

# OPERATING SYSTEMS

## CS3500; CH - 8

**PROF. SUKHENDU DAS,  
DEPTT. OF COMPUTER SCIENCE AND ENGG.,  
IIT MADRAS, CHENNAI – 600036.**

---

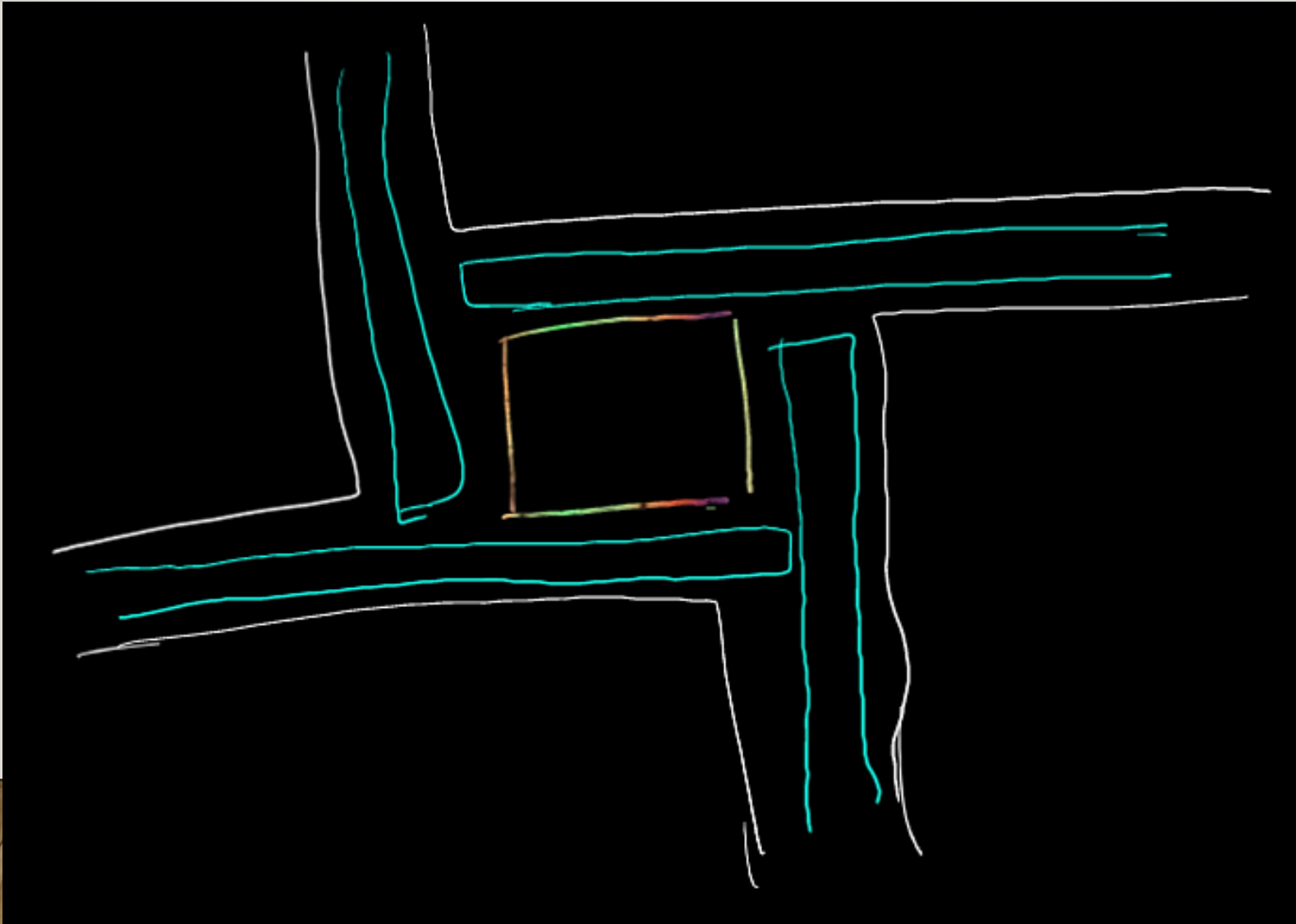
Email: [sdas@cse.iitm.ac.in](mailto:sdas@cse.iitm.ac.in)

