorithm

- A process has to know the identifier of all other processes
  - (it doesn't know, however, which one is still up); the process with the highest identifier, among those which are up, is selected.
- Any process could fail during the election procedure.
- When a process Pi detects a failure and a coordinator has to be elected
  - it sends an election message to all the processes with a higher identifier and then waits for an answer message:
  - If no response arrives within a time limit
    - Pi becomes the coordinator (all processes with higher identifier are down)
    - it broadcasts a coordinator message to all processes to let them know.
  - If an answer message arrives,
    - Pi knows that another process has to become the coordinator → it waits in order to receive the coordinator message.
    - If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the answer message) Pi resends the election message.
- When receiving an election message from Pi
  - a process Pj replies with an answer message to Pi and
  - then starts an election procedure itself(unless it has already started one) it sends an election message to all processes with higher identifier.
- Finally all processes get an answer message, except the one which becomes the coordinator.

The Bully Algorithm (cont'd)

The Algorithm

- By default, the state of a process is ELECTION-OFF

- Rule for election process initiator

  /* performed by a process P_i, which triggers the election procedure, or which starts an election after receiving itself an *election message* */

[RE1]: $state_{Pi}$:= ELECTION-ON.

P_i sends an *election message* to all processes with a higher identifier.

P_i waits for *answer message*.

if no *answer message* arrives before time-out **then**

P_i is the coordinator and sends a *coordinator message* to all processes.

else

P_i waits for a *coordinator message* to arrive.

if no *coordinator message* arrives before time-out **then**

restart election procedure according to RE1

end if

end if.

The Bully Algorithm (cont'd)

☞ Rule for handling an incoming *election message*

/* performed by a process P_i at reception of an *election message* coming from P_j */

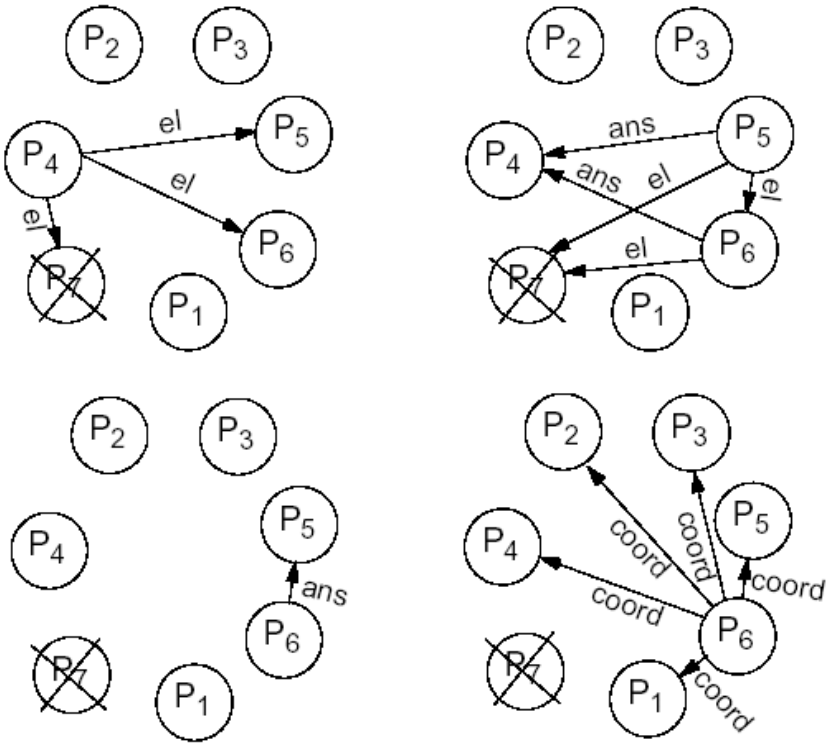
[RH1]: P_i replies with an *answer message* to P_j.

