

James Finn

CPE 330

19 December, 2007. 09:00-12:00

*Operating Systems.*

Seat No: \_\_\_\_\_

1	_____
2	_____
3	_____
4	_____
5	_____
6	_____
7	_____
8	_____
9	_____
10	_____
<b>Total</b>	_____

## Midterm Examination

Name \_\_\_\_\_

ID # \_\_\_\_\_

### General Instructions

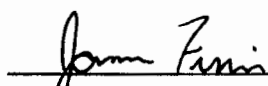
Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit. Your answers must be clearly marked: if I can't find or read an answer, it's wrong.

Each question is marked with the number of points assigned to that problem. The total number of points is 100.

**The examination is open book.** You may make use of books, handouts, notes and a dictionary. You may not use electronic devices of any kind, including telephones, pagers, computers or electronic calculators.

**Make sure your exam is not missing any pages. There are 11 pages total.**

**Instructor**

\_\_\_\_\_

(Dr. James Finn)

**All questions are approved by the department committee.**

\_\_\_\_\_

### Problem 1. System Calls (10 points)

In programming, what is the difference between an ordinary function call and an operating system call? For example, there is a function **sqrt** in the standard C math library; this is an ordinary function call for calculating square roots. In UNIX there is system call **fork** for creating processes. In a program, both are written as function calls:

```
double x = sqrt(2.0); /* function call */
int pid = fork();     /* system call  */
```

But a function call and a system call are not the same. What is the difference between function calls and system calls?

## **Problem 2. Multiprogramming (5 points)**

An operating system supports *multiprogramming* if multiple processes can run concurrently. A system is *multiuser* if more than one person can be logged in at a time, each running one or more processes. Is the problem of process scheduling more difficult on a multiuser multiprogramming system than it is on a single user multiprogramming system? Explain.

**Problem 3. In The Beginning Was The Command Line (10 points)**

Older interactive operating systems used a command line (text) interface exclusively. Graphical user interfaces (bitmapped graphic monitor with a mouse input device) became popular in the 1980s and most computer users now use them exclusively. Even though Microsoft Windows, various implementations of Unix and Linux, and the Macintosh OS X all have graphical interfaces, they all still support the option of using a command line interface as well. What advantages do a text interface offer over a graphical interface?

#### **Problem 4. Microkernels (5 points)**

The current trend in operating systems design is towards microkernels. The idea is that the kernel should be a very small program, and most operating system tasks can be run as ordinary processes. This approach is safer, as not all system tasks need to run with full kernel privileges. This approach is also more modular, a cleaner design, and easier to maintain and extend.

Although microkernels are a more modern approach to systems design, the Linux operating system uses a *monolithic* kernel in which the kernel is a very large program. Why has Linux chosen this approach?

### **Problem 5. Processes and Threads (10 points)**

The concept of a *process* is very old. Every operating system supports the ability to create and run processes. More recently, operating systems have begun to support the ability of a single process to have multiple *threads*. What benefits can we gain from multiple threads that we did not already have with multiple processes?

### **Problem 6. Interprocess Communication (10 points)**

In section 3.4 of Silberschatz, the book introduces *shared memory* and *message passing* as two methods that allow processes to communicate. The original Unix implementation supported only one method of interprocess communication: the pipe. One process writes bytes into a pipe, and the other process can read the bytes in the same order as they were written.

Compare pipes to shared memory and message passing. How do they compare with respect to:

- Efficiency
- Ease of use and problems with synchronization

### Problem 7. Java Threads (15 points)

Write a Java class `Message` that implements the `Runnable` interface. When run as a thread, the thread prints a short message identifying the name of the thread (which could be given as a constructor argument). The class should include a **main** method that creates and runs two thread instances. For example, a sample run might produce this output:

```
Hello, I am thread A
Hello, I am thread B
```

**Answer to Problem 7:**

```
public class Message implements Runnable
{
```



### **Problem 8. Scheduling Goals (10 points)**

A small business runs a mix of interactive and batch jobs on its computer. The batch jobs all have fixed time limits; the system knows how long each batch job will take to run at the time it is submitted. The operating system's process scheduler needs to meet these goals:

1. Throughput: finish as many jobs in as short a time as possible.
2. Good response time: interactive users should see quick response.
3. Fairness: every process in the system should get a fair share of time.

Are any of these goals contradictory (meaning do they conflict with each other)? Which ones and why?

