7.Deadlock

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 that will avoid deadlocks in this system.
Answer:
Each cross of the streets is considered as a resource, each line of cars is considered as a process. Mutual exclution: only one line of cars at a time can use the resource. Hold and wait: Each line of cars is holding one resource and is waiting for the next resource. No preemption: the resouce can not be released until the whole line of cars have passed it. Circular wait: there are 4 lines of cars l1, l2, l3, l4, l1 is waiting for l2, l2 is waiting for l3, l3 is waiting for l4, l4 is waiting for l1.
There are many ways to avoid the deadlocks in this system. one way is break the second condition: a line of cars can't hold a cross and wait, it's that no car of a line can stay in the cross.

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, 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 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:

The system is only deadlocked if a process cannot access the maximum amount of resources it needs. If three processes need a maximum of 2 resources each and the system has 4 of those resources total, that means one of the processes must already have the maximum amount of resources it needs to complete. After that processes completes, more resources are freed and the other processes can complete too.

7.8 Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one 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.

Proof:

Suppose N = Sum of all Need(i), A = Sum of all Allocation(i), M = Sum of all Max(i). Use contradiction to prove.

Assume this system is not deadlock free. If there exists a deadlock state, then A = m because there's only one kind of resource and resources can be requested and released only one at a time. From condition b, N + A = M < m + n. So we get N + m < m + n. So we get N < n. It shows that at least one process i that Need(i) = 0. From condition a, Pi can release at least 1 resource. So there are n-1 processes sharing m resources now, condition a and b still hold. Go on the argument, no process will wait permanently, so there's no deadlock.

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:

When a philosopher makes a request for their first chopstick, do not satisfy the request only

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 P0, P1, P2, P3, and P4 with the following values of Allocation:

and the following value of Need:
If the value of Available is (2 3 0), we can see that a request from process P0 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),

Processes could safely finish in the order P1, P3, P4, P2, P0.
For resource B (of which we now have 1 available as 2 were assumed assigned to process P0),

Processes could safely finish in the order P2, P3, P1, P0, P4.
And finally, for resource C (of which we have 0 available),

Processes could safely finish in the order P1, P2, P0, P3, P4.

As we can see, if we use the banker's algorithm for multiple resource types, the request for resources (0 2 0) from process P0 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.

7.13 Consider the following snapshot of a system:

	Allocation	Max	Available
	ABCD	ABCD	ABCD
P0	0012	0012	1520
P1	1000	1750	
P2	1354	2356	
P3	0632	0652	
P4	0014	0656	

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 process P1 arrives for (0,4,2,0), can the request be granted immediately?

Answer:

a. The values of Need for processes P0 through P4 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? Yes. With Available being equal to (1, 5, 2, 0), either process P0 or P3 could run. Once process P3 runs, it releases its resources, which allow all other existing processes to run.

c. The request can be granted immediately? This results in the value of Available being (1, 1, 0, 0). One ordering of processes that can finish is P0, P2, P3, P1, and P4.

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();
        }
}
```