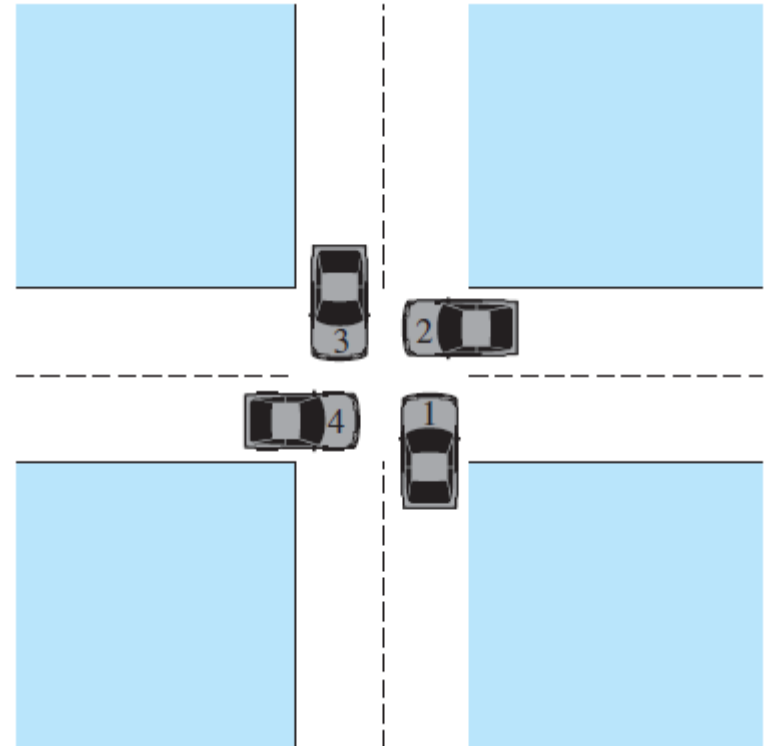


Module 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling



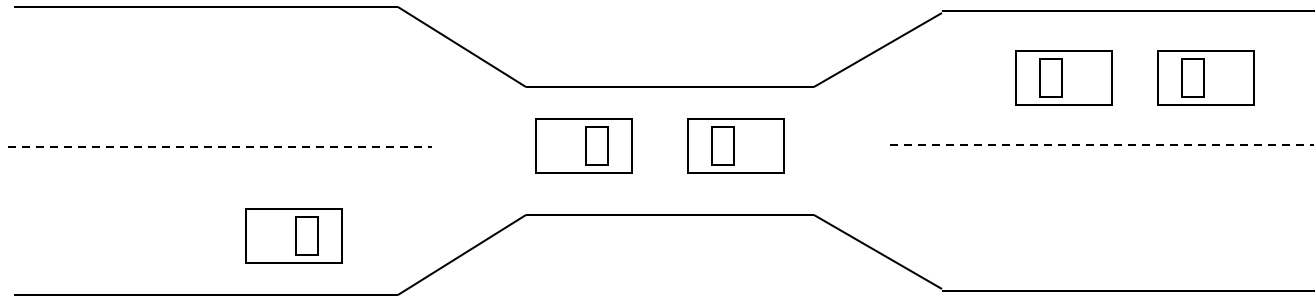
The Deadlock Problem

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

P_0
wait (A);
wait (B);

P_1
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, \dots, R_m
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - release

Resources

- Preemptable
 - Memory is an example.
- Non-Preemptable
 - If a process begun to burn a CD-ROM, suddenly taking the CD recorder away from it result in a garbled CD.

Deadlock

- A programmer who is developing multithreaded applications must pay particular attention to this problem: Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
 - **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 $\{P_0, P_1, \dots, P_n\}$ 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 .
- If one or more of these conditions are not met, deadlock cannot occur.

