Chapter 7: Deadlocks





Chapter 7: Deadlocks

- 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

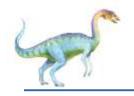




System Model

- System consists of resources
- 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





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 $\{P_0, P_1, ..., P_n\}$ 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 .





Deadlock Characterization

Possible strategies

- Mutual exclusion: allow shared resources to be used with ME.
- Hold and wait: Let a process start only after it receives all resources it requires (starvation) OR acquire only when you need it.
- No preemption: A process should wait only if the resources it needs is not available and they are not in possession of another waiting process. If being used by another waiting process then grab them and see if you can run. You wait only if the resource is with a running process. While waiting some of your resources might get grabbed
- Circular wait: Total ordering of resources so wont be cycle. Each process attains resources in order as in the total ordering.



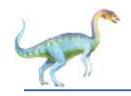


Resource-Allocation Graph

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 consisting of all the processes in the system
 - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system
- request edge directed edge $P_i \rightarrow R_i$
- **assignment edge** directed edge $R_i \rightarrow P_i$





Resource-Allocation Graph (Cont.)

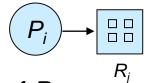
Process



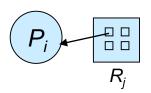
Resource Type with 4 instances



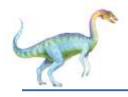
 \blacksquare P_i requests instance of R_i



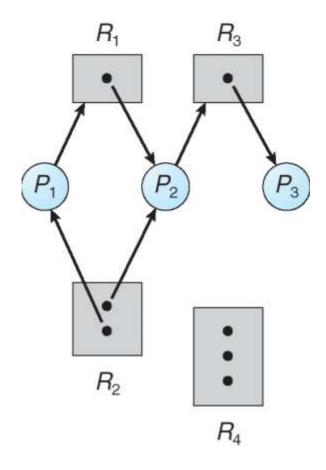
 \blacksquare P_i is holding an instance of R_j







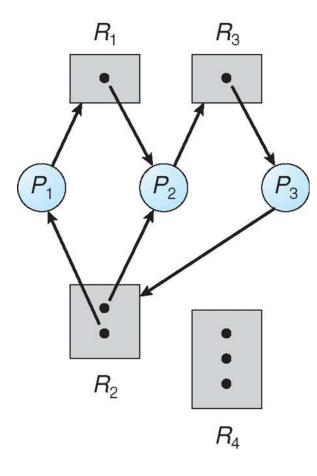
Example of a Resource Allocation Graph







Resource Allocation Graph With A Deadlock

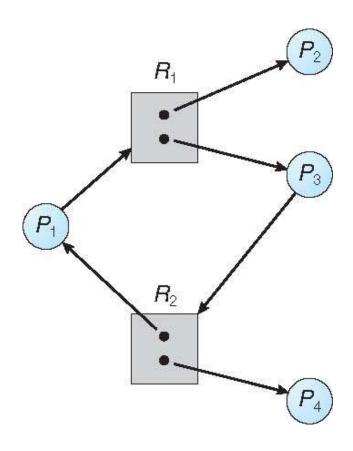


Is there a deadlock? How can we detect a deadlock? Is it a iff condition?

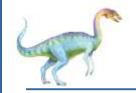




Graph With A Cycle But No Deadlock







Basic Facts

- 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





Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidence
- 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 (e.g., read-only files); 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 (waiting for nothing), or allow process to request resources only when the process has none allocated to it(hold nothing).
 - 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
- Alternatively, if you need something that is not available then, it is held either by another waiting process or a running process. If held by waiting process then make the waiting process reliquish so that you can proceed. If held by running process then you wait. A process restarts after it gets relinguished and desired resources.
- Circular Wait impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



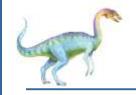


Deadlock Avoidance

Requires that the system has some additional *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





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 $\langle P_1, P_2, ..., P_n \rangle$ 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_i , with j < i.
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_i for j<i 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

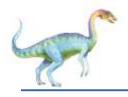




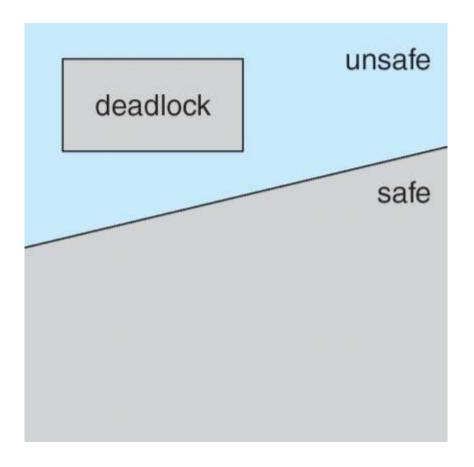
Basic Facts

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

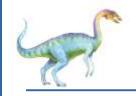




Safe, Unsafe, Deadlock State







Example

Total resources of type printer=12

	max	current
P0	10	5
P1	4	2
P2	9	2

Safe sequence P1,P0,P2 Instead if P2 has current as 3 then unsafe 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





