Overview: In this assignment, you are given part of a working thread system; your job is to complete it, and then to use it to solve a synchronization problem. The first step is to read and understand the partial thread system that has been written for you. This thread system implements thread fork and thread completion, along with semaphores for synchronization. Compile and run the program nachos to see a simple test of the code. Trace the execution path (by hand) for the simple test case provided.

When you trace the execution path, it is helpful to keep track of the state of each thread and which procedures are on each thread’s execution stack. You will notice that when one thread calls SWITCH, another thread starts running, and the first thing the newly running thread does is to return from SWITCH. This comment may seem cryptic to you at this point, but you will understand threads once you understand why the SWITCH that gets called is different from the SWITCH that returns. (Note: because the debuggers gdb and ddd do not understand threads, you will get bizarre results if you try to trace in gdb, or ddd, across a call to SWITCH.)

The files for this assignment, located in the threads/ and machine/ subdirectories, are:

- main.cc, threadtest.cc — a simple test of the thread routines;
- thread.h, thread.cc — thread data structures and thread operations such as thread fork, thread sleep and thread finish;
- scheduler.h, scheduler.cc — manages the list of threads that are ready to run;
- synch.h, synch.cc — synchronized access to lists using locks and condition variables (useful as an example of the use of synchronization primitives);
- system.h, system.cc — Nachos startup/shutdown routines;
- utility.h utility.cc — some useful definitions and debugging routines;
- switch.h, switch.s — assembly language magic for starting up threads and context switching between them;
- interrupt.h, interrupt.cc — manages enabling and disabling interrupts as part of the machine emulation (these are the only files in the machine subdirectory used in this project);
- timer.h, timer.cc — emulates a clock that periodically causes an interrupt to occur;
- stats.h — collects interesting statistics.

Properly synchronized code should work no matter what order the scheduler chooses to run the threads on the ready list. In other words, I should be able to put a call to Thread::Yield (causing the scheduler to choose another thread to run) anywhere in your code where interrupts are enabled.
without changing the correctness of your code. You will be asked to write properly synchronized code as part of the later assignments, so understanding how to do this is crucial to being able to do the project.

To aid you in this, code linked in with Nachos will cause Thread::Yield to be called on your behalf in a repeatable but unpredictable way. Nachos code is repeatable in that if you invoke ./nachos with the same arguments each time, it will do exactly the same thing. However, if you invoke ./nachos -rs #, with a different number each time, calls to Thread::Yield will be inserted at different (random) places in your code.

Make sure you run various test cases against your solutions to this problem; for instance, for part two, create multiple producers and consumers and demonstrate that the output can vary, within certain boundaries.

Warning: In this implementation of threads, each thread is assigned a small fixed-size execution stack. This may cause bizarre problems (such as segmentation faults at strange lines of code) if you declare large data structures to be automatic variables (e.g., int buf[1000]). You will probably not notice this during the semester, but if you do, you may change the size of the stack by modifying StackSize defined in switch.h.

You should write object-oriented code with C++ classes. Also, there should be no busy-waiting in any of your solutions to this assignment.

The Assignment:

1. Implement locks and condition variables. You may use either semaphores as a building block, or you may use more primitive thread routines (such as Thread::Sleep). The public interface to locks and condition variables is provided in synch.h. You need to define the private data and implement the interfaces. It should not require much code to implement either locks or condition variables.

2. Implement producer/consumer communication through a bounded buffer using locks and condition variables. The producer places characters from the string “Hello World” into the buffer one character at a time; it must wait if the buffer is full. The consumer pulls characters out of the buffer one at a time and prints them to the screen; it must wait if the buffer is empty. Test your solution with a multicharacter buffer and with multiple producers and consumers. Of course, with multiple producers or consumers, the output display will be gibberish.

3. Implement (non-preemptive) priority scheduling. Modify the thread scheduler to always return the highest priority thread. (You will need to create a new constructor for Thread to take another parameter — the priority level of the thread; leave the old constructor alone since you will need it for backward compatibility.) You may assume that there are a fixed, small number of priority levels — for this assignment, you’ll need two levels.

Can changing the relative priorities of the producer and consumer threads have any affect on the output? For instance, what happens with two producers and one consumer, when one of the producers is higher priority than the other? What if the two producers are at low priority, but the consumer is at high priority? Etc.

Your final submitted program must compile to a binary which is invoked as
./nachos -P num <other options as appropriate>

where the argument num can be the integer string "2" or "3" corresponding to a problem number above. The argument must cause the appropriate multithreaded system to be created so that I can exercise your system. Include sufficient output in your functions so that a user can determine what is happening, and in what order, but omit any debugging messages that you may have inserted to test your code.

**Submitting:** Follow the guidelines in the mechanics handout to submit your solutions.

**Professional Polish:** One possibility for professional polish points is to solve additional synchronization problems — ask me for details. For other ideas, consult with your group members.

**Due Date:** 11:59 PM, Wednesday, February 11, 2007