University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 2007 John Kubiatowicz

Midterm I
SOLUTIONS
October 10th, 2007
CS162: Operating Systems and Systems Programming

Your Name:	
SID Number:	
Discussion Section:	

General Information:
This is a **closed book** exam. You are allowed 1 page of **hand-written** notes (both sides). You have 3 hours to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	**Possible**	**Score**
1	15	
2	17	
3	25	
4	25	
5	18	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494 4

Problem 1: Short Answer [15pts]

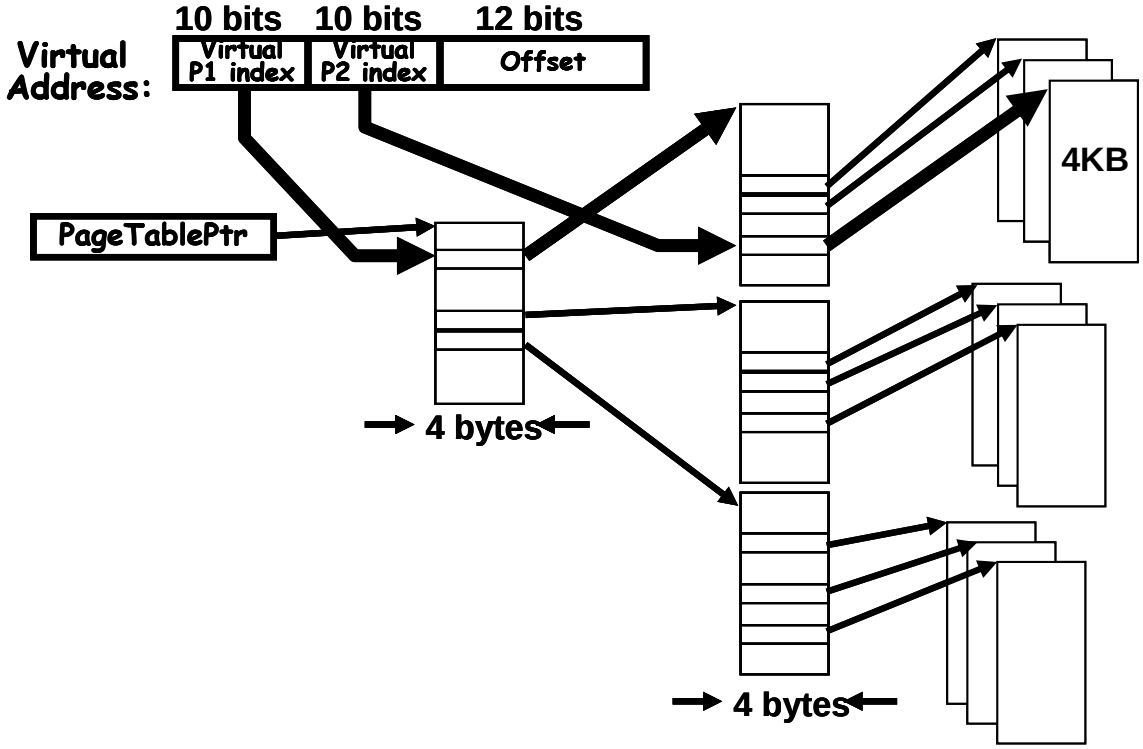
Problem 1a[2pts]: Name two ways in which processes on the same processor can communicate with one another. If any of the techniques you name require hardware support, explain.

1) *Shared memory. Typically requires hardware support for cache coherence and translation mechanisms (page tables).*
2) *Message passing. Done through kernel, typically doesn't require much hardware support.*
3) *File System. Hardware support might include the disk, although this is necessary for other uses as well.*

Problem 1b[2pts]: What is an interrupt? What is the function of an interrupt controller?

An interrupt is an electrical signal that causes the processor to stop the current execution and begin to execute interrupt handler code. The interrupt controller is a component between I/O devices and the processor that filters incoming interrupt signals on their way to the processor; among other things, it allows the operating system to mask out certain interrupts to enable the establishment of interrupt priority.

