CSCI 4730/6730 OS
(Chap #8 Deadlocks – Part 3)

In Kee Kim

Department of Computer Science

University of Georgia

Announcement

- No class on Thursday (10/21)
  - Please work on PA #2
- PA #2 deadline is 10/25

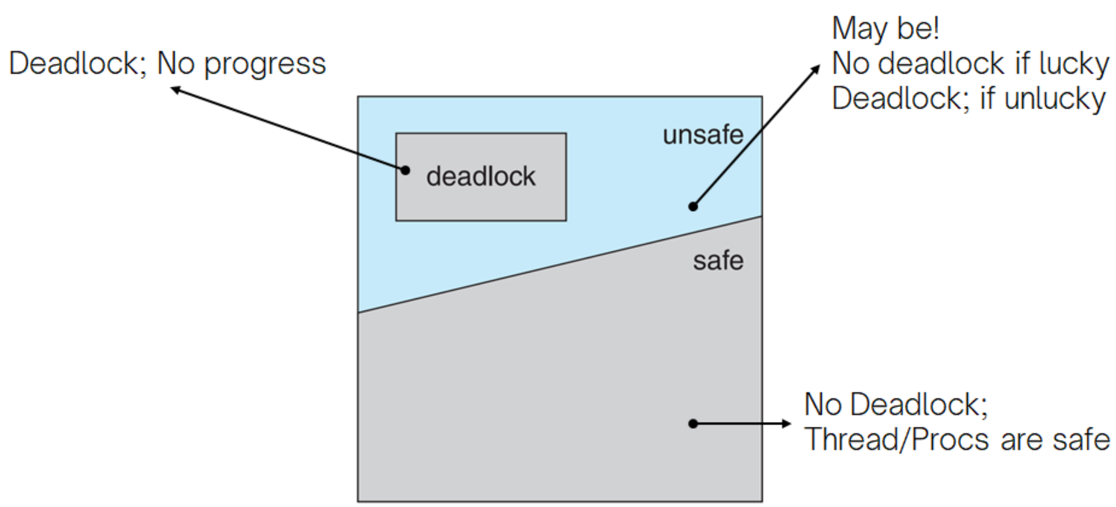
PA #2 questions?

Recap: Deadlock Handling in OS

- ❑ Deadlock Detection and Recovery
- ❑ Deadlock Prevention
- ❑ $\boxed{\text{Deadlock Avoidance}}$
- ❑ Deadlock Ignorance

Recap: Deadlock Avoidance

- If we or OS have prior knowledge of how resources will be requested, it is possible to determine if threads are entering an "**unsafe**" state.



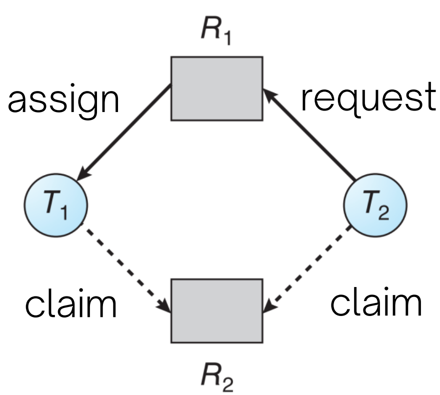
NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks

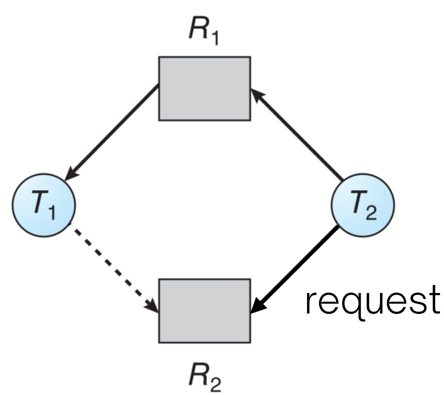
Recap: Avoidance Algorithms

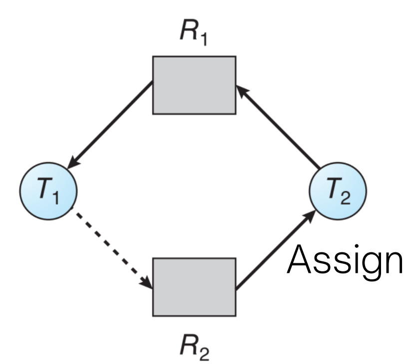
- Single instance of a resource type
  - e.g., a file, a printer, a scanner…
  - Use **Resource-allocation graph**
- Multiple instances of a resource type
  - e.g., CPU, IO, MEM, Network BW…
  - Use **Banker's algorithm**