[RH2]: **if** $state_{Pi}$:= ELECTION-OFF **then**

start election procedure according to RE1

end if

The Bully Algorithm (cont'd)



- If P_6 crashes before sending the coordinator message, P_4 and P_5 restart the election process.

☞ The best case: the process with the second highest identifier notices the coordinator's failure. It can immediately select itself and then send n-2 coordinator messages.

☞ The worst case: the process with the lowest identifier initiates the election; it sends n-1 election messages to processes which themselves initiate each one an election ⇒ O(n^2) messages.

The Ring-based Algorithm

- We assume that the processes are arranged in a logical ring
  - Each process knows the address of one other process, which is its neighbor in the clockwise direction.
- The algorithm elects a single coordinator, which is the process with the highest identifier.
- Election is started by a process which has noticed that the current coordinator has failed.
  - The process places its identifier in an election message that is passed to the following process.
  - When a process receives an election message
    - It compares the identifier in the message with its own.
    - If the arrived identifier is greater, it forwards the received election message to its neighbor
    - If the arrived identifier is smaller it substitutes its own identifier in the election message before forwarding it.
    - If the received identifier is that of the receiver itself → this will be the coordinator.
- The new coordinator sends an elected message through the ring.

The Ring-based Algorithm- An Optimization

- Several elections can be active at the same time.
  - Messages generated by later elections should be killed as soon as possible.
- Processes can be in one of two states
  - Participant or Non-participant.
    - Initially, a process is non-participant.

- The process initiating an election marks itself participant.
- Rules
  - For a participant process, if the identifier in the election message is smaller than the own, does not forward any message (it has already forwarded it, or a larger one, as part of another simultaneously ongoing election).
  - When forwarding an election message, a process marks itself participant.
  - When sending (forwarding) an elected message, a process marks itself non-participant.

The Ring-Based Algorithm (cont'd)

The Algorithm

- By default, the state of a process is NON-PARTICIPANT

- Rule for election process initiator

  /* performed by a process P_i, which triggers the election procedure */

[RE1]: $state_{Pi}$:= PARTICIPANT.

[RE2]: P_i sends an *election message* with *message.id* := *i* to its neighbour.

- Rule for handling an incoming *election message*

  /* performed by a process P_j, which receives an *election message* */

[RH1]: **if** *message.id* > *j* **then**

P_j forwards the received *election message*.

$state_{Pj}$:= PARTICIPANT.

elseif *message.id* < *j* **then**

if $state_{Pj}$ = NON-PARTICIPANT **then**

P_j forwards an *election message* with *message.id* := *j*.

$state_{Pj}$:= PARTICIPANT

end if

else

P_j is the coordinator and sends an *elected message* with *message.id* := *j* to its neighbour.

$state_{Pj}$:= NON-PARTICIPANT.

end if.

The Ring-Based Algorithm (cont'd)

- Rule for handling an incoming *elected message*

  /* performed by a process P_i, which receives an *elected message* */

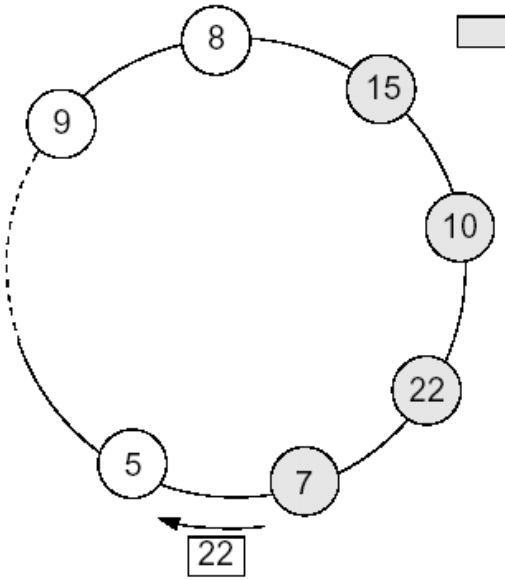
[RD1]: **if** *message.id* $\neq$ *i* **then**

P_i forwards the received *elected message*.

