Chapter 8

Deadlocks

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Agenda

- Introduction
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock State

- A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set
- The events with which we are mainly concerned here are resource acquisition and release

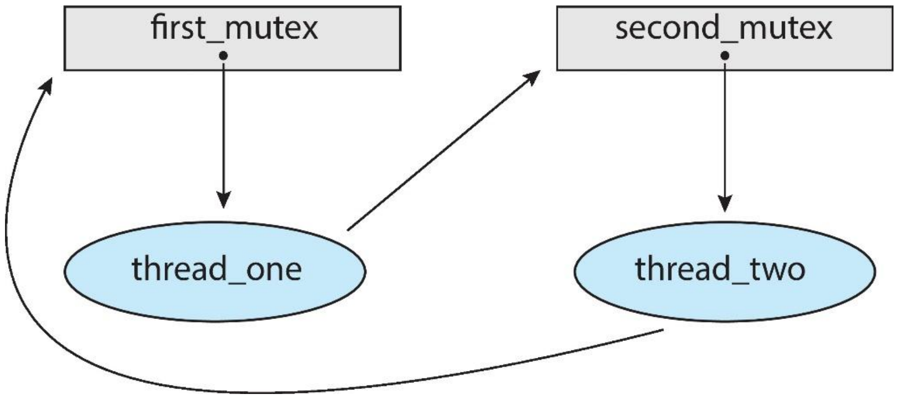
Deadlock in Multithreaded Application

- Deadlock example

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
...
```

Resource-allocation graph




```
// thread_one runs in this function
void *do_work_one(void *param) {
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}
```

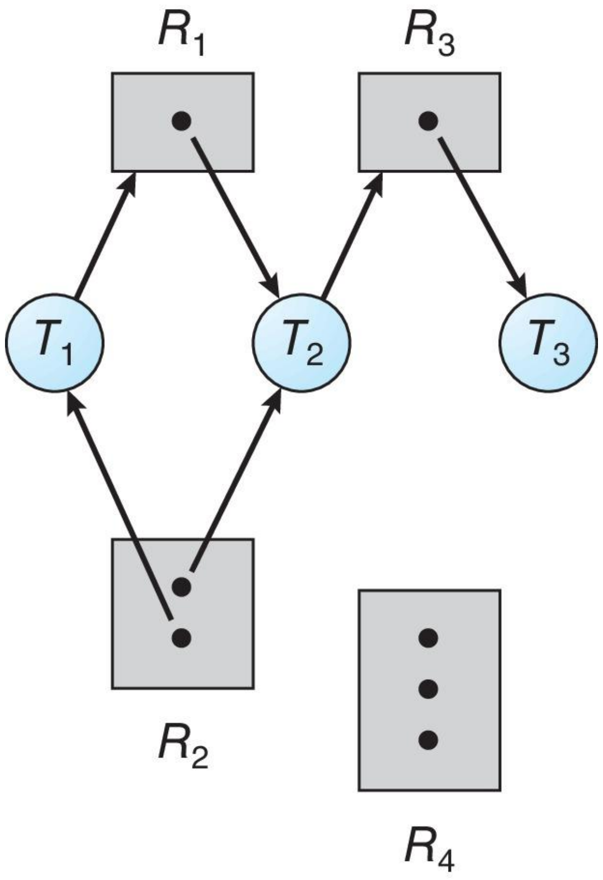
```
// thread_two runs in this function
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously
- **Mutual exclusion**: only one thread at a time can use a resource
- **Hold and wait**: a thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**: a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
- **Circular wait**: there exists a set $\{T_0, T_1, \ldots, T_n\}$ 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, …, 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.