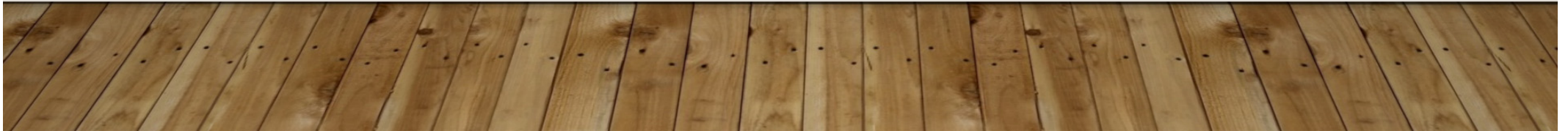
URL: <http://www.cse.iitm.ac.in/~vplab/os.html>

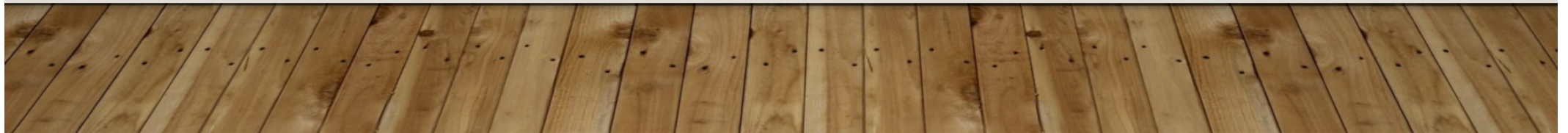
Sept – 2022.

# DEADLOCKS

---







# OUTLINE

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

# System Model

- 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.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

