

Chapter 7: Deadlocks





Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock





Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system





The Deadlock Problem

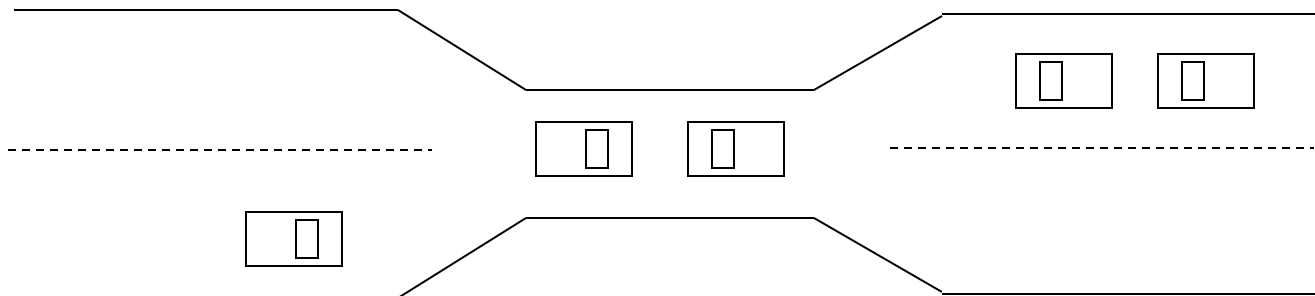
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
 - System has 2 disk drives
 - P_1 and P_2 each hold one disk drive and each needs another one
- Example
 - semaphores A and B , initialized to 1

P_0	P_1
acquire(A);	acquire(B)
acquire(B);	acquire (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
- Note – Most OSes do not prevent or deal with deadlocks





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**





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 .





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 $R_j \rightarrow P_i$



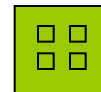


Resource-Allocation Graph (Cont.)

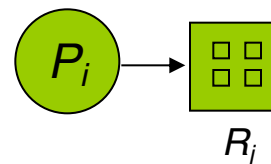
■ Process



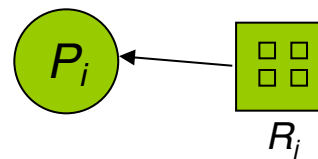
■ Resource Type with 4 instances



■ P_i requests instance of R_j

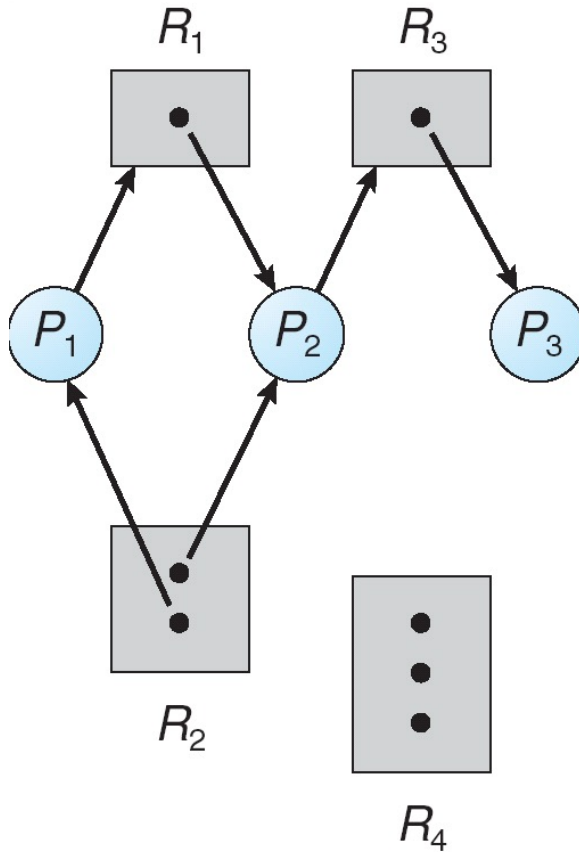


■ P_i is holding an instance of R_j





Example of a Resource Allocation Graph



$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

- One instance of resource type R_1 ,
- Two instances of resource type R_2 ,
- One instance of resource type R_3 ,
- Two instances of resource type R_4