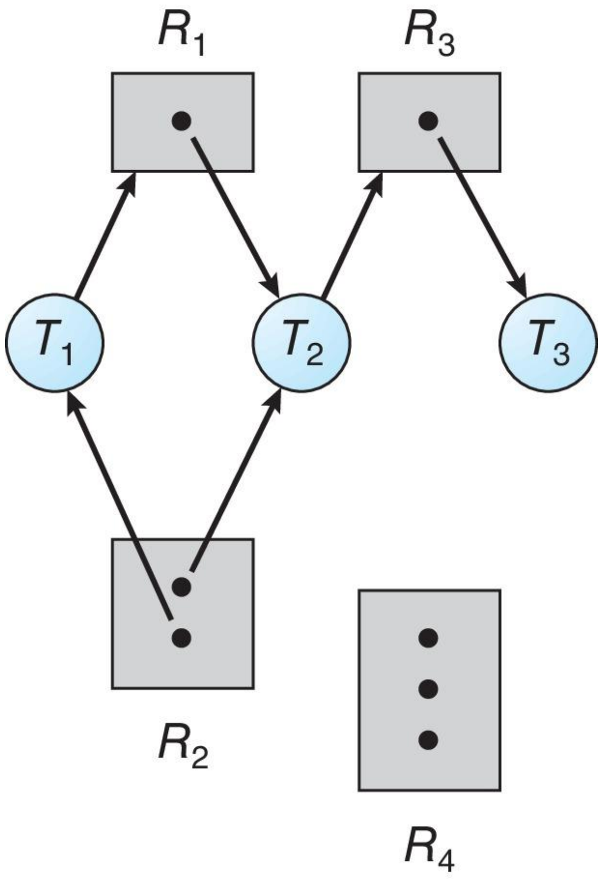
Resource-Allocation Graph

- Deadlock can be described more precisely in terms of a directed graph called **resource-allocation graph**
- A set of vertices V and a set of edges E
- V: two types of vertices
  - $T = \{T_1, T_2, \ldots, T_n\}$: a set of threads
  - $R = \{R_1, R_2, \ldots, R_m\}$: a set of resource types
- E: two types of directed edges
  - $T_i \rightarrow R_j$: **request edge**
  - $R_j \rightarrow T_i$: **assignment edge**
  - If a request edge is fulfilled, it become assignment edge



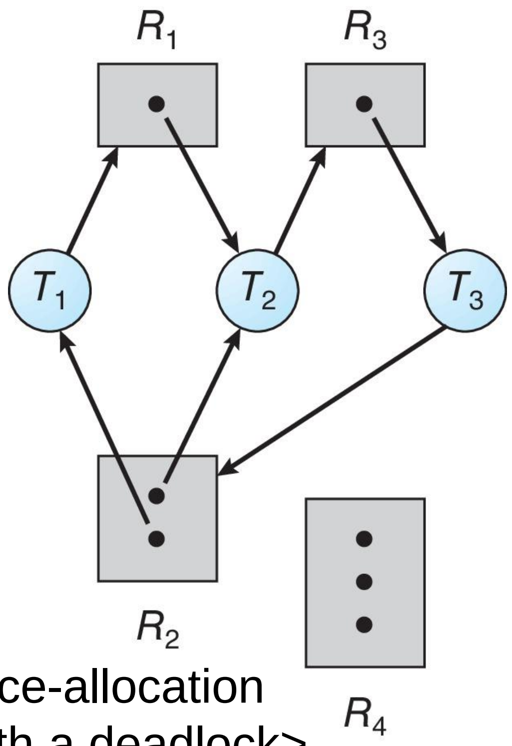
Resource-Allocation Graph

- Example of resource-allocation graph
  - One instance of R_1
  - Two instances of R_2
  - One instance of R_3
  - Three instances 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

