

Lecture 7:

Deadlock

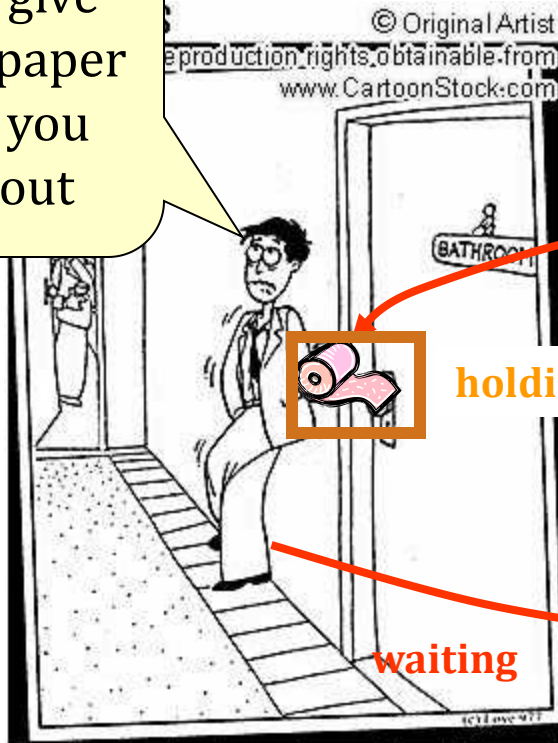
Yinqian Zhang @ 2021, Spring

copyright@Bo Tang

Deadlock

I won't give you the paper unless you come out

I won't come out unless you give me the paper



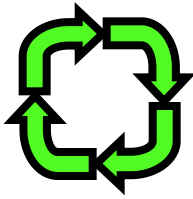
waiting

holding

waiting

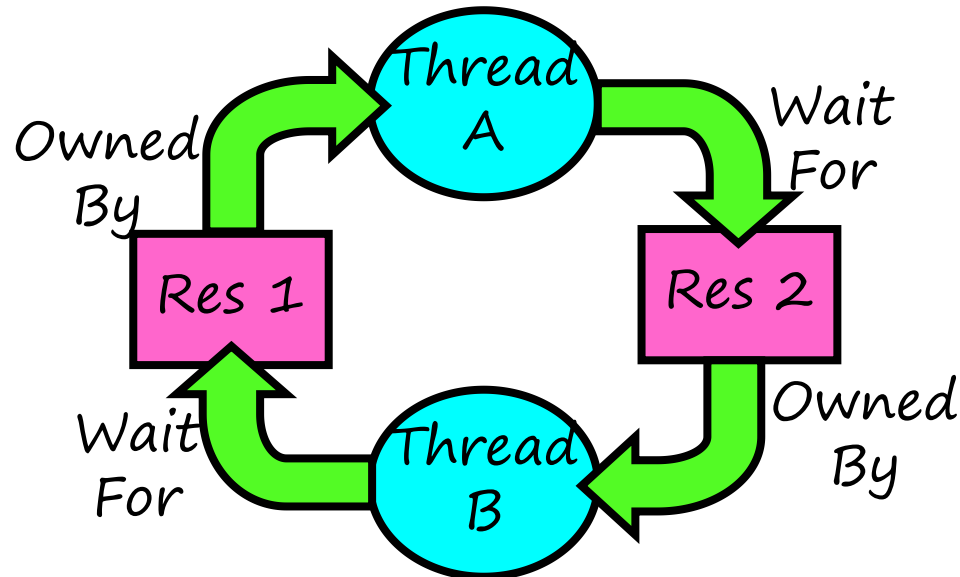


Starvation vs Deadlock



✧ Starvation vs. Deadlock

- ✧ Starvation: thread waits indefinitely
 - ✱ Low-priority thread waiting for resources constantly in use by high-priority threads
- ✧ Deadlock: circular waiting for resources
 - ✱ Thread A owns Res 1 and is waiting for Res 2
Thread B owns Res 2 and is waiting for Res 1



- ✧ Deadlock \Rightarrow Starvation but not vice versa
 - ✱ Starvation can end (but does not have to)
 - ✱ Deadlock cannot end without external intervention

Conditions for Deadlock

✧ Deadlock not always deterministic

<u>Thread A</u>	<u>Thread B</u>
<code>sem_wait(x);</code>	<code>sem_wait(y);</code>
<code>sem_wait(y);</code>	<code>sem_wait(x);</code>
<code>sem_post(y);</code>	<code>sem_wait(x);</code>
<code>sem_post(x);</code>	<code>sem_wait(y);</code>

- ✧ Deadlock will not always happen with this code
 - ✱ Have to have exactly the right timing
 - ✱ So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant...

