



Chapter 6. Deadlocks

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne.
(2018). *Operating System Concepts* (10th ed.). Wiley.

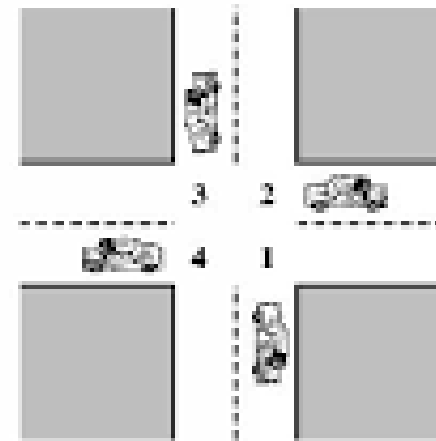
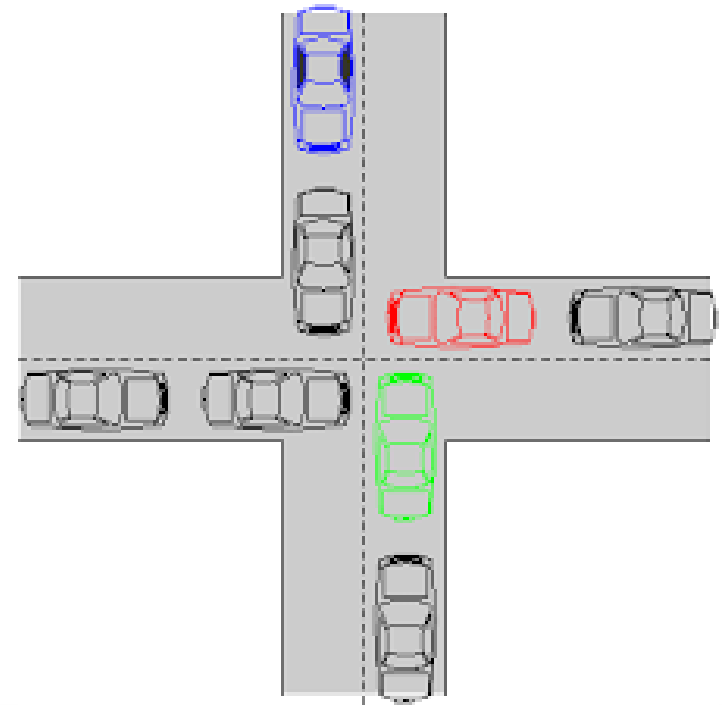


Contents

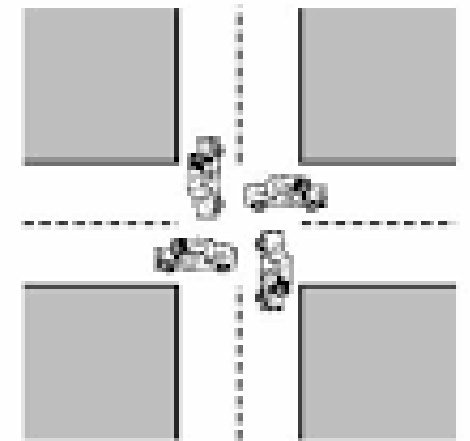
- 1. System Model
- 2. Deadlock Characterization
- 3. Methods for Handling Deadlocks
- 4. Deadlock Prevention
- 5. Deadlock Avoidance
- 6. Deadlock Detection
- 7. Recovery from Deadlock

1. System Model

- In a multiprogramming 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**.



