

# CSC 539: Operating Systems Structure and Design

Spring 2006

## Process deadlock

- deadlock prevention
- deadlock avoidance
- deadlock detection
- recovery from deadlock

1

## Process deadlock

in general, can partition system resources into equivalence types or classes

e.g, CPUs, RAM, files, printers, tape drives, ...

when a process requests a resource from a particular resource class,  
any available resource in the class is allocated to the process

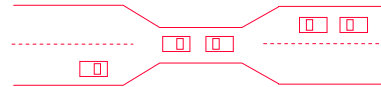
- generally, resources go through the sequence:
  1. request
  2. use
  3. release

*deadlock* occurs when a set of processes is waiting for resources that are held by other processes in the set

2

## Deadlock examples

real-world example: single-lane bridge



- 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 & rollback)
- several cars may have to be backed up if a deadlock occurs
- starvation is possible

system example: 2 tape drives

- $P_1$  and  $P_2$  each hold one tape drive and each needs another one

system example: semaphores A and B, initialized to 1

```
P0: wait(A);      P1: wait(B);
      wait(B);      wait(A);
```

3

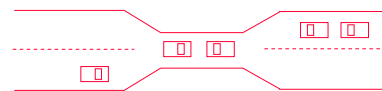
## Deadlock characterization

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  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that:
  - $P_0$  is waiting for a resource held by  $P_1$
  - $P_1$  is waiting for a resource held by  $P_2$
  - ...
  - $P_n$  is waiting for a resource held by  $P_0$

consider single-lane bridge example:

- 4 necessary conditions?

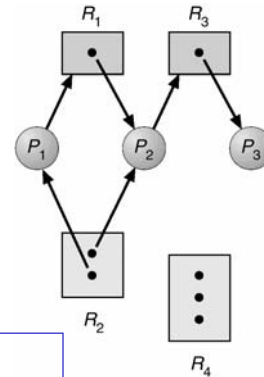


4

## System resource allocation graph

we can use a directed graph to represent the state of the system as it allocates resources and deals with requests for resources

- nodes of graph represent processes or resources
  - process nodes will have circles around them
  - resource nodes will have rectangles around them
- edges of graph represent allocations or requests
  - process to resource edge is a request
  - resource to process edge is an allocation



in this example:

- $P_1$  has  $R_2$ , has requested  $R_1$
- $P_2$  has  $R_1$  &  $R_2$ , has requested  $R_3$
- $P_3$  has  $R_3$

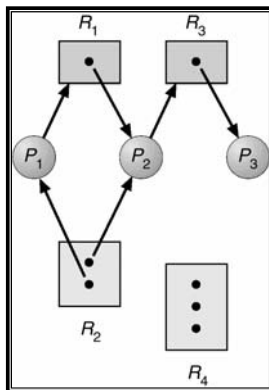
5

## cycle = deadlock?

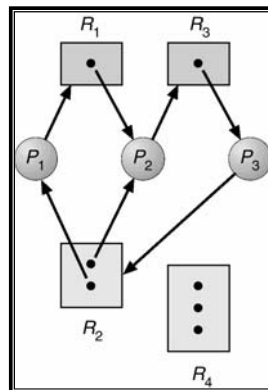
If graph contains no cycles  $\rightarrow$  no deadlock

If graph contains a cycle  $\rightarrow$

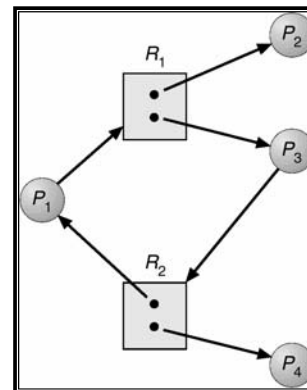
- if only one instance per resource type, then deadlock.
- if several instances per resource type, deadlock is *possible*



no deadlock



deadlock



no deadlock

6

## Methods for dealing with deadlock

*Prevention or Avoidance:* ensure system never enters a deadlocked state

- deadlock prevention scheme  
ensure that at least one of the necessary conditions cannot hold
- deadlock avoidance scheme  
require the OS to know the resource usage requirements of all processes  
for each request, the OS decides if it could lead to a deadlock before granting

