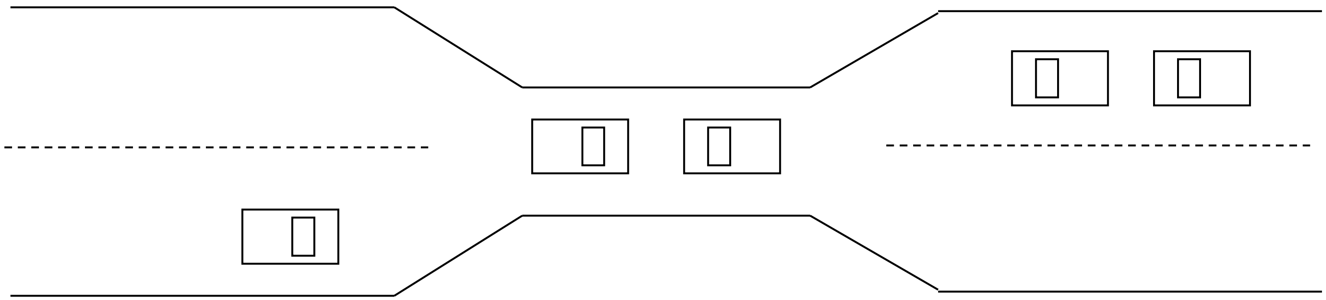
Deadlocks

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example1
  - ➔System has 2 tape drives
  - ➔P_1 and P_2 each hold one tape drive and each needs another one
- Example2
  - ➔Semaphores *A* and *B*, initialized to 1

P_0	P_1
Wait (A);	*Wait(B)*
Wait (B);	*Wait(A)*

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.

System Model

- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizing a resource:
  - Makes Request for resource
  - Uses the resource
  - Releases the resource

  - Request and release of resources is done through system calls

Deadlock Characterization

Deadlock can arise if following four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource(i.e. resource is non-sharable).
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that

  P_0 is waiting for a resource that is held by P_1
  P_1 is waiting for a resource that is held by P_2
  …, P_{n-1} is waiting for a resource that is held by P_n
  and P_n is waiting for a resource that is held by P_0.

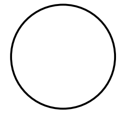
Resource-allocation Graph

A set of vertices V and a set of edges E.

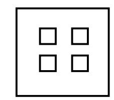
- V is partitioned into two types:
  - ➔ $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - ➔ $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Request edge – directed edge $P_{\mathrm{i}} \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