Problem 1c[3pts]: Suppose that we have a two-level page translation scheme with 4K-byte pages and 4-byte page table entries (includes a valid bit, a couple permission bits, and a pointer to another page/table entry). What is the format of a 32-bit virtual address? Sketch out the format of a complete page table.



Problem 1d[2pts]: Which company was the first to develop a workstation with a mouse and overlapping windows?

Xerox (The Xerox PARC research lab)

Problem 1e[2pt]: What is a thread-join operation?

A thread-join operation allows a thread to wait for another thread to finish. It puts the calling thread to sleep and wakes when the target thread exits.

Problem 1f[2pts]: True or False: When designing a multithreaded application, you must use synchronization primitives to make sure that the threads do not overwrite each other's registers. Explain.

FALSE. The thread scheduler is responsible for making sure that each thread has its own set of register values (stored in the TLB). A TLB is never shared between threads.

Problem 1g[2pts]: True or False: A system that provides segmentation without paging puts holes in the physical address space, forcing the operating system to waste physical memory. Explain.

FALSE. Segmentation potentially puts holes in the VIRTUAL space, but doesn't require holds in the physical address space, since the segmentation mapping is free to use any contiguous blocks of memory.

Problem 2: Multithreading [17 pts]

Consider the following two threads, to be run concurrently in a shared memory (all variables are shared between the two threads):

Thread A	Thread B

`for (i=0; i<5; i++) {`
`  x = x + 1;`
`}`
	`for (j=0; j<5; j++) {`
	`  x = x + 2;`
	`}`

Assume a single-processor system, that load and store are atomic, that x is initialized to 0 *before either thread starts*, and that x must be loaded into a register before being incremented (and stored back to memory afterwards). The following questions consider the final value of x after both threads have completed.

Problem 2a[2 pts]: Give a *concise* proof why $x \leq 15$ when both threads have completed.

Each x=x+1 statement can either do nothing (if erased by Thread B) or increase x by 1. Each x=x+2 statement can either do nothing (if erased by Thread A) or increase x by 2. Since there are 5 of each type, and since x starts at 0, x is ≥ 0 and $x \leq (5\times1)+(5\times2)=15$

Problem 2b[4 pts]: Give a *concise* proof why $x \neq 1$ when both threads have completed.

Every store into x from either Thread A or B is ≥ 0, and once x becomes ≥ 0, it stays ≥ 0. *The only way for x=1 is for the last x=x+1 statement in Thread A to load a zero and store a one. However, there are at least four stores from Thread A previous to the load for the last statement, meaning that it couldn't have loaded a zero.*

Problem 2c[3pts]: Suppose we replace '**x = x+2**' in Thread B with an atomic double increment operation **atomicIncr2(x)** that cannot be preempted while being executed. What are all the possible final values of `x`? Explain.

Final values are 5, 7, 9, 11, 13, or 15. The x=x+2 statements can be "erased" by being between the load and store of an x=x+1 statement. However, since the x=x+2 statements are atomic, the x=x+1 statements can never be "erased" because the load and store phases of x=x+2 cannot be separated. Thus, our final value is at least 5 (from Thread A) with from 0 to 5 successful updates of x=x+2.

Problem 2d[3pts]: What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? Be explicit.

Need to save the processor registers, stack pointer, program counter into the TCB of the thread that is no longer running. Need to reload the same things from the TCB of the new thread.

When the threads are from different processes, need to not only save and restore what was given above, but you also need to load the pointer for the top-level page-table of the new address space. You don't need to save the old pointer, since this will not change and is already stored in the PCB.

Problem 2e[2pts]: Under what circumstances can a multithreaded program complete more quickly than a non-multithreaded program? Keep in mind that multithreading has context-switch overhead associated with it.

When there is a lot of blocking that may occur (such as for I/O) and parts of the program can still make progress while other parts are blocked.

Problem 2f[3pts]: What is simultaneous multithreading (or "Hyperthreading")? Name two ways in which this differs from the type of multithreading done by the operating system.