$state_{Pj}$:= NON-PARTICIPANT.

end if.

The Ring-Based Algorithm (cont'd)



state = PARTICIPANT

If, in this stage, process 8 initiates another election, the *election message* will be killed by process 15

- With one single election started:
  - On average: $n/2$ (election) messages needed to reach maximal node; n (election) messages to return to maximal node; n messages to rotate *elected message*.
    Number of messages: $2n + n/2$.
  - Worst case: n-1 messages needed to reach maximal node;
    Number of messages: $3n - 1$.

- The ring algorithm is more efficient on average then the bully algorithm.

Summary (Distributed Mutual Exclusion)

- In a distributed environment no shared variables (semaphores) and local kernels can be used to enforce mutual exclusion. Mutual exclusion has to be based only on message passing.
- There are two basic approaches to mutual exclusion: non-token-based and token-based.
- The central coordinator algorithm is based on the availability of a coordinator process which handles all the requests and provides exclusive access to the resource. The coordinator is a performance bottleneck and a critical point of failure. However, the number of messages exchanged per use of a CS is small.
- The Ricart-Agrawala algorithm is based on fully distributed agreement for mutual exclusion. A request is multicast to all processes competing for a resource and access is provided when all processes have replied to the request. The algorithm is expensive in terms of message traffic, and failure of any process prevents progress.
- Ricart-Agrawala's second algorithm is token-based. Requests are sent to all processes competing for a resource but a reply is expected only from the process holding the token. The complexity in terms of message traffic is reduced compared to the first algorithm. Failure of a process (except the one holding the token) does not prevent progress.

Summary (Distributed Mutual Exclusion)

- The token-ring algorithm very simply solves mutual exclusion. It is requested that processes are logically arranged in a ring. The token is permanently passed from one process to the other and the process currently holding the token has exclusive right to the resource. The algorithm is efficient in heavily loaded situations.

- For many distributed applications it is needed that one process acts as a coordinator. An election algorithm has to choose one and only one process from a group, to become the coordinator. All group members have to agree on the decision.

- The bully algorithm requires the processes to know the identifier of all other processes; the process with the highest identifier, among those which are up, is selected. Processes are allowed to fail during the election procedure.

- The ring-based algorithm requires processes to be arranged in a logical ring. The process with the highest identifier is selected. On average, the ring based algorithm is more efficient then the bully algorithm.

Deadlocks

- Mutual exclusion, hold-and-wait, No-preemption and circular wait.
- Deadlocks can be modeled using resource allocation graphs
- Handling Deadlocks
  - Avoidance (requires advance knowledge of processes and their resource requirements)
  - Prevention (collective/ordered requests, preemption)
  - Detection and recovery (local/global WFGs, local/centralized deadlock detectors; Recovery by operator intervention, termination and rollback)

Resource Management Policies

- Load Estimation Policy
  - How to estimate the workload of a node
- Process Transfer Policy
  - Whether to execute a process locally or remotely
- Location Policy
  - Which node to run the remote process on
- Priority Assignment Policy
  - Which processes have more priority (local or remote)
- Migration Limiting policy
  - Number of times a process can migrate

Process Management

- Process migration
  - Freeze the process on the source node and restart it at the destination node
  - Transfer of the process address space
  - Forwarding messages meant for the migrant process
  - Handling communication between cooperating processes separated as a result of migration
  - Handling child processes

