8.1 System model

Introduction

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***. We discussed this issue briefly in the chapter Synchronization Tools as a form of liveness failure. There, we defined deadlock as 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*.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

In this chapter, we describe 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. Deadlock problems—as well as other liveness failures—are becoming more challenging as demand continues for increased concurrency and parallelism on multicore systems.

Chapter objectives

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

Section glossary

deadlock: The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads.

System model

A system consists of a finite number of resources to be distributed among a number of competing threads. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as network interfaces and DVD drives) 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. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

The various synchronization tools discussed in the chapter Synchronization Tools, such as mutex locks and semaphores, are also system resources; and on contemporary computer systems, they are the most common sources of deadlock. However, definition is not a problem here. A lock is typically associated with a specific data structure—that is, one lock may be used to protect access to a queue, another to protect access to a linked list, and so forth. For that reason, each instance of a lock is typically assigned its own resource class.

Note that throughout this chapter we discuss kernel resources, but threads may use resources from other processes (for example, via interprocess communication), and those resource uses can also result in deadlock. Such deadlocks are not the concern of the kernel and thus not described here.

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 resources requested may not exceed the total number of resources available in the system. In other words, a thread cannot request two network interfaces if the system has only one.

Under the normal mode of operation, a thread may utilize a resource in only the following sequence:

1. **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.

2. **Use.** The thread can operate on the resource (for example, if the resource is a mutex lock, the thread can access its critical section).
3. **Release.** The thread releases the resource.

The request and release of resources may be system calls, as explained in the chapter Operating-System Structures. Examples are the `request()` and `release()` of a device, `open()` and `close()` of a file, and `allocate()` and `free()` memory system calls. Similarly, as we saw in the chapter Synchronization Tools, request and release can be accomplished through the `wait()` and `signal()` operations on semaphores and through `acquire()` and `release()` of a mutex lock. For each use of a kernel-managed resource by a thread, the operating system checks to make sure that the thread has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated, 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 queue of threads waiting 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. The resources are typically logical (for example, mutex locks, semaphores, and files); however, other types of events may result in deadlocks, including reading from a network interface or the IPC (interprocess communication) facilities discussed in the chapter Processes.

To illustrate a deadlocked state, we refer back to the dining-philosophers problem from Section The dining-philosophers problem. In this situation, resources are represented by chopsticks. If all the philosophers get hungry at the same time, and each philosopher grabs the chopstick on her left, there are no longer any available chopsticks. Each philosopher is then blocked waiting for her right chopstick to become available.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in the chapter Synchronization Tools are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise, deadlock can occur, as described next.

PARTICIPATION ACTIVITY 8.1.1: Section review questions.

1) A deadlock occurs when ____.

- a process is waiting for I/O to a device that does not exist
- the system has no available free resources
- every process in a set is waiting for an event that can

only be caused by another process in the set

2) Processes can wait while _____ a resource.
   - requesting
   - using
   - releasing

8.2 Deadlock in multithreaded applications

Prior to examining how deadlock issues can be identified and managed, we first illustrate how deadlock can occur in a multithreaded Pthread program using POSIX mutex locks. 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()`.

Two mutex locks are created and initialized in the following code example:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

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

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown in Figure 8.2.1.

Figure 8.2.1: Deadlock example.

```
/* 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);
}
```


In this example, `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 `first_mutex` while `thread_two` acquires `second_mutex`.

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.

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. Whereas 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.

Livelock can be illustrated with the Pthreads `pthread_mutex_trylock()` function, which attempts to acquire a mutex lock without blocking. The code example in Figure 8.2.2 rewrites the example from Figure 8.2.1 so that it now uses `pthread_mutex_trylock()`. This situation can lead to livelock if `thread_one` acquires `first_mutex`, followed by `thread_two` acquiring `second_mutex`. Each thread then invokes `pthread_mutex_trylock()`, which fails, releases their respective locks, and repeats the same actions indefinitely.

Figure 8.2.2: Livelock example.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&first_mutex);
      if
(pthread_mutex_trylock(&second_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&second_mutex);
         pthread_mutex_unlock(&first_mutex);
         done = 1;
     }
     else
         pthread_mutex_unlock(&first_mutex);
   }
   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   int done = 0;

   while (!done) {
     pthread_mutex_lock(&second_mutex);
     if (pthread_mutex_trylock(&first_mutex))
{
         /**
          * Do some work
          */
         pthread_mutex_unlock(&first_mutex);
         pthread_mutex_unlock(&second_mutex);
         done = 1;
     }
     else
         pthread_mutex_unlock(&second_mutex);
   }

   pthread_exit(0);
}
```


©zyBooks 10/20/23 10:00 1812110
Christian Johnson
USCGA7345ElwakilFall2023

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.

PARTICIPATION ACTIVITY 8.2.1: Section review questions.

1) The code example below _____ lead to deadlock between two threads.


