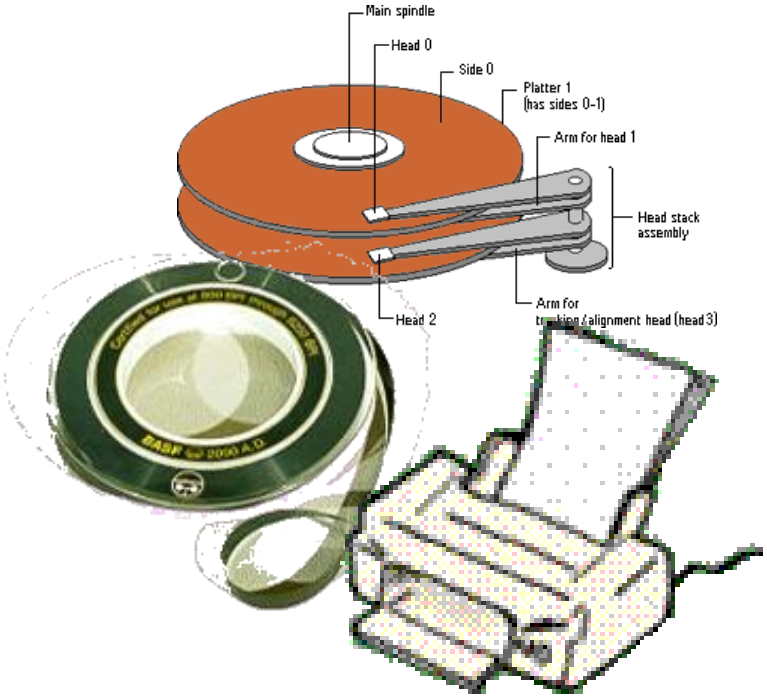
Module 4
Distributed Deadlock Detection

Deadlock

Resources



Examples of computer resources

- printers
- tape drives
- tables

Processes need access to resources in reasonable order

Suppose a process holds resource A and requests resource B at same time another process holds B and requests A both are blocked and remain so

Resources

Deadlocks occur when …

processes are granted exclusive access to devices

we refer to these devices generally as **resources**

Preemptable resources

can be taken away from a process with no ill effects

Non-preemptable resources

will cause the process to fail if taken away

Resources

Sequence of events required to use a resource

1. **request** the resource
2. **use** the resource
3. **release** the resource

Must wait if request is denied requesting process may be blocked may fail with error code

Introduction to Deadlocks

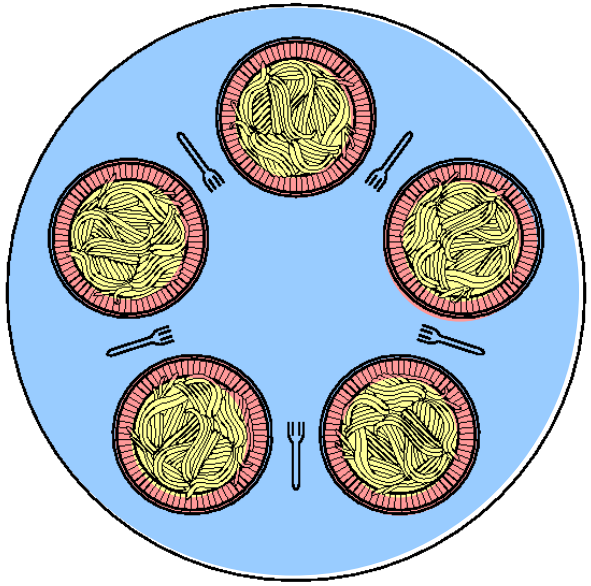
Formal definition :

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

Usually the event is release of a currently held resource

None of the processes can …

- run
- release resources
- be awakened



1. Mutual exclusion condition
   - each resource assigned to 1 process or is available
2. Hold and wait condition
   - process holding resources can request additional
3. No preemption condition
   - previously granted resources cannot forcibly taken away
4. Circular wait condition
   - must be a circular chain of 2 or more processes
   - each is waiting for resource held by next member of the chain

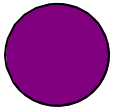
Deadlock Strategies

Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions

Resource Allocation Graph

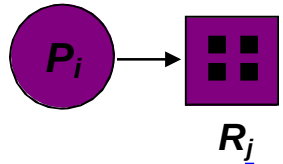
- Process



- Resource Type with 4 instances

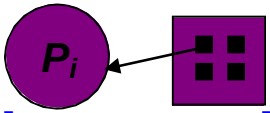
The sequence of Process's recourse utilization

