Advanced Operating System

Unit – II

Dr.M.Lalli

Dept of CS, Trichy

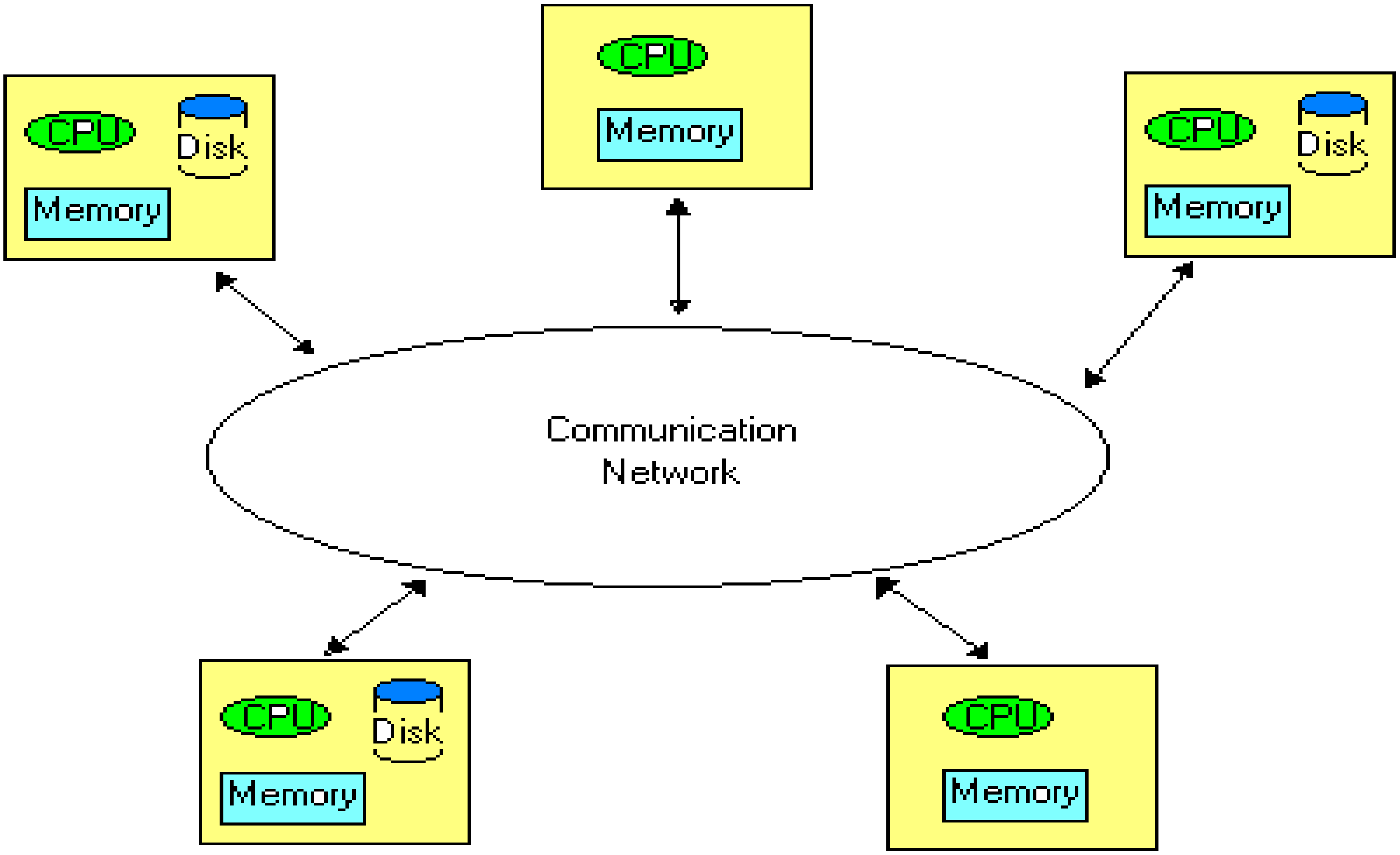
Unit - II

Distributed Operating Systems: System Architectures – Design Issues- Communication Models- Clock Synchronization – Mutual Exclusion – Election Algorithms- distributed Deadlock Detection

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.

Architecture of Distributed OS



What is Distributed Operating Systems?

- ❑It extends the concepts of resource management and user friendly interface for shared memorycomputers a step further, encompassing
  a distributed system consisting of several computing autonomous connected by a communicating computers network.

- ❑A distributed OS is one that looks to its users like an centralized OS but runs on multiple, independent CPUs. The key concept is transparency. In other words, the use of multiple processors should be invisible to the user

Issues in Distributed OS

- ❑ Global Knowledge
- ❑ Naming
- ❑ Scalability
- ❑ Compatibility
- ❑ Process Synchronization
- ❑ Resource Management
- ❑ Security
- ❑ Structuring