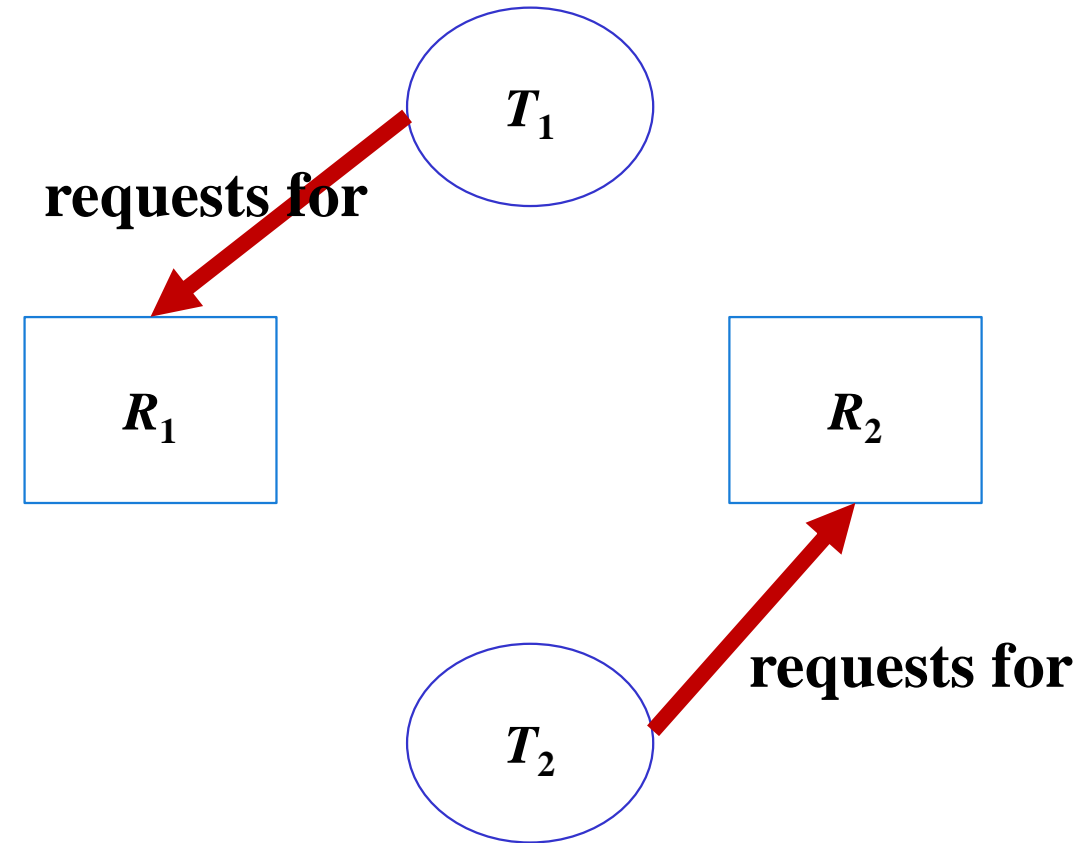
(a) Deadlock possible



(b) Deadlock

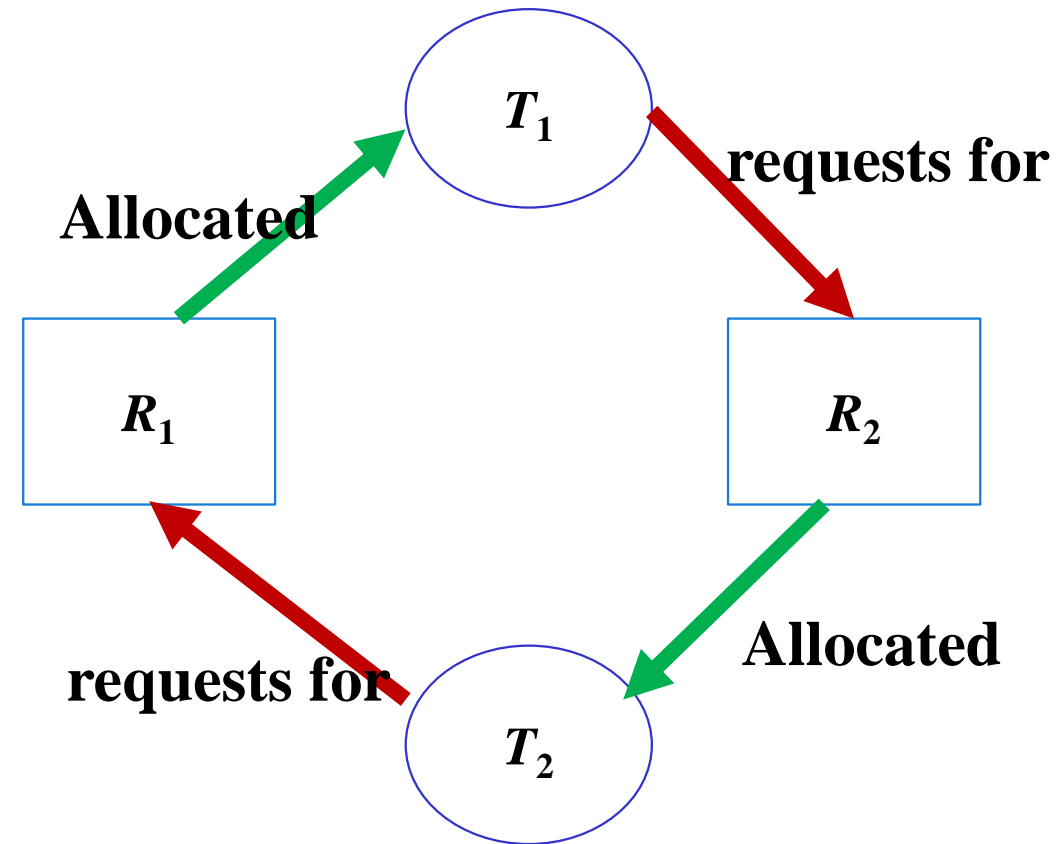
1. System Model

- Two Processes exist : T_1 and T_2
- Two Resources exist : R_1 and R_2
- T_1 requests for R_1
- T_2 requests for R_2



1. System Model

- Two Processes exist : T_1 and T_2
- Two Resources exist : R_1 and R_2
- T_1 requests for R_1
- T_2 requests for R_2
- Since, both the resources are free, so the request is granted.
 - R_1 is allocated towards T_1 and R_2 is allocated towards T_2 .
- Now, T_1 requests for R_2 and, T_2 requests for R_1
- Now, the question is: “Can we grant the request?”
 - No, the request can not be granted
 - Because, R_2 and R_1 are not free.
 - R_2 is held by T_2 and R_1 is held by T_1



This situation indicates that the requests are not be ever fulfilled.

- Both T_1 and P_2 wait for the resources **forever**.
- The system is said to be in “**Deadlock**” state.



