Advanced Operating System

Unit – II

Dr.M.Lalli

Dept of CS, Trichy

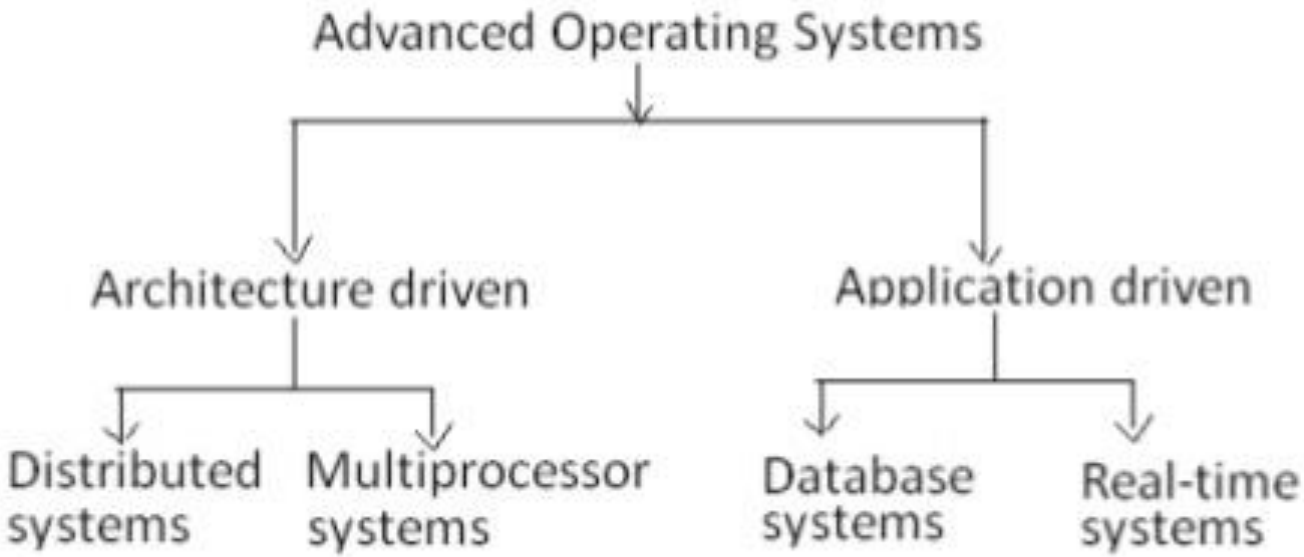
Distributed Operating Systems:

- System Architectures
- Design issues
- Communication models
- Clock synchronization
- Mutual exclusion
- Election algorithms
- Distributed Deadlock detection

What is OS?

- Operating System is a software, which makes a computer to actually work.
- It is the software the enables all the programs we use.
- The OS organizes and controls the hardware.
- OS acts as an interface between the application programs and the machine hardware.
- Examples: Windows, Linux, Unix and Mac OS, etc.,

Types of Advanced OSs



- ✓ Distributed Operating Systems
- ✓ Multiprocessor Operating Systems
- ✓ Database Operating Systems
- ✓ Real-time Operating Systems

What is Distributed Systems?

- Distributed System is used to describe a system with the following characteristics:
- Consists of several computers that do not share a memory or a clock;
- The computers communicate with each other by exchanging messages over a communication network; and
- Each computer has its own memory and runs its own operating system.

Characteristcs of Distributed systems:

- Concurrency
- No global clock
- Independent failure

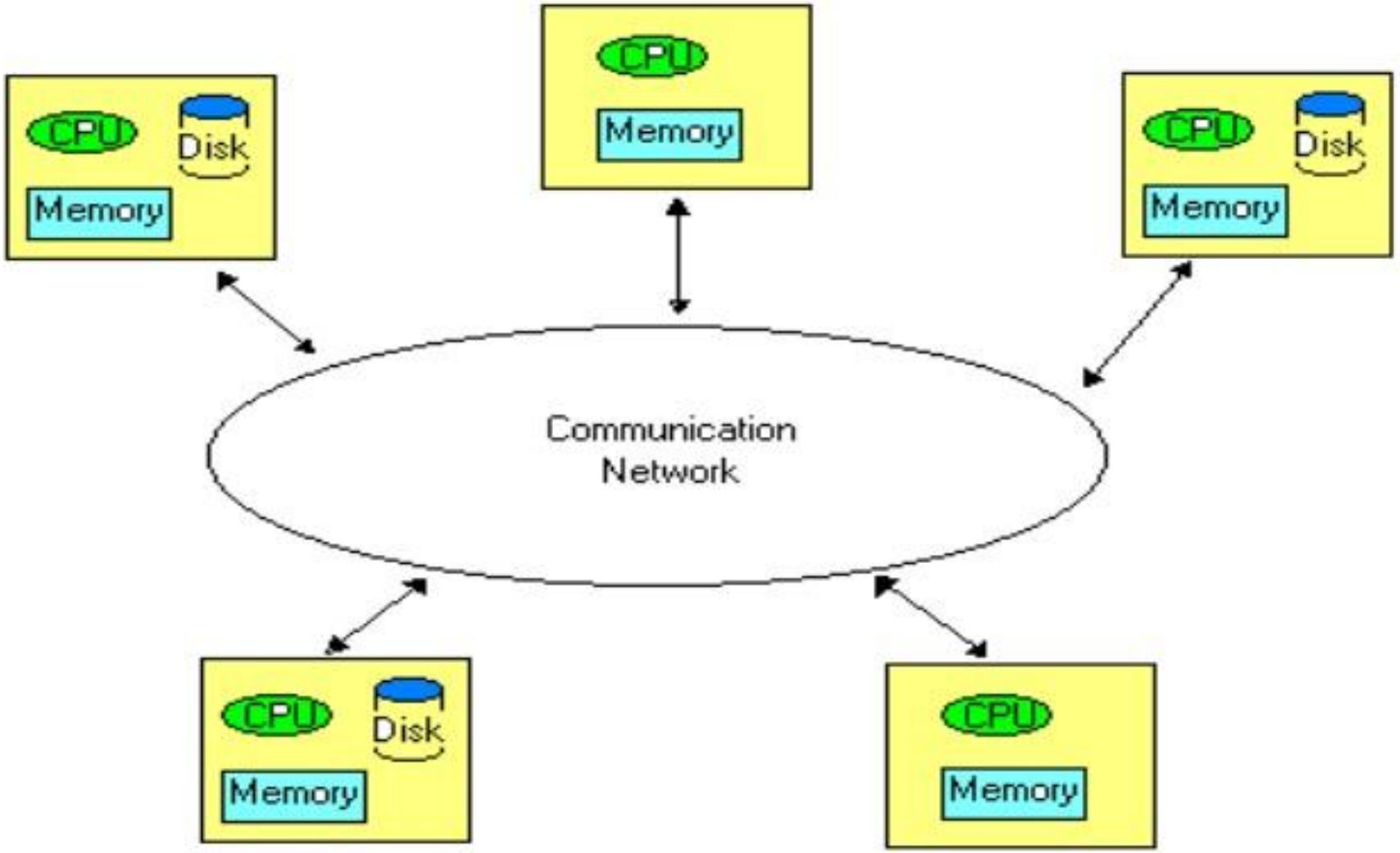
Advantages of Distributed systems:

- Resource sharing
- Enhance performance
- Improved reliability and availability
- Modular Expandability

Limitations of Distributed systems:

- Lack of common memory and common clock

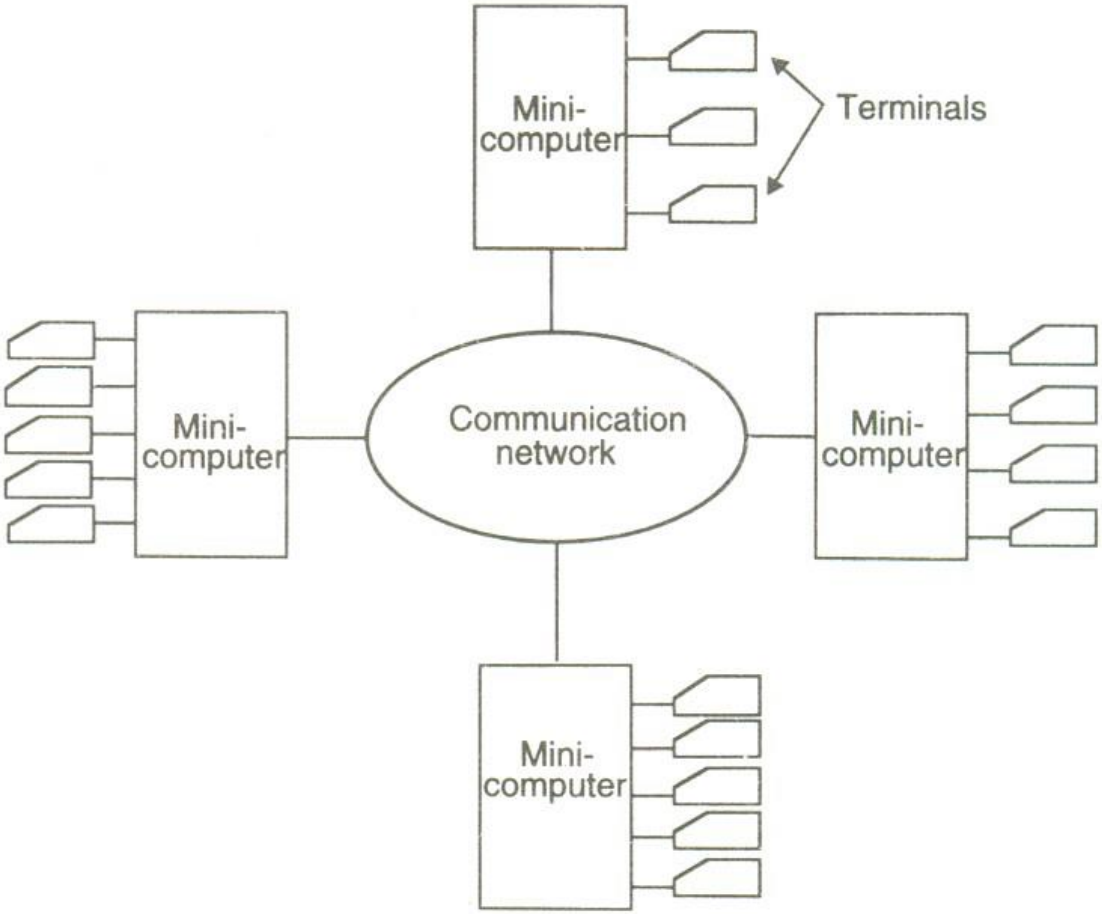
Architecture of Distributed OS



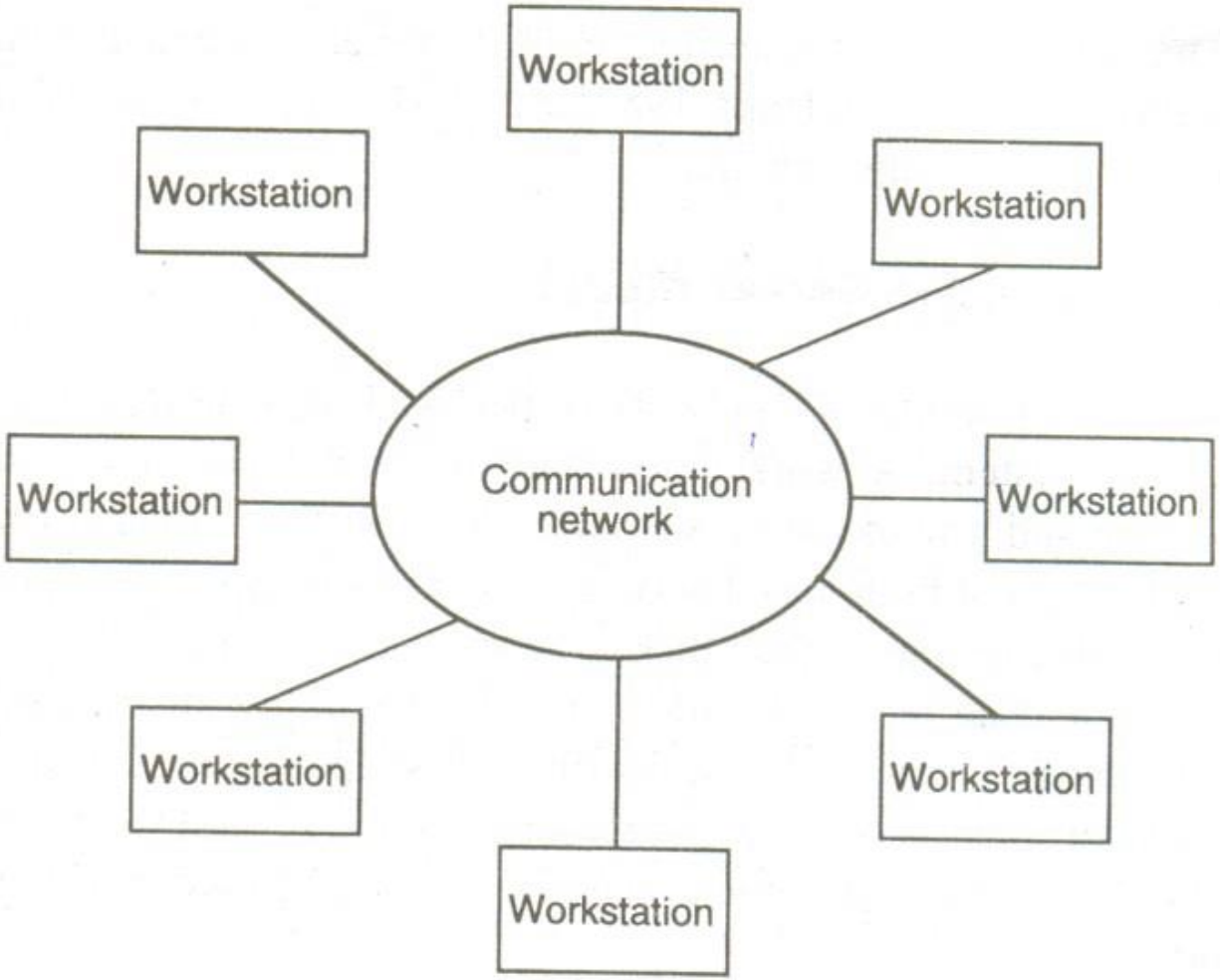
System Architecture Types

1. Mini Computer Model
2. Workstation
3. Workstation Server
4. Processor Pool Model
5. Hybrid Model

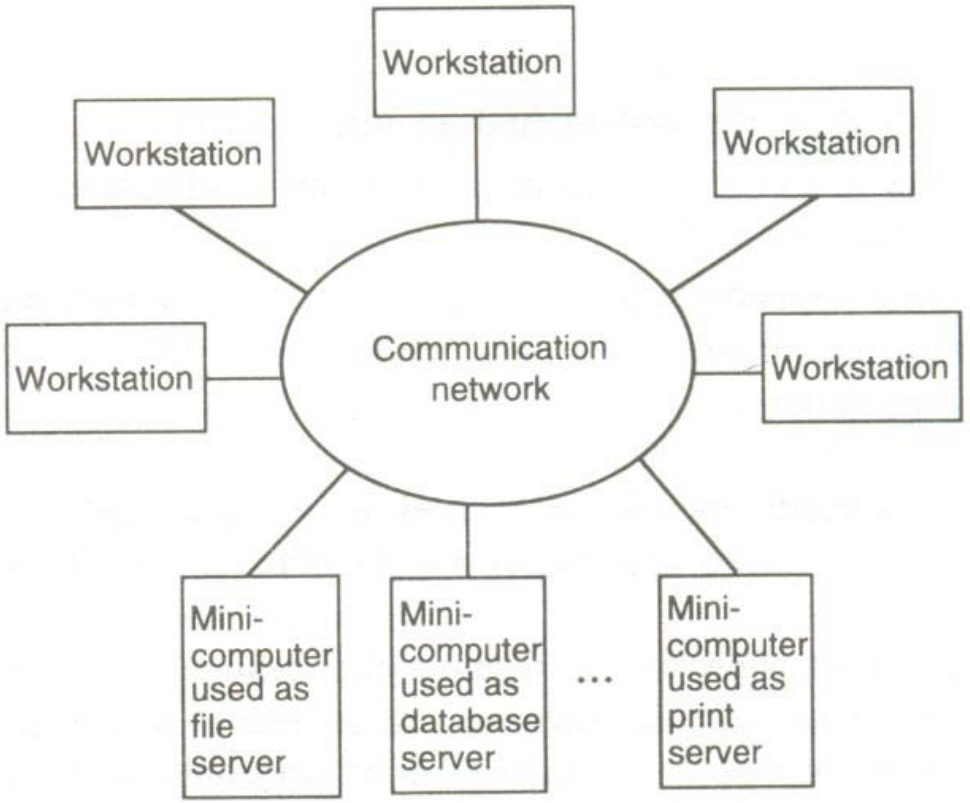
Mini Computer Model



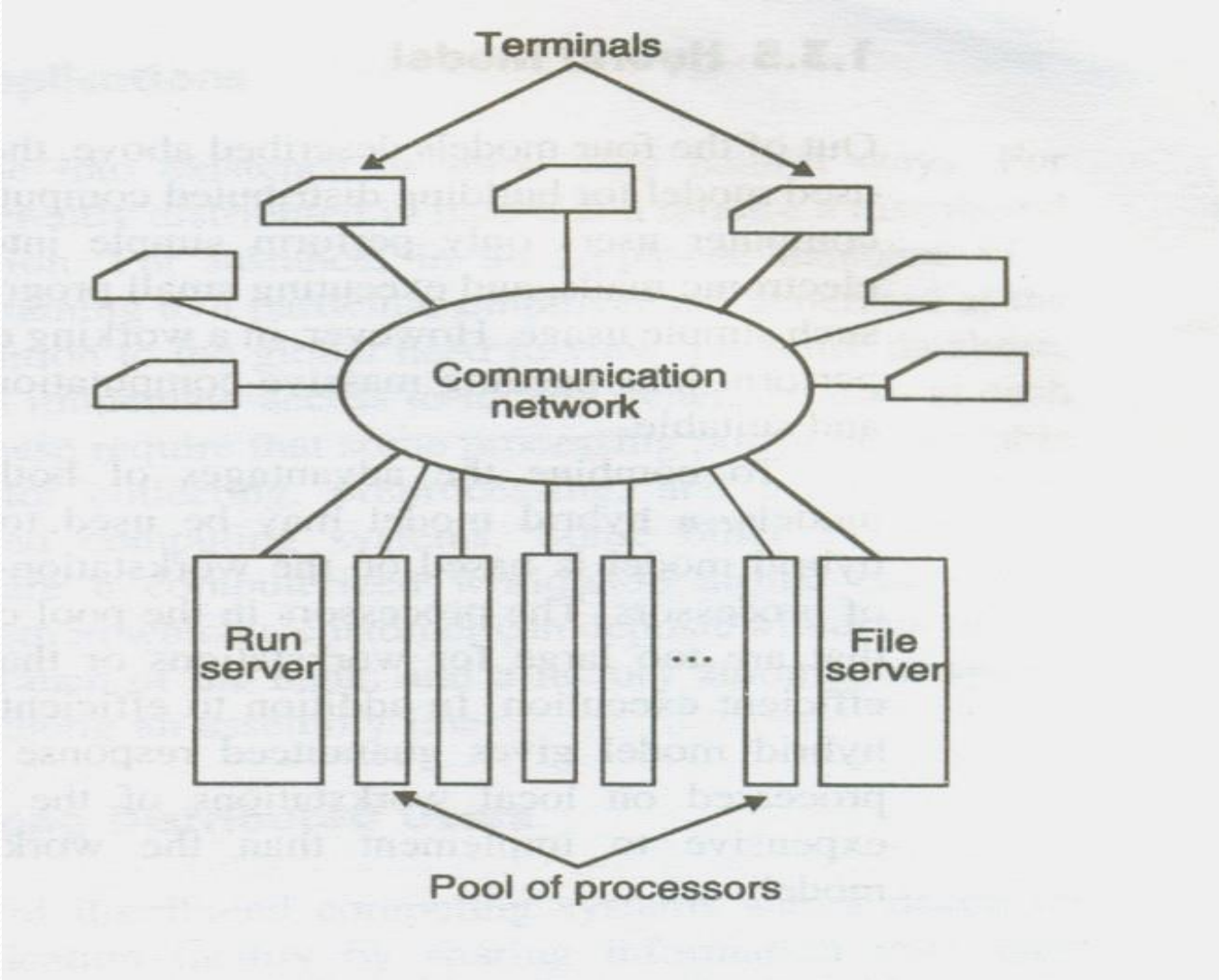
Workstation Model



Workstation Server Model



Processor pooled Model



Issues in Distributed OS

- Global Knowledge
- Naming
- Scalability
- Compatibility
  - Binary Level
  - Execution Level
  - Protocol Level
- Process Synchronization
- Resource Management
  - Data Migration
  - Computational Migration
  - Distributed Scheduling
- Security
- Structuring
  - Monolithic Kernel
  - Collective Kernel Structure
  - Object Oriented Operating Systems
- Client Server Computing Model

Global Knowledge

- No Global Memory
- No Global Clock
- Unpredictable Message Delays

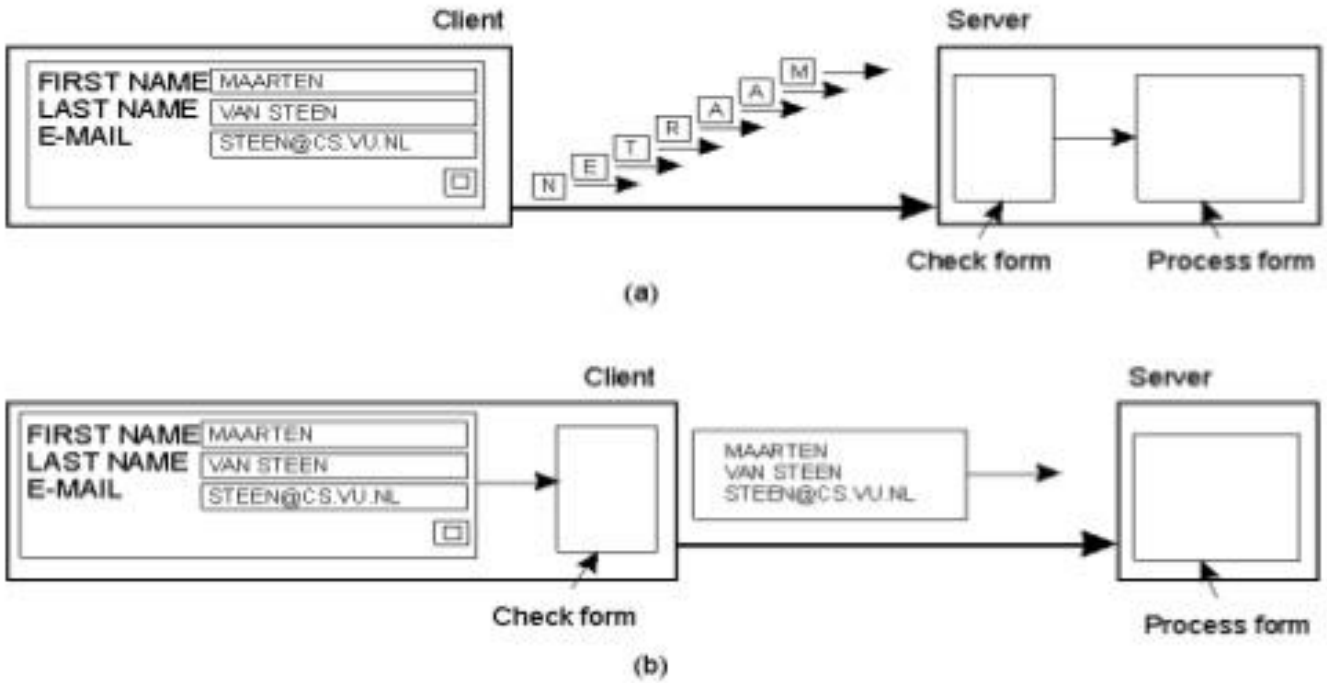
Naming

- Name refers to objects [Files, Computers etc]
- Name Service Maps logical name to physical address
- Techniques
- LookUp Tables [Directories]
- Algorithmic
- Combination of above two

