

Lecture 13

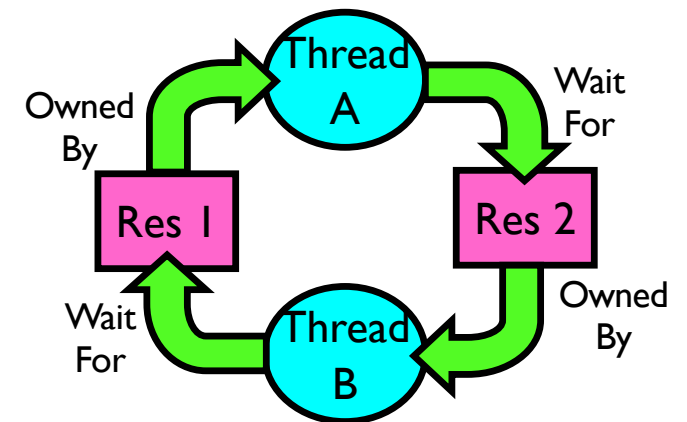
Deadlock

Prof. Yinqian Zhang

Spring 2023

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 will not always happen
 - Need the exactly right timing
 - Bugs may not exhibit during testing
- Deadlocks occur with multiple resources
 - 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

Process A

sem_wait(**x**)
sem_wait(**y**)

sem_post(**y**)
sem_post(**x**)

Process B

sem_wait(**y**)
sem_wait(**x**)

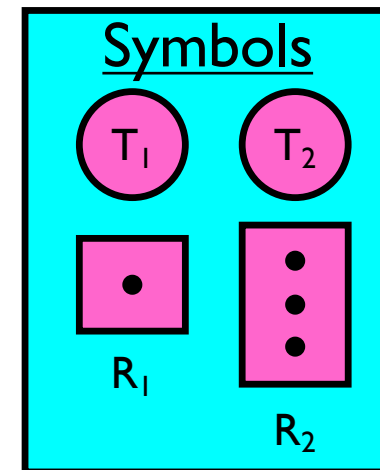
sem_post(**x**)
sem_post(**y**)

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

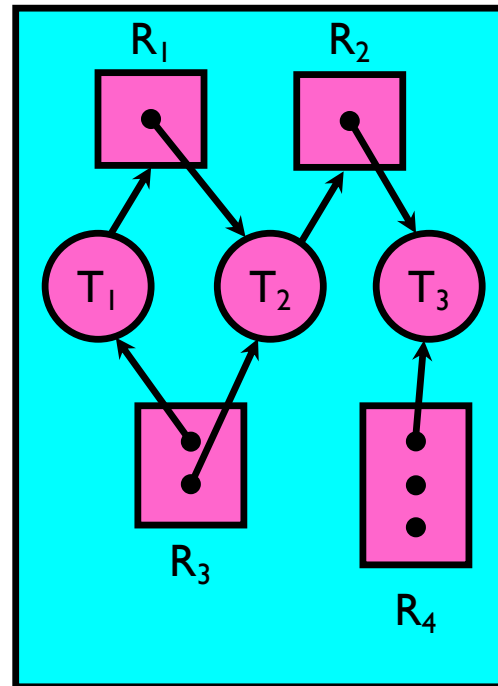
- System Model
 - A set of Threads T_1, T_2, \dots, T_n
 - Resource types R_1, R_2, \dots, R_m
CPU cycles, memory space, I/O devices
 - Each resource type R_i has W_i instances
 - Each thread utilizes a resource as follows:
 - Request() / Use() / Release()
- Resource-Allocation Graph:
 - V is partitioned 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 $T_i \rightarrow R_j$
 - assignment edge – directed edge $R_j \rightarrow T_i$