*Detection & Recovery:*

- allow deadlocks to occur
- system implements an algorithm for deadlock detection, if so then recover

*Ignorance:*

- assume/pretend deadlocks never occur in the system
- used by most operating systems, including UNIX & Windows

7

## Deadlock prevention

to prevent deadlock, must eliminate one of the four necessary conditions

- **Mutual Exclusion:** can't eliminate since some resources are not shareable
- **Hold and Wait:** options exist, but wasteful and may lead to starvation
  1. grant all resources when process starts, or
  2. allocate and deallocate resources in complete groups
- **No Preemption:** options exist, but taking away resources can be tricky
- **Circular Wait:** most likely candidate  
impose an ordering on resources  $[R_0, R_1, \dots, R_n]$ , and require that each process requests resources in increasing order

e.g.,  $P_0$  wants  $R_0, R_1,$  and  $R_2$   
 $P_1$  wants  $R_1, R_2,$  and  $R_3$

*informal proof that resource ordering ensures no circular wait?*

8

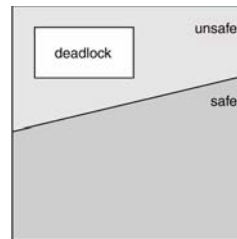
## Deadlock avoidance

if have some knowledge of future resource demands (e.g., max number of resources needed per process), can avoid deadlock situations

- **safe state:** a state is safe if the system can allocate resources to each process (up to its maximum need) in some sequence and still avoid a deadlock
- **safe sequence:** a sequence of processes  $\{P_0, P_1, \dots, P_n\}$  is safe for the current allocation state if, for each  $P_i$ , the resources that  $P_i$  can still request are either free or else held by  $\{P_0, P_1, \dots, P_{i-1}\}$ 
  - a safe sequence implies that each process need only wait for lower numbered processes to finish before it can finish

Note: unsafe  $\neq$  deadlock

- a safe state is guaranteed no deadlock
- an unsafe state *may* lead to deadlock



9

## Example: safe or unsafe?

suppose there are 12 instances of a given resource (e.g., tape drives)  
in the current state, resources are allocated to 3 processes

Process P: max need = 10, currently allocated = 5  
Process Q: max need = 4, currently allocated = 2  
Process R: max need = 9, currently allocated = 2

is this a safe state?

if so, identify a safe sequence.

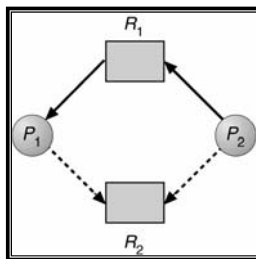
10

## Resource allocation graph algorithm

if there is only one instance of each resource class, then unsafe states can be avoided by considering a generalized resource allocation graph

- allocation edge  $R_i \rightarrow P_j$  if  $R_i$  is currently held by  $P_j$  (as before)
- request edge  $P_i \rightarrow R_j$  if  $P_i$  has requested  $R_j$  (as before)
- claim edge  $P_i \dashrightarrow R_j$  if  $P_i$  may eventually request  $R_j$

unsafe avoidance: do not allocate resource if it creates a cycle



allocate  $R_2$  to  $P_1$ : OK

allocate  $R_2$  to  $P_2$ : NOT ALLOWED

- leads to cycle in graph (unsafe state)

11

## Banker's algorithm

the banker's algorithm can avoid deadlocks even with multiple instances

- looks at each request for resources and tests if the request moves the system into an unsafe state
- if the system is still safe, then the request is granted
- if the system would become unsafe, then the request is denied

utilizes the following data structures:

- Available[m] number of resources of  $R_m$  that are unallocated
- Max[n][m] max demand of  $P_n$  for  $R_m$
- Allocation[n][m] number of  $R_m$  that are allocated to  $P_n$
- Need[n][m] number of  $R_m$  that may be needed by  $P_n$   
*Note: Need[n, m] = Max[n, m] - Allocation[n, m]*

as in C++, will treat a matrix as a vector of vectors (Max[i] is max resources demand for  $P_i$ )  
for vectors  $X$  &  $Y$ ,  $X \leq Y$  if and only if  $X[i] \leq Y[i]$  for all  $i$

12

## Example: banker's algorithm

suppose have: 3 types of resources (6 of type A, 2 of type B, 2 of type C)  
3 processes (P, Q, and R)

	Max	Allocation	Available	Need
	A B C	A B C	A B C	A B C
P	2 1 0	0 1 0	1 1 0	2 0 0
Q	3 1 1	2 0 0		1 1 1
R	5 1 2	3 0 2		2 1 0

is the system in a safe state?