Recap: Claim Edge

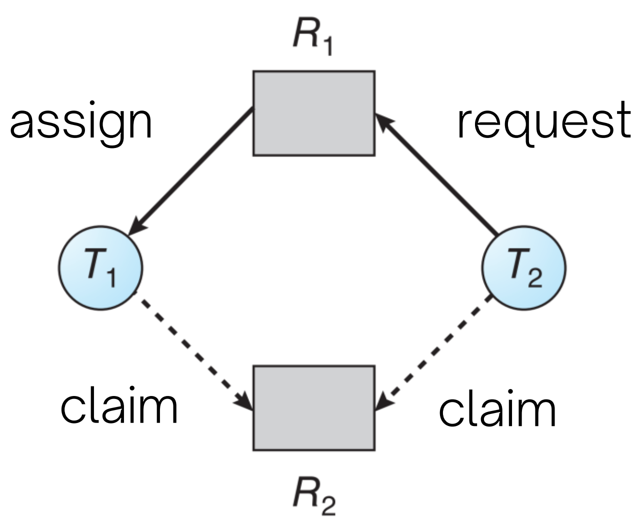
- **Claim edge** $T_i \dashrightarrow R_j$ indicated that thread T_i may request resource R_j; represented by a dashed line
- Claim edge ($T_i \dashrightarrow R_j$) converts to request edge ($T_i \longrightarrow R_j$) when the thread T_i requests the resource R_j
- Request edge ($T_i \longrightarrow R_j$) converted to an assignment edge ($T_i \longleftarrow R_j$) when the resource R_j is allocated to the thread T_i
- Resources must be claimed ***a priori*** in the system

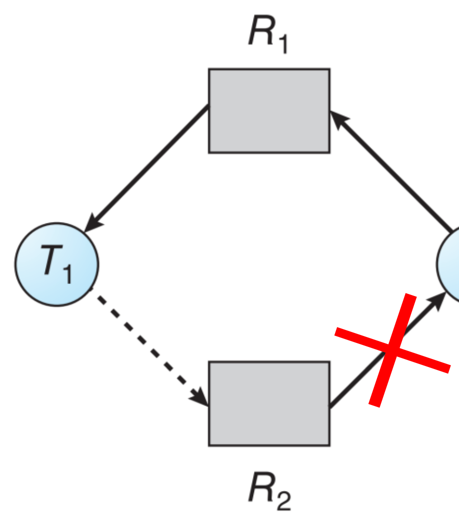






Recap: Avoidance with Resource-Allocation Graph





- Suppose that thread T_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does ***not result in the formation of a cycle*** in the resource allocation graph

Recap: Banker's Algorithm

- Use Banker's algorithm
  - **when multiple instances exist in a resource type**
- Assume that:
  - A resource can have multiple instances
    - Common in real-world systems. e.g., CPU, memory, disk, network
  - OS has ***N*** threads, ***M*** resource types
- Initially, each thread must declare the maximum number of resources it will need. Ugh – ☹
- Calculate a safe sequence if possible.

Recap: Banker's Algorithm

- Simple Example
  - A system with **12** CPU resources. Each resource is used exclusively by a thread. The current state looks like this:

Thread	Max Needs	Allocated	Current Needs
T0	10	5	**?**
T1	4	2	**?**
T2	7	3	**?**

$$\sum \boldsymbol{Allocated} = \mathbf{10}$$

In this example,
What is workable sequence?

<T1, T2, T0>

Suppose T2 requests and is given one more resource. What happens then?

Banker's Algorithm Data Structures

- ***N***: # of threads, ***M***: # of resource types
- **`Available[M]`**
  - # of available resource instances for each resource type
  - e.g. `Available[j] == k` means R_j has k instances
- **`Max[N][M]`**
  - maximum demand of each threads
  - e.g. `max[i][j] == k` means T_i wants k R_j's

Banker's Algorithm Data Structures

- **`Allocation[N][M]`**
  - # of resource instances allocated to each thread
  - e.g. `Allocation[i][j] == k`
    means T_i currently has k R_j's
- **`Need[N][M]`**
  - no. of resource instances still needed by each thread
  - e.g. `Need[i][j] == k`
    means T_i still needs k R_j' s
  - `Need[i][j] == Max[i][j] - Allocation[i][j]`

Banker's Algorithm Data Structures

- **`Work[M]`**
  - no. of resource instances available for work (by all processes)
  - e.g. `Work[j] == k` means K R_j's are available
- **`Finish[N]`**
  - record of finished processes
  - e.g. P_i is finished if `Finish[i] == true`

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

   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.
   Otherwise…

Safety Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			

What are the total amount of A, B, and C?