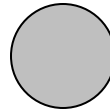
Resource-Allocation Graph

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

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

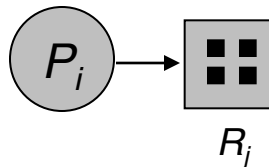
- Process



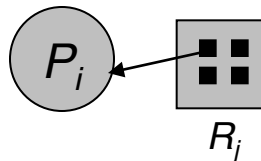
- Resource Type with 4 instances



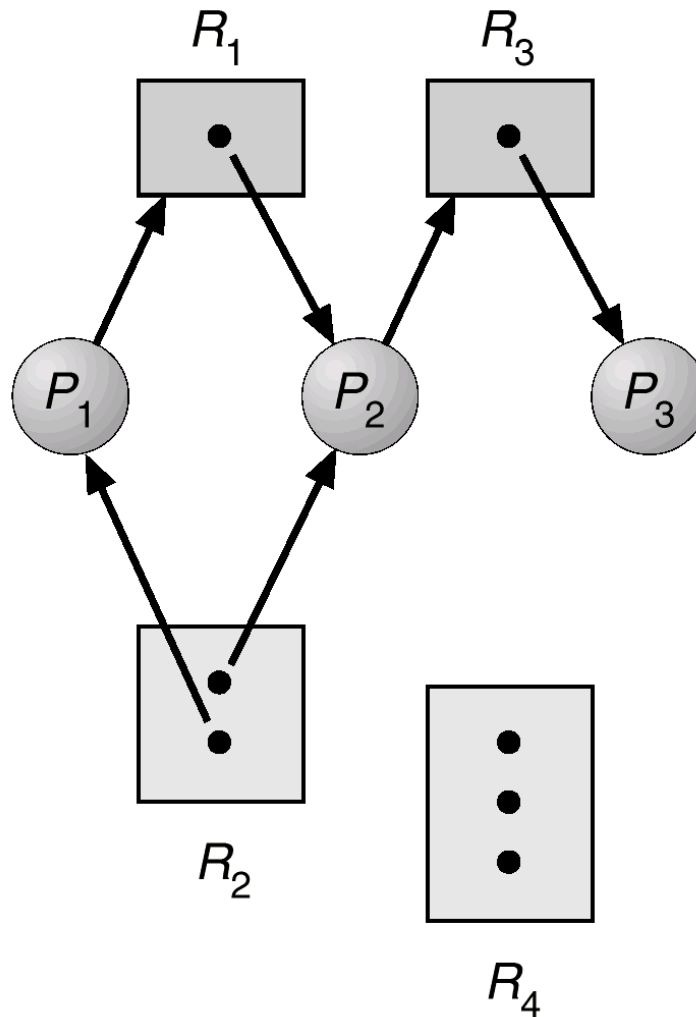
- P_i requests instance of R_j



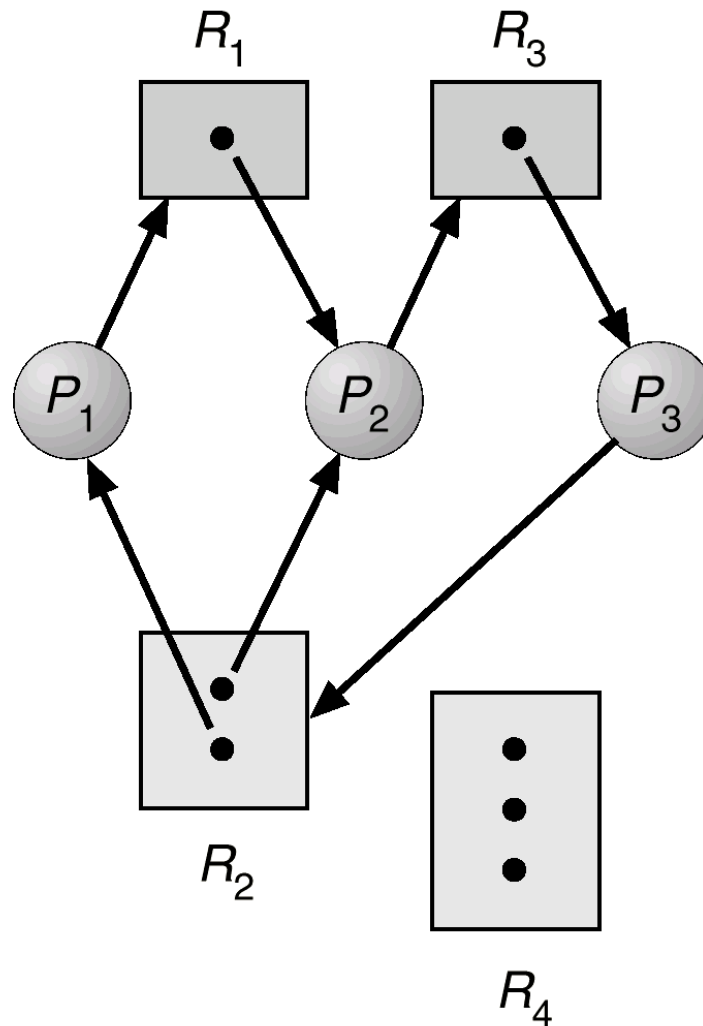
- P_i is holding an instance of R_j



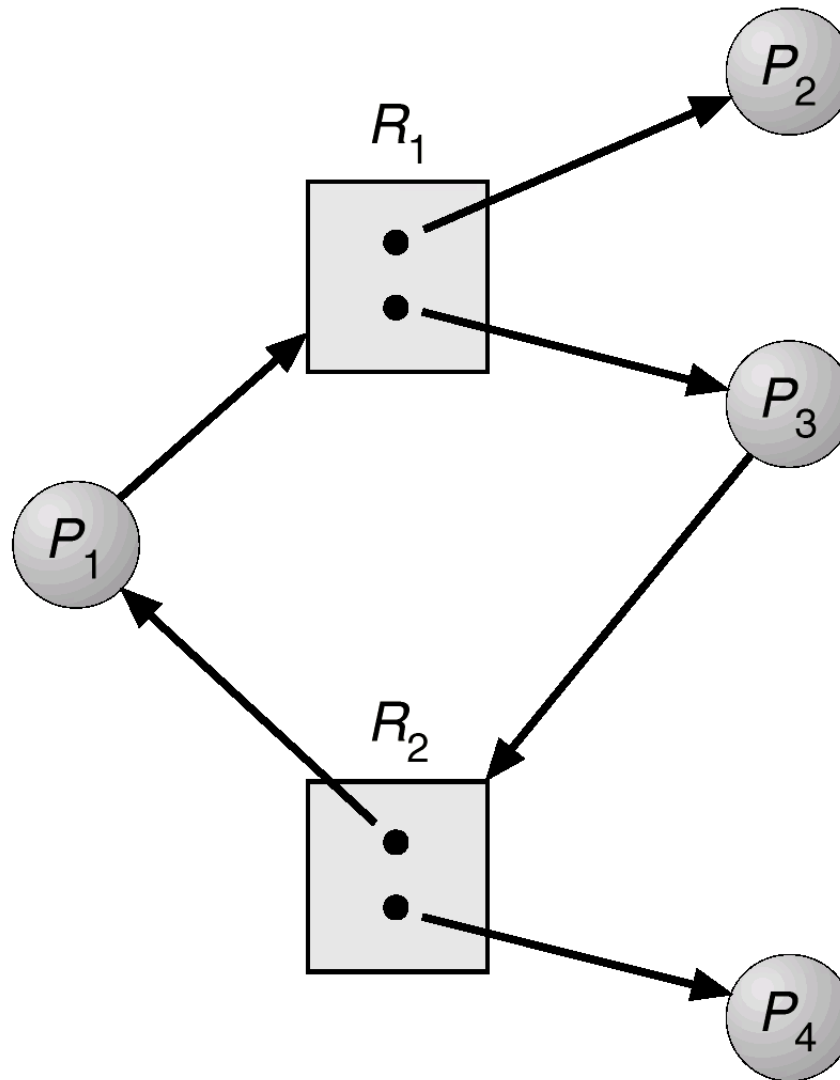
Example of a Resource Allocation Graph



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.

