CHAPTER 7

Deadlocks

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.

7.1

```
semaphore ok_to_cross = 1;

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

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

7.2

```
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();
    }
}
```

7.3 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.

7.4 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.

7.5 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.

7.6 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 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.

7.7 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.

7.8 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.

7.9 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.

7.10

a. What is the content of the matrix *Need*? 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. Is the system in a safe state? Yes. With *Available* being 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 allow all other existing processes to run.

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately? Yes, it can. This results in the value of *Available* being (1, 1, 0, 0). One ordering of processes that can finish is P_0, P_2, P_3, P_1, and P_4.

7.11 Using the terminology of 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.

7.12 An argument for installing deadlock avoidance in the system is that we could ensure 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.

7.13 Deadlock is possible because the four necessary conditions hold in the following manner: 1) mutual exclusion is required for chopsticks, 2) the philosophers hold onto the chopstick in hand while they wait for the other chopstick, 3) there is no preemption of chopsticks in the sense that a chopstick allocated to a philosopher cannot be forcibly taken away, and 4) there is a possibility of circular wait. Deadlocks could be avoided by overcoming the conditions in the following manner: 1) allow simultaneous sharing of chopsticks, 2) have the philosophers relinquish the first chopstick if they are unable to obtain the other chopstick, 3) allow chopsticks to be forcibly taken away if a philosopher has had a chopstick for a long period of time, and 4) enforce a numbering of the chopsticks and always obtain the lower numbered chopstick before obtaining the higher numbered one.

7.14 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 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.

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

7.17 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.

7.18 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);
}
```