# 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)
```

*/\*\* Refer "Liveness" discussed in Process Synch*

*Also self-study "Livelock; e.g. -last gif animation*

# 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 other threads (subset of 4 ?)
3. **No preemption:** a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
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  $T_2$ , ...,  $T_{n-1}$  is waiting for a resource that is held by  $T_n$ , and  $T_n$  is waiting for a resource that is held by  $T_0$ .

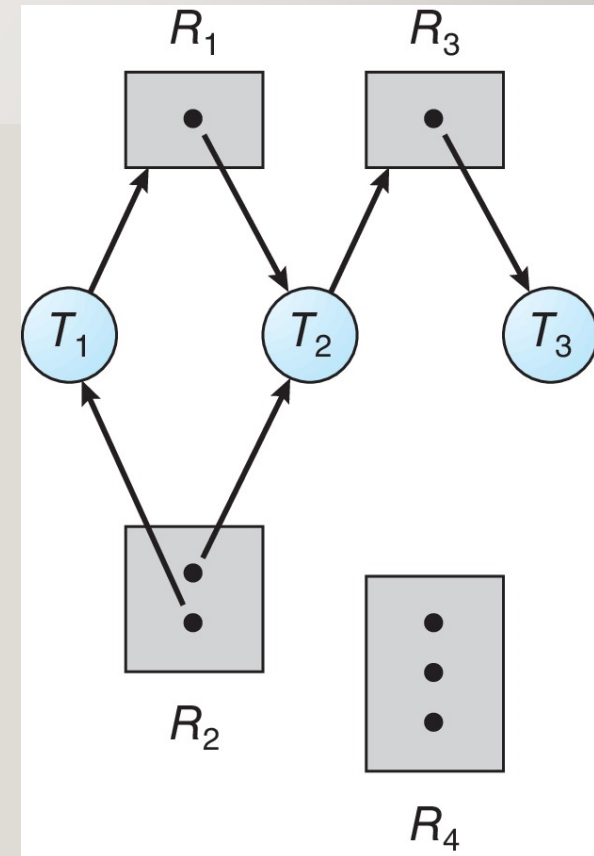


# Resource-allocation Graph

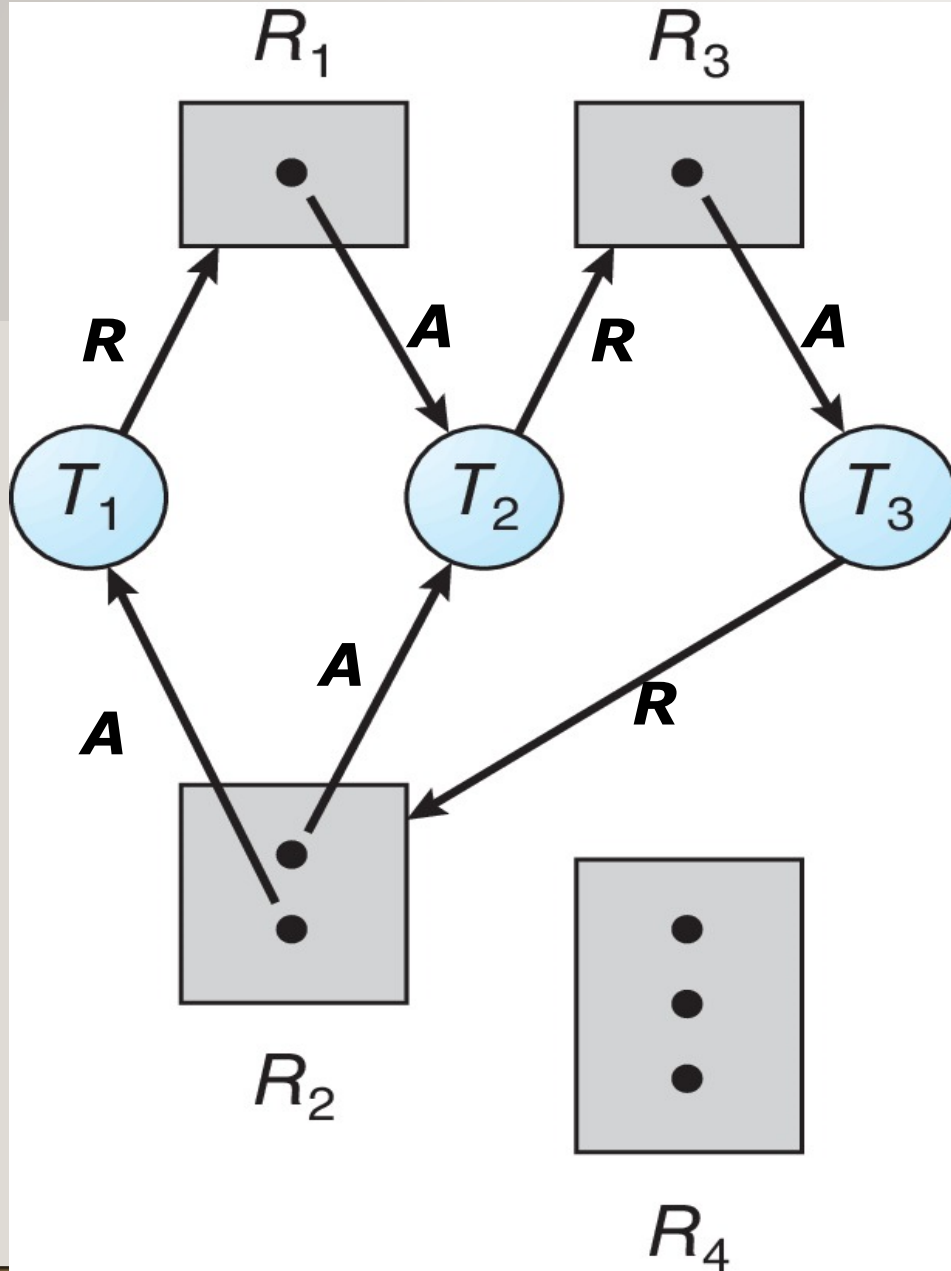
- A set of vertices  $V$  and a set of edges  $E$ .
  - $V$  is partitioned into two types:
    - $T = \{T_1, T_2, \dots, T_n\}$ , the set consisting of all the threads in the system.
    - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
  - **request edge** –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



$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$   
