CSE 120 Principles of Computer Operating Systems
Fall Quarter, 2000
Midterm Exam

Instructor: Geoffrey M. Voelker

Name _____
Student ID _____

Attention: This exam has five questions worth a total of 75 points. You have 90 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	/10
3	/10
4	/20
5	/25
Total	/75

1. (10 pts) Which of the following require assistance from hardware to implement correctly? For those that do, circle them and name the hardware support.

   (a) system call – *System call instruction, trap to the OS, change modes, etc.*

   (b) context switch

   (c) process fork – *Requires system call, privileged instructions to manipulate VM state, etc.*

   (d) thread fork

   (e) acquiring a lock – *Privileged instructions to turn off interrupts, atomic instructions (test-and-set)*

   (f) releasing a lock

   (g) preventing deadlock

   (h) preventing starvation

   Releasing a lock (interrupts) was also a reasonable answer. I was generous if you were creative with the other items.

2. (10 pts) Round-robin schedulers (e.g., the Nachos scheduler) maintain a *ready list* or *run queue* of all runnable threads (or processes), with each thread listed at most once in the list. What can happen if a thread is listed twice in the list? Briefly explain how this could cause programs that use synchronization primitives to break on a uniprocessor.

The problem is that a thread on the ready list twice results in a spurious wakeup. Consider a thread executing the Tweedledum procedure in the condition variable solution to problem 5. If a thread is on the ready list twice, then the first time it runs on the CPU it will execute inside of Tweedledum and eventually Wait on the condition variable. But, since it is on the ready list twice, it will run again, without any other thread having called Signal on the condition variable. This could, for example, cause that thread to execute Shout twice in a row, violating the problem constraints.

The most common error on this problem was to think of the thread on the list twice as two different threads, with two different stacks, etc. Although this is not accurate, I still gave partial credit when an answer with this interpretation demonstrated a synchronization problem.

3. (10 pts) In lecture, we discussed using atomic instructions (e.g., test-and-set) to implement spinlocks. With spinlocks, threads spin in a loop until the lock is freed. We motivated the use of blocking locks and semaphores as an improvement over spinlocks because having threads spin can be very inefficient in terms of processor utilization. However, spinlocks are not always less efficient than blocking locks. Briefly describe a scenario where spinlocks would be more efficient than blocking locks.

Spinlocks are more efficient when the critical sections are very short (e.g., one or a few instructions that increment a shared variable), or when the critical section is rarely contested (i.e., Acquire almost always will succeed). Spinlocks are more efficient in these cases than, say, semaphores, because the overhead to acquire the lock is incredibly low (one instruction).

I also accepted the multiprocessor answers from the book.

4. (20 pts) Consider the following test program for an implementation of locks and condition variables in Nachos. It begins when the "main" Nachos thread calls ThreadTest(). Trace the execution of this program until it prints out the message "STOP HERE", and (a) write down the sequence of context switches that occurred up this point, (b) list the queues that the threads are on at this point, and their relative order if more than one thread is on a queue (currentThread, the readyList, and any wait queues associated with synchronization primitives), and (c) the thread states of each of the threads. For example, "A -> B" signifies that thread A context switches to thread B, "readyList: A" signifies that A is on the ready list, and the possible states of a thread are JUST_CREATED, RUNNING, READY, and BLOCKED.

   Assume that the scheduler runs threads in FIFO order with no time-slicing (non-preemptive scheduling), all threads have the same priority, and threads are placed on wait queues in FIFO order.

```
Lock *l;
Condition *cv;

void A(int arg) {
    l->Acquire();
    cv->Signal();
    currentThread->Yield();
    cv->Wait();
    l->Release();
}

void B(int arg) {
    l->Acquire();
    cv->Wait();
    currentThread->Yield();
    cv->Signal();
    l->Release();
}
```

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

    l = new Lock("lock");
    cv = new Condition("cv");

    t = new Thread("A");
    t->Fork(A, 0);
    t = new Thread("B");
    t->Fork(B, 0);
    currentThread->Yield();
    currentThread->Yield();
    printf(``STOP HERE\n'');
}
```

(a) Context switches: main -> *A* -> *B* -> *main* -> *A* -> *main*

(b) Thread queues:

currentThread: *main*

readyList: *B*

Lock:

Condition: *A*

(c) Thread states:

main: *RUNNING*

A: *BLOCKED*

B: *READY*

5. (25 pts) Tweedledum and Tweedledee are separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation. The *Sleep()* routine blocks the calling thread, and the *Wakeup()* routine unblocks a specific thread if that thread is blocked; otherwise, the behavior of *Wakeup()* is unpredictable (*Wakeup()* is like the Nachos *Scheduler::ReadyToRun* method).

```
void
Tweedledum()
{
    while (1) {
        Sleep();
        x = ShoutInsult(X);
        Wakeup(Tweedledee thread);
    }
}
```

```
void
Tweedledee()
{
    while (1) {
        x = ShoutInsult(X);
        Wakeup(Tweedledum thread);
        Sleep();
    }
}
```

(a) The code shown above exhibits a well-known synchronization flaw. Outline a scenario in which this code would fail, and the outcome of that scenario. Regurgitate the buzzword for this synchronization flaw.

One scenario is if Tweedledee runs first, Shouts, and then calls Sleep. Since Tweedledum has yet to run, its Wakeup is lost. Then Tweedledum runs, and immediately falls asleep. In this case, both threads are blocking, and the program has deadlocked (the buzzword).

(b) Show how to fix the problem using semaphores, replacing the *Sleep()* and *Wakeup()* calls with semaphore *Wait* (P/down) and *Signal* (V/up) operations. Note: Disabling interrupts is not an option.

```
Semaphore dum = 0;
Semaphore dee = 1;
```

```
void
Tweedledum()
{
    while (1) {
        dum.Wait();
        x = ShoutInsult(X);
        dee.Signal();
    }
}
```

```
void
Tweedledee()
{
    while (1) {
        dee.Wait();
        x = ShoutInsult(X);
        dum.Signal();
    }
}
```

(c) Implement Tweedledum and Tweedledee using a mutex and condition variable.

This is exactly the pingpong problem in the lecture notes. Both procedures do the same thing.

```
Lock mutex;
Condition cv;

void
Tweedledum()
{
    mutex.Acquire();
    while (1) {
        x = ShoutInsult(X);
        cv.Signal(&mutex);
        cv.Wait(&mutex);
    }
    mutex.Release();
}
```

```
void
Tweedledee()
{
    mutex.Acquire();
    while (1) {
        x = ShoutInsult(X);
        cv.Signal(&mutex);
        cv.Wait(&mutex);
    }
    mutex.Release();
}
```