Homework #3 Solution

Assigned problems are: 7.2, 7.7, Section 7.8 questions, 8.4, and 8.13

Problem 7.2

Explain why spinlocks are not appropriate for uniprocessor systems, yet may be suitable for multiprocessor systems.

Spinlocks consume CPU resources, which may be a problem if you only have one CPU. That one CPU may be needed to resolve whatever situation the spinlock is waiting for.

In a multiprocessor setup, then a spinlock is more appropriate you have multiple CPUs available to do work. Also, a context change is more expensive on a multiprocessor system and may outweigh the cost of the spinlock as well.

Problem 7.7

Show that, if the wait and signal operations are not executed atomically, then mutual exclusion may be violated.

If wait() and/or signal() are not atomic then the increment and decrement of the semaphore value may be performed incorrectly when executed concurrently.

This increment/decrement anamoly was shown in the beginning of Chapter 7 on pages 190-191, exhibiting the nature of race conditions and the need for process synchronization. This is the same reason why wait() and signal() must be atomic.

Also, if 2 wait() functions are incorrectly run concurrently, as a result of not being atomic, then you can have two (or more) processes entering their critical section at one time.

Likewise, if 2 signal() functions are incorrectly run, then you may block all future access to any critical sections and deadlock the system.

Section 7.8 questions

a) Solaris: What is an adaptive mutex and how does it work?

The adaptive mutex in the Solaris operating system is a mutex with special cases for the number of processors in a system.

- On uniprocessor systems, an adaptive mutex never uses a spinlock when waiting to protect a critical section.
- On multiprocessor systems, the adaptive mutex will spinlock on the processor on which the thread is already running. However, if the thread is not currently running, then it simply blocks until the mutex lock is available.

b) Windows 2000: What are dispatcher objects and how do they work?

In Windows 2000, dispatcher objects are used for thread synchronization outside of the kernel. Dispatcher objects can be used to build sempahores, mutexes or events.

A dispatcher object is either in a signaled or nonsignaled state. In a signaled state, the protected object is available, and the thread is not blocked. In nonsignaled, then the thread blocks until the dispatcher objects lock is available.

Problem 8.4

Consider the traffic deadlock depicted in Figure 8.8.

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

The four conditions necessary for deadlock to be possible are:

1. **mutual exclusion** - only one car can occupy each intersection at a time
2. **hold and wait** - cars can hold an intersection while waiting in a line for access to the next intersection
3. **no preemption** - cars cannot be removed from their spot in the traffic flow, except by moving forward
4. **circular wait** - the set of cars in the deadlock situation includes the cars in the middle of the intersection

b) State a simple rule that will avoid deadlocks in this system.

Install traffic lights that only allow flow in one direction or the other at a time.

You can still envision a possible deadlock if a city block is completely full of cars turning left or right. I think you'd need to add a criteria to the problem requiring that cars will eventually leave the city block as well to prevent this.

Problem 8.13

Answer the following questions using the Banker's Algorithm

a) What is the content of the matrix "Need"?

The Need matrix = Max matrix values - Allocation matrix values

	Need			
	A	**B**	**C**	**D**
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

---

b) Is the system in a safe state?

Execute the Banker's algorithm to determine whether the system is safe or not.

Iteration #1

The initial Work vector is a copy of the Available vector:

Work = (1, 5, 2, 0)

P0's Need vector <= Work, so mark it finished and add its Allocation to the Work vector, giving:

(updated) Work = Work (1, 5, 2, 0) + P0 Allocation (0, 0, 1, 2) (updated) Work = (1, 5, 3, 2)

Iteration #2

P2's Need vector <= Work, so mark it finished and add its Allocation to the Work vector:

(updated Work = Work (1, 5, 3, 2) + P2 Allocation (1, 3, 5, 4)

(updated) Work = (2, 8, 8, 6)

Iteration #3

P1's Need vector <= Work, so mark it finished and add its Allocation.

(updated) Work = Work (2, 8, 8, 6) + P1 Allocation (1, 0, 0, 0)

(updated) Work = (3, 8, 8, 6)

Iteration #4

P3's Need vector <= Work, so mark it finished and add its Allocation:

(updated Work = Work (3, 8, 8, 6) + P3 Allocation (0, 6, 3, 2)

(updated) Work = (3, 14, 11, 8)

Iteration #5

P5's Need vector <= Work. Mark it finished and we're done!

The system is safe.

---

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

We will only grant P1's request if it still leaves the system in a "safe state". Use the Banker's algorithm to check this.

Adding P1's request means the following changes:

- Adding these resources to P1's Allocation matix entry
- Updating P1's Need matrix accordingly (Need = Max - Allocation)
- Reducing the Available vector to reflect the allocation: Available = (1,5,2,0) - (0,4,2,0) = (1,1,0,0)

Now, with these updated structures, run the Banker's algorithm:

Iteration #1

The initial Work vector is a copy of the Available vector:

Work = (1, 1,0, 0)

P0's Need vector <= Work, so mark it finished and add its Allocation to the Work vector, giving:

(updated) Work = Work (1, 1, 0, 0) + P0 Allocation (0, 0, 1, 2)

(updated) Work = (1, 1, 1, 2)

Iteration #2

P2's Need vector <= Work, so mark it finished and add its Allocation to the Work vector:

(updated) Work = Work (1, 1, 1, 2) + P2 Allocation (1, 3, 5, 4)

(updated) Work = (2, 4, 6, 6)

Iteration #3

P1's Need vector <= Work, so mark it finished and add its Allocation.

(updated) Work = Work (2, 4, 6, 6) + P1 Allocation (1, 4, 2, 0)

(updated) Work = (3, 8, 8, 6)

Iteration #4 & #5 (shortcut)

Our Work vector is now > any remaining Need vectors, so we're done.

The system is safe, so allocation to P1 is OK.