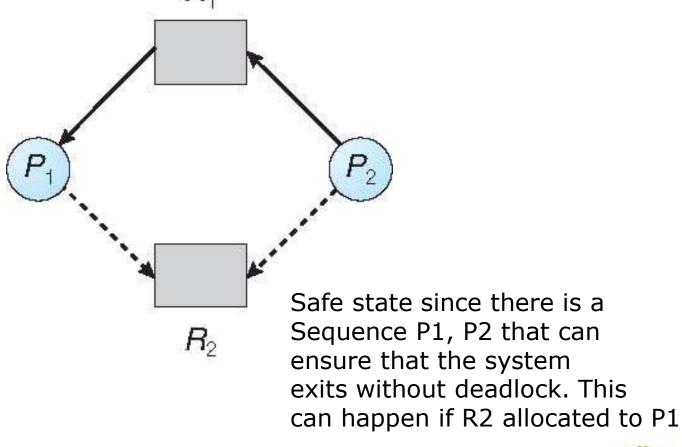
Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_i ; 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

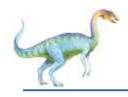




Resource-Allocation Graph

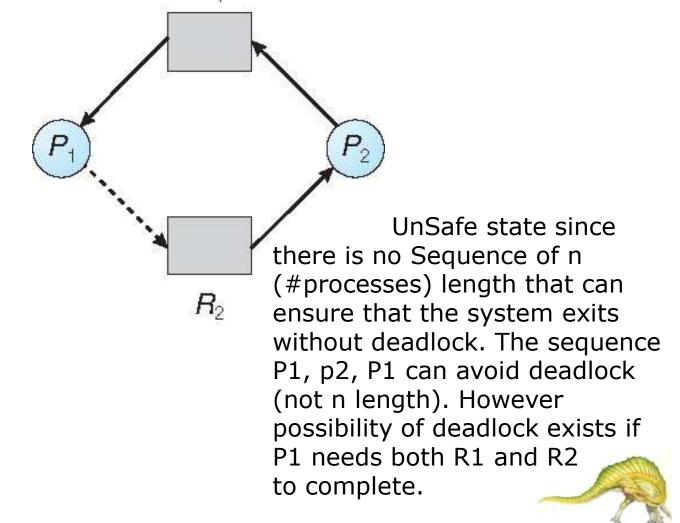


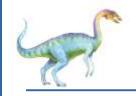




Unsafe State In Resource-Allocation Graph

 R_{1}





Safe state vs Unsafe State

Bankers Algorithm assumes the strict case that a process needs all its mentioned resources simultaneously. Hence, on making available to each process, all its requested resources at some point in time, the algorithm checks if there is a sequence of the processes that can permit all processes to complete without a deadlock.

If such a sequence exists that then state is a safe state. However, if such a sequence does not exists it is not necessary that there be a deadlock since the processes might not in reality be requesting all the resources at one go but might be relinquishing resources in between.





Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_i
- 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 of resources
- 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
- Bankers algorithm is a resource allocation and deadlock avoidance algorithm. Before allocating a particular resource it checks whether resource can be allocated without deadlock condition arising.
 - -Determine if current state is safe. Else report unsafe
 - If current state safe and a new request comes, then check if state is safe on granting request. Else do not grant request





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_i 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 more instances of R_i 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 $i = 0, 1, ..., n-1$

- 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; Finish[i] = true go to step 2
- 4. If **Finish** [i] == true for all i, then the system is in a safe state Else unsafe 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_i

- If Request_i ≤ Need_i go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
- 2. If $Request_i \le Available$, 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 ⇒ 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 (5instances), and C (7 instances)

Snapshot at time T_0 :

	Allocation	<u>Max</u>	<u>Available</u>
	ABC	ABC	ABC
P_0	010	753	332
P_1	200	3 2 2	
P_2	302	902	
P_3	2 1 1	222	
P_4	002	433	





Example (Cont.)

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

	<u>Need</u>	
	ABC	
P_0	7 4 3	
P_1	122	
P_2	600	
P_3	0 1 1	
P_4	4 3 1	

■ The system is in a safe state since the sequence P_1 , P_3 , P_4 , P_2 , P_0 satisfies safety criteria





New Request Made

- When a process makes a new request then first check if we have resource available to meet the request
- Second check whether the system will be in a safe state if the request is granted.