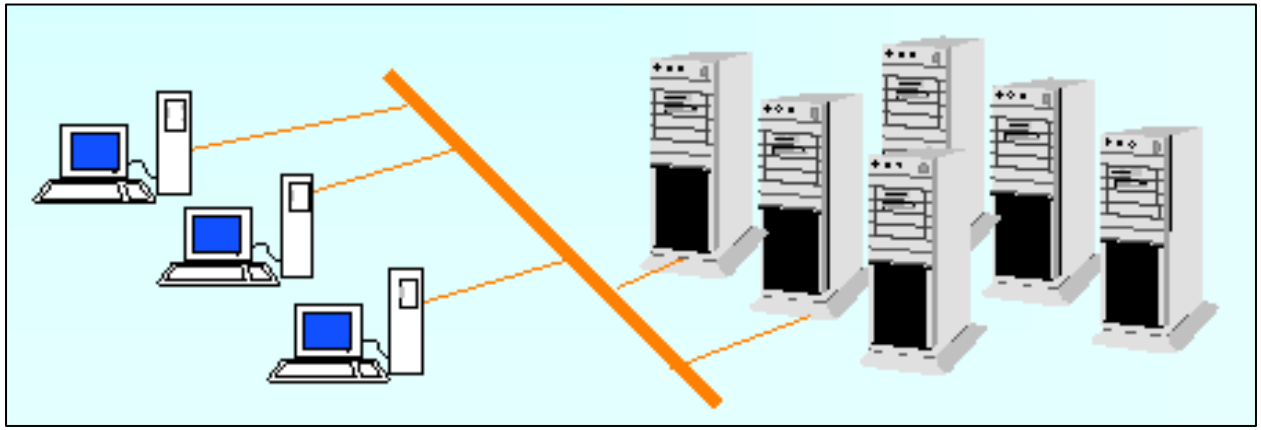
- Process migration in heterogeneous systems

Process Migration

- **Load Balancing**
  - Static load balancing - CPU is determined at process creation.
  - Dynamic load balancing - processes dynamically migrate to other computers to balance the CPU (or memory) load.
- **Migration architecture**
  - One image system
  - Point of entrance dependent system (the deputy concept)

A Mosix Cluster

- Mosix (from Hebrew U): Kernel level enhancement to Linux that provides dynamic load balancing in a network of workstations.
- Dozens of PC computers connected by local area network (Fast-Ethernet or Myrinet).
- Any process can migrate anywhere anytime.

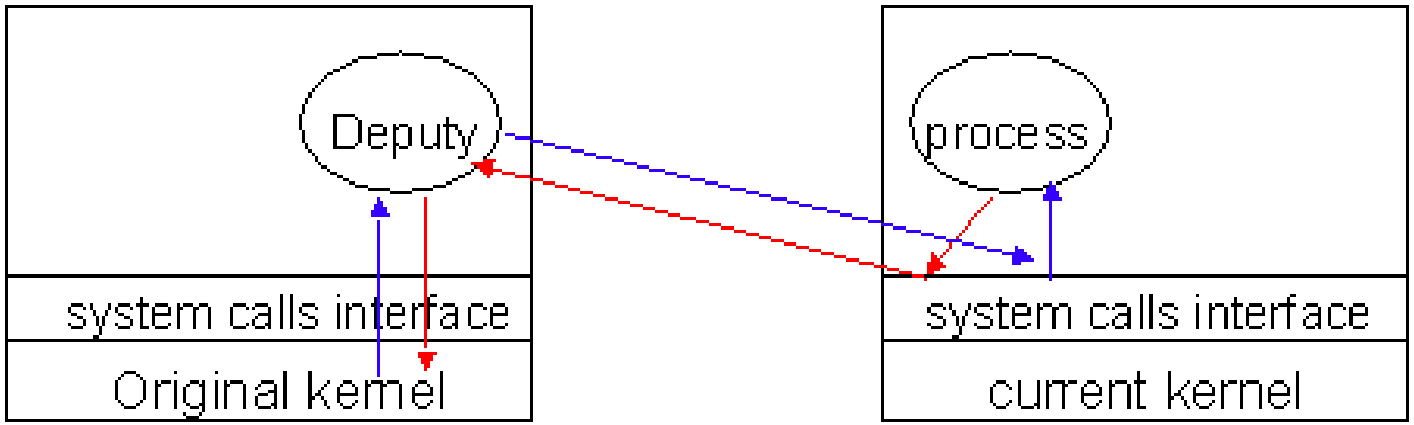


An Architecture for Migration

Architecture that fits one system image.
Needs location transparent file system.