```
/* 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);
}
```


- will not
- can
- will

Section glossary

livelock: A condition in which a thread continuously attempts an action that fails.

8.3 Deadlock characterization

In the previous section we illustrated how deadlock could occur in multithreaded programming using mutex locks. We now look more closely at conditions that characterize deadlock.

Necessary conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion.** At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
2. **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.
3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
4. **Circular wait.** A set { , , …, } of waiting threads must exist such that is waiting for a resource held by , is waiting for a resource held by , …, is waiting for a resource held by , and is waiting for a resource held by .

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 8.5, however, that it is useful to consider each condition separately.

PARTICIPATION ACTIVITY 8.3.1: Mid-section review question.

1) The four necessary conditions of deadlock must be present for deadlock to occur.
   - ○ True
   - ○ False

Resource-allocation graph

Deadlocks can be described more precisely in terms of a directed graph called a ***system resource-allocation graph***. This graph consists of a set of vertices and a set of edges . The set of vertices is partitioned into two different types of nodes: = { , , ..., }, the set consisting of all the active threads in the system, and = { , , ..., }, the set consisting of all resource types in the system.

A directed edge from thread to resource type is denoted by → ; it signifies that thread has requested an instance of resource type and is currently waiting for that resource. A directed edge from resource type to thread is denoted by → ; it signifies that an instance of resource type has been allocated to thread . A directed edge → is called a ***request edge***; a directed edge → is called an ***assignment edge***.

Pictorially, we represent each thread as a circle and each resource type as a rectangle. As a simple example, the resource allocation graph shown in the animation below illustrates the deadlock situation from the program in Figure 8.2.1. Since resource type may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points only to the rectangle , whereas an assignment edge must also designate one of the dots in the rectangle.

PARTICIPATION ACTIVITY 8.3.2: Resource-allocation graph for the program in Figure 8.3.1.

Animation content:

Step 1: thread_one acquires first_mutex. Step 2: thread_two acquires second_mutex. Step 3: thread_one wants second_mutex, which could be freed up by thread_two. Step 4: thread_two wants first_mutex, and as it is still holding second_mutex which thread_one is waiting for, there is a cycle in the graph: first_mutex → T1 → second_mutex → T2 → first_mutex.

Animation captions:

1. thread_one acquires first_mutex.
2. thread_two acquires second_mutex.
3. thread_one wants second_mutex, which could be freed up by thread_two.
4. thread_two wants first_mutex, and as it is still holding second_mutex which thread_one is waiting for, there is a cycle in the graph: first_mutex → T1 → second_mutex → T2 → first_mutex.

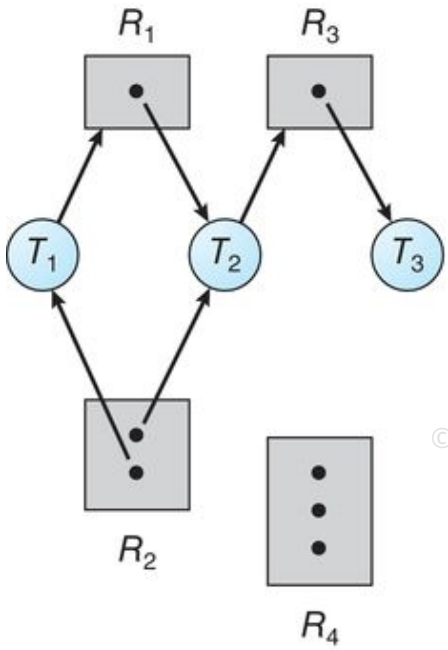
When thread requests an instance of resource type , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously***

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.

The resource-allocation graph shown in Figure 8.3.1 depicts the following situation.

- The sets , , and :
  - = { , , }
  - = { , , , }
  - = { → , → , → , → , → , → }
- Resource instances:
  - One instance of resource type
  - Two instances of resource type
  - One instance of resource type
  - Three instances of resource type
- Thread states:
  - Thread is holding an instance of resource type and is waiting for an instance of resource type .
  - Thread is holding an instance of and an instance of and is waiting for an instance of .
  - Thread is holding an instance of .

Figure 8.3.1: Resource-allocation graph.



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.

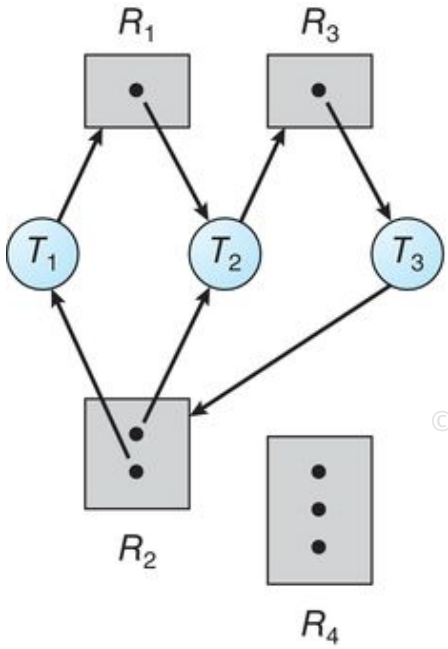
If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 8.3.1. Suppose that thread requests an instance of resource type . Since no resource instance is currently available, we add a request edge → to the graph (Figure 8.3.2). At this point, two minimal cycles exist in the system:

```
T1 → R1 → T2 → R3 → T3 → R2 → T1
T2 → R3 → T3 → R2 → T2
```

Threads , , and are deadlocked. Thread is waiting for the resource , which is held by thread . Thread is waiting for either thread or thread to release resource . In addition, thread is waiting for thread to release resource .

Figure 8.3.2: Resource-allocation graph with a deadlock.

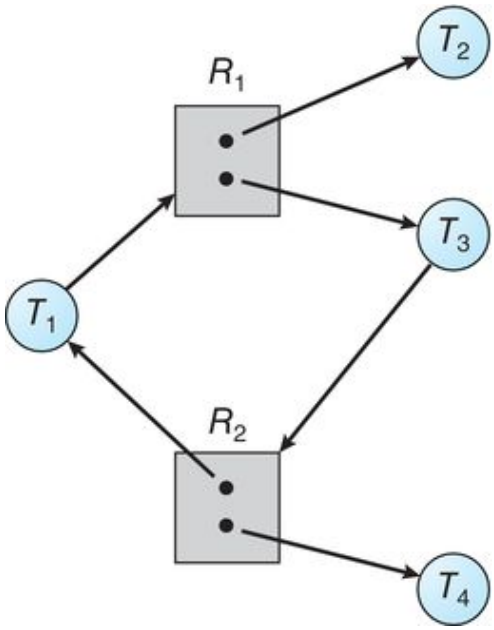


Now consider the resource-allocation graph in Figure 8.3.3. In this example, we also have a cycle:

```
T1 → R1 → T3 → R2 → T1
```

However, there is no deadlock. Observe that thread may release its instance of resource type . That resource can then be allocated to , breaking the cycle.

Figure 8.3.3: Resource-allocation graph with a cycle but no deadlock.



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.

PARTICIPATION ACTIVITY 8.3.3: Section review question.

1) One necessary condition for deadlock is ____, which states that a process must be holding one resource and waiting to acquire additional resources.

- hold and wait
- mutual exclusion
- no preemption

2) The presence of a cycle in a resource-allocation graph is ____.

- a necessary and sufficient condition for deadlock in the case that each resource has more than one instance
- a necessary and sufficient condition for a deadlock in the case that each resource has exactly one instance
- is neither necessary nor sufficient for indicating deadlock in the case that each resource has exactly one instance

3) If a resource-allocation graph has a cycle, the system must be in a deadlock.

- True
- False

Section glossary

system resource-allocation graph: A directed graph for precise description of deadlocks.

request edge: In a system resource-allocation graph, an edge (arrow) indicating a resource request.

assignment edge: In a system resource-allocation graph, an edge (arrow) indicating a resource assignment.

8.4 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.
- We can 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. Some systems—such as databases—adopt the third solution, allowing deadlocks to occur and then managing the recovery.

Next, we elaborate briefly on the three methods for handling deadlocks. Then, in Section 8.5 through Section 8.8, we present detailed algorithms. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

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 one of the necessary conditions (Section Necessary conditions) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 8.5.

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. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread. We discuss these schemes in Section 8.6.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 8.7 and Section 8.8.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by threads that cannot run and because more and more threads, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is

nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per month), the extra expense of the other methods may not seem worthwhile.

In addition, methods used to recover from other liveness conditions, such as livelock, may be used to recover from deadlock. In some circumstances, a system is suffering from a liveness failure but is not in a deadlocked state. We see this situation, for example, with a real-time thread running at the highest priority (or any thread running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

PARTICIPATION ACTIVITY 8.4.1: Section review questions.

1) To handle deadlocks, operating systems most often ____.
   - ignore the problem of deadlocks
   - use protocols to prevent or avoid deadlocks
   - detect and recover from deadlocks

2) Deadlock _____ handles deadlocks by ensuring that one of their necessary conditions cannot hold.
   - avoidance
   - prevention
   - recovery

Section glossary

deadlock prevention: A set of methods intended to ensure that at least one of the necessary conditions for deadlock cannot hold.

deadlock avoidance: An operating system method in which processes inform the operating system of which resources they will use during their lifetimes so the system can approve or deny requests to avoid deadlock.

8.5 Deadlock prevention

As we noted in Section Necessary conditions, for a deadlock to occur, each of the four necessary conditions must hold. 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

The mutual-exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources 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 threads attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A thread never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several threads.

Hold and wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a thread requests a resource, it does not hold any other resources. One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is, of course, impractical for most applications due to the dynamic nature of requesting resources.

An alternative protocol allows a thread to request resources only when it has none. A thread may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. For example, a thread may be allocated a mutex lock for its entire execution, yet only require it for a short duration. Second, starvation is possible. A thread that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other thread.

No preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a thread is holding some resources and requests another resource that cannot be

immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. 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. While it is waiting, some of its resources may be preempted, but only if another thread requests them. A thread can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions. It cannot generally be applied to such resources as mutex locks and semaphores, precisely the type of resources where deadlock occurs most commonly.

Circular wait

The three options presented thus far for deadlock prevention are generally impractical in most situations. However, the fourth and final condition for deadlocks — the circular-wait condition — presents an opportunity for a practical solution by invalidating one of the necessary conditions. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.

To illustrate, we let = { , , …, } be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function : → , where is the set of natural numbers. We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. For example, the lock ordering in the Pthread program shown in Figure 8.2.1 could be