- P_i requests instance of R_j



Request edge

- P_i is holding an instance of R_j

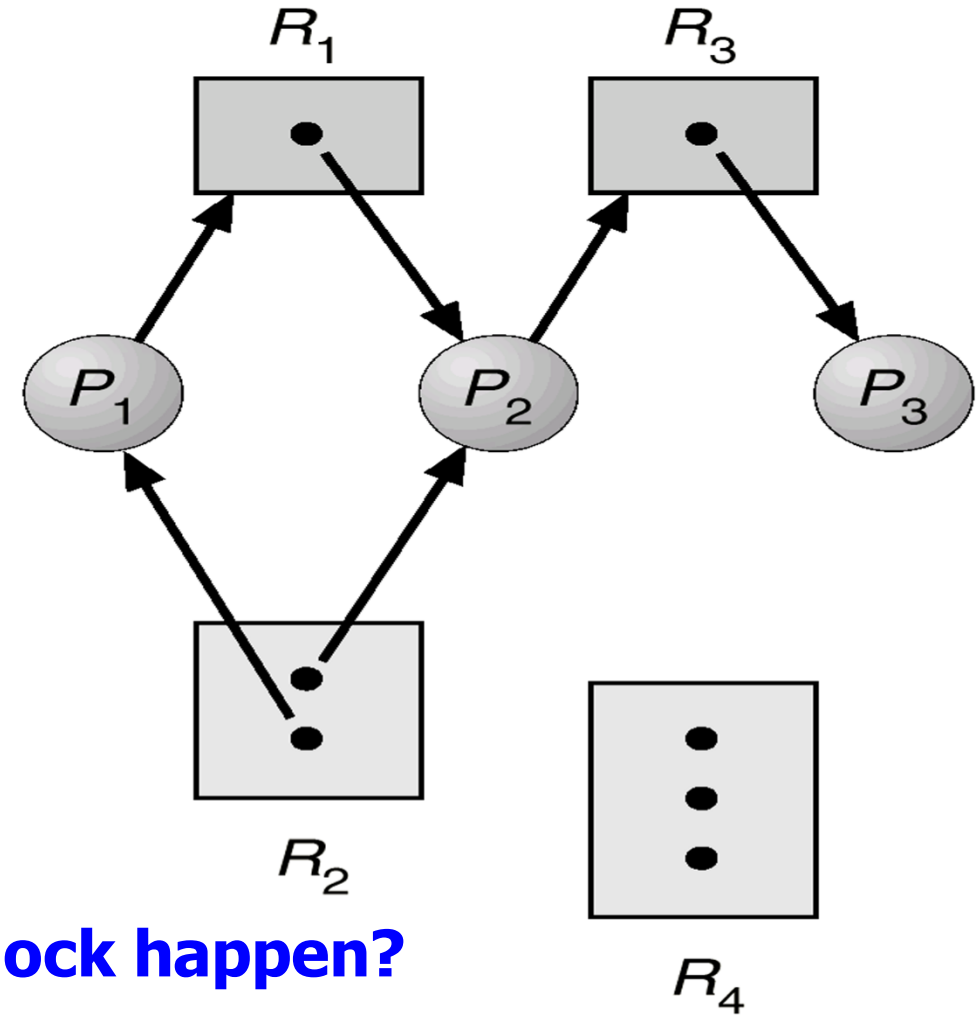


Assignment edge

- P_i releases an instance of R_j

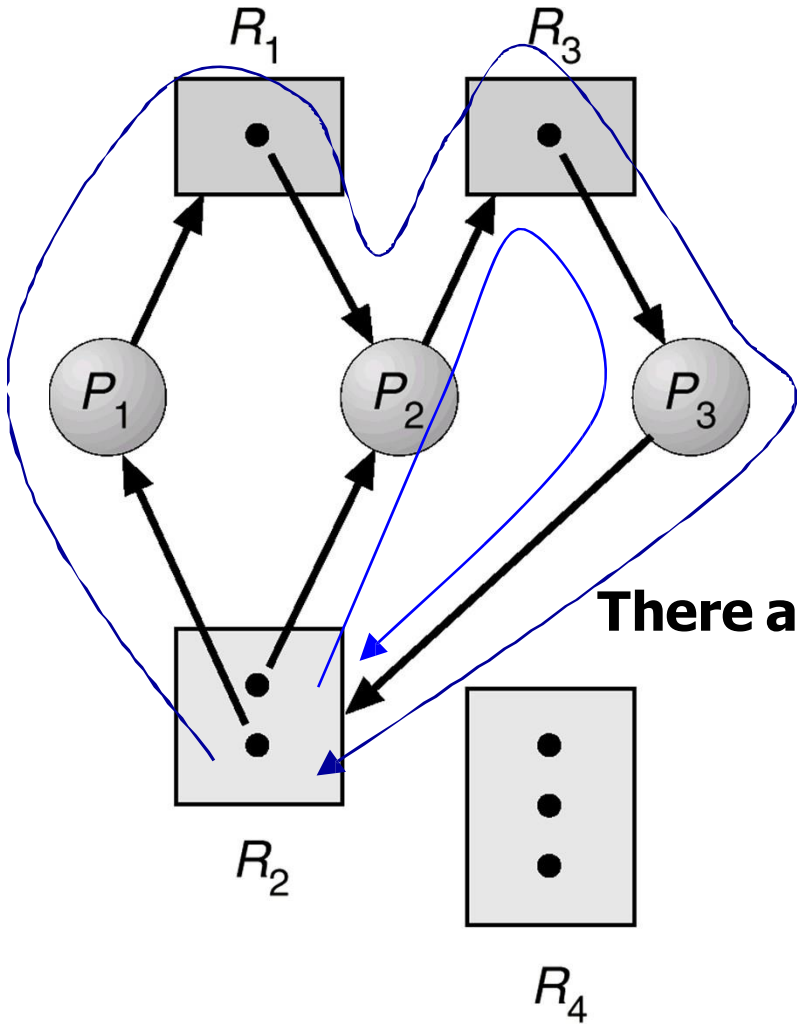
R_j

Resource-allocation graph



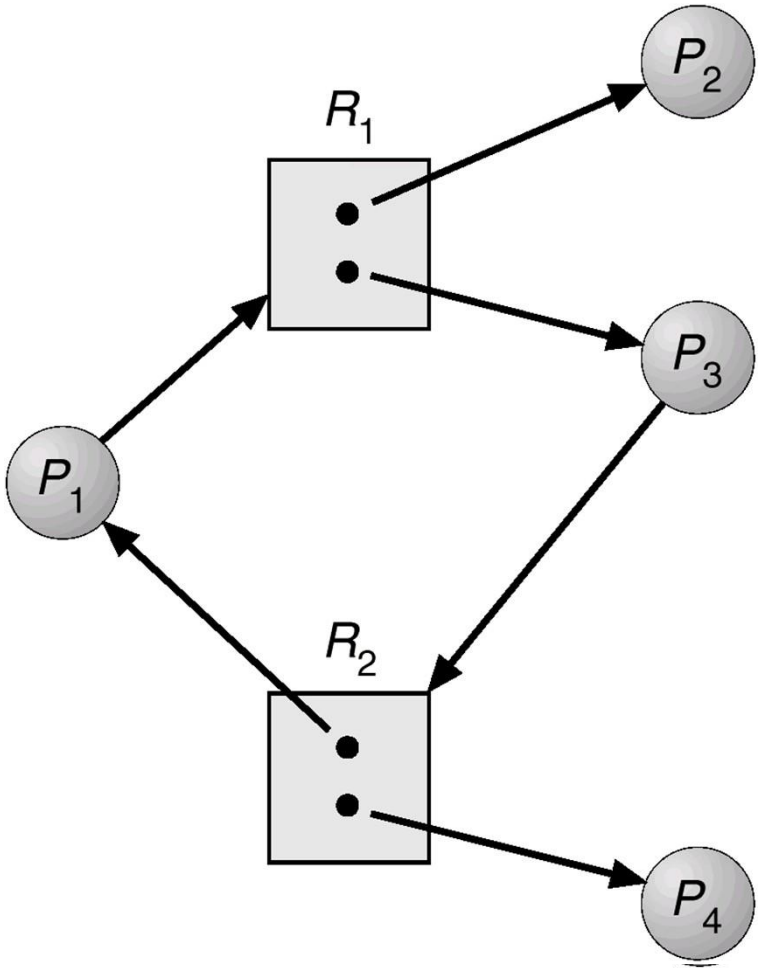
Can a deadlock happen?

Resource Allocation Graph With A Deadlock



There are two cycles found.