(Mosix previous versions)

Architecture for Migration (cont.)



Architecture that fits entrance dependant systems.
Easier to implement based on current Unix.

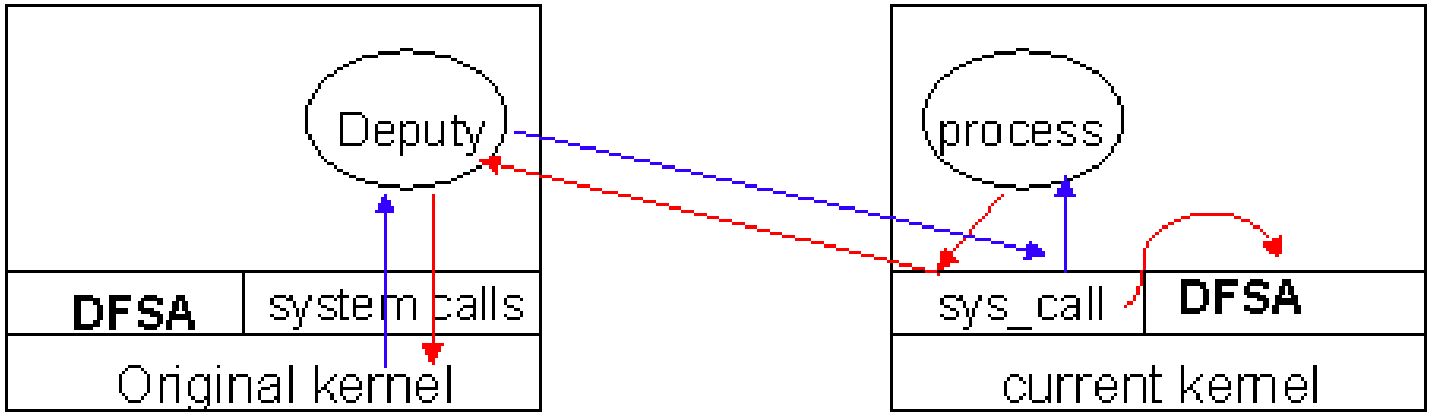
(Mosix current versions)

Mosix: File Access

Each file access must go back to deputy…

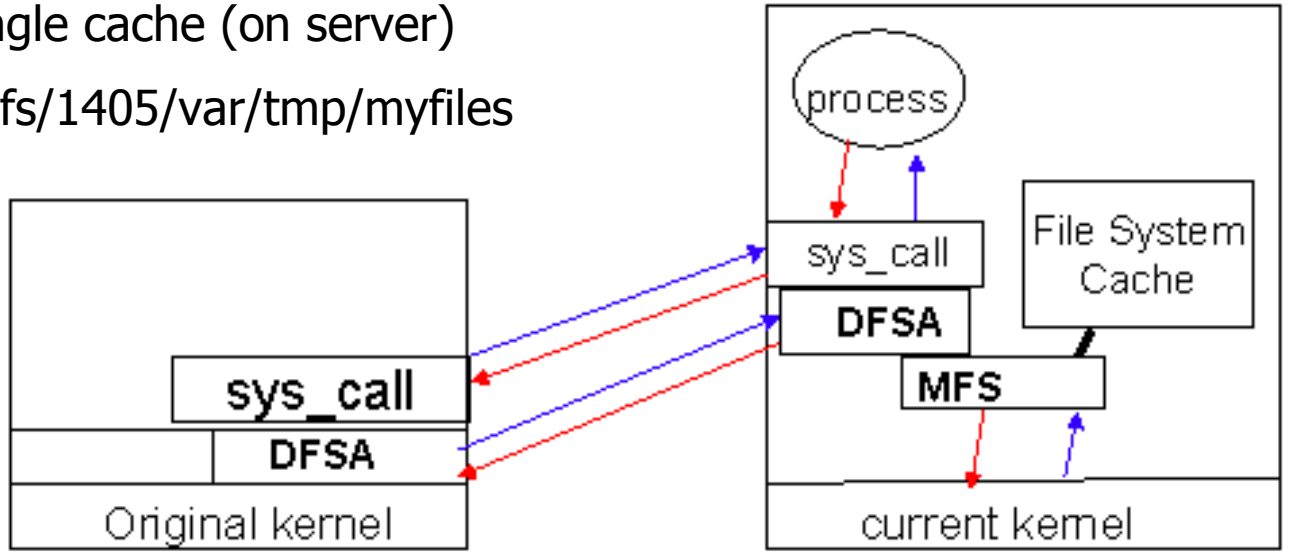
= = Very Slow for I/O apps.

