CS 162, Spring 1996
Midterm
Professor

Problem #1 (5 points)

Of the following items, circle those that are stored in the thread control block.
(a) CPU registers
(b) page table pointer
(c) stack pointer
(d) ready list
(e) segment table
(f) thread priority
(g) program counter

Problem #2 (10 points)

Write down the sequence of context switches that would occur in Nachos if the "main" thread were to call the following code. Assume that the CPU scheduler runs threads in FIFO order, with no time-slicing and all threads having the same priority. The WillJoin flag is used to signify that the thread will be joined to by its parent. For example, "child2 => child1" would signify that child2 switches to child1.

```
void
Thread::SelfTest2() {
        Thread *t1 = new Thread("child 1", WillJoin);
        Thread *t2 = new Thread("child 2", WillJoin);

        t1->Fork((VoidFuncionPtr) &Thread::Yield, t1);
        t2->Fork((VoidFuncionPtr) &Thread::Yield, t2);
        t2->Join();
        t1->Join();
}
```

Problem #3 (10 points)

For the following implementation of Thread::Join(), say whether it either (i) works, (ii) doesn't work, or (iii) is dangerous -- that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why and show how to fix it so it does work. You may assume parents always call Thread::Join() on their children threads; Thead::Join is a method on the child thread, not the parent.

```
class Thread {
        Semphore *finished;     // synch parent and child

        Thread::Thread() {
                finished == new Semaphore(0);   // initial value = 0
                // plus other standard stuff from the Nachos code
```

```
        }

        Thread::~Thread() {
                delete finished;
                // plus other standard stuff from the Nachos code
        }
};

void    // called by parent to wait for child thread
Thread::Join()
{
        finished->P();  // wait for thread to finish
}

void    // called by child thread when it is done
Thread::Finish()
{
        Thread *oldThread = kernel->currentThread;
        Thread *nextThread;

        (void) kernel->interrupt->SetLevel(IntOff);     // first turn i
        finished->V();  // then wake up parent
        delete this;    // deallocate current thread
        nextThread = kernel->scheduler->FindNextToRun();       // find
        kernel->currentThread = nextThread;
        SWITCH(oldThread, nextThread);  //context switch to it
}
```

Problem #4 (10 points)

For the following implementation of atomic transfer, say whether it either (i) works, (ii) doesn't work, or (iii) is dangerous -- that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why and show how to fix it so it does work. The problem statement is as follows: The atomic transfer routine dequeues an item from one queue and enqueues it on another. The transfer must appear to occur atomically: there should be no interval of time during which an external thread can determine that an item has been removed from one queue but not yet placed on another. In addition, the implementation must be highly concurrent -- it must allow multiple transfers between unrelated queues to happen in parallel. You may assume that queue1 and queue2 never refer to the same queue.

```
void AtomicTransfer (Queue *queue1, *queue2) {
        Item thing;     // thing being transferred

        queue1->lock.Acquire();
        thing = queue1->Dequeue();
        if (thing != NULL) {
                queue2->lock.Acquire();
                queue2->Enqueue(thing);
                queue2->lock.Release();
        }
```

```
        queue1->lock.Release();
}
```

Problem #5 (15 points)

A countermeasure is a strategy by which a user (or an application) exploits the characteristics of the CPU scheduling policy to get as much of the CPU time as possible. For example, if the CPU scheduler trusts users to give accurate estimates of how long each job will run so that it can give high priority to short jobs, then a countermeasure would be for the user to tell the system that the user's jobs are always short (even if untrue). Devise a countermeasure strategy for each of the following CPU scheduling policies; your strategy should minimize an indvidual application's response time (even if it hurts overall performance). You may assume perfect knowledge -- for example, your strategy can be based on which jobs will arrive in the future, where your application is in the queue and how long the jobs ahead of you will run before blocking. Your strategy should also be robust -- it whould work properly even if there are no other jobs in the system, or there are only short jobs, or only long running jobs, etc. If no strategy will improve your application's response time, then indicate that.

(a) last in first out

(b) round robin, assuming jobs are always put at the end of the ready list when they become ready to run

(c) multilevel feedback queues, where jobs are always put on the highest priority queue when they become ready to run

Problem 6: (20 points)

For the following implementations of the "H2O" problem, say whether it either (i) works, (ii) doesn't work, or (iii) is dangerous -- that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why and show how to fix it so it does work. Here is the original problem description: You've just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure hReady when it's ready to react, and each O atom invokes a procedure oReady when it's ready. For this problem, you are to write the code for hReady and oReady. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the procedures must call the procedure makeWater (which just prints out a debug message that water was made). After the makeWater call, two instances of hReady and one instance of oReady should return. Your solution must avoid starvation and busy-waiting. You may assume that the semaphore implementaion enforces FIFO order for wakeups -- the thread waiting longest in P() always grabs the semaphore after a V().

(a) Here is a proposed solution to the "H2O" problem:

```
int numHudrogen = 0;
Semaphore pair OfHydrogen(0);   // initially 0
Semaphore oxygen(0);                    // initially 0

void hReady() {
        numHydrogen++;
        if ((numHydrogen % 2) == 0) {
                pairOfHydrogen->V();
```

```
        }
        oxygen->P();
}

void oReady() {
        pairOfHydrogen->P();
        makeWater();
        oxygen->V();
        oxygen->V();
}
```

(b) Another proposed solution to the "H2O" problem:

```
Semaphore hPresent(0);          // initially 0
Semaphore waitForWater(0);      // initially 0

void hReady() {
        hPresent->V();
        waitForWater->P();
}

void oReady() {
        hPresent->P();
        hPresent->P();
        makeWater();
        waitForWater->V();
        waitForWater->V();
}
```

Extra Credit Question (2 points)

Which provides the best average response time when there are multiple servers (bank tellers, supermarket cash registers, airline ticket takers, etc.): a single FIFO queue or a FIFO queue per server? Why? Assume that you can't predict how long any customer is going to take at the server, and that once you pick a queue to wait in, you are stuck and can't change queues.