Resource Allocation Graph With A Cycle But No Deadlock



- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

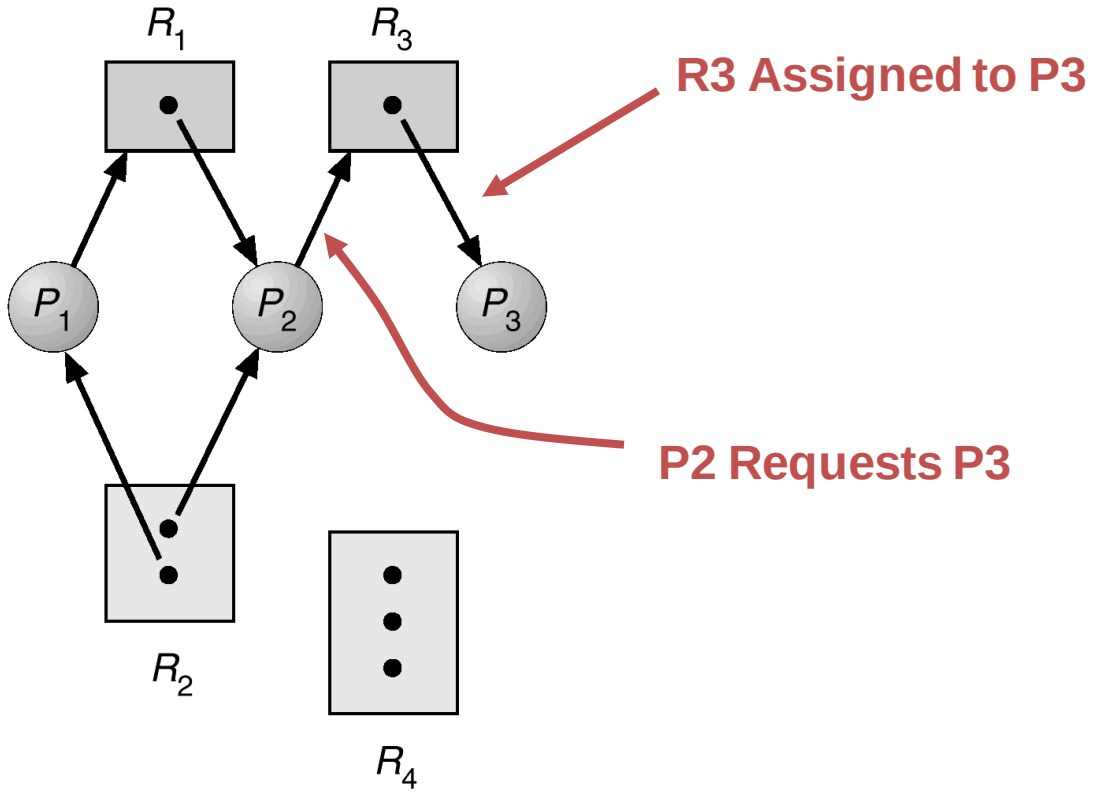
Resource allocation graph

If the graph contains no cycles, then no process is deadlocked.
If there is a cycle, then:

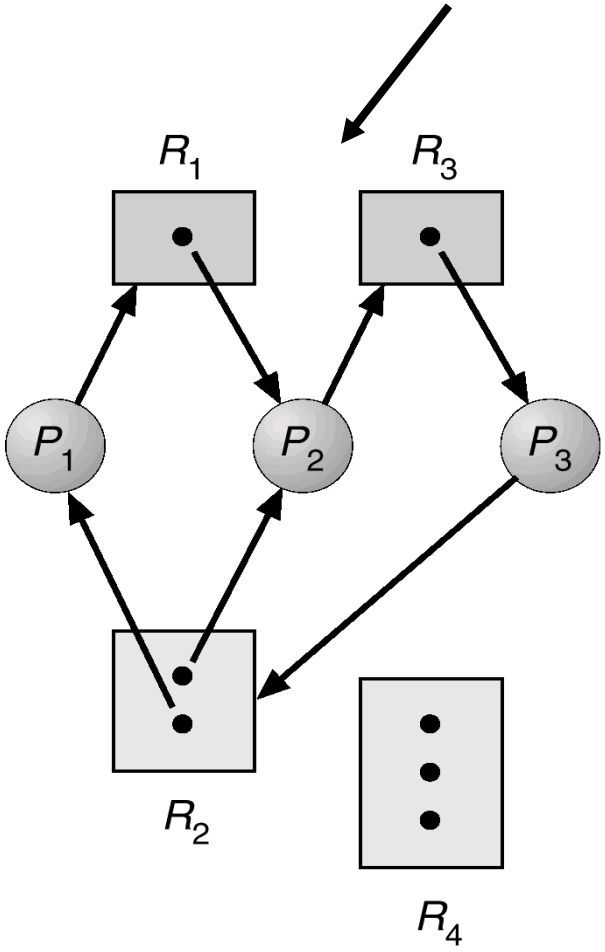
a) If resource types have multiple instances, then deadlock MAY exist.

b) If each resource type has 1 instance, then deadlock has occurred.

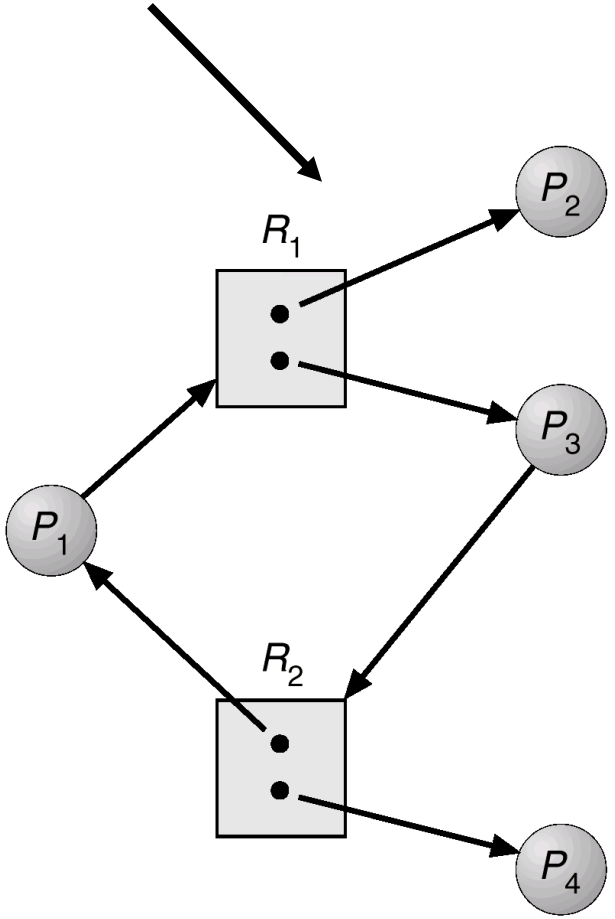
Resource allocation graph



Resource allocation graph with a deadlock.



Resource allocation graph with a cycle but no deadlock.