**Problem 9. Scheduling Algorithms (15 points)**

This table shows the arrival time and CPU burst time (in milliseconds) for four processes:

Process	Arrival Time	Burst Time
$P_1$	0	5
$P_2$	1	8
$P_3$	7	5
$P_4$	8	3

Draw a Gantt chart and calculate the average process waiting time for each of these scheduling algorithms:

**Part (a)** First-Come, First-Served

**Part (b)** Shortest-Job-First (non-preemptive)

**Part (c)** Shortest-Job-First (preemptive)

### Problem 10. Burst Estimation (10 points)

Usually, a process scheduler does not know the next burst time of a ready process and it must make a quick estimate. One standard method is to use an *exponential average*, where:

$t_n$  = actual length of  $n$ th CPU burst

$\tau_n$  = estimate of  $n$ th CPU burst

We make estimates by calculating a weighted average of the previous burst and the previous estimate:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

The weight  $\alpha$  is between 0 and 1, inclusive. Usually we use  $\alpha = 1/2$ .

A new process arrives that uses increasingly longer CPU bursts. It has this behavior:

$$t_n = 2(n + 1)$$

If we use  $\alpha = 1/2$  and an initial estimate  $\tau_0 = 6$ , calculate the values of  $\tau_n$  for  $n = 1$  to 5.

**Answer to Problem 10:**

$n$	0	1	2	3	4	5
$\tau_n$	6					

2008, Sem 2 Spring 2550

James Finn

CPE 330

5 March, 2008. 09:00-12:00

Operating System

Seat No: \_\_\_\_\_

1 \_\_\_\_\_

2 \_\_\_\_\_

3 \_\_\_\_\_

4 \_\_\_\_\_

5 \_\_\_\_\_

6 \_\_\_\_\_

7 \_\_\_\_\_

8 \_\_\_\_\_

**Total** \_\_\_\_\_

## Final Examination

Name \_\_\_\_\_

ID # \_\_\_\_\_

### General Instructions

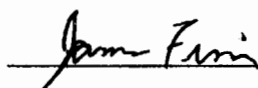
Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit. Your answers must be clearly marked: if I can't find or read an answer, it's wrong.

Each question is marked with the number of points assigned to that problem. The total number of points is 100.

**The examination is open book.** You may make use of books, handouts, notes and a dictionary. You may not use electronic devices of any kind, including telephones, pagers, computers or electronic calculators.

**Make sure your exam is not missing any pages. There are 11 pages total.**

Instructor



(Dr. James Finn)

**All questions are approved by the department committee.**

\_\_\_\_\_

### Problem 1. Process Synchronization (20 points)

A programmer has to write a program with two processes, Process 1 and Process 2. Each process runs in a loop that repeatedly calls an event: Process 1 calls **event1**, and Process 2 calls **event2**. The events have to alternate, with **event1** going first, so that the sequence of events looks like this: **event1, event2, event1, event2, ...**

The programmer comes up with three solutions to the problem. The first two are software-only, and the third uses semaphores. Here are sketches his solutions:

#### Solution #1:

Shared variables:  <code>int turn = 1;</code>	
Process 1:  <code>for(;;)   if (turn == 1)   {     event1();     turn = 2;   }</code>	Process 2:  <code>for(;;)   if (turn == 2)   {     event2();     turn = 1;   }</code>

#### Solution #2:

Shared variables:  <code>int turn = 1;</code>	
Process 1:  <code>for(;;)   if (turn == 1)   {     turn = 2;     event1();   }</code>	Process 2:  <code>for(;;)   if (turn == 2)   {     turn = 1;     event2();   }</code>

*Problem continues on next page.*

**Solution #3:**

Shared variables:  <code>Semaphore proc1 = new Semaphore(1);</code> <code>Semaphore proc2 = new Semaphore(0);</code>	
Process 1:  <code>for (;;) {</code> <code>proc1.acquire();</code> <code>event1();</code> <code>proc2.release();</code> <code>}</code>	Process 2:  <code>for (;;) {</code> <code>proc2.acquire();</code> <code>event2();</code> <code>proc1.release();</code> <code>}</code>

These code sketches are not complete. In reality, we would have to use a `MutableInteger` object for solutions #1 and #2, because you can't share a mutable `int` in Java. Also, we didn't write any exception handling code in Solution #3. Don't worry about this.

**Part (a)**

One of the three solutions has a bug in the algorithm and will not work correctly. Which solution has the bug and what exactly is the problem?

