

Chapter 10

Deadlock

What is Deadlock?

- Two or more entities need a resource to make progress, but will never get that resource
- Examples from everyday life:
 - Gridlock of cars in a city
 - Class scheduling: Two students want to swap sections of a course, but each section is currently full.
- Examples from Operating Systems:
 - Two processes spool output to disk before either finishes, and all free disk space is exhausted
 - Two processes consume all memory buffers before either finishes

Deadlock Illustration

A set of processes is in a DEADLOCK state when every process is waiting for an event initiated by another process in the set

Process A

Request X
Request Y
⋮
Release X
Release Y

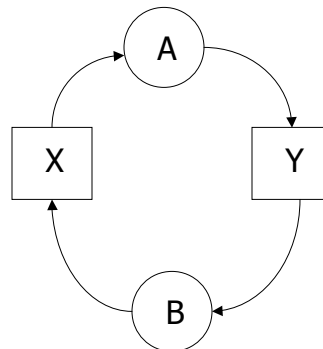
Process B

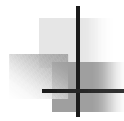
Request Y
Request X
⋮
Release Y
Release X

Deadlock Illustration

- A requests & receives X
- B requests & receives Y
- A requests Y and blocks
- B requests X and blocks

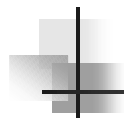
The "Deadly Embrace"





Terminology

- Preemptible vs. Non-preemptible
- Shared vs. Exclusive resource
 - Example of Shared resource: File
 - Example of Exclusive resource: Printer



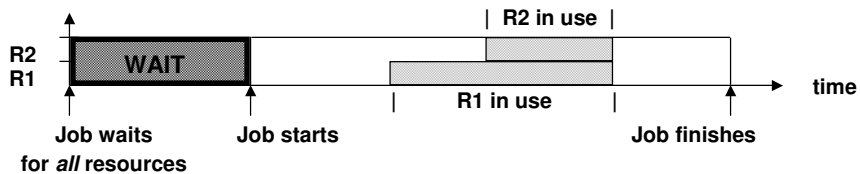
Terminology ...

- Indefinite postponement
 - Job is continually denied resources needed to make progress

Example: High priority processes keep CPU busy 100% of time, thereby denying CPU to low priority processes

Three Solutions to Deadlock

#1: Mr./Ms. Conservative (*Prevention*)

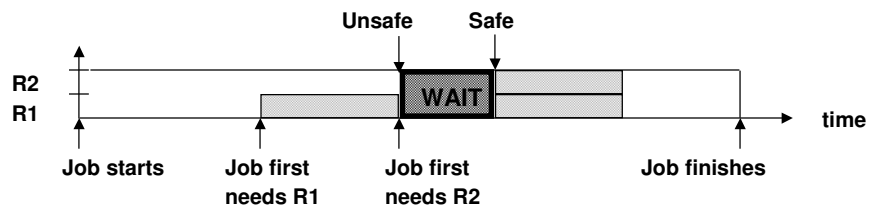


“We had better not allocate if it could ever cause deadlock”

Process **waits** until all needed resource free
Resources **underutilized**

Three Solutions to Deadlock ...

#2: Mr./Ms. Prudent (*Avoidance*)

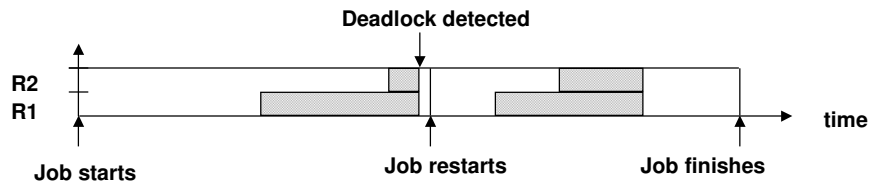


“If resource is free and with its allocation we can still guarantee that everyone will finish, **use it.**”

Better resource utilization
Process still waits

Three Solutions to Deadlock...

#3: Mr./Ms. Liberal (*Detection/Recovery*)



"If it's free, use it -- why wait?"

Good resource utilization, minimal process wait time
Until deadlock occurs....

Names for Three Methods on Last Slide

1) Deadlock Prevention

- Design system so that possibility of deadlock is avoided *a priori*

2) Deadlock Avoidance

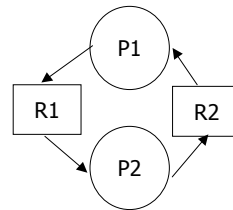
- Design system so that if a resource request is made that *could* lead to deadlock, then block requesting process.
- Requires knowledge of future requests by processes for resources.