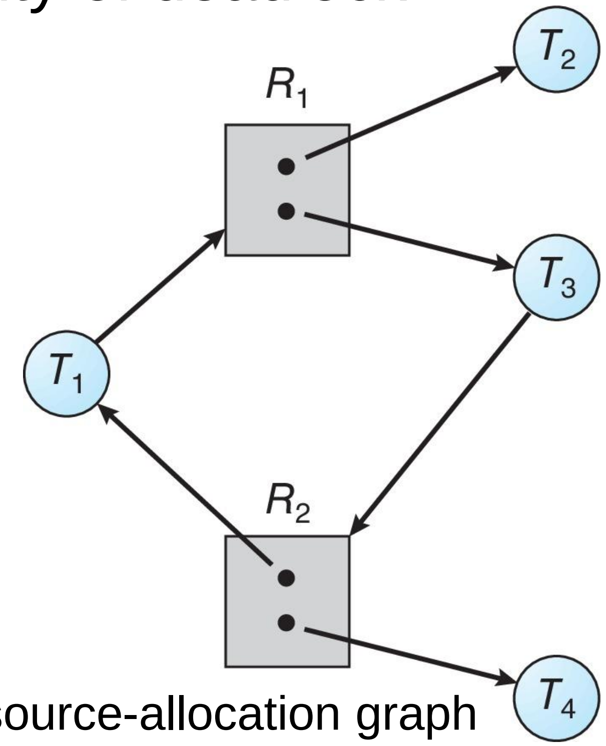


Resource-Allocation Graph

- 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



<Resource-allocation graph with a deadlock>



<Resource-allocation graph with a cycle but no deadlock>

Methods for Handling Deadlocks

- Three ways to handle deadlock problem
  - Ignore the problem and pretend that deadlocks never occur in the system
  - Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state
    - Deadlock prevention
    - Deadlock avoidance
  - Allow the system to enter a deadlock state, detect it, and recover

Methods for Handling Deadlock

- Deadlock prevention
  - A set of methods to ensure that at least one of necessary conditions cannot hold
  - Constraint on how requests for resources

- Deadlock avoidance
  - Keep the system in safe state in which deadlock cannot occur (using additional information)
    - Resources currently available or allocated to each thread
    - Additional information about future requests and release of each thread

Agenda

- Introduction
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from 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
  - Many resources are intrinsically non-sharable
- **Hold and wait**: must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Allocate all required resources before it begins execution or allow a thread to request resources only when it has none allocated to it
  - Low resource utilization and starvation possible

Deadlock Prevention

- **No Preemption**
  - If a thread 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

- **Circular Wait**: impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration
  - 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

  ex) F(`first_mutex`)=1, F(`second_mutex`)=5

The thread can request an instance of resource R_j if and only if $F(R_j) > F(R_i)$

