Nachos Assignment 1: Threads and Synchronization

Sanzheng Qiao
Department of Computing and Software

August 27, 2007

Due: October 9, Tuesday, 11:59 pm.

In this assignment you are to modify and extend the existing thread system and solve several synchronization problems.

The fundamental concept to understand is context switch. Run the program nachos for a simple test of the existing thread system and trace the execution of Thread::SelfTest. 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 that this comment seems 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 program. You will be asked to write properly synchronized code used in 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 unpredicatable 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, Nachos is unpredictable in that 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 defined in threads/thread.h.

Although the solution 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.

1. Implement lock and condition directly using interrupt enable and disable to provide atomicity. We have provided implementations that use semaphores; your job is to provide alternative implementations without using semaphores. It is helpful to study the implementation of semaphore, which uses interrupt enable and disable to provide atomicity.

2. Implement synchronous send and receive of one word (integer) messages (also known as Ada-style rendezvous), using condition variables. Create a "Mailbox" class with operations `Send(int message)` and `Receive(int *message)`. Send atomically waits until Receive is called on the same Mailbox and moves the message into the Receive buffer. Once the message is moved, both can return (known as blocking send and blocking receive). Your solution should work even if there are multiple senders and receivers for the same mailbox.

3. Complete the implementation of the "alarm clock" class in threads/alarm by implementing

   `WaitUntil(int x)`.

   Threads call `WaitUntil(x)` to suspend execution until time has advances to at least `x` from now. This is useful for threads that operate in real-time, for example, for blinking the cursor once per second. There is no requirement that threads start running immediately after waking up; just put them on the ready queue after they have waited for approximately the right amount of time.