Cycle & Knot

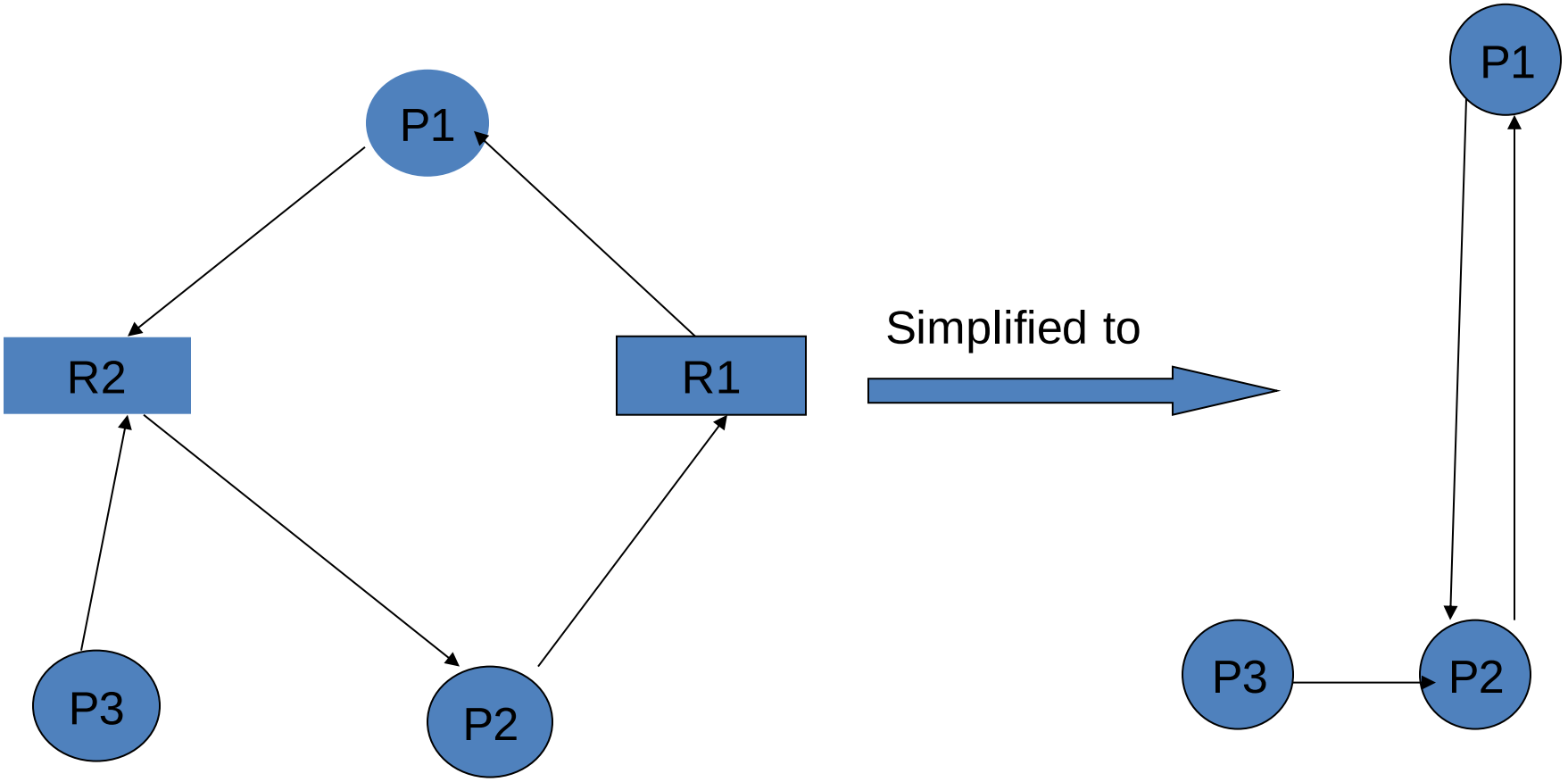
A cycle is a necessary condition for deadlock

If there is a only single unit of each resource type involved in the cycle, a cycle is both a necessary and a sufficient condition for a deadlock to exists

If one or more of the resource types involved in the cycle have more than one unit, a knot is a sufficient condition for a deadlock to exists

Wait-for graph

Wait-for graph == remove the resources from the usual graph and collapse edges



Strategies for dealing with Deadlocks

1. just ignore the problem altogether (Most OS does this)
2. detection and recovery
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions

Along with the 3 strategies.. We should be clear about the two terms

Resource deadlocks

This occurs when two or more processes waits permanently for resource held by others.

Communication deadlocks

This occurs among a set of processes when they are blocked waiting for messages from other processes in the set in order to start execution but there are no messages in transit between them.

This method use some advance knowledge for a resource usage of process to predict the future state of the system for avoiding allocations that can eventually lead to a deadlock

Deadlock avoidance algorithm follow the below step:

Process-request(resource)

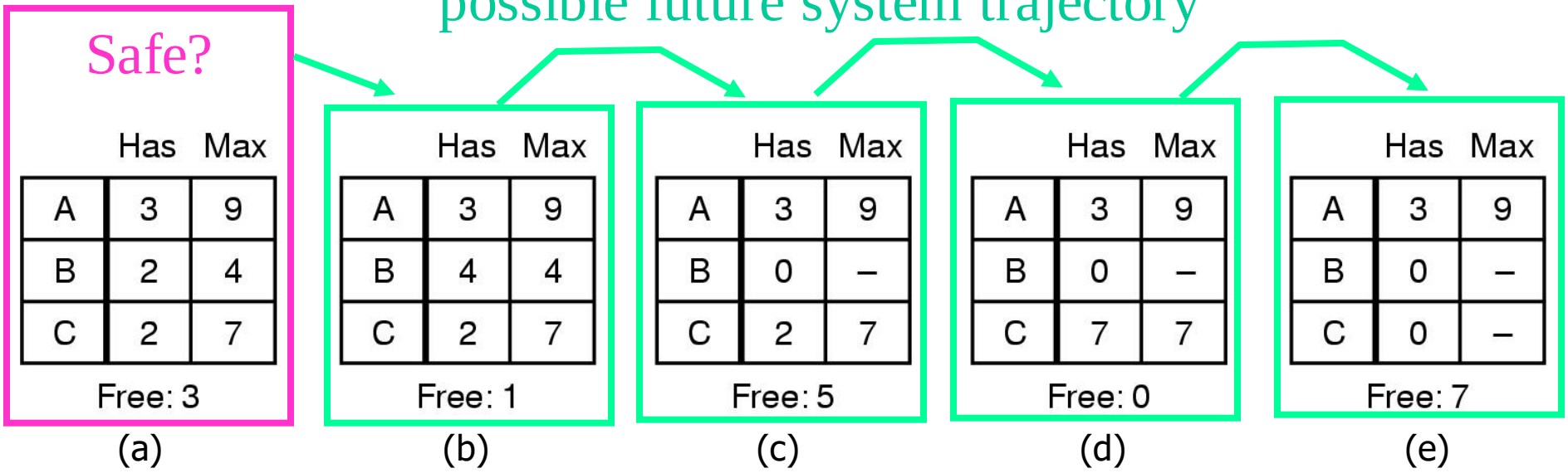
Resource – available – not allocated immediately

Knowledge is analyzed – whether it is safe to allocate the resource

- processor
- not leading to deadlock
- If it is safe
- resource is allocated else request is deferred

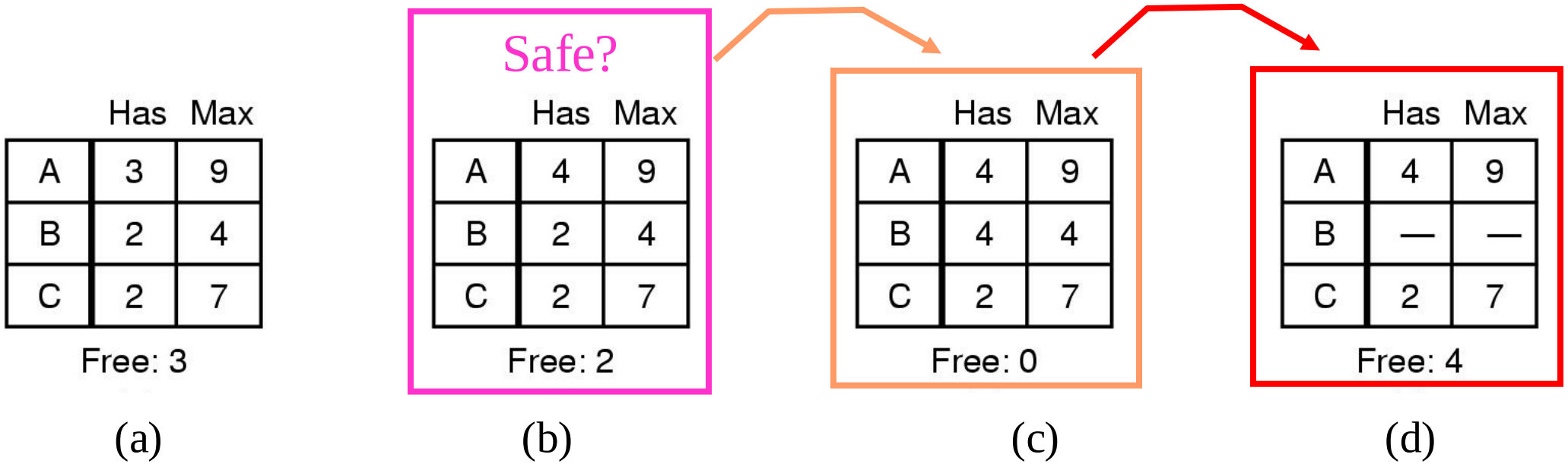
(Example: 10 Resources of 1 Type)