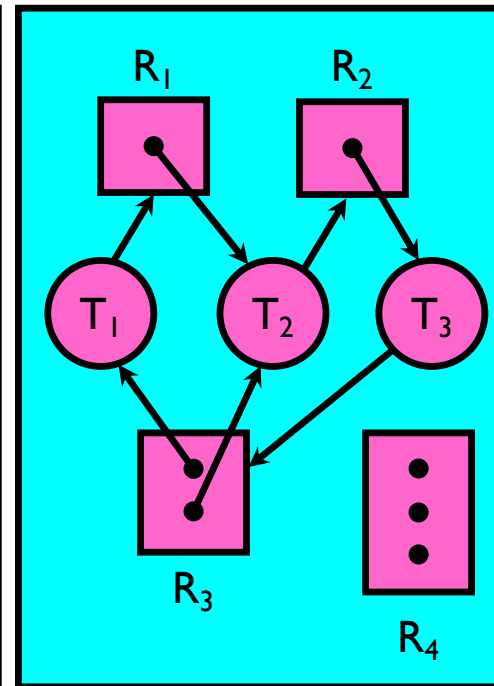
Resource Allocation Graph Examples

- Recall:

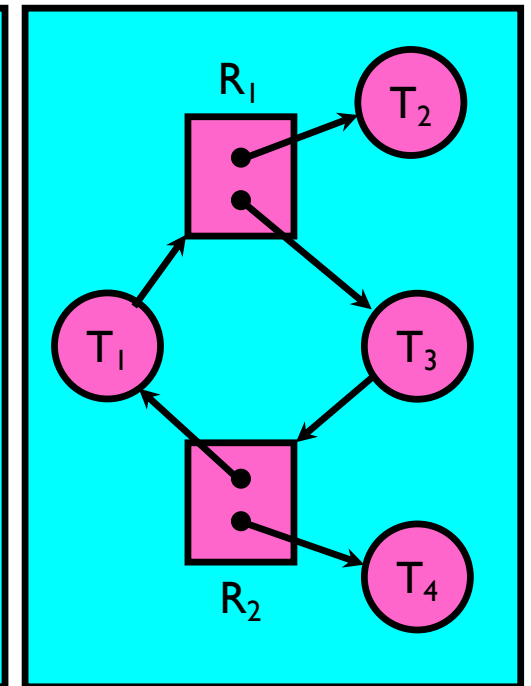
- request edge - directed edge $T_i \rightarrow R_j$
- assignment edge - directed edge $R_j \rightarrow T_i$



Simple Resource
Allocation Graph



Allocation Graph
With Deadlock



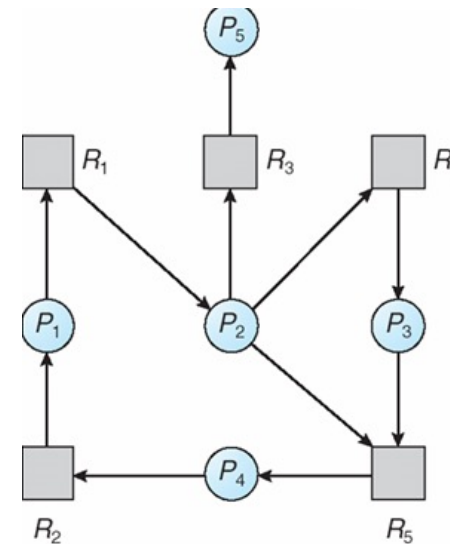
Allocation Graph with
Cycle, but No Deadlock

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
 - **Deadlock detection**
 - Requires deadlock detection algorithm
 - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will never enter a deadlock
 - **Deadlock prevention**
 - 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 with Resource Allocation Graphs

- Only one of each type of resource \Rightarrow look for cycles
- More than one resource of each type
 - More complex deadlock detection algorithm
 - Next page



Several Instances per 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 $\text{Request}[i,j] = k$, 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
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 $\text{Finish}[i] = \text{true}$ 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 if Deadlock Detected?