**Answer to Part (a):**

*Problem continues on next page*

**Part (b)**

Of the two solutions that are correct, one is much better than the other. Which is the better solution and why?

**Answer to Part (b)**

## Problem 2. Java Locks (10 points)

In Java, you can't use an `int` as a shared variable among threads. In programs in class and on the homework, we worked around this problem by using a simple `MutableInteger` class:

```
public class MutableInteger
{
    private int value;

    public MutableInteger(int initial)
    {
        value = initial;
    }

    public int getValue()
    {
        return value;
    }

    public void setValue(int val)
    {
        value = val;
    }
}
```

When a mutable data type is shared among multiple threads, its methods usually should be declared **synchronized** to enforce mutual exclusion among threads. Suppose that we use the `MutableInteger` class instead of the `int` **turn** in the first two solutions of Problem 1. Can we use it as written above, or do we have to declare the `getValue` and `setValue` methods to be **synchronized**? Explain.

### Answer to Problem 2



### Problem 3. Locks and Monitors (10 points)

In Java, a thread can voluntarily suspend itself when it needs to wait for some event. It does this by calling the **wait** method on the appropriate object. For example, a Consumer thread might have to wait if the queue of data to consume is empty. When things change, another thread can call **notify** to wake up a waiting thread, or call **notifyAll** to wake all waiting threads.

#### Part (a)

Why does Java have both **notify** and **notifyAll** methods? Only one thread at a time can hold the lock on an object, so why isn't **notify** sufficient?

#### Part (b)

In the Monitor type described in section 6.7.1 of the book (pages 233-5), programmers work with Condition variables that are similar to Java wait queues. if **c** is a Condition, a thread can suspend itself to wait for some event by calling **c.wait()** and another thread can call **c.signal()** to wake one waiting thread. In the book's model of Monitors, there is no method **c.signalAll()** method to wake all waiting threads. Why don't Monitor Conditions need a **signalAll()** method?

#### **Problem 4. Deadlock Prevention (10 points)**

Several conditions are necessary for deadlock to occur. You need mutual exclusion: a thread that holds the lock for a resource has exclusive access to it and other threads wishing to acquire the lock must wait for it to be released. You also need “hold and wait”: a thread owns the lock on one resource and is waiting to acquire the lock on a different resource.

We can prevent deadlock by eliminating the possibility of “hold and wait”. We can require that any process that will need to hold more than one lock concurrently acquires them all at once.

This solution will work but it is not desirable. What is bad about preventing “hold and wait”?

## **Problem 5. Deadlock and Starvation (10 points)**

### **Part (a)**

A bridge is only wide enough for one car. Two cars enter the bridge from opposite ends and meet in the middle. Now neither car can proceed unless the other car backs up. Is this an example of *deadlock* or of *starvation*? Explain.

### **Part (b)**

In Bangkok, the circular road around Victory Monument is an example of a *traffic circle*. One day, a car approaches a traffic circle. It needs to enter the circle but there are so many cars in the circle already that there is no room for the new car, and it has to wait a very very very very long time. Is this an example of deadlock or of starvation? Explain.

### **Problem 6. Page Replacement Algorithms (15 points)**

A process is given three page frames when it runs. It references the following string of page numbers in this order: 1, 2, 3, 2, 1, 4, 3, 2.

When the process runs, pages 1, 2 and 3 are loaded into the three free frames, but when page 4 is first referenced, a page must be replaced. Which page will be replaced by page 4 for each of the following replacement strategies:

- a) Optimal
- b) Least-recently-used (LRU)
- c) FIFO

**Answer to Problem 6:**

a)

b)

c)

## **Problem 7. Paging and Hardware Support (10 points)**

### **Part (a)**

We discussed in lecture how some operating system services require hardware support in order to be practical. The use of demand paging for virtual memory requires hardware support. What kind of support is needed? Why would paging be impractical without this hardware support?

### **Answer to Part (a)**

### **Part (b)**

A computer has hardware support for demand paging. The operating system wants to use LRU (or a good approximation algorithm such as a reference bit) for its page-replacement strategy. Is additional hardware support necessary, and, if so, what support?

### **Answer to Part (b)**

## **Problem 8. Unix File System (15 points)**

### **Part (a)**

The Unix file system supports two kind of links: *hard* links and *symbolic* links (also called *soft* links). Explain the difference between them.

### **Part (b)**

We said that Unix files do not have names! How is this possible?