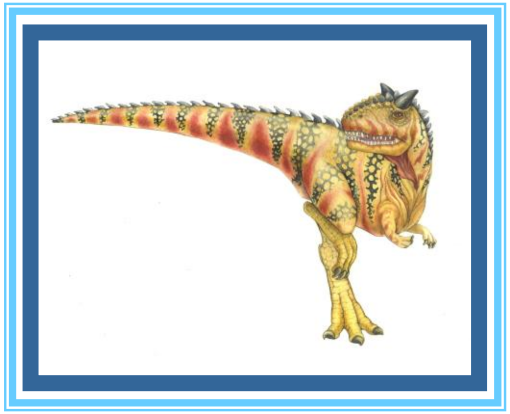
# Chapter 8: Deadlocks



# Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

# Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

# System Model

- System consists of a finite number of resources to be distributed among a number of competing threads
- 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**
- If a thread requests an instance of a resource type, the allocation of ***any*** instance of the type should satisfy the request.
- Synchronization tools such as mutex locks and semaphores, are also system resources.

# 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(s1)
wait(s2)
```

- $T_2$:

```
wait(s2)
wait(s1)
```

# 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 $T_0$.

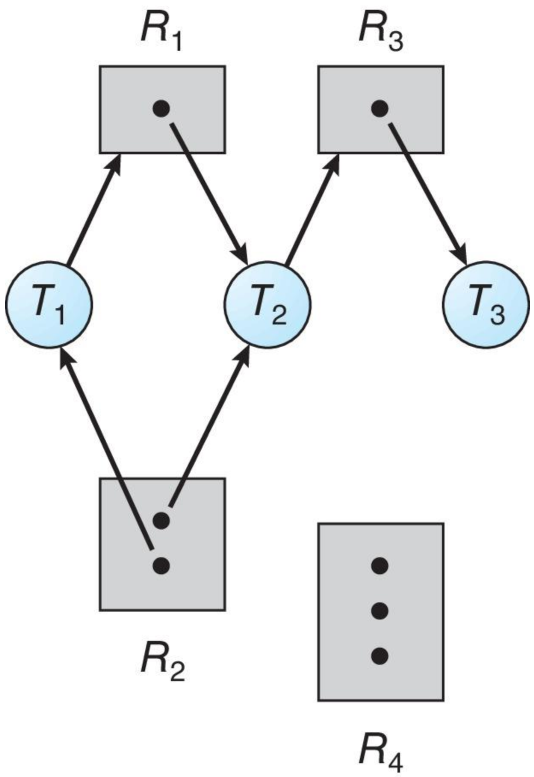
# Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

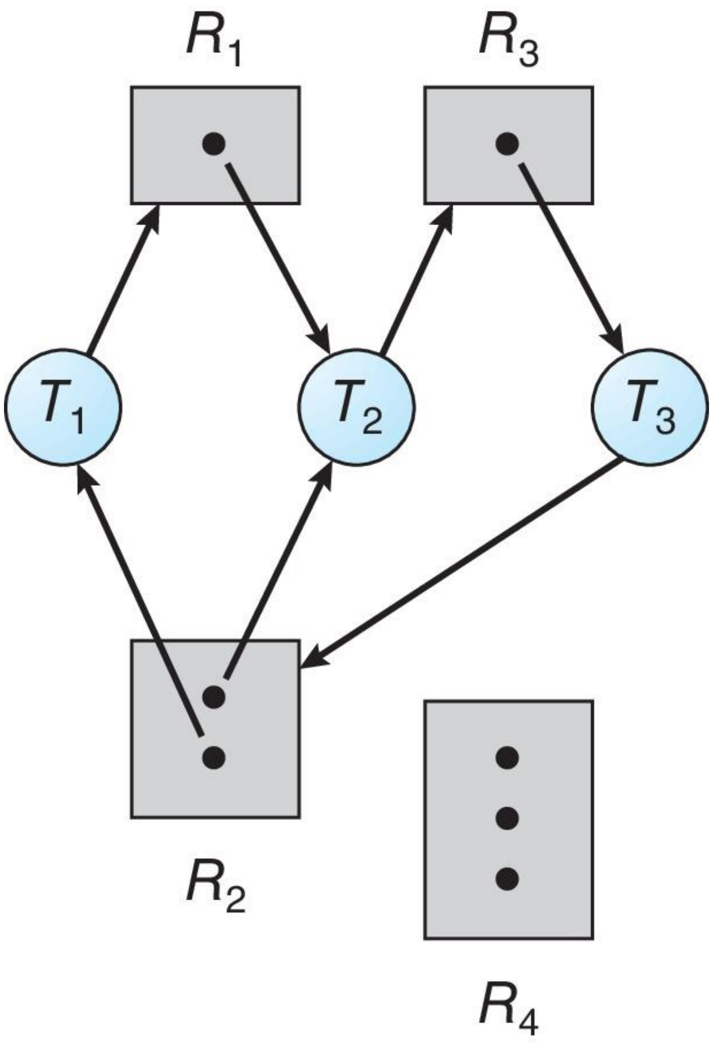
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed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

# Resource Allocation Graph Example

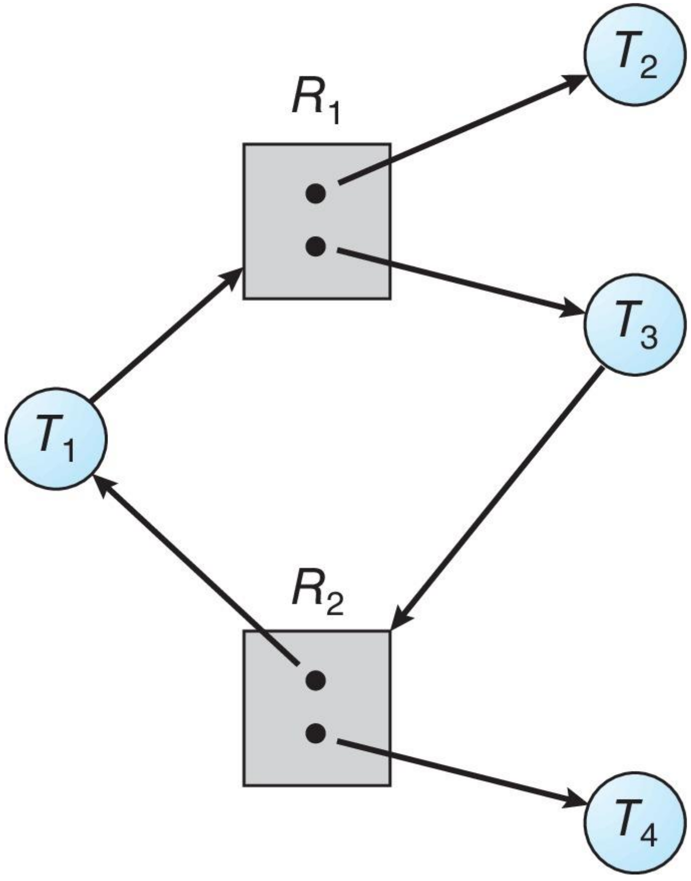
- One instance of $R_1$
- Two instances of $R_2$
- One instance of $R_3$
- Three instance of $R_4$
- $T_1$ holds one instance of $R_2$ and is waiting for an instance of $R_1$
- $T_2$ holds one instance of $R_1$, one instance of $R_2$, and is waiting for an instance of $R_3$
- $T_3$ is holds one instance 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
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

# 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.
  - It is up to kernel and application developers to write programs that handle deadlocks.

# 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
  - In general, we cannot prevent deadlocks by denying the mutual-exclusion condition.
- **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.
  - Allow thread to request resources only when the thread has none allocated to it.
    - Low resource utilization; starvation possible

# Deadlock Prevention (Cont.)

- **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.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

**`first_mutex = 1`**
**`second_mutex = 5`**

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);
}
```

