Nachos Assignment #1: Build a thread system

Tim Brecht
Computer Science 4321/5421
Due date: Tuesday – September 23, 1997, 2:30 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.)

The files for this assignment are:

main.cc, threadtest.cc — a simple test of our 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 — synchronization routines: semaphores, locks, and condition variables.

list.h, list.cc — generic list management (LISP in C++).

synchlist.h, synchlist.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 — manage enabling and disabling interrupts as part of the machine emulation.

timer.h, timer.cc — emulate a clock that periodically causes an interrupt to occur.

stats.h – collect 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, 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.

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

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 (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 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.

The assignment is items 1.a, 1.b, 2, 4, and 15 listed below.

1.a Implement locks and condition variables. You may either use semaphores as a building block, or you may use more primitive thread routines (such as *Thread::Sleep*). We have provided the public interface to locks and condition variables in synch.h. You need to define the private data and implement the interface. Note that it should not take you very much code to implement either locks or condition variables. Example code is provided to help get you started.

1.b Implement Barrier Synchronization. Barrier synchronization can be used in situation sin which threads need to synchronize before the end of execution (e.g., you may need to wait until all threads have finished executing before the computation is completed). The *Barrier::Reached()* primitive is executed when the threads need to synchronize. Each thread waits at the barrier for every other thread. After all threads have reached the synchronization point all threads proceed with their execution. Example code is provided to help get you started.

2. Implement producer/consumer communication through a bounded buffer, using locks and condition variables. (A solution to the bounded buffer problem is described in Silberschatz and Galvin, using semaphores and there are probably other solutions in the various texts listed in the course outline.)

   Your bounded buffer code is being implemented to ensure that your implemenations for part 1a and 1b are correct. As part of the input (command line or otherwise) you should determine the number of producer threads, the number of consumer threads and the size of the buffer. Each producer places a number into the buffer one number at a time; it must wait if the buffer is full. Each 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. One way to track which producer is producing what values is to have each of them start at different intervals (e.g., producer 0 starts at 0, producer 1 starts at 1,000, producer 2 starts at 2,000, etc). Test your solution with a different sized multi-numbered buffers and with different numbers of multiple producers and consumers. Of course, with multiple producers or consumers, the output display will be gobbledygook; the point is to illustrate that your implementation is correct. Be sure to demonstrate that your solution does execute correctly. Remember, to invoke "nachos -rs #", with a different number for different test runs.

4. Implement an "alarm clock" class. Threads call "Alarm::GoToSleepFor(int howLong)" to go to sleep for a period of time. The alarm clock can be implemented using the hardware Timer device (cf. timer.h). When the timer interrupt goes off, the Timer interrupt handler checks to see if any thread that had been asleep needs to wake up now. There is no requirement that threads start running immediately after waking up; just put them on the ready queue after they have waited for the approximately the right amount of time. Building this timer will be helpful when you implement round-robin preemptive scheduling in assignment 2.

15. You are to implement synchronization mechanisms to be used in a collaborative application. The application is a shared document processor. In this application multiple users at different locations can collaborate on the same document. Each document is divided into sections. Any number of people can read the same section at the same time but only one person can modify a section at a given time. Also, if one or more people are reading a section that section can not be modified until all of those people

are finished reading that section. In this application the priority of reading and writing is completely determined by the order of arrival of the readers and writers. For example, if three readers R1, R2, and R3 are reading Section A and a someone (W1) wants to modify Section A. W1 must wait until the three readers R1, R2, and R3 are finished. If while the writer (W1) is waiting a fourth reader (R4) arrives R4 would be required to wait until W1 is finished. The mechanisms you are required to are: *RWlock::ReadAcquire(), RWlock::ReadRelease(), RWlock::WriteAcquire()* and *RWlock::WriteRelease()*. These synchronization mechanisms will be useful when you implement/modify the file system in assignment 4.