 $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

**Threads  $T_1$ ,  $T_2$ , and  $T_3$  are deadlocked.**

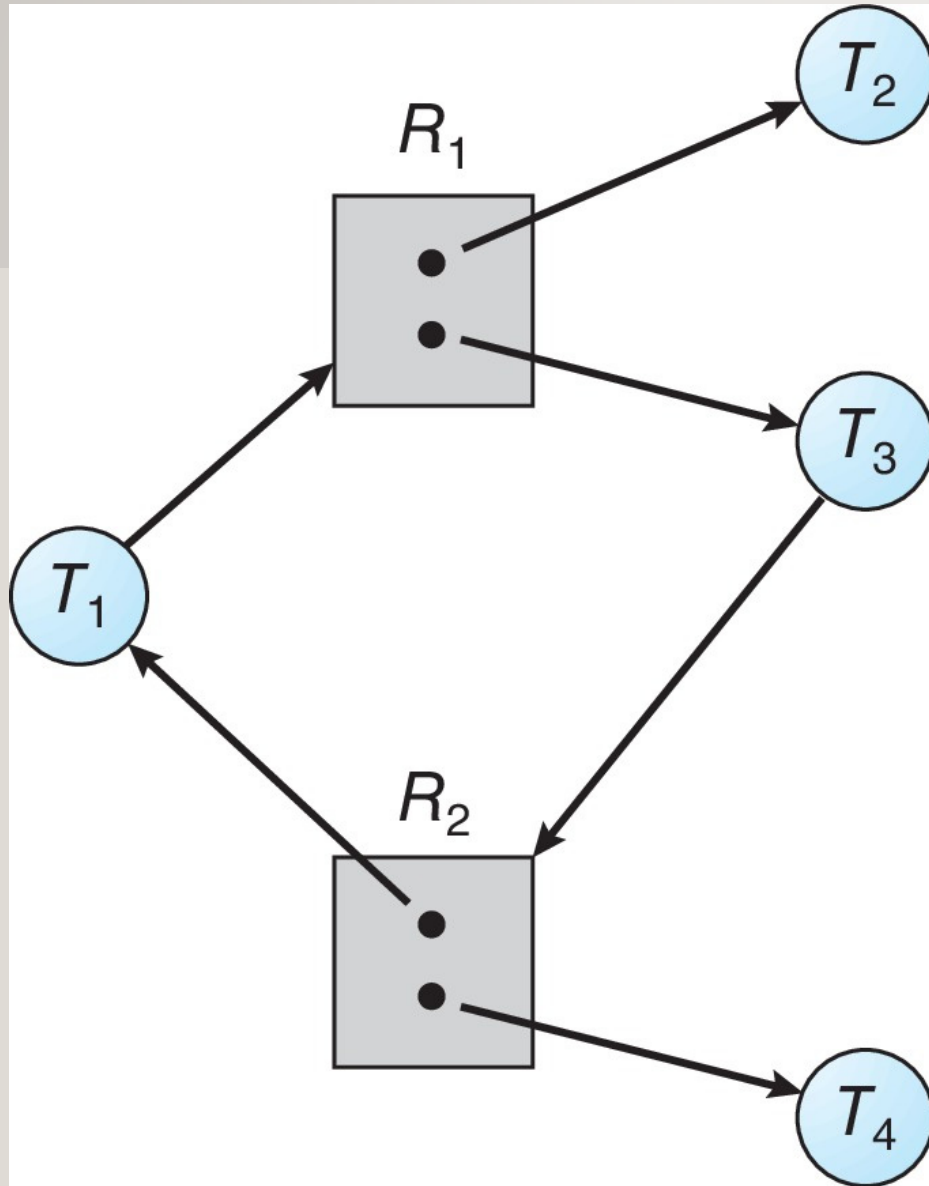
Algm./method For cycle detection in a graph ?

-

Complexity ?

-

# Graph with a Cycle But no Deadlock



**$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$**

**Thread  $T_4$  may release its instance of resource type  $R_2$ .**

**That resource can then be allocated to  $T_3$ , breaking the cycle.**

# Cycle in Graph and 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 (used in kernel, APIs):
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover (Appcln - databases)
- Ignore the problem and pretend that deadlocks never occur in the system (Linux and Win\* - wow).

# 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.)

- **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:

$F(\text{first\_mutex}) = 1$   
 $F(\text{second\_mutex}) = 5$

code for `thread_two` could not be written as follows:

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