# 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

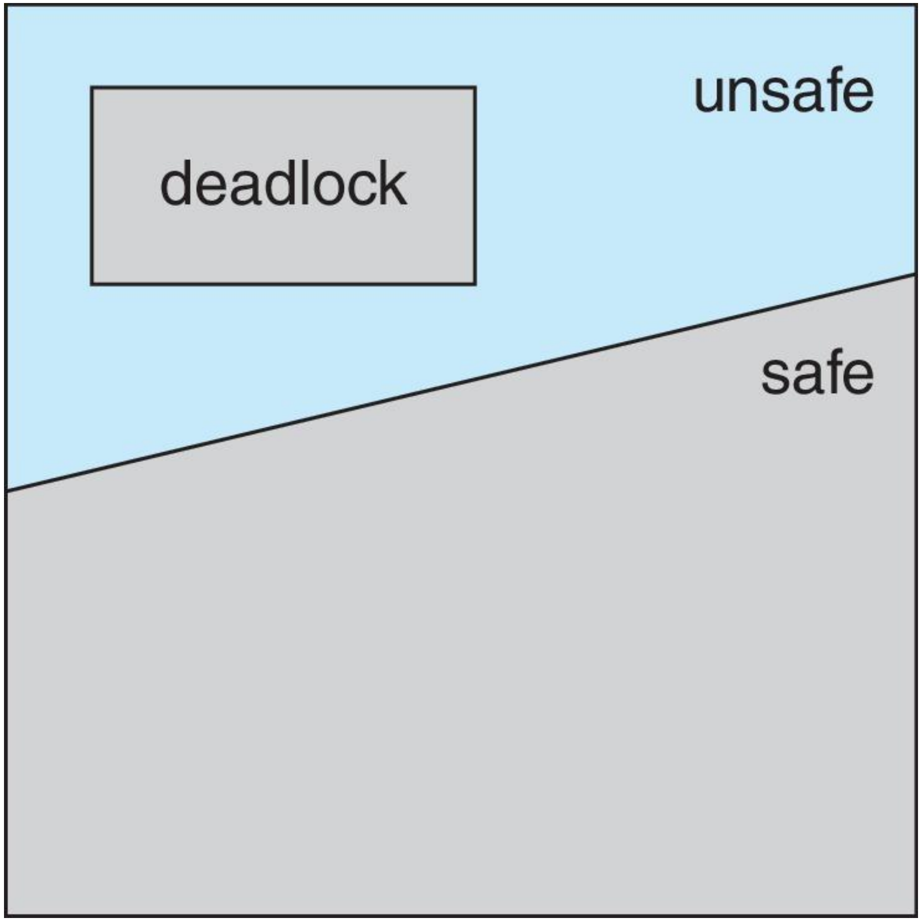
# Safe State

- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<T_1, T_2, \ldots, T_n>$ of ALL the threads (**safe sequence**) in the systems such that for each $T_i$, the resources that $T_i$ can still request can be satisfied by 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 $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

