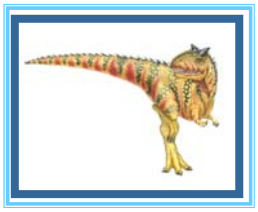
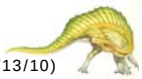
Chapter 7: Deadlocks



(slides improved by R. Doemer, 05/13/10)

Chapter 7: Deadlocks

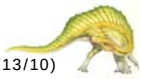
- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock



(slide modified by R. Doemer, 05/13/10)

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

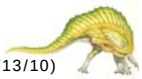


(slide modified by R. Doemer, 05/13/10)

The Deadlock Problem

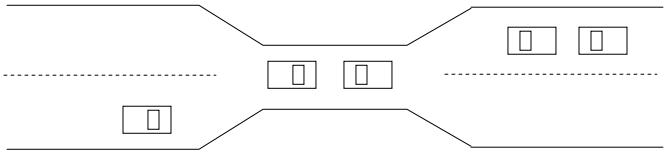
- A set of **blocked processes**, each holding a resource and waiting to acquire a resource held by another process in the set
- Application Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example with semaphores
  - Binary semaphores *A* and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

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)



(slide modified by R. Doemer, 05/13/10)

Bridge Crossing Example

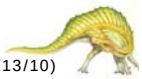


- Traffic across bridge has only one lane available
- Each section of the bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved by cars backing up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note –
  Most operating systems do not prevent or deal with deadlocks!

(slide modified by R. Doemer, 05/13/10)

Dead Locks, 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 utilizes a resource as follows:
  - **request**
  - **use**
  - **release**



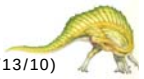
(slide modified by R. Doemer, 05/13/10)

Deadlock Characterization

Deadlock can arise *if and only 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 $\{P_0, P_1, \ldots, P_n\}$ 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.

Note that these four are *necessary conditions*!

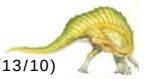


(slide modified by R. Doemer, 05/13/10)

Resource-Allocation Graph

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
- E is partitioned into two types:
  - request edge – directed edge $P_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow P_i$



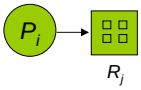
(slide modified by R. Doemer, 05/13/10)

Resource-Allocation Graph

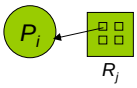
- Process

- Resource type with 4 instances

- P_i requests instance of R_j

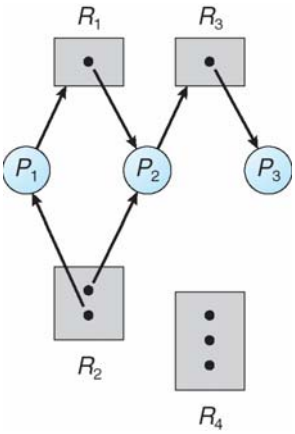


- P_i is holding an instance of R_j



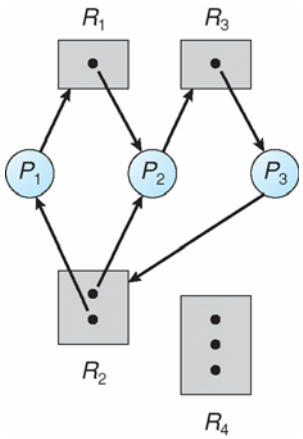
(slide modified by R. Doemer, 05/13/10)

Example of a Resource-Allocation Graph

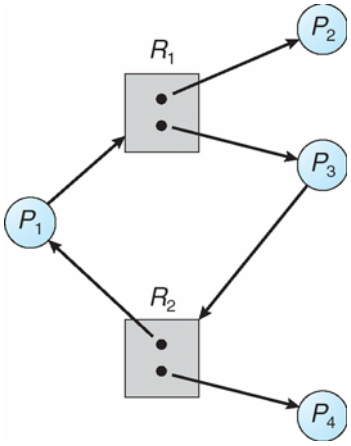


(slide modified by R. Doemer, 05/13/10)

Resource-Allocation Graph With A Deadlock

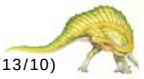


Graph With A Cycle But No Deadlock



Basic Facts

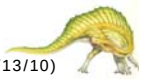
- If resource-allocation graph contains no cycles
  ⇒ no deadlock!
- If resource-allocation graph contains a cycle
  ⇒
  - if only one instance exists per resource type, then it is a deadlock
  - if several instances exist per resource type, then there is a *possibility* of a deadlock



(slide modified by R. Doemer, 05/13/10)

Methods for Dealing with Deadlocks

- Ensure that the system will *never* enter a deadlock state
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
  - Recovery from deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX and Windows



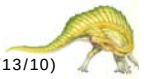
(slide modified by R. Doemer, 05/13/10)

Deadlock Prevention

To **prevent** deadlocks from occurring,
we can restrain the ways request can be made.