Simultaneous multithreading utilizes spare issue slots in a superscalar processor to support multithreading. It allows multiple threads to be loaded into the processor simultaneously.

Simultaneous multithreading differs from the type of multithreading done by the operating system in several ways: (1) the multiplexing is done in hardware rather than software, (2) the multiplexing can interleave threads on an instruction by instruction basis, whereas the operating system interleaves with a 10ms (or more) quantum (3) the overhead for switching between threads is a single cycle in simultaneous multithreading rather than many cycles in the operating system (4) simultaneous multithreading can only handle a small number of threads at a time, whereas the operating system can handle many threads.

Problem 3: Hoare Scheduling [25pts]

Assume that you are programming a multiprocessor system using threads. In class, we talked about two different synchronization primitives: Semaphores and Monitors. In this problem, we are going to implement Monitors with Hoare scheduling by using Semaphores

The interface for a Semaphore is as follows:

```
public class Semaphore {
   public Semaphore(int initialValue) {
      /* Create and return a semaphore with initial value: initialValue */
      …
   }
   public P() {
      /* Call P() on the semaphore */
      …
   }
   public V() {
      /* Call V() on the semaphore */
   }
}
```

As we mentioned in class, a Monitor consists of a Lock and one or more Condition Variables. The interfaces for these two types of objects are as follows:

```
public class Lock {
   public Lock() {
      /* Create new Lock */
      …
   }

   public void Acquire() {
      /* Acquire Lock */
      …
   }
   public void Release() {
      /* Release Lock */
      …
   }
}
```

```
public class CondVar {
   public CondVar(Lock lock) {
      /* Creates a condition variable
         associated with Lock lock. */
      …
   }
   public void Wait() {
      /* Block on condition variable *'/
      …
   }
   public void Signal() {
      /* Wake one thread (if it exists) */
      …
   }
}
```

Problem 3a[3pts]: What is the difference between Mesa and Hoare scheduling for monitors? Focus your answer on what happens during the execution of Signal(). *Hint: Mesa scheduling requires all conditional wait statements to be wrapped in "while" loops (see 3b).*

> *With Mesa scheduling, the signaling thread keeps the lock and CPU and puts the signaled thread on the ready queue for later execution. With Hoare scheduling, the signaling thread gives the lock and CPU to the signaled thread and sleeps. Later, when the signaled thread executes wait() or releases the lock, it returns the lock to the signaled thread and wakes it.*
>
> *Note that we didn't give full credit for answers that said that the signaled thread runs immediately without also saying either that the signaling thread sleeps or that the signaling thread gives the CPU to the signaled thread.*

Problem 3b[2pts]: When programming with a Mesa-scheduled monitor, the programmer must enclose all conditional wait operations with a while loop like this:

```
while (condition unsatisfied)
   C.Wait();
```

What might happen if they did not do this? How might this happen?

If they only checked the condition once, then it is possible that the thread would exit wait() with the condition unsatisfied. This might happen because a signaled thread is only placed on the ready queue, not run immediately; hence another thread could change the condition between the time that the signaled thread was placed on the ready queue and the time that it exited wait().

Note that we did not give full credit if you didn't mention an intervening thread changing the condition. We also did not give full credit to people who claimed that multiple threads would enter the critical section at the same time – this is not true.

Problem 3c[2pts]: Explain why a Hoare-scheduled monitor allows the programmer to replace the "while" statement in (3b) with an "if".

Since the signaled thread is run immediately after being signaled, there is no chance for another thread to change the condition before the first thread can deal with it.

Problem 3d[3pts]: Assume that we implement a Hoare-scheduled monitor with semaphores. Clearly, we will need one semaphore to implement mutual exclusion. Further, e*ach* condition variable will utilize a semaphore to hold its queue of sleeping threads. Give at least 3 reasons why the following implementation of wait() and signal() is incorrect:

```
wait() { CVSemi.p(); }
signal() {CVSemi.v(); }
```

There are many answers here. A few are as follows:

1) *calling of signal() when no waiting threads will cause subsequent call of wait() to exit without waiting.*
2) *wait() neglects to release lock, hence causing deadlock.*
3) *Signal doesn't put signaling thread to sleep*
4) *Wait does not wake sleeping signaler, if there is one*

We did not accept claims that there is no lock in the above implementation; the lock is implemented outside wait and signal. We also did not accept comments that the lock was not properly acquired and released in wait() and signal(); the lock is acquired and released in code that surrounds the calling of wait and signal – not in the implementation of wait and signal.

