

Deadlocks



CS 502
WPI MetroWest/Southboro Campus
Spring 99

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.
 - P1 and P2 each hold one tape drive and each needs another one.
- Example:
 - Semaphores A and B, initialized to 1

P_0	P_1
<code>wait (A);</code>	<code>wait (B);</code>
<code>wait (B);</code>	<code>wait (A);</code>

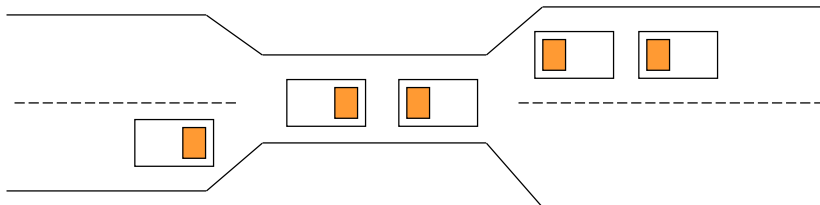
2/22/99

Deadlocks

2

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 roll back).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



2/22/99

Deadlocks

3

System Model

- Resource Types R_1, R_2, \dots, R_{m-1}
 - 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 task 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



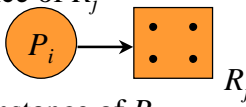
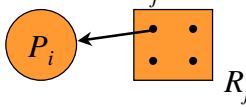
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the *processes* in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all *resource types* in the system.
- *request edge*: directed edge $P_i \rightarrow R_j$
 P_i is requesting a resource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

2/22/99

Deadlocks

6

Resource Allocation Graph (Cont.)

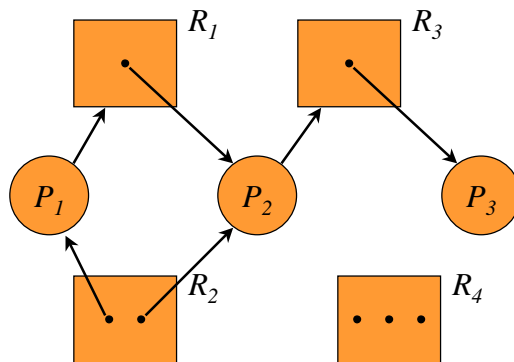
- Process

- Resource Type with 4 instances

- P_i requests instance of R_j

- P_i is holding an instance of R_j


2/22/99

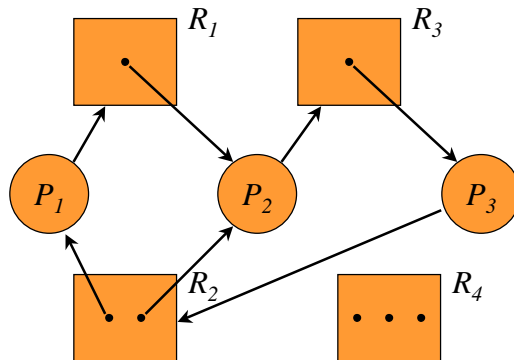
Deadlocks

7

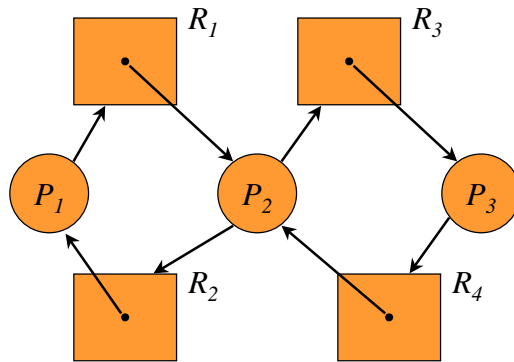
Example of a Graph with No Cycles



Example of a Graph with a Cycle



Example of a Graph With a Cycle

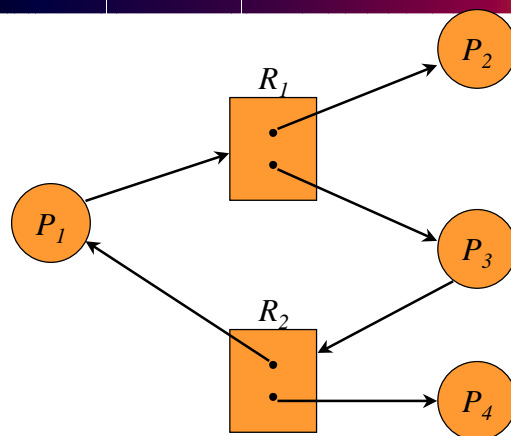


2/22/99

Deadlocks

10

Another Example of a Graph With a Cycle



2/22/99

Deadlocks

11

Fundamental Facts



- If a graph contains no cycles than no deadlock.
- If a graph contains a cycle than
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlock



- 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



Constrain the ways resource requests can be made, precluding one of the necessary conditions:

- **Mutual Exclusion:** not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait:** must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - Require a 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 order of all resource types and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

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

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it *may* need (over entire life of process).
- 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 a safe state if there exists a safe sequence for 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 resource needs are not immediately available, then can wait until all have finished.
 - When is finished, can obtain needed resources, execute, return allocated resources, and terminate.
 - When terminates, can obtain its needed resources, and so on.

