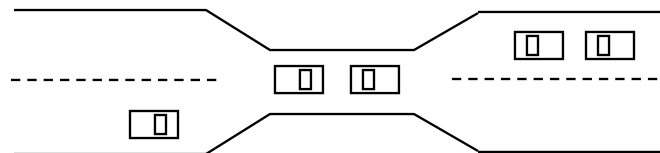


Chapter 8: Deadlocks

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

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

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example of a deadlock problem:
 - System has 2 tape drives.
 - Process P_1 and process P_2 each hold one tape drive and each needs another one.
- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- The resources may be either physical (printers, tape drives, memory space, and CPU cycles) or logical (files and semaphores).
- A process must request a resource before using it, and must release the resource after using it.

System Model

- A process may utilize a resource in only the following sequence:
 1. Request: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
 2. Use: The process can operate on the resource (ex., print on printer).
 3. Release: The process releases the resource.
- The request and release of resources are system calls (examples: request and release device, open and close file, and allocate and free memory system calls).
- Request and release of other resources can be accomplished through the wait and signal operations on semaphores.

Deadlock Characterization




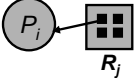
Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait:** a process is holding at least one resource and is waiting to acquire additional resources held by other processes (i.e., you hold a resource and wait for another one).
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task (ex., in FCFS a process use the CPU until it terminates).
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_{n-1}\}$ 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_0 is waiting for a resource that is held by P_0 .

Resource-Allocation Graph

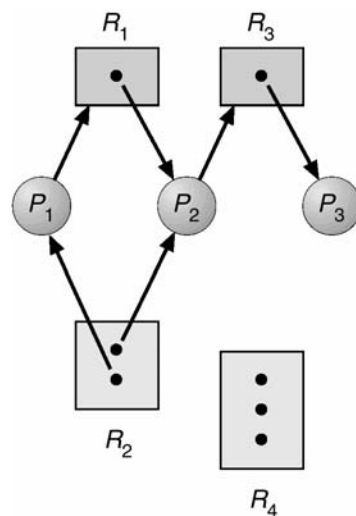
- A deadlock can be described in terms of a directed graph called system resource-allocation graph.
- A set of vertices V and a set of edges E .
 - V is partitioned into two types:
 - * $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - * $R = \{R_1, R_2, \dots, R_m\}$, 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 of same type:
For example four printers 
- P_i requests instance of R_j 
- P_i is holding an instance of R_j 

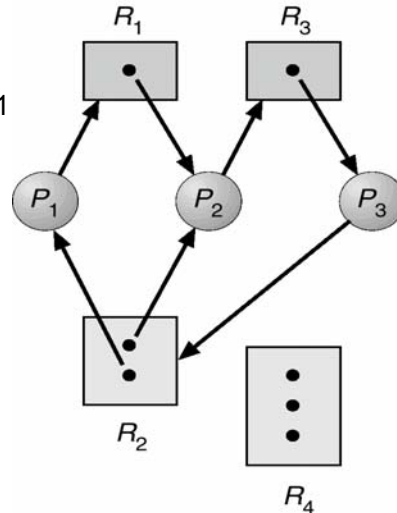
Example of a Resource Allocation Graph

- Process State:
 - P1 is holding an instance of R2 and waiting for an instance of R1.
 - P2 is holding an instance of R1 and R2, and is waiting for an instance of R3.
 - P3 is holding an instance of R3.
- The graph does not contain any cycles. What does that mean?



Resource Allocation Graph With A Deadlock

- Two minimal 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.
 - P_2 is waiting for R_3 , which is held by P_3 .
 - P_3 is waiting for P_1 or P_2 to release R_2 .
 - P_1 is waiting for P_2 to release R_1 .

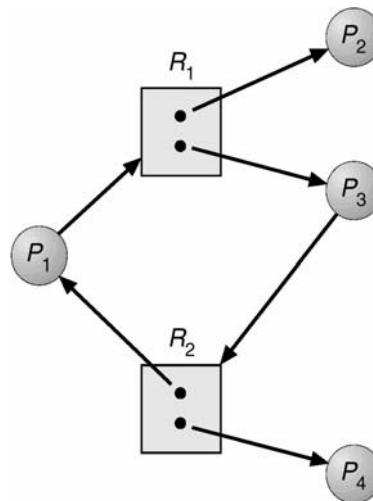


Operating System Concepts

8.9

Resource Allocation Graph With A Cycle But No Deadlock

- We have a cycle:
 - $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- There is no deadlock.
- P_4 may release its instance of R_2 and that resource can then be allocated to P_3 breaking the cycle.
- 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.