✧ Deadlocks occur with multiple resources

- ✧ Means you cannot decompose the problem
- ✧ Cannot solve deadlock for each resource independently
- ✧ System with 2 disk drives and two threads
 - ✱ Each thread needs 2 disk drives to function
 - ✱ Each thread gets one disk and waits for another one

Four requirements for Deadlock

✧ Mutual exclusion

- ✧ Only one thread at a time can use a resource.

✧ Hold and wait

- ✧ Thread holding at least one resource is waiting to acquire additional resources held by other threads

✧ No preemption

- ✧ Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

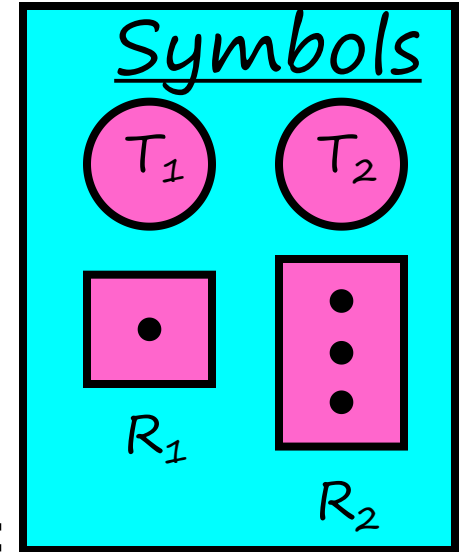
✧ Circular wait

- ✧ There exists a set $\{T_1, \dots, T_n\}$ of waiting threads
 - ✧ T_1 is waiting for a resource that is held by T_2
 - ✧ T_2 is waiting for a resource that is held by T_3
 - ✧ ...
 - ✧ T_n is waiting for a resource that is held by T_1

Resource-Allocation Graph

❖ System Model

- ❖ A set of Threads T_1, T_2, \dots, T_n
- ❖ Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
- ❖ Each resource type R_i has W_i instances
- ❖ Each thread utilizes a resource as follows:
 - ❖ Request () / Use () / Release ()



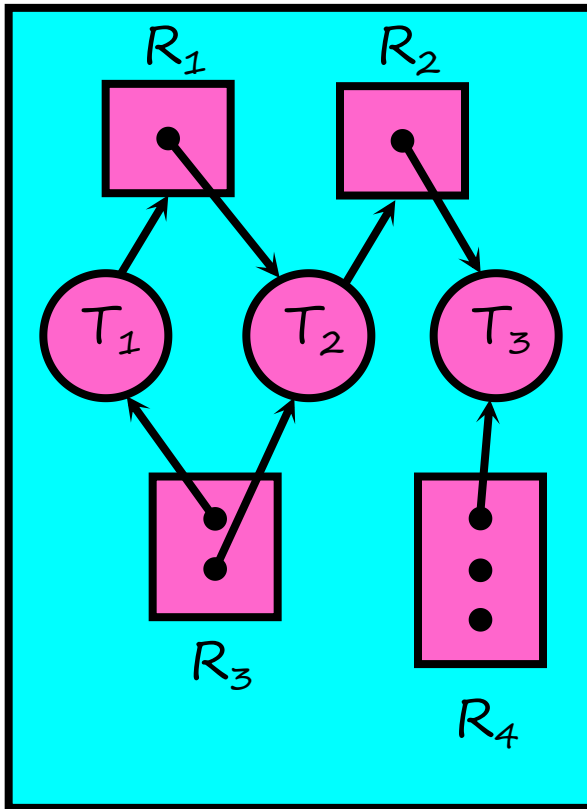
❖ Resource-Allocation Graph:

- ❖ V is partitioned into two types:
 - ❖ $T = \{T_1, T_2, \dots, T_n\}$, the set threads in the system.
 - ❖ $R = \{R_1, R_2, \dots, R_m\}$, the set of resource types in system
- ❖ request edge – directed edge $T_1 \rightarrow R_j$
- ❖ assignment edge – directed edge $R_j \rightarrow T_i$