# Safe, Unsafe, Deadlock State



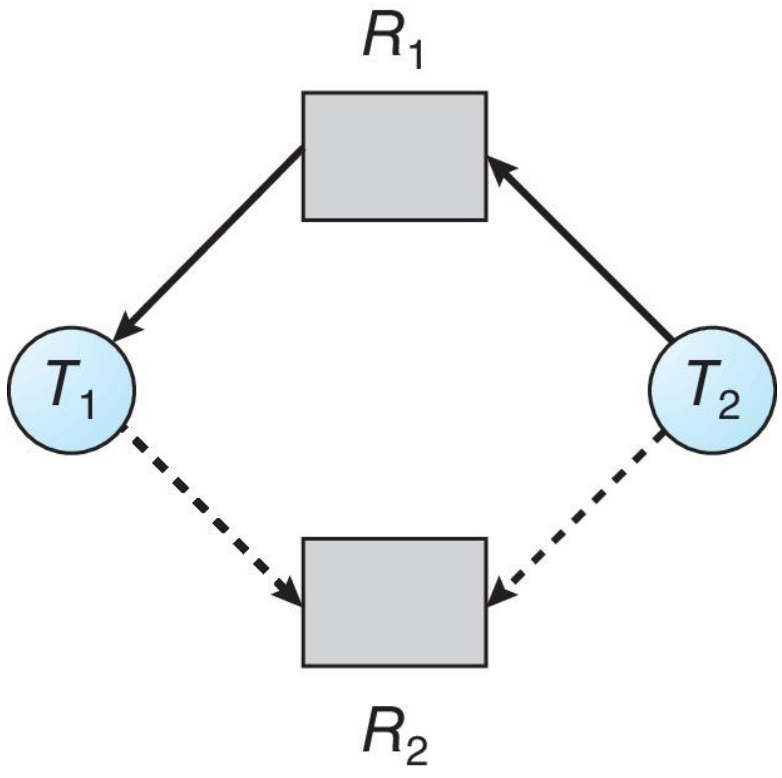
# Avoidance Algorithms

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

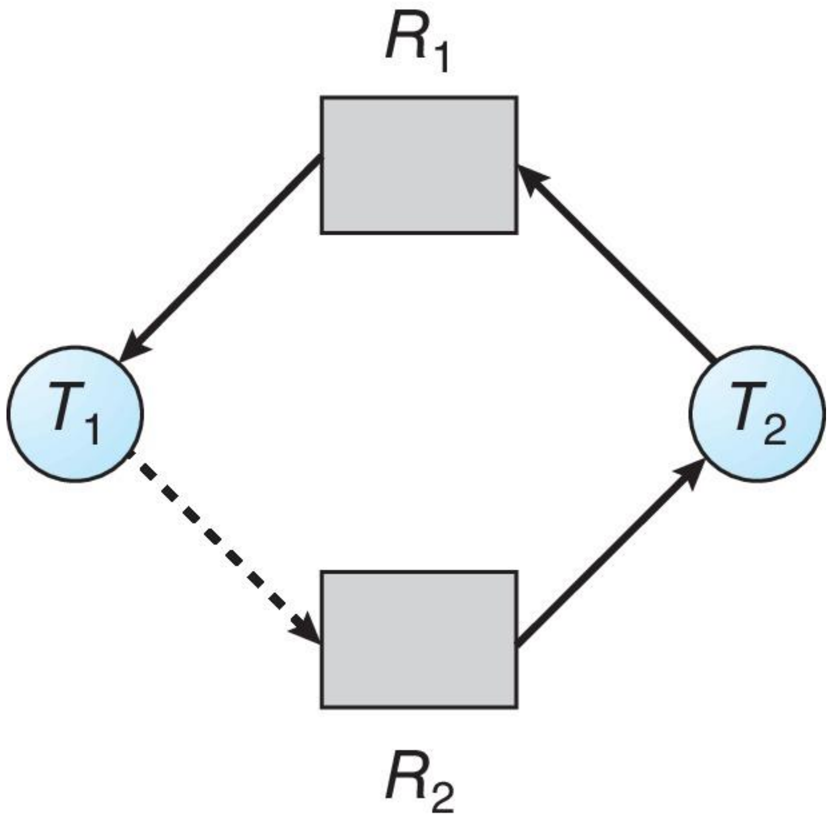
# Resource-Allocation Graph Scheme

- **Claim edge** $T_i \rightarrow R_j$ indicated that process $T_j$ 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 $R_j$
- 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
- When a new thread enters the system, it must declare the maximum use of the resources.
  - number may not exceed the total number of resources in the system
- 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 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**: $n$ x $m$ matrix. If *Max* [$i,j$] = $k$, then process $T_i$ 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

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

# Data Structures for the Banker's Algorithm

- To simplify the presentation of the banker's algorithm, we establish the given notation
- Let $X$ and $Y$ be vectors of length $n$
  - We say that $\leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \ldots, n.$
    - $X = (1,7,3,2)$ and $Y = (0,3,2,1)$ is $Y \leq X$ (??)
- In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$

# 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:

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

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

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

3. $\boldsymbol{Work = Work + Allocation_i}$
   ***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$_i$*** = request vector for process $T_i$. If ***Request$_i$* [*j*] = *k*** then process $T_i$ wants ***k*** instances of resource type $R_j$

1. If $\textit{Request}_i \le \textit{Need}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\textit{Request}_i \le \textit{Available}$, 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:

$$\textit{Available} = \textit{Available} - \textit{Request}_i;$$

$$\textit{Allocation}_i = \textit{Allocation}_i + \textit{Request}_i;$$

$$\textit{Need}_i = \textit{Need}_i - \textit{Request}_i;$$

- 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 of Banker's Algorithm

- 5 threads $T_0$ through $T_4$;

  3 resource types:

  *A* (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time $T_0$:

| | *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 (Cont.)

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

| | *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

# Example: $T_1$ Request (1,0,2)

- Check that Request ≤ 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 < $\boldsymbol{T_1}$, $\boldsymbol{T_3}$, $\boldsymbol{T_4}$, $\boldsymbol{T_0}$, $\boldsymbol{T_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{T_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{T_0}$ be granted?

