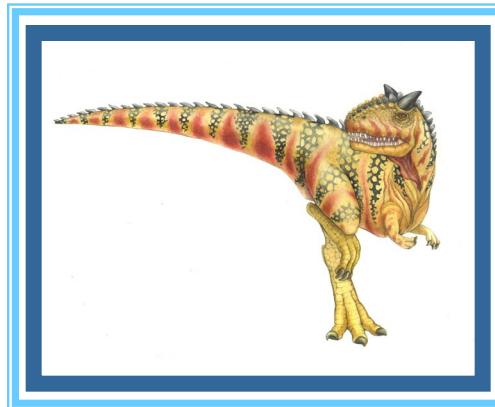


Chapter 8: Deadlock





Introduction - Deadlock

- In a **multiprogramming** environment, several *processes* may compete for a finite number of resources.
- Similarly, in a **multiprocessing** 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**.





Introduction

- There, we defined **deadlock** as a situation in which *every thread/process in an application is waiting for an event that can be caused only by another thread/process in the thread/process set.*
- This chapter describes methods that application developers as well as operating-system programmers can use to prevent or deal with **deadlocks**.
 - Although some applications can identify programs that may **deadlock**, operating systems typically *do not provide deadlock-prevention facilities*, and it remains the responsibility of programmers to ensure that they design **deadlock-free** programs.





System Model

- A **multiprocessing system** consists of a finite number of **resources** to be distributed among several competing threads.
- The **resources** may be partitioned into several types, each consisting of some number of identical instances.
 - CPU cycles, files, and I/O devices (such as printers, DVD drives, etc) are examples of resource types.
 - ▶ If a system has **four CPUs**, then the **resource type CPU has four instances**. Similarly, the resource type **network** may have **two instances**.
 - ▶ If a thread requests an instance of a resource type, the allocation of **any** instance of the type should satisfy the request.
 - .





System Model

- Note that this chapter discusses *kernel resources*, but threads may use resources from other processes (for example, via *inter-process communication*), and those resource uses can also cause **deadlock**.
- A thread must *request* a resource before using it and must *release* the resource after using it.
- A thread may *request* as many resources as it requires to carry out its designated task.
- Obviously, the number of ***requested resources*** may not exceed the total available in the system (***Request ≤ Resources***).
 - In other words, a *thread cannot request two network interfaces if the system has only one*.





System Model

- Under the normal mode of operation, a thread may utilize a resource in only the following sequence:
 - **Request:** The thread requests the resource.
 - ▶ *If the request cannot be granted immediately (for example, if a mutex lock is currently held by another thread), then the requesting thread must **wait** until it can acquire the resource.*
 - **Use:** The thread can operate on the resource
 - ▶ For example, if the resource is a **mutex lock**, the thread can access its **critical section (CS)**.
 - **Release:** The thread releases the resource.





System Model

- The ***request*** and ***release*** of resources are **system calls**.
 - Examples include the ***request()*** and ***release()*** of a device, the ***open()*** and ***close()*** of a file, and the ***allocate()*** and ***free()*** of memory; these are system calls.
 - Similarly, the ***wait()*** and ***signal()*** operations on **semaphores** and ***acquire()*** and ***release()*** of a ***mutex lock*** are system calls.
- For each use of a ***kernel-managed resource*** by a **thread**, the OS checks to make sure that the thread's ***resource request*** has been allocated.

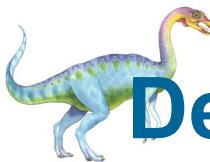




System Model

- A **system table** records whether each **resource** is free or allocated.
- For each allocated resource, the table also records the thread to which it is allocated.
- If a **thread requests a resource** that is currently allocated to another thread, it can be added to a **waiting-queue** of threads for this resource.
- 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 Applications

- To illustrate how deadlock can occur in a multithreaded **Pthread** program using **mutex locks**.
 - A **pthread (POSIX thread)** is a standardized programming interface (API) for creating and managing threads within a **single process**, particularly on **Unix-like operating systems**.
- The **`pthread_mutex_init()`** function initializes an unlocked mutex.
- **Mutex locks** are **acquired** and **released** using **`pthread_mutex_lock()`** and **`pthread_mutex_unlock()`**, respectively.
- If a thread attempts to acquire a **locked mutex**, the call to **`pthread_mutex_lock()`** blocks the thread until the owner of the mutex lock invokes **`pthread_mutex_unlock()`**.





