CENG 2034 - Operating Systems

Week 12: Deadlocks

Burak Ekici

June 2, 2023

Outline

System Model

- System consists of resources

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ – CPU cycles, memory space, I/O devices

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ – CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ – CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ – CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:
  - request

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ – CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:
  - request
  - use

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$ – CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock with Semaphores

- Data:

Deadlock with Semaphores

- Data:
  - A semaphore S_1 initialized to 1

Deadlock with Semaphores

- Data:
  - A semaphore S_1 initialized to 1
  - A semaphore S_2 initialized to 1

Deadlock with Semaphores

- Data:
  - A semaphore S_1 initialized to 1
  - A semaphore S_2 initialized to 1
- Two threads T_1 and T_2

  T_1:

  wait(s_1)
  wait(s_2)

  T_2:

  wait(s_2)
  wait(s_1)

Outline

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

1. Mutual exclusion: only one thread at a time can use a resource

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

1. Mutual exclusion: only one thread at a time can use a resource
2. Hold and wait: a thread holding at least one resource is waiting to acquire additional resources held by others

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

1. Mutual exclusion: only one thread at a time can use a resource
2. Hold and wait: a thread holding at least one resource is waiting to acquire additional resources held by others
3. No preemption: a resource can be released only voluntarily by the holding thread after the task completion

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

1. Mutual exclusion: only one thread at a time can use a resource
2. Hold and wait: a thread holding at least one resource is waiting to acquire additional resources held by others
3. No preemption: a resource can be released only voluntarily by the holding thread after the task completion
4. Circular wait: there exists a set $\{T_0, T_1, \dots, T_n\}$ of waiting threads such that

   T_0 is waiting for a resource that is held by T_1
   T_1 is waiting for a resource that is held by T_2
   $\dots$
   T_{n-1} is waiting for a resource that is held by T_n,
   and T_n is waiting for a resource that is held by T_0

Resource-Allocation Graph

A set of vertices V and a set of edges E

Resource-Allocation Graph

A set of vertices V and a set of edges E

- V is partitioned into two types:

$T = \{T_1, T_2, \ldots, T_n\}$	the set consisting of all the threads in the system
$R = \{R_1, R_2, \ldots, R_m\}$	the set consisting of all resource types in the system

Resource-Allocation Graph

A set of vertices V and a set of edges E

- V is partitioned into two types:

$T = \{T_1, T_2, \ldots, T_n\}$	the set consisting of all the threads in the system
$R = \{R_1, R_2, \ldots, R_m\}$	the set consisting of all resource types in the system

- Edges:

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

Example (Resource Allocation Graph)

- One instance of R_1

Example (Resource Allocation Graph)

- One instance of R_1
- Two instances of R_2

Example (Resource Allocation Graph)

- One instance of R_1
- Two instances of R_2
- One instance of R_3

Example (Resource Allocation Graph)

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4

Example (Resource Allocation Graph)

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1

Example (Resource Allocation Graph)

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3

Example (Resource Allocation Graph)

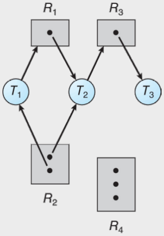
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3

Example (Resource Allocation Graph)

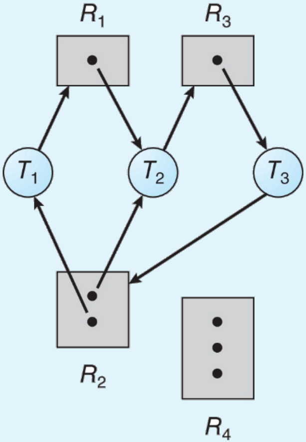
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3

Example (Resource Allocation Graph)

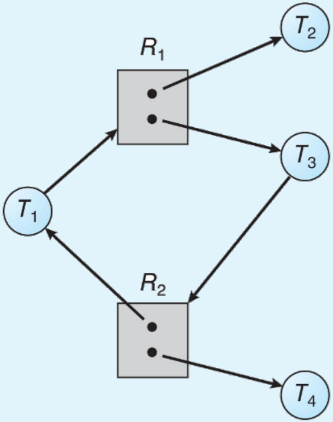
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3



Resource-Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock

Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒

Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock

Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Outline

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system

Outline

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a thread requests a resource, it does not hold any other resources

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it
  - Low resource utilization; starvation possible

Deadlock Prevention (cont'd)

- No Preemption:

Deadlock Prevention (cont'd)

- No 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

Deadlock Prevention (cont'd)

- No 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 thread is waiting

Deadlock Prevention (cont'd)