Prevent one of the four necessary conditions!

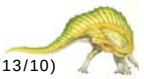
- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible



(slide modified by R. Doemer, 05/13/10)

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
  - This is the most realistic way of deadlock prevention!

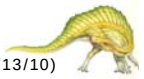


(slide modified by R. Doemer, 05/13/10)

Deadlock Avoidance

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

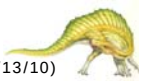
- Simplest and most useful model requires that each process declares the *maximum number* of resources of each type that it may need
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- **Resource-allocation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



(slide modified by R. Doemer, 05/13/10)

Safe State

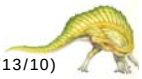
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all processes such that for each P_i, the resources that P_i can still request can be satisfied by the currently available resources *plus* the resources held by all the P_j, with $j < i$
- That is:
  - If resource needs of P_i 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



(slide modified by R. Doemer, 05/13/10)

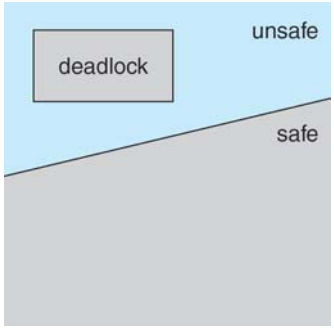
Basic Facts

- If a system is in safe state ⇒ not in deadlock state
- If a system is in unsafe state ⇒ possibility of deadlock
- **Avoidance**
  ⇒ ensure that a system will never enter an unsafe state!

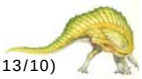


(slide modified by R. Doemer, 05/13/10)

Safe, Unsafe, and Deadlock State



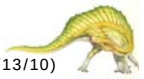
- Not all unsafe states are deadlock states!
- However, an unsafe state *may* lead to a deadlock
  (that cannot be avoided any more by the operating system).
- Only by staying in safe state, the operating system can avoid deadlocks!



(slide modified by R. Doemer, 05/13/10)

Avoidance Algorithms

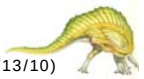
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm



(slide modified by R. Doemer, 05/13/10)

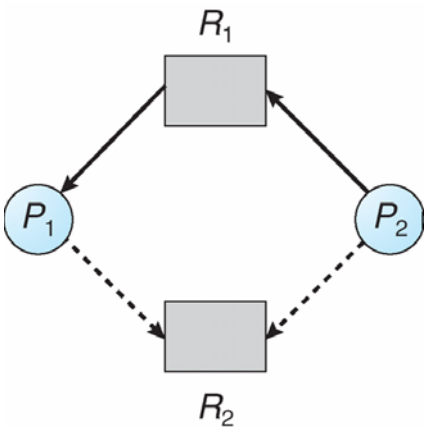
Resource-Allocation Graph Scheme

- **Claim edge** (represented by a dashed line): $P_i \rightarrow R_j$ indicates that process P_i *may* request resource R_j;
- Claim edge converts to request edge when a process requests a resource
- Request edge converts to 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



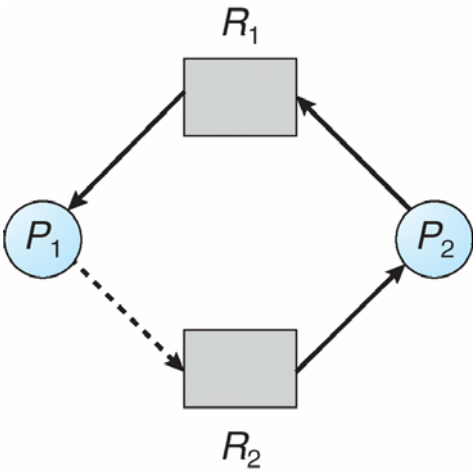
(slide modified by R. Doemer, 05/13/10)

Resource-Allocation Graph with Claim Edges



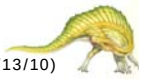
(slide modified by R. Doemer, 05/13/10)

Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

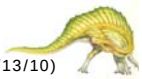
- Suppose that process P_i requests a resource R_j
- The request can be granted *only if* converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
- Notes:
  - Detecting a cycle in a resource-allocation graph with *n* processes requires an order of n^2 operations.
  - This algorithm is not applicable if multiple instances of resource types are available.



(slide modified by R. Doemer, 05/13/10)

Banker's Algorithm

- Multiple instances per resource type are supported
- 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



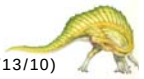
(slide modified by R. Doemer, 05/13/10)

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

- Note: *Need* = *Max* – *Allocation*



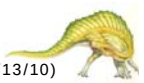
(slide modified by R. Doemer, 05/13/10)

Banker's Algorithm: Safety Algorithm

Safety Algorithm determines whether or not the system is in safe state:

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

Essentially:
Find a sequence of processes that *can finish*;
if all *can finish*, the system is in safe state!



