

Operating Systems

Prepared by Dr. Alaa Eldeen Sayed Ahmed

IT263 / CYB510
Deadlocks

Lecture 6



- Objectives:
 - System Model
 - Deadlock in Multithreaded Applications
 - Deadlock Characterization
 - Methods for Handling Deadlocks
 - Deadlock Prevention
 - Deadlock Avoidance
 - Deadlock Detection
 - Recovery from Deadlock

- Topics:
 - 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

Background

- In a multiprocessing/multithreading environment,
 - Several threads may compete for a finite number of resources
 - A thread requests resources:
 - If the **resources** are **not available** at that time, the **thread** enters a **waiting state**.
 - **Sometimes**, a waiting thread **can never again** change state, because the resources it has requested are held by other waiting threads
 - This situation is called a **deadlock**.
- **Deadlock** is a situation in which every process in a set of processes is waiting for an event that can be caused only by another process in the set, all of which are waiting

- System consists of resources
- 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.
- Under the **normal mode of operation**, a process/thread utilizes a resource as follows:
 - **Request:**
 - **Use:**
 - **Release:**

- **Request:**
 - The thread requests the resource. If the request cannot be granted immediately (*i.e., if a mutex lock is currently held by another thread*), then the requesting thread(process) must wait until it can acquire the resource.
- **Use:**
 - The thread can operate on the resource (*i.e., if the resource is a mutex lock, the thread can access its critical section*)
- **Release:**
 - The thread releases the resource

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

- Assume Two **semaphores(first_mutex and second_mutex)** are created and initialized to 1

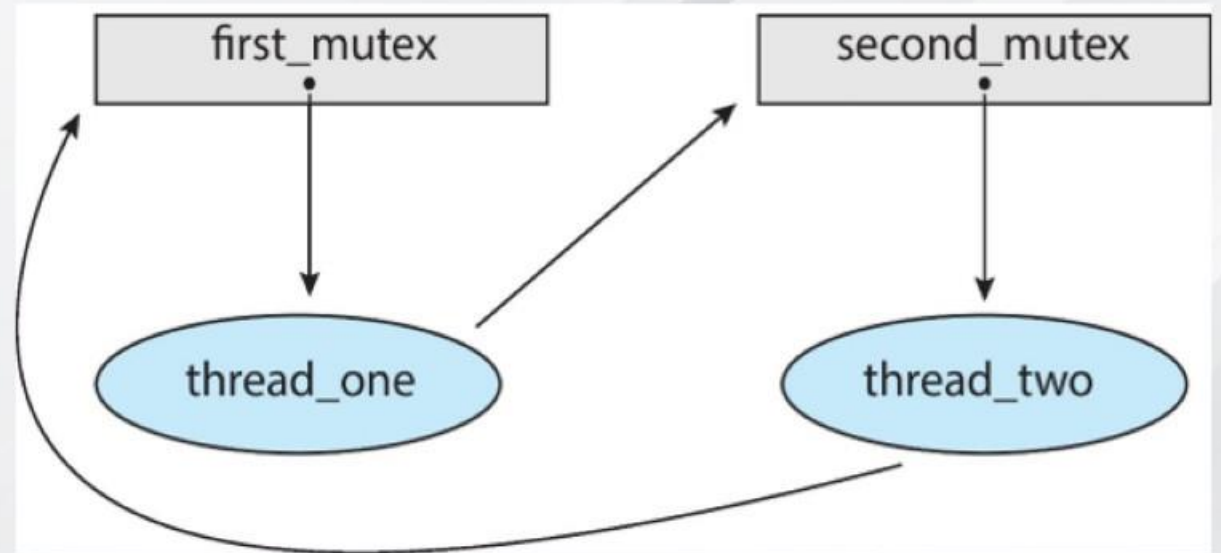
```
/* thread one runs in this function */
void do_work_one(void param)
{
    wait( first_mutex );
    wait(second_mutex );
    /**
     * Do some work
     */
    signal( second_mutex );
    signal( first_mutex );
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    wait( second_mutex );
    wait(first_mutex );
    /**
     * Do some work
     */
    signal(first_mutex );
    signal(second_mutex );
}
```