- No 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 thread is waiting
  - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention (cont'd)

- No 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 thread is waiting
  - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- Circular Wait:

Deadlock Prevention (cont'd)

- No 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 thread is waiting
  - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- Circular Wait:
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration

Circular Wait

- Invalidating the circular wait condition is most common

Circular Wait

- Invalidating the circular wait condition is most common
- Simply assign each resource (i.e., mutex locks) a unique number

Circular Wait

- Invalidating the circular wait condition is most common
- Simply assign each resource (i.e., mutex locks) a unique number
- Resources must be acquired in order

Outline

Deadlock Avoidance

Requires that the system has some additional a priori information available

Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each thread declare the maximum number of resources of each type that it may need

Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each thread 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

Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each thread 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 thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state

Safe State

- When a thread 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 $< T_1, T_2, \ldots, T_n >$ of ALL the threads in the systems such that

Safe State

- When a thread 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 $< T_1, T_2, \ldots, T_n >$ of ALL the threads in the systems such that
  - for each T_i, the resources that T_i can request $<=$ currently available resources $+$ resources held by all the T_j, with $j < i$

Safe State

- When a thread 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 $< T_1, T_2, \ldots, T_n >$ of ALL the threads in the systems such that
  - for each T_i, the resources that T_i can request $<=$ currently available resources $+$ resources held by all the T_j, with $j < i$
- That is:

Safe State

- When a thread 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 $< T_1, T_2, \ldots, T_n >$ of ALL the threads in the systems such that
  - for each T_i, the resources that T_i can request $<=$ currently available resources $+$ resources held by all the T_j, with $j < i$
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished

Safe State

- When a thread 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 $< T_1, T_2, \ldots, T_n >$ of ALL the threads in the systems such that
  - for each T_i, the resources that T_i can request $<=$ currently available resources $+$ resources held by all the T_j, with $j < i$
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate

Safe State

- When a thread 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 $< T_1, T_2, \ldots, T_n >$ of ALL the threads in the systems such that
  - for each T_i, the resources that T_i can request $<=$ currently available resources $+$ resources held by all the T_j, with $j < i$
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on ...

Basic Facts

- If a system is in safe state ⇒ no deadlocks

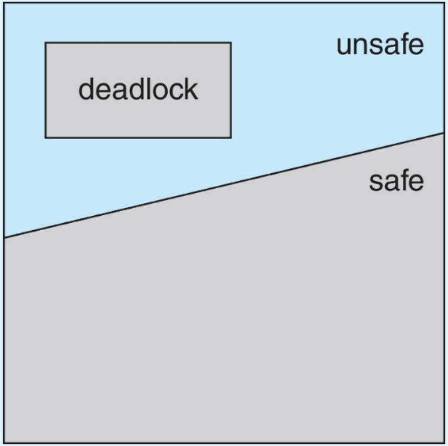
Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state

Safe, Unsafe, Deadlock State



Avoidance Algorithms

- Single instance of a resource type – Use a resource-allocation graph

Avoidance Algorithms

- Single instance of a resource type – Use a resource-allocation graph
- Multiple instances of a resource type – Use the Banker's Algorithm

Resource-Allocation Graph Scheme

- Claim edge $T_i \dashrightarrow R_j$ indicates that thread T_j may request resource R_j; represented by a dashed line

Resource-Allocation Graph Scheme

- Claim edge $T_i \dashrightarrow R_j$ indicates that thread T_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource

Resource-Allocation Graph Scheme

- Claim edge $T_i \dashrightarrow R_j$ indicates that thread T_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread

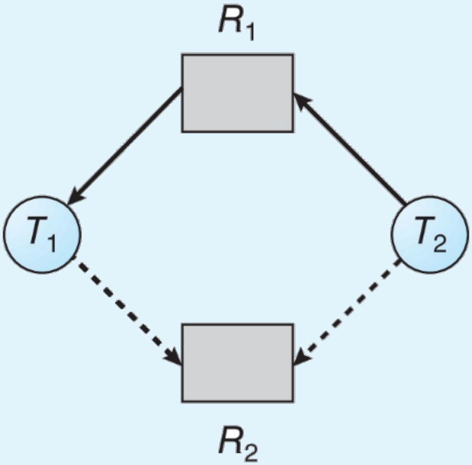
Resource-Allocation Graph Scheme

- Claim edge $T_i \dashrightarrow R_j$ indicates that thread T_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge

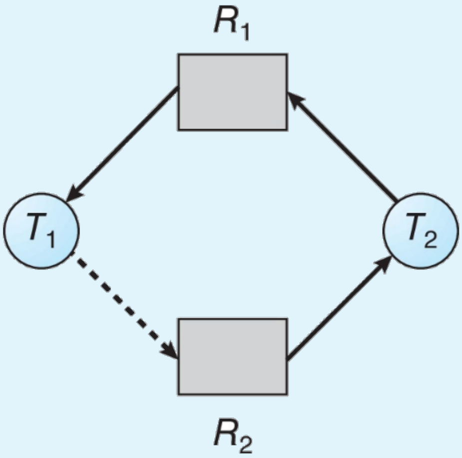
Resource-Allocation Graph Scheme

- Claim edge $T_i \dashrightarrow R_j$ indicates that thread T_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that thread T_i requests a resource R_j

Resource-Allocation Graph Algorithm

- 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

Banker's Algorithm

- Multiple instances of resources

Banker’s Algorithm

- Multiple instances of resources
- Each thread must a priori claim maximum use

Banker’s Algorithm

- Multiple instances of resources
- Each thread must a priori claim maximum use
- When a thread requests a resource, it may have to wait

Banker's Algorithm

- Multiple instances of resources
- Each thread must a priori claim maximum use
- When a thread requests a resource, it may have to wait
- When a thread 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 threads, and m = number of resources types.

Data Structures for the Banker's Algorithm

Let n = number of threads, 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

Data Structures for the Banker's Algorithm

Let n = number of threads, 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 $\mathrm{Max}[i, j] = k$, then thread T_i may request at most k instances of resource type R_j

Data Structures for the Banker's Algorithm

Let n = number of threads, 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 $\texttt{Max}[i, j] = k$, then thread T_i may request at most k instances of resource type R_j
- Allocation: $n \times m$ matrix. If $\texttt{Allocation}[i, j] = k$ then T_i is currently allocated k instances of R_j

Data Structures for the Banker's Algorithm

Let n = number of threads, 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 $\mathtt{Max}[i, j] = k$, then thread T_i may request at most k instances of resource type R_j
- Allocation: $n \times m$ matrix. If $\mathtt{Allocation}[i, j] = k$ then T_i is currently allocated k instances of R_j
- Need: $n \times m$ matrix. If $\mathtt{Need}[i, j] = k$, then T_i may need k more instances of R_j to complete its task

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

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

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 `i` such that both:

   `Finish[i]` = false

   $\text{Need}_i \leqslant$ `Work`

   If no such *i* exists, go to step 4

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 `i` such that both:

   `Finish[i]` = false

   $\text{Need}_i \leqslant$ `Work`

   If no such *i* exists, go to step 4

3. `Work` = `Work` + $\texttt{Allocation}_i$

   `Finish[i]` = true

   go to step 2

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 `i` such that both:
   - `Finish[i]` = false
   - $\texttt{Need}_i$ ⩽ `Work`

   If no such *i* exists, go to step 4
3. - `Work` = `Work` + $\texttt{Allocation}_i$
   - `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 Thread T_i

$\texttt{Request}_i$ = request vector for thread T_i. If $\texttt{Request}_i[j] = k$ then thread T_i wants k instances of resource type R_j

Resource-Request Algorithm for Thread T_i

$\texttt{Request}_i$ = request vector for thread T_i. If $\texttt{Request}_i[j] = k$ then thread T_i wants k instances of resource type R_j

1. If $\texttt{Request}_i \leqslant \text{Need}_i$ go to step 2. Otherwise, raise an error, as thread has exceeded its maximum claim

Resource-Request Algorithm for Thread T_i

$\texttt{Request}_i$ = request vector for thread T_i. If $\texttt{Request}_i[j] = k$ then thread T_i wants k instances of resource type R_j

1. If $\texttt{Request}_i \leqslant \texttt{Need}_i$ go to step 2. Otherwise, raise an error, as thread has exceeded its maximum claim
2. If $\texttt{Request}_i \leqslant \texttt{Available}$, go to step 3. Otherwise T_i must wait, since resources are not available

Resource-Request Algorithm for Thread T_i

`Request`$_i$ = request vector for thread T_i. If `Request`$_i[j] = k$ then thread T_i wants k instances of resource type R_j

1. If `Request`$_i \leqslant$ Need$_i$ go to step 2. Otherwise, raise an error, as thread has exceeded its maximum claim
2. If `Request`$_i \leqslant$ `Available`, go to step 3. Otherwise T_i must wait, since resources are not available
3. Pretend to allocate requested resources to T_i by modifying the state as follows:

$$\begin{aligned}
\texttt{Available} &= \texttt{Available} - \texttt{Request}_i; \\
\texttt{Allocation}_i &= \texttt{Allocation}_i + \texttt{Request}_i; \\
\texttt{Need}_i &= \texttt{Need}_i - \texttt{Request}_i;
\end{aligned}$$