(slide modified by R. Doemer, 05/13/10)

Banker's Algorithm: Resource-Request

Resource-Request Algorithm for Process P_i

determines whether we can grant a request by *Pi* now or it has to wait:

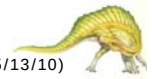
Request = request vector for process P_i.
If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2.
   Otherwise, raise error condition, since process exceeds its maximum!
2. If $Request_i \leq Available$, go to step 3.
   Otherwise P_i must wait, since resources are currently not available
3. Pretend to allocate requested resources to P_i by modifying the state:

   $Available = Available - Request$

   $Allocation_i = Allocation_i + Request_i$

   $Need_i = Need_i - Request_i$
4. Run the Safety Algorithm:
   - If *safe* $\Rightarrow$ allocate the requested resources to *Pi*
   - If *unsafe* $\Rightarrow$ *Pi* must wait; restore the previous resource-allocation state



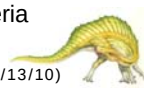
(slide modified by R. Doemer, 05/13/10)

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:

	Allocation	Max	Need	Available
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

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



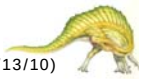
(slide modified by R. Doemer, 05/13/10)

Example of Banker's Algorithm

- Snapshot at time T_0:

	Allocation	Max	Need	Available
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- **Example:** P_1 requests (1,0,2)
- Request (1,0,2) ≤ Available (3,3,2), so Available becomes (2,3,0)
- Next, row P_1 := 3 0 2 3 2 2 0 2 0
- Finally, executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies the safety requirement
- So, request is granted (since the system stays in safe state)!



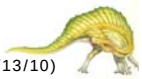
(slide modified by R. Doemer, 05/13/10)

Example of Banker's Algorithm

- Snapshot at time T_1:

	Allocation	Max	Need	Available
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- **Example Step 2:** P_4 requests (3,3,0)
- Request (3,3,0) > Available (2,3,0), so resources are not available!
- Request cannot be granted at this time
- Process P_4 needs to wait for resources to be released by other processes



(slide modified by R. Doemer, 05/13/10)

Example of Banker's Algorithm

- Snapshot at time T_2:

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- **Example Step 3:** P_0 requests (0,2,0)
- Request (0,2,0) ≤ Available (2,3,0), so Available becomes (2,1,0)
- Next, row P_0 := 0 3 0 7 5 3 7 2 3
- Finally, executing safety algorithm shows that there is *no sequence* that satisfies the safety requirement
- So, request cannot be granted (since system would be in unsafe state)!

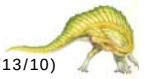
(slide modified by R. Doemer, 05/13/10)

Deadlock Detection

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

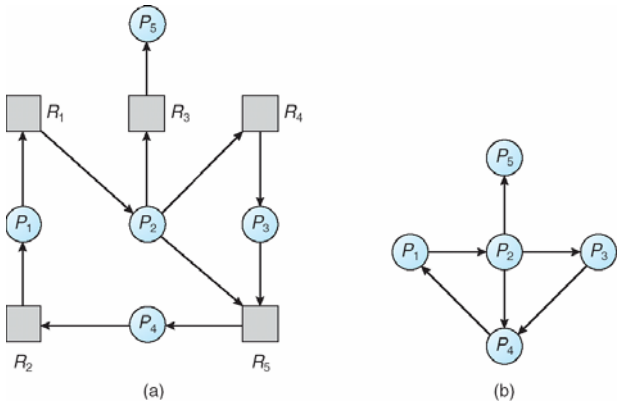
Deadlock Detection: Single Resources

- If there is only a single instance for each resource type, we can use a variation of the resource-allocation graph to detect deadlocks
- 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



(slide modified by R. Doemer, 05/13/10)

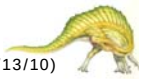
Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Deadlock Detection: Multiple Resources

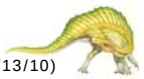
- If there are multiple instances for each resource type, we can use a variation of the Banker's algorithm to detect deadlocks
- **Available**:
  A vector of length *m* indicates the number of available resources of each type.
- **Allocation**:
  A *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request**:
  A *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.



(slide modified by R. Doemer, 05/13/10)

Deadlock Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *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*, then the system is deadlocked. All processes *i* are deadlocked, for which *Finish[i]* == *false*.



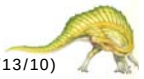
(slide modified by R. Doemer, 05/13/10)

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 $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = *true* for all *i*, so system is not in deadlocked state.



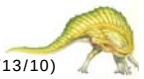
(slide modified by R. Doemer, 05/13/10)

Example (Cont.)

- P_2 requests an additional instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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



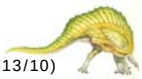
(slide modified by R. Doemer, 05/13/10)

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

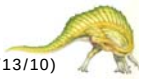
- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?



(slide modified by R. Doemer, 05/13/10)

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 rollbacks in cost factor



(slide modified by R. Doemer, 05/13/10)

End of Chapter 7