Chapter 7: Deadlocks

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Objectives/Outline

Objectives

- Develop conceptual understanding of deadlocks
- Present a number of different methods for preventing and avoiding deadlocks

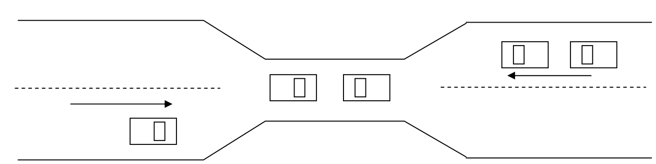
Outline

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

Introduction

- Deadlock is defined as the permanent blocking of a set of processes that are competing for a finite number of system resources
  - occurs when a set of processes are in a wait state and each process is waiting for a resource that is held by some other waiting process
  - all deadlocks involve conflicting resource needs by two or more processes
- Unlike other problems in multiprogramming systems, there is no efficient solution to the deadlock problem in the general case

Deadlock Characterization: Conditions for Deadlock



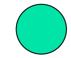
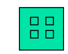
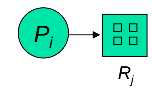
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

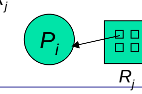
Deadlock Characterization: Conditions for Deadlock (cont.)

- The four necessary conditions for a deadlock:
  - Mutual Exclusion: processes require exclusive control of their resources (no sharing)
  - Hold and Wait: process may wait for a resource while holding others
  - No Preemption: resources cannot be preempted; a process will only voluntarily give up a resource after completing its task with this resource.
  - Circular wait: there exists a set $\{P_0, P_1, ..., P_n\}$ 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
    - Example: semaphores *A* and *B*, initialized to 1

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

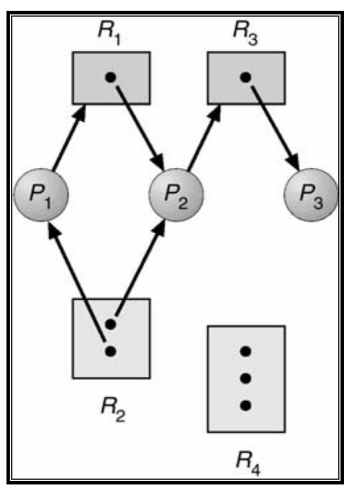
Deadlock Characterization: Resource-Allocation Graph

- A set of vertices *V* and a set of edges *E*
- V is partitioned into two types: $P = \{P_1, P_2, ..., P_n\}$, the set of all processes in the system and $R = \{R_1, R_2, ..., R_m\}$, the set of all resource types in the system
- A request edge is a directed edge $P_1 \rightarrow R_j$ and an assignment edge is a directed edge $R_j \rightarrow P_i$
- Process is represented by 
- Resource type with 4 instances is represented by 
- P_i requests instance of R_j represented by 

- P_i is holding an instance of R_j. This is represented by 

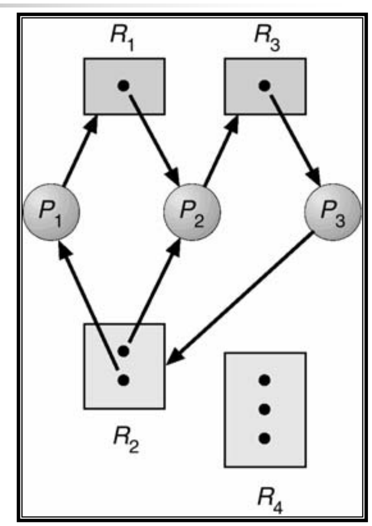
Example of a Resource Allocation Graph

- Processes: P ={P1, P2, P3}
- Resource types: R={R1, R2, R3, R4}
- Edges: 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: R_1 (one), R_2 (two), R_3 (one), R_4 (three)

