Nachos Assignment #1: Build a thread system

Tom Anderson
CSE 451

Due date: Tuesday, November 2, 5:00 p.m.

In this assignment, we give you part of a working thread system; your job is to complete it, and then to use it to solve several synchronization problems.

The first step is to read and understand the partial thread system we have written for you. This thread system implements thread fork, thread completion, along with semaphores for synchronization. Run the program **nachos** for a simple test of our code. Trace the execution path (by hand) for the simple test case we provide.

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 new thread does is to return from SWITCH. We realize this comment will 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 gdb does not understand threads, you will get bizarre results if you try to trace in gdb across a call to SWITCH.)

Properly synchronized code should work no matter what order the scheduler chooses to run the threads on the ready list. In other words, we 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 call it repeatedly with the same arguments, it will do exactly the same thing each time. However, if you invoke ‘‘nachos -rs \#”, with a different number each time, calls to Thread::Yield will be inserted at different places in the code.

Warning: in our 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 -- for example, *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 the StackSize define in switch.h.

Although the solutions can be written as normal C routines, you will find organizing your code to be easier if you structure your code as C++ classes. Also, there should be no busy-waiting in any of your solutions to this assignment.

Don't worry if you don't have to write much code for each of these; the assignment is largely conceptual, not a test of how much code you can write.

1. Implement condition variables using interrupt enable and disable to provide atomicity.

2. Implement synchronous send and receive of one word messages (also known as Ada-style rendezvous), using condition variables. Create a "MailBox" class with operations, "MailBox::Send(int message)" and "Mailbox::Receive(int *message)". Send atomically waits until Receive is called on the same MailBox; and then copies the message into the Receive buffer. Once the copy is made, both can return. Similarly, the Receive waits until Send is called, at which point the copy is made, and both can return. Your solution should work even if there are multiple Senders and Receivers for the same MailBox. (Hint: this is equivalent to a 0-length bounded buffer!)

3. Implement *void Thread::Join* () in Nachos. Add an argument to the thread constructor that says whether or not a Join will be called on this thread. Your solution should properly delete the thread control block whether or not Join is to be called, and whether or not the forked thread finishes before the Join is called. Note that you do not need to implement Join so that it returns the value returned by the forked thread. (This should seem trivial, but you'll find it a useful feature in later assignments.)

4. Complete the implementation of the "alarm clock" class defined in alarm.cc, by implementing the "WaitUntil(int x)" function. Threads call "WaitUntil" to suspend exeuction until time has advanced to at least *now + x*. This is useful for threads that operate in real-time, for example, for blinking the cursor once per second. For this part, there is no requirement that threads grab the processor immediately after waking up; just put them on the ready queue after they have waited for approximately the right amount of time.

5. Implement preemptive priority scheduling in Nachos; priority scheduling is a key building block for real-time systems. Add a call to the Thread class to set the priority of the thread. When a thread is added to the ready list that is higher priority than the currently running thread, the current thread gives up the processor to the new thread. Similarly, when threads are waiting for a lock, semaphore or condition variable, the highest priority waiting thread should be woken up first.

An issue with priority scheduling is "priority inversion" -- if a high priority thread needs to wait for a low priority thread (for instance, for a lock held by a low priority thread, or in Join for a thread to complete), and a middle priority thread is on the ready list, then the high priority thread will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is to have the waiting thread "donate" its priority to the low priority thread while it is holding the lock. Implement this fix.