

CHAPTER 7

Deadlock is a problem that can arise only in a system with multiple active asynchronous processes. It is important that the students learn the three basic approaches to deadlock: prevention, avoidance, and detection (although the terms *prevention* and *avoidance* are easy to confuse).

It can be useful to pose a deadlock problem in human terms and ask why human systems never deadlock. Can the students transfer this understanding of human systems to computer systems?

Projects can involve simulation: create a list of jobs consisting of requests and releases of resources (single type or multiple types). Ask the students to allocate the resources to prevent deadlock. This basically involves programming the Banker's Algorithm.

The survey paper by Coffman, Elphick, and Shoshani [1971] is good supplemental reading, but you might also consider having the students go back to the papers by Havender [1968], Habermann [1969], and Holt [1971a]. The last two were published in *CACM* and so should be readily available.

Exercises

7.1 Consider the traffic deadlock depicted in Figure 7.10.

- a. Show that the four necessary conditions for deadlock indeed hold in this example.
- b. State a simple rule for avoiding deadlocks in this system.

Answer:

- a. The four necessary conditions for a deadlock are (1) mutual exclusion; (2) hold-and-wait; (3) no preemption; and (4) circular wait. The mutual exclusion condition holds since only one car can occupy a space in the roadway. Hold-and-wait occurs where a car holds onto its place in the roadway while it waits to advance in the roadway. A car cannot be removed (i.e. preempted) from its position in the roadway. Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance. The circular wait condition is also easily observed from the graphic.
- b. A simple rule that would avoid this traffic deadlock is that a car may not advance into an intersection if it is clear it will not be able immediately to clear the intersection.

7.2 Assume 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 they 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.

7.3 The program example shown in Figure 7.4 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.

7.4 In Section 7.4.4, we describe 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.

Answer:

Add a new lock to this function. This third lock must be acquired before the two locks associated with the accounts are acquired. The `transaction()` function now appears as follows:

```
void transaction(Account from, Account to, double amount)
{
    Semaphore lock1, lock2, lock3;
    wait(lock3);
    lock1 = getLock(from);
    lock2 = getLock(to);

    wait(lock1);
        wait(lock2);

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

                signal(lock3);
                signal(lock2);
                signal(lock1);
}
```

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

- a. Runtime overheads
- b. System throughput

Answer:

A deadlock-avoidance scheme tends to increase the runtime overheads due to the cost of keep track of the current resource allocation. However, a deadlock-avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. In that sense, a deadlock-avoidance scheme could increase system throughput.

7.6 In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, 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 process (the process needs or wants more resources than allowed).
- d. Decrease **Max** for one process (the process decides it does not need that many resources)
- e. Increase the number of processes
- f. Decrease the number of processes

Answer:

- a. Increase **Available** (new resources added)—This could safely be changed without any problems.
- b. Decrease **Available** (resource permanently removed from system)—This could have an effect on the system and introduce the possibility of deadlock as the safety of the system assumed there were a certain number of available resources.
- c. Increase **Max** for one process (the process needs more resources than allowed, it may want more)—This could have an effect on the system and introduce the possibility of deadlock.
- d. Decrease **Max** for one process (the process decides it does not need that many resources)—This could safely be changed without any problems.

- e. Increase the number of processes—This could be allowed assuming that resources were allocated to the new process(es) such that the system does not enter an unsafe state.
- f. Decrease the number of processes—This could safely be changed without any problems.

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

Answer:

Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and, therefore it will return its resources when done.

7.8 Consider a system consisting of m resources of the same type being shared by n processes. A process 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 process is between 1 and m resources
- b. The sum of all maximum needs is less than $m + n$

Answer:

Using the terminology of Section Section 7.6.2, we have:

- a. $\sum_{i=1}^n Max_i < m + n$
- b. $Max_i \geq 1$ for all i

Proof: $Need_i = Max_i - Allocation_i$

If there exists a deadlock state then:

- c. $\sum_{i=1}^n Allocation_i = m$

Use a. to get: $\sum Need_i + \sum Allocation_i = \sum Max_i < m + n$

Use c. to get: $\sum Need_i + m < m + n$

Rewrite to get: $\sum_{i=1}^n Need_i < n$

This implies that there exists a process P_i such that $Need_i = 0$. Since $Max_i \geq 1$ it follows that P_i has at least one resource that it can release. Hence the system cannot be in a deadlock state.

