

Part II

Process Management

Chapter 7: Deadlocks

System Model

□ System resources are utilized in the following way:

❖ **Request:** If a process makes a request to use a system resource which cannot be granted immediately, then the requesting process blocks until it can acquire the resource.

❖ **Use:** The process can operate on the resource.

❖ **Release:** The process releases the resource.

□ ***Deadlock Definition:*** A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.

Deadlock: Necessary Conditions

□ *For a deadlock to occur, each of the following four conditions must hold.*

- ❖ **Mutual Exclusion:** At least one resource must be held in a non-sharable way.
- ❖ **Hold and Wait:** A process must be holding a resource and waiting for another.
- ❖ **No Preemption:** Resource cannot be preempted.
- ❖ **Circular Wait:** A waits for B, B waits for C, C waits for A.

Handling Deadlocks

- ❑ **Deadlock Prevention and Avoidance:** Making sure deadlocks will never happen.
 - ❖ **Prevention:** Ensuring one of the four conditions fails.
 - ❖ **Avoidance:** The OS needs more information so that it can determine if the current request can be satisfied or delayed.
- ❑ **Deadlock :** Allow a system to enter a deadlock situation, and then detect and recover.
- ❑ **Ignore Deadlock:** Pretend deadlocks never occur in the system.

Deadlock Prevention: 1/4

Mutual Exclusion

- By ensuring that at least one of the four conditions cannot hold, we can **prevent** the occurrence of a deadlock.
- **Mutual Exclusion**: Some sharable resources must be accessed exclusively (*e.g.*, printer), which means we cannot deny the mutual exclusion condition.

Deadlock Prevention: 2/4

Hold and Wait

- ❑ No process can hold some resources and then request for other resources.
- ❑ Two strategies are possible:
 - ❖ A process must acquire *all* resources before it runs.
 - ❖ When a process requests for resources, it must hold none (*i.e.*, returning resources before requesting for more).
- ❑ **Resource utilization** may be low, since many resources will be held and unused for a long time.
- ❑ **Starvation** is possible. A process that needs some popular resources may have to wait indefinitely.

Deadlock Prevention: 3/4

No Preemption

- Resources that are being held by the requesting process are preempted. There are two strategies:
 - ❖ If a process is holding some resources and requesting for some others that are being held by other processes, **the resources of the requesting process are preempted**. The preempted resources become available.
 - ❖ If the requested resources are not available:
 - If they are being held by processes that are waiting for additional resources, these resources are preempted and given to the requesting process.
 - Otherwise, the requesting process waits until the requested resources become available. While it is waiting, its resources may be preempted.
 - **This works only if the state of the process and resources can be saved and restored easily (e.g., CPU & memory⁷).**