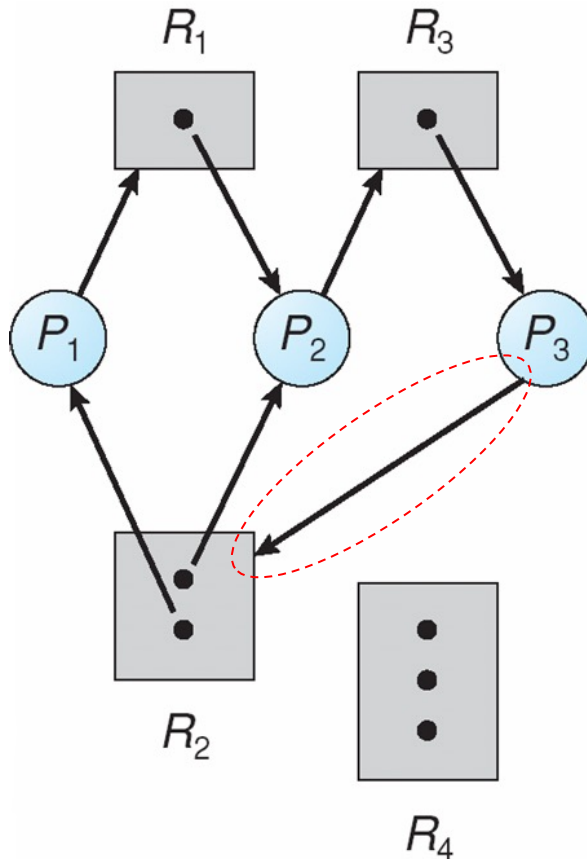
Process states

- P_1 is holding an instance of R_2 and waiting for an R_1
- P_2 is holding an R_1 and an R_2 and is waiting for an R_3
- P_3 is holding an R_3





Resource Allocation Graph With A Deadlock



Suppose P_3 requests an instance of resource type R_2 .

Two cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

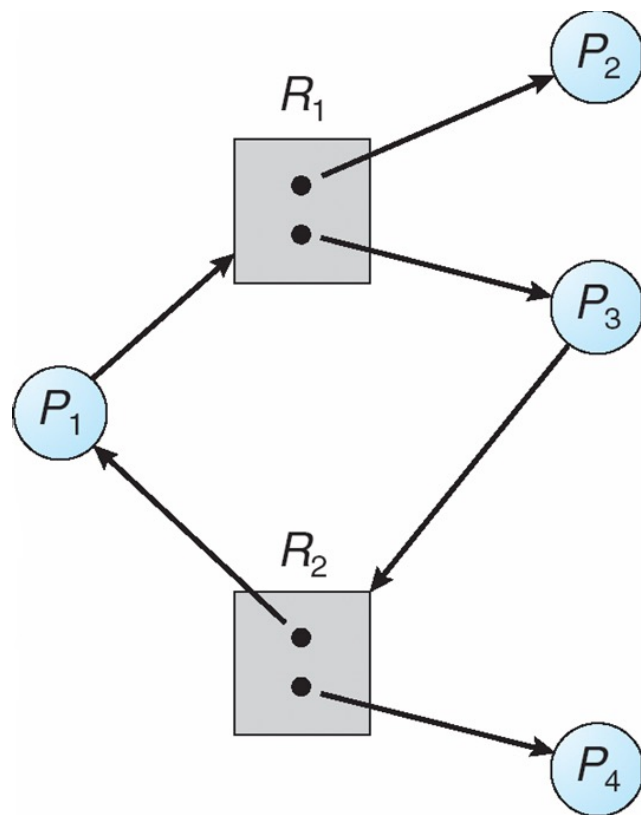
$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_1 , P_2 , P_3 are deadlocked.





Graph With A Cycle But No Deadlock



We have a cycle but no deadlock.

Process P_4 may release its instance of resource type R_2 . R_2 can then be allocated to P_3 , breaking the cycle.





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
- Allow the system to *enter* a deadlock state and then *recover*
- *Ignore* the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX





Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (i.e. a printer); must hold for nonsharable resources (i.e. read-only files)
- **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





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

■ 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 on how resources are to be requested.