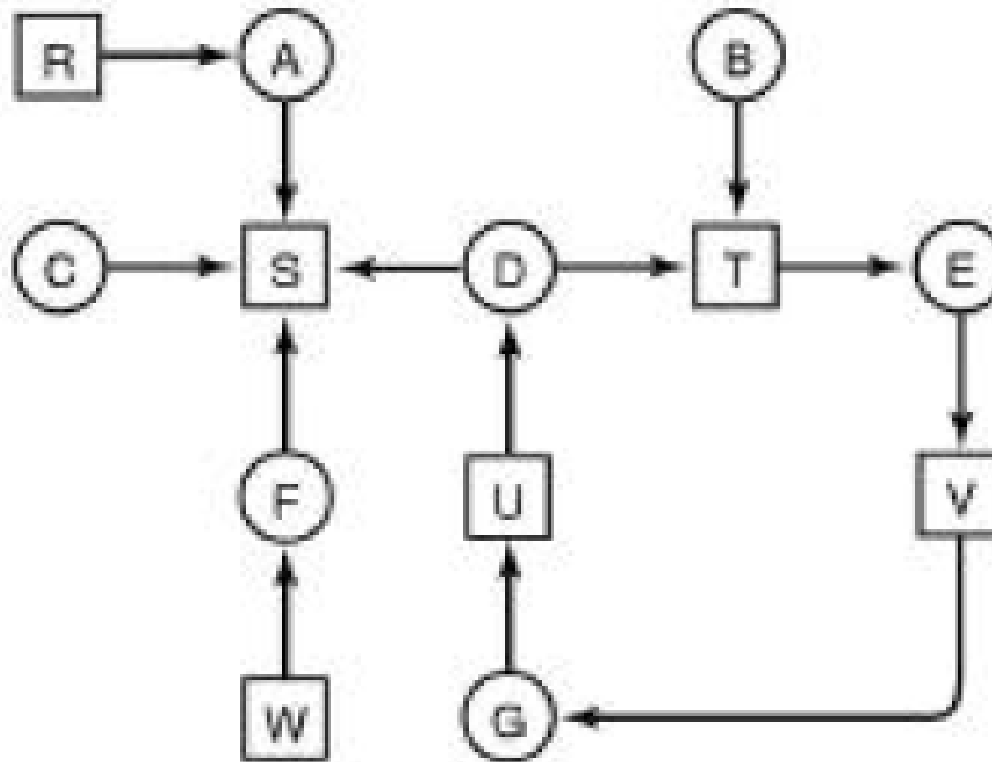
Possibility of Deadlock

1. Mutual exclusion
2. No preemption
3. Hold and wait

Existence of Deadlock

1. Mutual exclusion
2. No preemption
3. Hold and wait
4. Circular wait

Any Deadlock?



Methods for Handling Deadlocks

- Three general approaches exist for dealing with deadlock.
 - First, one can **prevent** deadlock by adopting a policy that eliminates one of the conditions (conditions 1 through 4).
 - Second, one can **avoid** deadlock by making the appropriate dynamic choices based on the current state of resource allocation.
 - Third, one can attempt to **detect** the presence of deadlock (conditions 1 through 4 hold) and take action to recover.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
 - a printer cannot be simultaneously shared by several processes.
 - Read-only files are a good example of a sharable resource.
- **we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable.**
- **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, or allow process to request resources only when the process has none.
 - Low resource utilization; starvation possible.

Example

- Consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer.
- **Protocol 1:** the process must initially request the tape drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- **Protocol 2:** the process requests initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Deadlock Prevention (Cont.)

- **No Preemption** –
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
 - 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.

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.**

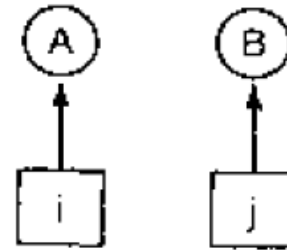
$$\begin{aligned}F(\text{tape drive}) &= 1, \\F(\text{disk drive}) &= 5, \\F(\text{printer}) &= 12.\end{aligned}$$

- For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

Deadlock Prevention (Cont.)

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

(a)



(b)

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 3-13(b). We can get a deadlock only if *A* requests resource *j* and *B* requests resource *i*. Assuming *i* and *j* are distinct resources, they will have different numbers. If $i > j$, then *A* is not allowed to request *j* because that is lower than what it already has. If $i < j$, then *B* is not allowed to request *i* because that is lower than what it already has. Either way, deadlock is impossible.