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.
- A set of vertices **V** and a set of edges **E**.
 - **V** is partitioned into two types:
 - ▶ $P = \{P_1, P_2, \dots, P_n\}$,
 - the set consisting of all the processes in the system
 - ▶ $R = \{R_1, R_2, \dots, R_m\}$,
 - the set consisting of all resource types in the system
 - **Request edge** – directed edge $P_i \rightarrow R_j$
 - **Assignment edge** – directed edge $R_j \rightarrow P_i$ (or $P_i \leftarrow R_j$)

- Deadlock is possible, if :
 - **thread_one** acquires **first_mutex** and **thread_two** acquires **second_mutex**
 - **thread_one** then waits for **second_mutex** and **thread_two** waits for **first_mutex**.



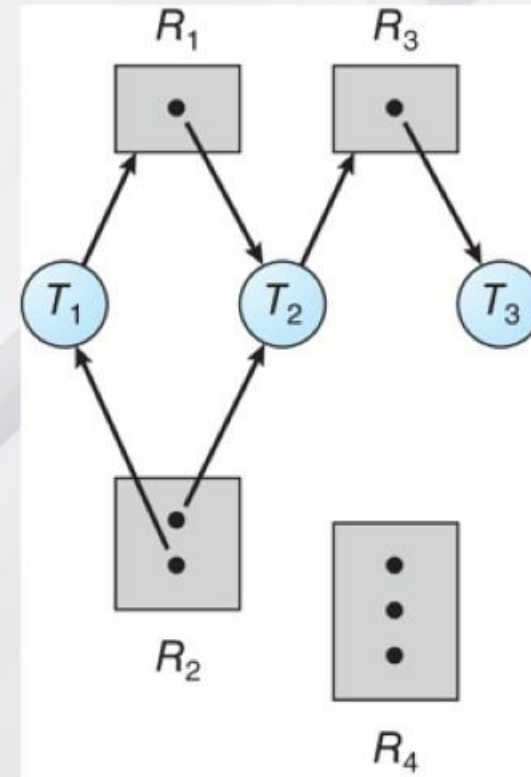
Can be illustrated with a **resource allocation graph**:

Circular Wait

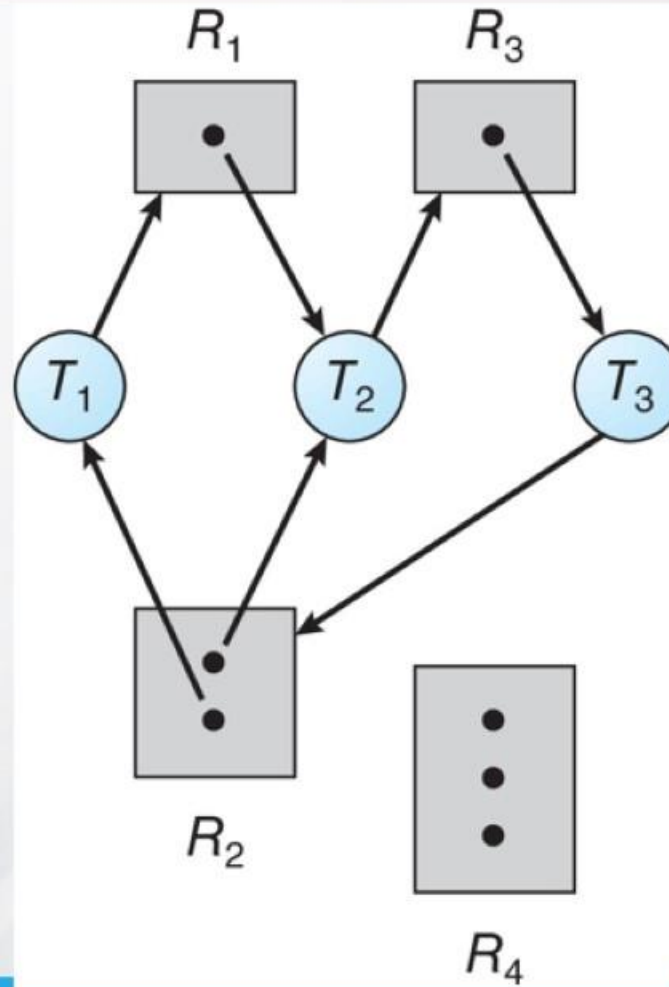
- Note that, even though **deadlock is possible**,
 - it will not occur if **thread_one** can **acquire and release** the **mutex** locks for **first mutex** and **second mutex** **before** **thread_two** **attempts to acquire** the locks.
- And, of course, the **order** in which the threads run **depends** on how they are scheduled by the CPU scheduler.
- This example illustrates a problem with handling deadlocks:
 - it is **difficult** to **identify** and **test** for deadlocks that may occur only **under certain scheduling** circumstances