3) Deadlock Detection and Recovery

- Algorithm to detect deadlock
- Recovery scheme

4 Necessary Conditions for Deadlock

- Mutual Exclusion
 - Non-sharable resources
- Hold and Wait
 - A process must be holding resources and waiting for others
- No pre-emption
 - Resources are released voluntarily
- Circular Wait



Deadlock Prevention

Deny one or more of the necessary conditions

- Prevent "Mutual Exclusion"
 - Use only sharable resources

=> Impossible for practical systems

Deadlock Prevention ...

- Prevent "Hold and Wait"
 - (a) Preallocation - process must request and be allocated all of its required resources before it can start execution
 - (b) Process must release all of its currently held resources and re-request them along with request for new resources
- => Very inefficient
- => Can cause "indefinite postponement": jobs needing lots of resources may never run

Deadlock Prevention ...

- Allow "Resource Preemption"
 - Allowing one process to acquire exclusive rights to a resource currently being used by a second process
- => Some resources can not be preempted without detrimental implications (e.g., printers, tape drives)
- => May require jobs to restart

Deadlock Prevention ...

- Prevent Circular Wait
 - Order resources and
 - Allow requests to be made only in an increasing order

Preventing Circular Wait

Impose an ordering on Resources:

1	W
2	X
3	Y
4	Z

Process:	A	B	C	D	A	B	C	D
Request:	W	X	Y	Z	X	Y	Z	W

After first 4 requests:

D / Z	A / W	B / X
	C / Y	

Process D cannot request resource W
without voluntarily releasing Z first

Problems with Linear Ordering Approach

- (1) Adding a new resource that upsets ordering requires all code ever written for system to be modified!
- (2) Resource numbering affects efficiency
 - => A process may have to request a resource well before it needs it, just because of the requirement that it must request resources in ascending sequence

Deadlock Avoidance

- OS never allocates resources in a way that could lead to deadlock
 - => Processes must tell OS in advance how many resources they will request

Banker's Algorithm

- Banker's Algorithm runs each time:
 - a process requests resource - *Is it Safe?*
 - a process terminates - *Can I allocate released resources to a suspended process waiting for them?*
- A new state is safe if and only if every process can complete after allocation is made
 - => Make allocation, then check system state and de-allocate if safe/unsafe

Definition: Safe State

- State of a system
 - An enumeration of which processes hold, are waiting for, or might request which resources
- Safe state
 - No process is deadlocked, and there exists no possible sequence of future requests in which deadlock could occur.
 - or alternatively,
 - No process is deadlocked, and the current state will not lead to a deadlocked state

Deadlock Avoidance

Safe State:

	Current Loan	Max Need
Process 1	1	4
Process 2	4	6
Process 3	5	8

Available = 2

Deadlock Avoidance

Unsafe State:

	Current Loan	Max Need
Process 1	8	10
Process 2	2	5
Process 3	1	3

Available = 1

Safe to Unsafe Transition

Current state being safe does not necessarily imply future states are safe

Current Safe State:

	Current Loan	Maximum Need	
Process 1	1	4	
Process 2	4	6	
Process3	5	8	Available = 2

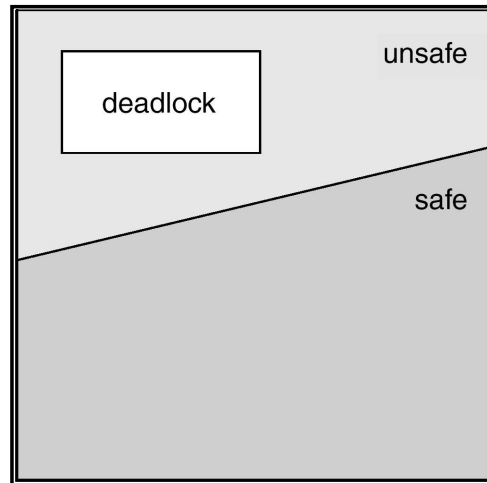
Suppose Process 3 requests and gets one more resource

	Current Loan	Maximum Need	
User1	1	4	
User2	4	6	
User3	6	8	Available = 1

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.

Safe, Unsafe , Deadlock State



Banker's Algorithm

Taken from Operating System Concepts, 6th Ed, Silberschatz, et al, 2003

- 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.

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.

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

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize:
 $Work = Available$
 $Finish[i] = false$ for $i = 1, 2, 3, \dots, n$.
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.

Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively. Initialize:

$Work = Available$

$Finish[i] = false$ for $i = 1, 2, 3, \dots, n$.