Summary of Deadlock Prevention

Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Deadlock Avoidance

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

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Deadlock Avoidance

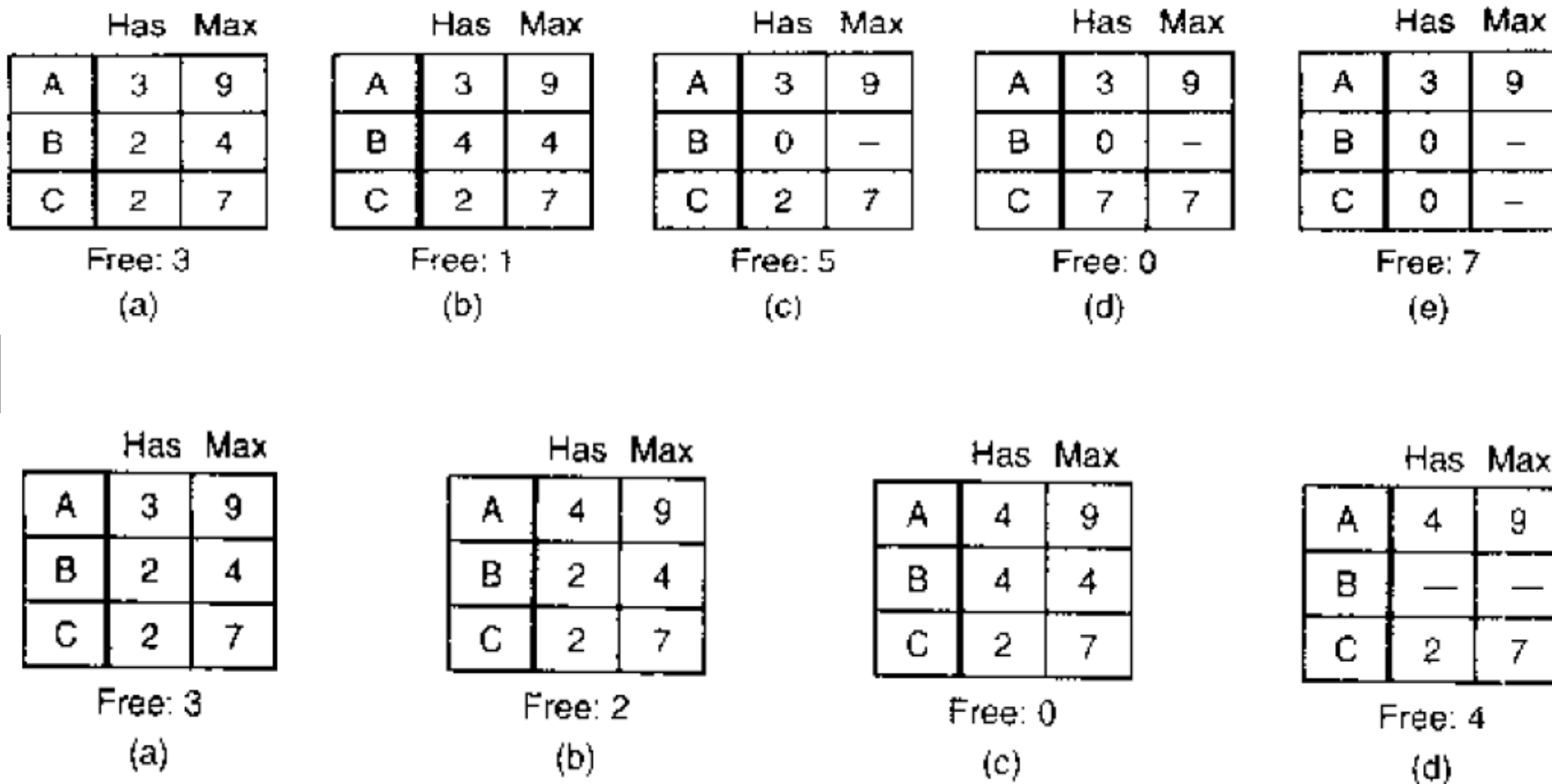


Figure 3-10. Demonstration that the state in (b) is not safe.