Global Knowledge

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

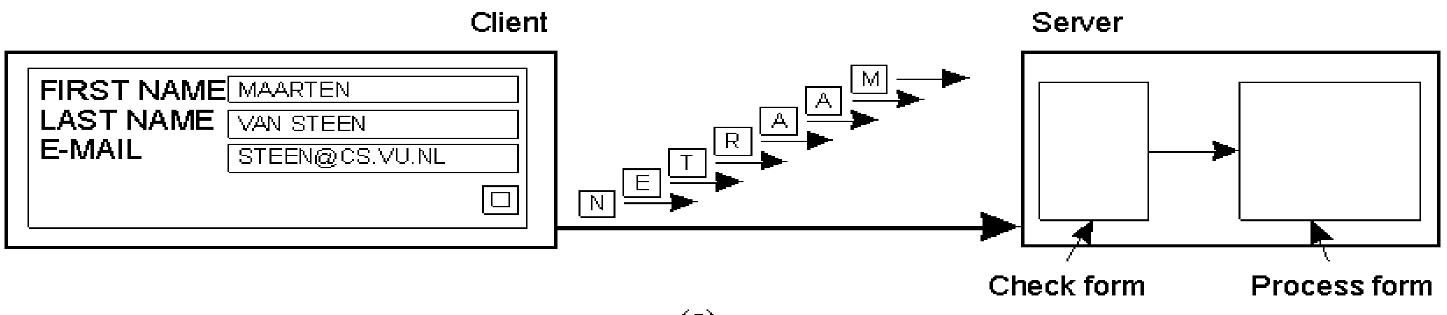
Naming

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

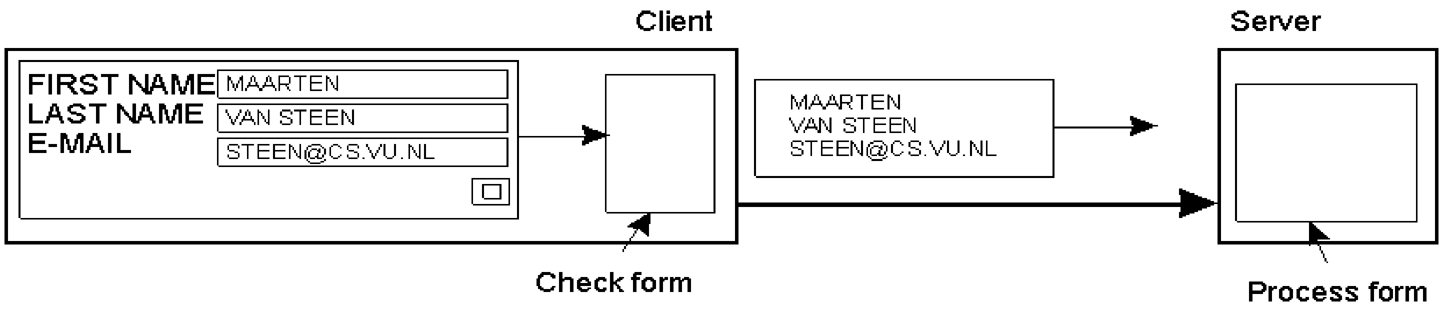
Scalability

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

Scaling Techniques (1) Hide Communication Latencies



(a)



(b)

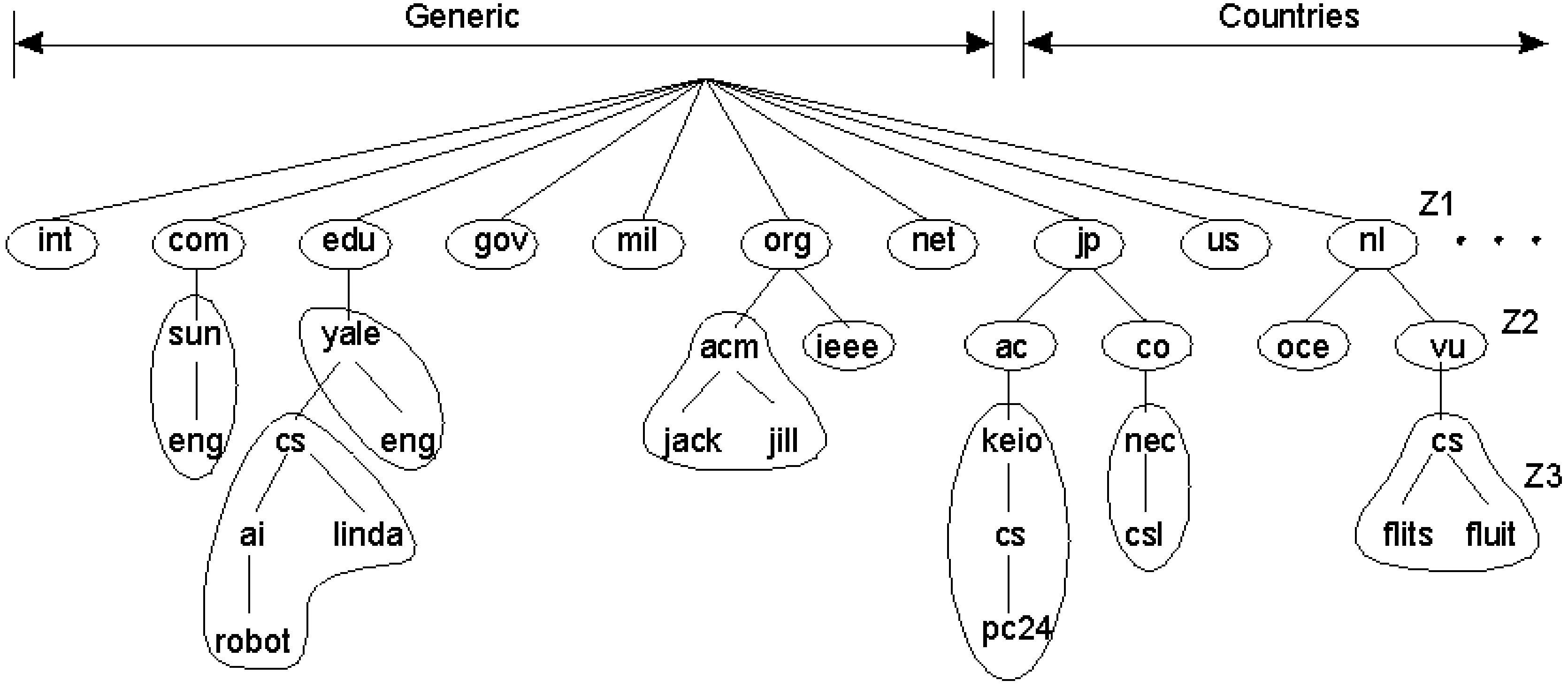
The difference between letting:

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 Primitives

- ❑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

Message Passing cont..