Deadlock in Multithreaded Applications

- Assume that **two mutex locks, first_mutex and second_mutex** are created and initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

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

- Two threads—**thread_one** and **thread_two**—are created, and both of these threads have access to both mutex locks.
- The **thread_one** and **thread_two** run in the functions **do_work_one()** and **do_work_two()**, respectively, as shown in **Figure 8.1**.





Deadlock in Multithreaded Applications

```
/* 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); }
```

Figure 8.1 A deadlock example





Deadlock in Multithreaded Applications

- In the **deadlock** example (shown in **Figure 8.1**):
 - **thread_one** attempts to acquire the **mutex locks** in the order (1) **first_mutex**, (2) **second_mutex**.
 - At the same time, **thread_two** attempts to acquire the **mutex locks** in the order (1) **second_mutex**, (2) **first_mutex**.
- **Deadlock** is possible if **thread_one** acquires only the **first_mutex** while **thread_two** acquires only the **second_mutex**.
- **Note that** a **deadlock** 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 mutex locks.





Deadlock in Multithreaded Applications

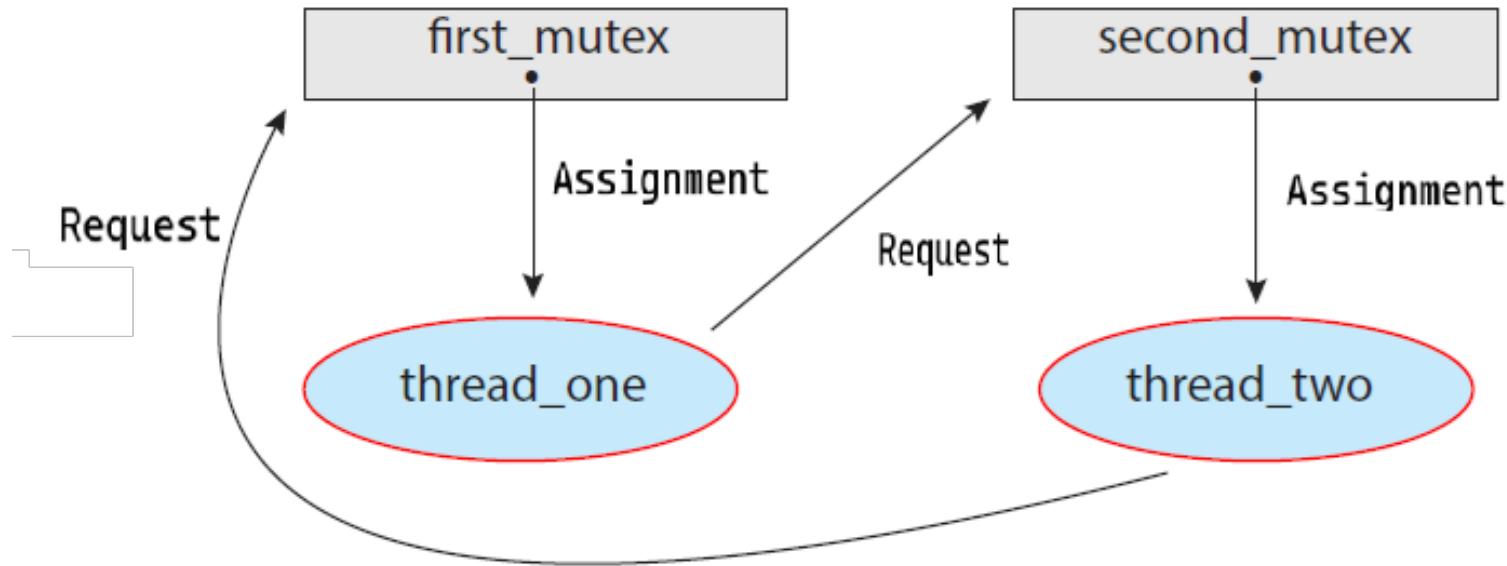


Figure 8.3 Resource-allocation graph for program in Figure 8.1.





Deadlock Conditions

- A **deadlock situation** can arise if the following **four conditions** hold simultaneously in a system:
 - **Mutual exclusion:** *only one thread at a time can use a resource.* If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
 - **Hold and wait:** A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
 - **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread after its completion of the 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 .

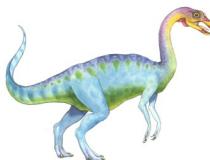




Deadlock with Mutex Locks

- We emphasize that all **four conditions** must hold together for a deadlock to occur.
- The **circular-wait condition** implies the **hold-and-wait condition**, so the **four conditions** are not completely independent.