Scalability

- Grow with time
- Scaling Dimensions – Size, Geographically & Administratively
- Techniques – Hiding Communication Latencies, Distribution & Caching

Scaling Techniques (1)
Hide Communication Latencies



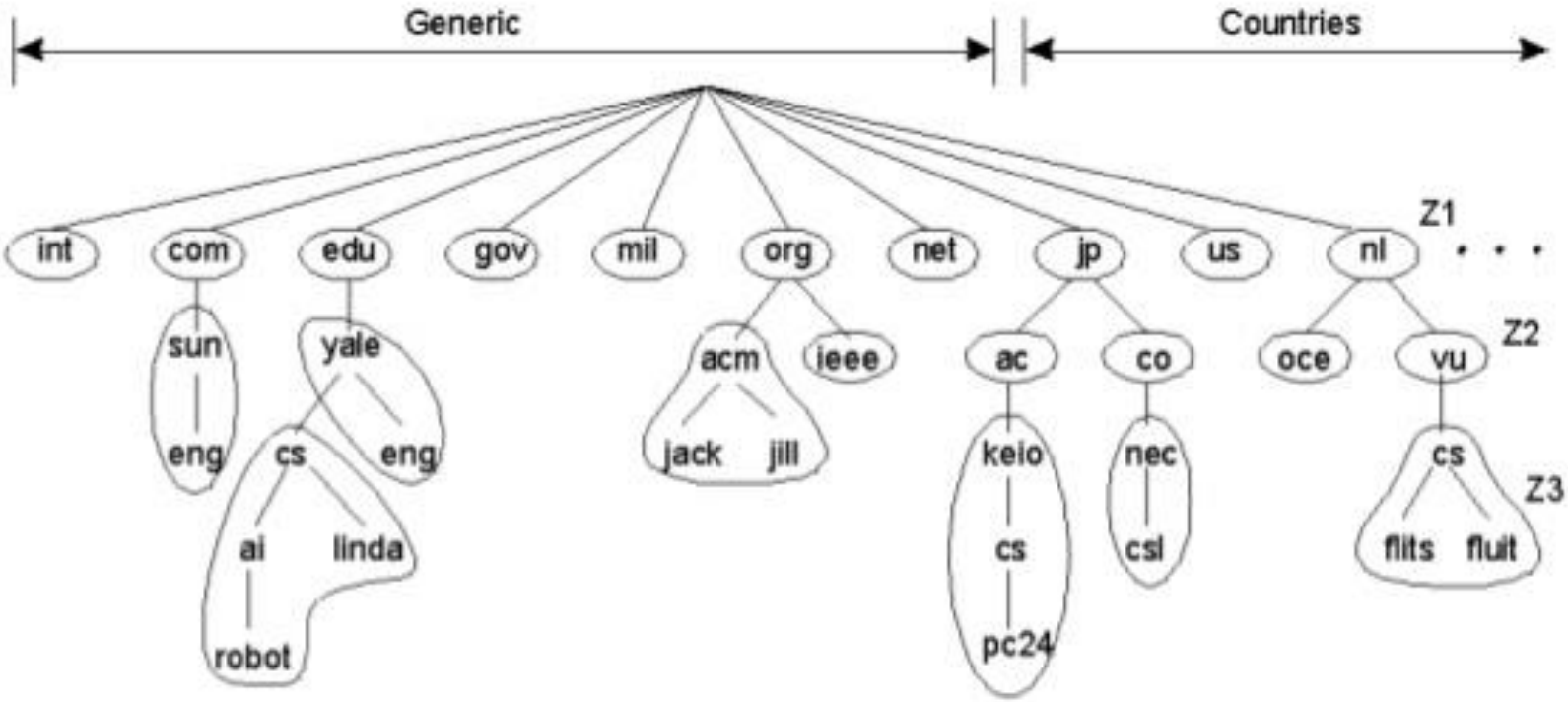
The difference between letting:

a) a server or

b) a client check forms as they are being filled

Scalability cont....

Scaling Techniques (2)
Distribution



An example of dividing the DNS name space into zones.

Scalability cont....

Scaling Techniques(3) Replication

- Replicate components across the distributed system
- Replication increases availability , balancing load distribution
- Consistency problem has to be handled

Compatibility

- Interoperability among resources in system
- Levels of Compatibility – Binary Level, Execution Level & Protocol Level

Process Synchronization

- Difficult because of unavailability of shared memory
- Mutual Exclusion Problem

Resource Management

- Make both local and remote resources available
- Specific Location of resource should be hidden from user
- Techniques
- Data Migration [DFS, DSM]
- Computation Migration [RPC]
- Distributed Scheduling [Load Balancing]

Security

- Authentication – a entity is what it claims to be
- Authorization – what privileges an entity has and making only those privileges available

Structuring

- Techniques
- Monolithic Kernel
- Collective Kernel [Microkernel based , Mach, V- Kernel, Chorus and Galaxy]
- Object Oriented OS [Services are implemented as objects, Eden, Choices, x-kernel, Medusa, Clouds, Amoeba & Muse]
- Client-Server Computing Model [Processes are categorized as servers and clients]

Communication Models

- High level constructs [Helps the program in using underlying communication network]
- Two Types of Communication Models
  - Message passing
  - Remote Procedure Calls

Message Passing

- Two basic communication primitives
- SEND(a,b) , a→ Message , b → Destination
- RECEIVE(c,d), c → Source , d → Buffer for storing the message
- Client-Server Computation Model
- Client sends Message to server and waits
- Server replies after computation

Design Issues

- Blocking vs Non blocking primitives
- Non blocking
  - SEND primitive return the control to the user process as soon as the message is copied from user buffer to kernel buffer
  - Advantage : Programs have maximum flexibility in performing computation and communication in any order
  - Drawback : Programming becomes tricky and difficult
- Blocking
  - SEND primitive does not return the control to the user process until message has been sent or acknowledgement has been received
  - Advantage : Program's behaviour is predictable
  - Drawback :Lack of flexibility in programming

Design Issues cont..

- Synchronous vs Asynchronous Primitives
- Synchronous
  - SEND primitive is blocked until corresponding RECEIVE primitive is executed at the target computer
- Asynchronous
  - Messages are buffered
  - SEND primitive does not block even if there is no corresponding execution of the RECEIVE primitive
  - The corresponding RECEIVE primitive can be either blocking or non-blocking

Details to be handled in Message Passing

- Pairing of Response with Requests
- Data Representation
- Sender should know the address of Remote machine
- Communication and System failures

Remote Procedure Call (RPC)

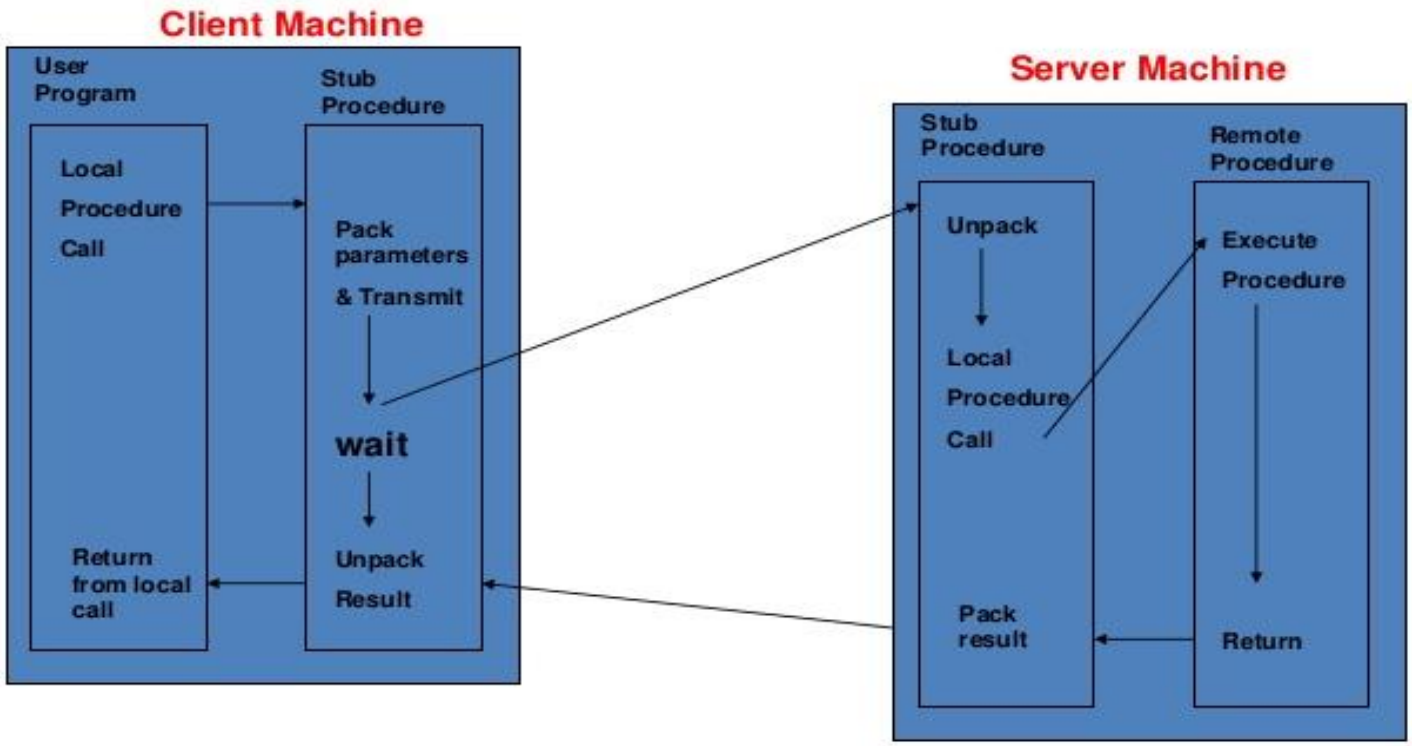
- RPC is an interaction between a client and a server
- Client invokes procedure on sever
- Server executes the procedure and pass the result back to client
- Calling process is suspended and proceeds only after getting the result from server

RPC Design issues

- Structure
- Binding
- Parameter and Result Passing
- Error handling, semantics and Correctness