Design Issues

- ***Blocking vs Non blocking primitives***
- ***Nonblocking***
- 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 behavior 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)

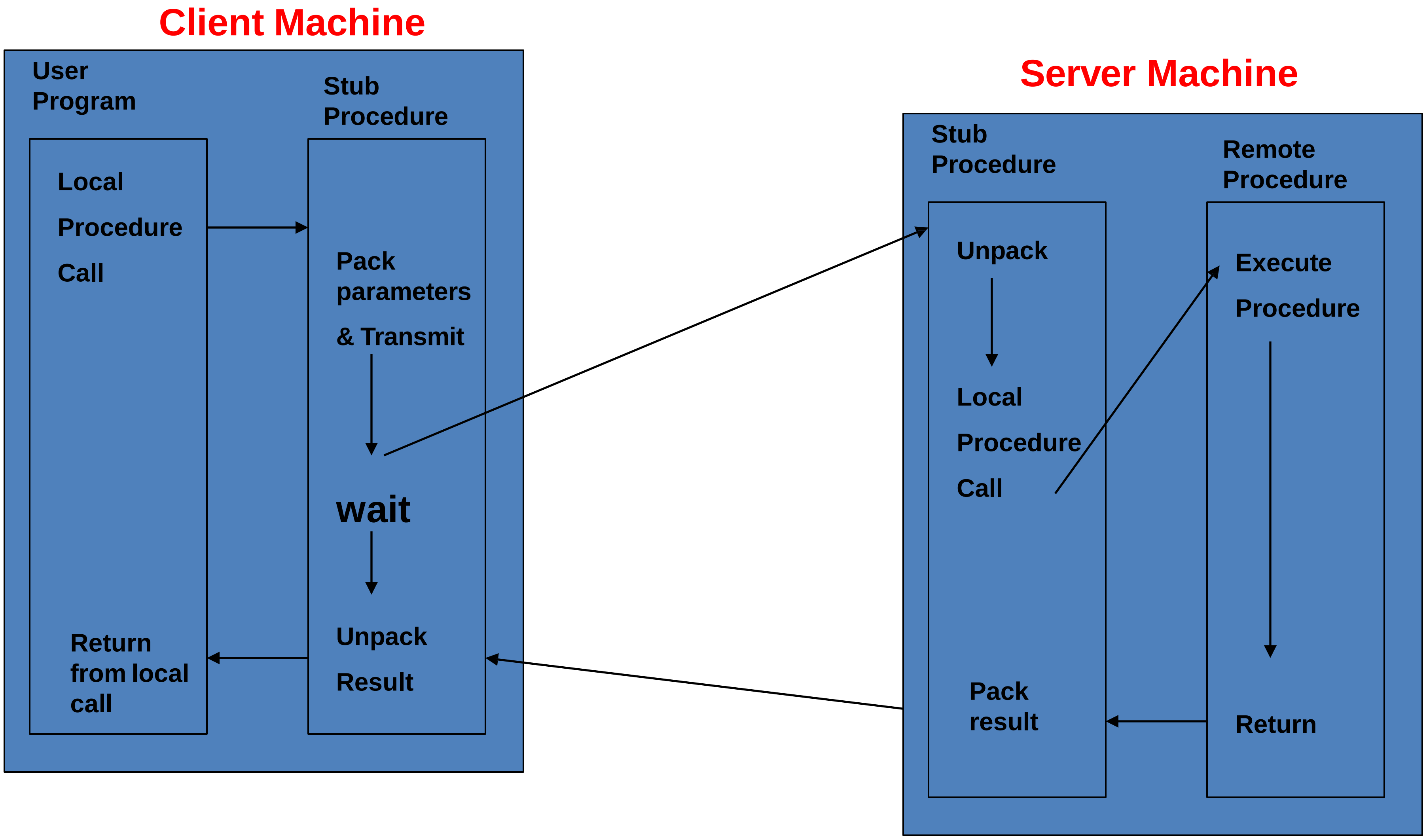
- ❑ RPC is an interaction between a client and a server
- ❑ Client invokes procedure on sever
- ❑ Server executes the procedure and pass the result back to client
- ❑ Calling process is suspended and proceeds only after getting the result from server