Operating System Concepts

8.10

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
 - The system can use either a deadlock prevention or avoidance.
- 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 (need to restart your computer if a deadlock occur).

Deadlock Prevention

- For a deadlock to occur, each of the four necessary conditions must hold.
 - By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.
1. Mutual Exclusion – not required for sharable resources; must hold for non-sharable resources.
 - For example, a printer cannot be simultaneously shared by several processes.
 - A process never needs to wait for a sharable resource.
 2. Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - One protocol requires each process to request and be allocated all its resources before it begins execution,
 - Or another protocol allows a process to request resources only when the process has none. So, before it can request any additional resources, it must release all the resources that it is currently allocated.

Deadlock Prevention (Cont.)

- Example to illustrate the difference between the two protocols: A process copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer.



- Protocol 1: request all tape, disk, printer and hold for entire execution; note printer will be idle for a long time.
- Protocol 2: request tape and disk only after copying release both then request disk and printer after printing release both.

Deadlock Prevention (Cont.)

- Two main disadvantages to these protocols:
 1. Low resource utilization: since many of the resources may be allocated but unused for a long time.
 2. starvation possible: A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

Deadlock Prevention (Cont.)

3. No Preemption – To ensure that this condition does not hold, we can use the following protocol:
- 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.)

4. Circular Wait – To ensure that the circular-wait condition never holds is to determine a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.
- Example: Let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types.
Assign to each resource type a unique integer number to compare two resources and to determine whether one proceeds another in ordering.
For example: If the set of resource types R includes tape drives, disk drives, and printers, then a function F might be defined as follows:
 $F(\text{tape drive}) = 1$; $F(\text{disk drive}) = 5$; $F(\text{Printer}) = 12$.

Deadlock Prevention (Cont.)

- A protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration.
 - A process can initially request any number of instances of a resource type R_i .
 1. That process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.
 - From the previous example; a process wants to use the tape drive and printer at the same time, must first request the tape drive and then the printer.
 2. When a process requests an instance of resource type R_j , it has released any resources R_i such that $F(R_i) \geq F(R_j)$.
 - By applying 1 and 2 then the circular-wait condition cannot hold.

Deadlock Avoidance

- Requires that the system has some additional *a priori* information about 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

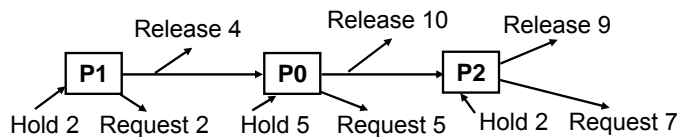
- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if for each P_i , the resources that 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.

Safe State Example

- A system with 12 tape drives and 3 processes; P0, P1, P2.
- The following Table at state t_0 .

Process	Max Need	Current hold
P0	10	5
P1	4	2
P2	9	2

9 tape drives total at t_0 been held, so a total of 3 tape drives are free available.



- At t_0 the system is in a safe state, because the sequence $\langle P1, P0, P2 \rangle$ satisfies the safety condition.

Safe State Example (Cont.)

- For unsafe state suppose at t1, P2 requests and is holding one more tape drive as shown below:

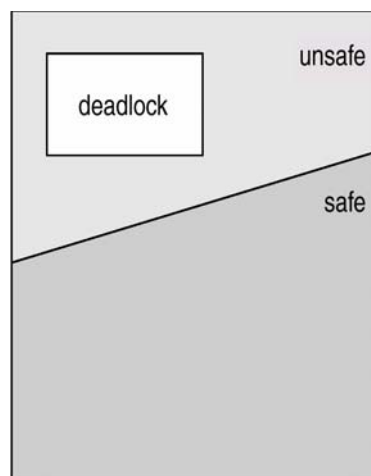
Process	Max Need	Current hold
P0	10	5
P1	4	2
P2	9	3

10 tape drives total at t1 been held, so a total of 2 tape drives are free available.