code for `thread_two` could not be written as follows

```
// thread_one runs in this function
void *do_work_one(void *param) {
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}
```

```
// thread_two runs in this function
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Agenda

- Introduction
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock Avoidance

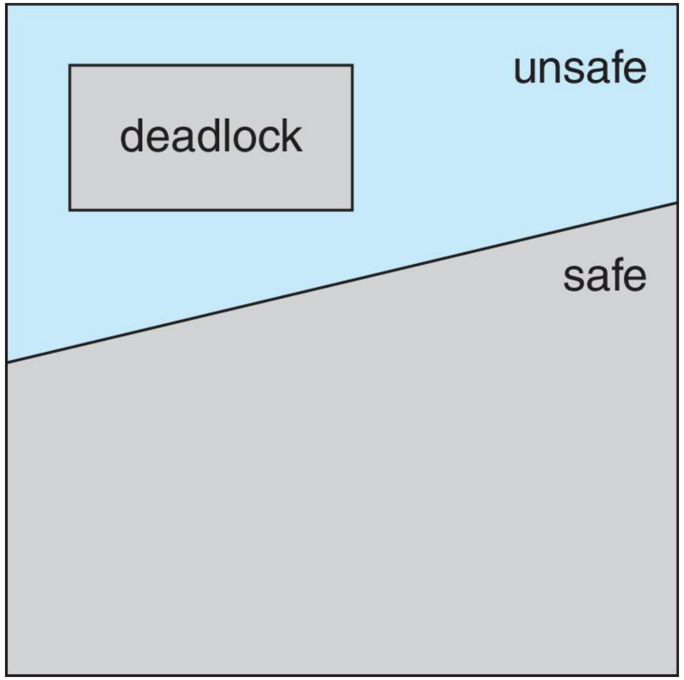
- Require additional information about how resources are to be requested.
  ex) Each thread declares maximum # of resources of each type it may request
- Deadlock avoidance algorithm dynamically examines resource-allocation state to ensure a circular-wait condition can never exist
- Resource-allocation state is defined by
  - # of available resources
  - # of allocated resources
  - Maximum demands of threads

Safe State

- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- **Safe state**: there exists a **safe sequence** of all threads
- **Safe sequence**: a sequence $<T_1, T_2, \ldots, T_n>$ is safe if, for each T_i, the resources that T_i can still request can be satisfied by currently available resources plus the resources held by all T_j, with $j < i$
  - If the resources that T_i 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

Safe State and Deadlock

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



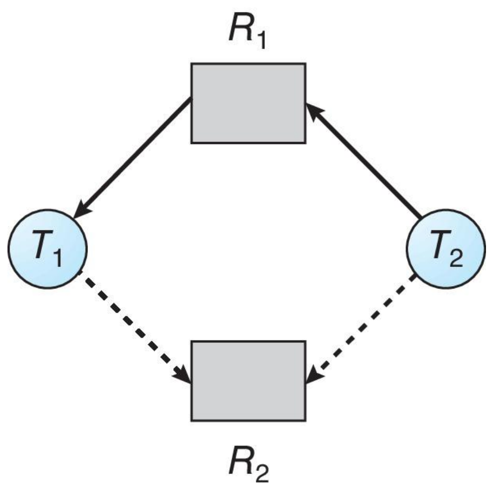
Avoidance Algorithm

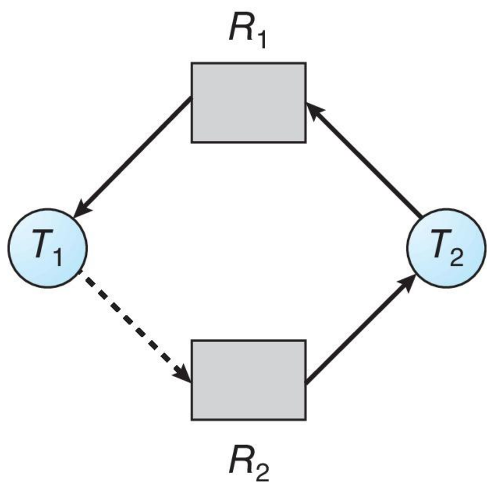
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the Banker's Algorithm

Resource-Allocation Graph Algorithm

- **Claim edge** $T_i \rightarrow R_j$: thread T_i may request resource R_j at some time in the future; represented by a dashed line
  - Claim edge converts to request edge when a thread requests a resource
  - When a resource is released by a thread, assignment edge reconverts to a claim edge
  - Resources must be claimed a priori in the system
- 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





Unsafe State!

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
  - **Available**: Vector of length m. If ***Available[j]=k***, there are k instances of resource type R_j available
  - **Max**: n x m matrix. If ***Max[i][j]=k***, then thread T_i may request at most k instances of resource type R_j
  - **Allocation**: n x m matrix. If ***Allocation[i][j]=k*** then T_i is currently allocated k instances of R_j
  - **Need**: n x m matrix. If ***Need[i][j]=k***, then T_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*** and ***Finish*[*i*]=*false*** for *i*=0,1,…,*n-1*