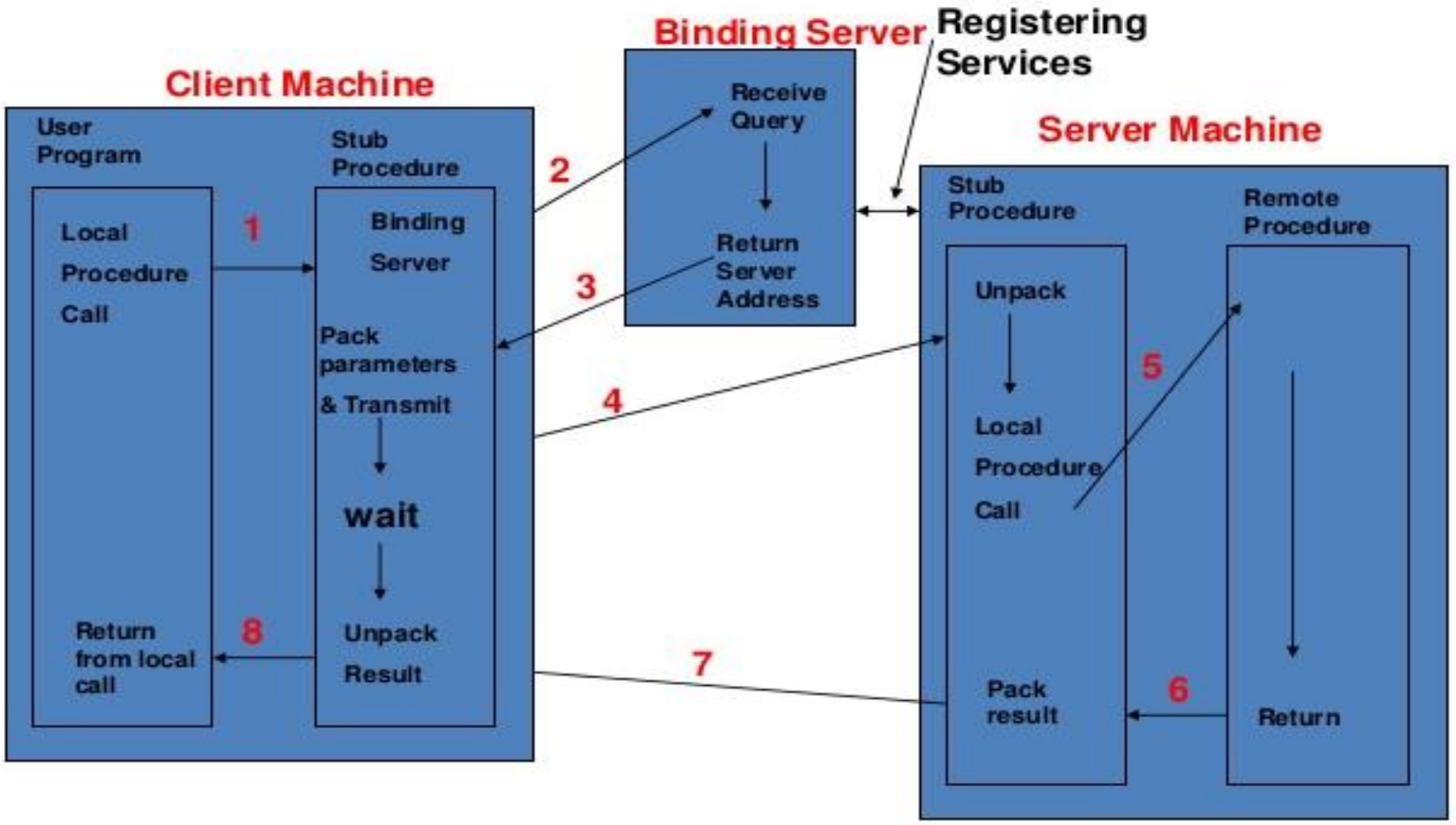
Structure

❑ RPC mechanism is based upon stub procedures.



Binding

- Determines remote procedure and machine on which it will be executed
- Check compatibility of the parameters passed
- Use Binding Server



Parameter and Result Passing

- Stub Procedures Convert Parameters & Result to appropriate form
- Pack parameters into a buffer
- Receiver Stub Unpacks the parameters
- Expensive if done on every call
- Send Parameters along with code that helps to identify format so that receiver can do conversion
- Alternatively Each data type may have a standard format. Sender will convert data to standard format and receiver will convert from standard format to its local representation
- Passing Parameters by Reference

Error handling, Semantics and Correctness

- RPC may fail either due to computer or communication failure
- If the remote server is slow the program invoking remote procedure may call it twice.
- If client crashes after sending RPC message
- If client recovers quickly after crash and reissues RPC
- Orphan Execution of Remote procedures
- RPC Semantics
- At least once semantics
- Exactly Once
- At most once

Correctness Condition

- Given by Panzieri & Srivastava
- Let Ci denote call made by machine & Wi represents corresponding computation
- If C2 happened after C1 ($C1 \rightarrow C2$) & Computations W2 & W1 share the same data, then to be correct in the presence of failures RPC should satisfy
- $C1 \rightarrow C2$ implies $W1 \rightarrow W2$

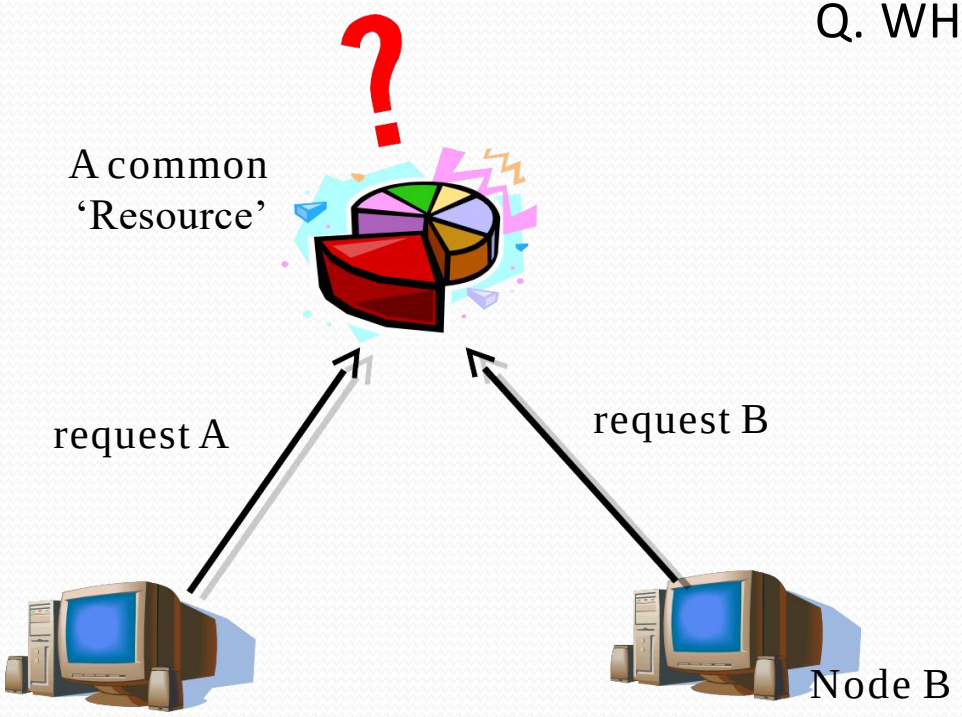
Clock synchronization

What is a Real-time system ?

A Real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations but also on the time when the results are produced.

Real-time systems usually are in strong interaction with their physical environment. They receive data, process it, and return results in right time. E.g. A Distributed real time system.

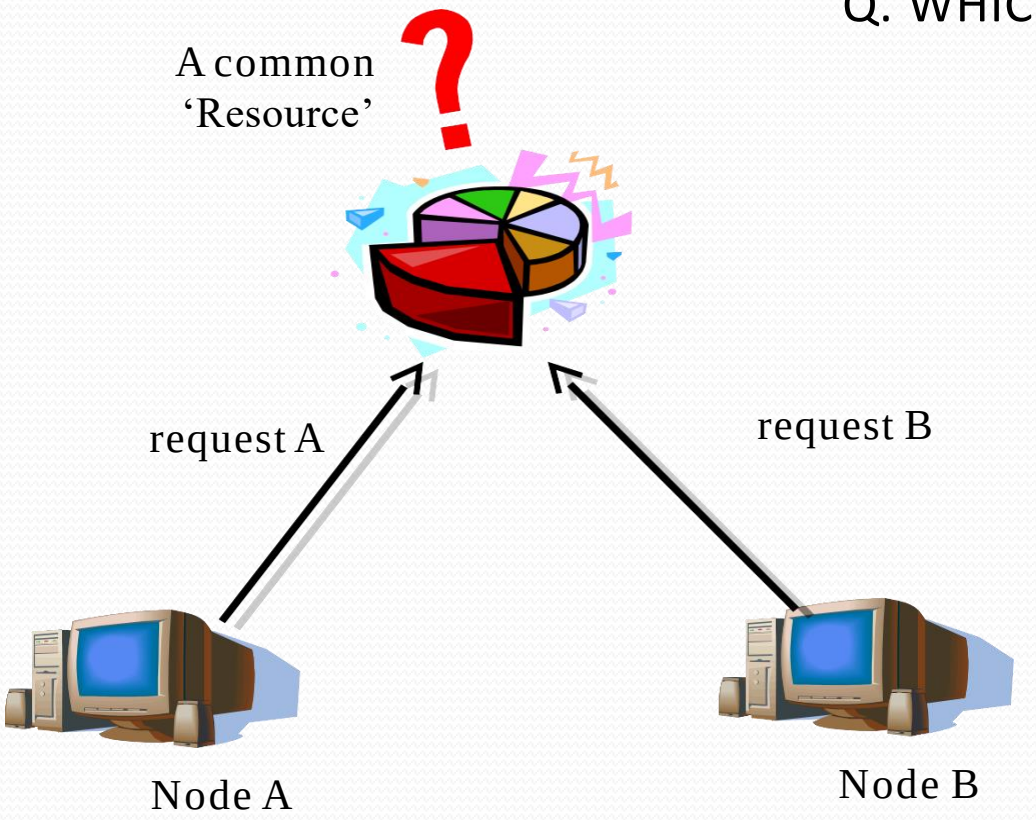
Q. WHICH REQUEST WAS MADE FIRST?



A Distributed System

Problem !

Q. WHICH REQUEST WAS MADE FIRST?

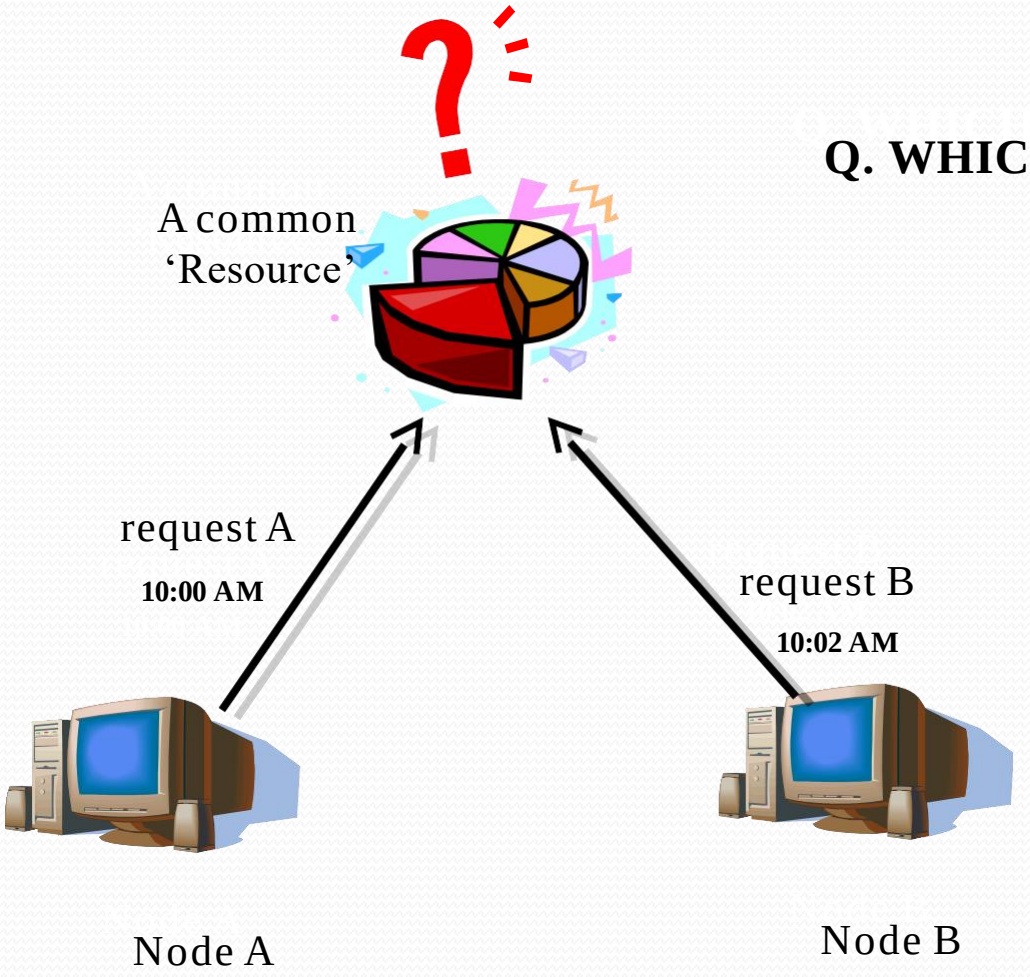


A Distributed System

Solution

A Global Clock ??

Global Synchronization?



A Distributed System

Q. WHICH REQUEST WAS MADE FIRST?

Solution

Individual Clocks?

Are individual clocks accurate, precise?

One clock might run faster/slower?

Drifting of clock

- A quartz crystal oscillates at well defined frequency and oscillations are counted (by hardware) in a register.
- After a certain number of oscillations, an interrupt is generated; this is the clock tick.
- At each clock tick, the computer clock is incremented by software.