possible future system trajectory



Demonstration that the state in (a) is safe

? *possible future system trajectory where all processes terminate*



Demonstration that the state in b is not safe

? *possible future system trajectories:*
trajectory will lead to Deadlock state

It works on assumptions that advance knowledge of the resource requirements of various resources are available.
Assumes that no. of processors that compete for a resource are fixed.
Also assumes the no. of units available are fixed
These algorithms work restricts resource allocation in DS considering the unsafe condition which is unlikely to happen.

Some devices (such as printer) can be spooled

- only the printer daemon uses printer resource
- thus deadlock for printer eliminated

Not all devices can be spooled

Principle:

- avoid assigning resource when not absolutely necessary
- as few processes as possible actually claim the resource

Idea: A process request a resource, it does not hold any other resources.

Resource allocation polices are:

Require processes to request resources before starting

a process never has to wait for what it needs

instead requesting the resource before starting, process can request during execution

Even here it has to obey the idea of not holding the resource and requesting.

Advantages of second policy

This approach is useful when the many processors does not know the resources until they have started running

Some long process –needs resources at the end, so it is easy to request for resource during execution rather than having at the beginning.

Collective request is simple and effective but has following problems:

- Low resource utilization
- Causes starvation of a process that needs many resources.
- Accounting question is also araised.

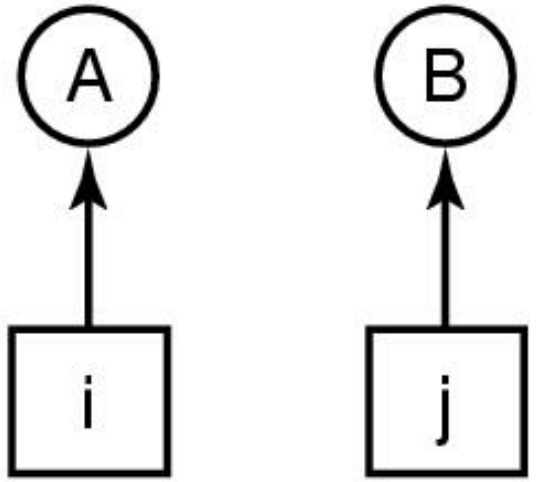
Resource is assigned a unique global number to impose a total ordering of all resource type.

Resource allocation policy

Any process can request a resource,but the requesting resource number should be greater than what it is holding

Ex. P1 is holding a resource of no. 3 it can request a resource of no. greater than 3, if it is 2, then it has to release 3 and request it. Because of this in RAG they is no possibility of creating cycles.

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

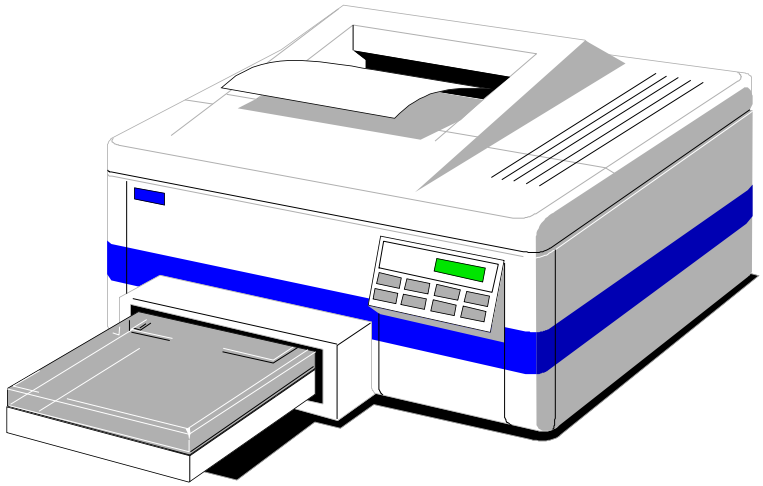


Normally **ordered** resources,
allocation in fixed ordering
The resource graph cannot contain cycles!

This is not a viable option

Consider a process given the printer

- halfway through its job
- now forcibly take away printer
- !!??



Resource allocation policies to prevent deadlock:

Blocking – two ways

Preempt the resource and block

Block if necessary preempt the resource

1. Process – request – resource-not available, all the resources are preempted from it and it is blocked, and once the requested resource + preempted resources are available , the process is unblocked.
2. Process – request – resource – not available, checks – blocked process, if resource is available, preempt the resources and block it as first case and proceed…

DS Deadlock Detection

Use Wait For Graph (WFG or TWF)