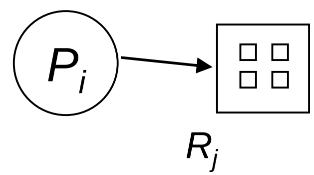
- Process



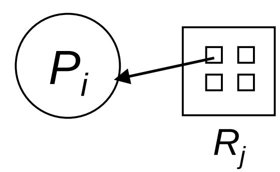
- Resource Type with 4 instances



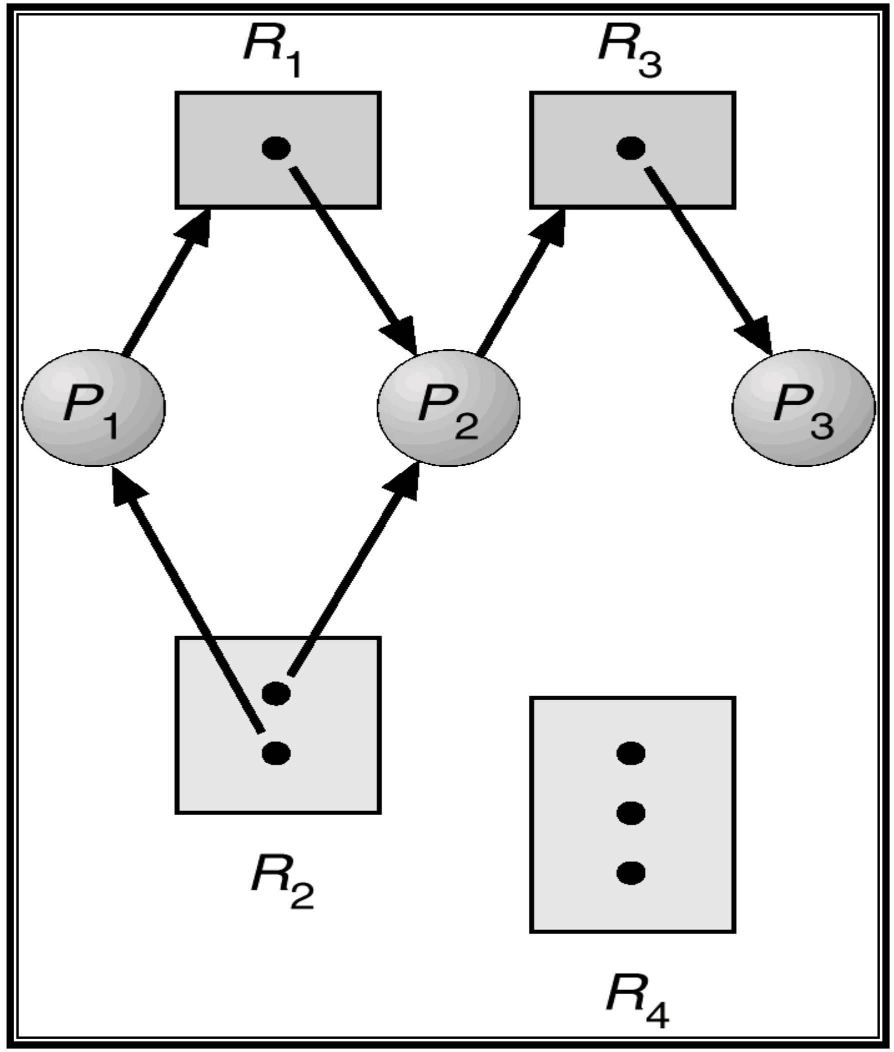
- P_i requests instance of R_j



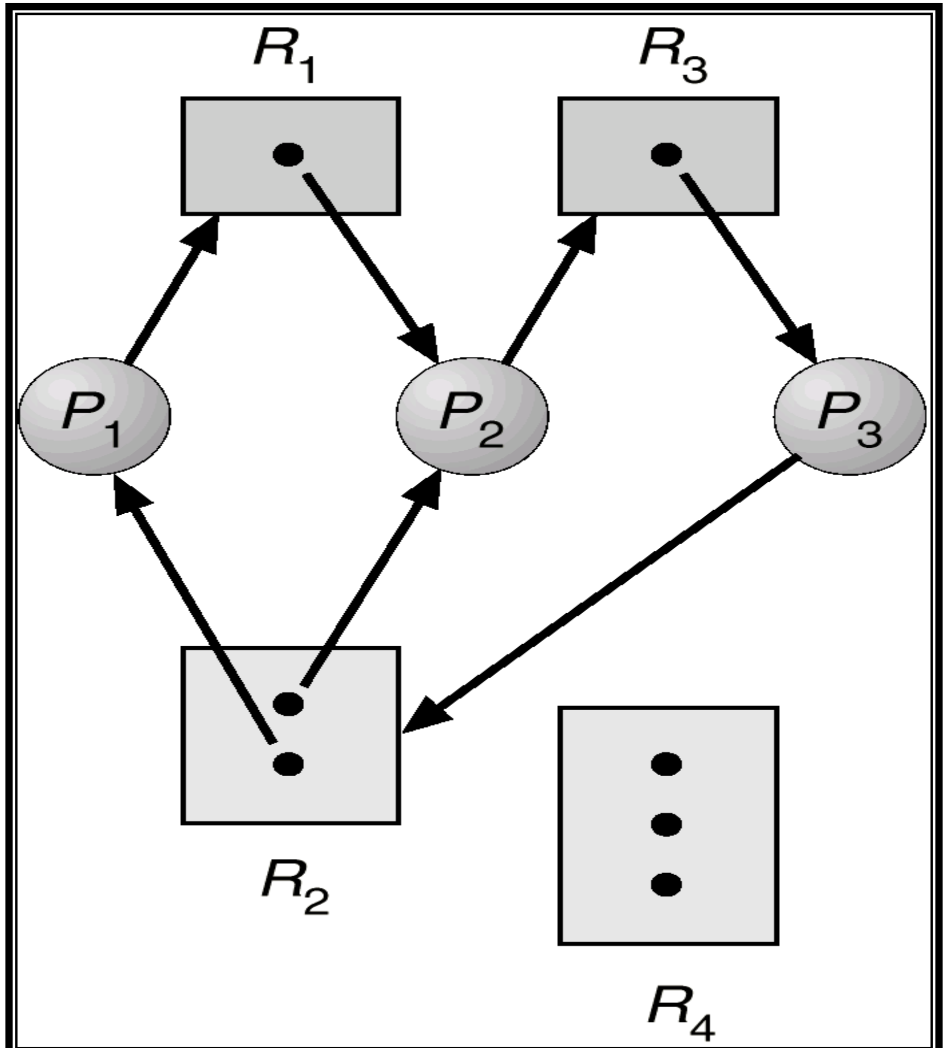