Resource Allocation Graph Example

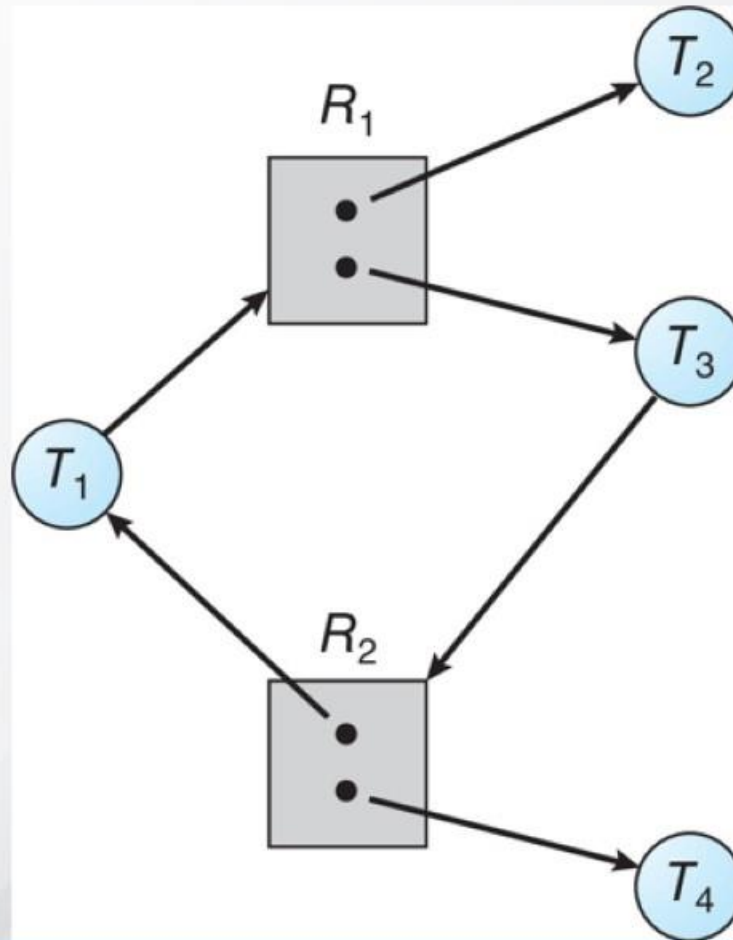
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



- 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.

Deadlock Prevention

Simply, Invalidate one of the four necessary conditions for deadlock:

1. Mutual Exclusion – We **cannot** prevent deadlocks **by denying the mutual-exclusion condition**

- not required for sharable resources (*e.g., read-only files*);
- must hold for non-sharable resources (*e.g. a **mutex** lock cannot be simultaneously shared by several threads*)

Deadlock Prevention

2. Hold and Wait – We need to: “ensure that the hold-and-wait condition **never occurs in the system**”

- must guarantee that whenever a process requests a resource, it does not hold any other resources
 - **Approach #1:** require process to request and be allocated all its resources **before it begins execution**,
 - **Cons:** **impractical** for most applications and if applied Low resource utilization
 - **Approach #2:** allows a thread to request resources **only** when it has **none allocated** to it.
 - **Cons:** may lead to **starvation**, *wait indefinitely*
 - *If the needed R is always allocated to some other thread*

Deadlock Prevention (Cont.)

3.No Preemption – of resources that have already been allocated, to prevent deadlock, we need to ensure that this condition **does not hold**:

- If a process is holding some resources and requests another that cannot be immediately allocated to it, *then all resources currently being held are released (preempted)*
 - Preempted resources are added to the list of resources for which the process is waiting for
 - Process will be restarted only when it can regain its **old** resources as well as the **new** ones that it is requesting

Deadlock Prevention (Cont.)