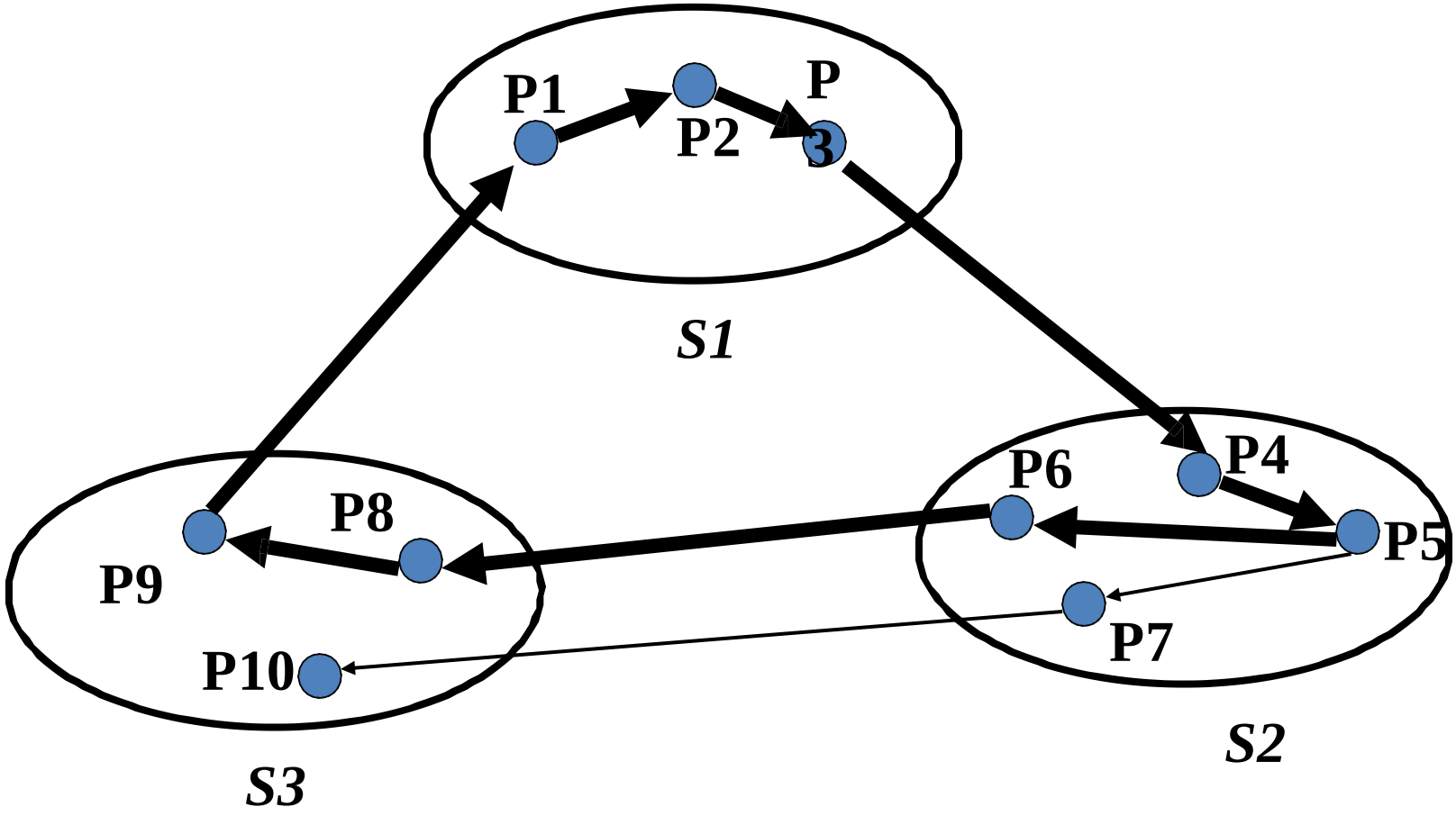
- All nodes are processes (threads)
- Resource allocation is done by a process (thread) sending a request message to another process (thread) which manages the resource (client - server communication model,RPC paradigm)

– A system is deadlocked IFF there is a directed cycle (or knot) in a global WFG

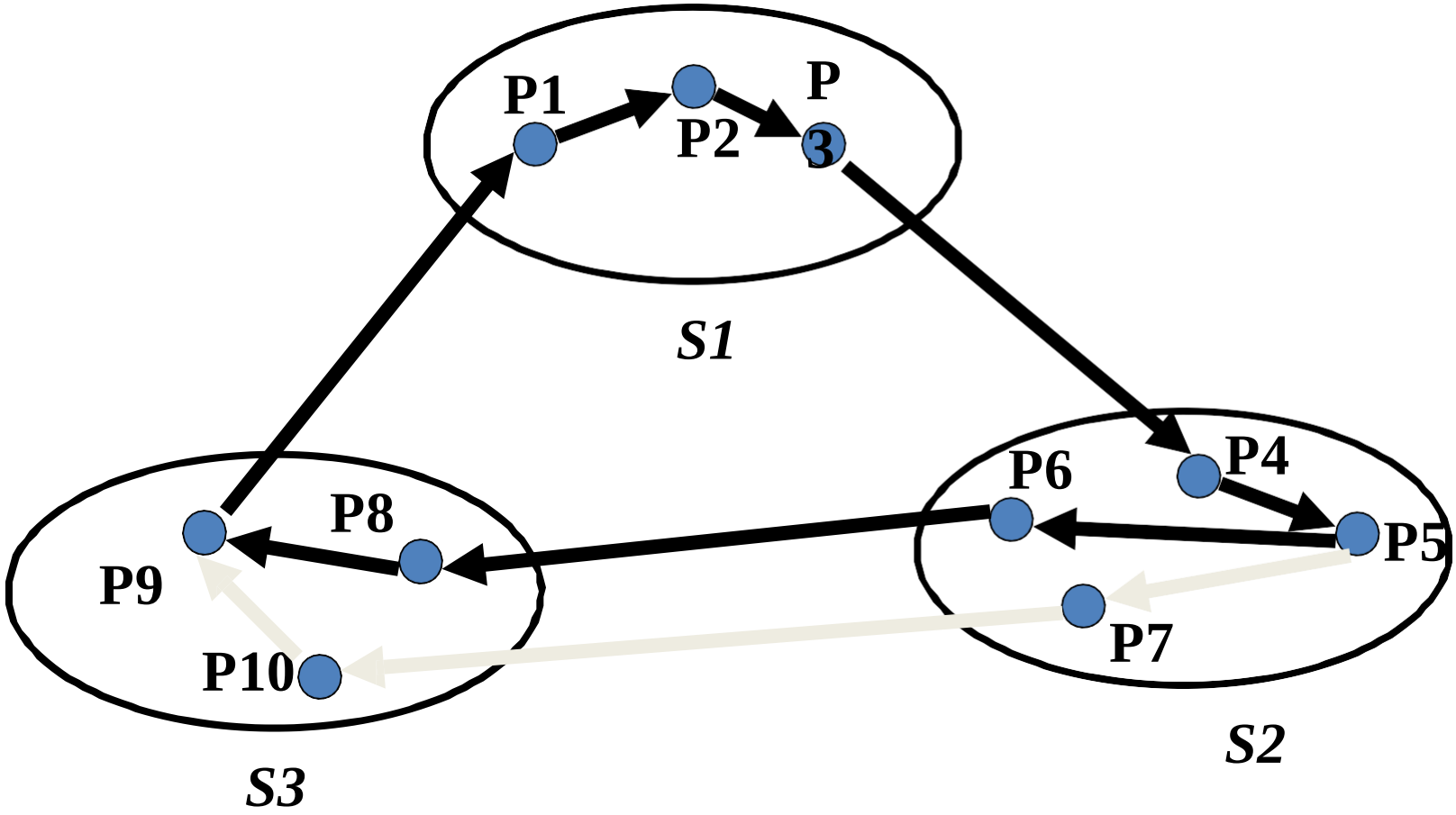
DS Deadlock Detection, Cycle vs. Knot

- The AND model of requests requires all resources currently being requested to be granted to un- block a computation
  - A **cycle is sufficient** to declare a deadlock with this model
- The OR model of requests allows a computation making multiple different resource requests to un-block as soon as any are granted
  - A **cycle** is a **necessary** condition
  - A **knot** is a **sufficient** condition

Deadlock in the AND model; there is a cycle but no knot
No Deadlock in the OR model



