

CSE 120 Principles of Computer Operating Systems
Fall Quarter, 2001
Midterm Exam *Solutions*

Instructor: Geoffrey M. Voelker

Name _____

Student ID _____

Attention: You have 80 minutes to complete the questions on the exam. As with any exam, 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 a reasonable assumption, state the assumption in your answer. You may write on the backs of pages if needed.

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 lock ordering take (acquiring multiple locks in a consistent order)?

Prevention.

(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 the slides from “Kernel Threads” to “Kernel vs. User Threads” in the Threads lecture.

(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.

Complete the class `Bridge` below using **locks** and **condition variables** to synchronize (do not manipulate interrupts or invoke thread methods, or add methods to locks and condition variables). Use pseudocode in your answer. It does not have to compile, and you can use any syntax or method names you are comfortable with (but it does have to look like code).

Each car is a thread. A car thread calls `arrive` when it arrives at the bridge and `depart` when it leaves the bridge. Threads pass in their direction of travel as a parameter, which can be either `WEST` or `EAST` (two positive pre-defined integer constants). Threads block in `arrive` until it is safe to cross, but they should not block if it possible for them to travel in their direction.

```
class Bridge {
    Lock lock = new Lock();
    Condition cv = new Condition(lock);
    int current = -1;
    int cars = 0;

    Bridge () { }

    void arrive (int dir) {
        lock.acquire();
        while (cars > 0 && current != dir) {
            cv.sleep();
        }
        cars++;
        current = dir;
        lock.release();
    }

    void depart (int dir) {
        lock.acquire();
        cars--;
        if (cars == 0) {
            current = -1;
            cv.wakeAll();
        }
        lock.release();
    }
}
```

5. (15 pts) Consider the following test program for an implementation of `KThread.join` in Nachos. It begins when the main Nachos thread calls `KThread.selfTest`. 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. Assume `join` uses a wait queue of some kind in its implementation.

```
public static void selfTest() {
    KThread t1 = new KThread (new A());
    t1.setName ("A");
    System.out.println ("fee");
    t1.fork ();
    System.out.println ("foe");
    t1.join ();
    System.out.println ("fun");
}

static class A implements Runnable {
    public void run () {
        KThread t2 = new KThread (new B());
        t2.setName ("B");
        System.out.println ("foo");
        t2.fork ();
        System.out.println ("far");
        t2.join ();
        System.out.println ("fum");
    }
}

static class B implements Runnable {
    public void run () {
        System.out.println ("fie");
    }
}
```

Assume that the scheduler runs threads in FIFO order with non-preemptive scheduling (no preemptive time-slicing), and threads are placed on wait queues in FIFO order. Trace the execution of this program until it returns from `selfTest` and (a) write the sequence of context switches that occurred up to this point, (b) write the output of the program, and (c) list the queues that the threads are on, and their relative order if more than one thread is on a queue.

(a) Context switches: $main \rightarrow A \rightarrow B \rightarrow A \rightarrow main$

(b) Output: *fee foe foo far fie fum fun*

(c) Thread queues when `selfTest` returns:

currentThread: *main (main executes selfTest)*

readyQueue: *(nothing, A & B have finished and main is executing)*

A's join wait queue: *(nothing, A has finished)*

B's join wait queue: *(nothing, B has finished)*