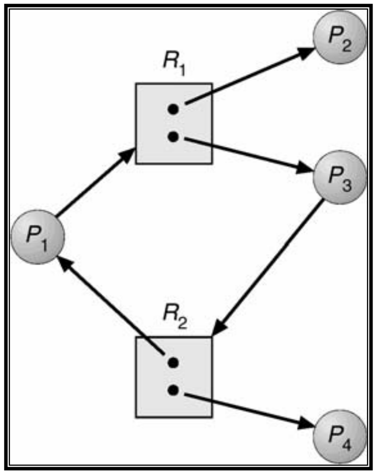


Resource Allocation Graph with a Deadlock

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



Resource Allocation Graph with a Cycle But No Deadlock



Methods for Handling Deadlocks

- How can we handle a deadlock situation?
  - Ensure that the system will *never* enter a deadlock state
    - In this case, the system can use either deadlock prevention or deadlock avoidance techniques
  - Allow the system to enter a deadlock state and then recover
    - In this case, the system employs deadlock detection and deadlock recovery techniques
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
    - This results in deterioration of system performance and results in restarting the system manually

Deadlock Prevention

- By ensuring that at least one of the four necessary deadlock conditions cannot hold, we prevent the occurrence of a deadlock
- Mutual Exclusion
  - Not required for sharable resources such as read-only files
  - Must hold for non-sharable resources such as a printer
- Hold and Wait: must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require the process to request and be allocated all its resources before it begins execution, or allow the process to request resources only when the process has no other resources
  - Low resource utilization and starvation is 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
  - other solutions?? If requested resources are held by waiting processes, preempt them from the waiting processes and allocate them to the requesting process; otherwise wait
- Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Activity

- Prove that the circular-wait condition can not hold under each of the following conditions
  - A process holding Ri can request Rj iff F(Rj)>F(Ri)
  - If a process request Rj then it has released all resources Ri for which F(Ri) >= F(Rj)

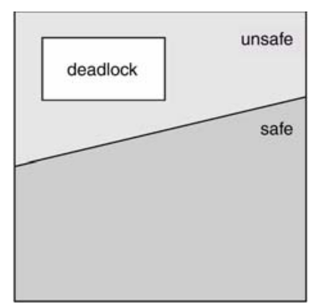
Deadlock Avoidance

- 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
  - We want to insure that the resource-allocation state is safe

Deadlock Avoidance: 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 of all processes
- Sequence $<P_1, P_2, \ldots, P_n>$ 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

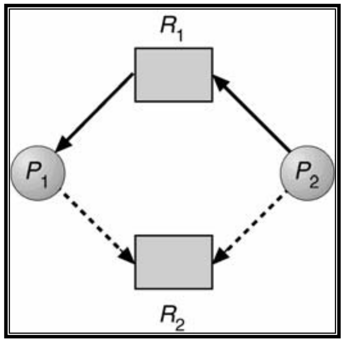
Deadlock Avoidance: Safe State



- 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

Resource-Allocation Graph Algorithm

- Applicable to a system with ONE instance of each resource
- Claim edge $P_i \rightarrow R_j$ indicated that process P_j 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



Deadlock Avoidance: Banker's Algorithm

- Applicable to a system with multiple instances of each resource
- Analogy to a banking system
  - Could be used in 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 claim maximum resources usage in advance
- When a process requests a resource it may have to wait

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

Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]

Data Structures for the Banker's Algorithm: Example

- Assume that there are 5 processes P0 through P4; 3 resource types A (10 instances), B (5 instances), and C (7 instances)
- n = ?, m = ?
- available [A] = *?*
- available [B] = *?*
- available [C] = *?*
- Snapshot at time:

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

Banker's Algorithm

- *Check whether a request from process i can be satisfied*
  - *if the request from process i cannot be satisfied*
    - *error or deny the request*
  - *else*
    - *Pretend to allocate*
    - *check safety*
    - *if current system is safe **then***
      grant the allocation to the request
    - *else deny the request*
      restore original state if necessary

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 an *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

Matrix Need is defined as Max – Allocation

	Available	Need
	A B C	A B C
	3 3 2	
P0		7 4 3
P1		1 2 2
P2		6 0 0
P3		0 1 1
P4		4 3 1

Sequence <P1, P3, P4, P2, P0> satisfies safety criteria

Example of Safety Algorithm

- Assume that there are 5 processes P0 through P4; 3 resource types A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T0:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

The content of the matrix Need is defined to be Max – Allocation

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

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*

Resource-Request Algorithm for Process P_i

- Suppose that P1 requests (1,0,2)
- Check that Request ≤ Available ; that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	2 3 0
P1	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement
- Next, can request for (3,3,0) by P4 be granted?
- Lastly, can request for (0,2,0) by P0 be granted? Question for you!