Deadlock in both the AND model and the OR model; there are cycles and a knot



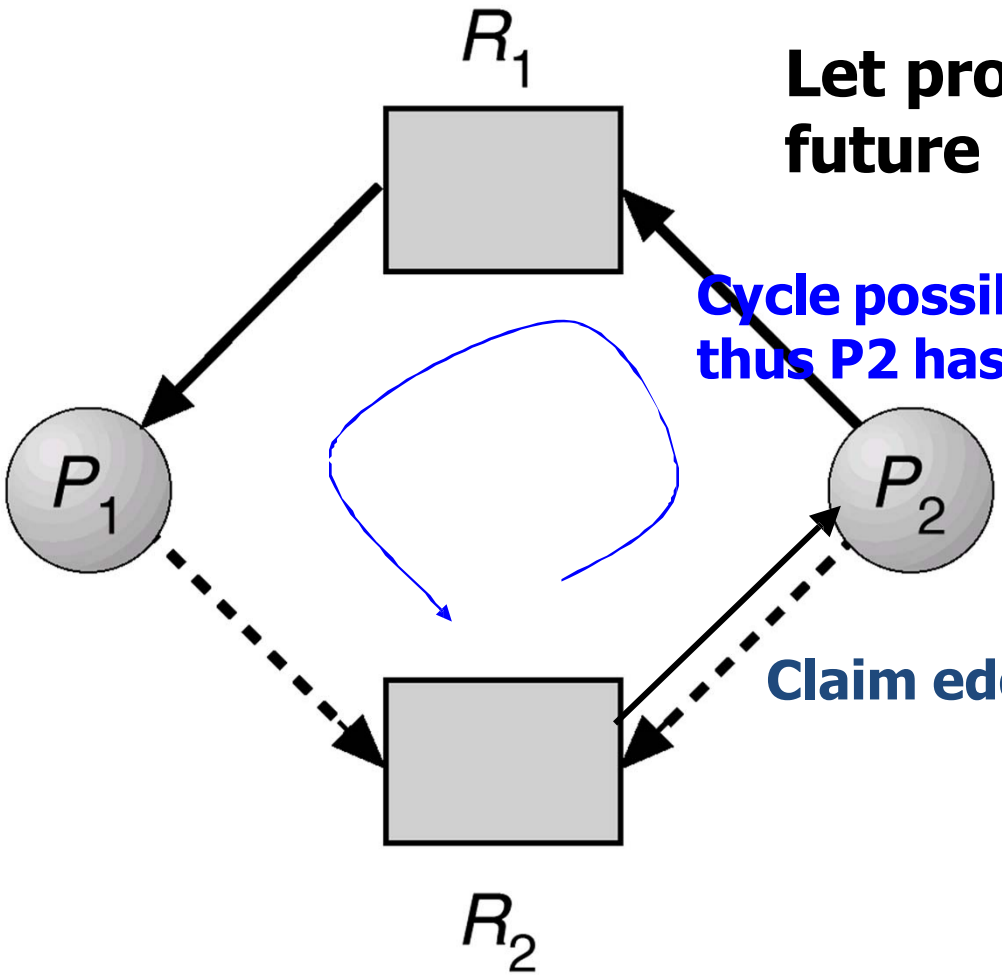
Deadlock Handling Strategies in DS

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

Distributed Deadlock Prevention

- A method that might work is to order the resources and require processes to acquire them in strictly increasing order. This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible.
- With global timing and transactions in distributed systems, two other methods are possible -- both based on the idea of assigning each transaction a global timestamp at the moment it starts.
- When one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger timestamp.
- We can then allow the wait only if the waiting process has a lower timestamp.
- The timestamp is always increasing if we follow any chain of waiting processes, so cycles are impossible --- we can used decreasing order if we like.
- It is wiser to give priority to old processes because
  - they have run longer so the system have larger investment on these processes.
  - they are likely to hold more resources.
  - A young process that is killed off will eventually age until it is the oldest one in the system, and that eliminates starvation.

Deadlock Avoidance



Let processes supply OS with future resource requests

Cycle possibly formed (unsafe state), thus P2 has to wait for a safe state

Claim edge (future request)

Control Organization for Deadlock Detection

- Centralized Control
- Distributed Control
- Hierarchical Control

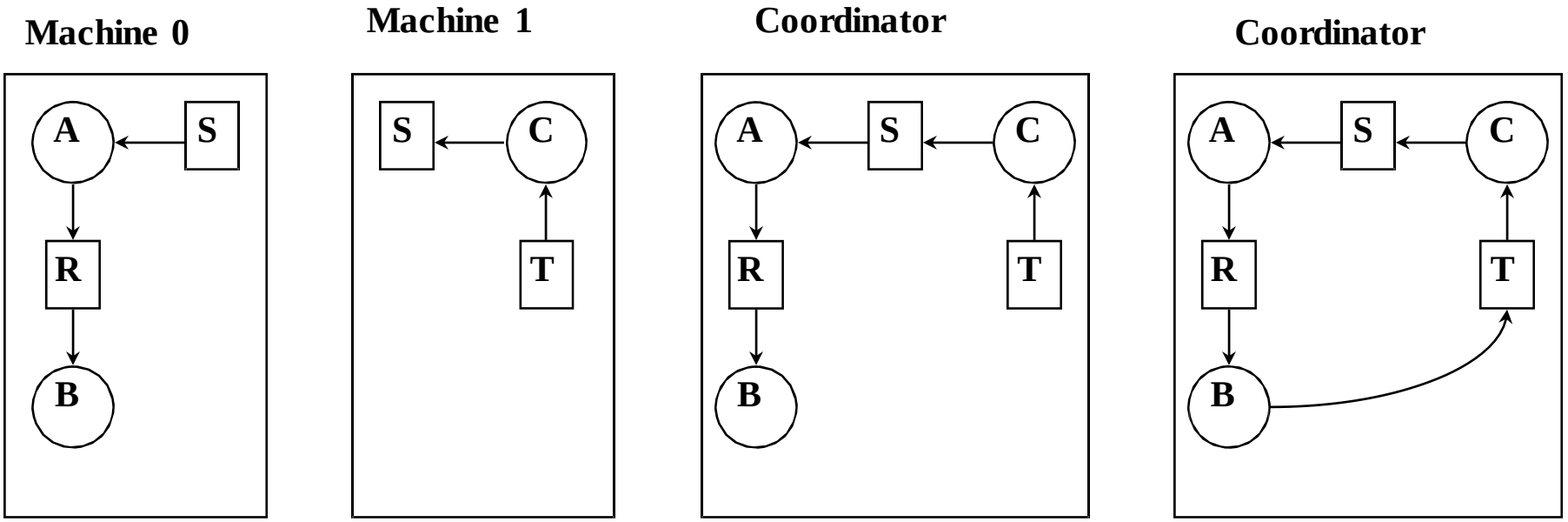
Issues in Deadlock Detection & Resolution

- **Detection**
  - **Progress: No undetected deadlocks**
  - **Safety: No false deadlocks**
- **Resolution**

Centralized Deadlock Detection

- We use a centralized deadlock detection algorithm and try to imitate the non-distributed algorithm.
  - Each machine maintains the resource graph for its own processes and resources.
  - A centralized coordinator maintain the resource graph for the entire system.
  - When the coordinator detect a cycle, it kills off one process to break the deadlock.
  - In updating the coordinator's graph, messages have to be passed.
    - Method 1) Whenever an arc is added or deleted from the resource graph, a message have to be sent to the coordinator.
    - Method 2) Periodically, every process can send a list of arcs added and deleted since previous update.
    - Method 3) Coordinator ask for information when it needs it.

