Deadlocks

Practice Exercises

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

Answer:

- Two cars crossing a single-lane bridge from opposite directions.
- A person going down a ladder while another person is climbing up the ladder.
- Two trains traveling toward each other on the same track.

8.2 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.

Answer:
An unsafe state may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. Consider the situation where a system has twelve resources allocated among processes P_0, P_1, and P_2. The resources are allocated according to the following policy:

	Max	Current	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

Currently, there are two resources available. This system is in an unsafe state. Process P_1 could complete, thereby freeing a total of four resources, but we cannot guarantee that processes P_0 and P_2 can complete. However, it is possible that a process may release resources before requesting any further resources. For example, process P_2 could release a resource, thereby increasing the total number of resources to five. This allows pro-

cess P_0 to complete, which would free a total of nine resources, thereby allowing process P_2 to complete as well.

8.3 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	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 T_1 arrives for (0,4,2,0), can the request be granted immediately?

Answer:

a. The values of ***Need*** for processes P_0 through P_4, respectively, are (0, 0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0), and (0, 6, 4, 2).

b. The system is in a safe state. With ***Available*** equal to (1, 5, 2, 0), either process P_0 or P_3 could run. Once process P_3 runs, it releases its resources, which allows all other existing processes to run.

c. The request can be granted immediately. The value of ***Available*** is then (1, 1, 0, 0). One ordering of processes that can finish is P_0, P_2, P_3, P_1, and P_4.

8.4 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 $A \cdots E$, deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent deadlock by adding a sixth object F. Whenever a thread wants to acquire the synchronization lock for any object $A \cdots E$, it must first acquire the lock for object F. This solution is known as **containment**: the locks for objects $A \cdots E$ are contained within the lock for object F. Compare this scheme with the circular-wait scheme of Section 8.5.4.

Answer:

This is probably not a good solution because it yields too large a scope. It is better to define a locking policy with as narrow a scope as possible. The circular wait approach is a reasonable approach to avoiding deadlock, and does not increase the scope of holding a lock.

8.5 Prove that the safety algorithm presented in Section 8.6.3 requires an order of $m \times n^2$ operations.

Answer:

The figure below provides Java code that implements the safety algorithm of the banker's algorithm (the complete implementation of the banker's algorithm is available with the source-code download for this text).

```
for (int i = 0; i < n; i++) {
   // first find a thread that can finish
   for (int j = 0; j < n; j++) {
      if (!finish[j]) {
         boolean temp = true;
         for (int k = 0; k < m; k++) {
            if (need[j][k] > work[k])
               temp = false;
         }

         if (temp) { // if this thread can finish
            finish[j] = true;
            for (int x = 0; x < m; x++)
               work[x] += work[j][x];
         }
      }
   }
}
```

As can be seen, the nested outer loops—both of which loop through n times—provide the n^2 performance. Within these outer loops are two sequential inner loops that loop m times. The Big O of this algorithm is therefore $O(m \times n^2)$.

8.6 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?

Answer:

An argument for installing deadlock avoidance in the system is that we could ensure that deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run.
An argument against installing deadlock-avoidance software is that deadlocks occur infrequently, and they cost little when they do occur.

8.7 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.

Answer:
Starvation is a difficult topic to define, as it may mean different things for different systems. For the purposes of this question, we will define starvation as the situation in which a process must wait beyond a reasonable period of time—perhaps indefinitely—before receiving a requested resource. One way of detecting starvation would be to first identify a period of time—T—that is considered unreasonable. When a process requests a resource, a timer is started. If the elapsed time exceeds T, then the process is considered to be starved.

One strategy for dealing with starvation would be to adopt a policy whereby resources are assigned only to the process that has been waiting the longest. For example, if process P_a has been waiting longer for resource X than process P_b, the request from process P_b would be deferred until process P_a's request has been satisfied.

Another strategy would be less strict. In this scenario, a resource might be granted to a process that had waited less than another process, providing that the other process was not starving. However, if another process was considered to be starving, its request would be satisfied first.

8.8 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 T_0 asks for (2,2,1), it gets them. If T_1 asks for (1,0,1), it gets them. Then, if T_0 asks for (0,0,1), it is blocked (resource not available). If T_2 now asks for (2,0,0), it gets the available one (1,0,0), as well as one that was allocated to T_0 (since T_0 is blocked). T_0'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.