- P_i is holding an instance of R_j



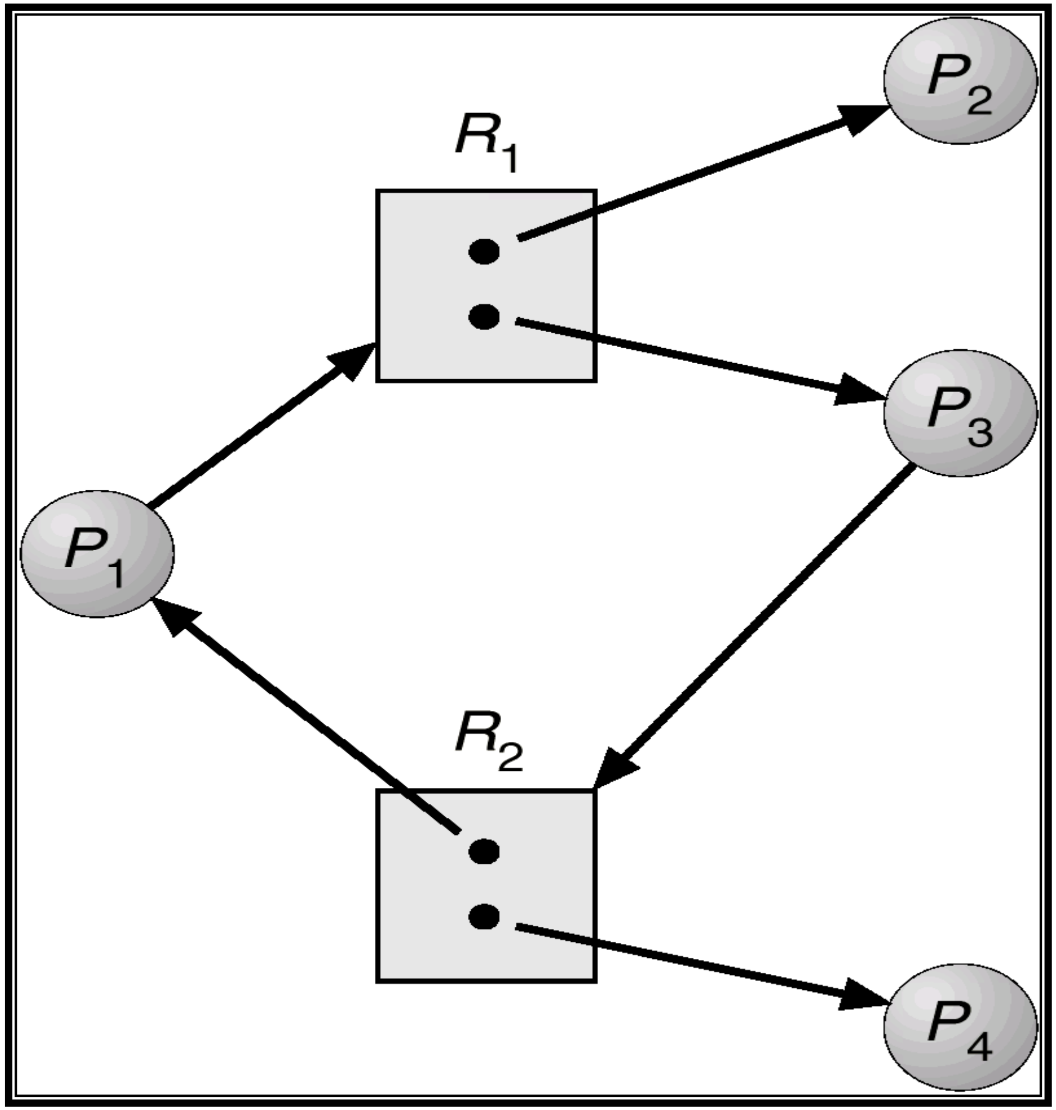
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- 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

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.