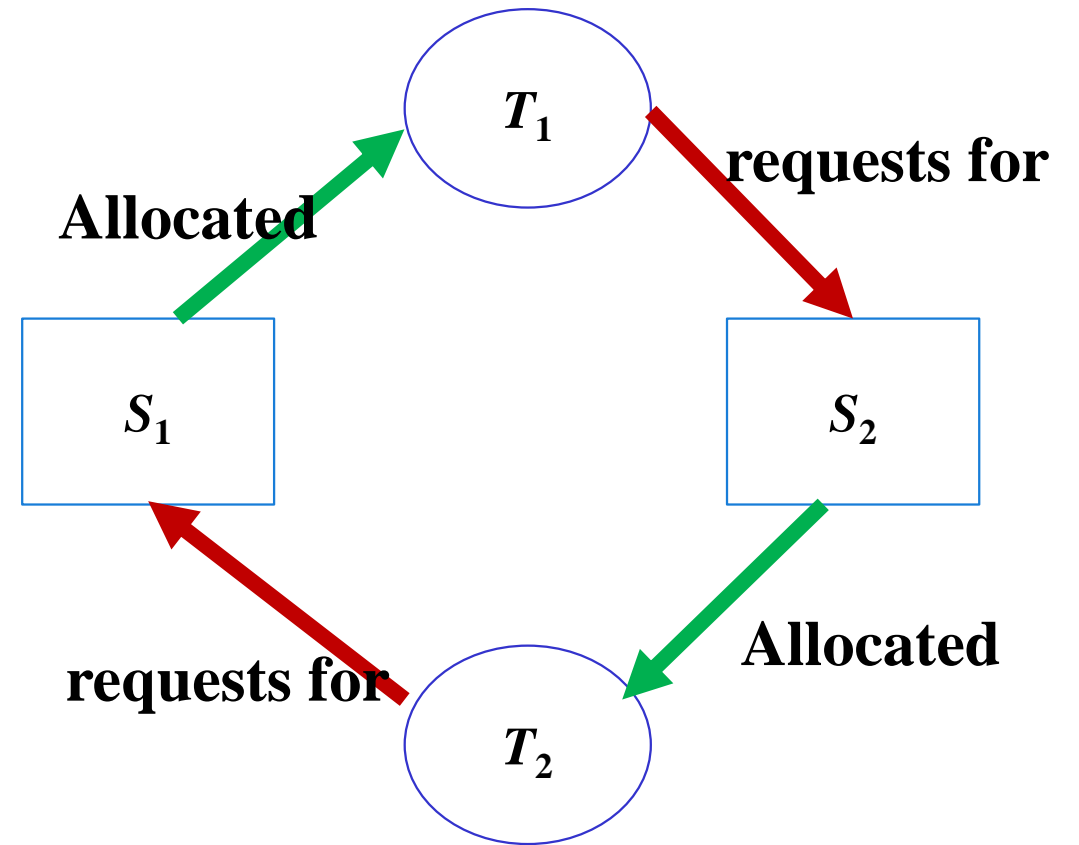
1. System Model

- 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**

1. System Model

Deadlock with Semaphore

- 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)`





1. System Model

Livelock

- **Livelock** is another form of liveness failure. It is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons.
 - Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set.
 - Livelock occurs when a thread continuously attempts an action that fails. Livelock is similar to what sometimes happens when two people attempt to pass in a hallway: One moves to his right, the other to her left, still obstructing each other's progress. Then he moves to his left, and she moves to her right, and so forth. They aren't blocked, but they aren't making any progress.



1. System Model

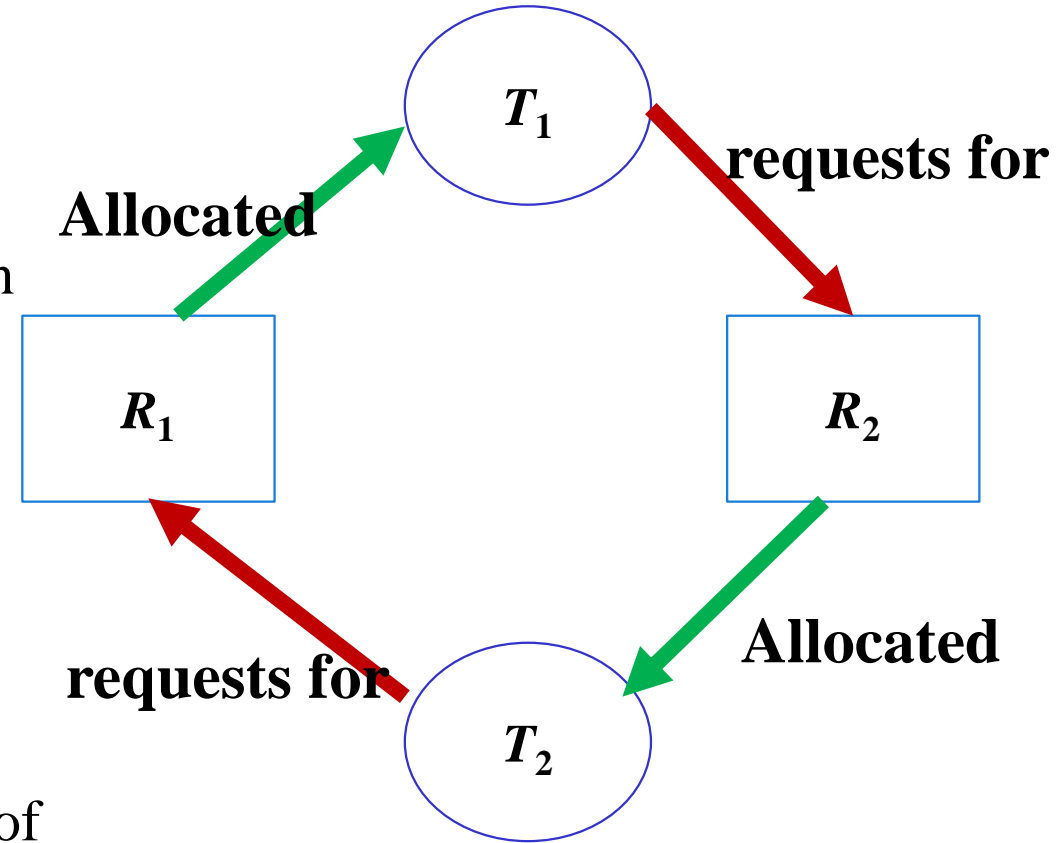
Livelock

- Livelock typically occurs when threads retry failing operations at the same time. **It thus can generally be avoided by having each thread retry the failing operation at random times.** This is precisely the approach taken by Ethernet networks when a network collision occurs. Rather than trying to retransmit a packet immediately after a collision occurs, a host involved in a collision will backoff a random period of time before attempting to transmit again.
- Livelock is less common than deadlock but nonetheless is a challenging issue in designing concurrent applications, and like deadlock, it may only occur under specific scheduling circumstances.

2. Deadlock Characterization



Necessary Conditions

- 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 the process has completed its task
 - **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 .