RPC Design issues

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

Structure

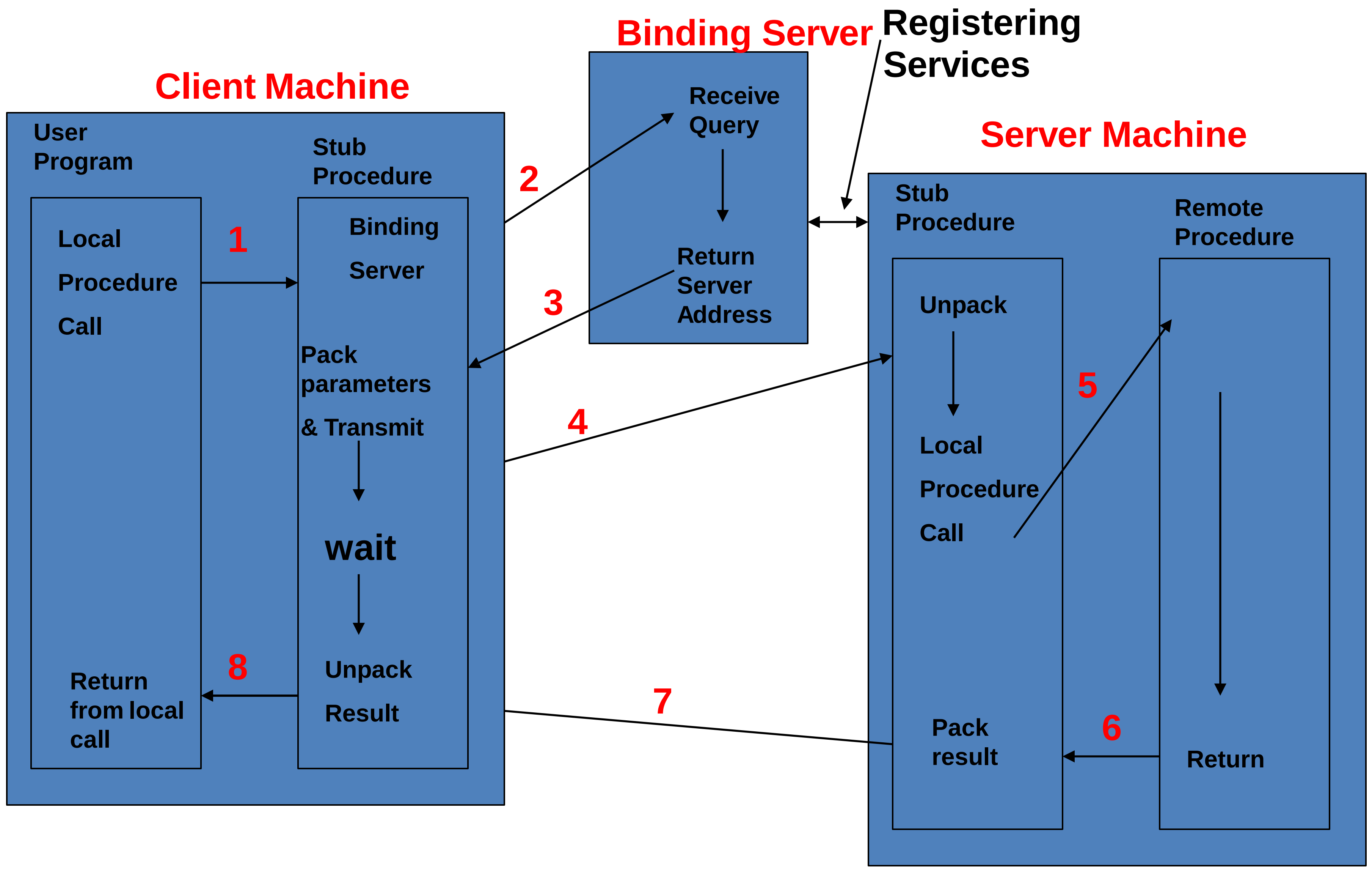
- RPC mechanism is based upon stub procedures.



Binding

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

Binding


Binding Server
Registering Services
Client Machine
Server Machine
User Program
Local Procedure Call
Return from local call
Stub Procedure
Binding Server
Pack parameters & Transmit
wait
Unpack Result
Receive Query
Return Server Address
Stub Procedure
Unpack
Local Procedure Call
Pack result
Remote Procedure
Return
1
2
3
4
5
6
7
8

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 → C2) & Computations W2 & W1 share the same data, then to be correct in the presence of failures RPC should satisfy
  - **C1 → C2 implies W1 → W2**

DEADLOCKS

EXAMPLES:

- "It takes money to make money".
- You can't get a job without experience; you can't get experience without a job.

BACKGROUND:

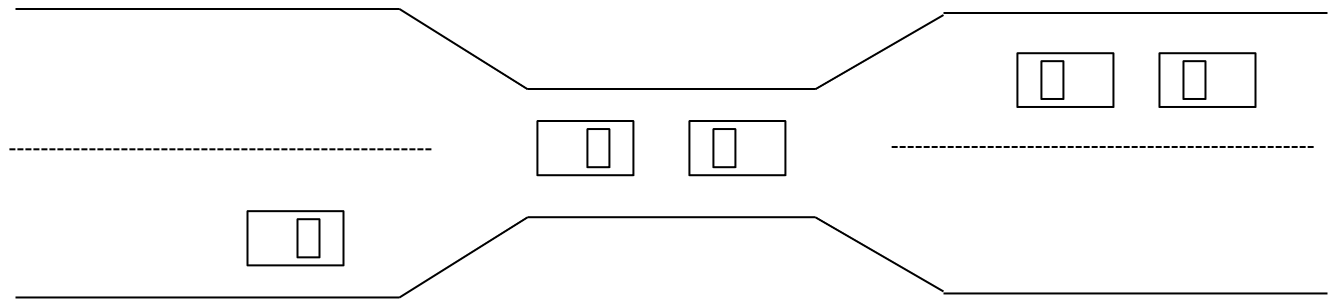
The cause of deadlocks: Each process needing what another process has. This results from sharing resources such as memory, devices, links.

Under normal operation, a resource allocations proceed like this::

1. Request a resource (suspend until available if necessary).
2. Use the resource.
3. Release the resource.

DEADLOCKS

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

DEADLOCK CHARACTERISATION

NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur:

Mutual exclusion

One or more than one resource must be held by a process in a non-sharable (exclusive) mode.

Hold and Wait

A process holds a resource while waiting for another resource.

No Preemption

There is only voluntary release of a resource - nobody else can make a process give up a resource.

Circular Wait

Process A waits for Process B waits for Process C .... waits for Process A.

RESOURCE ALLOCATION GRAPH

A visual (mathematical) way to determine if a deadlock has, or may occur.

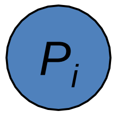
G = (V, E) The graph contains nodes and edges.