```
/* 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

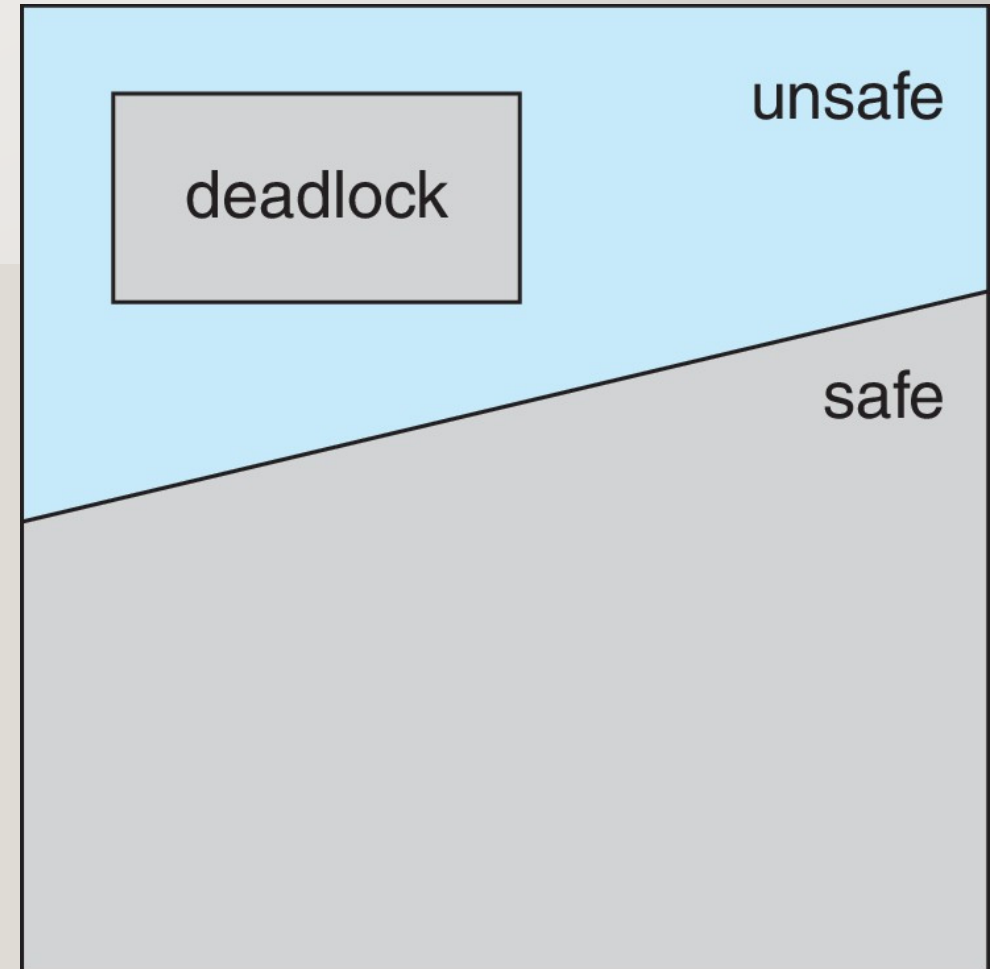
- Requires that the system has some additional **a priori** information available
  - Simplest and most useful model requires that each thread declare the **maximum number** of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state 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  $\langle T_1, T_2, \dots, T_n \rangle$  of ALL the threads 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

# Safe, Unsafe, Deadlock State

- 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.

