

## Chapter 7: Deadlocks



### Contents

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

AE4B330SS

Lecture 7 / Page 2

Silberschatz, Galvin and Gagne ©2005

### 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.
  - $P_1$  and  $P_2$  each hold one tape drive and if each requests another drive, the 2 processes will be in a deadlock state
- Example
  - Consider a System with one printer and one DVD drive
  - Suppose that process  $P_1$  is holding DVD and process  $P_2$  is holding the printer
  - If  $P_1$  requests the printer and  $P_2$  requests the DVD drive, a deadlock occurs
- Example
  - Semaphores  $A$  and  $B$ , initialized to 1 (mutexes)

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

AE4B330SS

Lecture 7 / Page 3

Silberschatz, Galvin and Gagne ©2005

### System Model

- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, files & I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
  - If a system has 2 CPUs, then resource type CPU has two instances
- Each process utilizes a resource as follows:
  - request
  - use
  - release

AE4B330SS

Lecture 7 / Page 4

Silberschatz, Galvin and Gagne ©2005

## Deadlock Characterization

**Necessary Conditions:** Deadlock can occur if all 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:** resources cannot be preempted; that is, 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  $\{P_0, P_1, \dots, P_n\}$  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  $P_0$ .

**Note:** The above conditions are **NECESSARY** but not sufficient for deadlock

AE4B33OSS

Lecture 7 / Page 5

Silberschatz, Galvin and Gagne ©2005

## Resource Allocation Graph





- A set of vertices  $V$  and a set of edges  $E$
- $V$  is partitioned into two types (bipartite graph):
  - $P = \{P_1, P_2, \dots, P_n\}$ , 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.
- $P_i \rightarrow R_j$ 
  - process  $i$  has requested an instance of resource  $j$  called a *request edge*
- $R_j \rightarrow P_i$ 
  - an instance of resource  $j$  has been assigned to process  $i$  called an *assignment edge*

AE4B33OSS

Lecture 7 / Page 6

Silberschatz, Galvin and Gagne ©2005

## Resource-Allocation Graph

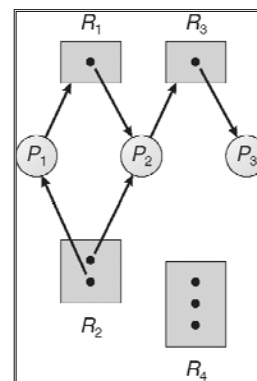
- Process 
- Resource Type with 4 instances 
- $P_i$  requests an instance of  $R_j$  
- $P_i$  is holding an instance of  $R_j$  

AE4B33OSS

Lecture 7 / Page 7

Silberschatz, Galvin and Gagne ©2005

## Example of a Resource Allocation Graph

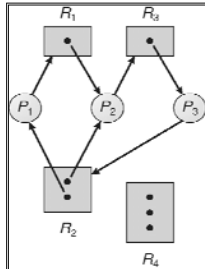


AE4B33OSS

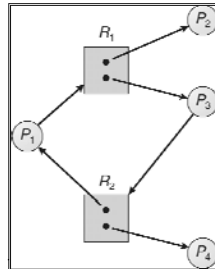
Lecture 7 / Page 8

Silberschatz, Galvin and Gagne ©2005

### Resource Allocation Graph With A Cycle



Deadlock



No Deadlock

#### Conclusions:

- If graph contains no cycles  $\Rightarrow$  no deadlock.
- If graph contains a cycle  $\Rightarrow$  then the system may or may not be in a deadlock state
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

AE4B330SS

Lecture 7 / Page 9

Silberschatz, Galvin and Gagne ©2005

### Methods for Handling Deadlocks

- **Ostrich approach:** Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
- **Deadlock Prevention:** Ensure that at least one of the necessary conditions cannot hold.
- **Deadlock Avoidance:** Control allocation of resources so that the system will *never* enter a deadlock state.
- **Deadlock Detection & Recovery:** Allow the system to enter a deadlock state, detect it, and then recover.