- 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





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 **sequence** $\langle P_1, P_2, \dots, P_n \rangle$ of **ALL the processes** in the systems such that 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 $j < i$
- That is:
 - 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





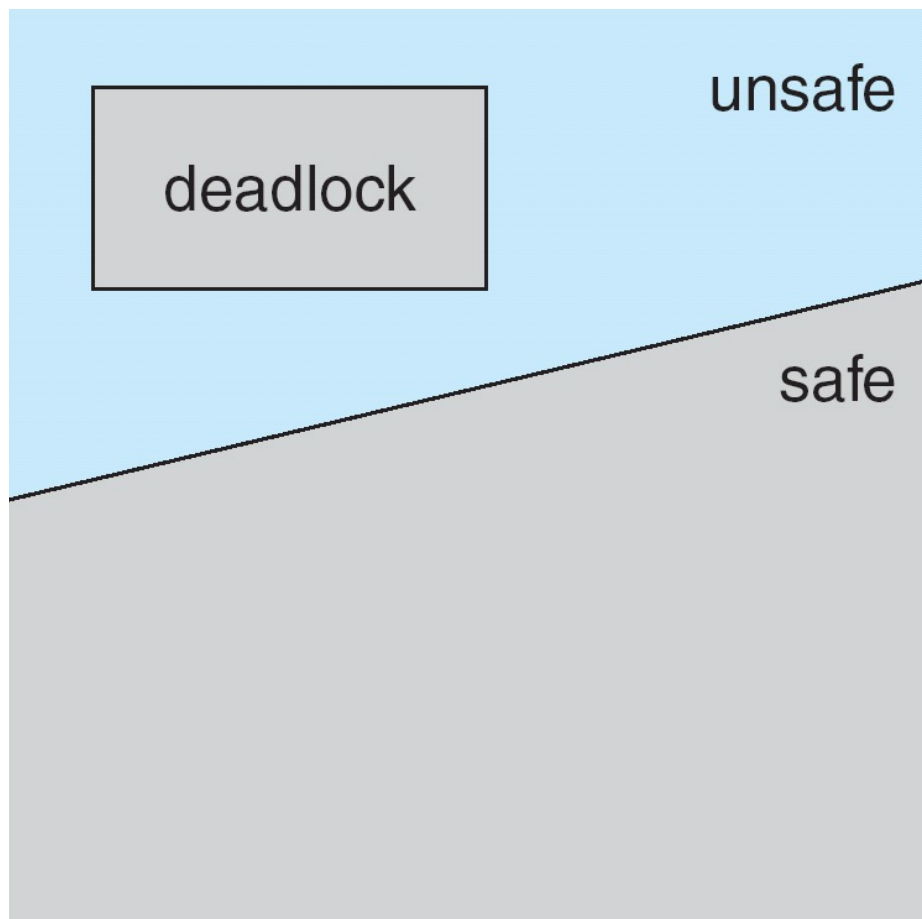
Basic Facts

- 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





Avoidance algorithms

- **Single instance** of a resource type
 - Use a resource-allocation graph

- **Multiple instances** of a resource type
 - Use the banker's algorithm





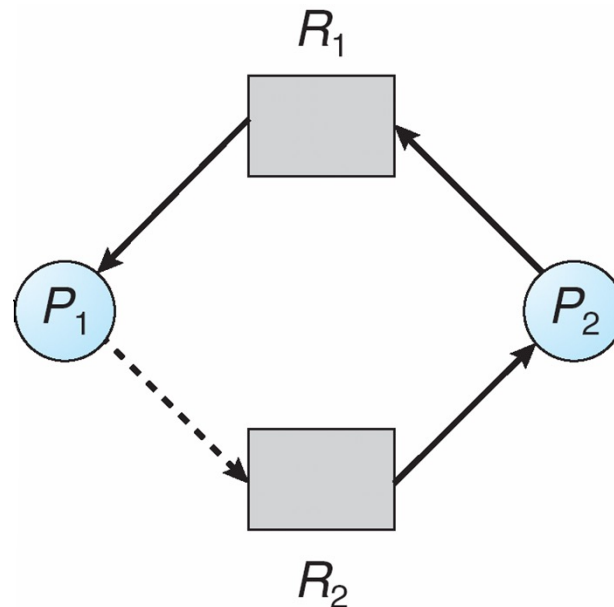
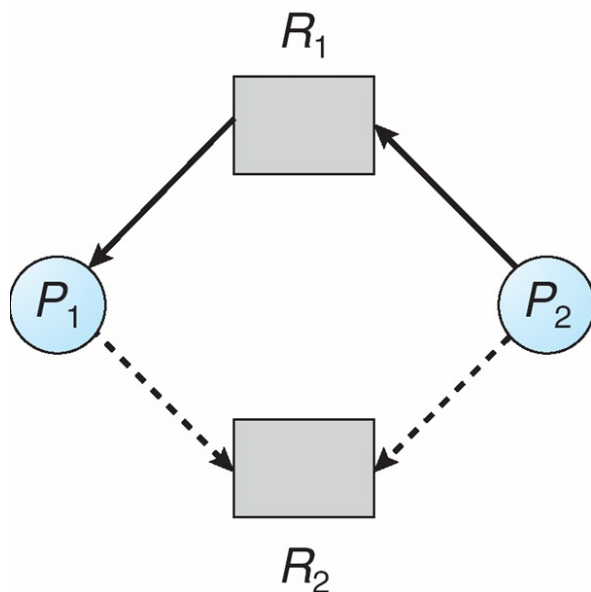
Resource-Allocation Graph Scheme