- This sequence <P1, P0, P2> is unsafe; only P1 can be satisfied. P0 and P2 must wait yielding to a deadlock.

Basic Facts

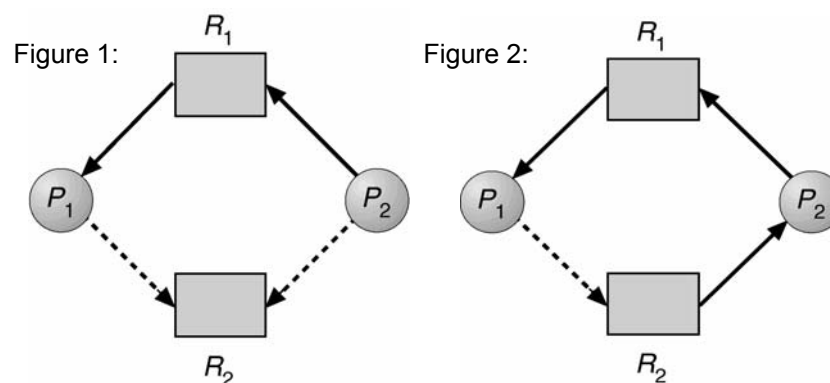
- If a system is in safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.
- Safe, unsafe, deadlock state spaces.



Resource-Allocation Graph Algorithm

- Claim edge $P_i \dashrightarrow R_j$ 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.
- If a cycle is found in a graph, then the allocation will put the system in an unsafe state.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.

Resource-Allocation Graph For Deadlock Avoidance



- Figure 2 shows an unsafe state in a resource-allocation graph, because there is a cycle in the graph.
- The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type.

Banker's Algorithm

- Banker's algorithm is applicable to a resource-allocation system with multiple instances of each type.
- From its name, it could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all customers.
- 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 resource types.
 - *Available*: Vector of length m indicates the number of available resources of each type. If $available[j] = k$, there are k instances of resource type R_j available.
 - *Max*: $n \times m$ matrix defines the maximum demand of each process. 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 defines the number of resources of each type currently allocated to each process. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of resource type R_j .
 - *Need*: $n \times m$ matrix indicates the remaining resource need of each process. 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 (Banker's Algorithm)

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

Initialize:

$Work := Available$

$Finish[i] = false$ for $i = 1, 2, \dots, n$.

2. Find an *i* such that both:

(a) $Finish[i] = false$

(b) $Need_i \leq Work$

If no such *i* (no process) 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 .

• When a request for resources is made by process P_i , the following actions are taken:

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 : (Need= Max - Allocation)
- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety criteria.

Process	Allocation	Max	Available	Need	Available after Release
	A B C	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2	7 4 3	7 5 5
P1	2 0 0	3 2 2		1 2 2	5 3 2
P2	3 0 2	9 0 2		6 0 0	10 5 7
P3	2 1 1	2 2 2		0 1 1	7 4 3
P4	0 0 2	4 3 3		4 3 1	7 4 5

Operating System Concepts

8.29

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

- Check that Request \leq Need (that is, $(1,0,2) \leq (1,2,2)$ is *true*).
- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2)$ is *true*).

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>	<u>Available After Release</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0	7 5 5
P_1	3 0 2	0 2 0		5 3 2
P_2	3 0 2	6 0 0		10 5 7 (finish)
P_3	2 1 1	0 1 1		7 4 3
P_4	0 0 2	4 3 1		7 4 5

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.

Operating System Concepts

8.30

Example (Cont.)

- Can request for (3,3,0) by P_4 be granted?
 - Check Request \leq Need (that is, $(3,3,0) \leq (4,3,1)$) is true).
 - Check Request \leq Available (that is, $(3,3,0) \leq (2,3,0)$ is not true).
 - So, the request for (3,3,0) by P_4 can not be granted.
- Can request for (0,2,0) by P_0 be granted?
 - Check Request \leq Need (that is, $(0,2,0) \leq (7,4,3)$ is true).
 - Check Request \leq Available (that is, $(0,2,0) \leq (2,3,0)$ is true).
 - However, since Available becomes 2 for A, 1 for B, and 0 for C, we can not find a sequence to satisfy the safety requirements. Therefore, the resulting state is unsafe and the request for (0,2,0) by P_0 can not be granted.