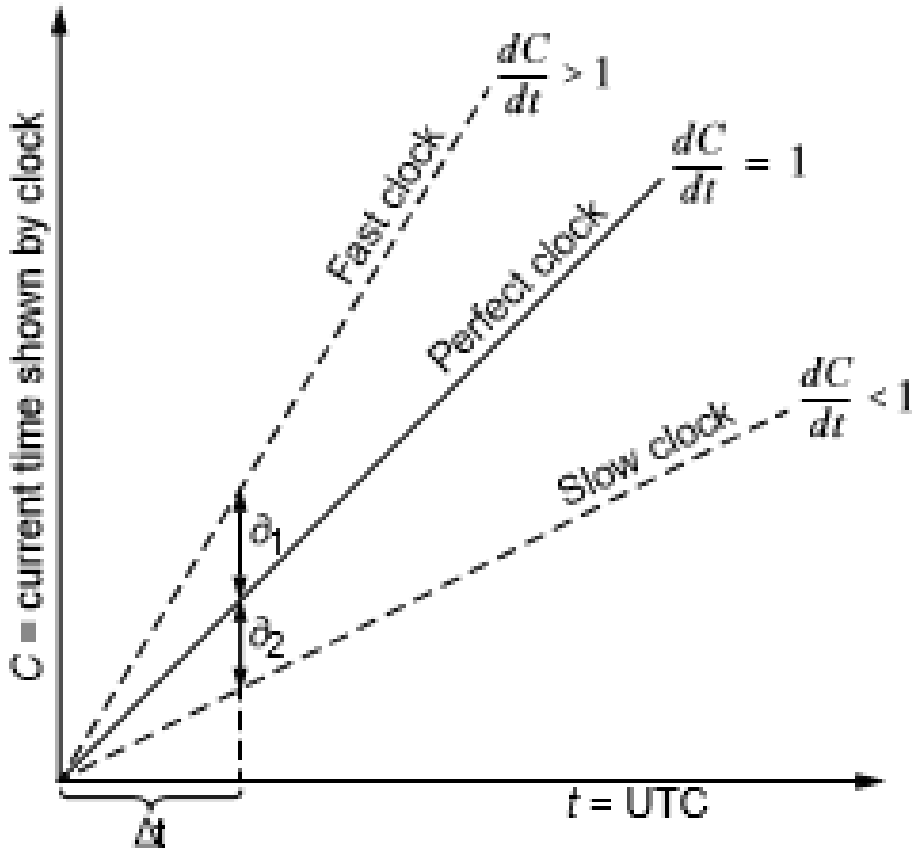
The Problems:

1. Crystals cannot be tuned perfectly. Temperature and other external factors can also influence their frequency.

Clock drift: the computer clock differs from the real time

2. Two crystals are neveridentical.

Clock skew: the computer clocks on different processors of the distributed system show different time.



∂_1: drift of first clock after Δt.

∂_2: drift of second clock after Δt.

$\partial_1 + \partial_2$: skew between clocks after Δt.

Synchronized Distributed clock

- Time driven systems: in statically scheduled systems activities are started at "precise" times in different points of the distributed system.
- Time stamps: certain events or messages are associated with a time stamp showing the actual time when they have been produced; certain decisions in the system are based on the "exact" time of the event or event ordering.
- Calculating the duration of activities: if such an activity starts on one processor and finishes on another (e.g. transmitting a message), calculating the duration needs clocks to be synchronized.

Physical Clock

These are electronic devices that count oscillations occurring in a crystal.

- Also called timer, usually a quartz crystal, oscillating at a well defined frequency.
- Timer is associated with two registers: A Counter and a Holding Register, counter decreasing one at each oscillations.
- Synchronizing Physical Clocks: done by two methods:
  - External Synchronization
  - Internal Synchronization

The Universal Time

- The standard for measurement of time intervals:
- International Atomic Time (TAI): It defines the standard second and is based on atomic oscillators.
- Coordinated Universal Time (UTC): is based on TAI, but is kept in step with astronomical time (by occasionally inserting or deleting a "leap
- second").
- UTC signals are broadcast from satellites and land based radio stations.

External Synchronization

Synchronization with a time source external to the distributed systems, such as UTC broadcasting system.

One processor in the system (possibly several) is equipped with UTC receivers (time providers).

By external synchronization the system is kept synchronous with the "real time".

This allows to exchange consistently timing information with other systems and with users.

Internal Synchronization

Synchronization among processors of the system.

It is needed in order to keep a consistent view of time over the system.

few processors synchronize externally and the whole system is kept consistent by internal synchronization.

Sometimes only internal synchronization is performed (we don't care for the drift from external/ real time).

Logical Clocks

ssume no central time source –

Each system maintains its own local clock .

No total ordering of events .

Allow to get global ordering on events.

Assign sequence numbers to messages –

All cooperating processes can agree on order of event.

Lamport Timestamps

It is used to provide a partial ordering of events with minimal overhead.

It is used to synchronize the logical clock. It follows some simple rules:

A process increments its counter before each event in that process i.e. Clock must tick once between every two events.

When a process sends a message, it includes its timestamp with the message.

On receiving a message, the receiver process sets its counter to be the maximum of the message counter and increments its own counter .

‘Happened Before’ Relation

a ‘Happened Before’ b : a→b

- If a and b are events in the same process, and a
- comes before b, then a → b.
- If
- a :message sent
- b : receipt of the same message then a → b.
- Transitive: If a → b and b → c then a → c.
- Two distinct events a and b are said to be concurrent if a -/->b and b -/->a

Clock Synchronization Algorithms

Centralized Algorithms

- There exists one particular node, the so called Time server node and clock time of this node is used as reference.
- Passive time server: The other machines ask periodically
- for the time. The goal is to keep the clocks of all other nodes synchronized with the time server.
- Active time server: the Time server is active, broadcasting other machines periodically about the time.

Disadvantages:

- Single point of failure i.e. less reliable.
- Propagation delay is unpredictable.

Distributed Algorithms

- There is no particular time server.
- The processors periodically reach an agreement on the clock value by averaging the time of neighbours clock and its local clock.
- This can be used if no UTC receiver exists (no external synchronization is needed). Only internal synchronization is performed.
- Processes can run on different machines and no global clock to judge which event happens first.

Cristian's Algorithm

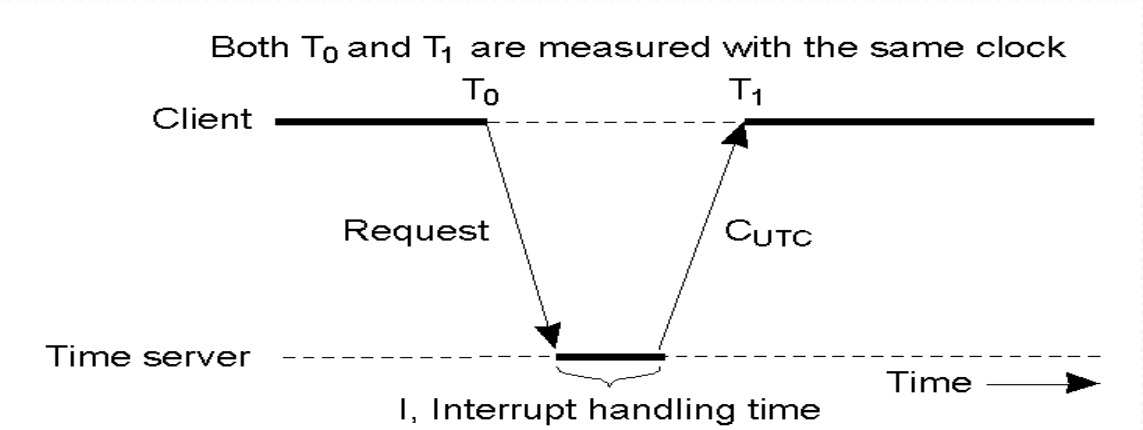
Cristian's Algorithm is centralized algorithm.

The simplest algorithm for setting time, it issues a Remote Procedure Call to time server and obtain the time.

A machine sends a request to time server in "d/2" seconds, where d=max difference between a clock and UTC.

The time server sends a reply with current UTC when receives the request.

The machine measures the time delay between time server sending the message and machine receiving it. Then it uses the measure to adjust the clock.



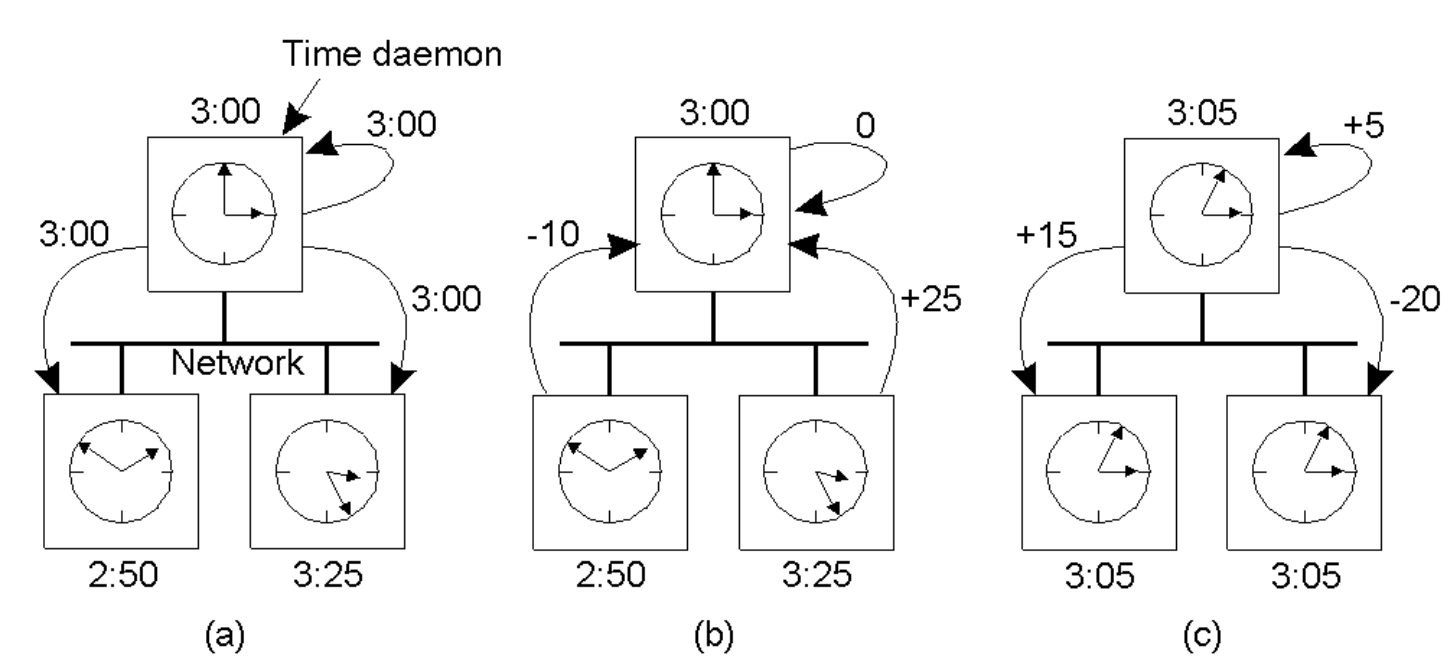
The best estimate of message propagation time= (To + T1)/2 The new time can be set to the time returned by server plus time that elapsed since server generated the timestamp:

$$Tnew = Tserver + (To + T1)/2$$

Berkeley Algorithm

- It is also a Centralized algorithm and its Time server is an Active Machine.
- The server polls each machine periodically, asking it for the time.
- When all the results are in, the master computes the
- average time.
- Instead of sending the updated time back to slaves, which would introduce further uncertainty due to network delays, it sends each machine the offset by
- which its clock needs adjustment.
- If master machine fails, any other slave could be elected to take over.

Berkeley Algorithm



- The time daemon sends synchronization query to other machines in group.
- The machines sends timestamps as a response to query.
- The Server averages the three timestamps and tells everyone how to adjust their clock by sending offsets.