```
F(first_mutex) = 1
F(second_mutex) = 5
```


We can now consider the following protocol to prevent deadlocks: Each thread can request resources only in an increasing order of enumeration. That is, a thread can initially request an instance of a resource—say, . After that, the thread can request an instance of resource if and only if ( ) > ( ). For example, using the function defined above, a thread that wants to use both `first_mutex` and `second_mutex` at the same time must first request `first_mutex` and then `second_mutex`. Alternatively, we can require that a thread requesting an instance of resource must have released any resources such that ( ) ≥ ( ). Note also that if

several instances of the same resource type are needed, a ***single*** request for all of them must be issued.

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of threads involved in the circular wait be { , , …, }, where is waiting for a resource , which is held by thread . (Modulo arithmetic is used on the indexes, so that is waiting for a resource held by .) Then, since thread is holding resource while requesting resource , we must have ( ) < ( ) for all . But this condition means that ( ) < ( ) < … < ( ) < ( ). By transitivity, ( ) < ( ), which is impossible. Therefore, there can be no circular wait.

Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering. However, establishing a lock ordering can be difficult, especially on a system with hundreds—or thousands—of locks. To address this challenge, many Java developers have adopted the strategy of using the method `System.identityHashCode(Object)` (which returns the default hash code value of the `Object` parameter it has been passed) as the function for ordering lock acquisition.

It is also important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a function that transfers funds between two accounts. To prevent a race condition, each account has an associated mutex lock that is obtained from a `get_lock()` function such as that shown in Figure 8.5.1. Deadlock is possible if two threads simultaneously invoke the `transaction()` function, transposing different accounts. That is, one thread might invoke

```
transaction(checking_account, savings_account, 25.0)
```

and another might invoke

```
transaction(savings_account, checking_account, 50.0)
```

Figure 8.5.1: Deadlock example with lock ordering.

```
void transaction(Account from, Account to, double
amount)
{
  mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);

  acquire(lock1);
    acquire(lock2);

      withdraw(from, amount);
      deposit(to, amount);