2. Deadlock Characterization

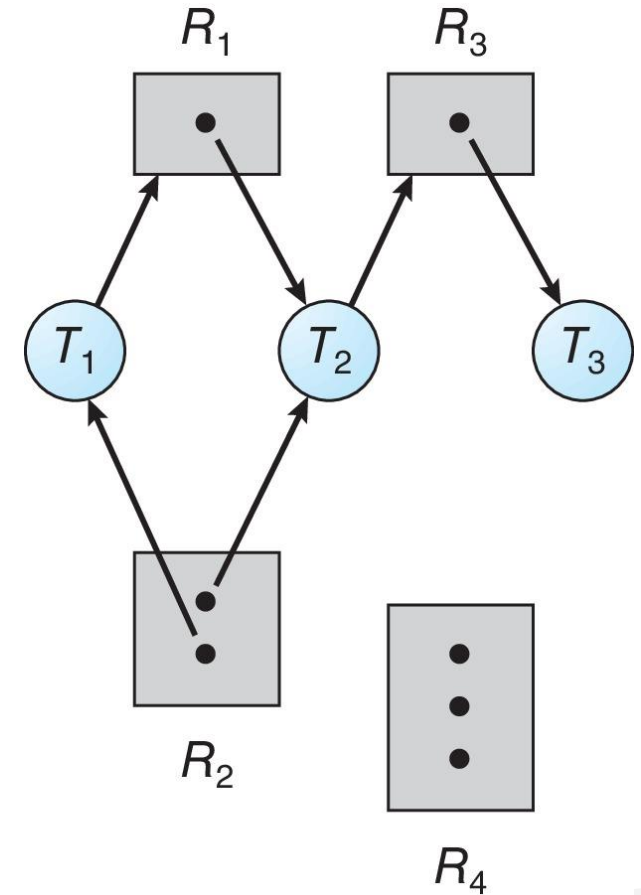
Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.
- The graph consists of 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
- E
 - Request edge: a directed edge from thread T_i to resource type R_j : $T_i \rightarrow R_j$
 - Assignment edge: a directed edge from resource type R_j to thread T_i : $R_j \rightarrow T_i$
- A thread is represented by 
- A resource type is represented by 
- An instance of a resource is represented by a dot: ●

2. Deadlock Characterization

Resource-Allocation Graph Example

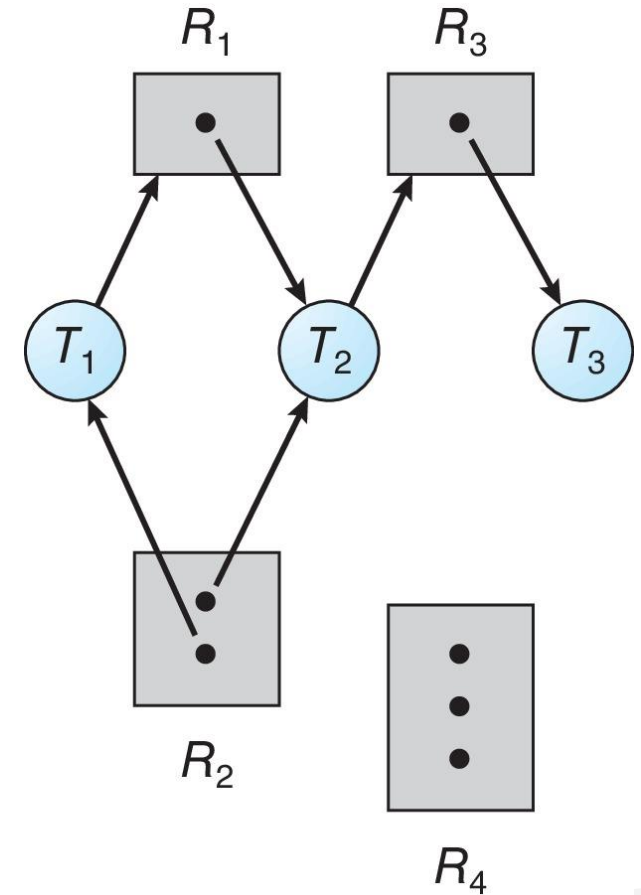
- Resource instances:
 - One instance of R_1
 - Two instances of R_2
 - One instance of R_3
 - Three instance of R_4
- Thread states:
 - T_1 holds one instance of R_2 and is waiting for an instance of R_1
 - T_2 holds one instance of R_1 , one instance of R_2 , and is waiting for an instance of R_3
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