- **All three above options are almost impractical in most situations**

4.Circular Wait – preventing it is a practical solution

- imposing a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

How to Invalidate Circular Wait

- Invalidating the circular wait condition is most common.
- Simply, assign each resource (*i.e. **mutex** locks*) a **unique number**, (*this number determines whether one precedes another*)
- An **ordering among** all synchronization objects, Then, resources must be acquired in order.
- i.e. each thread can **request** resources only in
- an increasing order of enumeration and **released** in a decreasing order

How to Invalidate Circular Wait

- Example: **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)
{
    wait( first_mutex );
    wait(second_mutex );
    /**
    * Do some work
    */
    signal( second_mutex );
    signal( first_mutex );
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
    wait( second_mutex );
    wait(first_mutex );
    /**
    * Do some work
    */
    signal(first_mutex );
    signal(second_mutex );
}
```

Deadlock Prevention (Cont.)

- **Deadlock-prevention** algorithms **prevent** deadlocks by limiting how **requests & releases** can be made.
- **Pros:** The limits ensure that at least one of the necessary conditions for deadlock cannot occur
- **Cons:** Possible side effects of preventing deadlocks by this method,
 - low device utilization and
 - reduced system throughput

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

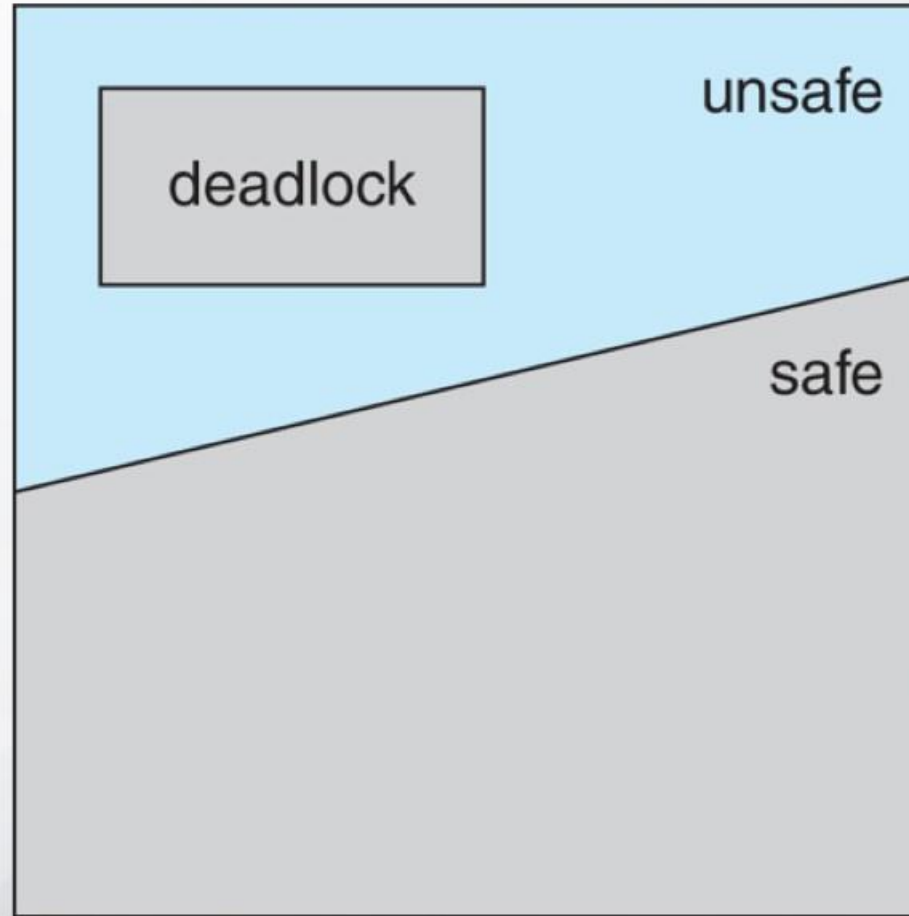
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *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 $j < i$
- That is:
 - If 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

- 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



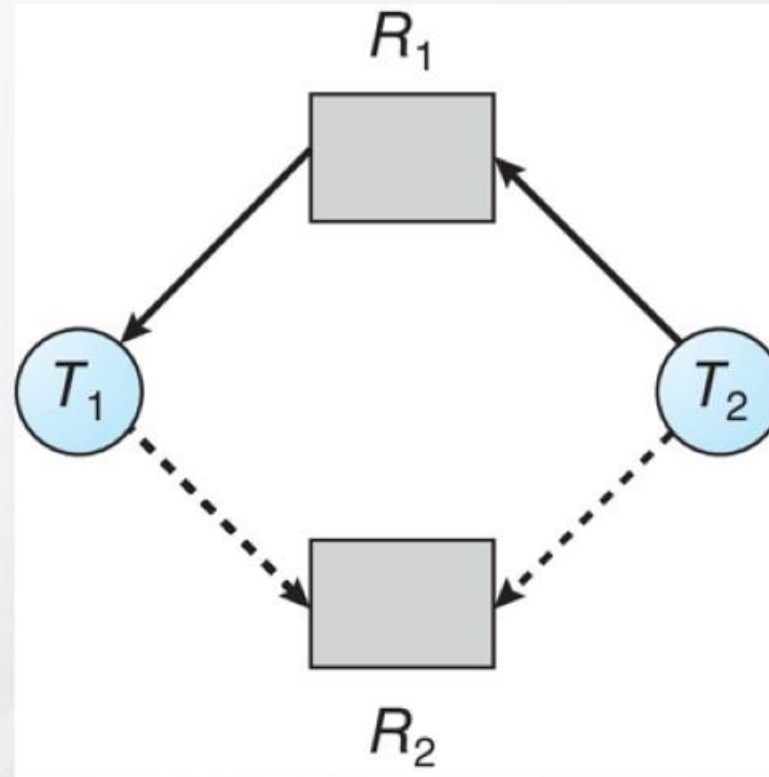
Avoidance Algorithms

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

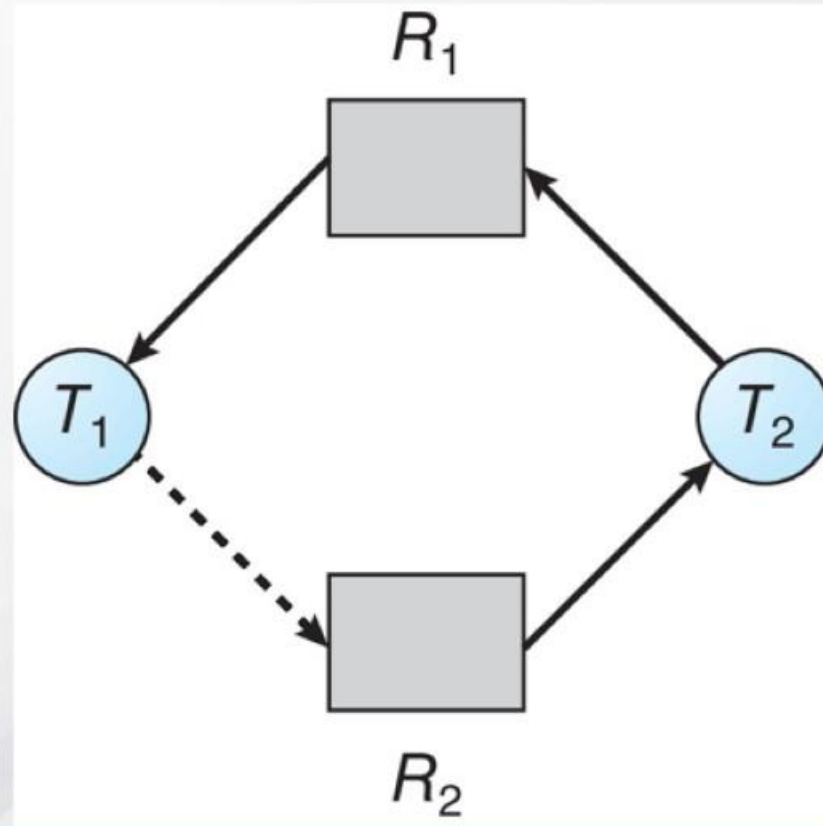
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, 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 process P_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
- 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

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:** $n \times m$ 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
 $Need[i, j] = Max[i, j] - Allocation[i, j]$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n , respectively. Initialize:
 $Work = Available$
 $Finish[i] = false$ for $i = 0, 1, \dots, n-1$
2. Find an 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

Resource-Request Algorithm for Process P_i

$Request_i$ = 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:

$Available = Available - Request_i$