Deadlock Prevention: 4/4

Circular Waiting

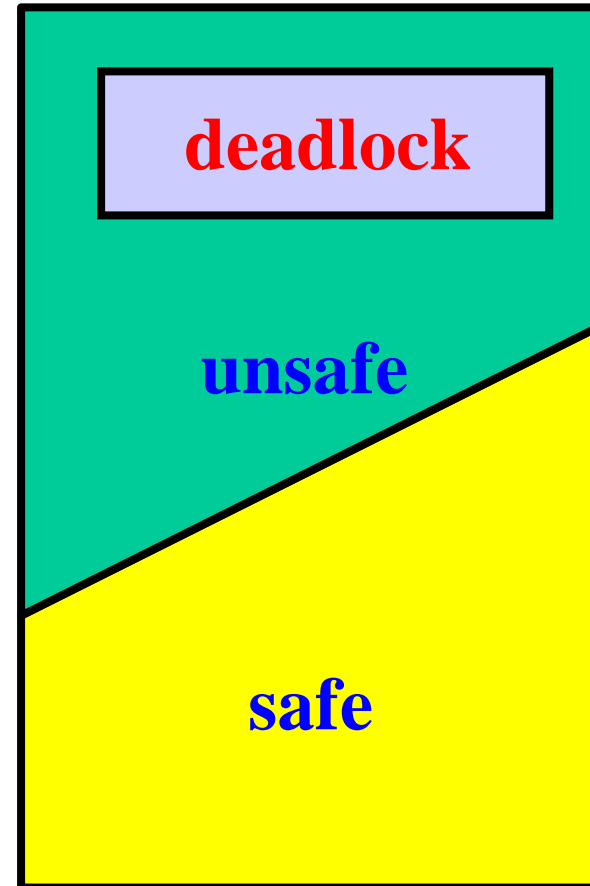
- ❑ To break the circular waiting condition, we can order all resource types (*e.g.*, tapes, printers).
- ❑ A process can only request resources higher than the resource types it holds.
- ❑ Suppose the ordering of tapes, disks, and printers are 2, 4, and 8. If a process holds a disk (4), it can ask for a printer (8) and cannot request a tape (2).
- ❑ A process must release some lower order resources to request a lower order resource. To get tapes (1), a process must release its disk (4).
- ❑ In this way, no deadlock is possible. Why?

Deadlock Avoidance: 1/5

- ❑ Each process provides the **maximum number of resources of each type** it needs.
- ❑ With these information, there are algorithms that can ensure the system will never enter a deadlock state. This is *deadlock avoidance*.
- ❑ A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a *safe sequence* if for each process P_i in the sequence, its resource requests can be satisfied by the **remaining resources** and **the sum of all resources** that are being held by P_1, P_2, \dots, P_{i-1} . This means we can suspend P_i and run P_1, P_2, \dots, P_{i-1} until they complete. Then, P_i will have all resources to run.

Deadlock Avoidance: 2/5

- ❑ A state is *safe* if the system can allocate resources to each process (up to its maximum, of course) in some order and still avoid a deadlock.
- ❑ Thus, a state is *safe* if there is a safe sequence. Otherwise, if no safe sequence exists, the system state is *unsafe*.
- ❑ An unsafe state is not necessarily a deadlock state. On the other hand, a **deadlock state is an unsafe state.**



Deadlock Avoidance: 3/5

- A system has 12 tapes and three processes *A*, *B*, *C*.

At time t_0 , we have:

	Max needs	Current holding	Will need
<i>A</i>	10	5	5
<i>B</i>	4	2	2
<i>C</i>	9	2	7

3 free tapes

- Then, $\langle B, A, C \rangle$ is a safe sequence (safe state).
- The system has $12 - (5 + 2 + 2) = 3$ free tapes.
- Since *B* needs 2 tapes, it can take 2, run, and return 4. After *B* completes, the system has $(3 - 2) + 4 = 5$ tapes. *A* now can take all 5 tapes and run. Finally, *A* returns 10 tapes for *C* to take 7 of them.

Deadlock Avoidance: 4/5

- A system has 12 tapes and three processes *A*, *B*, *C*. At time t_1 , *C* has one more tape:

	Max needs	Current holding	Will need
<i>A</i>	10	5	5
<i>B</i>	4	2	2
<i>C</i>	9	3	6

- The system has $12 - (5 + 2 + 3) = 2$ free tapes.
- At this point, only *B* can take these 2 and run. It returns 4, making 4 free tapes available.
- But, none of *A* and *C* can run, and a deadlock occurs.
- The problem is due to granting *C* one more tape.

Deadlock Avoidance: 5/5

- ❑ A *deadlock avoidance algorithm* ensures that the system is always in a safe state. Therefore, no deadlock can occur.
- ❑ Resource requests are granted only if in doing so the system is still in a safe state.
- ❑ Consequently, resource utilization may be *lower* than those systems without using a deadlock avoidance algorithm.