2. Deadlock Characterization

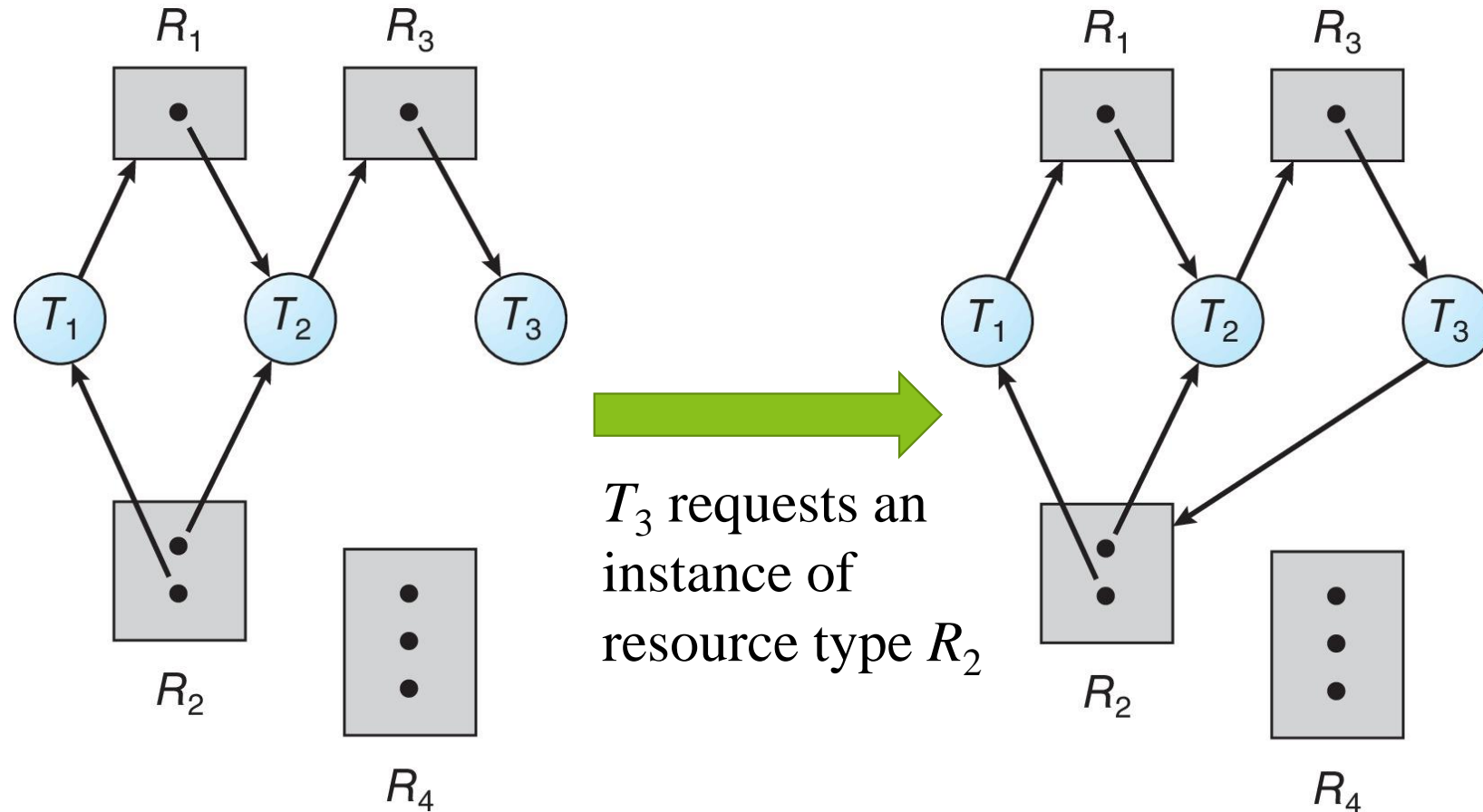
Resource-Allocation Graph Example

- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
 - If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each thread involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.



2. Deadlock Characterization

Resource-Allocation Graph with a Deadlock

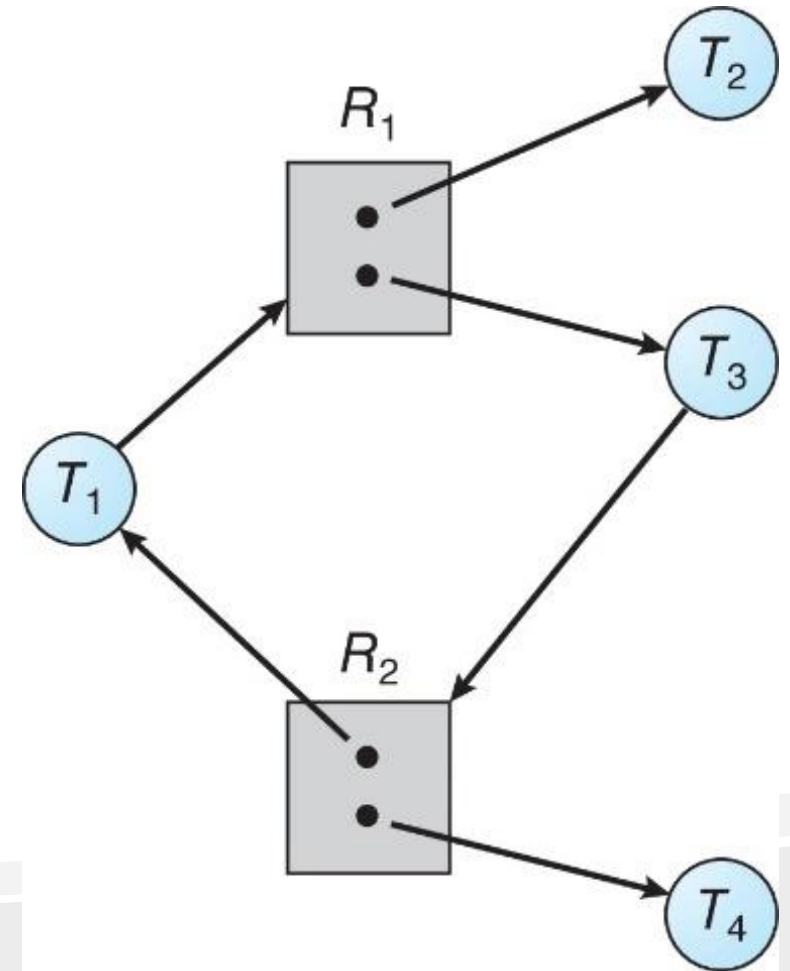


- T_3 requests an instance of resource type R_2 but no resource instance of R_2 is currently available.
 - At this point, two minimal cycles exist:
 $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
 $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$
 - Threads T_1 , T_2 , and T_3 are deadlocked.

2. Deadlock Characterization

Resource-Allocation Graph with a Deadlock ?

- In this example, we also have a cycle:
 - $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
 - There is no deadlock.
 - Observe that thread T_4 may release its instance of resource type R_2 . That resource can then be allocated to T_3 , breaking the cycle.