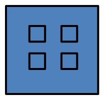
V Nodes consist of processes = { P1, P2, P3, ...} and resource types { R1, R2, ...}

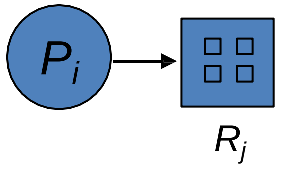
E Edges are (Pi, Rj) or (Ri, Pj)

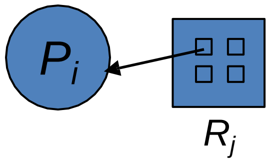
An arrow from the **process** to **resource** indicates the process is **requesting** the resource. An arrow from **resource** to **process** shows an instance of the resource has been **allocated** to the process.

Process is a circle, resource type is square; dots represent number of instances of resource in type. Request points to square, assignment comes from dot.







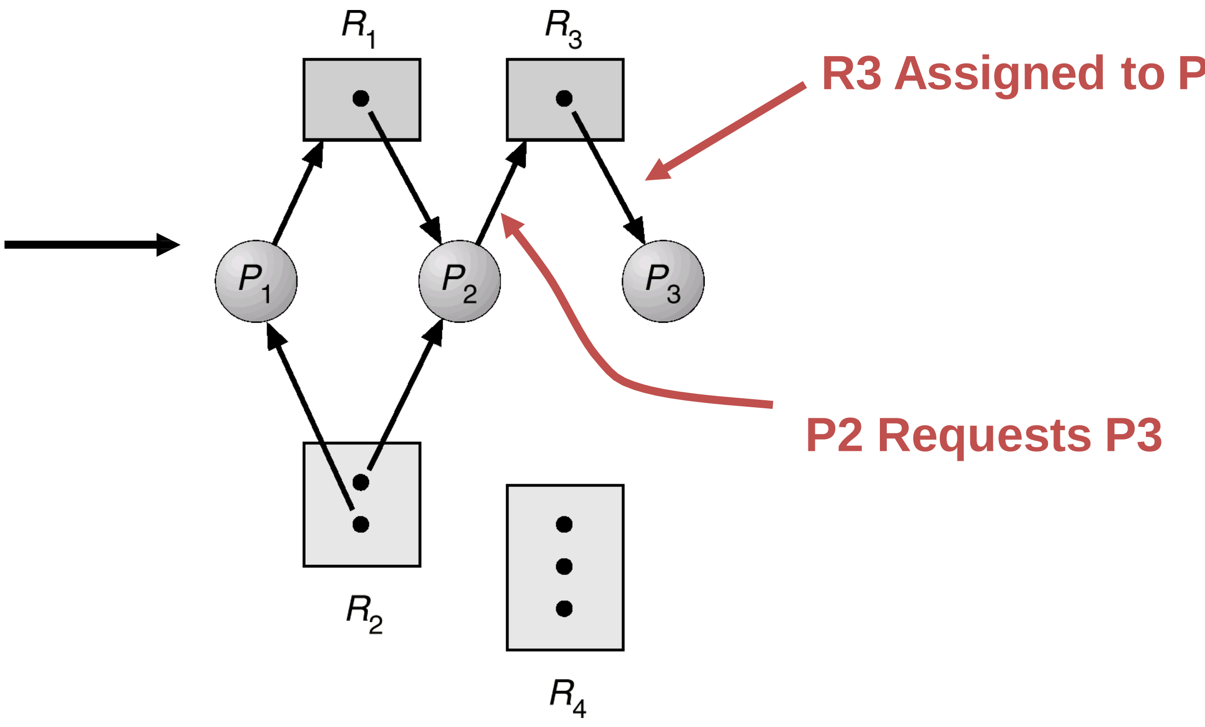


DEADLOCKS

RESOURCE ALLOCATION GRAPH

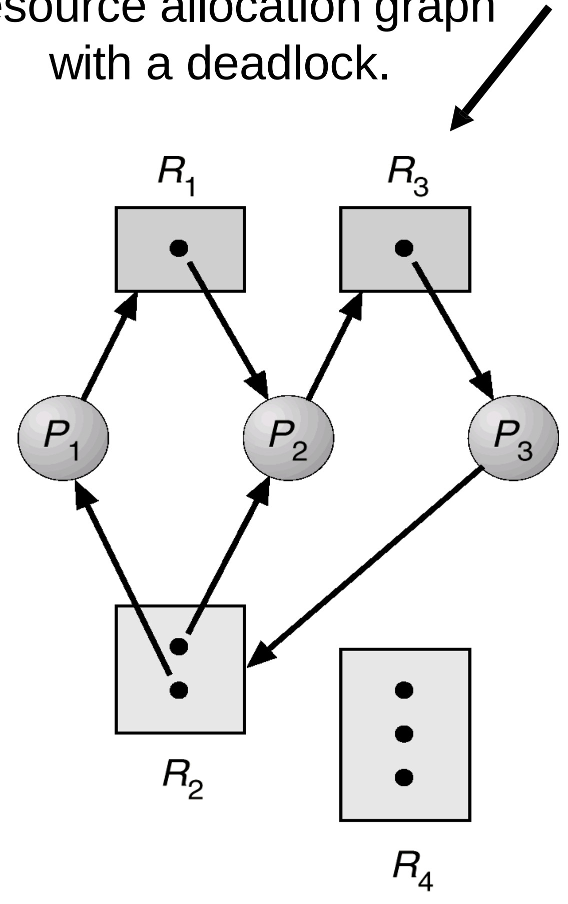
- If the graph contains no cycles, then no process is deadlocked.
- If there is a cycle, then:
    a) If resource types have multiple instances, then deadlock MAY exist.
    b) If each resource type has 1 instance, then deadlock has occurred.