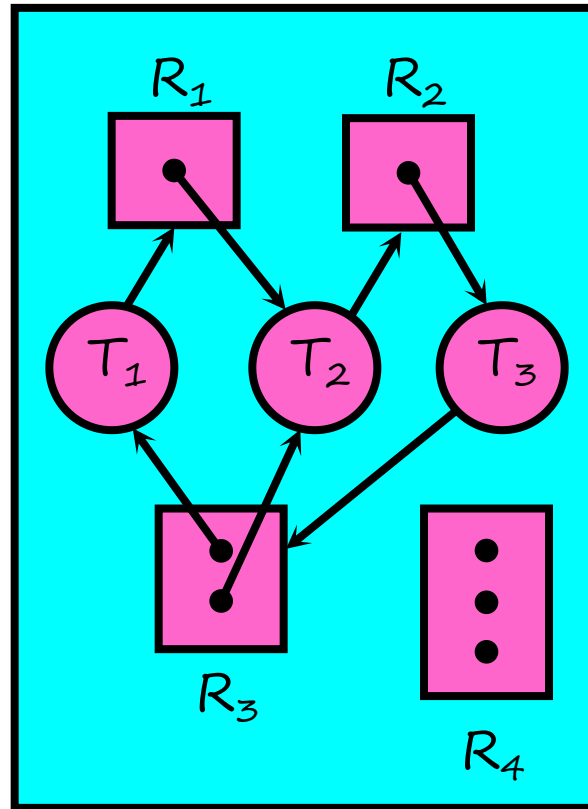
Resource Allocation Graph Examples

✧ Recall:

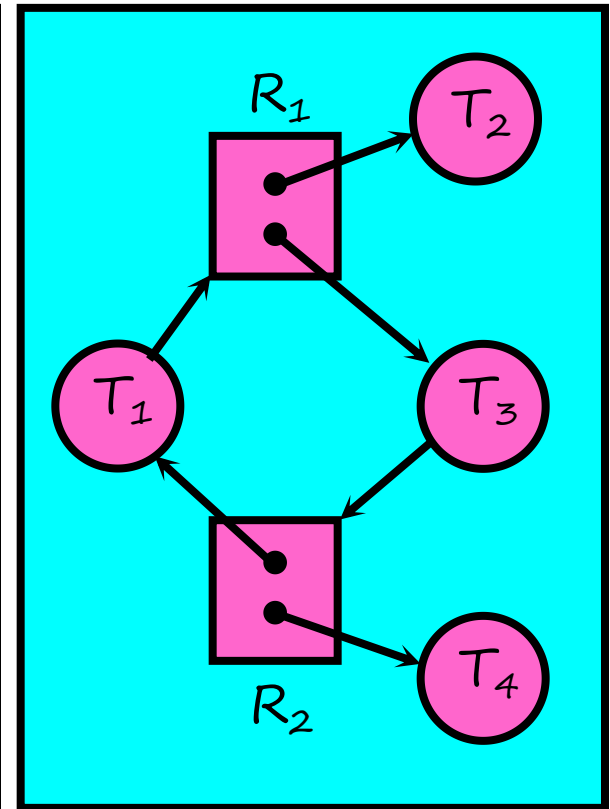
- ✧ request edge – directed edge $T_1 \rightarrow R_j$
- ✧ assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource
Allocation Graph