2. Deadlock Characterization

Resource-Allocation Graph

- 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



3. Methods for Handling Deadlocks

- Generally speaking, we can deal with the deadlock problem in one of three ways:
 - We can ignore the problem altogether and pretend that deadlocks never occur in the system.
 - We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
 - **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
 - **Deadlock avoidance** requires that the operating system be given **additional information** in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the thread should wait.
 - We can allow the system to enter a deadlocked state, detect it, and recover.
 - **Deadlock Detection**
 - **Recovery from deadlock**



4. 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
 - impractical for most applications due to the dynamic nature of requesting resources.
 - Allow thread to request resources only when the thread has **none** allocated to it.
 - Before it can request any additional resources, it must release all the resources that it is currently allocated.
 - Disadvantage: Low resource utilization; starvation possible



4. Deadlock Prevention

— 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.
 - In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the thread is waiting.
 - The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Alternatively, if a thread requests some resources, we first check whether they are available.
 - If they are, we allocate them.
 - If they are not, we check whether they are allocated to some other thread that is waiting for additional resources.
 - If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. If the resources are neither available nor held by a waiting thread, the requesting thread must wait.
 - This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions.



4. Deadlock Prevention

— Circular Wait:

- The three options presented thus far for deadlock prevention are generally impractical in most situations.
- One way to ensure that this condition never holds is to impose a **total ordering of all resource types**
 - We can now consider the following protocol to prevent deadlocks: Each thread can request resources only in an increasing order of enumeration.
 - Alternatively, we can require that a thread requesting an instance of resource R_j must have released any resources R_i which succeed R_j .



4. Deadlock Prevention

- 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` has assigned 1 and `second_mutex` has assigned 5:

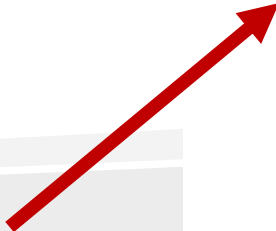
code for `thread_two` **could not** be written as follows:

```
/* 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);
}
```





5. Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
 - Simplest and most useful model requires that each thread declares 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**

5. Deadlock Avoidance

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 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
- If no such sequence exists, then the system state is said to be **unsafe**.

5. Deadlock Avoidance

Safe State

- To illustrate, consider a system with **twelve** resources and three threads: T_0 , T_1 , and T_2 .
- Thread T_0 requires ten resources, thread T_1 may need as many as four, and thread T_2 may need up to nine resources.
- Suppose that, at time t_0 , thread T_0 is holding five resources, thread T_1 is holding two resources, and thread T_2 is holding two resources. (Thus, there are **three** free resources.)

	<u>Maximum Needs</u>	<u>Current Needs</u>
T_0	10	5
T_1	4	2
T_2	9	2

At time t_0 , the system is in a safe state. The sequence $\langle T_1, T_0, T_2 \rangle$ satisfies the safety condition.

5. Deadlock Avoidance

Safe State

- A system can go from a safe state to an unsafe state. Suppose that, at time t_1 , thread T_2 requests and is allocated one more resource. Thus, there are **two** free resources. **The system is no longer in a safe state.**

	<u>Maximum Needs</u>	<u>Current Needs</u>
T_0	10	5
T_1	4	2
T_2	9	3

- At this point, only thread T_1 can be allocated all its resources.



5. Deadlock Avoidance

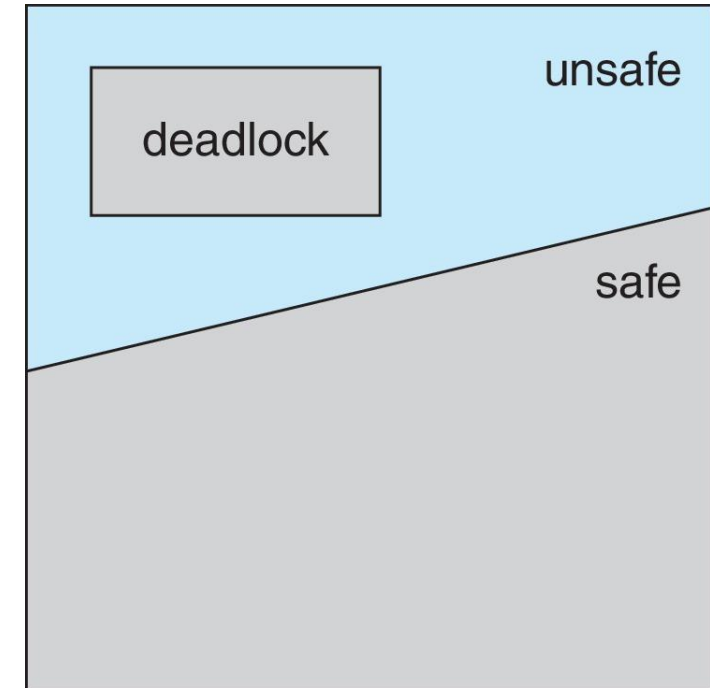
Safe State

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
- The idea is simply to ensure **that the system will always remain in a safe state.**
 - Initially, the system is in a safe state.
 - Whenever a thread requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or the thread must wait.
 - The request is granted only if the allocation leaves the system in a safe state.

5. Deadlock Avoidance

Safe, Unsafe, Deadlock State

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however.
 - An unsafe state may lead to a deadlock.
 - As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.
 - In an unsafe state, the operating system cannot prevent threads from requesting resources in such a way that a deadlock occurs. The behavior of the threads controls unsafe states.
- 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.**