    release(lock2);
  release(lock1);
}
```


PARTICIPATION ACTIVITY 8.5.1: Section review question.

1) Deadlock prevention by denying the _____ condition is the simplest way to prevent deadlocks.
   - mutual exclusion
   - no preemption
   - circular wait

2) Assume there are three resources, , , and , that are each assigned unique integer values 15, 10, and 25, respectively. What is a resource ordering which prevents a circular wait?
   - , ,
   - , ,
   - , ,

8.6 Deadlock avoidance

Deadlock-prevention algorithms, as discussed in Section 8.5, prevent deadlocks by limiting how

requests can be made. The limits ensure that at least one of the 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. For example, in a system with resources *R*1 and *R*2, the system might need to know that thread will request first *R*1 and then *R*2 before releasing both resources, whereas thread will request *R*2 and then *R*1. With this knowledge of the complete sequence of requests and releases for each thread, the system can decide for each request whether or not the thread should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each thread declare the ***maximum number*** of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the threads. In the following sections, we explore two deadlock-avoidance algorithms.

Linux lockdep tool

Although ensuring that resources are acquired in the proper order is the responsibility of kernel and application developers, certain software can be used to verify that locks are acquired in the proper order. To detect possible deadlocks, Linux provides `lockdep`, a tool with rich functionality that can be used to verify locking order in the kernel. `lockdep` is designed to be enabled on a running kernel as it monitors usage patterns of lock acquisitions and releases against a set of rules for acquiring and releasing locks. Two examples follow, but note that `lockdep` provides significantly more functionality than what is described here:

- The order in which locks are acquired is dynamically maintained by the system. If `lockdep` detects locks being acquired out of order, it reports a possible deadlock condition.
- In Linux, spinlocks can be used in interrupt handlers. A possible source of deadlock occurs when the kernel acquires a spinlock that is also used in an interrupt handler. If the interrupt occurs while the lock is being held, the interrupt handler preempts the kernel code currently holding the lock and then spins while attempting to acquire the lock, resulting in deadlock. The general

strategy for avoiding this situation is to disable interrupts on the current processor before acquiring a spinlock that is also used in an interrupt handler. If `lockdep` detects that interrupts are enabled while kernel code acquires a lock that is also used in an interrupt handler, it will report a possible deadlock scenario.

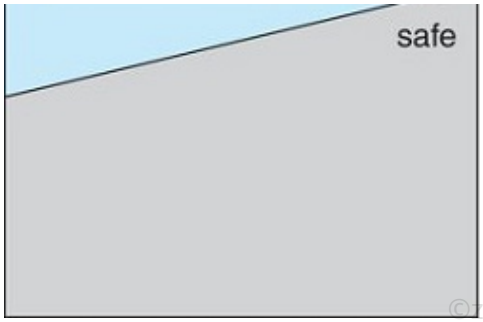
`lockdep` was developed to be used as a tool in developing or modifying code in the kernel and not to be used on production systems, as it can significantly slow down a system. Its purpose is to test whether software such as a new device driver or kernel module provides a possible source of deadlock. The designers of `lockdep` have reported that within a few years of its development in 2006, the number of deadlocks from system reports had been reduced by an order of magnitude. Although `lockdep` was originally designed only for use in the kernel, recent revisions of this tool can now be used for detecting deadlocks in user applications using Pthreads mutex locks. Further details on the `lockdep` tool can be found at https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

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. More formally, a system is in a safe state only if there exists a ***safe sequence***. A sequence of threads < , , ..., > is a safe sequence for the current allocation state if, for each , the resource requests that can still make can be satisfied by the currently available resources plus the resources held by all , with < . In this situation, if the resources that needs are not immediately available, then can wait until all have finished. When they have finished, can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When terminates, can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 8.6.1). 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.

Figure 8.6.1: Safe, unsafe, and deadlocked state spaces.



PARTICIPATION ACTIVITY 8.6.1: Mid-section review question.

1) Only an unsafe state can lead to deadlock.
   - True
   - False

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

	Maximum Needs	Current Needs
	10	5
	4	2
	9	2

At time , the system is in a safe state. The sequence < , , > satisfies the safety condition. Thread can immediately be allocated all its resources and then return them (the system will then have five available resources); then thread can get all its resources and return them (the system will then have ten available resources); and finally thread can get all its resources and return them (the system will then have all twelve resources available).

A system can go from a safe state to an unsafe state. Suppose that, at time , thread requests and is allocated one more resource. The system is no longer in a safe state. At this point, only thread can be allocated all its resources. When it returns them, the system will have only four available resources. Since thread is allocated five resources but has a maximum of ten, it may request five more resources. If it does so, it will have to wait, because they are unavailable. Similarly, thread may request six additional resources and have to wait, resulting in a deadlock.

Our mistake was in granting the request from thread for one more resource. If we had made wait until either of the other threads had finished and released its resources, then we could have avoided the deadlock.

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.

In this scheme, if a thread requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

Resource-allocation-graph algorithm

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 in Section Resource-allocation graph 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 → indicates that thread may request resource at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When thread requests resource , the claim edge → is converted to a request edge. Similarly, when a resource is released by , the assignment edge → is reconverted to a claim edge → .

Note that the resources must be claimed a priori in the system. That is, before thread starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge → to be added to the graph only if all the edges associated with thread are claim edges.

Now suppose that thread requests resource . 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. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of operations, where is the number of threads in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, thread will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 8.6.2. Suppose that requests . Although is currently free, we cannot allocate it to , since this action will create a cycle in the graph (Figure 8.6.3). A cycle, as mentioned, indicates that the system is in an unsafe state. If requests , and requests , then a deadlock will occur.

Figure 8.6.2: Resource-allocation graph for deadlock avoidance.

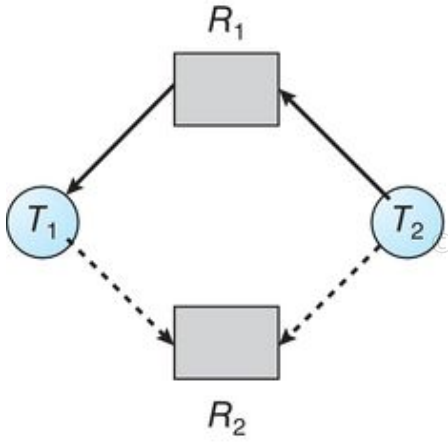
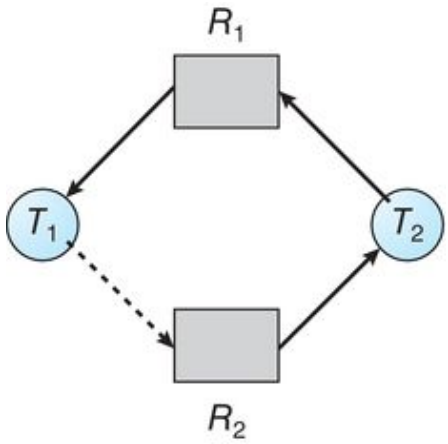


Figure 8.6.3: An unsafe state in a resource-allocation graph.



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.

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 user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the thread must wait until some other thread releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where is the number of threads in the system and is the number of resource types:

- **Available.** A vector of length indicates the number of available resources of each type. If ***Available***[ ] equals , then instances of resource type are available.
- **Max.** An × matrix defines the maximum demand of each thread. If ***Max***[ ][ ] equals , then thread may request at most instances of resource type .
- **Allocation.** An × matrix defines the number of resources of each type currently allocated to each thread. If ***Allocation***[ ][ ] equals , then thread is currently allocated instances of resource type .
- **Need.** An × matrix indicates the remaining resource need of each thread. If ***Need***[ ][ ] equals , then thread may need more instances of resource type to complete its task. Note that ***Need***[ ][ ] equals ***Max***[ ][ ] - ***Allocation***[ ][ ].

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let and be vectors of length . We say that ≤ if and only if [ ] ≤ [ ] for all = 1, 2, ..., . For example, if = (1,7,3,2) and = (0,3,2,1), then ≤ . In addition, < if ≤ and ≠ .

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as $\boldsymbol{Allocation}_i$ and $\boldsymbol{Need}_i$. The vector $\boldsymbol{Allocation}_i$ specifies the resources currently allocated to thread ; the vector $\boldsymbol{Need}_i$ specifies the additional resources that thread may still request to complete its task.

Safety algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length and , respectively. Initialize ***Work*** = ***Available*** and ***Finish***[ ] = ***false*** for = 0, 1, ..., − 1.
2. Find an index such that both
   a. ***Finish***[ ] == ***false***
   b. ***Need*** ≤ ***Work***

   If no such exists, go to step 4.
3. ***Work*** = ***Work*** + $\boldsymbol{Allocation}_i$ ***Finish***[ ] = ***true*** Go to step 2.
4. If ***Finish***[ ] == ***true*** for all , then the system is in a safe state.

This algorithm may require an order of operations to determine whether a state is safe.

Resource-request algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let ***Request***$_i$ be the request vector for thread . If ***Request***$_i$ [ ] == , then thread wants instances of resource type . When a request for resources is made by thread , the following actions are taken:

1. If ***Request***$_i \leq$ ***Need***$_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If ***Request***$_i \leq$ ***Available***, go to step 3. Otherwise, must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread by modifying the state as follows:

```
Available = Available-Requesti
Allocationi = Allocationi + Requesti
Need = Needi-Requesti
```

If the resulting resource-allocation state is safe, the transaction is completed, and thread is allocated its resources. However, if the new state is unsafe, then must wait for ***Request***$_i$, and the old resource-allocation state is restored.

An illustrative example

To illustrate the use of the banker's algorithm, consider a system with five threads through and three resource types , , and . Resource type has ten instances, resource type has five instances, and resource type has seven instances. Suppose that the following snapshot represents the current state of the system:

	Allocation	Max	Available
	A B C	*A B C*	*A B C*
	0 1 0	7 5 3	3 3 2
	2 0 0	3 2 2	
	3 0 2	9 0 2	
	2 1 1	2 2 2	
	0 0 2	4 3 3	

The content of the matrix ***Need*** is defined to be ***Max*** − ***Allocation*** and is as follows:

	Need
	A B C
	7 4 3
	1 2 2
	6 0 0
	0 1 1
	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence < , , , , > satisfies the safety criteria. Suppose now that thread requests one additional instance of resource type and two instances of resource type , so $\boldsymbol{Request}_1$ = (1,0,2). To decide whether this request can be immediately granted, we first check that $\boldsymbol{Request}_1 \leq \boldsymbol{Available}$—that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	Need	Available
	A B C	*A B C*	*A B C*
	0 1 0	7 4 3	2 3 0
	3 0 2	0 2 0	
	3 0 2	6 0 0	
	2 1 1	0 1 1	
	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence < , , , , > satisfies the safety requirement. Hence, we can immediately grant the request of thread .

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

PARTICIPATION ACTIVITY 8.6.2: Section review questions.

1) Suppose that there are ten resources available to three processes. At time 0, the following data is collected. The table indicates the process, the maximum number of resources needed by the process, and the number of resources currently owned by each process. Which of the following correctly characterizes this state?

Process	Maximum Needs	Currently Owned
	10	4
	3	1
	6	4

- It is safe.
- It is not safe.
- The state cannot be determined.

2) Suppose that there are 12 resources available to three processes. At time 0, the following data is collected. The table indicates the process, the maximum number of resources needed by the process, and the number of resources currently owned by each process. Which of the following correctly characterizes this state?

Process	Maximum Needs	Currently Owned

Process	Maximum Needs	Currently Owned
	10	4
	3	2
	7	4

- It is safe.
- It is not safe.
- The state cannot be determined.

Section glossary

safe sequence: In deadlock avoidance, a sequence of processes <P1, P2, ..., Pn> in which, for each Pi, the resource requests that Pi can make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i.

claim edge: In the deadlock resource-allocation-graph algorithm, an edge indicating that a process might claim a resource in the future.

banker's algorithm: A deadlock avoidance algorithm, less efficient than the resource-allocation graph scheme but able to deal with multiple instances of each resource type.

8.7 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

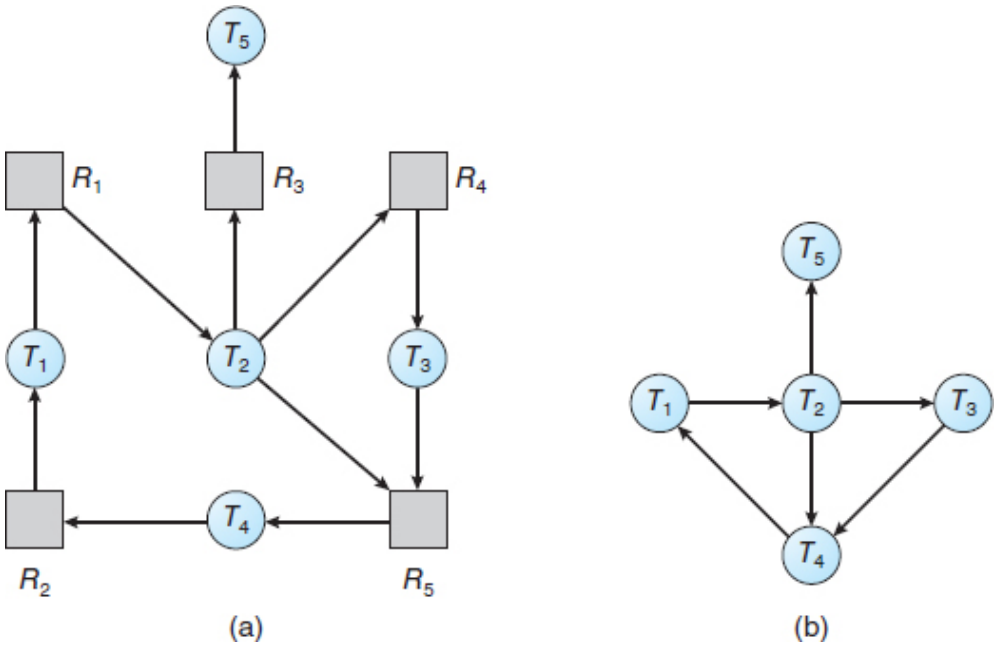
Next, we discuss these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from 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. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from to in a wait-for graph implies that thread is waiting for thread to release a resource that needs. An edge → exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges → and → for some resource . In Figure 8.7.1, we present a resource-allocation graph and the corresponding wait-for graph.

Figure 8.7.1: (a) Resource-allocation graph. (b) Corresponding wait-for graph.



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*( ) operations, where is the number of vertices in the graph.

The BCC toolkit described in Section BCC provides a tool that can detect potential deadlocks with Pthreads mutex locks in a user process running on a Linux system. The BCC tool `deadlock_detector` operates by inserting probes which trace calls to the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. When the specified

process makes a call to either function, `deadlock_detector` constructs a wait-for graph of mutex locks in that process, and reports the possibility of deadlock if it detects a cycle in the graph.

Several instances of a resource type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock-detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section Banker's algorithm):

- **Available.** A vector of length indicates the number of available resources of each type.
- **Allocation.** An × matrix defines the number of resources of each type currently allocated to each thread.
- **Request.** An × matrix indicates the current request of each thread. If ***Request***[ ][ ] equals , then thread is requesting more instances of resource type .

Deadlock detection using java thread dumps

Although Java does not provide explicit support for deadlock detection, a ***thread dump*** can be used to analyze a running program to determine if there is a deadlock. A thread dump is a useful debugging tool that displays a snapshot of the states of all threads in a Java application. Java thread dumps also show locking information, including which locks a blocked thread is waiting to acquire. When a thread dump is generated, the JVM searches the wait-for graph to detect cycles, reporting any deadlocks it detects. To generate a thread dump of a running application, from the command line enter:

```
Ctrl-\ (UNIX, Linux, or macOS)
Ctrl-Break (Windows)
```

In the source-code download for this text, we provide a Java example of the program shown in Figure 8.2.1 and describe how to generate a thread dump that reports the deadlocked Java threads.

The ≤ relation between two vectors is defined as in Section Banker's algorithm. To simplify notation, we again treat the rows in the matrices ***Allocation*** and ***Request*** as vectors; we refer to them as $\boldsymbol{Allocation}_i$ and $\boldsymbol{Request}_i$. The detection algorithm described here simply investigates every possible allocation sequence for the threads that remain to be completed. Compare this algorithm with the banker's algorithm of Section Banker's algorithm.

1. Let ***Work*** and ***Finish*** be vectors of length and , respectively. Initialize ***Work*** = ***Available***. For = 0, 1, …, −1, if ***Allocation***$_i$ ≠ 0, then ***Finish***[ ] = ***false***. Otherwise, ***Finish***[ ] = ***true***.
2. Find an index such that both
   a. ***Finish***[ ] == ***false***
   b. ***Request***$_i$ ≤ ***Work***

   If no such exists, go to step 4.
3. ***Work*** = ***Work*** + ***Allocation***$_i$ ***Finish***[ ] = ***true*** Go to step 2.
4. If ***Finish***[ ] == ***false*** for some , 0 ≤ < , then the system is in a deadlocked state. Moreover, if ***Finish***[ ] == ***false***, then thread is deadlocked.

This algorithm requires an order of operations to detect whether the system is in a deadlocked state.

You may wonder why we reclaim the resources of thread (in step 3) as soon as we determine that ***Request***$_i$ ≤ ***Work*** (in step 2b). We know that is currently ***not*** involved in a deadlock (since ***Request***$_i$ ≤ ***Work***). Thus, we take an optimistic attitude and assume that will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five threads through and three resource types , , and . Resource type has seven instances, resource type has two instances, and resource type has six instances. The following snapshot represents the current state of the system:

	Allocation	Request	Available
	A B C	*A B C*	*A B C*
	0 1 0	0 0 0	0 0 0
	2 0 0	2 0 2	
	3 0 3	0 0 0	
	2 1 1	1 0 0	
	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence < , , , , > results in ***Finish***[ ] == *true* for all .

Suppose now that thread makes one additional request for an instance of type . The ***Request***

matrix is modified as follows:

	Request
	A B C
	0 0 0
	2 0 2
	0 0 1
	1 0 0
	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by thread , the number of available resources is not sufficient to fulfill the requests of the other threads. Thus, a deadlock exists, consisting of threads , , , and .

Detection-algorithm usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How ***often*** is a deadlock likely to occur?
2. How ***many*** threads will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked threads will be idle until the deadlock can be broken. In addition, the number of threads involved in the deadlock cycle may grow.

Managing deadlock in databases

Database systems provide a useful illustration of how both open-source and commercial software manage deadlock. Updates to a database may be performed as ***transactions***, and to ensure data integrity, locks are typically used. A transaction may involve several locks, so it comes as no surprise that deadlocks are possible in a database with multiple concurrent transactions. To manage deadlock, most transactional database systems include a deadlock detection and recovery mechanism. The database server will periodically search for cycles in the wait-for graph to detect deadlock among a set of transactions. When deadlock is detected, a victim is selected and the transaction is aborted and rolled back, releasing the locks held by the victim transaction and freeing the remaining transactions from

deadlock. Once the remaining transactions have resumed, the aborted transaction is reissued. Choice of a victim transaction depends on the database system; for instance, MySQL attempts to select transactions that minimize the number of rows being inserted, updated, or deleted.

Deadlocks occur only when some thread makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting threads. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of threads but also the specific thread that "caused" the deadlock. (In reality, each of the deadlocked threads is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable thread.

Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and causes CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked threads "caused" the deadlock.

PARTICIPATION ACTIVITY 8.7.1: Section review questions.

1) A wait-for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type.
   - ○ True
   - ○ False

Section glossary

wait-for graph: In deadlock detection, a variant of the resource-allocation graph with resource nodes removed; indicates a deadlock if the graph contains a cycle

thread dump: In Java, a snapshot of the state of all threads in an application; a useful debugging tool for deadlocks.

8.8 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.

Process and thread termination

To eliminate deadlocks by aborting a process or thread, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state. Similarly, if the process was in the midst of updating shared data while holding a mutex lock, the system must restore the status of the lock as being available, although no guarantees can be made regarding the integrity of the shared data.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term ***minimum cost*** is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated

Resource preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback.**

   If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

   Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
3. **Starvation.**

   How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

   In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

PARTICIPATION ACTIVITY 8.8.1: Section review questions.

1) Which of the following techniques is not appropriate for recovery from deadlock?

- terminating one process at a time until the deadlock cycle is eliminated.
- terminating all deadlocked processes.
- taking the appropriate resources from one or more of the deadlocked processes and terminating these processes.

2) To recover from a deadlock using resource preemption,

- any ordering of resources and processes may be used.
- a process whose resources have been preempted may continue from the point of preemption.
- ensure that starvation does not occur from always preempting resources from the same process.

Section glossary

recovery mode: A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

©zyBooks 10/20/23 10:00 1812110
Christian Johnson
USCGA7345ElwakilFall2023

8.9 Summary

- Deadlock occurs in a set of processes when every process in the set is waiting for an event that can only be caused by another process in the set.
- There are four necessary conditions for deadlock: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. Deadlock is only possible when all four conditions are present.
- Deadlocks can be modeled with resource-allocation graphs, where a cycle indicates deadlock.
- Deadlocks can be prevented by ensuring that one of the four necessary conditions for deadlock cannot occur. Of the four necessary conditions, eliminating the circular wait is the only practical approach.
- Deadlock can be avoided by using the banker's algorithm, which does not grant resources if doing so would lead the system into an unsafe state where deadlock would be possible.
- A deadlock-detection algorithm can evaluate processes and resources on a running system to determine if a set of processes is in a deadlocked state.
- If deadlock does occur, a system can attempt to recover from the deadlock by either aborting one of the processes in the circular wait or preempting resources that have been assigned to a deadlocked process.

8.10 Practice exercises

EXERCISE 8.10.1: (Problem 8.1 in the 10th edition).

(a) List three examples of deadlocks that are not related to a computer-system environment.

EXERCISE 8.10.2: (Problem 8.2 in the 10th edition).

(a) Suppose that a system is in an unsafe state. Show that it is possible for the threads to complete their execution without entering a deadlocked state.

EXERCISE 8.10.3: (Problem 8.3 in the 10th edition).

Consider the following snapshot of a system:

	Allocation	Max	Available

	Allocation	Max	Available
	A B C D	*A B C D*	*A B C D*
	0 0 1 2	0 0 1 2	1 5 2 0
	1 0 0 0	1 7 5 0	
	1 3 5 4	2 3 5 6	
	0 6 3 2	0 6 5 2	
	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

(a) What is the content of the matrix ***Need***?

(b) Is the system in a safe state?

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

EXERCISE 8.10.4: (Problem 8.4 in the 10th edition).

(a) A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects ⋯ , deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent deadlock by adding a sixth object . Whenever a thread wants to acquire the synchronization lock for any object ⋯ , it must first acquire the lock for object . This solution is known as ***containment***: the locks for objects ⋯ are contained within the lock for object . Compare this scheme with the circular-wait scheme of Section Circular wait.

EXERCISE 8.10.5: (Problem 8.5 in the 10th edition).

(a) Prove that the safety algorithm presented in Section Banker's algorithm requires an order of operations.