- **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
- Request edge converted to an assignment edge when the resource is allocated to the process
- 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



Unsafe State In Resource-Allocation Graph

Suppose that P2 requests R2. although R2 is currently free, we can not allocate it to P2, since this action may create a cycle if P1 requests R2 as well.





Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if **converting** the request edge to an assignment edge **does not result in the formation of a cycle** in the 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:** # of available resources of each type. Vector of length m . If available $[j] = k$, there are k instances of resource type R_j available
- **Max:** maximum demand of each process. $n \times m$ matrix. If $Max[i, j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** # of resources of each type currently allocated to each process. $n \times m$ matrix. If $Allocation[i, j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** the remaining resource need of each process. $n \times m$ 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 $i = 0, 1, \dots, n-1$

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





Resource-Request Algorithm for Process P_i

$Request$ = 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 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><i>Need</i></u>		
	<i>A</i>	<i>B</i>	<i>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: P_1 Request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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





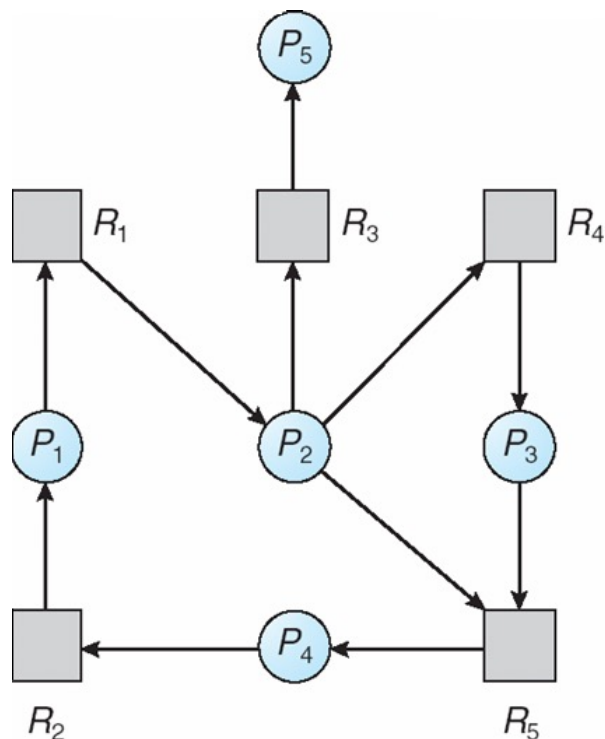
Single Instance of Each Resource Type

- 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 a cycle in the graph. If there is a cycle, there exists a deadlock
- 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



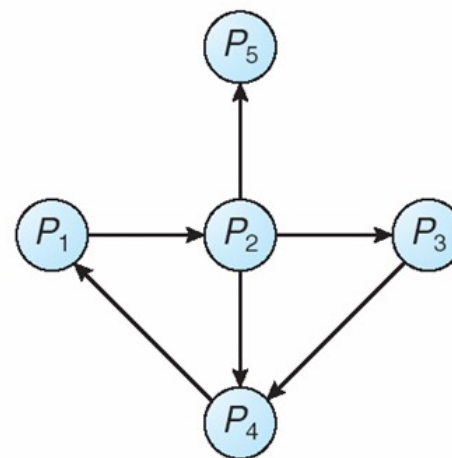


Resource-Allocation Graph and Wait-for Graph



(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 $Request[i_j] = k$, 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 $i = 1, 2, \dots, n$, 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





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 $O(m \times n^2)$ 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?
 - $\langle P_0, P_2, P_3, P_1, P_4 \rangle$
 - Can reclaim resources held by process P_0 , but still 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 be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to the deadlocked processes will be idle until the deadlock can be broken.





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 for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor



End of Chapter 7