Resource allocation graph

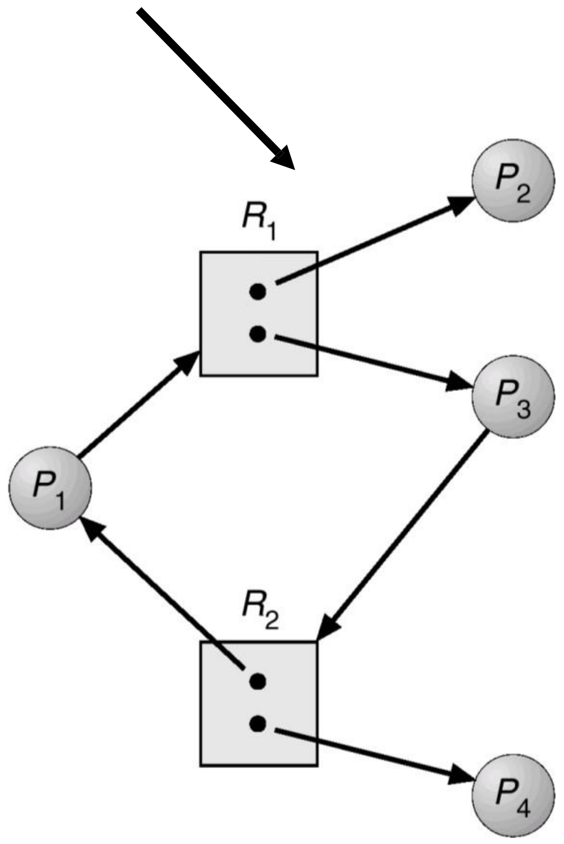


RESOURCE ALLOCATION GRAPH

Resource allocation graph with a deadlock.



Resource allocation graph with a cycle but no deadlock.



DEADLOCKS

Strategy

HOW TO HANDLE DEADLOCKS – GENERAL STRATEGIES

There are three methods:

Ignore Deadlocks: ← Most Operating systems do this!!

Ensure deadlock **never** occurs using either

Prevention	Prevent any one of the 4 conditions from happening.
Avoidance	Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations..

Allow deadlock to happen. This requires using both:

Detection	Know a deadlock has occurred.
Recovery	Regain the resources.

Deadlock Prevention

Do not allow one of the four conditions to occur.

Mutual exclusion:

a) Automatically holds for printers and other non-sharables.
b) Shared entities (read only files) don't need mutual exclusion (and aren’t susceptible to deadlock.)
c) Prevention not possible, since some devices are intrinsically non-sharable.

Hold and wait:

a) Collect all resources before execution.
b) A particular resource can only be requested when no others are being held. A sequence of resources is always collected beginning with the same one.
c) Utilization is low, starvation possible.

Deadlock Prevention

Do not allow one of the four conditions to occur.

No preemption:

a) Release any resource already being held if the process can't get an additional resource.
b) Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it. Otherwise wait.

Circular wait:

a) Number resources and only request in ascending order.

EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.

Prevention is generally the easiest to implement.

Deadlock Avoidance

If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.

Possible states are:

Deadlock	No forward progress can be made.
Unsafe state	A state that **may** allow deadlock.
Safe state	A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.

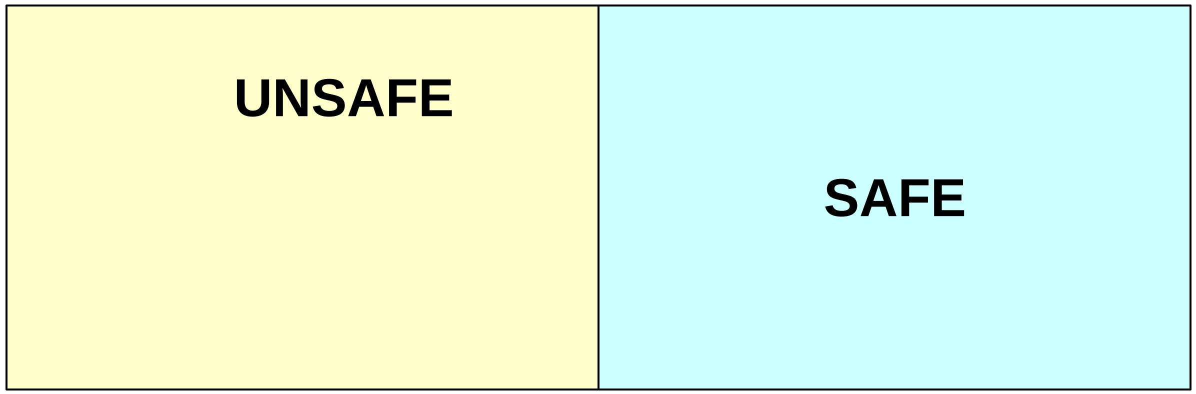
The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.

DEADLOCKS

Deadlock Avoidance

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.



Only with luck will processes avoid deadlock.

O.S. can avoid deadlock.

DEADLOCKS

Deadlock Avoidance

Let's assume a very simple model: each process declares its maximum needs. In this case, algorithms exist that will ensure that no unsafe state is reached. *Maximum needs* does NOT mean it *must* use that many resources – simply that it *might* do so under some circumstances.

There are multiple instances of the resource in these examples.

EXAMPLE:

There exists a total of 12 resources. Each resource is used exclusively by a process. The current state looks like this:

Process	Max Needs	Allocated	Current Needs
P0	10	5	5
P1	4	2	2
P2	9	3	7

In this example, < p1, p0, p2 > is a workable sequence.

Suppose p2 requests and is given one more resource. What happens then?

DEADLOCKS

Deadlock Avoidance

Safety Algorithm

A method used to determine if a particular state is safe. It's safe if there exists a sequence of processes such that for all the processes, there's a way to avoid deadlock:

The algorithm uses these variables:

Need[I] – the remaining resource needs of each process.
Work - Temporary variable – how many of the resource are currently available.
Finish[I] – flag for each process showing we've analyzed that process or not.

need <= available + allocated[0] + .. + allocated[I-1] ← **Sign of success**

Let **work** and **finish** be vectors of length **m** and **n** respectively.