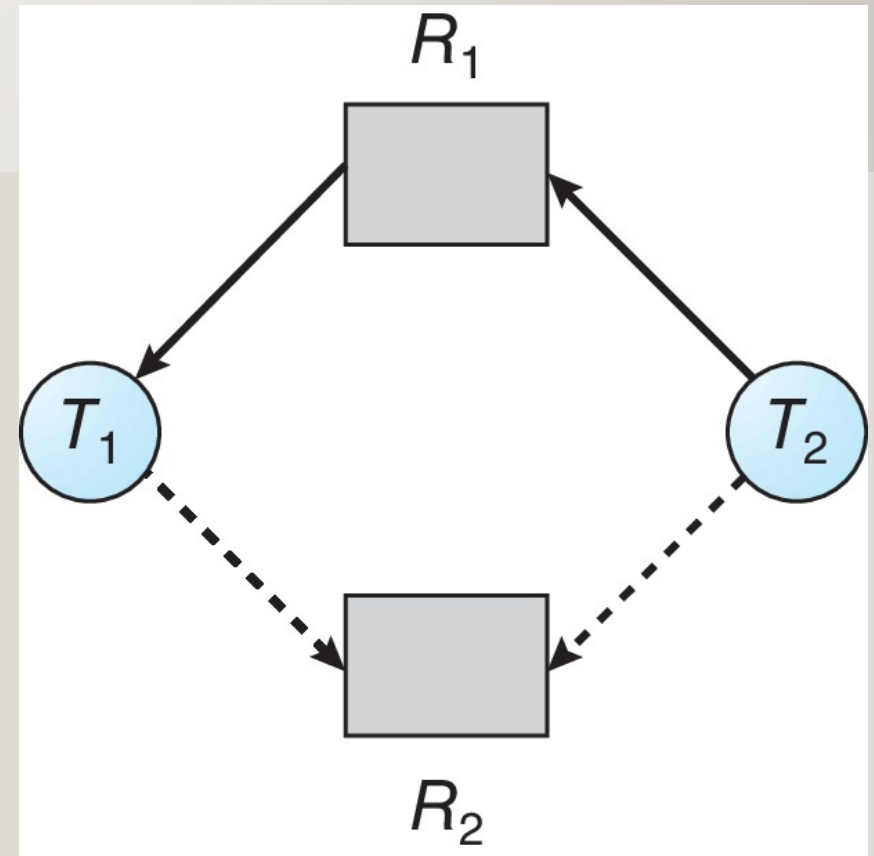


# Avoidance Algorithms

- 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

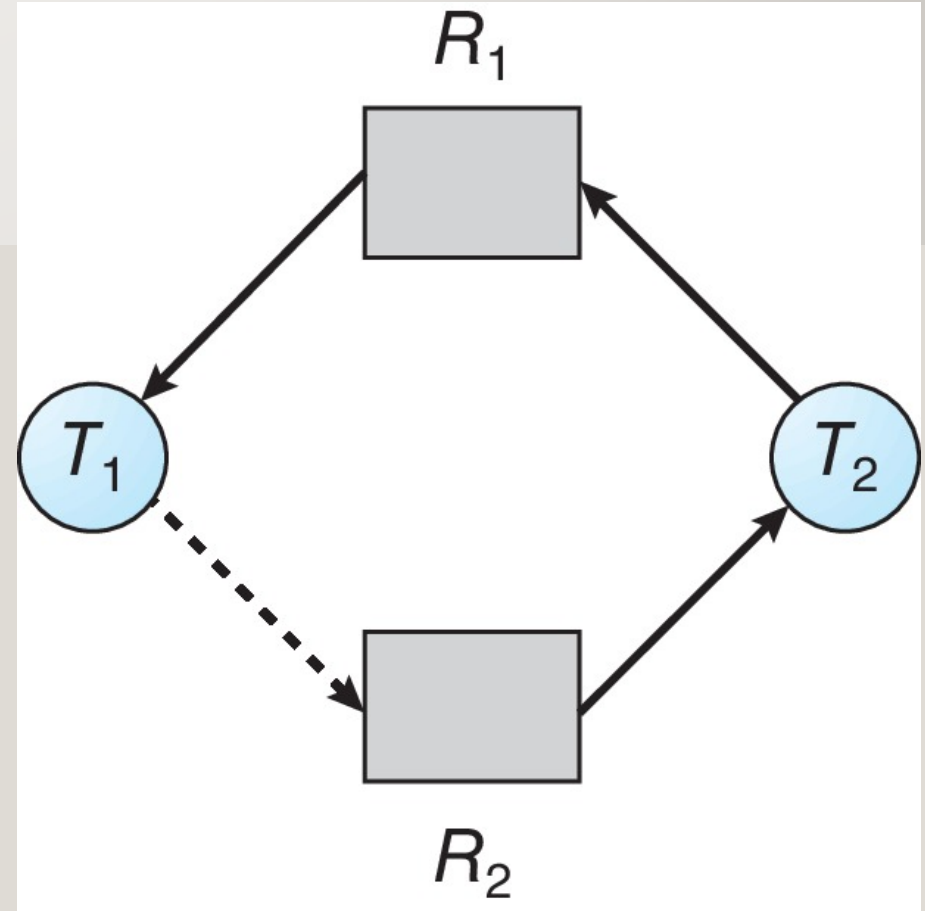


# Unsafe State In Resource-allocation Graph

Suppose that  $T_2$  requests  $R_2$ .

Although  $R_2$  is currently free (see fig. in previous slide), we cannot allocate it to  $T_2$ , since this action will create a cycle in the graph ( See Figure).

A cycle, as mentioned, indicates that the system is in an unsafe state.



# 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
- 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 ( & *no concurrent/parallel threads/processes*)



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

The vector  $Allocation_i$  specifies the resources currently allocated to thread  $T_i$ ; the vector  $Need_i$  specifies the additional resources that thread  $T_i$  may still request to complete its task (*both rows of the corresponding matrices*)

# 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<sub>i</sub> ≤ Work**

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

3. **Work = Work + Allocation<sub>i</sub>**

**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  $Request_i \leq Need_i$  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  $T_i$  must wait, since resources are not available
3. Pretend (*virtually*) to allocate requested resources to  $T_i$  by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe  $\Rightarrow$  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 (5 instances), and C (7 instances)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	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.)