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

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

   (b) $\boldsymbol{Need_i \leq Work}$

   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

Resource-Request Algorithm

$\boldsymbol{Request_i}$ be the request vector for thread $\boldsymbol{T_i}$.

If $\boldsymbol{Request_i[j]=k}$ then thread $\boldsymbol{T_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i} \le \boldsymbol{Need_i}$, go to step 2. Otherwise, raise error condition, since thread has exceeded its maximum claim.
2. If $\boldsymbol{Request_i} \le \boldsymbol{Available}$, go to step 3. Otherwise $\boldsymbol{T_i}$ must wait, since resources are not available.
3. Pretend to allocate requested resources to $\boldsymbol{T_i}$ by modifying the state as follows:

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

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

Example: Banker's Algorithm

- 5 threads T_0 through T_4
- 3 resource types: *A* (10 inst.), *B* (5 inst.), and *C* (7 inst.)
- Snapshot at time t_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2	7 4 3
T_1	2 0 0	3 2 2		1 2 2
T_2	3 0 2	9 0 2		6 0 0
T_3	2 1 1	2 2 2		0 1 1
T_4	0 0 2	4 3 3		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

Example: Banker's Algorithm

- Suppose T_1 requests (1, 0, 2)
  - Check that ***Request*** $\leq$ ***Available*** (that is, (1,0,2) ≤ (3,3,2) → True

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

- Executing safety algorithm shows that sequence <T_1, T_3, T_4, T_0, T_2> satisfies safety requirement

Can request for (3,3,0) by T_4 be granted?
Can request for (0,2,0) by T_0 be granted?

Agenda

- Introduction
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock Detection

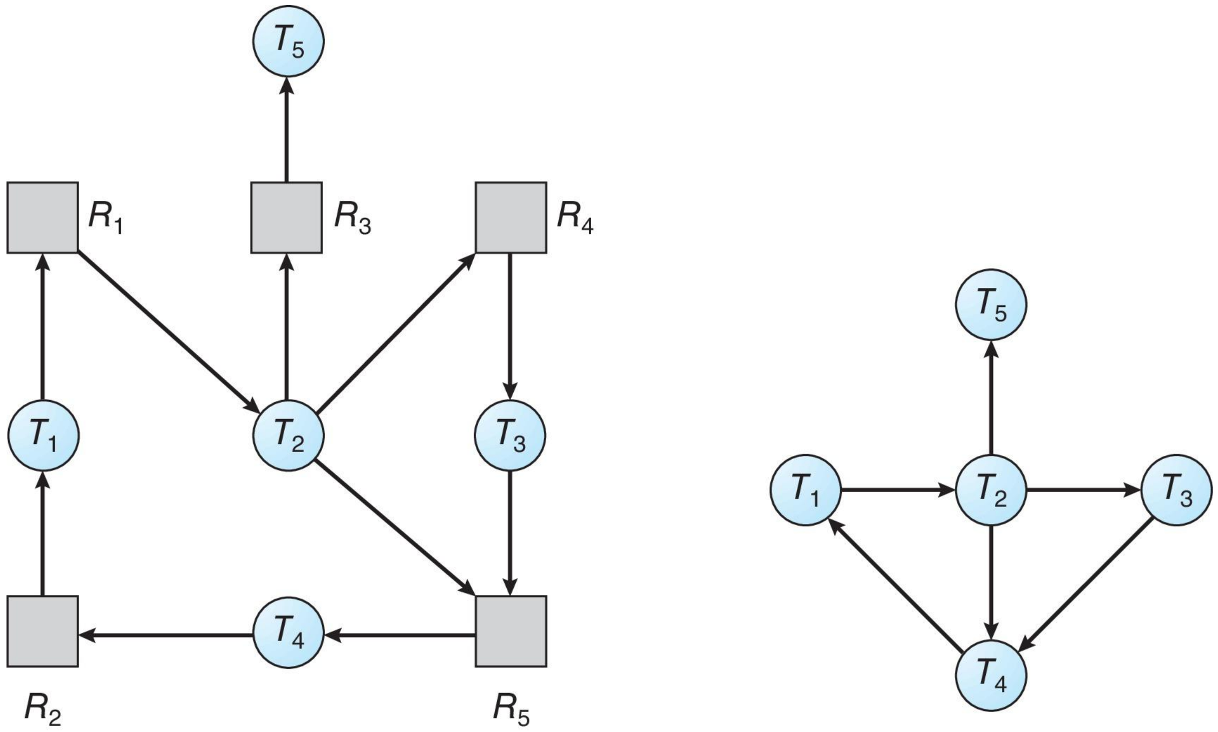
- Allow system to enter deadlock state

- Detection algorithm
  - Single instance of each resource type
  - Multiple instances of a resource type (skip, Additional slides)

- Recovery scheme

Single Instance of Each Resource Type

- **Wait-for graph**: removing resource nodes from resource-allocation graph and collapsing the appropriate edges

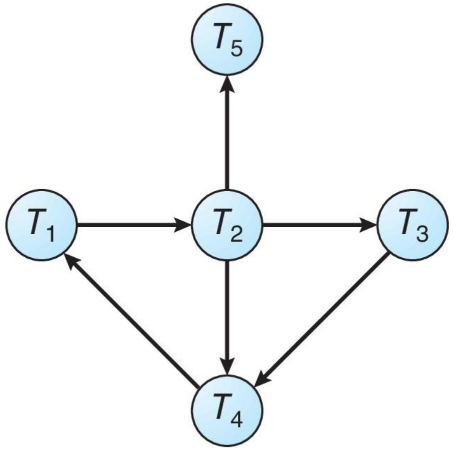


(a) Resource-Allocation Graph

(b) Corresponding wait-for graph

Single Instance of Each Resource Type

- A deadlock exists if and only if the wait-for graph contains a cycle.
- To detect deadlock, system should
  - Maintain wait-for graph