Resource-Allocation Graph

- **Deadlocks** can be described more precisely in terms of a directed graph called a **resource-allocation graph**.
- This graph consists of a set of vertices V and a set of edges E .
- *The set of vertices V is partitioned into two different types of nodes: $T = \{T_1, T_2, \dots, T_n\}$, the set consisting of all the active **threads** in the set, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all **resource** types in the set.*





Resource-Allocation Graph

- A **directed edge** from thread T_i to resource type R_j is denoted by $T_i \rightarrow R_j$ and is called a request edge,
 - It signifies that thread T_i has **requested** an instance of resource type R_j and is currently waiting for that resource.
- A **directed edge** from resource type R_j to thread T_i is denoted by $R_j \rightarrow T_i$, is called an assignment edge,
 - It signifies that an instance of resource type R_j has been allocated to thread T_i .





Resource-Allocation Graph

- The graph has 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

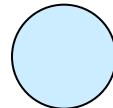
- Pictorially, we represent each thread T_i as a **circle** and each **resource** R_j as a **rectangle**.
- Since resource type R_j may have more than one instance, we represent each such instance as a **dot within the rectangle**.
- Note that a **request edge** points to only the rectangle R_j , whereas an **assignment edge** must also designate one of the dots in the rectangle.



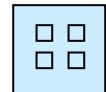


Resource-Allocation Graph

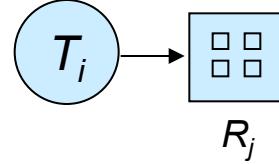
- Thread



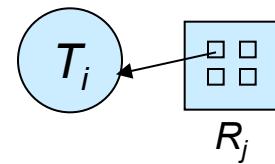
- Resource Type with 4 instances

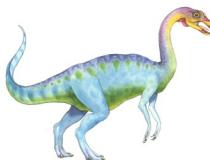


- T_i requests an instance of R_j



- T_i is holding (assigned) an instance of R_j





Resource-Allocation Graph

- When a thread T_i requests an instance of resource type R_j , a request edge is inserted in the **resource-allocation graph**.
- When this request can be fulfilled, the **request edge is instantaneously transformed to an assignment edge**.
- **When** the thread no longer needs access to the resource, it **releases** the resource.
- As a result, the **assignment edge is deleted**.





Example of a Resource Allocation Graph

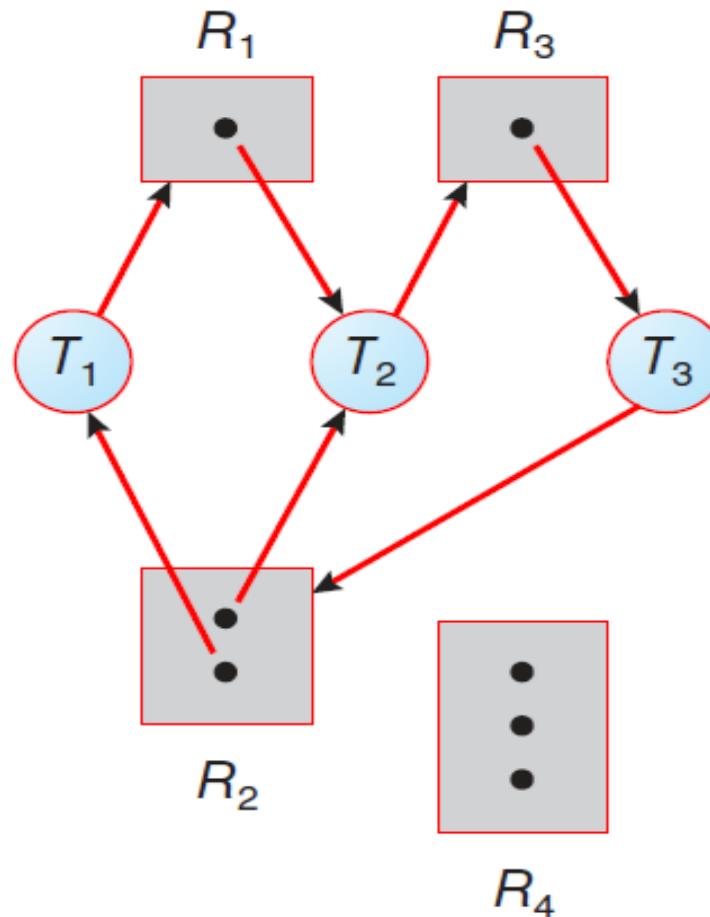


Figure 8.5 Resource-allocation graph with a deadlock.