Problem 3e[3pts]: Explain why we need an additional semaphore to implement `signal()` under Hoare scheduling as you describe in problem (3a). *Hint: when a waiting thread (T_1) is signaled by another thread (T_2), something special must happen immediately and something else must happen when this signaled thread either releases the monitor lock or executes* `wait()`*…*

> *We need an additional semaphore on which the signaler (T_2) can sleep and which can be used to cause the signaler to wake up and continue later. People who didn't explicitly mention that the signaler must sleep on this semaphore didn't get full credit.*

Problem 3f[6pts]: Given your answer to (3e), show how to implement the Lock class for a Hoare-scheduled monitor using Semaphores. Let us call the additional semaphore from (3e) the *SDefer* semaphore. *If if helps you think about the problem, you can assume that the SDefer.v() operation wakes up sleepers from SDefer.p() operations in Last-in-First-Out (LIFO) order, although this is not strictly necessary.* Note that this semaphore (and possibly one other variable) may be referenced from within the condition variable implementation you will write in (3g). However, these are not part of the public interface. In the following, make sure to implement the HoareLock(), Acquire(), and Release() methods. HoareLock() is the initialization method and should not be empty! *Hint: the Semaphore interface does not allow querying of the size of its waiting queue; you may need to track this yourself.* None of the methods should require more than five lines.

```
public class HoareLock {
    Semaphore SDefer = null;
    Semaphore SLock = null;
    int deferredCount = 0;              /* Count number of threads waiting on SDefer */

    public HoareLock() {
        SDefer = new Semaphore(0);      /* Used to put threads to sleep: all .p()'s sleep */
        SLock = new Semaphore(1);       /* Used for mutual exclusion: only on .p() gets to run */
    }
    public void Acquire() {
        SLock.p();                      /* Acquire lock */
    }
    public void Release() {
        if (deferredCount > 0) {
            deferredCount--;            /* Decrement deferred count*/
            SDefer.v();                 /* Signaler waiting, wake/pass lock*/
        } else
            SLock.();                   /* No signaler waiting, unlock lock */
    }
}
```

Notes: People made the basic lock Acquire() and Release() very complex. It is not. Also, many people tried to count number of waiters on the lock by executing things like "variable++" before SLock.p(). This is bad coding and suffers from the same problem mentioned in Prob 2. Note that we only release the actual lock in Release() if we are not giving it back to a deferred thread. Also, a number of people didn't get the initial value correct for the "new Semaphore()" statements: these two semaphores are used for fundamentally different things and are thus initialized differently! Note that we need to count # threads sleeping on SDefer to decide whether to wake one or not during Release(). Also, notice that the "deferredCount--" statement could be in signal() [next page] instead; if here, however, it must be before SDefer.v() to avoid race condition (imagine that deferred thread runs immediately after SDefer.v() and runs until it executes "Release()").

Problem 3g[6pts]: Show how to implement the Condition Variable class for a Hoare-schedule monitor using Semaphores (and your HoareLock class from 3f). Be very careful to consider the semantics of signal() as discussed in your answers to (3a) and (3c). *Hint: consider what happens when (1) you signal without a waiting thread and (2) signal with a waiting thread. You may need to refer to methods and/or variables within the lock object. Also, remember that the Semaphore interface does not allow querying of the size of its waiting queue; you may need to track this yourself.* None of the methods should require more than five lines.

```
public class HoareCondVar {
    Semaphore SCondVar = null;
    HoareLock lock;                    /* Must save lock from initializer for use later */
    int waitCount = 0;                 /* Count of threads waiting on CV */

    public HoareCondVar(HoareLock lock) {
        SCondVar = new Semaphore(0);   /* Used to put threads to sleep: all .p()'s sleep */
        this.lock = lock;
    }
    public void Wait() {
        waitCount++;                   /* Keep track of number of waiters*/
        lock.Release();                /* Release lock or wake up signaler */
        SCondVar.p();                  /* Sleep on conditional variable */
    }
    public void Signal() {
        if (waitCount > 0) {           /* Waiter exists, do something */
            waitCount--;               /* avoid race cond by decrementing count */
            lock.deferredCount++;      /* indicate that signaler deferred. */
            SCondVar.v();              /* Wake up waiter/transfer lock */
            Lock.SDefer.p();           /* Defer signaler until woken */
        }
    }
}
```