5. Deadlock Avoidance

Avoidance Algorithms

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

5. Deadlock Avoidance

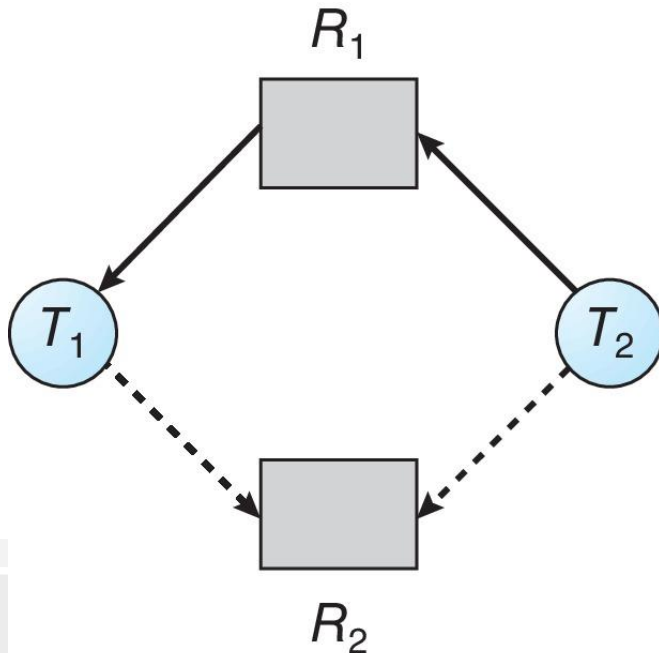
Resource-Allocation Graph Scheme

- **Claim edge** $T_i \rightarrow R_j$ indicated that process T_i **may** request resource R_j at some time in the future; represented by a dashed line.
- The claim edge converts to a request edge when a thread requests a resource.
- The request edge converted to an assignment edge when the resource is allocated to the thread.
- When a resource is released by a thread, the assignment edge reconverts to a claim edge.
- **Resources must be claimed a *priori* in the system.**
 - That is, before thread T_i starts executing, all its claim edges must already appear in the resource-allocation graph.

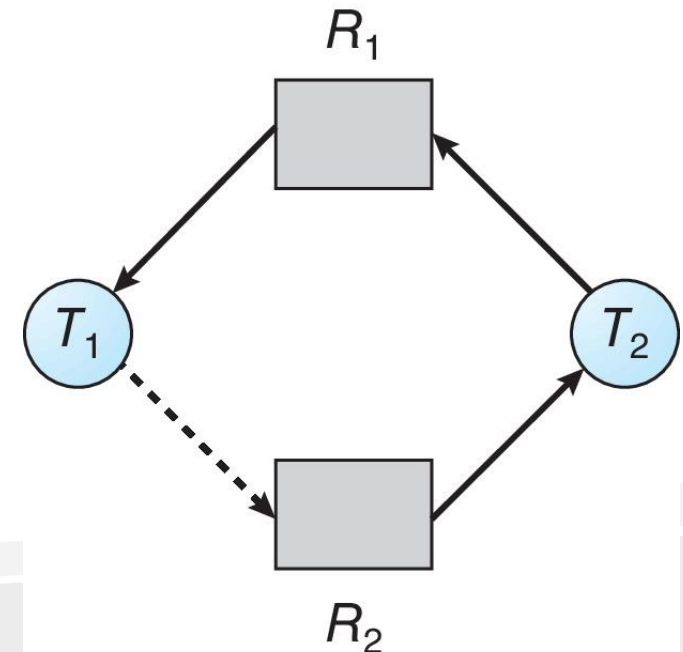
5. Deadlock Avoidance

Resource-Allocation Graph Scheme

- 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
 - An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of threads in the system.



Suppose that T_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to T_2 , since this action will create a cycle in the graph.



An unsafe state in a resource-allocation graph



5. Deadlock Avoidance

Banker's Algorithm

- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- Multiple instances of resources
- Each thread must declare a priori claim maximum use.
 - When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need. This 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.



5. Deadlock Avoidance

Data Structures for the Banker's Algorithm

- Let n be the number of threads, and m be the 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.

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$



5. Deadlock Avoidance

Safety Algorithm

- To find out whether or not a system is in a safe state

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.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.



5. Deadlock Avoidance

Resource-Request Algorithm for Thread T_i

- Let $\mathbf{Request}_i$ be the request vector for thread T_i . If $\mathbf{Request}_i[j] == k$, then thread T_i wants k instances of resource type R_j .
- When a request for resources is made by thread T_i , the following actions are taken:
 1. If $\mathbf{Request}_i \leq \mathbf{Need}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
 2. If $\mathbf{Request}_i \leq \mathbf{Available}$, 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:
 - $\mathbf{Available} = \mathbf{Available} - \mathbf{Request}_i$;
 - $\mathbf{Allocation}_i = \mathbf{Allocation}_i + \mathbf{Request}_i$;
 - $\mathbf{Need}_i = \mathbf{Need}_i - \mathbf{Request}_i$;

If the new state is safe \rightarrow the resources are allocated to T_i

If the new state is unsafe $\rightarrow T_i$ must wait, and the old resource-allocation state is restored

5. Deadlock Avoidance

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

5. Deadlock Avoidance

Example of Banker's Algorithm

The content of the matrix *Need* is defined to be

$$\text{Need} = \text{Max} - \text{Allocation}$$

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

5. Deadlock Avoidance

Example of Banker's Algorithm

Determine the sequence of processes that may result in a safe state

Work	Need(i)	P(i)	Allocation
A B C	A B C		A B C
3 3 2	1 2 2	P1	2 0 0
5 3 2	0 1 1	P3	2 1 1
7 4 3	4 3 1	P4	0 0 2
7 4 5	6 0 0	P2	3 0 2
10 4 7	7 4 3	P0	0 1 0

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

5. Deadlock Avoidance

Example of Banker's Algorithm

- P1 requests (1,0,2)

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

Can request for (3,3,0) by P_4 be granted? No

Can request for (0,2,0) by P_0 be granted? No



6. Deadlock Detection

- Generally speaking, we can deal with the deadlock problem in one of three ways:
 - We can ignore the problem altogether and pretend that deadlocks never occur in the system.
 - We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never *enter* a deadlocked state.
 - **Deadlock prevention**
 - **Deadlock avoidance**
-  — We can allow the system to enter a deadlocked state, detect it, and recover.
 - **Deadlock Detection**
 - **Recovery from deadlock**



6. Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:
 - An algorithm that examines the state of the system to determine whether a deadlock has occurred
 - An algorithm to recover from the deadlock
- We discuss these two requirements
 - a single instance of each resource type
 - several instances of each resource type.