Resource-Request Algorithm for Thread T_i

$\texttt{Request}_i$ = request vector for thread T_i. If $\texttt{Request}_i[j] = k$ then thread T_i wants k instances of resource type R_j

1. If $\texttt{Request}_i \leqslant \text{Need}_i$ go to step 2. Otherwise, raise an error, as thread has exceeded its maximum claim
2. If $\texttt{Request}_i \leqslant \texttt{Available}$, go to step 3. Otherwise T_i must wait, since resources are not available
3. Pretend to allocate requested resources to T_i by modifying the state as follows:

$$\begin{aligned}
\texttt{Available} &= \texttt{Available} - \texttt{Request}_i; \\
\texttt{Allocation}_i &= \texttt{Allocation}_i + \texttt{Request}_i; \\
\text{Need}_i &= \text{Need}_i - \texttt{Request}_i;
\end{aligned}$$

- If safe $\Rightarrow$ the resources are allocated to T_i

Resource-Request Algorithm for Thread T_i

$\texttt{Request}_i$ = request vector for thread T_i. If $\texttt{Request}_i[j] = k$ then thread T_i wants k instances of resource type R_j

1. If $\texttt{Request}_i \leqslant \texttt{Need}_i$ go to step 2. Otherwise, raise an error, as thread has exceeded its maximum claim
2. If $\texttt{Request}_i \leqslant \texttt{Available}$, go to step 3. Otherwise T_i must wait, since resources are not available
3. Pretend to allocate requested resources to T_i by modifying the state as follows:

$$\begin{aligned}\texttt{Available} &= \texttt{Available} - \texttt{Request}_i;\\ \texttt{Allocation}_i &= \texttt{Allocation}_i + \texttt{Request}_i;\\ \texttt{Need}_i &= \texttt{Need}_i - \texttt{Request}_i;\end{aligned}$$

- If safe $\Rightarrow$ the resources are allocated to T_i
- If unsafe $\Rightarrow$ T_i must wait, and the old resource-allocation state is restored

Example (Banker's Algorithm)

- 5 threads T_0 through T_4;

Example (Banker's Algorithm)

- 5 threads T_0 through T_4;
- 3 resource types: A (10 instances) B (5 instances) C (7 instances)

Example (Banker's Algorithm)

- 5 threads T_0 through T_4;
- 3 resource types: A (10 instances) B (5 instances) C (7 instances)
- Current state snapshot:

	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	

Example (Banker's Algorithm (cont'd))

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

Example (Banker's Algorithm (cont'd))

- The content of the matrix Need is defined to be `Max` - `Allocation`
- 3 resource types: A (10 instances) B (5 instances) C (7 instances)

Example (Banker's Algorithm (cont'd))

- The content of the matrix Need is defined to be `Max - Allocation`
- 3 resource types: A (10 instances) B (5 instances) C (7 instances)
- Current-state snapshot:

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

Example (Banker's Algorithm (cont'd))

- The content of the matrix `Need` is defined to be `Max - Allocation`
- 3 resource types: A (10 instances) B (5 instances) C (7 instances)
- Current-state snapshot:

	Need
	$A\ B\ C$
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

- The system is in a safe state since the sequence

$$< T_1, T_3, T_4, T_2, T_0 >$$

satisfies safety criteria

Outline

Deadlock Detection

- Allow system to enter deadlock state

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain wait-for graph

Single Instance of Each Resource Type

- Maintain wait-for graph
    - Nodes are threads

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are threads
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are threads
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j
- Periodically invoke an algorithm that searches for a cycle in the graph

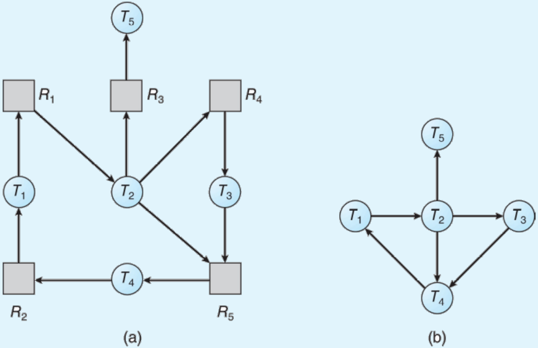
Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are threads
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j
- Periodically invoke an algorithm that searches for a cycle in the graph
- If there is a cycle, there exists a deadlock