AE4B330SS

Lecture 7 / Page 10

Silberschatz, Galvin and Gagne ©2005

### Deadlock Prevention

- Prevent at least one of the necessary conditions from happening
  - Mutual exclusion
  - Hold and wait
  - No preemption (i.e. *have* preemption)
  - Circular wait

AE4B330SS

Lecture 7 / Page 11

Silberschatz, Galvin and Gagne ©2005

### Eliminate Mutual Exclusion

- Shared resources do not require mutual exclusion
  - e.g. read-only files
- If all resources were sharable there would not be any deadlock
- But some resources cannot be shared (at the same time)
  - e.g. printers
- The mutual-exclusion condition must hold for non-sharable resources
- We can not prevent deadlocks by denying the mutual exclusion condition, because some resources are intrinsically non-sharable

AE4B330SS

Lecture 7 / Page 12

Silberschatz, Galvin and Gagne ©2005

### Eliminate Hold & Wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - One approach requires that each process be allocated *all* of its resources before it begins executing (eliminates the wait possibility)
  - Alternatively, only allow a process to request resources when it currently has none (eliminates the hold possibility)
- Two main disadvantages
  - Low resource utilization;
  - starvation is possible.

AE4B330SS

Lecture 7 / Page 13

Silberschatz, Galvin and Gagne ©2005

### Eliminate “No Preemption”

- Make a process automatically release its current resources if it cannot obtain new resource it wants
  - restart the process when it can obtain everything (old as well as new resources)
- Alternatively, the desired resources can be preempted from other waiting processes

AE4B330SS

Lecture 7 / Page 14

Silberschatz, Galvin and Gagne ©2005

### Eliminate Circular Wait

- Impose a total ordering on all the resource types, and force each process to request resources in increasing order.
  - For example:  $R = \{R_1, R_2, \dots, R_N\}$  be a set of resource types, we assign each resource type to unique integer. We define one-to-one function  $F: R \rightarrow N$ , where  $N$  is set of natural numbers
  - Suppose  $R = \{ \text{tape drives, disk drives, printers} \}$ , then  $F$  might be defined as
    - $F(\text{tape drive}) = 1$
    - $F(\text{disk drive}) = 5$
    - $F(\text{printer}) = 12$
- *Another approach:* require a process to release larger numbered resources when it obtains a smaller numbered resource.

AE4B330SS

Lecture 7 / Page 15

Silberschatz, Galvin and Gagne ©2005

### Deadlock Prevention (Drawbacks)

- EACH of these prevention techniques may cause a decrease in utilization of resources.
- For this reason, prevention isn't necessarily the best technique.
- Prevention is generally the easiest to implement.

AE4B330SS

Lecture 7 / Page 16

Silberschatz, Galvin and Gagne ©2005

## Deadlock Avoidance

- In deadlock avoidance, the necessary conditions are untouched.
- Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
- 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.
- Tough
  - Hard to determine all resources needed in advance
  - Good theoretical problem, not as practical to use

AE4B330SS

Lecture 7 / Page 17

Silberschatz, Galvin and Gagne ©2005

## 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 safe sequence of all processes.
- **Sequence** of processes  $\langle P_1, P_2, \dots, P_n \rangle$  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.

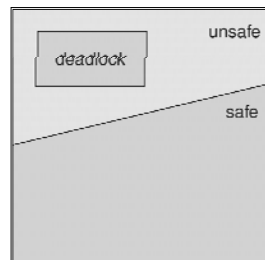
AE4B330SS

Lecture 7 / Page 18

Silberschatz, Galvin and Gagne ©2005

## 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.



AE4B330SS

Lecture 7 / Page 19

Silberschatz, Galvin and Gagne ©2005

## Example 1

- Max no. of resources: 12 tape drives
- |       | Max needs | Current Allocation (at time $t_0$ ) |
|-------|-----------|-------------------------------------|
| $P_0$ | 10        | 5                                   |
| $P_1$ | 4         | 2                                   |
| $P_2$ | 9         | 2                                   |
- Currently, there are 3 free tape drives
- The OS is in a *safe* state, since sequence  $\langle P_1, P_0, P_2 \rangle$  satisfies the safety condition.
- A system can go from a safe state to an unsafe state

AE4B330SS

Lecture 7 / Page 20

Silberschatz, Galvin and Gagne ©2005

### Example 2

- Same as last slide, now suppose at time  $t_1$ , process  $P_2$  requests and is allocated one more tape drive.
- Now process  $P_2$  3 tape drives allocated currently.

	Max needs	Current Allocation
$P_0$	10	5
$P_1$	4	2
$P_2$	9	<u>3</u>

- Currently, there are 2 free tape drives
- The OS is in an *unsafe* state, since sequence  $\langle P_1, P_0, P_2 \rangle$  does not satisfies the safety condition.

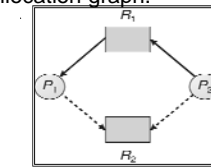
AE4B330SS

Lecture 7 / Page 21

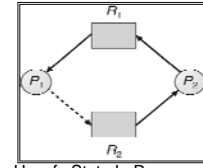
Silberschatz, Galvin and Gagne ©2005

### 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 changes to a request edge when the process actually requests the resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- All resources must be claimed before system start-up.
- An unsafe state is caused by a cycle in the resource allocation graph.



Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph

AE4B330SS

Lecture 7 / Page 22

Silberschatz, Galvin and Gagne ©2005

### Banker's Algorithm

- Used when there exists **multiple** instances of a resource type
- Each process must **declare in advance the maximum** number of instances of each resource type that it may need
- 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

AE4B330SS

Lecture 7 / Page 23

Silberschatz, Galvin and Gagne ©2005

### 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:**  $n \times m$  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  $R_j$  to complete its task.

$$Need[i, j] = Max[i, j] - Allocation[i, j].$$

AE4B330SS

Lecture 7 / Page 24

Silberschatz, Galvin and Gagne ©2005

### Safety Algorithm

Allocation<sub>i</sub> means resources allocated to process P<sub>i</sub>  
 Need<sub>i</sub> means resources needed for the process P<sub>i</sub>

- Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:  
 $Work = Available$   
 $Finish[i] = false$  for  $i = 0, 1, \dots, n-1$ .
- Find an *i* such that both:  
 (a)  $Finish[i] = false$   
 (b)  $Need_i \leq Work$   
 If no such *i* exists, go to step 4.
- $Work = Work + Allocation_i$   
 $Finish[i] = true$   
 go to step 2.
- If  $Finish[i] == true$  for all *i*, then the system is in a safe state.

➤ This algorithm may require an  $O(m \times n^2)$  operations to determine whether a state is safe

AE4B330SS

Lecture 7 / Page 25

Silberschatz, Galvin and Gagne ©2005

### Resource-Request Algorithm for Process P<sub>i</sub>

$Request_i$  = request vector for process P<sub>i</sub>.  
 $Request_i[j] == k$  means that process P<sub>i</sub> wants *k* instances of resource type R<sub>j</sub>.

- If  $Request_i \leq Need_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
- If  $Request_i \leq Available$ , go to step 3. Otherwise P<sub>i</sub> must wait, since resources are not available.
- Pretend to allocate requested resources to P<sub>i</sub> 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<sub>i</sub>.
  - If unsafe  $\Rightarrow$  P<sub>i</sub> must wait for Request<sub>i</sub>, and the old resource-allocation state is restored

AE4B330SS

Lecture 7 / Page 26

Silberschatz, Galvin and Gagne ©2005

### Example of Banker's Algorithm

- 5 processes P<sub>0</sub> through P<sub>4</sub>; 3 resource types A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T<sub>0</sub>:

	<u>Allocation</u>			<u>Max</u>	<u>Need</u>	<u>Total</u>
	A	B	C	A	B	C
P <sub>0</sub>	0	1	0	7	5	3
P <sub>1</sub>	2	0	0	3	2	2
P <sub>2</sub>	3	0	2	9	0	2
P <sub>3</sub>	2	1	1	2	2	0
P <sub>4</sub>	0	0	2	4	3	1
				<u>Allocated</u>		
				7 2 5		
				<u>Available</u>		
				3 3 2		

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

AE4B330SS

Lecture 7 / Page 27

Silberschatz, Galvin and Gagne ©2005

### Example (Cont.): P<sub>1</sub> requests (1,0,2)

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

	<u>Allocation</u>			<u>Max</u>	<u>Need</u>	<u>Available</u>
	A	B	C	A	B	C
P <sub>0</sub>	0	1	0	7	5	3
P <sub>1</sub>	3	0	2	3	2	2
P <sub>2</sub>	3	0	2	9	0	2
P <sub>3</sub>	2	1	1	2	2	0
P <sub>4</sub>	0	0	2	4	3	1
				<b>0 2 0</b>		

- Executing safety algorithm shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement.
- Can request for (3,3,0) by P<sub>4</sub> be granted? No
- Can request for (0,2,0) by P<sub>0</sub> be granted? No

AE4B330SS

Lecture 7 / Page 28

Silberschatz, Galvin and Gagne ©2005

### Deadlock Detection

- If there are no prevention or avoidance mechanisms in place, then deadlock may occur.
- For deadlock detection, the system must provide
  - An algorithm that examines the state of the system to detect whether a deadlock has occurred
  - And an algorithm to recover from the deadlock
- A detection-and-recovery scheme requires various kinds of overhead
  - Run-time costs of maintaining necessary information and executing the detection algorithm
  - Potential losses inherent in recovering from a deadlock