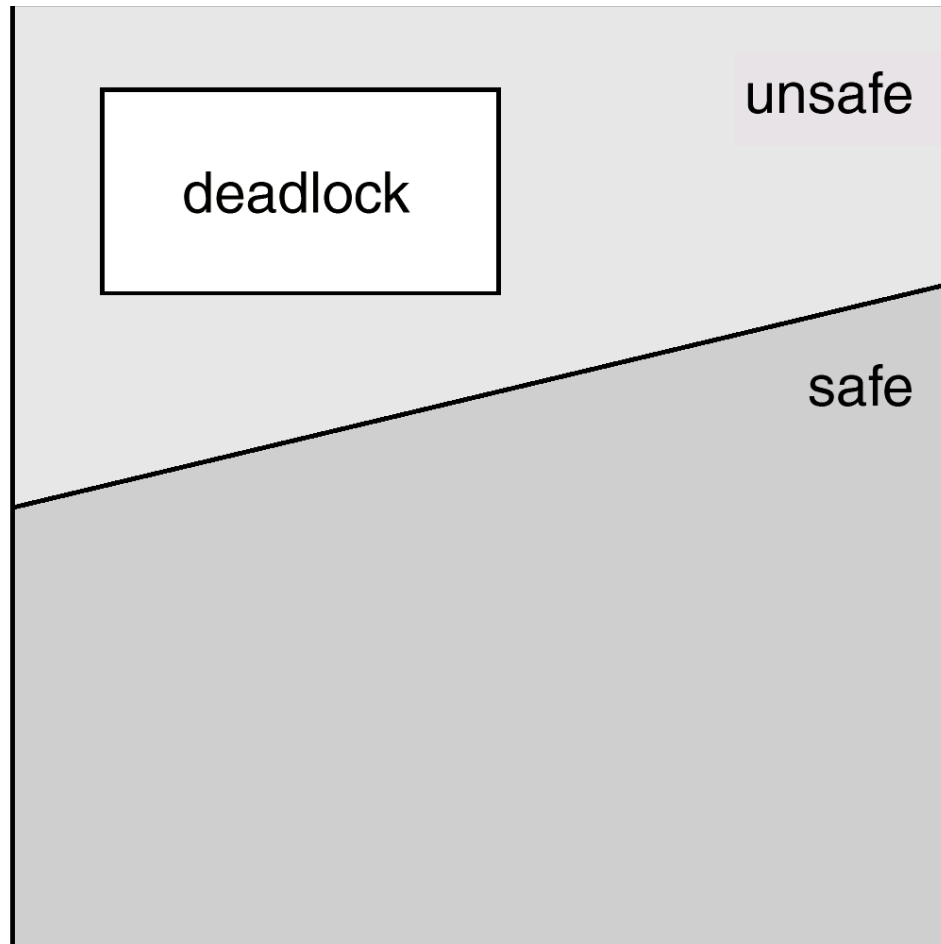
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ 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.

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.