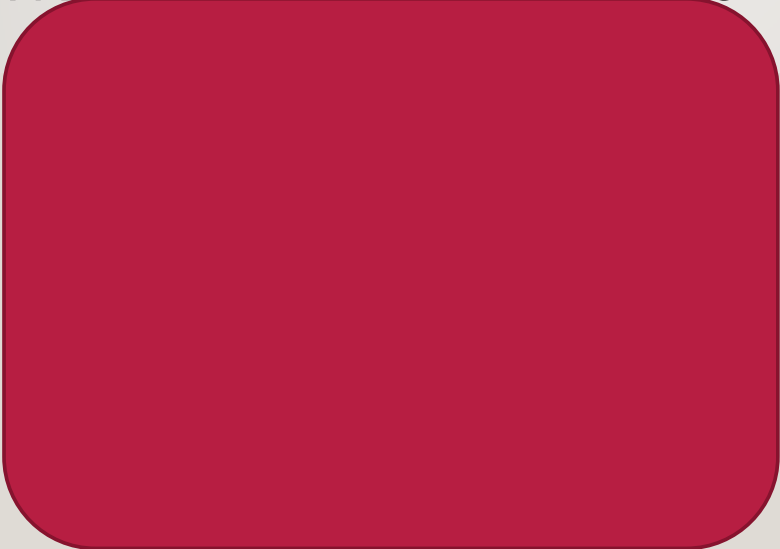
- The content of the matrix **Need** is defined to be **Max – Allocation**

	<u>Need</u>		
	A	B	C
$T_0$			
$T_1$			
$T_2$			
$T_3$			
$T_4$			

- The system is in a safe state since the sequence  $\langle T_1, T_3, T_4, T_2, T_0 \rangle$  satisfies safety criteria

# Example: $P_1$ Request (1,0,2)

- Check that Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow$  true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$T_0$			
$T_1$			
$T_2$			
$T_3$			
$T_4$			

- Executing safety algorithm shows that sequence  $\langle T_1, T_3, T_4, T_0, T_2 \rangle$  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?

# Deadlock Detection

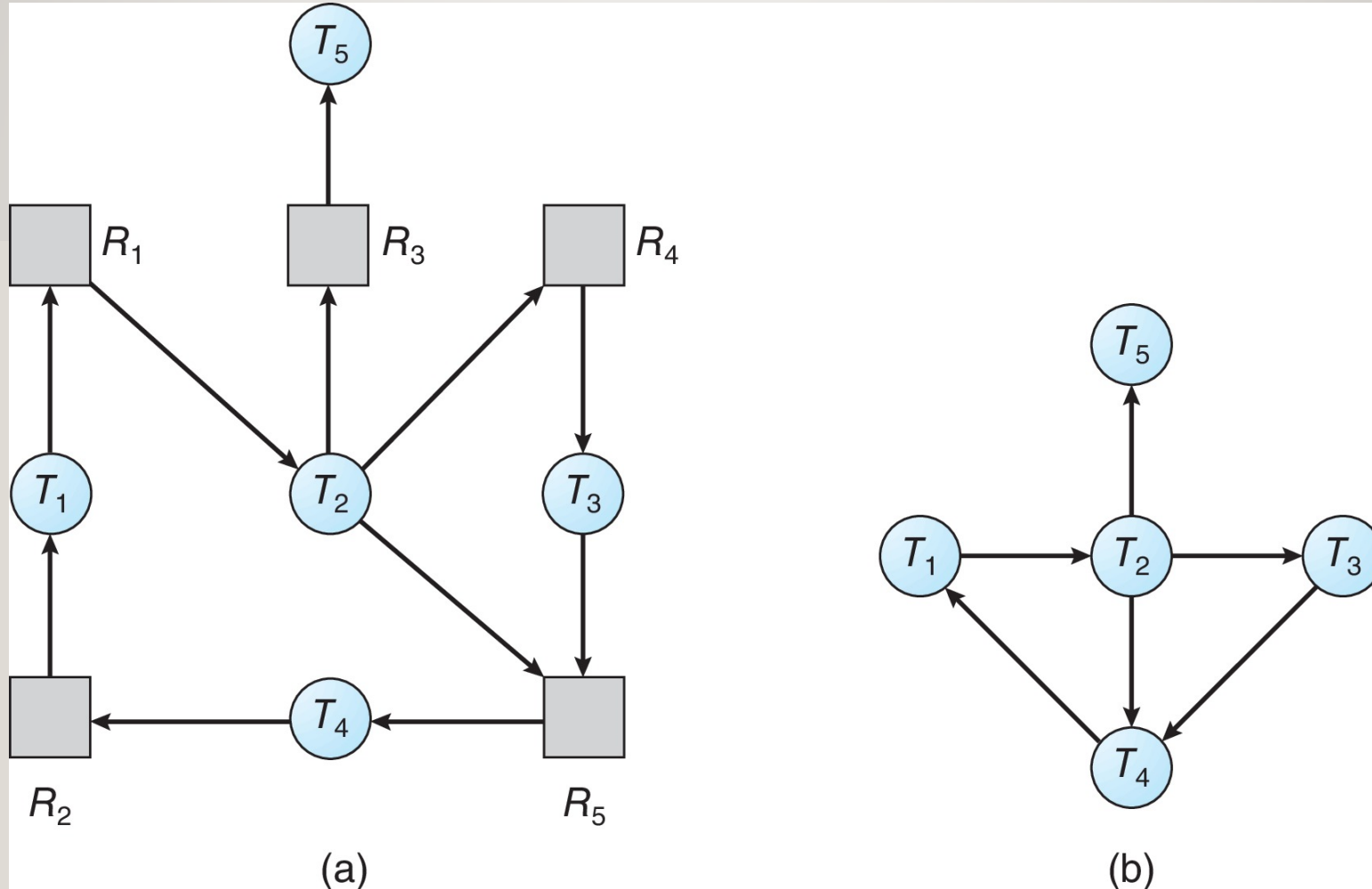
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