Deadlock Avoidance

Safety Algorithm

```
1.    Initialize work              = available
      Initialize  finish[i]   = false,   for i = 1,2,3,..n

2.    Find an i such that:
      finish[i] == false   and   need[i] <= work

      If no such i exists, go to step 4.

3.    work                    = work   +   allocation[i]
      finish[i]               = true
      goto step 2

4.    if finish[i]   == true for all i,    then the system is in a safe state.
```

DEADLOCKS

Deadlock Avoidance

Safety Algorithm

Do these examples:

Consider a system with: five processes, P0→ P4, three resource types, A, B, C.

Type A has 10 instances, B has 5 instances, C has 7 instances.

At time T0 the following snapshot of the system is taken.

Max Needs = allocated + can-be-requested

Is the system in a safe state?

		←	Alloc	→		←	Re	→		←	Avail	→
		A	B	C		A	qB	C		A	B	C
P0		0	1	0		7	4	3		3	3	2
P1		2	0	0		0	2	0				
P2		3	0	2		6	0	0				
P3		2	1	1		0	1	1				
P4		0	0	2		4	3	1				

DEADLOCKS

Deadlock Avoidance

Safety Algorithm

Do these examples:

Now try it again with only a slight change in the request by P1.

P1 requests one additional resource of type A, and two more of type C.

Request1 = (1,0,2).

Is Request1 < available?

Produce the state chart as if the request is Granted and see if it's safe. (We've drawn the chart as if it's granted.

Can the request be granted?

	←	Alloc	→	←	Req	→	←	Avail	→
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	1#	3	0#
P1	3#	0	2#	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

DEADLOCKS

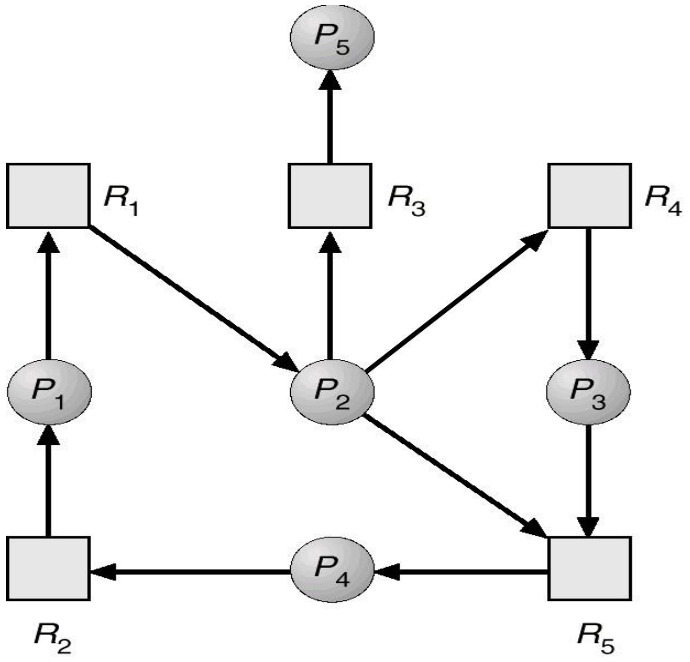
Deadlock Detection

Need an algorithm that determines if deadlock occurred.

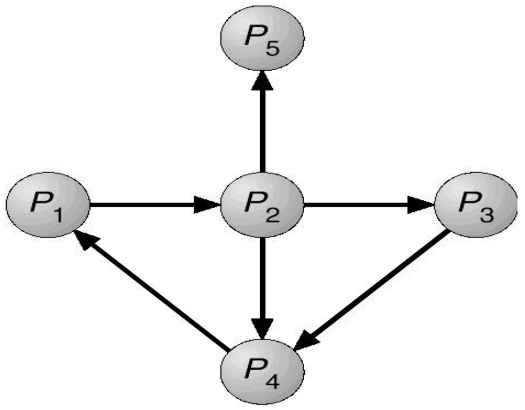
Also need a means of recovering from that deadlock.

SINGLE INSTANCE OF A RESOURCE TYPE

- Wait-for graph == remove the resources from the usual graph and collapse edges.
- An edge from p(j) to p(i) implies that p(j) is waiting for p(i) to release.



(a)



(b)

Deadlock Detection

SEVERAL INSTANCES OF A RESOURCE TYPE

Complexity is of order m * n * n.

We need to keep track of:

available	- records how many resources of each type are available.
allocation	- number of resources of type m allocated to process n.
request	- number of resources of type m requested by process n.

Let **work** and **finish** be vectors of length **m** and **n** respectively.

Deadlock Detection

```
1.  Initialize              work[ ] = available[ ]
    For  i = 1,2,...n,  if  allocation[i] != 0  then    // For all n processes
         finish[i]  =  false;  otherwise,  finish[i] = true;

2.  Find an i process such that:
    finish[i] == false and request[i] <= work

    If no such i exists, go to step 4.

3.   work  =  work + allocation[i]
     finish[i] = true
    goto step 2

4.  if finish[i] == false for some i, then the system is in deadlock state.  IF
    finish[i]  == false, then       process p[i] is deadlocked.
```

DEADLOCKS

Deadlock Detection

EXAMPLE

We have three resources, A, B, and C. A has 7 instances, B has 2 instances, and C has 6 instances. At this time, the allocation, etc. looks like this:

Is there a sequence that will allow deadlock to be avoided?

Is there more than one sequence that will work?