- Terminate process, force it to give up resources
 - Shoot a dining philosopher !?
 - But, not always possible
- Preempt resources without killing off process
 - Take away resources from process temporarily
 - Does not always fit with semantics of computation
- Roll back actions of deadlocked process
 - Common technique in databases (transactions)
 - Of course, deadlock may happen once again

Deadlock Prevention

- Try to ensure at least one of the conditions cannot hold to prevent deadlock
 - Remove "Mutual Exclusion": not possible for non-sharable resources
 - Remove "Hold and Wait" – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
 - Low resource utilization; starvation possible

Deadlock Prevention

- Try to ensure at least one of the conditions cannot hold to prevent deadlock
 - Remove "Preemption"
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention

- Try to ensure at least one of the conditions cannot hold to prevent deadlock
 - Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
 - $R = \{R_1, R_2, \dots, R_m\}$
 - One to one function $F: R \rightarrow N$
 - If a process request a resource R_i , it can request another resource R_j if and only if $F(R_i) < F(R_j)$
 - Or, it must first release all resource R_i such that $F(R_i) \geq F(R_j)$

Deadlock Avoidance

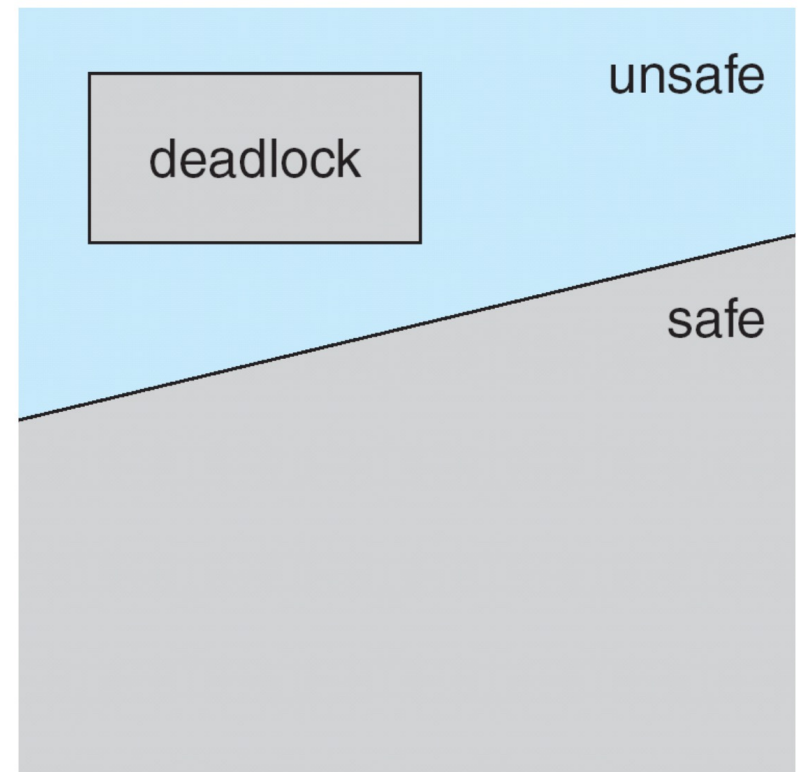
- Requires that the system has some additional a priori information available
 - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure 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

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 sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that **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 what 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

Safe, Unsafe, Deadlock State

- If a system is in safe state => no circular wait => no deadlocks
- If a system is in unsafe state => possibility of deadlock
- Deadlock avoidance => ensure that a system will never enter an unsafe state.



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

Banker's Algorithm (Cont'd)

- 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

$$\text{Need } [i,j] = \text{Max}[i,j] - \text{Allocation } [i,j]$$

Banker's Algorithm: Safety Algorithm

1. Let **Work** and **Finish** be vectors of length m and n , respectively.

Initialize:

Work = **Available**

Finish $[i]$ = false for $i = 0, 1, \dots, n-1$

2. Find an index i such that both:

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

(b) **Need** $_i \leq \mathbf{Work}$ (i.e., for all k , **Need** $_i[k] \leq \mathbf{Work}[k]$)

If no such i exists, go to step 4

3. **Work** = **Work** + **Allocation** $_i$

Finish $[i]$ = true

go to step 2

4. If **Finish** $[i] == \text{true}$ 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'd)

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

	<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	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	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, $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$

	<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?