Notes: Many people missed the fact that you need to preserve the incoming value of "lock" from the initializer if you want to use it later. Hence, the "this.lock=lock" statement. Make sure to look at the initial value in the "new Semaphore()" statement for SCondVar; this value is used because we are putting threads to sleep on that semaphore. Further, note that you must track the number of waiting threads on SCondVar.v() to know whether or not signal() needs to wake one up (same as the SDefer semaphore from the HoareLock implementation). Also, a number of people didn't understand the fact that we transfer the lock from signaler to signaled and back. See definition in (3a). Doing this is simple: we don't release the lock in signal(), rather just wake up the waiter and put the signaler to sleep. Then the waiter wakes up with the lock acquired. Further, when the waiter executes either Release() or wait(), we simply leave the lock acquired for use by the signaler. Finally, notice that both the "waitCount--" and lock.deferredCount++" ops must occur before SCondVar.v() in signal() to avoid race condition (imagine that the SCondVar.v() causes waiter to start executing immediately and that the waiter executes either Release() or Signal().); alternatively, "waitCount--" could be the last statement of wait() without problem.

Problem 4: Deadlock [25pts]

Problem 4a[3pts]: What is the difference between deadlock, livelock, and starvation?

Deadlock and Livelock are types of Starvation. Starvation is a lack of forward progress. Deadlock is a lack of forward progress that will never resolve because it involves a cyclic dependency. Livelock is a lack of forward progress that could go away – typically it involves higher priority jobs continually taking resources away from lower-priority jobs.

Problem 4b[2pts]: Suppose a thread is running in a critical section of code, meaning that it has acquired all the locks through proper arbitration. Can it get context switched? Why or why not?

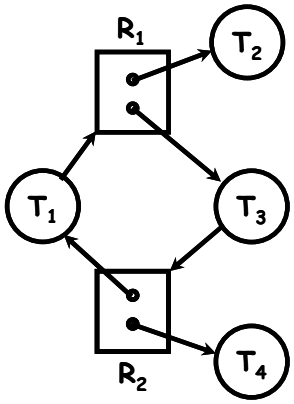
Yes. Grabbing a lock is different from disabling interrupts (and thus preventing context switching). The important thing here is that the acquisition of the lock prevents other threads from running in the same critical section at the same time; it does not generically prevent other threads from running.

Problem 4c[3pts]: What is priority donation and why is it important? Under what circumstances would an operating system need to implement priority donation?

Priority donation is the process of donating priority from a high priority job to a low priority job. When a high priority job sleeps waiting for a lock held by a low priority job, the high-priority job may be delayed behind a medium priority job, thus violating the priority scheduling. The only way to fix this is to temporarily raise priority of the low priority job to that of the waiting job; this is priority donation.

Problem 4d[3pts]: Does a cyclic dependency always lead to deadlock? Explain

No. The cycle dependency may only be temporary (eventually resolve itself). Here is an example given in class that has a cyclic dependency by is not deadlocked:



Problem 4e[8pts]: Consider a large table with multi-armed lawyers. In the center is a pile of chopsticks. In order to eat, a lawyer must have one chopstick in each hand. The lawyers are so busy talking that *they can only grab one chopstick at a time*. Design a BankerTable object using monitors that tracks resources via the Bankers algorithm. It should have public methods GrabOne() that allows a lawyer to grab one chopstick and ReleaseAll() that allows a lawyer to release all chopsticks that he. The GrabOne() method should put a lawyer to sleep if they cannot be granted a chopstick without potentially deadlocking the system. The ReleaseAll() method should wake up lawyers that can proceed. Assume a BankerCheck() method that takes a Lawer number, checks resources, and returns true if a given lawyer can be granted one new chopstick (you will implement this in 4g). *Assume Mesa scheduling and that your condition variable has a broadcast() method.* No method should have more than six (6) lines!

```
public class BankerTable {
   Lock lock = new Lock();
   Condition CV = new Condition(lock);

   int[] AllocTable;                          /* Keep track of allocations */
   int NumArms, NumChopsticks;                /* Keep other info */

   public BankerTable(int NumLawyers, int NumArms, int NumChopsticks) {

      AllocTable = new int[NumLawyers];       /* Array of allocations (init 0)*/
      this.NumArms = NumArms;                 /* Remember # arms/lawyer */
      this.NumChopsticks = NumChopsticks;     /* Remember # chopstocks/table */

   }
   public void GrabOne(int Lawyer) {

      lock.Acquire();

      /* Use banker algorithm to check whether allocation is ok */
      while (!BankerCheck(Lawyer))
         CV.wait();

      /* Move one chopstick from table to lawyer */
      NumChopsticks--;
      AllocTable[Lawyer]++;

      Lock.Release();

   }
   public void ReleaseAll(int Lawyer) {

      lock.Acquire();
      NumChopsticks += AllocTable[Lawyer]; /* Put chopsticks back on table*/
      AllocTable[Lawyer] = 0;                 /* Lawyer has none left */
      lock.Release();
   }
```

Problem 4f[3pts]: In its general form, the Banker's algorithm makes multiple passes through the set of resource takers (think of the deadlock detection algorithm on which it is based). Explain why this particular application allows the BankerCheck method to implement the Banker's algorithm by taking a single pass though the Lawyers' allocations.

The Banker's algorithm only needs a single pass because every resource taker has identical properties: every Lawyer has the same maximum allocation. As a result, if we can find a single Lawyer that can finish, given the remaining resources, we know that all Lawyers can finish. Reason: once that Lawyer finishes and returns their resources we know that there will be at least NumArms chopsticks on the table – hence everyone else can potentially finish. Thus, we don't have to go through the exercise of returning resources and reexamining the remaining Lawyers (as in the general specification of the Banker's algorithm).

Problem 4g[3pts]: Implement the BankerCheck method of the BankerTable Object. This method should implement the Banker's algorithm: return true if the given Lawyer can be granted one additional chopstick without taking the system out of a SAFE state. Do not blindly implement the Banker's algorithm: this method only needs to have the same external behavior as the Banker's algorithm for this application. This method can be written with as few as 7 lines. We will give full credit for a solution that takes a single pass through the lawyers, partial credit for a working solution, and no credit for a solution with more than 10 lines. Note that this method is part of the BankerTable Object and thus has access to local variables of that object…

```
    boolean BankerCheck(int Lawyer) {
    /*
     * This method implements the bankers algorithm for the
     * multi-armed laywer problem. It should return true if the
     * given lawyer can be granted one chopstick.
     */

        int i;
        AllocTable[Lawyer]++;           /* Hypothetically give chopstick to lawyer */
        for (i = 0; i < AllocTable.length; i++)
            /* Is current number on table (NumChopsticks-1) enough to let Lawyer i eat? */
            if ((AllocTable[i] + (NumChopsticks - 1)) >= NumArms)
                break;                  /* Yes! Break early */
        AllocTable[Lawyer]--;           /* Take away hypothetical */
        Return (i < AllocTable.length); /* Returns true if we broke at any time */

    }
}
```

Notes: We don't have to check for the case in which NumChopsticks == 0, since this would mean that (NumChopsticks-1) == -1, and the "if" statement would be self correcting, assuming no lawyer would have > NumArms: the if statement would always fail – exactly what we would want to do when NumChopsticks == 0.