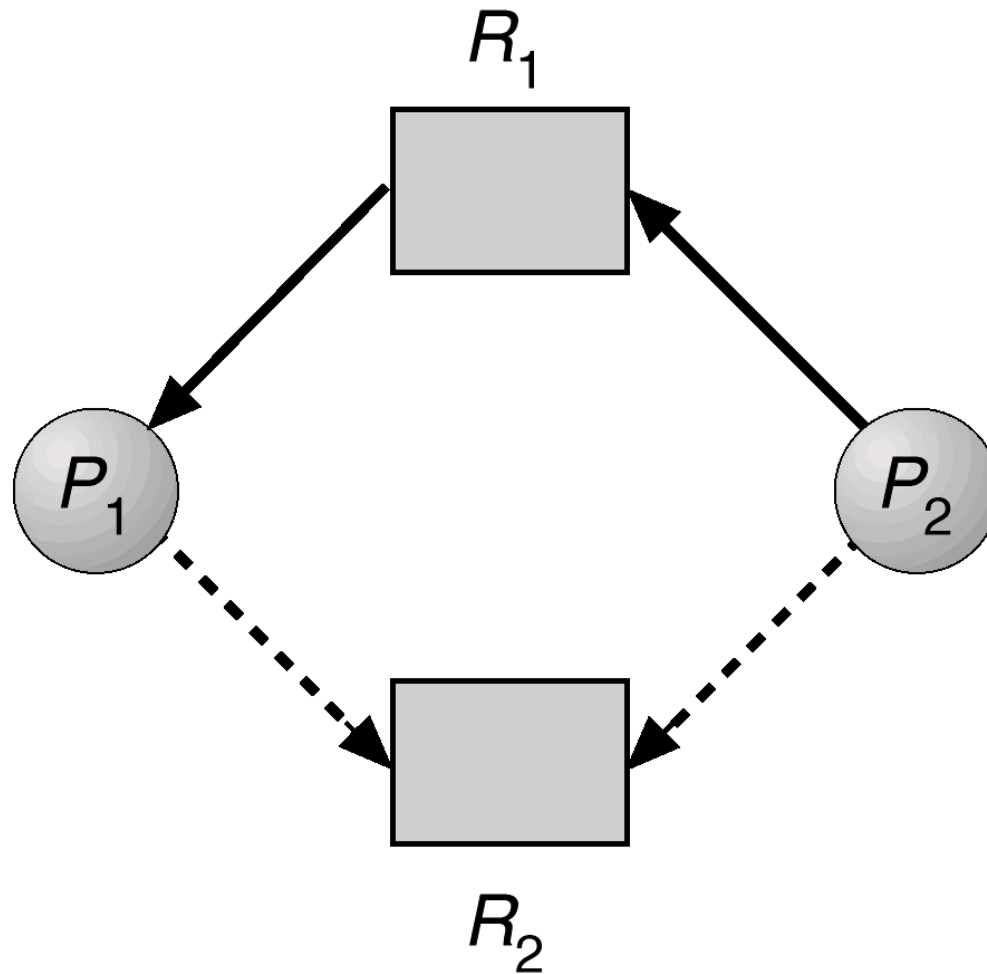
Safe, unsafe , deadlock state spaces



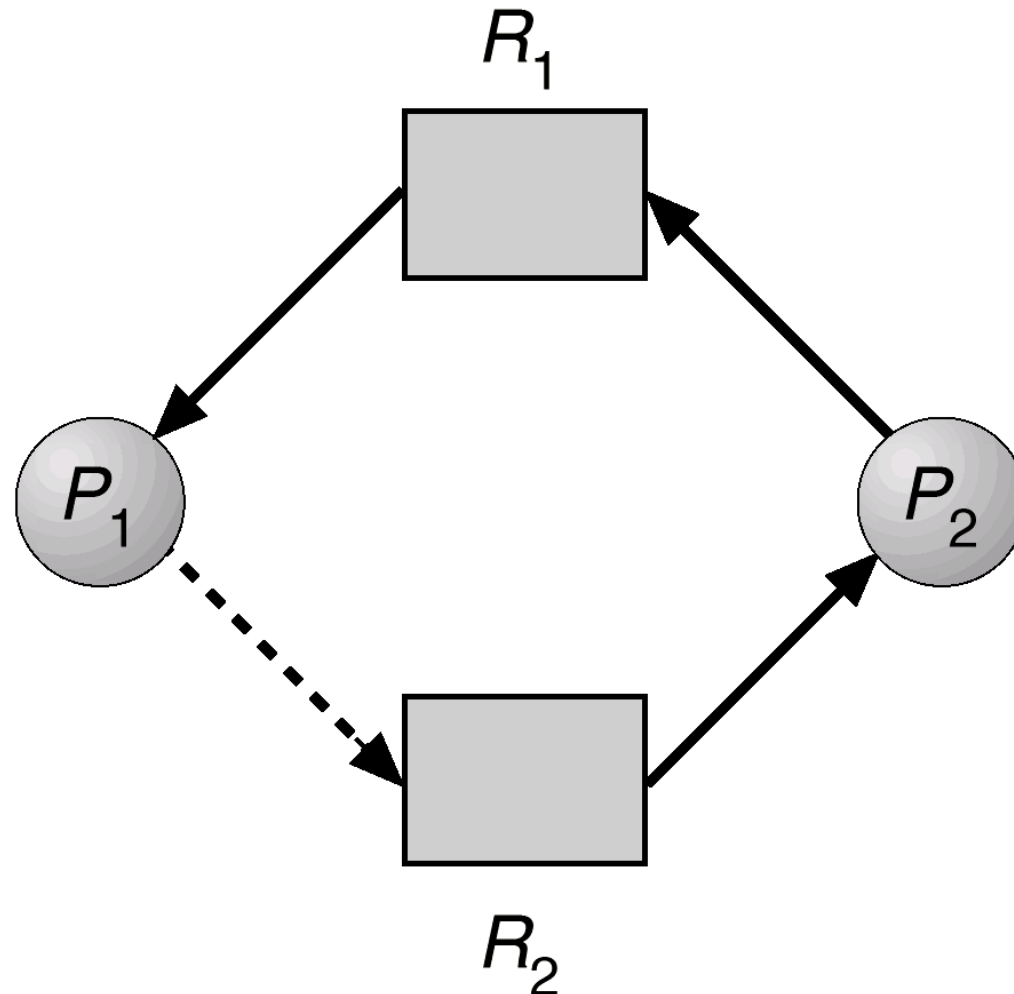
Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated 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 A 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*: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- *Allocation*: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j .
- *Need*: $n \times m$ 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* = 1, 3, ..., *n*.