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are threads
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j
- Periodically invoke an algorithm that searches for a cycle in the graph
- If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- `Available`: A vector of length m indicates the number of available resources of each type

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 thread

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 thread
- `Request`: An $n \times m$ matrix indicates the current request of each thread. If $\texttt{Request}[i][j] = k$, 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`	
`Finish[i]`	=	`false`	if `Allocation` $\neq 0$ for $i = 1, 2, \dots n$
`Finish[i]`	=	`true`	otherwise

Detection Algorithm

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

`Work`	=	`Available`	
`Finish[i]`	=	`false`	if `Allocation` $\neq 0$ for $i = 1, 2, \ldots n$
`Finish[i]`	=	`true`	otherwise

2. Find an index `i` such that both

`Finish[i]`	=	`false`
$\texttt{Request}_i$	$\leqslant$	`Work`

   Go to step 4 if no such `i` exists

Detection Algorithm

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

   `Work` = `Available`
   `Finish[i]` = `false` if `Allocation` $\neq 0$ for $i = 1, 2, \ldots n$
   `Finish[i]` = `true` otherwise

2. Find an index `i` such that both

   `Finish[i]` = `false`
   $\texttt{Request}_i \leqslant \texttt{Work}$

   Go to step 4 if no such `i` exists

3. $\texttt{Work} = \texttt{Work} + \texttt{Allocation}_i$
   `Finish[i]` = `true`

   Go to step 2

Detection Algorithm

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

$$\begin{array}{lcll} \texttt{Work} & = & \texttt{Available} & \\ \texttt{Finish[i]} & = & \texttt{false} & \text{if } \texttt{Allocation} \neq 0 \text{ for } i = 1, 2, \ldots n \\ \texttt{Finish[i]} & = & \texttt{true} & \text{otherwise} \end{array}$$

2. Find an index i such that both

$$\begin{array}{lcl} \texttt{Finish[i]} & = & \texttt{false} \\ \texttt{Request}_i & \leqslant & \texttt{Work} \end{array}$$

   Go to step 4 if no such i exists

3.
$$\begin{array}{lcl} \texttt{Work} & = & \texttt{Work} + \texttt{Allocation}_i \\ \texttt{Finish[i]} & = & \texttt{true} \end{array}$$

   Go to step 2

4. If Finish[i] == false, for some i s.t. $1 \leqslant \texttt{i} \leqslant n$, then the system is in deadlock state. Moreover, if Finish[i] == false, then T_i is deadlocked

Example (Detection Algorithm)

- 5 threads T_0 through T_4;

Example (Detection Algorithm)

- 5 threads T_0 through T_4;
- 3 resource types: A (7 instances) B (2 instances) C (6 instances)

Example (Detection Algorithm)

- 5 threads T_0 through T_4;
- 3 resource types: A (7 instances) B (2 instances) C (6 instances)
- Current-state snapshot:

	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	

Example (Detection Algorithm)

- 5 threads T_0 through T_4;
- 3 resource types: A (7 instances) B (2 instances) C (6 instances)
- Current-state snapshot:

	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	

- Sequence

$$< T_0, T_2, T_3, T_4, T_1 >$$

will result in `Finish[i] = true` for all i

Example (Detection Algorithm (cont'd))

- T_2 requests an additional instance of type C

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

Example (Detection Algorithm (cont'd))

- T_2 requests an additional instance of type C

	Request
	$A\ B\ C$
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

- State of system?

Example (Detection Algorithm (cont'd))

- T_2 requests an additional instance of type C

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

- State of system?
  - Can reclaim resources held by thread T_0, but insufficient resources to fulfill other threads; requests

Example (Detection Algorithm (cont'd))

- T_2 requests an additional instance of type C

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

- State of system?
  - Can reclaim resources held by thread T_0, but insufficient resources to fulfill other threads; requests
  - Deadlock exists, consisting of threads T_1, T_2, T_3, and T_4

Detection Algorithm Usage

- When, and how often, to invoke depends on:

Detection Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?

Detection Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many threads will be affected by deadlock when it happens?

Detection Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many threads will be affected by deadlock when it happens?
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked threads “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion
  3. Resources that the thread has used

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion
  3. Resources that the thread has used
  4. Resources that the thread needs to complete

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion
  3. Resources that the thread has used
  4. Resources that the thread needs to complete
  5. How many threads will need to be terminated

Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one thread at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion
  3. Resources that the thread has used
  4. Resources that the thread needs to complete
  5. How many threads will need to be terminated
  6. Is the thread interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart the thread for that state

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart the thread for that state
- Starvation – same thread may always be picked as victim, include number of rollback in cost factor

Thanks! & Questions?