[This page intentionally left blank]

Problem 5: Scheduling [18pts]

Problem 5a[2pts]: Can Lottery scheduling be used to provide strict priority scheduling? Why or why not?

No. By definition, every thread in the lottery scheduler has at least one token and will thus get some of the CPU under all circumstances. In a strict priority scheduler, the higher priority threads get CPU cycles exclusively at the expense of the low priority threads.

Problem 5b[2pts]: Can any of the three scheduling schemes (FCFS, SRTF, or RR) result in starvation? If so, how might you fix this?

Yes. The SRTF algorithm can result starvation of long tasks if short tasks keep arriving. The FCFS algorithm will result in starvation only if some thread runs forever. Finally, RR does not result in starvation, since all threads get some of the CPU. To fix SRTF, we might add some priority mechanism to make sure that threads that have waited for a long time without running get at least some of the CPU now and then (priority increments when threads don't get to run). The multi-level scheduler is a good approximation to SRTF that can be designed to avoid starvation. (the result wouldn't be quite SRTF any more).

To fix FCFS, we would need to find some way of preempting threads that have run forever, possibly by giving some CPU to other threads. Effectively, this would add elements of RR or multi-level scheduling into FCFS (it wouldn't be quite FCFS any more).

Problem 5c[2pts]: Suppose your notion of fairness was to give every thread in a multithreaded system an equal portion of the CPU. Could you do this with strictly user-level threads (every process has only one kernel thread but multiple user-level threads)? Explain.

We accepted two answers here (with justification!): (1) yes. In this case, since the total number of user-level threads are unknown to the kernel, we would need to introduce some sort of system call that allowed a process to tell the kernel how many user-level threads it was running. This would allow the kernel to adjust the fraction of CPU given to each process so that all user-level threads got an equal amount of the CPU. (2) no. Since the kernel doesn't know how many user-level threads each process has, it can't adjust priority accordingly. Note that, while answer (1) says how to fix this, it is subject to malicious manipulation by a user-level process, and hence may not be considered feasible from a security standpoint.

Problem 5d[2pts]: Explain how to fool the multi-level feedback scheduler's heuristics into giving a long-running task more CPU cycles.

By periodically doing I/O (such as a bunch of print statements), a given long-running process can keep its burst-length short and hence its priority high in the multi-level feedback scheduler.

Problem 5e[5pts]:
Here is a table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process 1	0	5
Process 2	4	5
Process 3	2	1
Process 4	7	2
Process 5	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1. *Assume that context switch overhead is 0 and that new RR processes are added to the **head** of the queue and new FCFS processes are added to the **tail** of the queue.*

Time Slot	FCFS	SRTF	RR
0	*1*	*1*	*1*
1	*1*	*1*	*1*
2	*1*	*3*	*3*
3	*1*	*1*	*1*
4	*1*	*1*	*2*
5	*3*	*1*	*1*
6	*2*	*2*	*2*
7	*2*	*4*	*4*
8	*2*	*4*	*5*
9	*2*	*5*	*1*
10	*2*	*5*	*2*
11	*4*	*5*	*4*
12	*4*	*2*	*5*
13	*5*	*2*	*2*
14	*5*	*2*	*5*
15	*5*	*2*	*2*

Problem 5f[3pts]:
For each process in each schedule above, indicate the queue wait time. Note that wait time is the total time spend waiting in queue (all the time in which the task is not running).

Scheduler	Process 1	Process 2	Process 3	Process 4	Process 5
FCFS wait	*0*	*2*	*3*	*4*	*5*
SRTF wait	*1*	*7*	*0*	*0*	*1*
RR wait	*5*	*7*	*0*	*3*	*4*

Problem 5g[2pts]:
Assume that we could have an oracle perform the best possible scheduling to reduce average wait time. What would be the optimal average wait time, and which of the above three schedulers would come closest to optimal? Explain.

Since SRTF is optimal, taking the future into account (i.e. as an oracle), just need to use the SRTF average. Average wait time in this case would be: (1+7+0+0+1)/5 = 1.8

[This page intentionally left blank]