13

## Banker's: safety algorithm

to find out whether a system is in a safe state:

1. define vector:  $Finish[k] = \text{false}$  (for all  $k \leq n$ )
2. find an  $i$  such that  $(Finish[i] == \text{false} \ \&\& \ Need[i] \leq \text{Available})$   
if no such  $i$ , then go to STEP 4
3. update:  $\text{Available} = \text{Available} + \text{Allocation}[i]$ ,  $Finish[i] = \text{true}$   
go to STEP 2
4. if  $(Finish[i] == \text{true})$  for all  $i$ , then the system is in a safe state  
otherwise, the system is in an unsafe state

	Max	Allocation	Available	Need
	A B C	A B C	A B C	A B C
P	2 1 0	0 1 0	2 1 1	2 0 0
Q	3 1 1	2 0 0		1 1 1
R	5 1 2	2 0 1		3 1 1

is this state  
safe?

14

## Banker's: resource request algorithm

let Request[i] be the request vector from  $P_i$

1. if Request[i] > Need[i], then ERROR (asked for too many resources) & EXIT
2. if Request[i] > Available[i], then deny request and  $P_i$  must wait  
(the requested resources are not currently available)
3. otherwise, pretend to allocate the resources  
 $Available = Available - Request[i]$   
 $Allocation[i] = Allocation[i] + Request[i]$   
 $Need[i] = Need[i] - Request[i]$
4. call the Safety algorithm, if safe then allocation is OK  
 otherwise, the request is denied and the pretended allocation is undone

	<b>Max</b>	<b>Allocation</b>	<b>Available</b>	<b>Need</b>	
	A B C	A B C	A B C	A B C	
P	2 1 0	0 1 0	2 1 1	2 0 0	
Q	3 1 1	2 0 0		1 1 1	
R	5 1 2	2 0 1		3 1 1	what if R asks for [1 0 1] ? <small>15</small>

## Example: banker's algorithm

suppose 5 processes:  $P_0$  through  $P_4$   
 3 resource types: A (10), B (5), and C (7)

snapshot at time  $T_0$ :

	<b>Max</b>	<b>Allocation</b>	<b>Available</b>	<b>Need</b>
	A B C	A B C	A B C	A B C
$P_0$	7 5 3	0 1 0	3 3 2	7 4 3
$P_1$	3 2 2	2 0 0		1 2 2
$P_2$	9 0 2	3 0 2		6 0 0
$P_3$	2 2 2	2 1 1		0 1 1
$P_4$	4 3 3	0 0 2		4 3 1

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

16

## Example: banker's algorithm (cont.)

	Max	Allocation	Available	Need
	A B C	A B C	A B C	A B C
P <sub>0</sub>	7 5 3	0 1 0	3 3 2	7 4 3
P <sub>1</sub>	3 2 2	2 0 0		1 2 2
P <sub>2</sub>	9 0 2	3 0 2		6 0 0
P <sub>3</sub>	2 2 2	2 1 1		0 1 1
P <sub>4</sub>	4 3 3	0 0 2		4 3 1

P<sub>1</sub> requests [1 0 2]

1. Request[i] ≤ Need[i]: [1 0 2] ≤ [1 2 2], so no ERROR
2. Request[i] ≤ Available[i]: [1 0 2] ≤ [3 3 2], so resources are available
3. pretend to allocate the resources