A Resource Allocation Graph Figure 7.1

- The **resource-allocation graph** shown in **Figure 7.1** depicts the following situation.
- The sets **T , R , and E** :
 - $T = \{T_1, T_2, T_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$





Resource Allocation Graph

- Given the definition of a **resource-allocation graph**, it can be shown that;
- *if the graph contains no cycles, then no process in the system is deadlocked.*
- *If the graph does contain a cycle, then a deadlock may exist.*





Resource Allocation Graph

- 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 process 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.

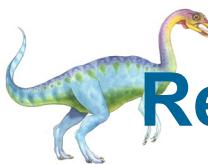




Resource Allocation Graph

- 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.
- Suppose that process T_3 requests an instance of resource type R_2 .
- Since no resource instance is currently available, we add a request edge $T_3 \rightarrow R_2$ to the graph (**Figure 8.6**). At this point, two minimal cycles exist in the system:





Resource Allocation Graph With A Deadlock

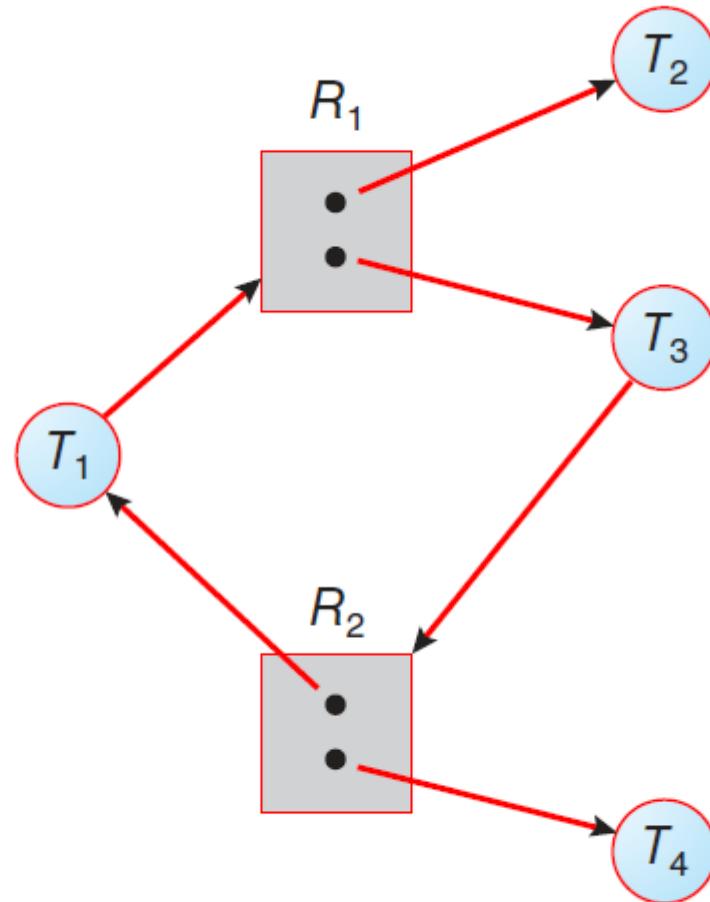


Figure 8.6 Resource-allocation graph with a cycle but no deadlock.





Basic Facts

- In summary, if a **resource-allocation graph** does not have a **cycle**, then the system is *not* in a **deadlocked state**.
- If there is a **cycle**, then the system *may* or *may not* be in a **deadlocked state**.
- This observation is important when we deal with the deadlock problem.





Methods for Handling Deadlocks

- It is possible to deal with the **deadlock problem** in one of **three ways**:
 - *Ignore the problem altogether 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 deadlocked state.*
 - *Allow the system to enter a deadlocked state, detect it, and recover.*
- The **first solution** is the one used by most operating systems, including **Linux** and **Windows**.
 - It is then up to kernel and application developers to write programs that handle deadlocks, typically using approaches outlined in the **second solution**.





Methods for Handling Deadlocks

- To ensure that deadlocks never occur, the system can use either a **deadlock prevention** or a **deadlock-avoidance** scheme.
- **Deadlock prevention** provides a set of methods to ensure that at least more than one of the **four necessary deadlock conditions** is not satisfied.
- These methods **prevent deadlocks** by constraining how requests for resources can be made.
- **Deadlock avoidance** requires that the OS be given additional information in advance concerning which **resources** a thread will **request** and use during its lifetime.