		←	Alloc	→		←	Re	→		←	Avail	→
		A	B	C		A	qB	C		A	B	C
P0		0	1	0		0	0	0		0	0	0
P1		2	0	0		2	0	2				
P2		3	0	3		0	0	0				
P3		2	1	1		1	0	0				
P4		0	0	2		0	0	2				

DEADLOCKS

Deadlock Detection

EXAMPLE

Suppose the Request matrix is changed like this. In other words, the maximum amounts to be allocated are initially declared so that this request matrix results.

Is there now a sequence that will allow deadlock to be avoided?

USAGE OF THIS DETECTION ALGORITHM

Frequency of check depends on how often a deadlock occurs and how many processes will be affected.

		←	Alloc	→		←	Re	→		←	Avail	→
		A	B	C		A	qB	C		A	B	C
P0		0	1	0		0	0	0		0	0	0
P1		2	0	0		2	0	2				
P2		3	0	3		0	0	1#				
P3		2	1	1		1	0	0				
P4		0	0	2		0	0	2				

Deadlock Recovery

So, the deadlock has occurred. Now, how do we get the resources back and gain forward progress?

PROCESS TERMINATION:

- Could delete all the processes in the deadlock -- this is expensive.
- Delete one at a time until deadlock is broken (time consuming).
- Select who to terminate based on priority, time executed, time to completion, needs for completion, or depth of rollback
- In general, it's easier to preempt the resource, than to terminate the process.

RESOURCE PREEMPTION:

- Select a victim - which process and which resource to preempt.
- Rollback to previously defined "safe" state.
- Prevent one process from always being the one preempted (starvation).

DEADLOCKS

Deadlock Recovery

COMBINED APPROACH TO DEADLOCK HANDLING:

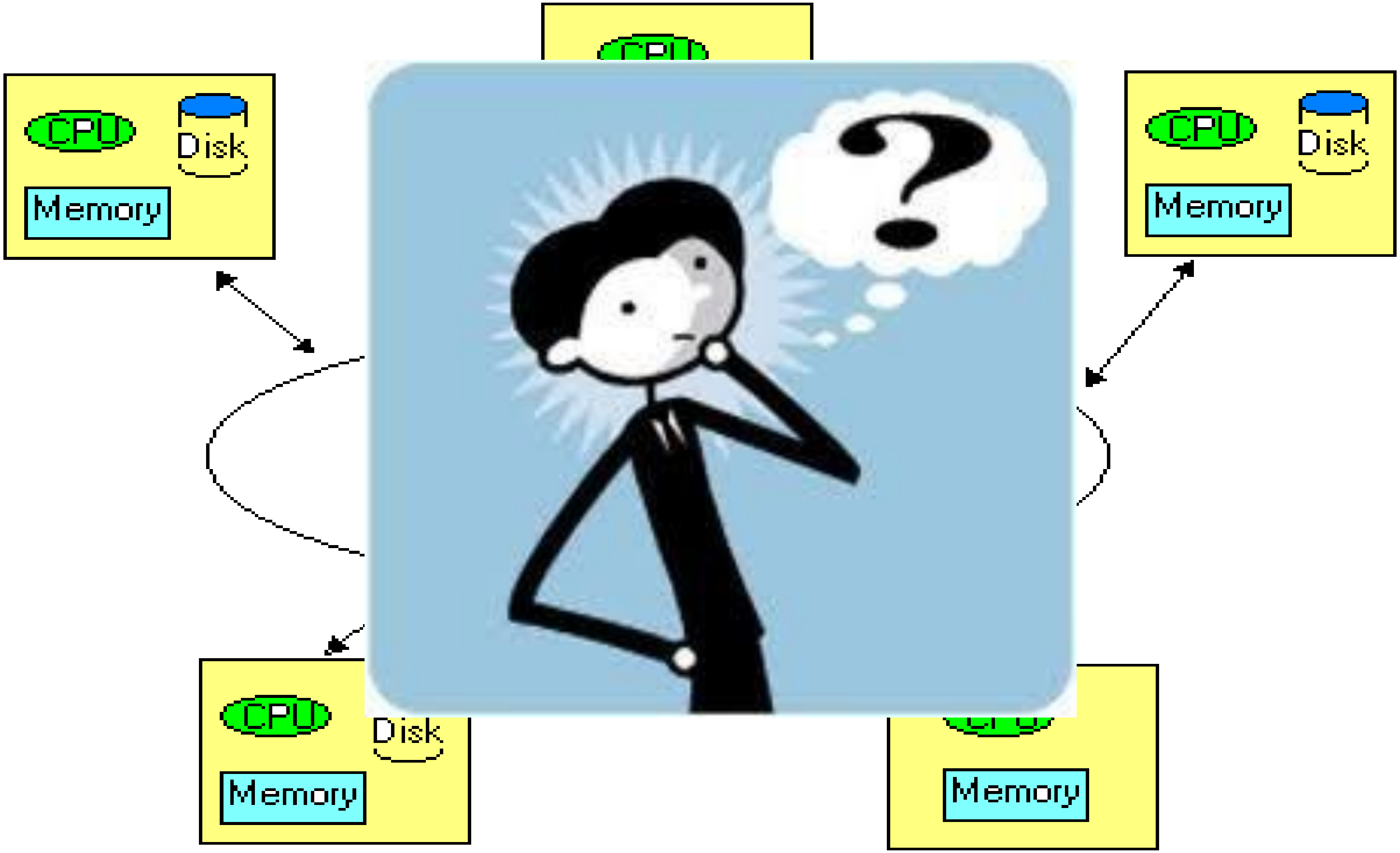
- Type of resource may dictate best deadlock handling. Look at ease of implementation, and effect on performance.
- In other words, there is no one best technique.
- Cases include:

Preemption for

memory,

Preallocation for

Thank U


CPU
Disk
Memory
CPU
Disk
Memory
CPU
Disk
Memory
CPU
Memory
Disk
?