2. Find and *i* such that both:

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

(b) *Need*_{*i*} ≤ *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.

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 $Request_i \leq Need_i$ 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 Determination of a Safe State

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

Claim matrix C

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

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

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

Claim matrix C

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

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

Example Determination of a Safe State

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

Claim matrix C

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

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

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

Claim matrix C

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

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

Example Determination of an Unsafe State

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

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

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

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

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

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

Example of Banker's Algorithm

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

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
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 defined to be Max – Allocation.

	<u>Need</u>
	<i>A B C</i>
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

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example (Cont.): P_1 request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$).

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

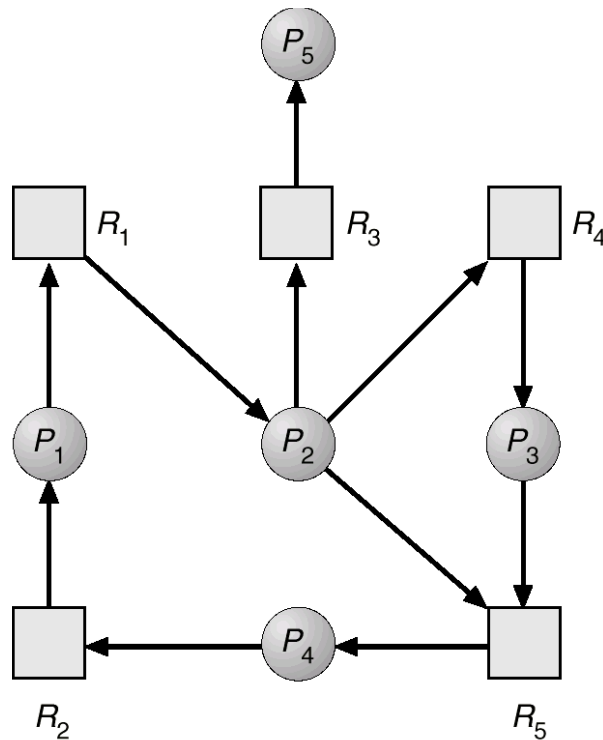
Deadlock Detection

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

Single Instance of Each Resource Type

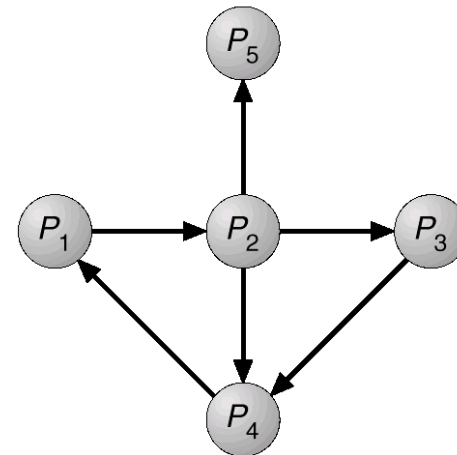
- Maintain *wait-for* graph
 - Nodes are processes.
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j .
- Periodically invoke an algorithm that searches for acycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph And Wait-for Graph



(a)

Resource-Allocation Graph



(b)

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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- *Request*: An $n \times m$ matrix indicates the current request of each process. If $Request[ij] = k$, 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 $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] := \text{false}$; otherwise, $Finish[i] := \text{true}$.

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

- (a) $Finish[i] = \text{false}$
- (b) $Request_i \leq Work$

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

Detection Algorithm (Cont.)

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.

Algorithm requires an order of $m \times n^2$ operations to detect whether the system is in deadlocked state.

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 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i .

Example (Cont.)

- P_2 requests an additional instance of type C .

	<u>Request</u>		
	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?
 - * one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- 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.
 - How many processes will need to be terminated.
 - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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

Combined Approach to Deadlock Handling

- Combine the three basic approaches
 - prevention
 - avoidance
 - detection

allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.