Safety Example – Need Matrix

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			

Is the system in a *safe state?*

Yes <T_1, T_3, T_4, T_2, T_0>, <T_1, T_3, T_4, T_0, T_2> are a safe sequence.

Now T1 Requests More

- $\texttt{Request}_1 = (1,0,2)$
- This request will be granted or not?
  - $\texttt{Request}_1 \leq \texttt{Available}$ – that is, $(1,0,2) \leq (3,3,2)$
  - Generate new state and test for safety

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	**2**	**3**	**0**	7	4	3
T_1	**3**	**0**	**2**	3	2	2						
T_2	3	0	2	9	0	2						
T_3	2	1	1	2	2	2						
T_4	0	0	2	4	3	3						

What is the seq for safe state?

Now T1 Requests More

- **`Request`$_1$ `= (1,0,2)`**
- This request will be granted or not?
  - **`Request`$_1$ $\leq$ `Available`** – that is, **`(1,0,2)`$\leq$`(3,3,2)`**
  - Generate new state and test for safety

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
T_0	0	1	0	7	5	3	**2**	**3**	**0**	7	4	3
T_1	**3**	**0**	**2**	3	2	2						
T_2	3	0	2	9	0	2						
T_3	2	1	1	2	2	2						
T_4	0	0	2	4	3	3						

Safe State = <T_1, T_3, T_4, T_0, T_2>

Then…

- From the last state of the example
  - Can request for (3,3,0) by T_4 be granted?
  - Can request for (0,2,0) by T_0 be granted?

Exercise for Banker's Algorithm

	Allocation		Max		Available		Need	
	A	B	A	B	A	B	A	B
T_0	2	2	5	4	2	1	3	2
T_1	3	3	8	7			5	4
T_2	1	1	2	2			1	1

The OS is in a *safe state:* <T2, T0, T1>

T1 requests: [1,0]

	Allocation		Max		Available		Need	
	A	B	A	B	A	B	A	B
T_0	2	2	5	4	1	1	3	2
T_1	**4**	**3**	8	7			4	4
T_2	1	1	2	2			1	1

Deadlock Detection & Recovery

- If there are no prevention or avoidance mechanisms in place, then deadlock ***may*** occur.
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Deadlock Detection

Deadlock Detection Algorithms

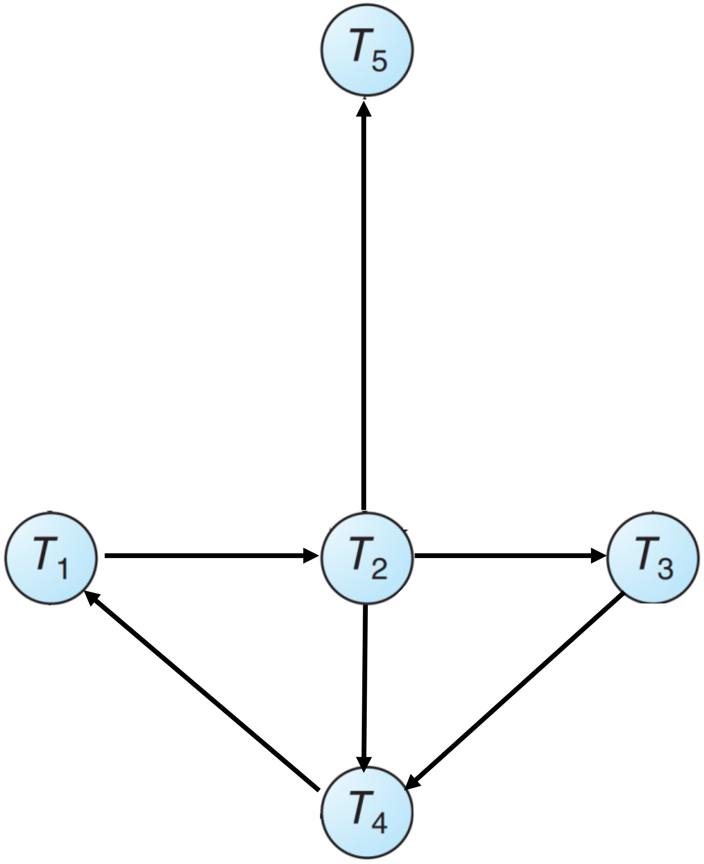
- Single Instance
  (Wait-for Graph)
- Multiple Instances
  (Banker's Algorithm)

Wait-for Graph

- Assume that each resource has only one instance.
- Create a **wait-for graph** by removing the resource types nodes from a resource allocation graph.
- Then detect a cycle
  - Deadlock exists *if and only if* the wait-for graph contains a cycle.