Answer:

a. Deadlock cannot occur, because preemption exists.

b. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process *C*.

8.9 Consider the following snapshot of a system:

	Allocation	***Max***
	A B C D	*A B C D*
T_0	3 0 1 4	5 1 1 7
T_1	2 2 1 0	3 2 1 1
T_2	3 1 2 1	3 3 2 1
T_3	0 5 1 0	4 6 1 2
T_4	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. $\boldsymbol{Available} = (0, 3, 0, 1)$

b. $\boldsymbol{Available} = (1, 0, 0, 2)$

Answer:

a. Not safe. Processes P_2, P_1, and P_3 are able to finish, but no remaining processes can finish.

b. Safe. Processes P_1, P_2, and P_3 are able to finish. Following this, processes P_0 and P_4 are also able to finish.

8.10 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 i is waiting and $\boldsymbol{Allocation}_i$ is as defined in Section 8.6? Explain your answer.

Answer:
Yes. The *Max* vector represents the maximum request a process may make. When calculating the safety algorithm, we use the *Need* matrix, which represents *Max* — *Allocation*. Another way to think of this is *Max* = *Need* + *Allocation*. According to the question, the *Waiting* matrix fulfills a role similar to the *Need* matrix; therefore, *Max* = *Waiting* + *Allocation*.

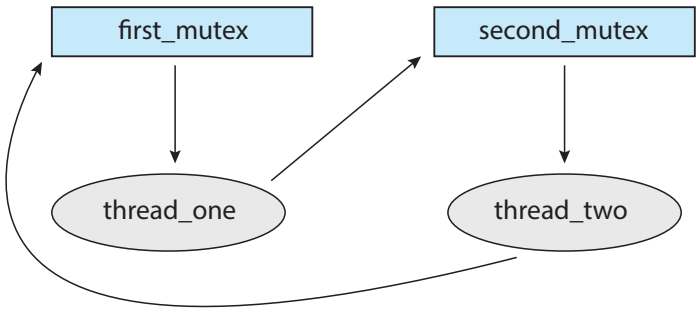
8.11 Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

Answer:
No. This follows directly from the hold-and-wait condition.

Exercises

8.12 Draw the resource-allocation graph that illustrates deadlock from the program example shown in Figure 8.1 in Section 8.2.

Answer:



8.13 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?

Answer:
Yes. (1) Mutual exclusion is maintained, as locks cannot be shared if there is a writer. (2) Hold and wait is possible, as a thread can hold one reader—writer lock while waiting to acquire another. (3) You cannot take a lock away, so no preemption is upheld. (4) A circular wait among all threads is possible.

8.14 The program example shown in Figure 8.1 doesn't always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.

Answer:
If `thread_one` is scheduled before `thread_two` and `thread_one` is able to acquire both mutex locks before `thread_two` is scheduled, deadlock will not occur. Deadlock can only occur if either `thread_one` or `thread_two` is able to acquire only one lock before the other thread acquires the second lock.

8.15 Which of the six resource-allocation graphs shown in Figure 8.12 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.

Answer:

a. (a) No deadlock. There is no cycle. T2 will finish first, followed by T1 and T3 in any order.

b. (b) Deadlock, as there is a cycle T1 -> R3 -> T3 -> R1 -> T1.

c. (c) No deadlock. There is no cycle. T3, T2, followed by T1.

d. (d) Deadlock, as there is a cycle T1 -> R2 -> T4 -> R1 -> T1.

e. (e) No deadlock. Although there is a cycle in Figure (e), R 1 -> T3 -> R2 -> T4 -> R4, if T2 releases R2, this either allows T3 to acquire R2, which breaks the cycle, or allows T1 to acquire R2. If T1 acquires R2, it will ultimately release both R1 and R2, which will also break the cycle.

f. (f) No deadlock. Although there is a cycle in Figure (f), T1 -> R2 -> T3 -> R1 -> T1, when either T2 or T4 finishes, it releases R2, which breaks the cycle.