Banker's algorithm depends on future information (i.e., information a head of time on the maximum resources that processes will need)

In practice, Banker's algorithm is rarely implemented, since processes don't know a head of time the maximum resources they will need

Summary: Banker's algorithm

- **if** *Request*[*i,j*] > *Need*[*i,j*], for all *j*, **then**
  - error;
- **if** *Request*[*i,j*] > *Available*[*j*], for all *j*, **then**
  - deny the request;
- - pretend to allocate
- **for all** *i,j* :
  - *Available*[*j*] := *Available*[*j*] - *Request*[*i,j*];
  - *Allocated*[*i,j*] := *Allocated*[*i,j*] + *Request*[*i,j*];
  - *Need*[*i,j*] := *Need*[*i,j*] - *Request*[*i,j*];
- - check safety
- **if** current system is safe **then**
  - grant the allocation to the request;
- **else**
  - deny the request
  - - restore original state if necessary
  - **for all** *i,j* :
    - *Available*[*j*] := *Available*[*j*] + *Request*[*i,j*];
    - *Allocated*[*i,j*] := *Allocated*[*i,j*] - *Request*[*i,j*];
    - *Need*[*i,j*] := *Need*[*i,j*] + *Request*[*i,j*];

Deadlock Detection

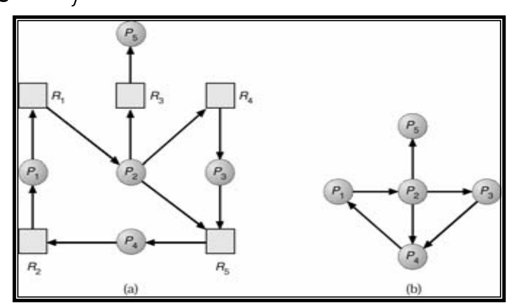
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

- Two different solutions exist:
  - For systems with single instance of each resource type
    - We define a deadlock-detection algorithm called wait-for graph
  - For systems with multiple instances of each resource type
    - We define a deadlock-detection algorithm that is a similar to the banker's algorithm

Detection algorithm for Single Instance of Each Resource Type

- Maintain a *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 wait-for graph



(a) Resource-Allocation Graph (b) Corresponding wait-for graph

Several Instances of a Resource Type

- Uses a variant of banker's algorithm
- Data structures
  - *Available*: A vector of length *m* indicates the number of available resources of each type
  - *Allocation*: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process
    - $Allocation_i$ the number of resources of each type currently allocated to process P_i (a vector of length *m*)
  - *Request*: An *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
    - $Request_i$ the current request of process P_i of each resource type (a vector of length *m*)
  - *Work* and *Finish* be vectors of length *m* and *n*, respectively

Detection Algorithm for Several Instances of a Resource Type

1. Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 0,1,2, ..., *n-1*, 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*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Complexity: requires $m.n^2$ operations

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*
  - Exercise: verify that.

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
$\mathbf{P_2}$	**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 requests of other processes
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Deadlock Recovery

- Report deadlock and let the operator deal with it manually
- Recover automatically from the deadlock
  - Process termination – abort one or more processes and reclaim all resources allocated to the terminated processes to break the circular wait
    - Aborting a process may or may not be easy, e.g. terminating a process in the midst of updating a file may have the file in incorrect state
    - Partial computations will be wasted
  - Resource preemption – preempt some resources from one or more deadlocked processes until deadlock is cycle is broken

Recovery from Deadlock: Process Termination

- There are two approaches
  - Abort all deadlocked processes
    - Great expense in terms of wasted partial computations
  - Abort one process at a time until the deadlock cycle is eliminated
    - Incurs considerable overhead; after each process is aborted, a deadlock detection must be invoked
- Which processes to terminate and the order of termination is a policy decision that should minimize the incurred costs
- Factors that affect the decision
  - 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

- Need to deal with three issues:
  - Selecting a victim
    - which resources to be preempted and from which process
    - minimize cost as in process termination
  - Rollback
    - return to some safe state (checkpoint), restart process from that state
    - roll back as far as necessary to break the deadlock
    - total rollback – abort the process and then restart it
  - Starvation –
    - how to ensure that the same process will not be always picked as a victim?
    - include the number of rollbacks in the cost factor

End of Chapter 7

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005