CSE 120 Principles of Computer Operating Systems

Fall Quarter, 2001

Midterm Exam

Instructor: Geoffrey M. Voelker

Name _____

Student ID _____

Attention: This exam has five questions worth a total of 55 points. You have approx. 80 minutes to complete the questions. As with any exam, you should read through the questions first and start with those that you are most comfortable with. If you believe that you cannot answer a question without making some assumptions, state those assumptions in your answer.

1	/10
2	/6
3	/9
4	/15
5	/15
Total	/55

1. (10 pts) Potpourri: Answer yes or no, or with a single term or short answer, as appropriate. You should go through these quickly, answering with the first answer that comes to mind — it probably is the correct one. Only dwell on ones you are unsure of after finishing the other questions.

   (a) Does the test-and-set instruction need to be a privileged instruction?
   No, it can be executed at user level.

   (b) What approach to dealing with deadlock does the Banker's algorithm implement?
   Avoidance.

   (c) Which of the following scheduling algorithms can lead to starvation? FIFO, Shortest Job First, Priority, Round Robin
   SJF, Priority

   (d) A program containing a race condition will always/sometimes/never result in data corruption or some other incorrect behavior?
   Sometimes...just because a race condition exists in code does not mean that a particular execution will encounter it.

   (e) A system that meets the four deadlock conditions will always/sometimes/never result in deadlock?
   Sometimes.

2. (6 pts) Categorize the following as one of the following: (I) interrupt, (E) exception, or (N) neither.

   (a) Timer – *(I)*
   (b) Segmentation violation – *(E)*
   (c) Keyboard input – *(I)*
   (d) Divide by zero – *(E)*
   (e) Procedure call – *(N)*
   (f) System call – *(E)*

3. (9 pts) Discuss the tradeoffs between user and kernel threads.

   (a) What are the advantages and disadvantages of each?

   (b) Assume we can make system calls as fast as procedure calls using some new hardware mechanism. Would this make one kind of thread clearly preferable over the other? Explain briefly.

(a) See slides 21-26 in Lecture 4.

(b) Kernel threads would be preferable. The primary disadvantage of kernel threads is the overhead of trapping to the kernel to manipulate them, context switch, etc. Some people mentioned that kernel threads still have the problem of having to be generic to all application needs, which is a good observation and I gave points for this.

4. (15 pts) The Coronado Bridge is undergoing repairs and only one lane is open for traffic. To prevent accidents, traffic lights have been installed at either end of the bridge to synchronize the traffic going in different directions. A car can only cross the bridge if there are no cars going the opposite direction on the bridge. Sensors at either end of the bridge detect when cars arrive and depart from the bridge, and these sensors control the traffic lights.

   Below is a skeleton implementation of two routines, *Arrive()* and *Depart()*. You may assume that each car is represented by a thread, and threads call *Arrive()* when they arrive at the bridge and *Depart()* when the leave the bridge. Threads pass in their direction of travel as input to the routines.

```
     int num_cars = 0;
     enum dir = {open, north, south};
     dir cur_dir = open;


     void
     Arrive (dir my_dir) {
A.-1   lock->Acquire();
A.0    while (cur_dir != my_dir &&
              cur_dir != open) {
         cv->Wait(lock);
       }

A.1    num_cars++;

A.2    cur_dir = my_dir;
A.3    lock->Release();
     }
```

```
     Lock *lock = new Lock();
     Condition *cv = new Condition();

     void
     Depart (dir my_dir) {
D.-1   lock->Acquire();
D.0    num_cars--;

D.1    if (num_cars == 0) {

D.2      cur_dir = open;
         cv->Broadcast(lock);
D.4    }
D.5    lock->Release();
     }
```

   (a) The code above does not have any synchronization statements. Outline an execution sequence (referring to the numbered statements) where two threads can cause two cars to travel on the bridge in opposite directions at the same time.

   Thread 1 (north): A.0, A.1 (context switch)

   Thread 2 (south): A.0, A.1

   and now we have two threads on the bridge in opposite directions.

   (b) Show how a condition variable and lock could be used to correctly synchronize the cars by annotating the above routines with calls to Condition and Lock operations.

   See above.

   (c) Draw brackets around the sections of code that are critical sections.

   Lines A.-1 – A.3, D.-1 – D.5.

   (d) Can this solution lead to starvation? Briefly explain.

   Yes. Once cars start moving in one direction, they will starve out cars waiting to go in the other direction indefinitely (as long as more cars continue to arrive in the first direction).

   (e) Can this solution lead to deadlock? Briefly explain.

   No, not all of the deadlock conditions are met (e.g., no circular wait). In specific terms, for example, there is no way for cars going in both directions to wait at the same time.

5. (15 pts) Consider the following test program for an implementation of Join in Nachos. It begins when the "main" Nachos thread calls ThreadTest(). You do not need to know the details of how Join is implemented. All you need to know is that when a "parent" thread calls Join on a "child" thread, the "parent" does one of two things: (1) if the "child" is still running, the "parent" blocks until the "child" finishes (at which point the "parent" is placed on the ready queue); (2) if the "child" has finished, the "parent" continues to execute without blocking. The semantics of Fork are exactly those of Nachos in Project 1.

```
void ThreadTest() {
    Thread *t;

    t = new Thread("A", 1);
    t->setPriority(10);
    printf(''fee '');
    t->Fork(A, 0);
    printf(''foe '');
    t->Join();
    printf(''fun\n'');
}
```

```
void A(int arg) {
    Thread *t2;

    t2 = new Thread(''B'', 1);
    t2->setPriority(20);
    printf(''foo '');
    t2->Fork(B, 0);
    printf(''far '');
    t2->Join();
    printf(''fum '');
}

void B(int arg) {
    printf(''fie '');
}
```

(a) Assume that the scheduler runs threads in round-robin order with no implicit time-slicing (i.e., non-preemptive scheduling), priorities are ignored, and threads are placed on queues in FIFO order. What will this test program print out?

fee foe foo far fie fum fun

(b) Now assume that the scheduler runs threads according to priority. You should not need to worry about how priority is implemented. It is enough to know that (1) the highest priority thread on the ready queue runs first, and (2) when a thread is added to the ready queue, it will preempt the current thread if the new thread has higher priority. Assume that there is *no* priority donation, and that the priority of the "main" thread is 0. What will the test program print out now?

fee foo fie far fum foe fun