  (Prevention or Avoidance)

- Allow the system to enter a deadlock state and then Detect & recover.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**
- **Hold and wait**
- **No preemption**
- **Circular wait**

Deadlock Prevention

Ensure that one of the 4 necessary conditions for deadlock does not hold.

Constraints the way requests can be made for resources

- **Mutual exclusion**
  - not required for sharable resources;
  - Must hold for non-sharable resources.

➔ Make non-sharable resources as sharable

➔ some resources are intrinsically non sharable

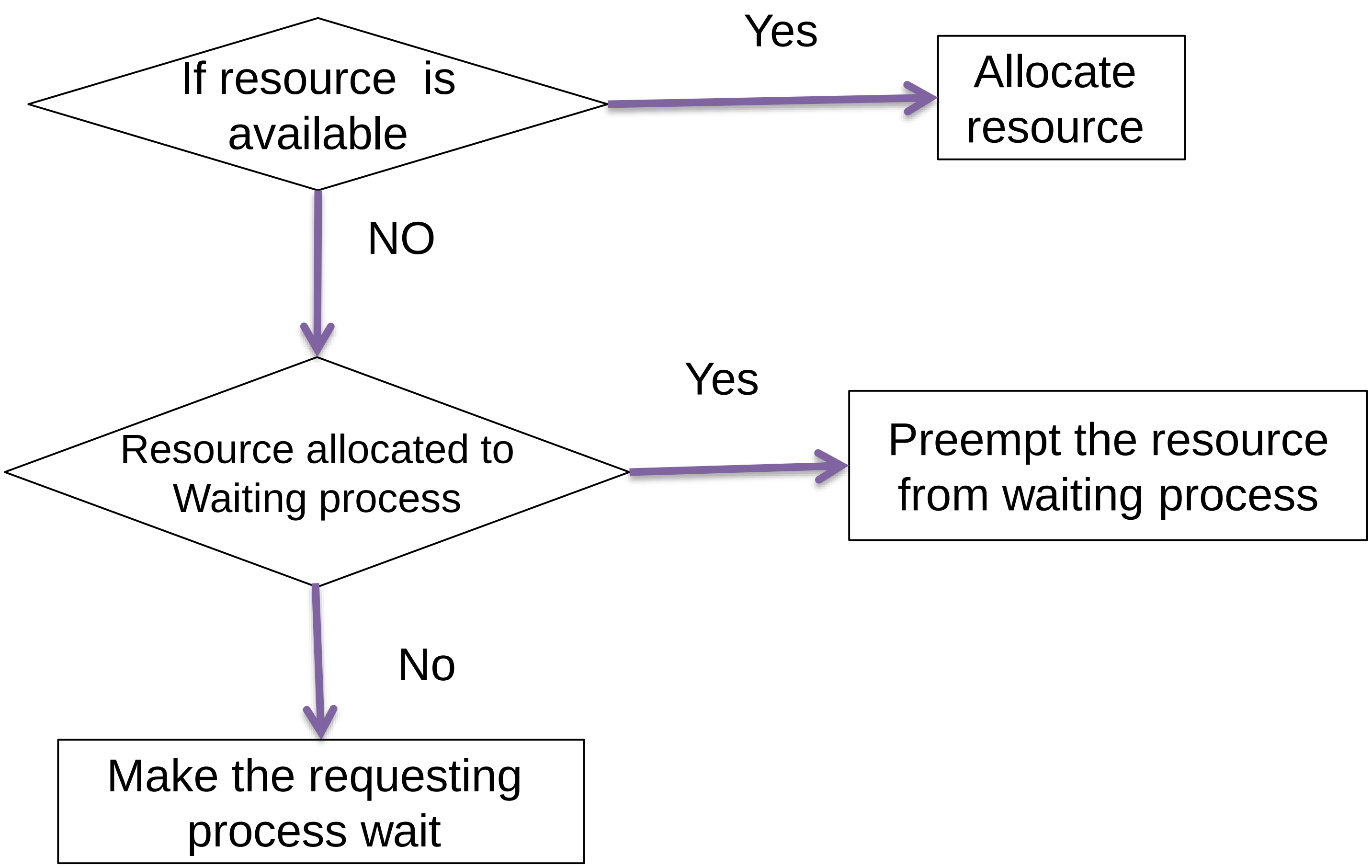
Deadlock Prevention

- **Hold and wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - ➔ Require process to request and be allocated all its resources before it begins execution,
    - ➢ Need to know all the requirement in the beginning itself
  - ➔ Allow process to request resources only when the process has none.
    - ➢ If a process has resource and requires additional resources, it must release all resources it is holding and then make request
  - ➔ Low resource utilization; Starvation possible

Deadlock Prevention (Cont.)