$Allocation_i = Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

- 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>Need</u>		<u>Allocation</u>		<u>Max</u>		<u>Available</u>
	$A\ B\ C$		$A\ B\ C$		$A\ B\ C$		$A\ B\ C$
P_0	7 4 3	P_0	0 1 0		7 5 3		3 3 2
P_1	1 2 2	P_1	2 0 0		3 2 2		
P_2	6 0 0	P_2	3 0 2		9 0 2		
P_3	0 1 1	P_3	2 1 1		2 2 2		
P_4	4 3 1	P_4	0 0 2		4 3 3		

The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety criteria

Example: P1 Request (1,0,2)

- Request $\leq$ Need (that is, $(1,0,2) \leq (1,2,2)$)
- 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>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	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

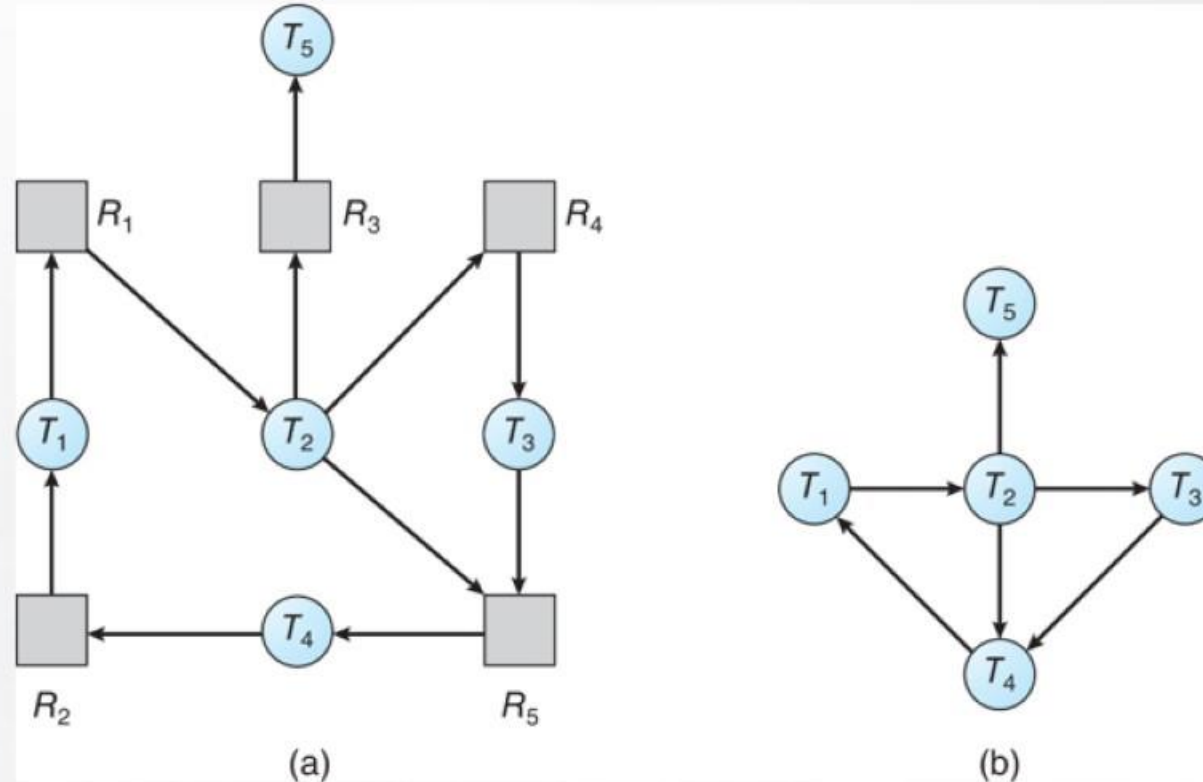
Deadlock Detection

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

Single Instance of Each Resource Type

- Maintain **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_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

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- 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 processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- 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. Resources process needs to complete
 5. How many processes will need to be terminated
 6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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



Thank You