7.9 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.

7.10 Consider again the setting in the preceding question. 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.

Answer:

When a philosopher makes a request for a chopstick, allocate the request if: 1) the philosopher has two chopsticks and there is at least one chopstick remaining, 2) the philosopher has one chopstick and there are at least two chopsticks remaining, 3) there is at least one chopstick remaining, and there is at least one philosopher with three chopsticks, 4) the philosopher has no chopsticks, there are two chopsticks remaining, and there is at least one other philosopher with two chopsticks assigned.

7.11 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.

Answer:

Consider a system with resources A , B , and C and processes P_0 , P_1 , P_2 , P_3 , and P_4 with the following values of *Allocation*:

Allocation			
	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

and the following value of *Need*:

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

If the value of *Available* is (2 3 0), we can see that a request from process P_0 for (0 2 0) cannot be satisfied as this lowers *Available* to (2 1 0) and no process could safely finish.

However, if we treat the three resources as three single-resource types of the banker's algorithm, we get the following:

For resource A (of which we have 2 available),

	Allocated	Need
P_0	0	7
P_1	3	0
P_2	3	6
P_3	2	0
P_4	0	4

Processes could safely finish in the order P_1, P_3, P_4, P_2, P_0 .

For resource B (of which we now have 1 available as 2 were assumed assigned to process P_0),

	Allocated	Need
P_0	3	2
P_1	0	2
P_2	0	0
P_3	1	1
P_4	0	3

Processes could safely finish in the order P_2, P_3, P_1, P_0, P_4 .

And finally, for resource C (of which we have 0 available),

	Allocated	Need
P_0	0	3
P_1	2	0
P_2	2	0
P_3	1	1
P_4	2	1

Processes could safely finish in the order P_1, P_2, P_0, P_3, P_4 .

As we can see, if we use the banker's algorithm for multiple resource types, the request for resources (0 2 0) from process P_0 is denied as it leaves the system in an unsafe state. However, if we consider the banker's algorithm for the three separate resources where we use a single resource type, the request is granted. Therefore, if we have multiple resource types, we must use the banker's algorithm for multiple resource types.

7.12 Consider the following snapshot of a system:

	<u>Allocation</u>	<u>Max</u>
	<i>A B C D</i>	<i>A B C D</i>
P_0	3 0 1 4	5 1 1 7
P_1	2 2 1 0	3 2 1 1
P_2	3 1 2 1	3 3 2 1
P_3	0 5 1 0	4 6 1 2
P_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 processes may complete. Otherwise, illustrate why the state is unsafe.

- Available** = (0, 3, 0, 1)
- Available** = (1, 0, 0, 2)

Answer:

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

7.13 Consider the following snapshot of a system:

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

Answer the following questions using the banker's algorithm:

- Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.
- If a request from process P_1 arrives for (1, 1, 0, 0), can the request be granted immediately?
- If a request from process P_4 arrives for (0, 0, 2, 0), can the request be granted immediately?

7.14 What is the optimistic assumption made in the deadlock-detection algorithm? How could this assumption be violated?

Answer:

The optimistic assumption is that there will not be any form of circular wait in terms of resources allocated and processes making requests for them. This assumption could be violated if a circular wait does indeed occur in practice.

7.15 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 both 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, design an algorithm that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, and vice versa).

Answer:

```
semaphore ok_to_cross = 1;

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

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

7.16 Modify your solution to Exercise 7.15 so that it is starvation-free.

Answer:

```
monitor bridge {
    int num_waiting_north = 0;
    int num_waiting_south = 0;
    int on_bridge = 0;
    condition ok_to_cross;
    int prev = 0;

    void enter_bridge_north() {
        num_waiting_north++;
        while (on_bridge ||
            (prev == 0 && num_waiting_south > 0))
            ok_to_cross.wait();
        num_waiting_north--;
        prev = 0;
    }

    void exit_bridge_north() {
        on_bridge = 0;
        ok_to_cross.broadcast();
    }

    void enter_bridge_south() {
        num_waiting_south++;
        while (on_bridge ||
            (prev == 1 && num_waiting_north > 0))
            ok_to_cross.wait();
        num_waiting_south--;
    }
}
```

```
        prev = 1;
    }

    void exit_bridge_south() {
        on_bridge = 0;
        ok_to_cross.broadcast();
    }
}
```