- **No preemption** – Approach -1
  - ➔ If a process that is holding some resources requests another resource that cannot be immediately allocated to it,
    - Release all resources currently being held by process
  - ➔ Preempted resources are added to the list of resources for which the process is waiting
  - ➔ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
  - ➔ Cannot be generally applied to resources like printers, tape drives etc.

No preemption (cont-)



Deadlock Prevention (Cont.)

- **Circular wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest model requires that each process declare the *maximum number* of resources of each type that it may need.

- The deadlock-avoidance algorithm
  - Dynamically finds the resource-allocation state in order to ensure that a circular-wait condition does not occur.

- Resource-allocation *state* is defined by
  - the number of available resources
  - the number of allocated resources,
  - the maximum requirement of the processes

Safe State

- When a process requests an available resource, the system needs to decide if immediate allocation leaves the system in a *safe state*.
- A state is safe if system can allocate resources to each process (Max requirement) in some order and still avoid deadlock

- System is in safe state if there exists a safe sequence of all processes.

- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

  → If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished

  → When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate

  → When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Example

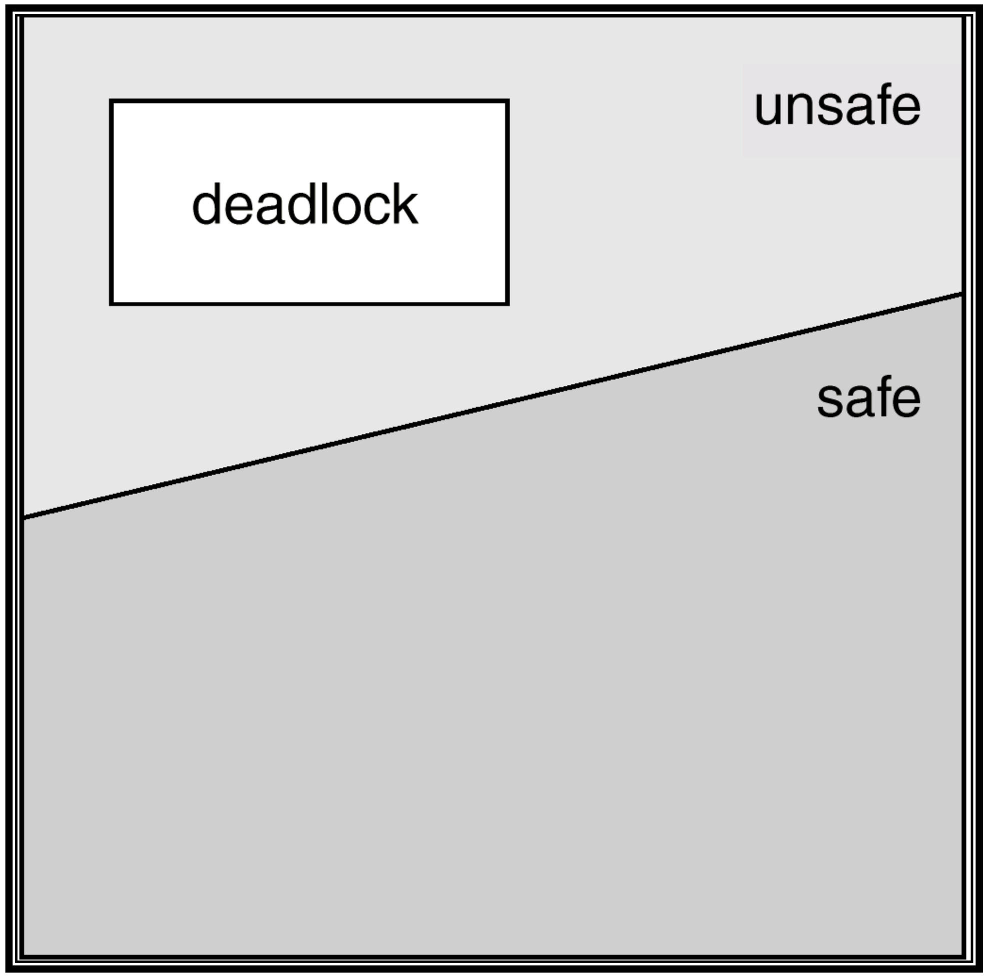
	Maximum need	Allocation
P0	10	5
P1	4	2
P2	9	2