EXERCISE 8.10.6: (Problem 8.6 in the 10th edition).

(a) Consider a computer system that runs 5,000 jobs per month and has no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and

the operator must terminate and rerun about ten jobs per deadlock. Each job is worth about two dollars (in CPU time), and the jobs terminated tend to be about half done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase of about 10 percent in the average execution time per job. Since the machine currently has 30 percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?
b. What are the arguments against installing the deadlock-avoidance algorithm?

EXERCISE 8.10.7: (Problem 8.7 in the 10th edition).

(a) Can a system detect that some of its threads are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

EXERCISE 8.10.8: (Problem 8.8 in the 10th edition).

Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources. If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread. The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.

For example, a system has three resource types, and the vector ***Available*** is initialized to (4,2,2). If thread asks for (2,2,1), it gets them. If asks for (1,0,1), it gets them. Then, if asks for (0,0,1), it is blocked (resource not available). If now asks for (2,0,0), it gets the available one (1,0,0), as well as one that was allocated to (since is blocked). 's ***Allocation*** vector goes down to (1,2,1), and its ***Need*** vector goes up to (1,0,1).

(a) Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur.

(b) Can indefinite blocking occur? Explain your answer.

EXERCISE 8.10.9: (Problem 8.9 in the 10th edition).

Consider the following snapshot of a system:

	Allocation	Max
	A B C D	*A B C D*
	3 0 1 4	5 1 1 7
	2 2 1 0	3 2 1 1
	3 1 2 1	3 3 2 1
	0 5 1 0	4 6 1 2
	4 2 1 2	6 3 2 5

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

(a) ***Available*** = (0,3,0,1)

(b) ***Available*** = (1,0,0,2)

EXERCISE 8.10.10: (Problem 8.10 in the 10th edition).

(a) Suppose that you have coded the deadlock-avoidance safety algorithm that determines if a system is in a safe state or not, and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining $\boldsymbol{Max}_i = \boldsymbol{Waiting}_i + \boldsymbol{Allocation}_i$, where $\boldsymbol{Waiting}_i$ is a vector specifying the resources for which thread is waiting and $\boldsymbol{Allocation}_i$ is as defined in Section 8.6? Explain your answer.

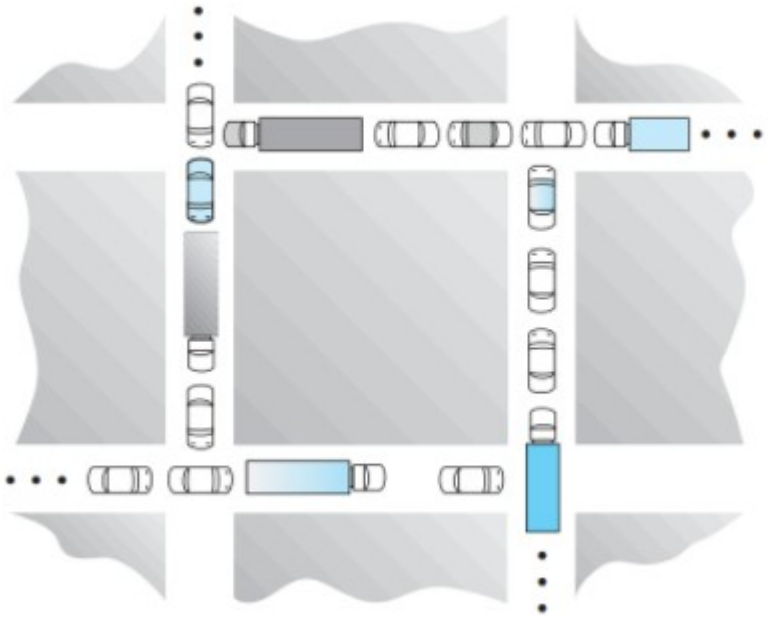
EXERCISE 8.10.11: (Problem 8.11 in the 10th edition).

(a) Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

8.11 Exercises

EXERCISE 8.11.1: (Problem 8.12 in the 10th edition).

Consider the traffic deadlock depicted in the figure below.



(a) Show that the four necessary conditions for deadlock hold in this example.

(b) State a simple rule for avoiding deadlocks in this system.

EXERCISE 8.11.2: (Problem 8.13 in the 10th edition).

(a) Draw the resource-allocation graph that illustrates deadlock from the program example shown in Figure 8.2.1 in Section 8.2.

EXERCISE 8.11.3: (Problem 8.14 in the 10th edition).

(a) In Section Deadlock, we described a potential deadlock scenario involv-ing processes and and semaphores S and Q. Draw the resource-allocation graph that illustrates deadlock under the scenario presented in that section.

EXERCISE 8.11.4: (Problem 8.15 in the 10th edition).

(a) Assume that a multithreaded application uses only reader-writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader-writer locks are used?

EXERCISE 8.11.5: (Problem 8.16 in the 10th edition).

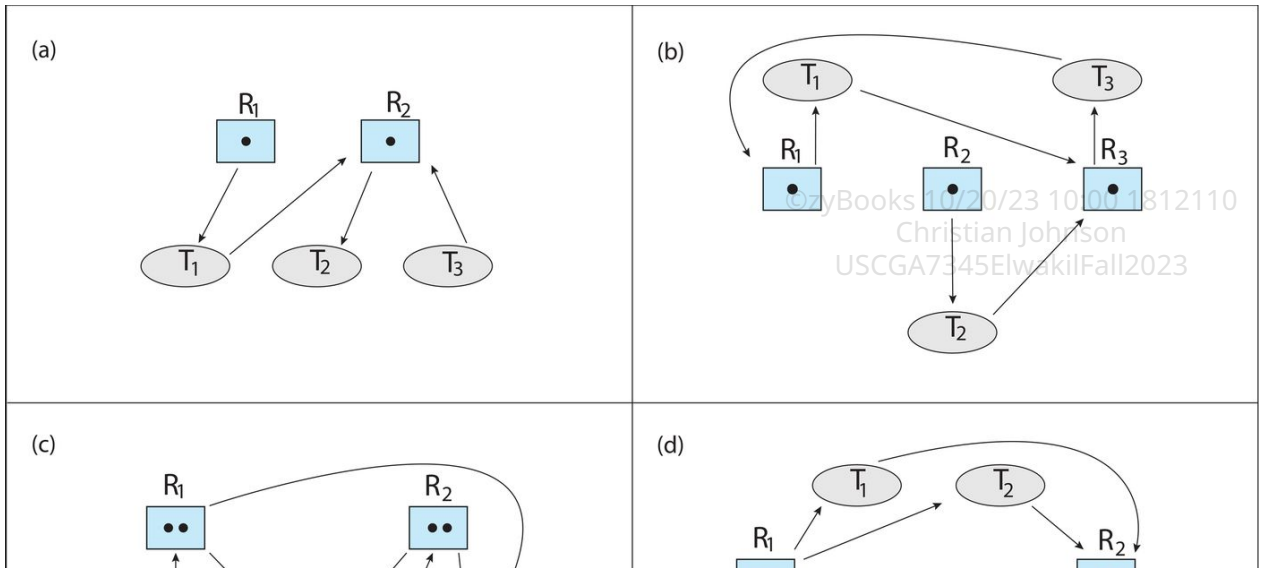
(a) The program example shown in Figure 8.2.1 doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.

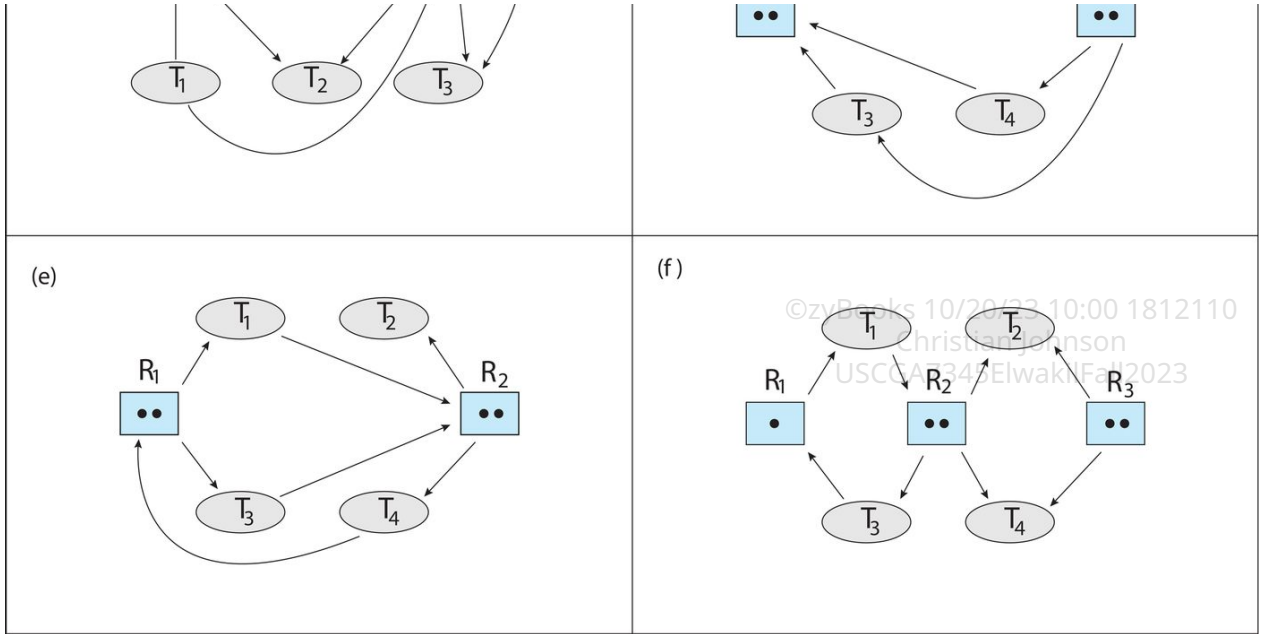
EXERCISE 8.11.6: (Problem 8.17 in the 10th edition).

(a) In Section No preemption, we described a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

EXERCISE 8.11.7: (Problem 8.18 in the 10th edition).

(a) Which of the six resource-allocation graphs shown in the figure below illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.





EXERCISE 8.11.8: (Problem 8.19 in the 10th edition).

Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

(a) Runtime overhead

(b) System throughput

EXERCISE 8.11.9: (Problem 8.20 in the 10th edition).

(a) In a real computer system, neither the resources available nor the demands of threads for resources are consistent over long periods (months). Resources break or are replaced, new processes and threads come and go, and new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase ***Available*** (new resources added).
b. Decrease ***Available*** (resource permanently removed from system).
c. Increase ***Max*** for one thread (the thread needs or wants more resources than allowed).
d. Decrease ***Max*** for one thread (the thread decides it does not need that many resources).
e. Increase the number of threads.

f. Decrease the number of threads.

EXERCISE 8.11.10: (Problem 8.21 in the 10th edition).

(a) Consider the following snapshot of a system:

	Allocation	Max
	A B C D	*A B C D*
	2 1 0 6	6 3 2 7
	3 3 1 3	5 4 1 5
	2 3 1 2	6 6 1 4
	1 2 3 4	4 3 4 5
	3 0 3 0	7 2 6 1

What are the contents of the ***Need*** matrix?

EXERCISE 8.11.11: (Problem 8.22 in the 10th edition).

(a) Consider a system consisting of four resources of the same type that are shared by three threads, each of which needs at most two resources. Show that the system is deadlock free.

EXERCISE 8.11.12: (Problem 8.23 in the 10th edition).

Consider a system consisting of resources of the same type being shared by threads. A thread can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

(a) The maximum need of each thread is between one resource and resources.

(b) The sum of all maximum needs is less than + .

EXERCISE

8.11.13: (Problem 8.24 in the 10th edition).

(a) Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

EXERCISE 8.11.14: (Problem 8.25 in the 10th edition).

(a) Consider again the setting in the preceding exercise. Assume now that each philosopher requires three chopsticks to eat. Resource requests are still issued one at a time. Describe some simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

EXERCISE 8.11.15: (Problem 8.26 in the 10th edition).

(a) We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1.

Show through an example that we cannot implement the multiple-resource-type banker's scheme by applying the single-resource-type scheme to each resource type individually.

EXERCISE 8.11.16: (Problem 8.27 in the 10th edition).

Consider the following snapshot of a system:

	Allocation	Max
	A B C D	*A B C D*
	1 2 0 2	4 3 1 6
	0 1 1 2	2 4 2 4
	1 2 4 0	3 6 5 1
	1 2 0 1	2 6 2 3
	1 0 0 1	3 1 1 2

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

(a) ***Available*** = (2,2,2,3)

(b) ***Available*** = (4,4,1,1)

(c) ***Available*** = (3,0,1,4)

(d) ***Available*** = (1,5,2,2)

EXERCISE 8.11.17: (Problem 8.28 in the 10th edition).

Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	*A B C D*	*A B C D*
	3 1 4 1	6 4 7 3	2 2 2 4
	2 1 0 2	4 2 3 2	
	2 4 1 3	2 5 3 3	
	4 1 1 0	6 3 3 2	
	2 2 2 1	5 6 7 5	

Answer the following questions using the banker's algorithm:

(a) Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

(b) If a request from thread arrives for (2, 2, 2, 4), can the request be granted immediately?

(c) If a request from thread arrives for (0, 1, 1, 0), can the request be granted immediately?

(d) If a request from thread arrives for (2, 2, 1, 2), can the request be granted immediately?

EXERCISE 8.11.18: (Problem 8.29 in the 10th edition).

(a) What is the optimistic assumption made in the deadlock-detection algorithm? How can this assumption be violated?

EXERCISE 8.11.19: (Problem 8.30 in the 10th edition).

(a) A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

EXERCISE 8.11.20: (Problem 8.31 in the 10th edition).

(a) Modify your solution to the previous exercise so that it is starvation-free.

8.12 Programming problems

EXERCISE 8.12.1: (Problem 8.32 in the 10th edition).

(a) Implement your solution to Exercise 8.12.19 using POSIX synchronization. In particular, represent northbound and southbound farmers as separate threads. Once a farmer is on the bridge, the associated thread will sleep for a random period of time, representing traveling across the bridge. Design your program so that you can create several threads representing the northbound and southbound farmers.

EXERCISE 8.12.2: (Problem 8.33 in the 10th edition).

(a) In Figure 8.5.1, we illustrate a `transaction()` function that dynamically acquires locks. In the text, we describe how this function presents difficulties for acquiring locks in a way that avoids deadlock. Using the Java implementation of `transaction()` that is provided in the source-code download for this text, modify it using the

`System.identityHashCode()` method so that the locks are acquired in order.

8.13 Programming projects

Banker's algorithm

For this project, you will write a program that implements the banker's algorithm discussed in Section Banker's algorithm. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. Although the code examples that describe this project are illustrated in C, you may also develop a solution using Java.

The banker

The banker will consider requests from customers for resources types, as outlined in Section Banker's algorithm. The banker will keep track of the resources using the following data structures:

```
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 4

/* the available amount of each resource */
int available[NUMBER_OF_RESOURCES];

/*the maximum demand of each customer */
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the amount currently allocated to each customer */
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

/* the remaining need of each customer */
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

The banker will grant a request if it satisfies the safety algorithm outlined in Section Safety algorithm. If a request does not leave the system in a safe state, the banker will deny it. Function prototypes for requesting and releasing resources are as follows:


```
int request_resources(int customer_num, int request[]);

void release_resources(int customer_num, int release[]);
```

The `request_resources()` function should return 0 if successful and −1 if unsuccessful.

Testing your implementation

Design a program that allows the user to interactively enter a request for resources, to release resources, or to output the values of the different data structures (`available`, `maximum`, `allocation`, and need) used with the banker's algorithm.

You should invoke your program by passing the number of resources of each type on the command line. For example, if there were four resource types, with ten instances of the first type, five of the second type, seven of the third type, and eight of the fourth type, you would invoke your program as follows:


```
./a.out 10 5 7 8
```

The `available` array would be initialized to these values.

Your program will initially read in a file containing the maximum number of requests for each customer. For example, if there are five customers and four resources, the input file would appear as follows:

```
6,4,7,3
4,2,3,2
2,5,3,3
6,3,3,2
5,6,7,5
```

where each line in the input file represents the maximum request of each resource type for each customer. Your program will initialize the `maximum` array to these values.

Your program will then have the user enter commands responding to a request of resources, a release of resources, or the current values of the different data structures. Use the command 'RQ' for requesting resources, 'RL' for releasing resources, and '*' to output the values of the different data structures. For example, if customer 0 were to request the resources (3, 1, 2, 1), the following command would be entered:

```
RQ 0 3 1 2 1
```

Your program would then output whether the request would be satisfied or denied using the safety algorithm outlined in Section Safety algorithm.

Similarly, if customer 4 were to release the resources (1, 2, 3, 1), the user would enter the following command:


```
RL 4 1 2 3 1
```

Finally, if the command '*' is entered, your program would output the values of the `available`, `maximum`, `allocation`, and need arrays.

8.14 Further reading

Most research involving deadlock was conducted many years ago. [Dijkstra (1965)] was one of the first and most influential contributors in the deadlock area.

Details of how the MySQL database manages deadlock can be found at http://dev.mysql.com/.

Details on the `lockdep` tool can be found at https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

8.15 Bibliography

[Dijkstra (1965)] E. W. Dijkstra, "*Cooperating Sequential Processes*", Technical report, Technological University, Eindhoven, the Netherlands (1965).