	Max	Allocation	Available	Need
	A B C	A B C	A B C	A B C
P <sub>0</sub>	7 5 3	0 1 0	2 1 0	7 4 3
P <sub>1</sub>	3 2 2	3 2 2		0 0 0
P <sub>2</sub>	9 0 2	3 0 2		6 0 0
P <sub>3</sub>	2 2 2	2 1 1		0 1 1
P <sub>4</sub>	4 3 3	0 0 2		4 3 1

the sequence  
P1, P3, P4, P0, P2  
satisfies the safety  
requirement 17

## Example: banker's algorithm (cont.)

	Max	Allocation	Available	Need
	A B C	A B C	A B C	A B C
P <sub>0</sub>	7 5 3	0 1 0	3 3 2	7 4 3
P <sub>1</sub>	3 2 2	2 0 0		1 2 2
P <sub>2</sub>	9 0 2	3 0 2		6 0 0
P <sub>3</sub>	2 2 2	2 1 1		0 1 1
P <sub>4</sub>	4 3 3	0 0 2		4 3 1

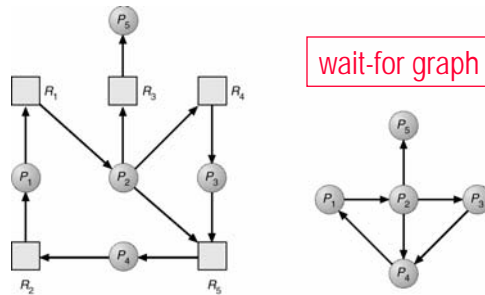
P<sub>4</sub> requests [3 3 0] ?

P<sub>0</sub> requests [0 2 0] ?

## Deadlock detection

if there is only a single instance of each resource, can construct a wait-for graph out of the resource allocation graph

- remove all resource nodes and collapse the edges



a deadlock has occurred if and only if there is a cycle in the wait-for graph

- to detect deadlock, need to periodically check for cycles  
can be accomplished in  $O(n^2)$  operations, where  $n$  is number of processes

19

## Deadlock detection algorithm

if multiple instances of resources, need test similar to safety algorithm

- define vector:  $Finish[k] = (Allocation[k] == 0)$
- find an  $i$  such that  $(Finish[i] == false \ \&\& \ Request[i] \leq Available)$   
if no such  $i$ , then go to STEP 4
- update:  $Available = Available + Allocation[i]$ ,  $Finish[i] = true$   
go to STEP 2
- if  $(Finish[i] == false)$  for any  $i$ , then  $P_i$  is deadlocked  
otherwise, no deadlock

	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			
$P_4$	2	1	1	1	0	0			
$P_5$	0	0	2	0	0	2			

not deadlocked

$P_0, P_2, P_3, P_1, P_4$  results in  
 $Finish[i] == true$  for all  $i$

20

## Deadlock detection algorithm (cont.)

1. define vector:  $Finish[k] = (Allocation[k] == 0)$
2. find an  $i$  such that  $(Finish[i] == false \ \&\& \ Request[i] \leq Available)$   
if no such  $i$ , then go to STEP 4
3. update:  $Available = Available + Allocation[i]$ ,  $Finish[i] = true$   
go to STEP 2
4. if  $(Finish[i] == false)$  for any  $i$ , then  $P_i$  is deadlocked  
otherwise, no deadlock

	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	1			
$P_4$	2	1	1	1	0	0			
$P_5$	0	0	2	0	0	2			

deadlock!

can reclaim  $P_0$ 's resources,  
but not enough to meet  
requests of other processes

21

## When to detect deadlock?

checking for deadlock is time consuming

- given  $n$  processes,  $m$  resource types, deadlock detection is  $O(m \cdot n^2)$

testing at every resource request would greatly slow down all requests

if deadlocks are rare or only affect a small number of processes, then may resort to testing periodically (e.g., once a day, when CPU use < 40%)

- note that if testing is sporadic, there may be many cycles in the resource graph
- will not be able to tell which of the deadlocked processes "caused" the deadlock

22

## Deadlock recovery

to recover from deadlock, must eliminate one of the necessary conditions

- this involves preempting a resource or aborting a deadlocked process

### preempting a resource

How do you select a victim?

When preempt a resource, how do you roll back the process?

How do you prevent starvation?

### aborting a process

Do you abort all deadlocked processes or one at a time?

Note: it is possible to combine the three basic approaches

prevention + avoidance + detection

allowing the use of the optimal approach for each class of resources

23