# Deadlock Detection

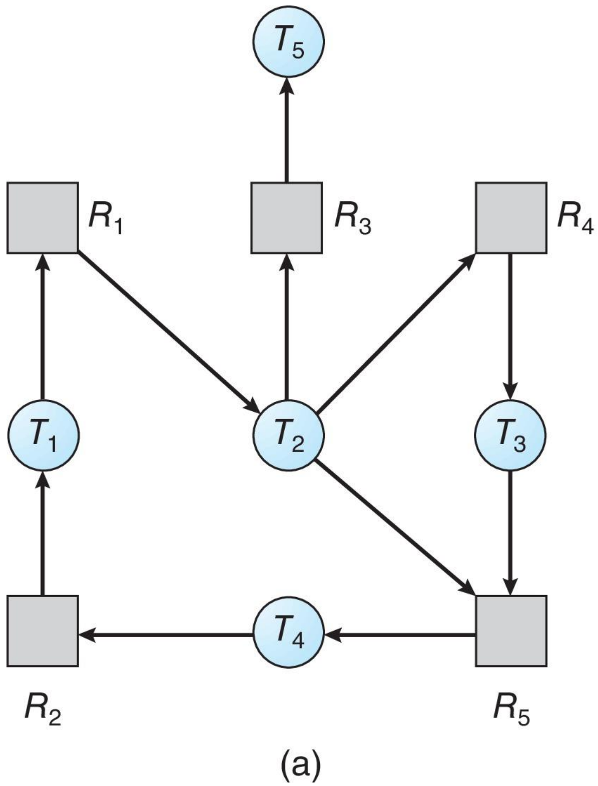
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

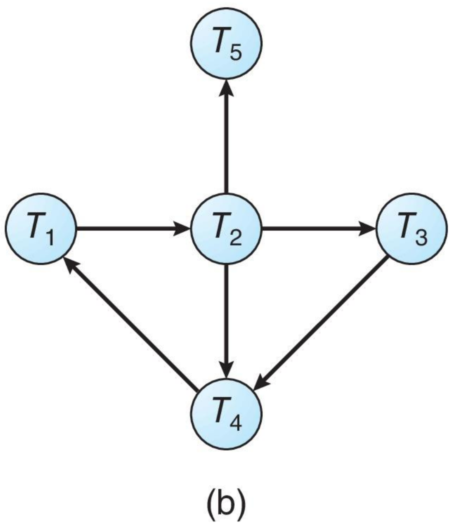
# Single Instance of Each Resource Type

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

- Wait-for graph is obtain from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges
- *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
  - Overhead!!

# Resource-Allocation Graph and  Wait-for Graph




Resource-Allocation Graph        Corresponding wait-for graph

# Several Instances of a Resource Type

- Wait-for graph scheme is not applicable for multiple instances 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 $\boldsymbol{Allocation_i \neq 0}$, then
      ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***
2. Find an index ***i*** such that both:
   a) ***Finish*[*i*] == *false***
   b) $\boldsymbol{Request_i \leq Work}$

   If no such ***i*** exists, go to step 4
3. $\boldsymbol{Work = Work + Allocation_i}$
   ***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 $T_i$ is deadlocked

# Detection Algorithm (Cont.)

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

# Example of Detection Algorithm

- Five threads $T_0$ through $T_4$; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time $T_0$:

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

# Example (Cont.)

- $T_2$ 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 (Cont.)

- $T_2$ 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 processes $T_1$, $T_2$, $T_3$, and $T_4$

# Detection-Algorithm Usage

- When, and how often, to invoke (Frequently, or infrequently?) depends on:
  - How often a deadlock is likely to occur?
  - The possible consequences of not catching them immediately i.e., How many threads will need to be rolled back?
    - one for each disjoint cycle
- Do deadlock detection after every resource allocation which cannot be immediately granted. Overhead!!
- Do deadlock detection only when there is some clue that a deadlock may have occurred.
  - CPU utilization reduces to 40% or so.
  - impossible to detect the processes involved in the original deadlock

# Recovery from Deadlock

- There are three basic approaches to recovery from deadlock:
  - Inform the system operator, and allow him/her to take manual intervention.
  - Terminate one or more threads involved in the deadlock
  - Preempt resources.

# Recovery from Deadlock:  Thread Termination

- **Thread 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** – Deciding which resources to preempt from which processes to minimize the cost.
- **Rollback** – return to some safe state, restart the thread for that state
  - roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process
- **Starvation** – same thread may always be picked as victim
  - include number of rollback in cost factor

# Practice Problem: Banker's Algorithm

Consider the following snapshot of a system:

| | *Allocation* | *Max* | *Available* |
|---|---|---|---|
| | *A B C D* | *A B C D* | *A B C D* |
| $T_0$ | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| $T_1$ | 1 0 0 0 | 1 7 5 0 | |
| $T_2$ | 1 3 5 4 | 2 3 5 6 | |
| $T_3$ | 0 6 3 2 | 0 6 5 2 | |
| $T_4$ | 0 0 1 4 | 0 6 5 6 | |

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from thread $T_1$ arrives for (0,4,2,0), can the request be granted immediately?

# End of Chapter 8