Total no of resources =12
safe sequence < P1,P0,P2>

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state

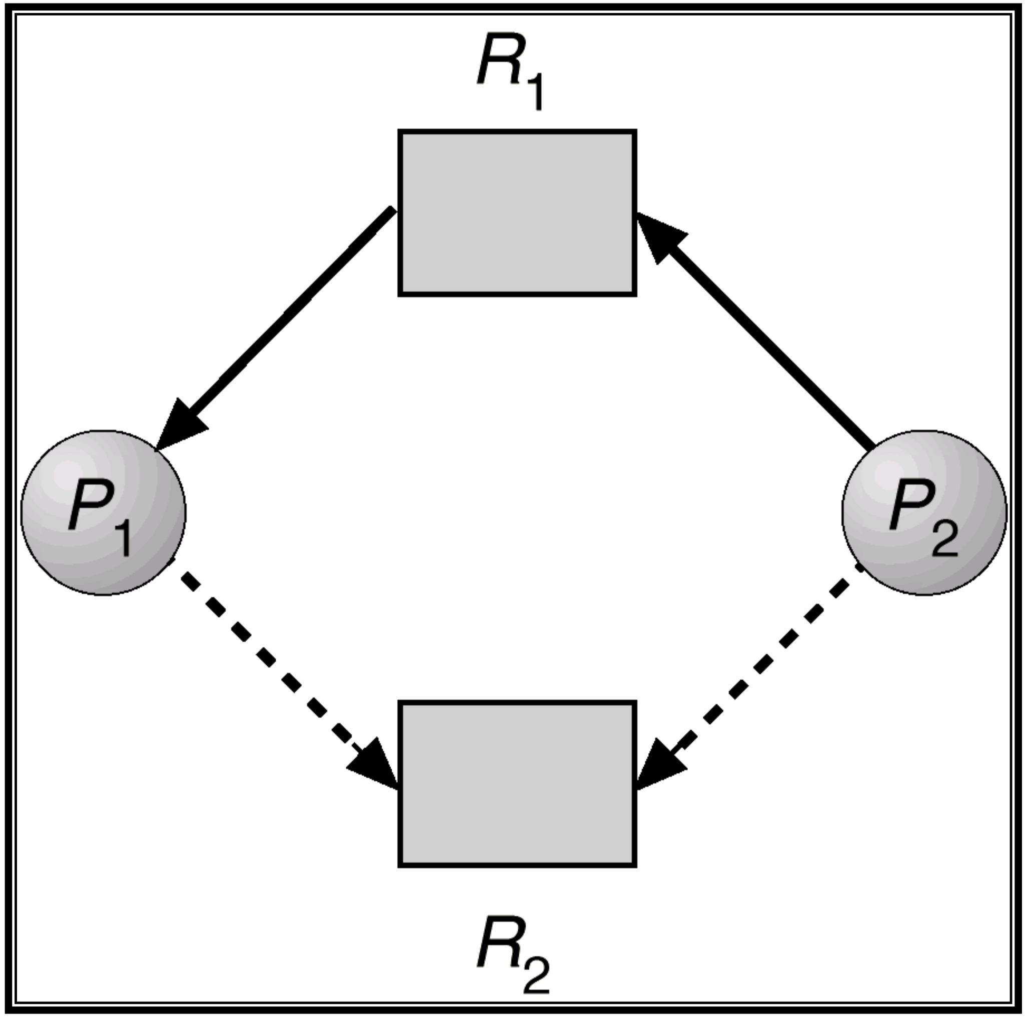
Safe, Unsafe , Deadlock State



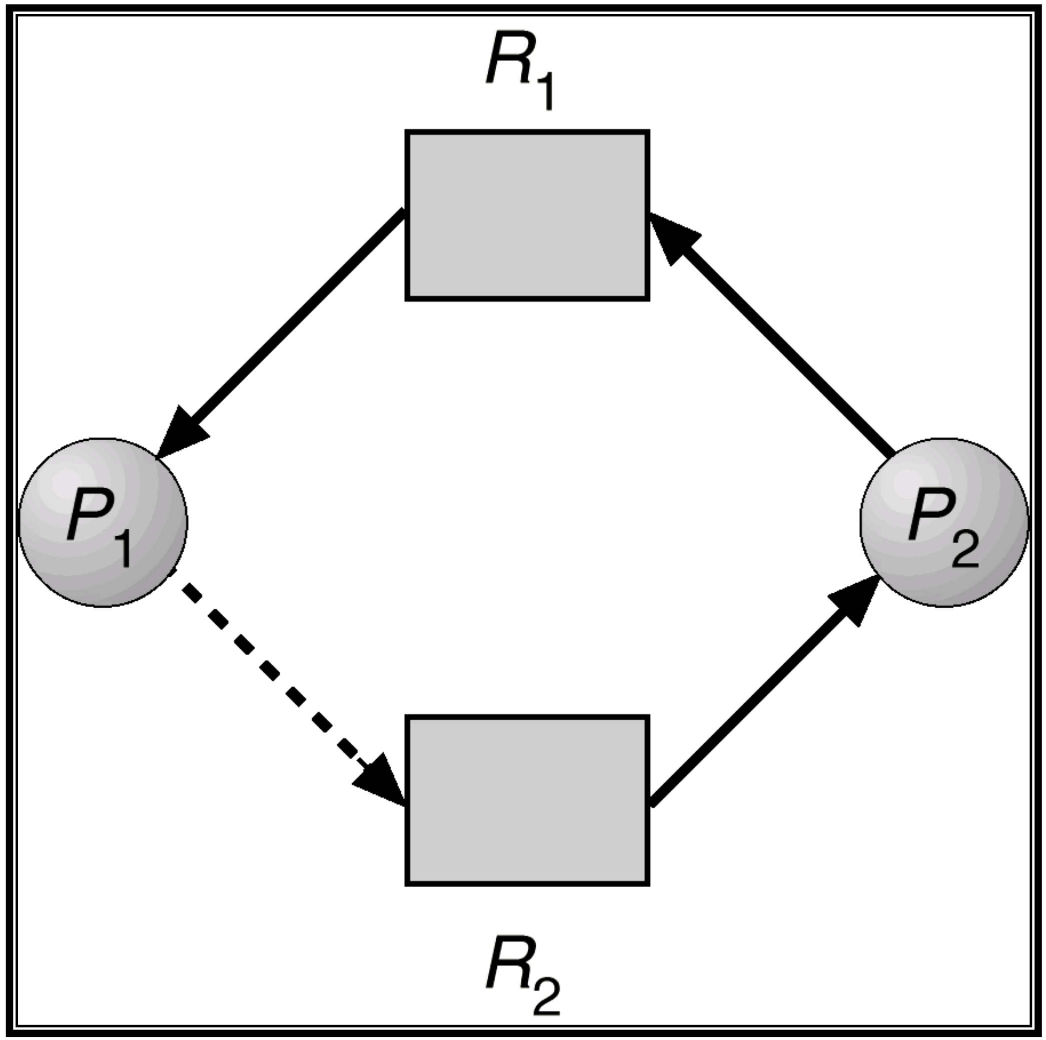
Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker’s Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- *Available:* Vector of length m. If Available [j] = k, there are k instances of resource type R_j available.
- *Max:* n x m matrix. If *Max* [I , j] = k, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* n x m matrix. If Allocation[I , j] = k then P_i is currently allocated k instances of $R_{j.}$
- *Need:* n x m matrix. If *Need*[I , j] = k, then P_i may need k more instances of R_j to complete its task.

Need [I , j] = *Max*[I , j] – *Allocation* [I , j]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* =0,1, …, *n-1.*

2. Find an index *i* such that both:

   (a) *Finish* [*i*] == *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Safe state example

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

(a) Initial state

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i.
If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ *then* go to step 2.
   Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3.
   Otherwise P_i must wait, since resources are not available.