Deadlock Prevention

- As we noted in the previous section, for a **deadlock** to occur, each of the **four necessary conditions** must hold;
 - **Mutual Exclusion**
 - **Hold and Wait**
 - **No Preemption**
 - **Circular Wait**
- By ensuring that at least one of these conditions cannot hold, we can ***prevent the occurrence of a deadlock***.
- We elaborate on this approach by examining each of the four necessary conditions separately.





Mutual Exclusion

- **Mutual Exclusion** – is not required for **sharable resources** (e.g., read-only files); it must hold for **non-sharable resources**.
 - **Sharable resources**, in contrast, *do not require mutually exclusive access* and thus cannot be involved in a deadlock.
 - ▶ **Read-only files** are a good example of a **sharable resource**. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
 - ▶ **A process never needs to wait for a sharable resource.**





Hold and Wait

- **Hold and Wait** – must guarantee that whenever a process requests a resource, *it does not hold any other resources*
 - Require process to request and be allocated all its resources before it begins execution or allow process to request resources only when the process has none allocated to it.
 - Low resource utilization; starvation possible





No Preemption

- **No Preemption** – The third necessary condition for deadlocks is that there be *no preemption of resources that have already been allocated*.
 - 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 process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting





Circular Wait

- **Circular Wait** – The fourth and final condition for deadlocks is the circular-wait condition.
 - One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

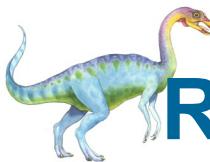




Deadlock Avoidance

- **Deadlock-prevention algorithms**, as discussed in the previous section *prevent deadlocks by limiting how requests can be made.*
 - The limits ensure that at least one of the four necessary conditions for **deadlock** cannot occur.
 - Possible side effects of preventing deadlocks by this method, however, are *low device utilization and reduced system throughput.*
- **An alternative method for avoiding deadlocks** is to require additional information about how resources are to be requested.





Resource-Allocation Graph Scheme

- If we have a **resource-allocation system** with only **one instance of each resource** type, we can use a variant of the resource-allocation graph defined here for **deadlock avoidance**.
- In addition to the **request** and **assignment** edges already described, we introduce a new type of edge, called a **claim edge**.
- A **claim edge** $T_i \rightarrow R_j$ indicates that process T_i may request resource R_j at some time in the future.
- **Claim edge resembles a request edge** in direction but is represented in the graph by a **dashed line** (**Figure 8.9**).





Resource-Allocation Graph

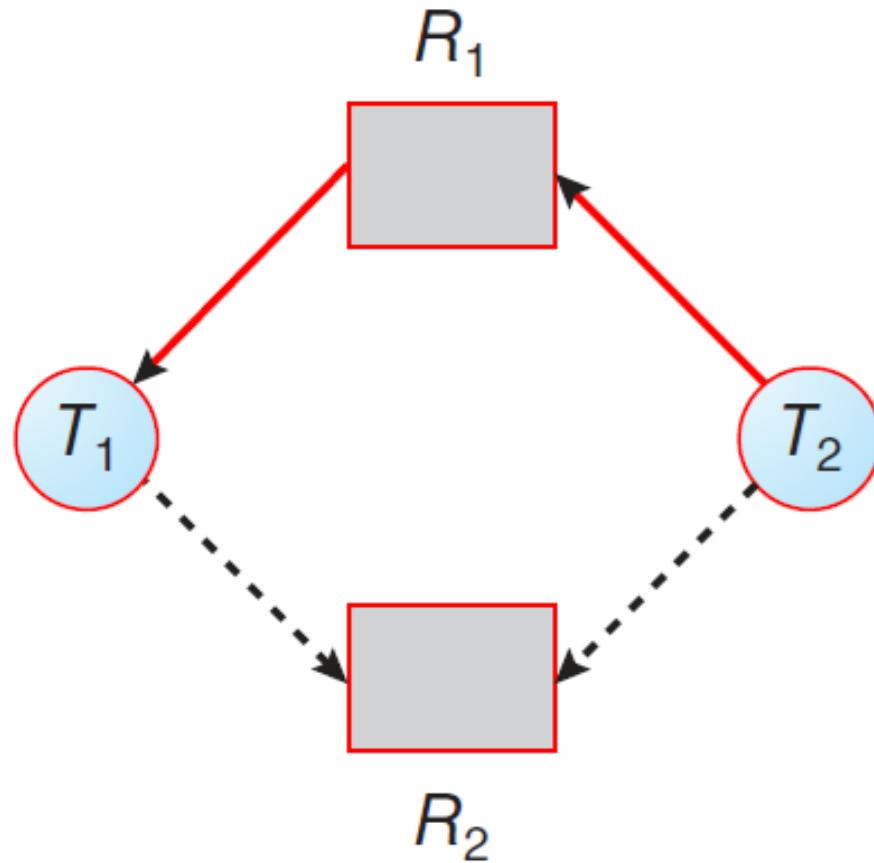
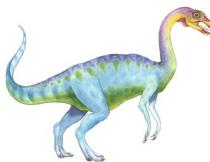