Banker's Algorithm: 1/4

- ❑ The system has m resource types and n processes.
- ❑ Each process must declare its maximum needs.
- ❑ The following arrays are used:
 - ❖ $Available[1..m]$: one entry for each resource. $Available[i]=k$ means resource type i has k units available.
 - ❖ $Max[1..n, 1..m]$: maximum demand of each process. $Max[i,j]=k$ means process i needs k units of resource j .
 - ❖ $Allocation[1..n, 1..m]$: resources allocated to each process. $Allocation[i,j]=k$ means process i is currently allocated k units of resource j .
 - ❖ $Need[1..n, 1..m]$: the remaining resource need of each process. $Need[i,j]=k$ means process i needs k more units of resource j . Thus, $Max = Allocation + Need$.

Banker's Algorithm: 2/4

- We will use $A[i, *]$ to indicate the i -th row of matrix A .
- Given two arrays $A[1..m]$ and $B[1..m]$, $A \leq B$ if $A[i] \leq B[i]$ for all i . Given two matrices $A[1..n, 1..m]$ and $B[1..n, 1..m]$, $A[i, *] \leq B[i, *]$ if $A[i, j] \leq B[i, j]$ for all j .
- When a resource request is made by process i , this algorithm calls the **Resource-Request** algorithm to determine if the request can be granted. The **Resource-Request** algorithm calls the **Safety Algorithm** to determine if a state is safe.

Banker's Algorithm: 3/4

Safety Algorithm

1. Let $Work[1..m]$ and $Finish[1..n]$ be two working arrays.
2. $Work := Available$ and $Finish[i] = FALSE$ for all i
3. Find an i such that both
 - ❖ $Finish[i] = FALSE$ // process i is not yet done
 - ❖ $Need[i,*] \leq Work$ // its need can be satisfiedIf no such i exists, go to Step 5
4. $Work = Work + Allocation[i,*]$ // run it and reclaim
 $Finish[i] = TRUE$ // process i completes
go to Step 3
5. If $Finish[i] = TRUE$ for all i , the system is in a safe state.

Banker's Algorithm: 4/4

Resource-Request Algorithm

1. Let $Request[1..n, 1..m]$ be the request matrix. $Request[i, j] = k$ means process i requests k units of resource j .
2. If $Request[i, *] \leq Need[i, *]$, go to Step 3. Otherwise, it is an error.
3. If $Request[i, *] \leq Available$, go to Step 4. Otherwise, process i waits.
4. Do the following:

$$Available = Available - Request[i, *]$$

$$Allocation[i, *] = Allocation[i, *] + Request[i, *]$$

$$Need[i, *] = Need[i, *] - Request[i, *]$$

If the result is a safe state (Safety Algorithm), the request is granted. Otherwise, process i waits and the

resource-allocation tables are restored back to the original.

Example: 1/4

- Consider a system of 5 processes A, B, C, D and E , and 3 resource types ($X=10, Y=5, Z=7$). At time t_0 , we have

	<i>Allocation</i>			<i>Max</i>			<i>Need=Max-Alloc</i>			<i>Available</i>		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
A	0	1	0	7	5	3	7	4	3	3	3	2
B	2	0	0	3	2	2	1	2	2			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- A safe sequence is $\langle B, D, E, C, A \rangle$. Since B 's $[1, 2, 2] \leq \text{Avail}$'s $[3, 3, 2]$, B runs. Then, $\text{Avail} = [2, 0, 0] + [3, 3, 2] = [5, 3, 2]$. D runs next. After this, $\text{Avail} = [5, 3, 2] + [2, 1, 1] = [7, 4, 3]$. E runs next.
- $\text{Avail} = [7, 4, 3] + [0, 0, 2] = [7, 4, 5]$. Since C 's $[6, 0, 0] \leq \text{Avail} = [7, 4, 5]$, C runs. After this, $\text{Avail} = [7, 4, 5] + [3, 0, 2] = [10, 4, 7]$ and A runs.
- There are other safe sequences: $\langle D, E, B, A, C \rangle$, $\langle D, B, A, E, C \rangle$, ...