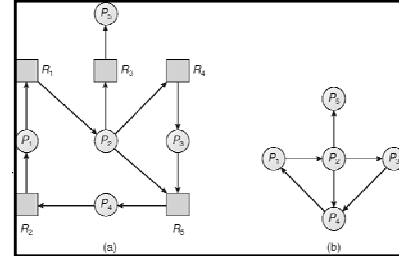
AE4B330SS

Lecture 7 / Page 29

Silberschatz, Galvin and Gagne ©2005

### Single Instance of Each Resource Type

- Requires the creation and maintenance of a wait-for graph (WFG)
  - The WFG is obtained by **removing** the resource nodes from a resource-allocation graph and **collapsing** the appropriate edges
  - Consequently; all nodes are processes in WFG
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$



Resource-Allocation Graph

Corresponding wait-for graph

AE4B330SS

Lecture 7 / Page 30

Silberschatz, Galvin and Gagne ©2005

### Single Instance of Each Resource Type

- Periodically invoke an algorithm that searches for a cycle in the WFG
  - If there is a cycle, there exists a deadlock
  - An algorithm to detect a cycle in a graph requires an  $O(n^2)$  operations, where  $n$  is the number of vertices in the graph

AE4B330SS

Lecture 7 / Page 31

Silberschatz, Galvin and Gagne ©2005

### Several Instances of a Resource Type

- If a resource type can have multiple instances, then an algorithm very similar to the banker's algorithm can be used.

#### Required data structures:

- **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  $R_j$ .

AE4B330SS

Lecture 7 / Page 32

Silberschatz, Galvin and Gagne ©2005



### Detection Algorithm

- Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:  
 $Work = Available$   
 For  $i = 0, 2, \dots, n-1$ , if  $Allocation_i \neq 0$ , then  $Finish[i] = false$ ;  
 otherwise,  $Finish[i] = true$ .
- 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.
- $Work = Work + Allocation_i$   
 $Finish[i] = true$   
 go to step 2.
- If  $Finish[i] == false$ , for some  $i$ ,  $0 \leq i < n$ , then the system is in deadlock state. Moreover, if  $Finish[i] == false$ , then process  $P_i$  is deadlocked  
 The algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlock state.

AE4B330SS

Lecture 7 / Page 33

Silberschatz, Galvin and Gagne ©2005

### 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>Total</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	7	2	6
$P_1$	2	0	0	2	0	2	<u>Allocated</u>		
$P_2$	3	0	3	0	0	0	7	2	6
$P_3$	2	1	1	1	0	0	<u>Available</u>		
$P_4$	0	0	2	0	0	2	0	0	0

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

Hence the system is not in a deadlock state

AE4B330SS

Lecture 7 / Page 34

Silberschatz, Galvin and Gagne ©2005

### Example (Cont.)

- $P_2$  requests an additional instance of type C. The *Request* matrix changes

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

- State of system?
  - System would now get deadlocked
  - Can reclaim resources held by process  $P_0$ , but not sufficient resources to fulfill the requests of the other processes;
  - Deadlock exists, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$ .

AE4B330SS

Lecture 7 / Page 35

Silberschatz, Galvin and Gagne ©2005

### Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- 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.
- If the detection algorithm is invoked for every resource request, such an action will incur a considerable **overhead** in computation time
- A less expensive alternative is to invoke the algorithm once per hour or when CPU utilization drops **below 40%**

AE4B330SS

Lecture 7 / Page 36

Silberschatz, Galvin and Gagne ©2005

## Recovery from Deadlock

- Two Approaches
  - Process termination
  - Resource preemption

AE4B330SS

Lecture 7 / Page 37

Silberschatz, Galvin and Gagne ©2005

## Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - This approach will break the deadlock, but at great expense
- Abort one process at a time until the deadlock cycle is eliminated
  - This approach incurs considerable overhead
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer the process will compute before completion.
  - Resources the process has used.
  - Resources process needs in order to complete its task.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

AE4B330SS

Lecture 7 / Page 38

Silberschatz, Galvin and Gagne ©2005

## Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When 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? We should abort a processes whose termination will incur minimum cost
  - **Rollback** – If we preempt a resource from a process, what should be done with that process? We must rollback the process to some safe state and restart it from that state
  - **Starvation** – Same process may always be picked as victim and this process never completes its task. Must ensure that a process can be picked as a victim only finite number of times. Simple solution is to include the number of rollbacks in the cost factor

AE4B330SS

Lecture 7 / Page 39

Silberschatz, Galvin and Gagne ©2005