Wait-for Graph

Simply, remove resources,
make connection between threads



If the graph has no cycles, no deadlock exists
If the graph has a cycle, deadlock might exist

Banker's Algorithm for Deadlock Detection

- Multiple instances of a resource type
  - ***n***: # of threads, ***m***: # of resource types
  - **Available**: A vector of length ***m*** indicates the number of available resources of each type
  - **Allocation**: An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process
  - **Request**: An ***n* x *m*** matrix indicates the current request of each process. If ***`Request[i][j]=k`***, then thread T_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 ***Work*** = ***Available***. For $i = 0, 1, \ldots, n-1$, if $\boldsymbol{Allocation_i} \neq 0$, then ***Finish***[$i$] = ***false***. Otherwise, ***Finish***[i] = ***true***.

2. Find an index i such that both

   a. ***Finish***[i] == ***false***

   b. $\boldsymbol{Request_i} \leq \boldsymbol{Work}$

   If no such i exists, go to step 4.

3. $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation_i}$
   Finish[i] = ***true***
   Go to step 2.

4. If ***Finish***[i] == ***false*** for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***[i] == ***false***, then thread T_i is deadlocked.

Detection Algorithm

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available***. For $i = 0, 1, ..., n-1$, 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, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***$[i]$ == ***false***, then thread T_i is deadlocked.

Safety Algorithm

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

   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.

Example

	Allocation			*Request*			*Available*		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

What are the total amount of A, B, and C?

Does Deadlock Exist?

The system is *not* in a deadlocked state since Sequence <T_0, T_2, T_3, T_1, T_4> or <T_0, T_2, T_3, T_4, T_1> will result in *Finish[i]* = *true* for all *i*

Example

- Change T_2 to request <0, 0, 1>

	Allocation			*Request*			*Available*		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	**1**			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

Deadlock exists.

Deadlock Detection Algorithm is Expensive

- Resource-Allocation Graph – ***O(n²)***, where ***n*** = |T| + |R|.
- Wait-for Graph – ***O(n²)***, where ***n*** = |T|.
- Banker's Algorithm ***O(m x n²)***, m: # resources, n: # Threads

Deadlock Detection Algorithm is Expensive

- Q: When should we run this expensive algorithm?
  - Just before granting a resource, check if granting it would lead to a cycle?
    - Each request is then ***O(n²) ~ O(m x n²)***
  - Whenever a resource request cannot be filled?
    - Each failed request is ***O(n²) ~ O(m x n²)***
  - On a regular schedule (hourly or …)?
    - May take a long time to detect deadlock
  - When CPU utilization drops below some threshold?
    - May take a long time to detect deadlock

Deadlock Detection Algorithm is Expensive

- In general, what does modern OS do?
  - **Do nothing!**
  - Leave it to programmers and/or applications

- Probably, **Ignore the deadlock** may not be a bad idea
  - Most operating systems, including UNIX, Linux
  - Cheap solution
  - Infrequent, manual reboots may be acceptable

Recovery from Deadlock

- Process Termination
- Resource Preemption

Process Termination

- Abort all deadlocked processes
  - Simple, but very expensive
  - Users will lose all the previous executions
- Abort one process at a time until the deadlock cycle is eliminated
  - Kill one process
  - Run deadlock detection algorithm
  - Repeating process
  - Detection algorithm is expensive (overhead)

Process Termination

- In which order should we choose to abort?
    1. Priority of the process
    2. How long process has computed, and how much longer to completion
    3. Resources the process has used
    4. How many processes will need to be terminated
    5. Is process interactive or batch?

Resource Preemption

- Preempt some resources from processes (thread) and give these resources to other processes until the deadlock is eliminated

Resource Preemption

- Discussion Issues
  - **Selecting a victim**
    - Which resources from which process?
    - Select minimum cost option
  - **Rollback**
    - Return to some safe state, restart process for that state
    - DBMS Transactions
  - **Starvation**
    - Same process may always be picked as victim, include number of rollback in cost factor

End of Chapter 8