Mutual exclusion
Election algorithms

Process Synchronization

Techniques to coordinate execution among processes

- One process may have to wait for another
- Shared resource (e.g. critical section) may require exclusive access

Centralized Systems

Mutual exclusion via:

- Test & set in hardware
- Semaphores
- Messages
- Condition variables

Distributed Mutual Exclusion

Assume there is agreement on how a resource is identified

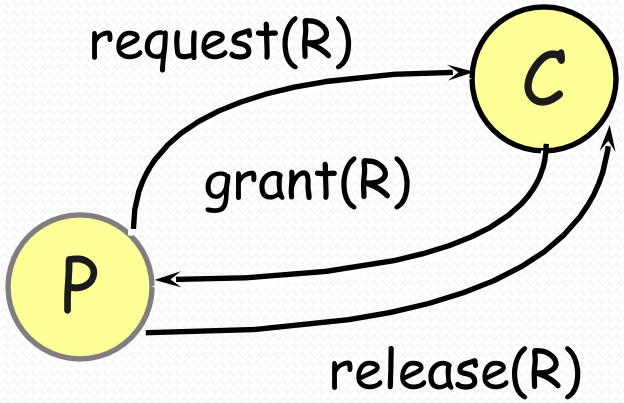
–Pass identifier with requests

Create an algorithm to allow a process to obtain exclusive access to a resource.

Centralized algorithm

- Mimic single processor system
- One process elected as coordinator

- Request resource
- Wait for response
- Receive grant
- access resource
- Release resource



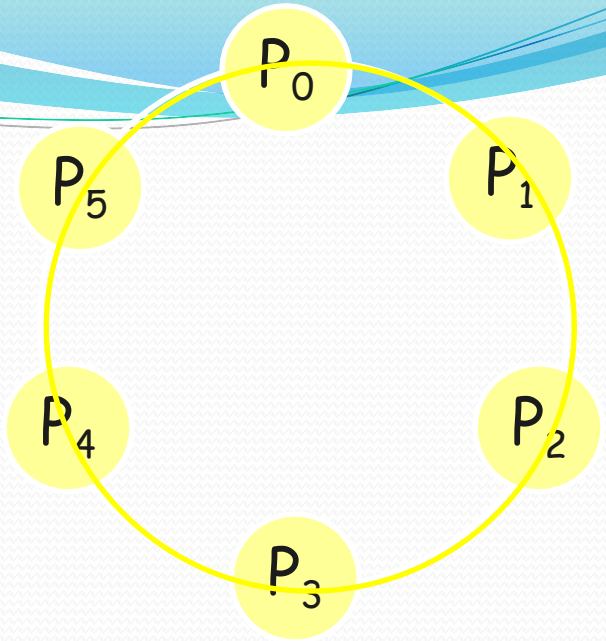
Benefits

- Fair
  - All requests processed in order
- Easy to implement, understand, verify

Problems

- Process cannot distinguish being blocked from a dead coordinator
- Centralized server can be a bottleneck

Token Ring algorithm



Assume known group of processes

- Some ordering can be imposed on group
- Construct logical ring in software
- Process communicates with neighbour

Initialization

- Process o gets token for resource R

Token circulates around ring

- From Pi to P(i+1)mod N

When process acquires token

- Checks to see if it needs to enter critical section
- If no, send ring to neighbor
- If yes, access resource

Hold token until done

Only one process at a time has token

- Mutual exclusion guaranteed

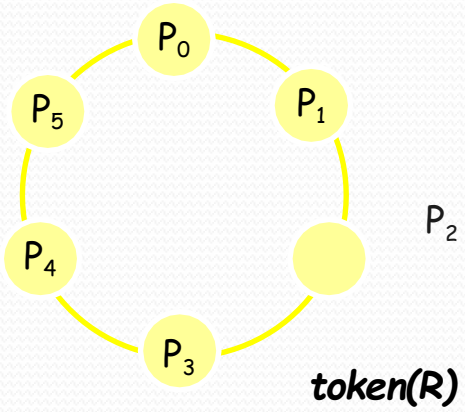
Order well-defined

- Starvation cannot occur

If token is lost (e.g. process died)

- It will have to be regenerated

Does not guarantee FIFO order



Ricart & Agrawala algorithm

- Distributed algorithm using reliable multicast and logical clocks
- Process wants to enter critical section:
  - Compose message containing:
    - Identifier (machine ID, process ID)
    - Name of resource
    - Timestamp (totally-ordered Lamport)
  - Send request to all processes in group
  - Wait until everyone gives permission
  - Enter critical section / use resource

Ricart & Agrawala algorithm

- When process receives request:
  - If receiver not interested:

    Send OK to sender
  - If receiver is in critical section

    Do not reply; add request to queue
  - If receiver just sent a request as well:

    Compare timestamps: received & sent messages

    Earliest wins

    If receiver is loser, send OK

    If receiver is winner, do not reply, queue
  - When done with critical section

    Send OK to all queued requests

- N points of failure
- A lot of messaging traffic
- Demonstrates that a fully distributed algorithm is possible

Lamport's Mutual Exclusion

Each process maintains request queue

- Contains mutual exclusion requests

- Requesting critical section:
  - Process Pi sends request(i, Ti) to all nodes
  - Places request on its own queue

Lamport time

- When a process Pj receives a request, it returns a timestamped ack

Entering critical section (accessing resource):

- Pi received a message (ack or release) from every other process with a timestamp larger than Ti
- Pi's request has the earliest timestamp in its queue

Difference from Ricart-Agrawala:

- Everyone responds … always - no hold-back
- Process decides to go based on whether its request is the earliest in its queue

Releasing critical section:

- Remove request from its own queue
- Send a timestamped release message
- When a process receives a release message
  - Removes request for that process from its queue
  - This may cause its own entry have the earliest timestamp in the queue, enabling it to access the critical section

Election Algorithms

Elections

- Need one process to act as coordinator
- Processes have no distinguishing characteristics
- Each process can obtain a unique ID

Bully algorithm

- Select process with largest ID as coordinator
- When process P detects dead coordinator:
  - Send election message to all processes with higher IDs.
    - If nobody responds, P wins and takes over.
    - If any process responds, P's job is done.
  - Optional: Let all nodes with lower IDs know an election is taking place.
- If process receives an election message
  - Send OK message back
  - Hold election (unless it is already holding one)
- A process announces victory by sending all processes a message telling them that it is the new coordinator
- If a dead process recovers, it holds an election to find the coordinator.

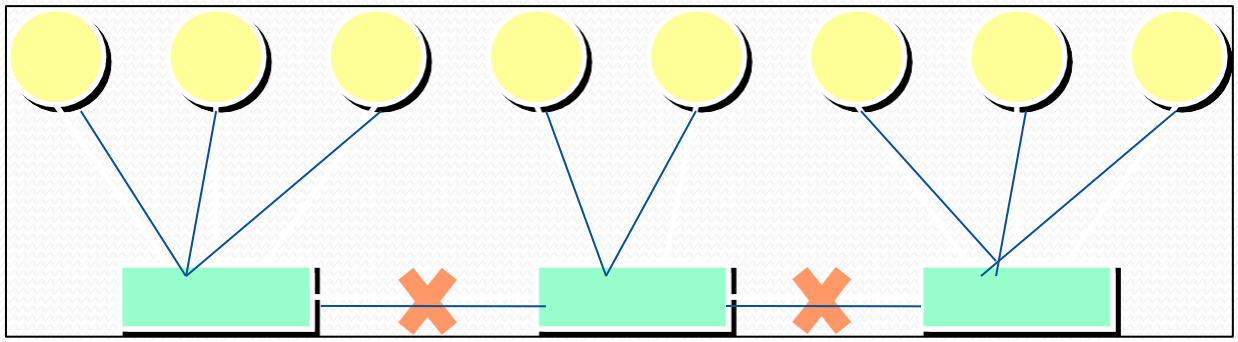
Ring algorithm

- Ring arrangement of processes
- If any process detects failure of coordinator
  - Construct election message with process ID and send to next process
  - If successor is down, skip over
  - Repeat until a running process is located
- Upon receiving an election message
  - Process forwards the message, adding its process ID to the body
- Eventually message returns to originator
  - Process sees its ID on list
- Circulates (or multicasts) a coordinator message announcing coordinator
  - E.g. lowest numbered process

Problems with elections

Network segmentation

- **Split brain**



Rely on alternate communication mechanism

- Redundant network, shared disk, serial line, SCSI

Deadlock detection

Deadlocks

Process is blocked on resource that will never be released.

Deadlocks *waste resources*

Deadlocks are rare:

Many systems ignore them

Resolved by explicit user intervention

Critical in many real-time applications

May cause damage, endanger life

Reusable Resources

- Number of units is "constant"
- Unit is either free or allocated; no sharing
- Process requests, acquires, releases units

Examples: memory, devices, files, tables

Consumable Resources

- Number of units varies at runtime
- Process may create new units
- Process may consume units

Examples: messages, signals

Examples of Deadlocks

```
p1: ...
    open(f1,w);
    open(f2,w);
    ...
```

```
p2: ...
      open(f2,w);
      open(f1,w);
       ...
```

Deadlock when executed concurrently

```
p1: if (C) send(p2,m);
      while(1){...
         recv(p2,m);
         send(p2,m);
         ... }
```

```
p2: ...
    while(1){...
       recv(p1,m);
       send(p1,m);
        ... }
```

Deadlock when C not true

*Deadlock, Livelock ,*Starvation

- Deadlock: Processes are blocked
- Livelock: Processes run but make no progress
- Both deadlock and livelock lead to starvation Starvation may have other causes
- ML scheduling where one queue is never empty
- Memory requests: unbounded stream of 100MB requests may starve a 200MB request

Approaches to Deadlock problem

1. Detection and Recovery
   Allow deadlock to happen and eliminate it
2. Avoidance (dynamic)
   Runtime checks disallow allocations that might lead to deadlocks
3. Prevention (static)
   Restrict type of request and acquisition to make deadlock impossible

System Model for Deadlock Detection,Avoidance, etc.

Assumptions

When a process requests a resource, either the request is fully granted or the process blocks

No partial allocation

A process can only release resources that it holds

Resource graph:

Vertices are processes, resources, resource units

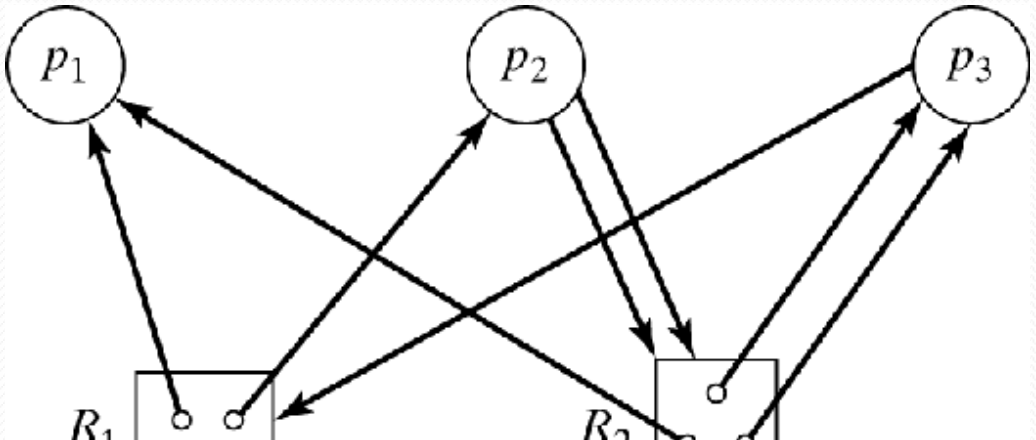
Edges (directed) represent requests and allocations of resources