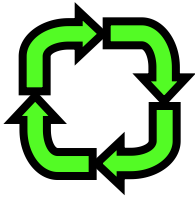


Allocation Graph
With Deadlock



Allocation Graph with
Cycle, but No Deadlock

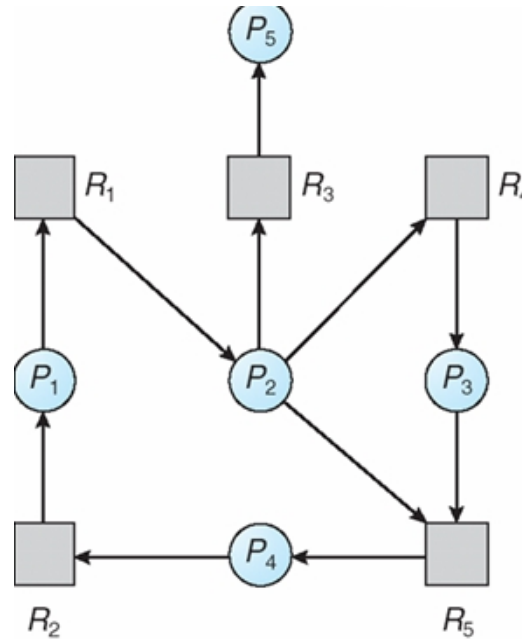
Methods for Handling Deadlocks



- ✧ Allow system to enter deadlock and then recover
 - ✧ Requires deadlock detection algorithm
 - ✧ Some technique for forcibly preempting resources and/or terminating tasks
- ✧ Ensure that system will *never* enter a deadlock
 - ✧ Need to monitor all resource acquisitions
 - ✧ Selectively deny those that *might* lead to deadlock
- ✧ Ignore the problem and pretend that deadlocks never occur in the system
 - ✧ Used by most operating systems, including UNIX

Deadlock Detection Algorithm

- ✧ Only one of each type of resource \Rightarrow look for cycles



- ✧ More than one resource of each type
 - ✧ More deadlock detection algorithm
 - ✧ Next page

Several Instances of a Resource Type

- ✧ **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 .

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively Initialize:
 - (a) $Work = Available$
 - (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] = false$; otherwise, $Finish[i] = true$
2. 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

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
 $Finish[i] = true$
go to step 2
4. If $Finish[i] == false$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked

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 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_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- ✧ Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i

Example (Cont.)

- ✧ P_2 requests an additional instance of type C

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

- ✧ State of system?
 - ✧ Can reclaim resources held by process P_0 (not deadlocked), but insufficient resources to fulfill other processes; requests
 - ✧ Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4

What to do when detect deadlock?

- ✧ Terminate thread, force it to give up resources
 - ✧ In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
 - ✧ Shoot a dining philosopher
 - ✧ But, not always possible
- ✧ Preempt resources without killing off thread
 - ✧ Take away resources from thread temporarily
 - ✧ Does not always fit with semantics of computation
- ✧ Roll back actions of deadlocked threads
 - ✧ For bridge example, make one car roll backwards (may require others behind him)
 - ✧ Common technique in databases (transactions)
 - ✧ Of course, if you restart in exactly the same way, may reenter deadlock once again
- ✧ Many operating systems use other options

Techniques for Preventing Deadlock

✧ Infinite resources

- ✧ Include enough resources so that no one ever runs out of resources. Examples:
 - ✱ Bay bridge with 12,000 lanes. Never wait!
 - ✱ Infinite disk space (not realistic yet?)

✧ No sharing of resources (totally independent threads)

- ✧ Not very realistic

✧ Do not allow waiting

- ✧ Technique used in Ethernet/some multiprocessor nets
 - ✱ Everyone speaks at once. On collision, back off and retry
- ✧ Inefficient, since have to keep retrying
 - ✱ Consider: driving to SUSTech; when hit traffic jam, suddenly you are transported back home and told to retry!

Techniques for Preventing Deadlock

- ✧ Make all threads request everything they will need at the beginning.
 - ✧ Problem: Predicting future is hard, tend to over-estimate resources. Example:
 - ✧ If need 2 chopsticks, request both at same time
 - ✧ Don not leave home until we know no one is using any intersection between home and SUSTech; only one car on the Bay Bridge at a time
- ✧ Force all threads to request resources in a particular order preventing any cyclic use of resources
 - ✧ Thus, preventing deadlock
 - ✧ Example (x.P, y.P, z.P,...)
 - ✧ Make tasks request disk, then memory, then...
 - ✧ Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Banker's Algorithm

- ✧ Multiple instances of each resource type
- ✧ 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* = 0, 1, ..., *n* - 1

2. Find an index *i* such that both:

(a) *Finish* [*i*] = *false*

(b) $Need_i \leq Work$ (i.e., for all *k*, $Need_i[k] \leq Work[k]$)

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

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*;

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><i>Need</i></u>		
	<i>A</i>	<i>B</i>	<i>C</i>
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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- ✧ 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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 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?

Thank You!