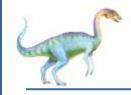
Example: P_1 Request (1,0,2)

Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true P1 now becomes (2,0,0) old + (1,0,2) new= (3,0,2) Available becomes (3,3,2)- (1,0,2)= (2,3,0)

	<u>Allocation</u>	<u>n Need</u>	<u>Available</u>
	ABC	ABC	ABC
P_0	010	7 4 3	230
P_1	302	020	
P_2	302	600	
P_3	211	011	
P_4	002	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P₄ be granted next?
- Can request for (0,2,0) by P₀ be granted after P1s request?





Example:

- Check P_4 Request (3,3,0): Request \leq Available (that is, (3,3,0) \leq (2,3,0) \Rightarrow false
- Check P0 request (0,2,0). Request ≤ Available P0 now becomes (0, 3, 0). Available is (2 0 0)

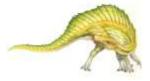
	<u>Allocation</u>	<u>n Need</u>	<u>Available</u>
	ABC	ABC	ABC
P_0	030	723	200
P_1	302	020	
P_2	302	600	
P_3	211	0 1 1	
P_4	002	4 3 1	

No Deadlocked!





- Safe state: There exists at least one permutation that will let all process' complete
- Unsafe state: If all process request for its max resources without returning any it holds in between, then, all process' can not complete.

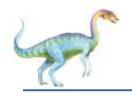




Deadlock Detection

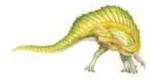
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

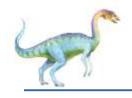




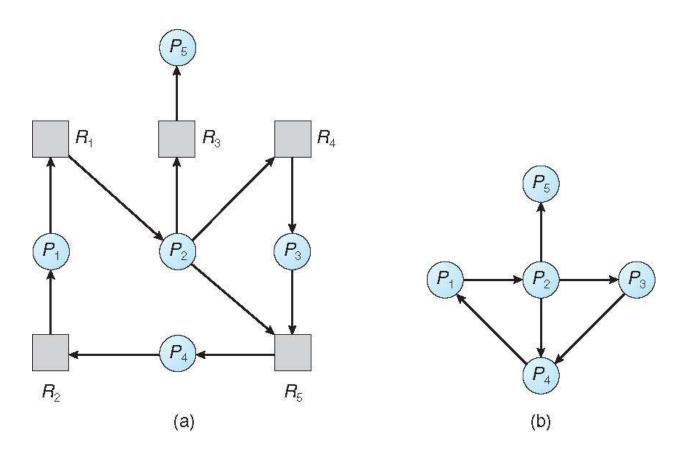
Single Instance of Each Resource Type

- 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 n² operations, where n is the number of vertices in the graph





Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

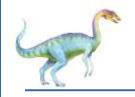




Several Instances of a Resource Type

- Available: A vector of length m indicates the number of available resources of each type
- **Allocation**: An **n** x **m** matrix defines the number of resources of each type currently allocated to each process
- 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_i.





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 ≠ 0, then Finish[i] = false; otherwise, Finish[i] = true
- 2. Find an index *i* such that both:
 - (a) Finish[i] == false
 - (b) **Request**_i ≤ **Work**

If no such *i* exists, go to step 4





Detection Algorithm (Cont.)

- 3. Work = Work + Allocation; Finish[i] = true go to step 2
- 4. If Finish[i] == false, for some i, $1 \le i \le n$, 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

Will the sequence in which the request is made make a difference to deadlock detection?





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	BC	ABC	ABC
P_0 0	1 0	000	000	
P ₁ 2 0	00 2	202		
P ₂ 30	3	000		
P ₃ 2	1 1	100		
P_4 0	0 2	002		

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in **Finish[i] = true** for all **i**



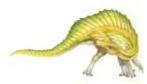


Example (Cont.)

P₂ requests an additional instance of type C

	<u>Reques</u>		
	ABC		
P_0	000		
P_1	202		
P_2	0 0 1		
P_3	100		
P_4	002		

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes; requests
 - Deadlock exists, consisting of processes P₁, P₂, P₃, and P₄





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.



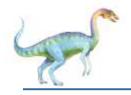


Deadlock Recovery

Deadlock Recovery can be done in one of the two ways:

- Process Termination: Terminate processes to break the circular wait
- Resource Preemption: preempt one or more resources from the deadlocked processes.





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?
 - 1. Priority of the process
 - How long process has computed, and how much longer to completion
 - Resources the process has used
 - 4. Resources process needs to complete
 - 5. How many processes will need to be terminated
 - 6. 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