Solution: Allow processes to access a distributed file system through the current kernel.



Mosix: File Access

- **DFSA**
  - Requirements (cache coherent, monotonic timestamps, files not deleted until all nodes finished)
  - Bring the process to the files.
- **MFS**
  - Single cache (on server)
  - /mfs/1405/var/tmp/myfiles



Other Considerations for Migration

- Not only CPU load!!!
- Memory.
- I/O - where is the physical device?
- Communication - which processes communicate with which other processes?

Resource Management of DOS

- A new online job assignment policy based on economic principles, competitive analysis.
- Guarantees near-optimal global lower-bound performance.
- Converts usage of heterogeneous resources (CPU, memory, IO) into a single, homogeneous cost using a specific cost function.
- Assigns/migrates a job to the machine on which it incurs the lowest cost.

Distributed File Systems (DFS)

- DFS is a distributed implementation of the classical file system model
  - Issues - File and directory naming, semantics of file sharing
- Important features of DFS
  - Transparency, Fault Tolerance
- Implementation considerations
  - caching, replication, update protocols
- The general principle of designing DFS: know the clients have cycles to burn, cache whenever possible, exploit usage properties, minimize system wide change, trust the fewest possible entries and batch if possible.

File and Directory Naming

- Machine + path /machine/path
  - one namespace but not transparent
- Mounting remote filesystems onto the local file hierarchy
  - view of the filesystem may be different at each computer
- Full naming transparency
  - A single namespace that looks the same on all machines

File Sharing Semantics

- One-copy semantics
  - Updates are written to the single copy and are available immediately
- Serializability
  - Transaction semantics (file locking protocols implemented - share for read, exclusive for write).
- Session semantics
  - Copy file on open, work on local copy and copy back on close

Example: Sun-NFS

- Supports heterogeneous systems
- Architecture
  - Server exports one or more directory trees for access by remote clients
  - Clients access exported directory trees by mounting them to the client local tree
  - Diskless clients mount exported directory to the root directory
- Protocols
  - Mounting protocol
  - Directory and file access protocol - stateless, no open-close messages, full access path on read/write
- Semantics - no way to lock files

Example: Andrew File System

- Supports information sharing on a large scale
- Uses a session semantics
  - Entire file is copied to the local machine (Venus) from the server (Vice) when open. If file is changed, it is copied to server when closed.
    - Works because in practice, most files are changed by one person

AFS File Validation

- Older AFS Versions
  - On open: Venus accesses Vice to see if its copy of the file is still valid. Causes a substantial delay even if the copy is valid.
  - Vice is stateless
- Newer AFS Versions

The Coda File System

- Descendant of AFS that is substantially more resilient to server and network failures.
- Support for “mobile” users.
- Directories are replicated in several servers (Vice)
- When the Venus is disconnected, it uses local versions of files. When Venus reconnects, it reintegrates using optimistic update scheme.

Naming and Security

- Naming
  - Important for achieving location transparency
  - Facilitates Object Sharing
  - Mapping is performed using directories. Therefore name service is also known as *Directory Service*
- Security
  - Client-Server model makes security difficult
  - Cryptography is the solution