Example: 2/4

- Now suppose process **B** asks for 1 **X** and 2 **Z**s. More precisely, $Request_B = [1,0,2]$. *Is the system still in a safe state if this request is granted?*
- Since $Request_B = [1,0,2] \leq Available = [3,3,2]$, this request may be granted as long as the system is safe.
- If this request is actually granted, we have the following:

	<i>Allocation</i>			<i>Max</i>			<i>Need=Max-Alloc</i>			<i>Available</i>		
	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>
<i>A</i>	0	1	0	7	5	3	7	4	3	2	3	0
<i>B</i>	3	0	2	3	2	2	0	2	0			
<i>C</i>	3	0	2	9	0	2	6	0	0			
<i>D</i>	2	1	1	2	2	2	0	1	1			
<i>E</i>	0	0	2	4	3	3	4	3	1			

$[3,0,2] = [2,0,0] + [1,0,2]$
 $[0,2,0] = [1,2,2] - [1,0,2]$
 $[2,3,0] = [3,3,2] - [1,0,2]$

Example: 3/4

	<i>Allocation</i>			<i>Max</i>			<i>Need=Max-Alloc</i>			<i>Available</i>		
	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>
<i>A</i>	0	1	0	7	5	3	7	4	3	2	3	0
<i>B</i>	3	0	2	3	2	2	0	2	0			
<i>C</i>	3	0	2	9	0	2	6	0	0			
<i>D</i>	2	1	1	2	2	2	0	1	1			
<i>E</i>	0	0	2	4	3	3	4	3	1			

- ❑ *Is the system in a safe state after this allocation?*
- ❑ Yes, because the safety algorithm will provide a safe sequence $\langle B, D, E, A, C \rangle$. Verify it yourself.
- ❑ Hence, *B*'s request of [1,0,2] can safely be made.

Example: 4/4

	<i>Allocation</i>			<i>Max</i>			<i>Need=Max-Alloc</i>			<i>Available</i>		
	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>
<i>A</i>	0	1	0	7	5	3	7	4	3	2	3	0
<i>B</i>	3	0	2	3	2	2	0	2	0			
<i>C</i>	3	0	2	9	0	2	6	0	0			
<i>D</i>	2	1	1	2	2	2	0	1	1			
<i>E</i>	0	0	2	4	3	3	4	3	1			

- ❑ After this allocation, *E*'s request $Request_E = [3, 3, 0]$ cannot be granted since $Request_E = [3, 3, 0] \leq [2, 3, 0]$ is false.
- ❑ *A*'s request $Request_A = [0, 2, 0]$ cannot be granted because the system will be unsafe.
- ❑ If $Request_A = [0, 2, 0]$ is granted, $Available = [2, 1, 0]$.
- ❑ None of the five processes can finish and the system is unsafe.

Deadlock Detection

- If a system does not use a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. Thus, we need
 - ❖ An algorithm that can examine the system state to determine if a deadlock has occurred. This is a *deadlock detection* algorithm.
 - ❖ An algorithm that can help recover from a deadlock. This is a *recovery* algorithm.
- A deadlock detection algorithm does not have to know the maximum need *Max* and the current need *Need*. It uses only *Available*, *Allocation* and *Request*.

Deadlock Detection Algorithm

1. Let $Work[1..m]$ and $Finish[1..n]$ be two working arrays.
2. $Work := Available$ and $Finish[i] = FALSE$ for all i
3. Find an i such that both

❖ $Finish[i] = FALSE$ // process i is not yet done

❖ $Request[i, *] \leq Work$ // its request can be satisfied

If no such i exists, go to Step 5

4. $Work = Work + Allocation[i, *]$ // run it and reclaim

$Finish[i] = TRUE$ // process i completes

go to Step 3

5. If $Finish[i] = TRUE$ for all i , the system is in a safe state. If $Finish[i] = FALSE$, then process P_i is deadlocked.

