CS 162 Midterm Exam
March 1996

Note that exam total is out of 70, not out of 100. Answers in italics.

Your Name: ***Ms. Perfect***

Your TA:

Your Section:

General Information:

This is a **closed book** examination. You have 90 minutes to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points given to the question; there are ***70*** points in all. Write all of your answers directly on this paper. Make your answers as concise as possible (you needn't cover every available nano-acre with writing). If there is something in a question that you believe is open to interpretation, then please go ahead and interpret **but** state your assumptions in your answer.

Problem 1: (5 points)

Of the following items, circle those that are stored in the thread control block.

Answer: a, c, f, g

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

main -> child1 -> child2 -> child1 -> child2 -> main
(If you assumed an implementation of join that V’ed the child to ask it to finish itself, then there would be an additional couple switches at the end -- we didn’t take off for this..)

Note that neither creating nor forking a thread switches to the child thread (assuming no preemptions occur)-- the parent keeps the CPU throughout, and just puts the child on the ready list. In the above example, the children don’t start running until “main” calls join.

Also note that when I fork the function Yield, when Yield runs, it will first call switch, then come back from switch, then the thread finishes. A common error was to assume that once Yield called switch, it never came back.

We gave some people half credit if we could deduce that you made only one of these errors.

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; Thread::Join is a method on the child thread, not the parent.

```
class Thread {
  Semaphore *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 interrupts off
    finished->V();  // then wake up parent
    delete this;    // deallocate current thread
    nextThread = kernel->scheduler->FindNextToRun());
                    // find the next thread to run
    kernel->currentThread = nextThread;
    SWITCH(oldThread, nextThread);
                    // context switch to it
}
```

Answer to problem 3: This is dangerous. There are actually 3 things wrong with the implementation:

1. if child finishes before parent gets to join, thread is deleted (along with semaphore used for synchronizing the parent and child). So when parent calls join, it is referencing a deallocated data structure (and someone else may in the meantime have allocated it for some other purpose, overwriting the semaphore value).

2. regardless of whether the parent gets to join before or after the child finishes, the child deletes its thread while still running in the context of the thread. This means the thread is still referencing its stack, even though the space for the stack may have been reallocated for some other purpose (and thus things like procedure call and return won't work properly).

3. There is no guarantee in Thread::Finish that there is another thread to run; thus, FindNextToRun might return NULL.

All three of these errors only show up sometimes, if exactly the right sequence of events occurs. For example, using data that has been deleted is dangerous, since it will only show up if the memory allocator hands out that data region again. You won't get an error if the memory allocator hands out some other data region.)

We were mainly looking for the first 2, so if you got both of those, and your fix worked, we gave you full credit. The simplest fix is to move the "delete this" to the last line in Thread::Join().

The most common error was noticing only one of the problems; we gave half credit if you came up with either of the first two problems.

Among the other common errors: (1) using an extra semaphore to have the child thread wait until the parent called join. This fixes #1, but not #2. (2) saying that the problem was that interrupts were never reenabled -- in Nachos, as in other systems, the thread that wakes up after a SWITCH is responsible for reenabling interrupts. (3) not realizing that Join is an operation on the child thread, even though the parent thread is the one that calls Join -- both Join and Finish operate on the same object and thus use the same semaphore.

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();
}
```

This is dangerous, since it may (but does not always) lead to deadlock. If one thread transfers from A to B, and another transfers from B to C and another from C to A, then you can get deadlock if they all acquire the lock on the first queue before any of them acquire the second.

The best fix from the point of view of maximizing concurrency is to use resource ordering -- to acquire both locks at the beginning of the routine, lowest addressed queue first:

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

   if (queue1 < queue2) {
      queue1->lock.Acquire();
      queue2->lock.Acquire();
   } else { // queue2 < queue1
      queue2->lock.Acquire();
      queue1->lock.Acquire();
   }
   thing = queue1->Dequeue();
   if (thing != NULL) {
      queue2->Enqueue(thing);
   }
   // release order doesn't matter
   queue1->lock.Release();
   queue2->lock.Release();
}
```

Another fix is to use a global lock -- this is not highly concurrent so we gave only partial credit for it. For example, you could put a global lock around the entire routine. Another fix is to use a global lock just to acquire the two queue locks, then release the global lock before dequeueing and enqueueing -- this is an example of acquiring all resources at the beginning. However, it is still not very concurrent: for example, if a thread needs to wait for one of the queue locks, then because it holds the global lock, no one else can make progress.