Process = Circle

Resource = Rectangle with small circles for each unit

Request = Edge from process to resource class

Allocation = Edge from resource unit to process



System Model: State Transitions

Request: Create new request edge $p_i \rightarrow R_j$

p_i has no outstanding requests

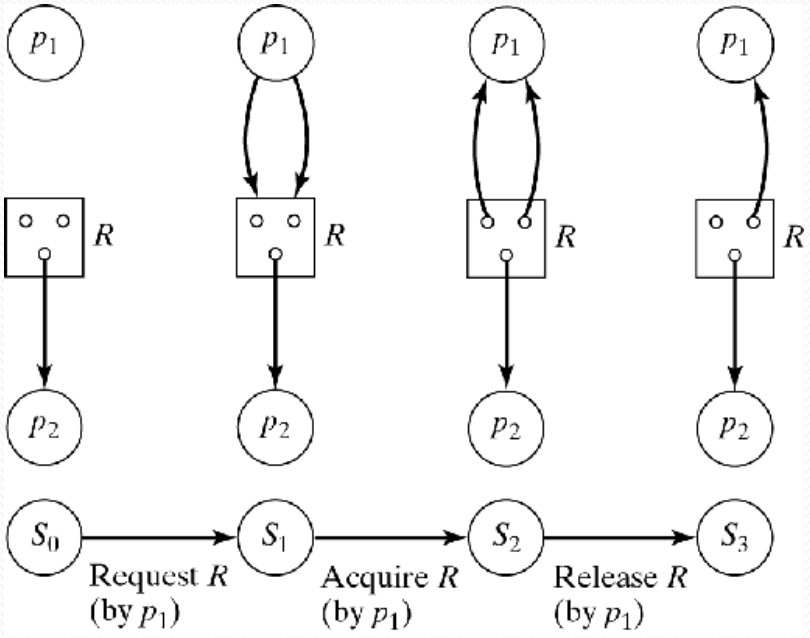
number of edges between p_i and R_j cannot exceed total units of R_j

Acquisition: Reverse request edge to $p_i \leftarrow R_j$

All requests of p_i are satisfiable

p_i has no outstanding requests

Release: Remove edge $p_i \leftarrow R_j$

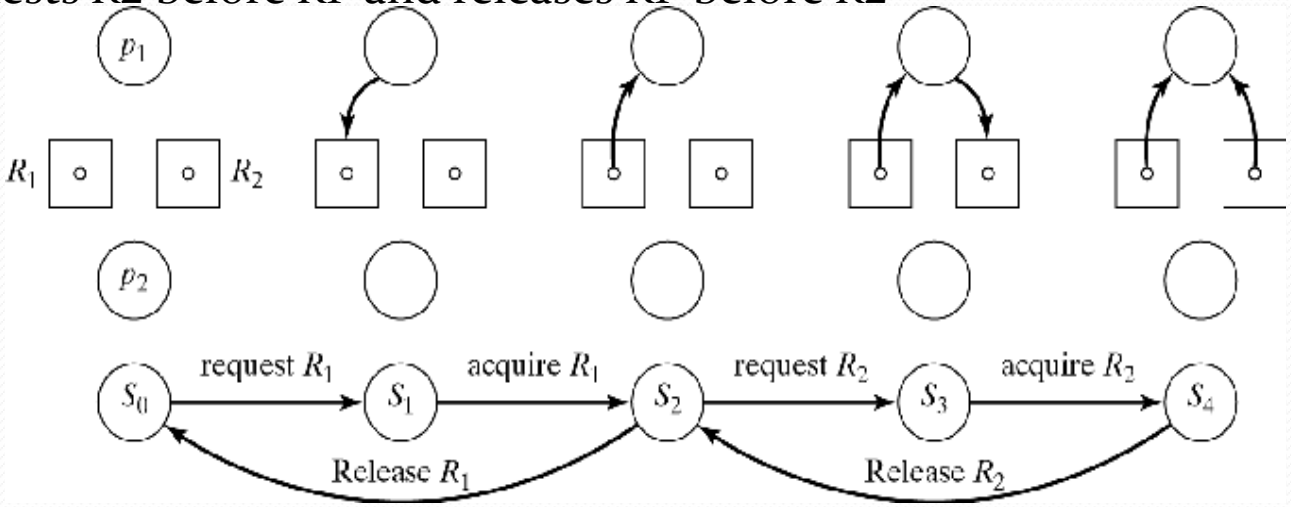


System Model

- A process is blocked in state S if it cannot request, acquire, or release any resource.
- A process is deadlocked in state S if it is currently blocked now and remains blocked in all states reachable from state S
- A state is a deadlock state if it contains a deadlocked process.
- State S is a safe stateif no deadlock state can be reached from S by any sequence of request, acquire, release.
- Example:

2 processes p1 , p2; 2 resources R1, R2,

- p1 and p2 both need R1 and R2
- p1 requests R1 before R2 and releases R2 before R1
- p2 requests R2 before R1 and releases R1 before R2

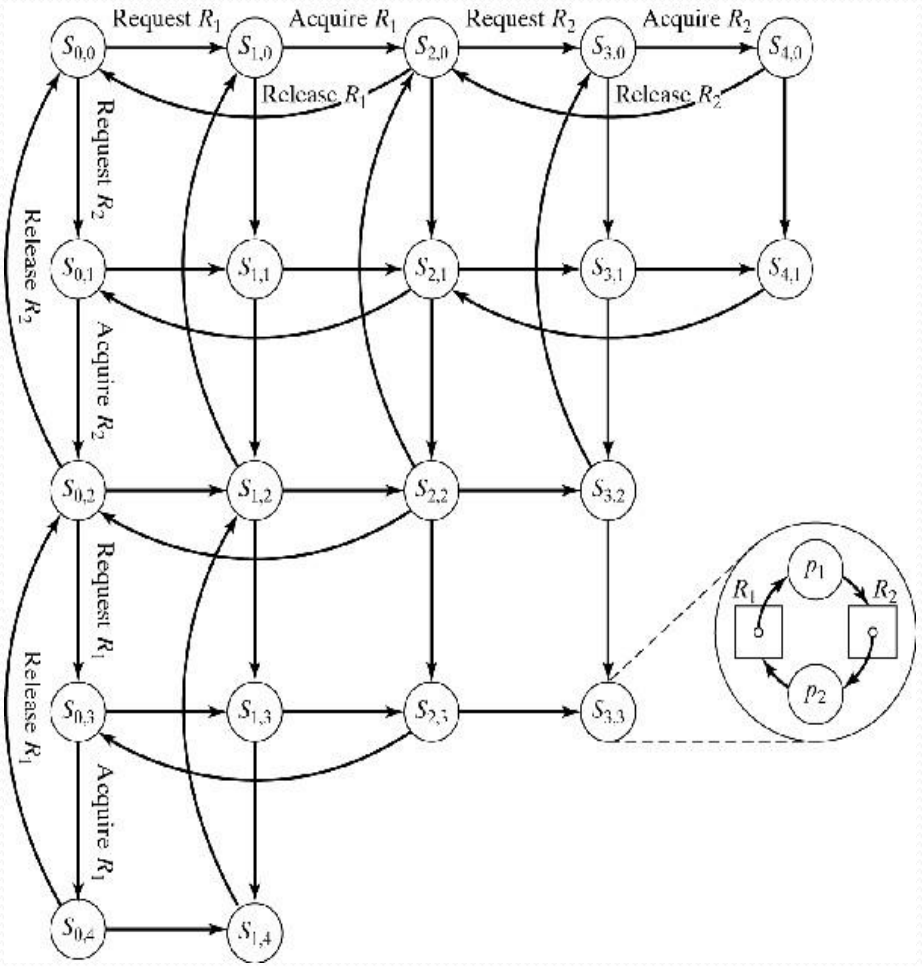


Example

p_1 and p_2 both need R_1 and R_2

p_1 requests R_1 before R_2 and releases R_2 before R_1

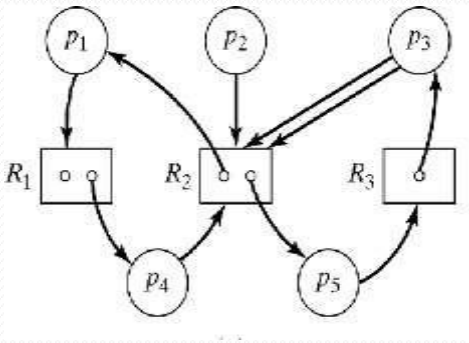
p_2 requests R_2 before R_1 and releases R_1 before R_2

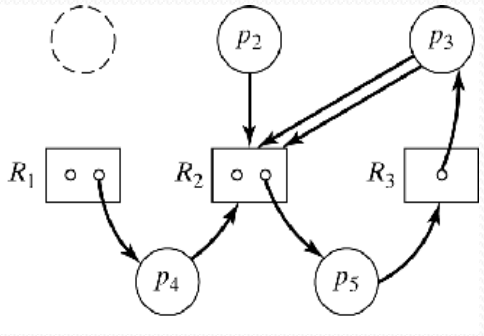


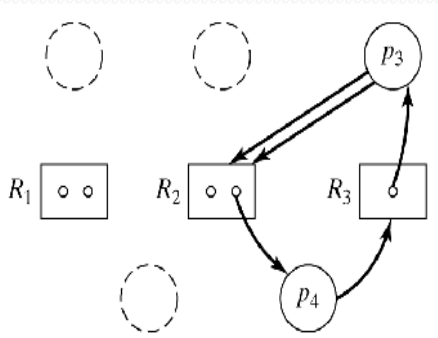
Deadlock Detection

Graph Reduction: Repeat the following

Select unblocked process p
Remove p and all request and allocation edges
Deadlock ⇔ Graph not completely reducible.
All reduction sequences lead to the same result.







Special Cases of Detection

Testing for whether a specific process ***p*** is deadlocked:

Reduce until ***p*** is removed or graph irreducible

Continuous detection:

1. Current state not deadlocked
2. Next state ***T*** deadlocked only if:
   a. Operation was a request by ***p*** and ***p*** is deadlocked in ***T***
3. Try to reduce ***T*** by ***p***

Special Cases of Detection

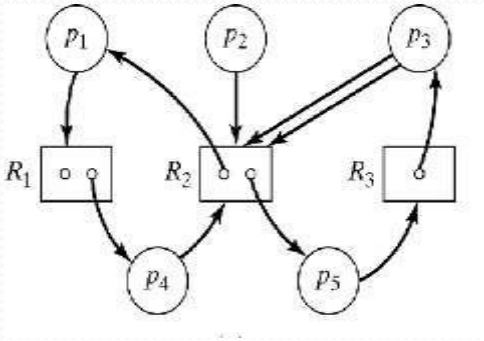
Immediate allocations

All satisfiable requests granted immediately

Expedient state: state with no satisfiable request edges

If all requests are granted immediately, all states are expedient.

Not expedient (p1->R1)



Special Cases of Detection

Immediate allocations, continued.

Knot : A set K of nodes such that

- Every node in K reachable from any other node in K
- No outgoing edges from any node in K

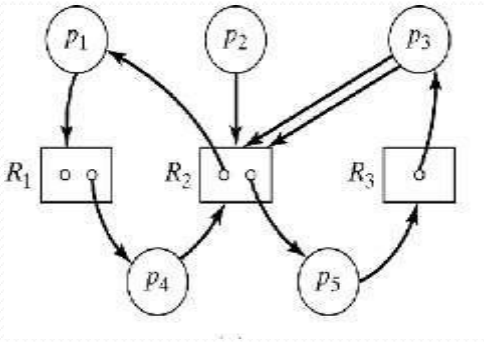
Knot in expedient state $\Rightarrow$ Deadlock :

Reason:

- All processes in K must have outstanding requests
- Expedient state means requests not satisfiable

(Remove R2->p1: knot R2,p3,R3,p5)

(Reverse edge p1->R1): expedient state



Special Cases of Detection

For single-unit resources, cycle $\Rightarrow$ deadlock

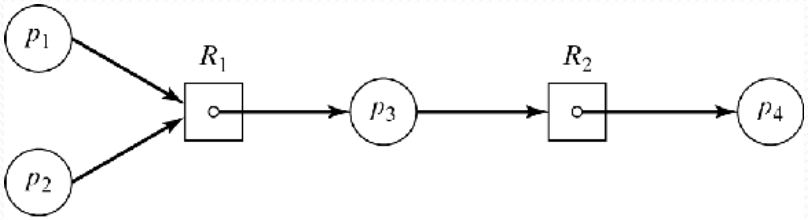
Every ***p*** must have a request edge to ***R***