Use *Request* here rather than *Need* as in the safety algorithm²³

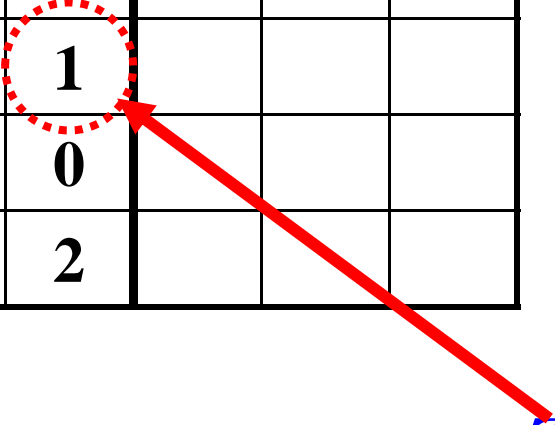
Example: 1/2

	<i>Allocation</i>			<i>Request</i>			<i>Available</i>		
	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>
<i>A</i>	0	1	0	0	0	0	0	0	0
<i>B</i>	2	0	0	2	0	2			
<i>C</i>	3	0	3	0	0	0			
<i>D</i>	2	1	1	1	0	0			
<i>E</i>	0	0	2	0	0	2			

- Suppose maximum available resource is [7,2,6] and the current state of resource allocation is shown above.
- *Is the system deadlocked?* No. We can run *A* first, making *Available*=[0,1,0].
- Then, we run *C*, making *Available*=[3,1,3]. This is followed by *D*, making *Available*=[5,2,4], and followed by *B* and *E*,

Example: 2/2

	<i>Allocation</i>			<i>Request</i>			<i>Available</i>		
	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>	<i>X</i>	<i>Y</i>	<i>Z</i>
<i>A</i>	0	1	0	0	0	0	0	0	0
<i>B</i>	2	0	0	2	0	2			
<i>C</i>	3	0	3	0	0	1			
<i>D</i>	2	1	1	1	0	0			
<i>E</i>	0	0	2	0	0	2			



- ❑ Suppose *C* requests for one more resource *Z*.
- ❑ Now, *A* can run, making *Available*=[0,1,0].
- ❑ However, none of *B*, *C*, *D* and *E* can run.
Therefore, *B*, *C*, *D* and *E* are deadlocked!

The Use of a Detection Algorithm

□ Frequency

- ❖ If deadlocks occur frequently, the detection algorithm should be invoked frequently.
- ❖ **Once per hour** or whenever **CPU utilization becomes low** (*i.e.*, below 40%). Low CPU utilization usually means more processes are waiting.

How to Recover: 1/3

- ❑ When a detection algorithm determines a deadlock has occurred, the algorithm may inform the system administrator to deal with it. Of, allow the system to *recover* from a deadlock.
- ❑ There are two options.
 - ❖ Process Termination
 - ❖ Resource Preemption
- ❑ These two options are not mutually exclusive (*i.e.*, can have both if needed).

Recovery: Process Termination: 2/3

- ❑ Abort all deadlocked processes
- ❑ Abort one process at a time until the deadlock cycle is eliminated
- ❑ **Problems:**
 - ❖ Aborting a process may not be easy. What if a process is updating or printing a large file? The system must find some way to maintain the states of the files and printer before they can be reused.
 - ❖ Termination may be determined by the priority/importance of a process.

Recovery: Resource Preemption:

3/3

- ❑ **Selecting a victim:** which resources and which processes are to be preempted?
- ❑ **Rollback:** If we preempt a resource from a process, what should be done with that process?
 - ❖ **Total Rollback:** abort the process and restart it
 - ❖ **Partial Rollback:** rollback the process only as far as necessary to break the deadlock.
- ❑ **Starvation:** We cannot always pick the same process as a victim. Some **limit** must be set.

The End