$i = 1$;

while ($i \leq n$) Do {

if ($\neg Finish[i] \ \&\& \ Need_i \leq Work$) {

$Finish[i] = True$;

$Work = Work + Allocation_i$;

$i = i + 1$;

}

else $i++$;

}

if ($Finish[i] == true$ for all i) return (SAFE)

else return (UNSAFE);

Resource-Request Algorithm for Process P_i

$Request$ = 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

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

Example (Cont.)

- The content of the matrix. *Need* is defined to be $Max - Allocation$.

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

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

Example P_1 Request (1,0,2) (Cont.)

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

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	1	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_0, P_2, P_4 \rangle$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Banker's Algorithm: Summary

(+) PRO's:

- ⊙ Deadlock never occurs.
- ⊙ More flexible & more efficient than deadlock prevention. (Why?)

(-) CON's:

- ⊙ Must know max use of each resource when job starts.
 - => No truly dynamic allocation
- ⊙ Process might block even though deadlock would never occur

Deadlock Detection

Allow deadlock to occur, then recognize that it exists

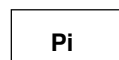
- Run deadlock detection algorithm whenever locked resource is requested
- Could also run detector in background

Resource Graphs

Graphical model of deadlock

Nodes:

1) Processes



2) Resources

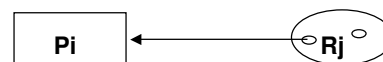


Edges:

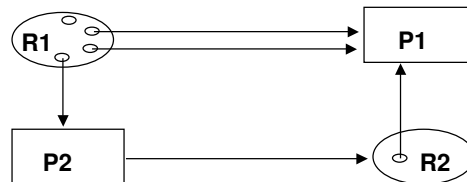
1) Request



2) Allocate



Resource Graphs: Example



P1 holds 2 units of R1

P1 holds 1 unit of R2

R1 has a total inventory of 4 units

P2 holds 1 unit of R1

P2 requests 1 unit of R2 (and is blocked)

Operations on Resource Graphs: An Overview

- 1) Process requests resources: Add arc(s)
- 2) Process acquires resources: Reverse arc(s)
- 3) Process releases resources: Delete arc(s)



Graph Reductions

- A graph is reduced by performing operations 2 and 3 (reverse, delete arc)
- A graph is completely reducible if there exists a sequence of reductions that reduce the graph to a set of isolated nodes
- A process P is not deadlocked if and only if there exists a sequence of reductions that leave P unblocked
- If a graph is completely reducible, then the system state it represents is not deadlocked



Operations on Resource Graphs: Details

1) P requests resources (Add arc)

Precondition:

- P must have no outstanding requests
- P can request any number of resources of any type

Operation:

- Add one edge (P, R_j) for each resource copy R_j requested

2) P acquires resources (Reverse arc)

Precondition:

- Must be available units to grant all requests
- P acquires all requested resources

Operation:

- Reverse all request edges directed from P toward resources

Operations on Resource Graphs: Details ...

3) P releases resources (Delete arc)

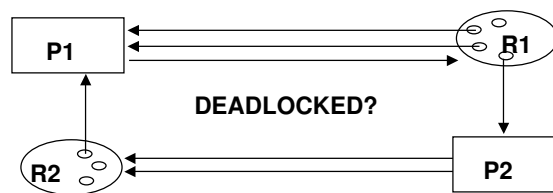
Precondition:

- P must have no outstanding requests
- P can release any subset of resources that it holds

Operation:

- Delete one arc directed away from resource for each released resource

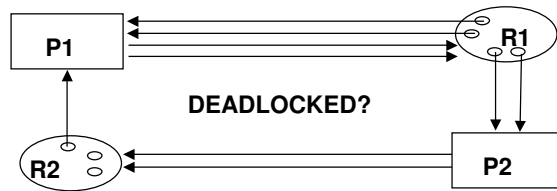
Resource Graphs



NO....One sequence of reductions:

- 1) P1 acquires 1 unit of R1
- 2) P1 releases all resources (finishes)
- 3) P2 acquires 2 units of R2
- 4) P2 releases all resources (finishes)

Resource Graphs ...



NO.... One sequence of Reductions:

- 1) P2 acquires 2 units of R2
- 2) P2 releases all resources (finishes)
- 3) P1 acquires 2 units of R1
- 4) P1 releases all resources (finishes)

Resource Graphs...

What if there was only 2 available unit of R2 ?

?

Can deadlock occur with multiple copies of just one resource?



Recovering from Deadlock

Once deadlock has been detected, the system must be restored to a non-deadlocked state

- 1) Kill one or more processes
 - Might consider priority, time left, etc. to determine order of elimination
- 2) Preempt resources
 - Preempted processes must rollback
 - Must keep ongoing information about running processes