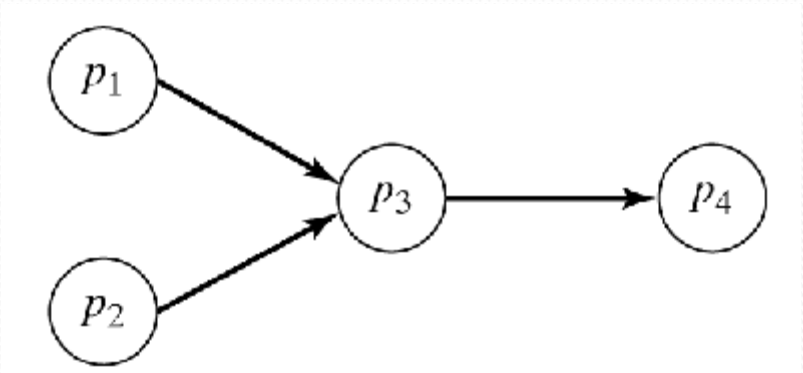
Every ***R*** must have an allocation edge to ***p***

R is not available and thus ***p*** is blocked

Wait-For Graph (wfg): Show only processes

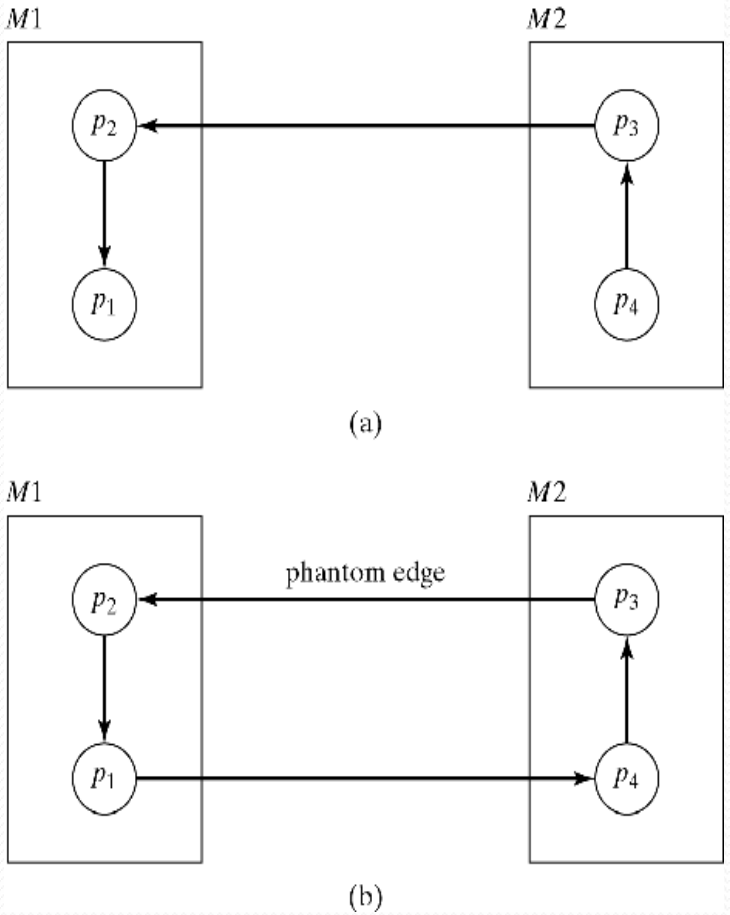
Replace $p_1 \rightarrow R \rightarrow p_2$ by $p_1 \rightarrow p_2$: p_1 waits for p_2





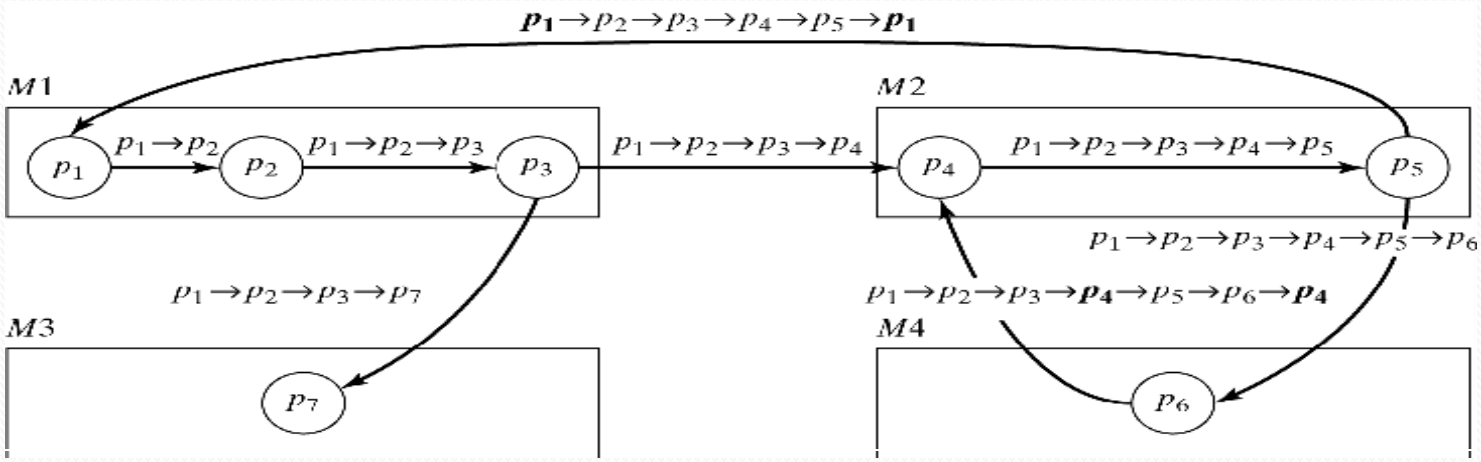
Deadlock detection in Distributed Systems

- Central Coordinator (CC)
  - Each machine maintains a local wfg
  - Changes reported to CC
  - CC constructs and analyzes global wfg
- Problems
  - Coordinator is a performance bottleneck
  - Communication delays may cause phantom deadlocks



Detection in Distributed Systems

- Distributed Approach
  - Detect cycles using probes.
  - If process p_i blocked on p_j, it launches probe $p_i \rightarrow p_j$
  - pj sends probe $p_i \rightarrow p_j \rightarrow p_k$ along all request edges, etc.
  - When probe returns to p_i, cycle is detected



Recovery from Deadlock

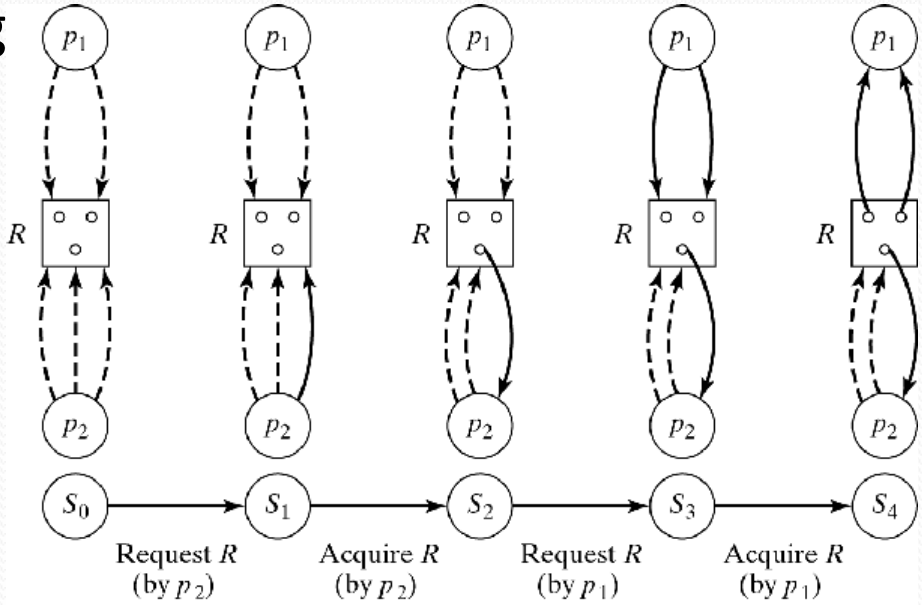
- Process termination
  - Kill all processes involved in deadlock; or
  - Kill one at a time. In what order?
    - By priority: consistent with scheduling
    - By cost of restart: length of recomputation
    - By impact on other processes: CS, producer/consumer
- Resource preemption

  Direct: Temporarily remove resource (e.g., Memory)

  Indirect: Rollback to earlier “checkpoint”

Dynamic Deadlock Avoidance

- Maximum Claim Graph
  - Process indicates *maximum* resources needed
    Potential request edg $p_i \rightarrow R_j$ (dashed)
  - May turn into *real* request edge

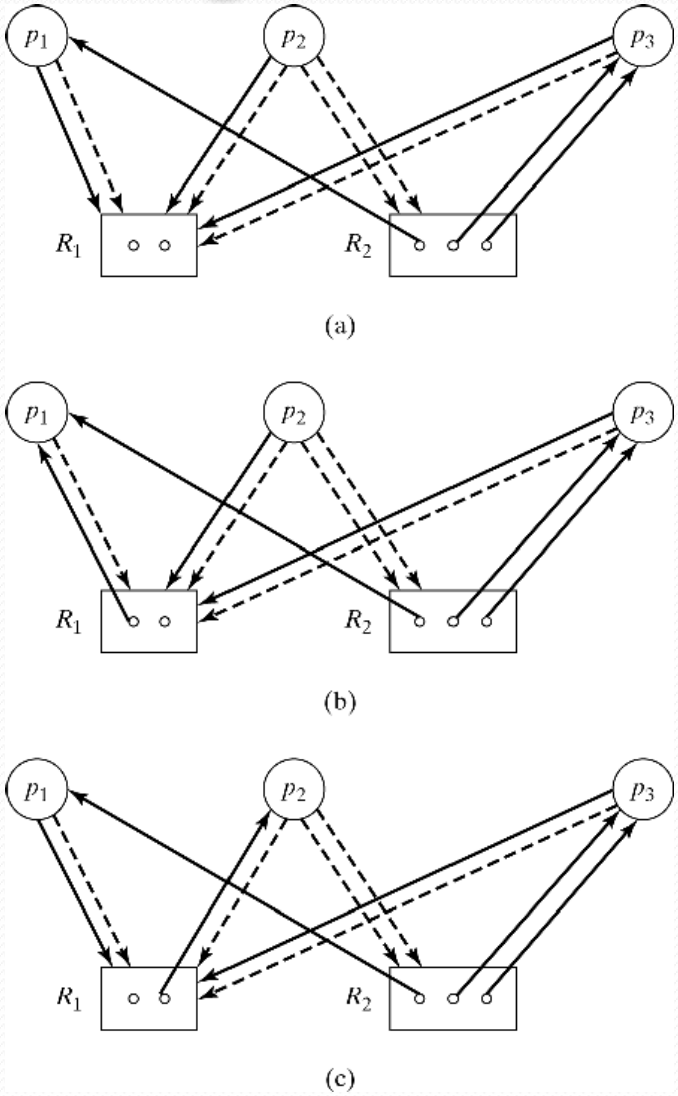


Dynamic Deadlock Avoidance

- Theorem: Prevent acquisitions that do not produce a completely reducible graph
  $\Rightarrow$ All state are safe.
- Banker's algorithm (Dijkstra):
  - Given a satisfiable request, $p \rightarrow R$, temporarily grant request, changing $p \rightarrow R$ to $R \rightarrow p$
  - Try to reduce new claim graph, treating claim edges as actual requests.
  - If new claim graph is completely reducible proceed. If not, reverse temporary acquisition $R \rightarrow p$ back to $p \rightarrow R$
- Analogy with banking: resources correspond to currencies, allocations correspond to loans, maximum claims correspond to credit limits

Example of banker's algorithm

- Claim graph (a). Which requests for R1 can safely be granted?
- If p1's request is granted, resulting claim graph (b) is reducible (p1,p3,p2).
- If p2's request is granted, resulting claim graph (c) is not reducible.
- Exercise: what about p3's request?



Deadlock Prevention

- Deadlock requires the following conditions:
  - Mutual exclusion:
    - Resources not sharable
  - Hold and wait:
    - Process must be holding one resource while requesting another
  - Circular wait:
    - At least 2 processes must be blocked on each other

Deadlock Prevention

- Eliminate mutual exclusion:
  - Not possible in most cases
  - Spooling makes I/O devices sharable
- Eliminate hold-and-wait
  - Request all resources at once
  - Release all resources before a new request
  - Release all resources if current request blocks
- Eliminate circular wait
  - Order all resources: $SEQ(R_i) \neq SEQ(R_j)$
  - Process must request in ascending order

Thank you