Figure 8.9 Resource-allocation graph with
claim edge

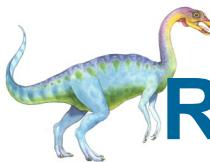




Resource-Allocation Graph Scheme

- **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.
 - That is, ***before process T_i starts executing, all its claim edges must already appear in the resource-allocation graph (See Figure 8.9a).***

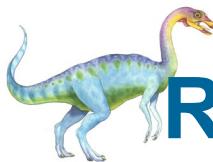




Resource-Allocation Graph Scheme

- Now suppose that process T_i requests resource R_j . The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph.
- To illustrate this algorithm, we consider the resource-allocation graph of **Figure 8.9**.
- Suppose that T_2 requests R_2 . Although R_2 is currently free, we cannot allocate it to T_2 , as this would create a cycle in the graph (**Figure 8.9a**), which is an **unsafe state**.
 - A **cycle** indicates that the system is in an **unsafe state**
- If T_1 requests R_2 and R_2 assigned T_2 and T_2 requests R_1 , then a **deadlock will occur** (**Figure 8.9a**)





Resource-Allocation Graph Scheme

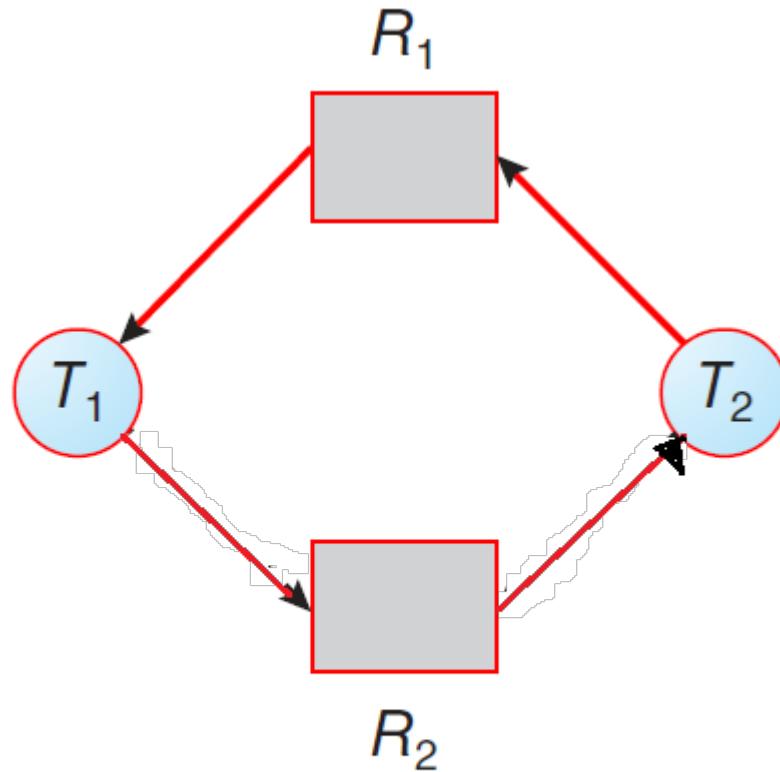


Figure 8.9a Resource-allocation graph with deadlock





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

- A state is **safe** if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock.
 - A safe state is one in which there is at least one sequence of resource allocations to threads that does not result in a deadlock – all of the threads can be run to completion
- More formally, a system is in a safe state only if there exists a **safe sequence**.
- A sequence of threads $\langle T_1, T_2, \dots, T_n \rangle$ is a **safe sequence** for the current allocation state if, for each T_i , the resource requests that T_i can still make can be satisfied by the currently available resources plus the resources held by all T_j , with $j < i$.





Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a **safe state**
- System is in a **safe state** if there exists a sequence $\langle T_1, T_2, \dots, T_n \rangle$ of ALL the processes in the system such that for each P_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 State: Basic Facts

- If a system is in **safe state**, \Rightarrow no deadlocks
- If a system is in **unsafe state** \Rightarrow possibility of deadlock
- **Deadlock Avoidance** \Rightarrow ensure that a system will never enter an **unsafe state**.





Deadlock Avoidance Algorithms

- **Single instance of a resource type:**
 - Use a resource-allocation graph to detect a deadlock

- **Multiple instances** of a resource type:
 - The **banker's algorithm** detects a deadlock





Single Instance of Each Resource Type

- If all resources have only a **single instance**, then we can define a **deadlock detection algorithm** that uses a variant of the resource-allocation graph, called a **wait-for graph** (see **Figure 8.11**).
- We obtain this graph from the **resource-allocation graph** by removing the resource nodes and collapsing the appropriate edges.
- As before, a deadlock exists in the system if and only if the **wait-for graph** contains a **cycle**.
- To detect deadlocks, the system needs to **maintain** the wait for graph and periodically **invoke an algorithm** that searches for a cycle in the graph.
- An algorithm to **detect a cycle** in a graph requires **$O(n^2)$** operations, where **n** is the number of **vertices in the graph**.





Single Instance of Each Resource Type

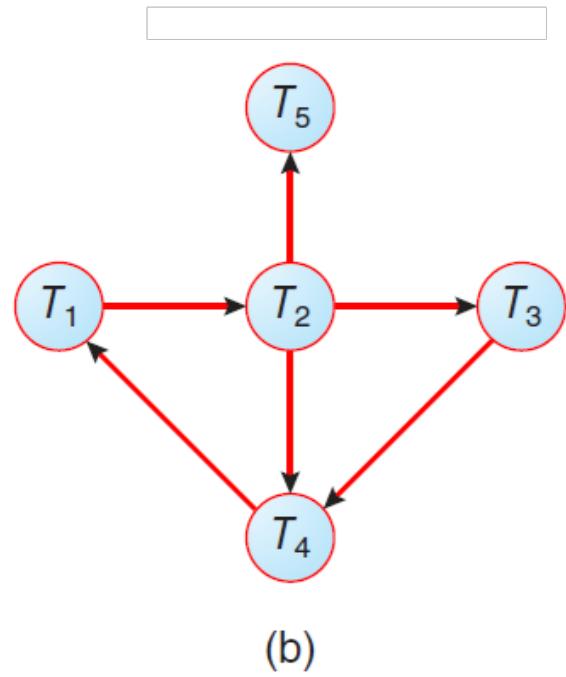
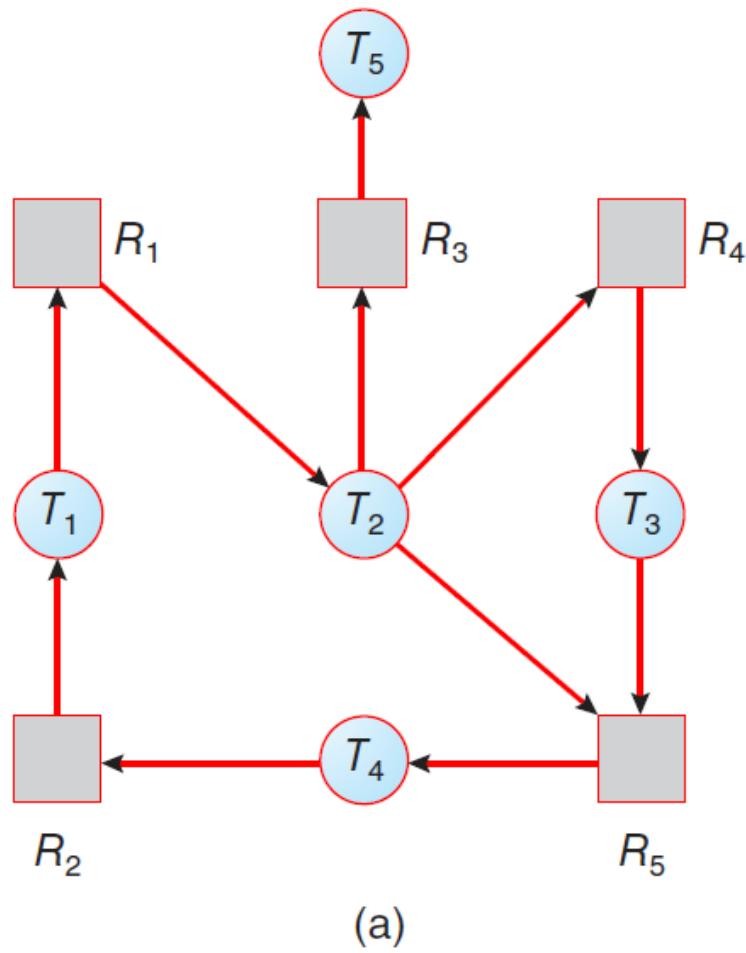