False Deadlocks



B release R, and ask for T

- One possible way to prevent false deadlock is to use the Lamport's algorithm to provide global timing for the distributed systems.
- When the coordinator gets a message that leads to a suspect deadlock:
  - It send everybody a message saying "I just received a message with a timestamp T which leads to deadlock. If anyone has a message for me with an earlier timestamp, please send it immediately"
  - When every machine has replied, positively or negatively, the coordinator will see that the deadlock has really occurred or not.

Centralized Deadlock-Detection Algorithms

- The Ho-Ramamoorthy Algorithms
  - The Two-Phase Algorithm
  - The One-phase Algorithm

Centralized Algorithms

- Ho-Ramamoorthy 2-phase Algorithm
  - Each site maintains a status table of all processes initiated at that site: includes all resources locked & all resources being waited on.
  - Controller requests (periodically) the status table from each site.
  - Controller then constructs WFG from these tables, searches for cycle(s).
  - If no cycles, no deadlocks.
  - Otherwise, (cycle exists): Request for state tables again.
  - Construct WFG based *only* on common transactions in the 2 tables.
  - If the same cycle is detected again, system is in deadlock.
  - Later proved: cycles in 2 consecutive reports *need not* result in a deadlock. Hence, this algorithm detects false deadlocks.

Centralized Algorithms...

- Ho-Ramamoorthy 1-phase Algorithm
  - Each site maintains 2 status tables: *resource status* table and
    process status table.
  - Resource table: transactions that have locked or are waiting for resources.
  - Process table: resources locked by or waited on by transactions.
  - Controller periodically collects these tables from each site.
  - Constructs a WFG from transactions common to both the tables.
  - No cycle, no deadlocks.
  - A cycle means a deadlock.

Distributed Deadlock-Detection Algorithms

- A Path-Pushing Algorithm
  - The site waits for deadlock-related information from other sites
  - The site combines the received information with its local TWF graph to build an updated TWF graph
  - For all cycles ‘EX -> T1 -> T2 -> Ex’ which contains the node ‘Ex’, the site transmits them in string form ‘Ex, T1, T2, Ex’ to all other sites where a sub-transaction of T2 is waiting to receive a message from the sub-transaction of T2 at that site

Edge-Chasing Algorithm

- Chandy-Misra-Haas's Algorithm:
  - A probe(i, j, k) is used by a deadlock detection process Pi. This probe is sent by the home site of Pj to Pk.
  - This probe message is circulated via the edges of the graph. Probe returning to Pi implies deadlock detection.
  - Terms used:
    - Pj is *dependent* on Pk, if a sequence of Pj, Pi1,.., Pim, Pk exists.
    - Pj is *locally dependent* on Pk, if above condition + Pj,Pk on same site.
    - Each process maintains an array *dependenti: dependenti(j)* is true if Pi knows that Pj is dependent on it. (initially set to false for all i & j).

Chandy-Misra-Haas's Algorithm

Sending the probe:

if Pi is locally dependent on itself then deadlock. else
for all Pj and Pk such that
(a) Pi is locally dependent upon Pj, and
(b) Pj is waiting on Pk, and
(c) Pj and Pk are on different sites, send probe(i,j,k) to the home site of Pk.

Receiving the probe:

if (d) Pk is blocked, and
(e)dependentk(i) is false, and
(f)Pk has not replied to all requests of Pj, then
begin
dependentk(i) := true;
if k = i then Pi is deadlocked else ...

Chandy-Misra-Haas's Algorithm

Receiving the probe:

…….

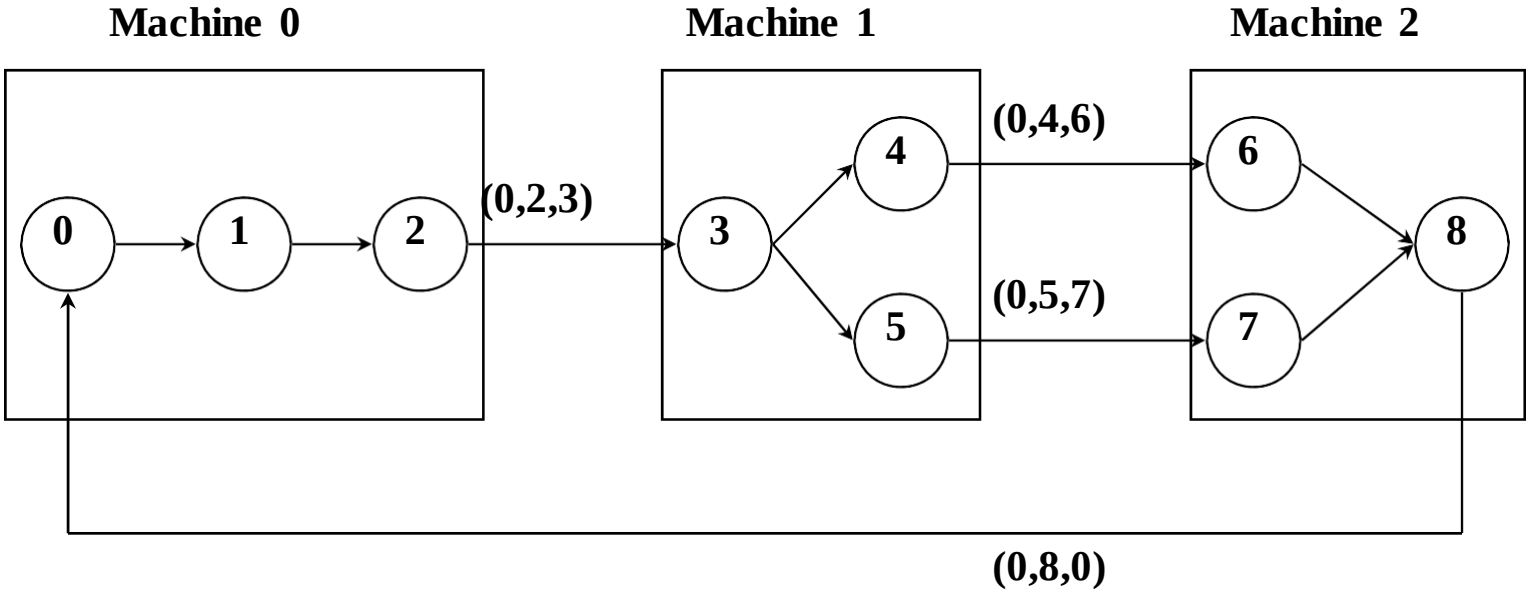
else for all Pm and Pn such that
(a') Pk is locally dependent upon Pm, and (b') Pm is waiting on Pn, and
(c') Pm and Pn are on different sites, send probe(i,m,n) to the home site of Pn.

end.

Performance:

For a deadlock that spans m processes over n sites, m(n-1)/2 messages are needed.
Size of the message 3 words. Delay in deadlock detection O(n).

Chandy-Misra-Haas Algorithm



- There are several ways to break the deadlock:
  - The process that initiates commit suicide -- this is overkilling because several process might initiates a probe and they will all commit suicide in fact only one of them is needed to be killed.
  - Each process append its id onto the probe, when the probe come back, the originator can kill the process which has the highest number by sending him a message. (Even for several probes, they will all choose the same guy)

Other Edge - Chasing Algorithms

- The Mitchell – Merritt Algorithm
- Sinha – Niranjan Algorithm