6. Deadlock Detection

Single Instance of Each Resource Type

- Uses a variant of the resource-allocation graph, called a **wait-for** graph.
 - We obtain the wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- Maintain wait-for graph
 - Nodes are threads
 - An edge $T_i \rightarrow T_j$ implies thread T_i is waiting for thread T_j to release a resource that T_i needs.
- 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.

6. Deadlock Detection

Single Instance of Each Resource Type

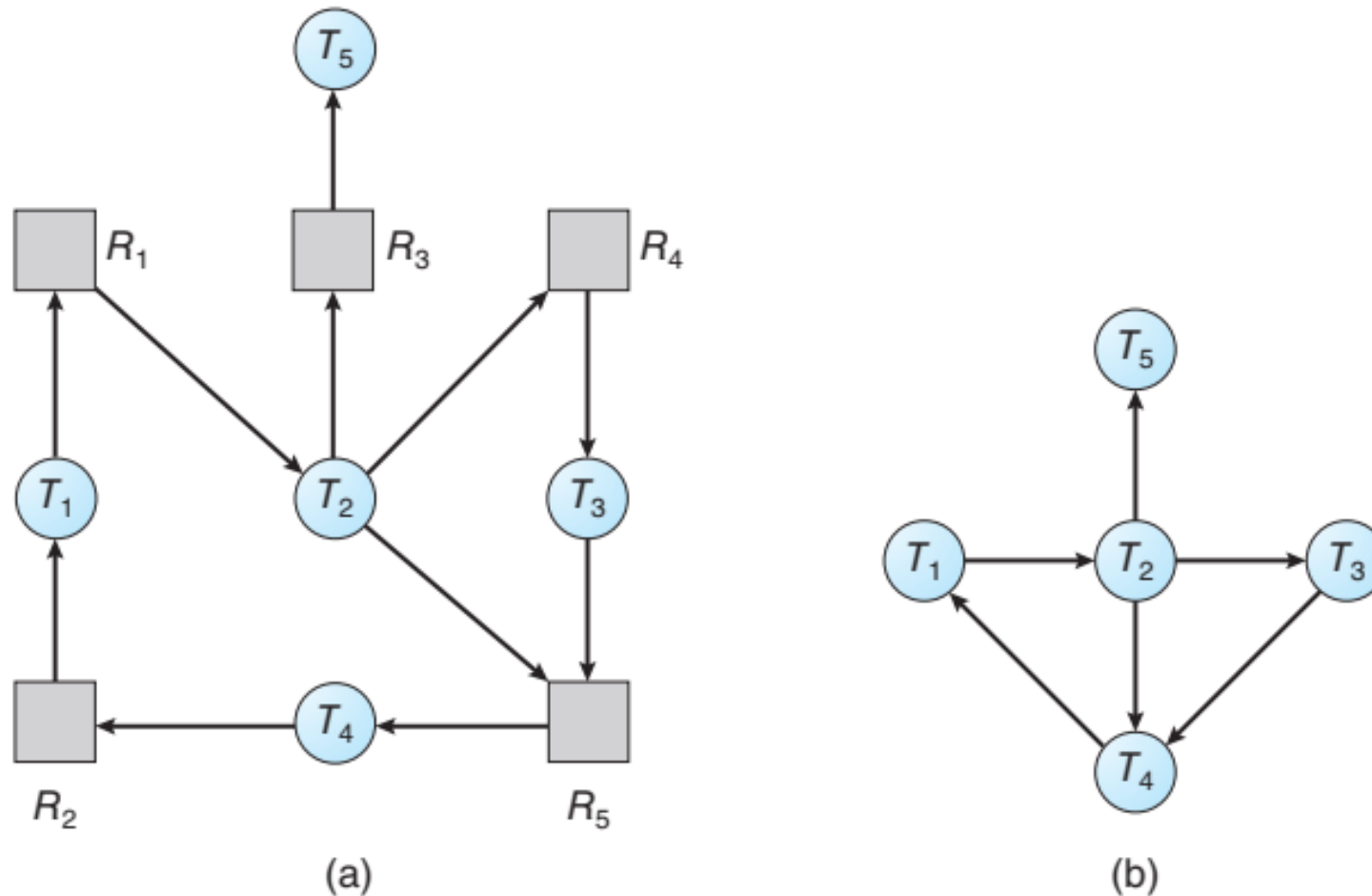


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.



6. Deadlock Detection

Several Instances of a Resource Type

- The deadlock detection algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.
- **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 .

6. Deadlock Detection

Several Instances of a Resource Type Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively. Initialize:
 - *Work* = *Available*
 - For $i = 0, 1, \dots, n - 1$, 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
3. $Work = Work + Allocation_i$
 $Finish[i] = true$
go to step 2
4. If $Finish[i] == false$, for some i , $0 \leq i < n$, then the system is in a deadlock state.
Moreover, if $Finish[i] == false$, then T_i is deadlocked.

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

6. Deadlock Detection

Several Instances of a Resource Type 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 .

6. Deadlock Detection

Several Instances of a Resource Type Example of Detection Algorithm

- T_2 requests an additional instance of type C.

	<u>Request</u>		<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C		A B C	A B C	A B C
T_0	0 0 0		T_0	0 0 0	0 0 0
T_1	2 0 2		T_1	2 0 2	
T_2	0 0 1		T_2	0 0 1	
T_3	1 0 0		T_3	1 0 0	
T_4	0 0 2		T_4	0 0 2	

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



7. Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
 - One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
 - Another possibility is to let the system **recover** from the deadlock automatically.
- There are two options for breaking a deadlock.
 - One is simply to abort one or more threads to break the circular wait.
 - The other is to preempt some resources from one or more of the deadlocked threads.



7. Recovery from Deadlock

Process & Thread 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? The question is basically an economic one; we should abort those processes whose termination will incur the *minimum cost*.
 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?



7. 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