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

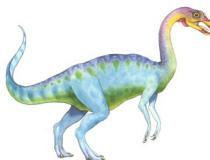




Banker's Algorithm

- The **resource-allocation-graph algorithm** is not applicable to a resource allocation system with **multiple instances** of each resource type.
- The **deadlock avoidance algorithm** that we describe next is applicable to such a system but is **less efficient** than the resource-allocation graph scheme.
- This algorithm is commonly known as the **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*.





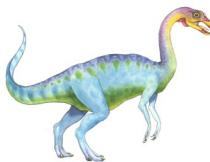
Data Structures for the Banker' s Algorithm

Let n = number of threads, and m = number of resource units.

- **Available:** is a **vector** with length m . If $\text{available}[j] = k$, there are k instances of resource type R_j available
- **Max:** is an $n \times m$ **matrix**. If $\text{Max}[i,j] = k$, then **thread** P_i may request at most k instances of resource type R_j
- **Allocation:** is an $n \times m$ **matrix**. If $\text{Allocation}[i,j] = k$, then P_i is currently allocated k instances of R_j
- **Need:** is an $n \times m$ **matrix**. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task

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





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 ≤ 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 thread P_i . If $\text{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise an **error condition**, since the thread has exceeded its maximum claim
2. If $\text{Request}_i \leq \text{Available}$, 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:

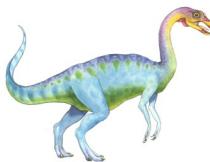
$$\text{Available} = \text{Available} - \text{Request}_i;$$

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

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

- 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



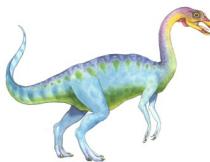


Example of Banker's Algorithm

- 5 threads P_0 through P_4 ;
- 3 resource types (**A**, **B**, and **C**):
A (10 instances), **B** (5 instances), and **C** (7 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available vector</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
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			





Example (Cont.)

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

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

The system is in a **safe state** since the thread execution sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 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
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 $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?





Recovery from Deadlock

- When a detection algorithm determines that a **deadlock** exists, several alternatives are available.
- **There are two options for breaking a deadlock:**
 - One is simply to **abort** one or more processes to break the **circular wait**.
 - The other is to **preempt** some resources from one or more of the **deadlocked threads**.

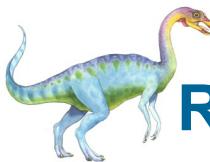




Eliminate deadlocks by aborting a thread

- To **eliminate deadlocks by aborting a thread**, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated threads.
 - **Abort all deadlocked threads**: This method clearly will *break the deadlock cycle*, but at great expense. The deadlocked threads may have been computing for a long time, and the results of these partial computations must be discarded and will likely have to be recomputed later.
 - **Abort one thread at a time until the deadlock cycle is eliminated**: This method incurs considerable overhead, since after each thread is **aborted**, a *deadlock-detection algorithm* must be invoked to determine whether any threads are still deadlocked.





Recovery from Deadlock: Resource Preemption

- To eliminate deadlocks using **resource preemption**, we successively **preempt some resources from threads and give these resources to other threads** until the *deadlock cycle is broken*.
- If preemption is required to deal with deadlocks, then three issues need to be addressed:
 - **Selecting a victim**
 - **Rollback**
 - **Starvation**





Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – Which resources and which threads are to be preempted? In **thread termination**, we must determine the preemption order to **minimize cost**.
- **Rollback** – If we preempt a resource from a thread, what should be done with that thread? Clearly, it cannot continue with its normal execution; it is missing some needed resource.
 - We must **roll back** the thread to some **safe state** and restart it from that state. Return to some **safe state**, restart the thread for that state.
- **Starvation** – How do we ensure that starvation will not occur? The same thread may always be picked as a **victim**. The most common solution is to include the number of **rollbacks** in the **cost factor**.