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i$$

- *If safe* $\Rightarrow$ *the resources are allocated to* P_i.
- *If unsafe* $\Rightarrow$ P_i *must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5instances) and *C* (7 instances).
- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix Need is equal to Max – Allocation.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

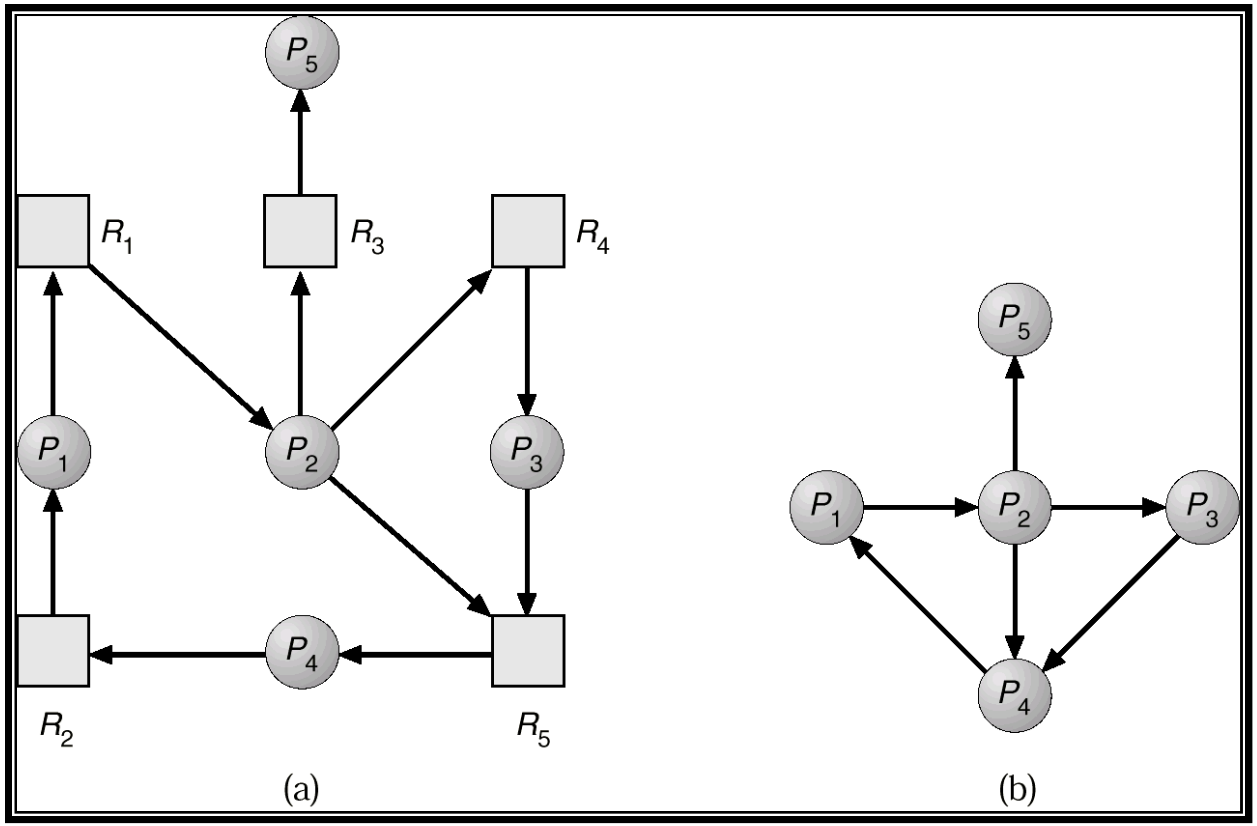
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - ➔Nodes are processes
  - ➔$P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph　　　Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length m indicates the number of available resources of each type.
- *Allocation:* An n x m matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An n x m matrix indicates the current request of each process. If *Request* $[i,j] = k$, then process P_i is requesting k more instances of resource type R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*,respectively. Initialize:
   (a) *Work = Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $<P_0, P_2, P_3, P_4, P_1>$ will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type C.

	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - ➔Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes requests.
  - ➔Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - ➔How often a deadlock is likely to occur?
  - ➔How many processes will need to be rolled back?
- Frequency of execution
  - Invoke the algorithm every time request for resource cannot be granted immediately.
  - Once per hour
  - When CPU utilization drops below 40%

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes ➔ May be expensive

- Abort one process at a time until the deadlock cycle is eliminated ➔ Invoke deadlock detection algorithm after every abort.

- In which order should we choose to abort?
  - ➔ Priority of the process.
  - ➔ How long process has computed, and how much longer to completion.
  - ➔ Resources the process has used.
  - ➔ Resources process needs to complete.
  - ➔ Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.