  - Invoke an algorithm to detect a cycle - $O(n^2)$



Wait-for graph

Detection-Algorithm Usage

- When should we invoke the detection algorithm?
  - How often is a deadlock likely to occur?
  - How many threads will be affected by deadlock when it happens?

- We may invoke detection algorithm whenever a request for allocation cannot be granted immediately
  - Deadlocks occur only when some threads makes a request that cannot be granted immediately
  - We can find the thread which caused deadlock.

Agenda

- Introduction
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Recovery from Deadlock

- Process termination

- Resource preemption

Process Termination

- Abort all deadlocked processes
  - Too expensive

- Abort one process at a time until the deadlock is eliminated
  - Order of priority
    - Time from start / time to completion
    - Resources the process has used / needs to complete
    - How many processes will need to be terminated?
    - Is the process interactive or batch?

Resource Preemption

- Selecting a victim
  - Minimizing cost (# of resources a process has, amount of time consumed so far, …)

- Rollback
  - The selected process should return to some safe state and restart it.

- Starvation
  - How can we guarantee that resources will not always be preempted from the same process?

ADDITIONAL SLIDES

Several Instances of a Resource Type

- **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 thread
- **Request**: An ***n* x *m*** matrix indicates the current request of each thread. 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:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *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

HANDONG GLOBAL UNIVERSITY

Detection Algorithm

3. ***Work* = *Work* + *Allocation*$_i$**
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i]* == *false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***T**$_i$* is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example: Detection Algorithm

- Five threads $\boldsymbol{T_0}$ through $\boldsymbol{T_4}$
- Three resource types: A (7 inst.), *B* (2 inst.), and *C* (6 inst.)
- Snapshot at time $\boldsymbol{t_0}$:

	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 <$\boldsymbol{T_0, T_2, T_3, T_1, T_4}$> will result in ***Finish[i] = true*** for all ***i***

Example: Detection Algorithm

- 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 processes; requests
  - Deadlock exists, consisting of threads T_1, T_2, T_3, and T_4