Fundamental Facts



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

Resource Allocation Graph Algorithm



- Claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j ; represented by a dashed line.
- Claim edge converts to a 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.

Banker's Algorithm

- Handles multiple instances of a resource.
- 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 . 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 additional 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:
 $Work := Available$
 $Finish[i] := \text{false}$ for $i = 1, 2, \dots, n$
2. Find an i such that both:
(a) $Finish[i] = \text{false}$
(b) $Need_i \leq Work$
If no such i exists, go to step 4.
3. $Work := Work + Allocation_i$
 $Finish[i] := \text{true}$
Go to step 2.
4. If $Finish[i] = \text{true}$ 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. Simulate the allocation of the requested resource 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>Available</u>				<u>Max</u>				<u>Allocation</u>		
A	B	C		A	B	C		A	B	C
3	3	2		P_0	7	5	3	0	1	0
				P_1	3	2	2	2	0	0
				P_2	9	0	2	3	0	2
				P_3	2	2	2	2	1	1
				P_4	4	3	3	0	0	2

Example (Cont.)

- The content of the matrix *Need* is defined as *Max - Allocation*

<u>Available</u>				<u>Max</u>				<u>Allocation</u>				<u>Need</u>		
A	B	C		A	B	C		A	B	C		A	B	C
3	3	2		P_0	7	5	3	0	1	0				
				P_1	3	2	2	2	0	0				
				P_2	9	0	2	3	0	2				
				P_3	2	2	2	2	1	1				
				P_4	4	3	3	0	0	2				

- Is the system in a safe state?

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

- Check that $Request_i \leq Need_i$
- Check that $Request_i \leq Available$
- Simulate grant of request:

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

- Is the resulting system in a safe state?

Example (Cont.): P_4 requests (3, 3, 0)

- Check that $Request_i \leq Need_i$
- Check that $Request_i \leq Available$
- Simulate grant of request:

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

- Is the resulting system in a safe state?

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.
- 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.

Several Instances of a Resource Type

Let n = number of processes and m = number of resource types.

- *Available*: Vector of length m . If $Available[j] = k$, there are k instances of resource type R_j available.
- *Allocation*: $n \times m$ matrix. If $Allocation[i, j] = k$, then P_i is currently allocated k instances of R_j .
- *Request*: $n \times m$ matrix. If $Request[i, j] = k$, then P_i is requesting k instances of resource type R_j .

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively.
Initialize:
 $Work := Available$
For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then $Finish[i] := \text{false}$, else $Finish[i] := \text{true}$
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.
3. $Work := Work + Allocation_i$
 $Finish[i] := \text{true}$
Go to step 2.
4. If $Finish[i] = \text{false}$ for some i , $1 \leq i \leq n$, then the system is in a deadlocked state. Moreover, if $Finish[i] = \text{false}$, then P_i is deadlocked.

Algorithm requires $O(m \cdot n^2)$ operations to detect deadlock.

Example of Detection Algorithm

- 5 processes P_0 through P_4 ; 3 resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0 :

<u>Available</u>			<u>Allocation</u>			<u>Request</u>				
A	B	C		A	B	C	A	B	C	
0	0	0		P_0	0	1	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

Example (Cont.)

- P_2 requests an additional instance of type C:

<u>Available</u>			<u>Allocation</u>			<u>Request</u>				
A	B	C		A	B	C	A	B	C	
0	0	0		P_0	0	1	0	0	0	0
				P_1	2	0	0	2	0	2
				P_2	3	0	3	0	0	1
				P_3	2	1	1	1	0	0
				P_4	0	0	2	0	0	2

- State of the system?

Detection Algorithm Usage



- When, and how often, to invoke depends on:
 - How often is deadlike likely to occur?
 - How many processes will need to be rolled back?
- If deadlock 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 be choose to abort?
 - Priority of the process.
 - Computation metric — current length or length to completion
 - Resources the process has used.
 - Resources the process is requesting.
 - Number of process requiring termination.
 - Interactive versus 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 rollbacks in cost factor.

Combined Approach to Deadlock Handling



- Combine the three basic approaches
 - Prevention
 - Avoidance
 - Detectionallowing the use of the optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.