CSE451 Midterm
Winter 2008

Short Answer (4 points each)

1) Suppose you were designing a scheduling strategy for the order-processing system of Amazon.com. At a given time, the system has a set of pending orders of a known size. How would you schedule orders to maximize throughput (orders / second)?

shortest job first

2) Suppose the goal were to ensure that large orders are serviced with reasonable latency. Is your algorithm from question 1 appropriate? If not, what algorithm would you use instead?

Not shortest job first (susceptible to deadlock) -- round-robin, FCFS, others...

3) Give one reason why Linux devotes a separate kernel stack to each thread (in addition to its user-mode stack).

To ensure adequate space exists on the stack; to protect the kernel from threads in the same address space; to allow for physical parallelism

4) Two threads are forked at nearly the same time. They each invoke methods on the following class as indicated in the comments. What are the possible output(s) of this program:

00 01 10 11

```
public class Fubar {
    int x = 0;
    int y = 0;

    // thread 1 executes this method
    pubic void threadOneMethod() {
      x++;
      y++;
    }

    // thread 2 executes this method
    public void threadTwoMethod() {
      System.out.print(x);
      System.out.print(y);
    }
}
```

5) The use of multiple locks can lead to deadlock. Is it possible to prevent such deadlocks by taking advantage of the “mutual exclusion” condition (one of the four necessary conditions for deadlock)? Explain your answer.

No. The mutual exclusion condition can only be broken if resources are shareable. Locks are inherently non-sharable.

6) The use of multiple locks can lead to deadlock. Is it possible to prevent such deadlocks by taking advantage of the “circular wait” condition (one of the four necessary conditions for deadlock)? Explain your answer.

Yes -- by imposing a fixed ordering on all lock acquisitions.

7) Explicit locks (10 points)

Java's `synchronized` statement automatically performs lock `acquire` and `release` operations. It is occasionally useful to use *explicit locks,* which require the programmer to manually invoke `acquire` and `release`. Your task is to implement the `ExplicitLock` class provided below. You may use any built-in feature of Java, including implicit locks (`synchronized` statements) and condition variables. Do not use a Semaphore or any other external synchronization primitives.

In all other respects, your lock should behave *exactly* like Java locks. In particular, recall that Java locks are re-entrant: a single thread can acquire the lock an arbitrary number of times.

It is not your responsibility to ensure that callers use the lock correctly. Do not worry about timeouts or interruption.

```
public class ExplicitLock {
    public void acquire() {
      Thread caller = Thread.currentThread();

      // TODO: implement the rest...
    }

    public void release() {
      // TODO: implement me
    }
}

// Example of usage; do not change this
public class ExplicitLockExample {
    private ExplicitLock lock = new ExplicitLock();

    public void foo(boolean b1,boolean b2) {

      lock.acquire();
      if (b1) {
          if (b2) {
            // do some stuff
            lock.release();
            // do some stuff
            return;
          }
          // do some stuff
          lock.release();
      }
      else {
          // do some stuff
          lock.release();
      }
      // do some stuff
    }
}
```

```
// solution
public class ExplicitLock {

    private int depth = 0;
    private Thread owner = null;

    public synchronized void acquire() throws InterruptedException {
      Thread caller = Thread.currentThread();

      if (caller == owner)
          ++depth;
      else {
          while (owner != null) wait();
          owner = caller;
          depth = 1;
        }
    }

    public synchronized void release()  {
      assert (Thread.currentThread() == owner);

      if (--depth == 0) {
          owner = null;
          notify();
      }
    }
}
```

8) Parallel execcounts (8 points)

Describe the implementation of a *parallel* version of the `execcnts` utility from project #1. The new utility is an ordinary user-mode program that returns the execution counts for **two** routines:

```
pexeccnts find . -name '*.c' ; find . -name '*.h'
```

Your program should execute both routines in parallel. It should print separate execution counts for both routines; the order of output does not matter. You may describe an implementation without providing a complete implementation. You can assume the existence of a correct system call implementation from project #1:

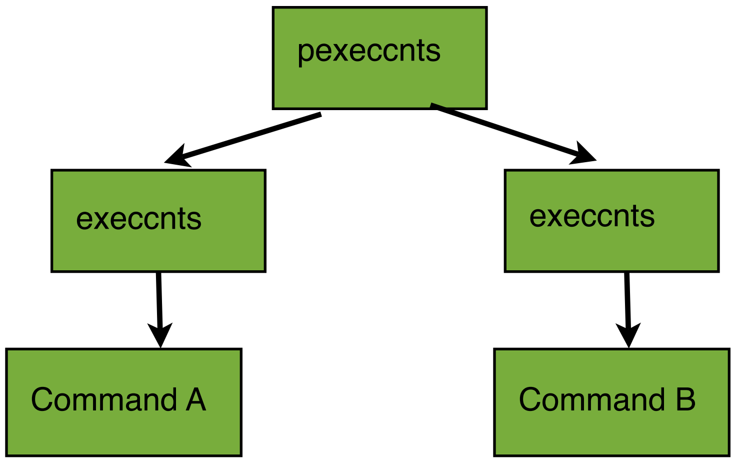
```
int getExecCounts( int pid, int* pArray );
```

Note: it as en error to call `getExecCounts` for a non-existent or dead process.

Hint: first figure out why the "obvious" approach doesn't work. Writing this down will get you partial credit.

The "obvious" approach is two simply fork two child processes from within the pexeccnts utility. However, it would be impossible to differentiate the system callls made by the two child processes.

Instead, the utility can fork two helper processes. The helper processes are identical to the (non-parallel) execcnts utility we defined for project #1. That is, the helper processes fork a single child process. After the child dies, the helper process returns its execcount, after accounting for its own contribution.


pexeccnts
execcnts
execcnts
Command A
Command B