8.16 Consider the following snapshot of a system:

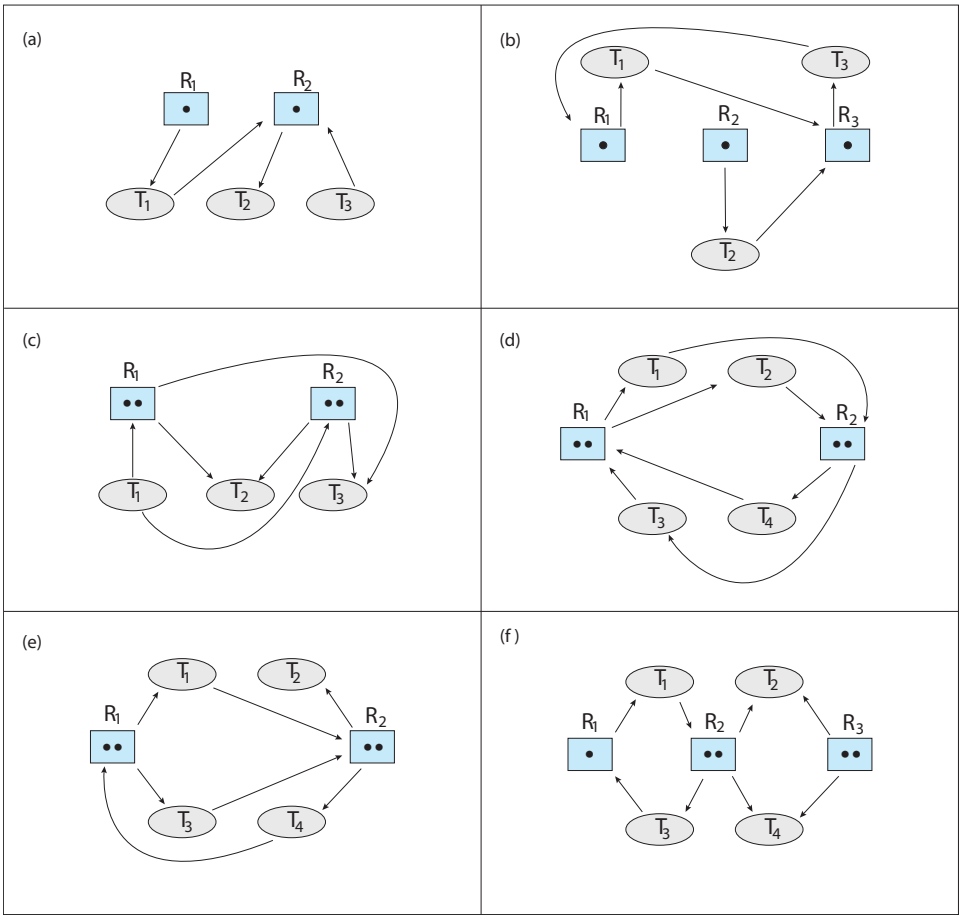


Figure 8.12 Resource-allocation graphs for Exercise 8.15.

	Allocation	*Max*
	A B C D	*A B C D*
T_0	2 1 0 6	6 3 2 7
T_1	3 3 1 3	5 4 1 5
T_2	2 3 1 2	6 6 1 4
T_3	1 2 3 4	4 3 4 5
T_4	3 0 3 0	7 2 6 1

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

Answer:
(4,2,2,1) (2,1,0,2) (4,3,0,2) (3,1,1,1) (4,2,3,1)

8.17 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.

Answer:
The following rule prevents deadlock: when a philosopher makes a request for the first chopstick, do not grant the request if there is no other philosopher with two chopsticks and if there is only one chopstick remaining.

8.18 Consider the following snapshot of a system:

	Allocation	*Max*
	A B C D	*A B C D*
T_0	1 2 0 2	4 3 1 6
T_1	0 1 1 2	2 4 2 4
T_2	1 2 4 0	3 6 5 1
T_3	1 2 0 1	2 6 2 3
T_4	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)

Answer:

a. Yes. One possible ordering is T_4, T_0, T_1, T_2, T_3

b. Yes. One possible ordering is T_2, T_4, T_3, T_1, T_0

c. No.

d. Yes. One possible ordering is T_3, T_1, T_2, T_4, T_0

8.19 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).

Answer:

```
semaphore ok_to_cross = 1;

void enter_bridge() {
    ok_to_cross.wait();
}

void exit_bridge() {
    ok_to_cross.signal();
}
```