NACHOS PROGRAMMING ASSIGNMENT 2

CS 370 – OPERATING SYSTEMS, LUMS

BUILD A THREAD SYSTEM

DUE DATE: THURSDAY OCTOBER 19, 2015, 11:59PM

This assignment has been adapted from the Nachos project of the CS 162 Course at Washington University.

'You should read the project mechanics document before starting work on this assignment.'

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 several synchronization problems.

The first step is to read and understand the partial thread system provided to 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 provided in the code. (You have already done this part in Programming Assignment 1).

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 the given 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.
The files for this assignment are:

- main.cc, threadtest.cc — a simple test of 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 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.

1

Implement locks and condition variables using semaphores as a building block. 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 you would not need to write much code to implement either locks or condition variables.

2

Implement ***send*** and ***receive***, using condition variables. Messages are sent on "*ports*" which allow senders and receivers to synchronize with each other. ***Send(port, int msg)*** atomically waits until ***Receive(port, int msg)*** is called on the same port, and then copies the msg 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. (Essentially, this is equivalent to a 0-length bounded buffer). Your solution should work even if there are multiple Senders and Receivers for the same port.

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.

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.

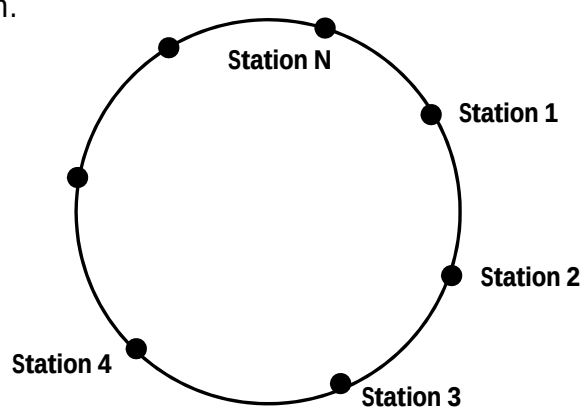
5

You have been hired by the Metropolitan Transit authority (MTA) to design controller their automated trains around the city in a circular loop. The stations in this loop are numbered 1 through N as depicted in the Figure below. Each station has a boarding area where passengers wait for the train in a queue. To enter the boarding area, passengers need to scan their ticket on an automated machine. The scanned information includes the source station and destination station of the passenger.

Suppose that the capacity of each train is 50 (i.e., only 50 passengers can be in the train at any given point in time). There are 5 trains running in the loop with an inter-arrival time of 5 minutes. When a train arrives at the station passengers waiting in the queue board the train. If the train becomes full, it does not take further passengers and the remaining passengers in the queue wait for the next train. A train stops at a given station only, if any of the following two conditions hold:

- There are one or more passengers waiting for the train in the boarding area of the given station.
- There are one or more passengers who need to get off from the train at the given station (i.e., these passengers have declared the given station as their destination station).

You can use semaphores, condition variables, or even send/receive to design the controller. Each passenger is represented by a thread. The passenger thread shouldn't exit until the passenger has reached the destination station.



THE END