# Single Instance Of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are threads
  - $T_i \rightarrow T_j$  if  $T_i$  is waiting for  $T_j$
- Periodically invoke an algorithm that searches for a cycle in the wait 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
- BCC tool *deadlock\_detector* operates by inserting probes which trace calls to the *pthread\_mutex\_lock()* and *pthread\_mutex\_unlock()* functions.



# 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
- **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 **Request**  $[i][j] = k$ , then thread  $T_i$  is requesting  $k$  more instances of resource type  $R_j$ .

# Detection Algorithm

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

**Work = Available**

For  $i = 1, 2, \dots, n$ , if **Allocation<sub>i</sub> ≠ 0**, then

**Finish[i] = false**; otherwise, **Finish[i] = true**

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

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

b) **Request<sub>i</sub> ≤ Work**

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

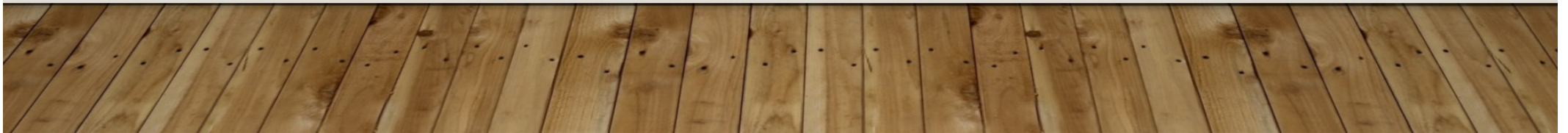
3. **Work = Work + Allocation<sub>i</sub>**

**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<sub>i</sub>** is deadlocked

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



## Detection Algorithm

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

**Work = Available**

For  $i = 1, 2, \dots, n$ , if **Allocation <sub>$i$</sub>   $\neq 0$** , then

**Finish $[i]$  = false**; otherwise, **Finish $[i]$  = true**

2. Find an index  $i$  such that both:

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

b) **Request <sub>$i$</sub>   $\leq$  Work**

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

3. **Work = Work + Allocation <sub>$i$</sub>**

**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.

## 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 index  $i$  such that both:

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

(b) **Need <sub>$i$</sub>   $\leq$  Work**

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

3. **Work = Work + Allocation <sub>$i$</sub>**

**Finish $[i]$  = true**

go to step 2

4. If **Finish  $[i]$  == true** for all  $i$ , then the system **is in a safe state**

# Example Of Detection Algorithm

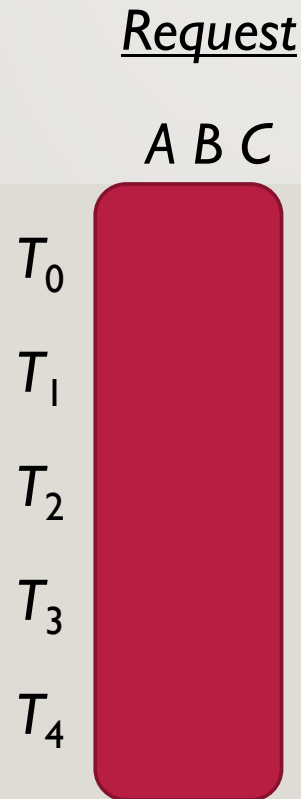
Five threads  $T_0$  through  $T_4$ ; three resource types  
A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time $t_0$	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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  $\langle T_0, T_2, T_3, T_1, T_4 \rangle$  will result in ***Finish[i] = true*** for all  $i$

# Example (Cont.)

- $T_2$  requests an additional instance of type **C**



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

# 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 threads “caused” the deadlock.

# Recovery From Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one process 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
- **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