A common error was to say that the original code did not provide an atomic transfer, because another thread can access queue2 after the item has been removed from queue1. However, although the item has been removed from queue1 and not yet put on queue2, the external thread cannot observe this, since it can't get a lock on queue1 -- it can't tell if the item has been removed from queue1 or not. So from the point of view of an external thread, the transfer is atomic -- the external thread sees the state of the world either as it was before the transfer was made, or after, but not partway through.

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 individual 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 should 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.

Note that this question asked how you could improve a single job's response time, not how you could improve average response time across all jobs. Also note that the question is impractical in that it assumes you can have perfect knowledge; in practice, countermeasures are not as efficient as the ones described below.

(a) last in first out

If you assume LIFO is non-preemptive, then the best countermeasure is to insert your job onto the queue just before the currently running job finishes. The problem with this is what if your job does I/O? It would be hard to arrange for your I/O to complete just before another job finishes, which leads to the kill & restart approach outlined below.

If you assume LIFO is preemptive, then if another thread preempts your job (and that thread will take a long time), then you can kill your job and restart it -- the restarted job will go on the front of the queue. Alternatively, you might have your job, right before it is preempted, create a copy of itself and kill the original. If the copy does a brief Alarm::WaitUntil, it can leap over the arriving job back to the front of the line.

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

We gave full credit if you said no countermeasure possible. However, a few of you came up with clever solutions -- for instance, carving up your job into pieces, each of which runs for less than a time slice.

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

Similar to (a), one approach is, right before a higher priority job would preempt you, fork a job to run the remaining part of your job (or do a little bit of I/O at this point) This way, your job would always be high priority. Many suggested always doing I/O right before getting time-sliced, but this is inefficient if you are the only runnable job -- remember multilevel feedback timeslices every job to drop its priority, even if there is no one else at that priority level. We gave a large share of credit for this. Some simply said, do lots of I/O; again, that will work, but it is inefficient.

As with (b), another possible countermeasure is carving up your job into pieces, each of which runs for less than the minimum time slice.

Problem 6: (20 points)

For the following implementations of the “H20” 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 implementation enforces FIFO order for wakeups -- the thread waiting longest in P() always grabs the semaphore after a V().

Problem 6 (a) Here is a proposed solution to the “H20” problem:

```
int numHydrogen = 0;
Semaphore pairOfHydrogen(0); // init 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();
}
```

This solution is dangerous. Threads calling hReady() access shared data without holding a lock! Many of you talked about what would happen if you got a timeslice after the increment and before the test, but remember that you can also get a timeslice in the middle of a C statement. For example, you could have N threads both increment numHydrogen, and because they do so without a lock, the result could be 1 instead of N -- in other words, no water would be made regardless of how many H's arrived.

The solution is to put a lock acquire before the first line in hReady, and release before the Semaphore::P. Some put the lock acquire after the increment, and that simply doesn't work!

Problem 6 (b) Another proposed solution to the "H20" 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();
}
```

This is dangerous, since it may lead to starvation. If two H's arrive, then the value of the hPresent semaphore will be 2. If two O's arrive, then they can each decrement hPresent, before either can decrement it twice. So no water is made, even though enough atoms have arrived.

The fix is to put a lock acquire before the first line in oReady, and a lock release after the two semaphore:P's. This way, only one oxygen looks for waiting H's at a time -- if there aren't enough H's for the first oxygen, there won't be enough for any of the later oxygens either.

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.

A single queue is better. Consider what happens if when you arrive, no one is waiting, but every server is busy. You'd much rather wait in a single queue for the first server that becomes free, than have to commit to one of the queues -- it's likely you would be waiting while there is a server free. In queueing theory terms, multiple queues for multiple servers is not "work conserving" -- a server can be idle even though there are people/jobs waiting. A single queue is always work conserving.

There's a much more rigorous mathematical proof of why single queues are better than multiple queues in Kleinrock, Queueing Theory, Volume 1, along with lots more useful stuff. For instance, if you arrive at a random point in time at a bus stop, and the buses arrive at fixed intervals, then your average wait will be the half the inter-bus interval. What happens if the buses arrive randomly, but with the same average inter-bus interval? It turns out randomness hurts -- your average wait will always be longer than if the buses arrive at fixed intervals.