Deadlock Detection

- If a system does not employ a deadlock prevention algorithm or a deadlock avoidance algorithm, then a deadlock situation may occur.
- Therefore, the system must provide:
 - Detection: an algorithm that examines the state of the system to determine whether a deadlock has occurred.
 - Recovery: an algorithm to recover from the deadlock.
- A detection and recovery scheme requires overhead that includes:
 - Run-time costs of maintaining the necessary information.
 - Executing the detection algorithm.
 - The potential losses inherent in recovering from a deadlock.

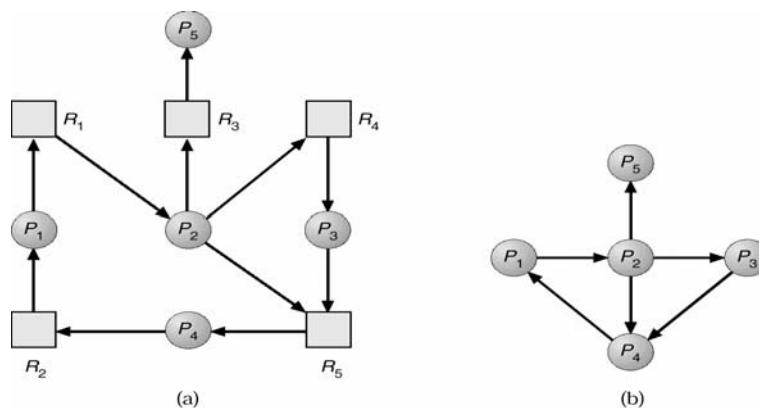
Single Instance of Each Resource Type

- A deadlock detection algorithm uses a variant of the resource-allocation graph called wait-for graph.
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \longrightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \longrightarrow R_q$ and $R_q \longrightarrow P_j$ for some resource R_q .
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to:
 - Maintain the wait-for graph.
 - Periodically invoke an algorithm that searches for a cycle 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.

Operating System Concepts

8.33

Resource-Allocation Graph And Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

Operating System Concepts

8.34

Several Instances of a Resource Type

- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:
 - Note: n is number of processes in the system and m is the number of resource types.
 - *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) $Request_i \leq Work$ (P_i not involved in deadlock)If no such i exists, go to step 4.

Detection Algorithm (Cont.)

3. $Work := Work + Allocation;$
 $Finish[i] := true$ (Assume P_i will require no more resources to complete).
 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>Ava-New</u>	<u>Allocation</u>	<u>Request</u>	<u>Available</u>	<u>Request (New)</u>
	A B C	A B C	A B C	A B C
0 1 0	P_0 0 1 0	0 0 0	0 0 0	0 0 0
7 2 4	P_1 2 0 0	2 0 2		2 0 2
3 1 3	P_2 3 0 3	0 0 <u>0</u>		0 0 <u>1</u>
5 2 4	P_3 2 1 1	1 0 0		1 0 0
7 2 6	P_4 0 0 2	0 0 2		0 0 2

- We claim that the system is not in a deadlock state.
- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all i .

Example (Cont.)

- Suppose P_2 requests an additional instance of type C .
- The Request matrix is modified as shown in the previous slide (New Request).
- 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 .

Recovery from Deadlock

- There are two options for breaking a deadlock:
 - To abort one or more processes to break the circular wait (Process Termination).
 - To preempt some resources from one or more of deadlock processes (Resource Preemption).

Recovery from Deadlock: Process Termination

- Two methods to eliminate deadlocks by aborting a process. In both methods, the system reclaims all resources allocated to the terminated processes:
 - Abort all deadlocked processes: It will break the deadlock cycle, but a great expense.
 - Abort one process at a time until the deadlock cycle is eliminated: Overhead, since, after each process aborted a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- 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?

Operating System Concepts

8.41

Recovery from Deadlock: Resource Preemption

- If preemption is required to deal with deadlocks, then three issues need to be addressed:
 - Selecting a victim – which resources and which processes are to be preempted? (minimize cost)
 - Rollback – If we preempt a resource from a process, what should be done with that process? We must roll back the process to some safe state, and restart it from that state.
 - Starvation – That is same process may always be picked as victim, include number of rollback is cost factor. Insure that a starvation will not occur. That is Guarantee that resources will not always be preempted from the same process.

Operating System Concepts

8.42

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.