Nachos Assignment #1: Build a thread system

CS 4411

Due date: 24-Feb-15 (Tuesday) 9:35am

Overview

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 a synchronization problem.

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 the synchronization problem below.

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.**

Requirements

The assignment items are listed below.

1. 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. The condition variables will adhere to Mesa semantics. The lock parameter to the Condition class methods is the lock required to get mutually exclusive access to the monitor. Note that the Broadcast method should awaken **all** processes waiting on the condition.

2. You have been hired to synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each car is represented by one thread, which executes the procedure *OneVehicle* when it arrives at the bridge:

```
OneVehicle(int direc)
{
  ArriveBridge(direc);
  CrossBridge(direc);
  ExitBridge(direc);
}
```

   In the code above, *direc* is either 0 or 1; it gives the direction in which the vehicle will cross the bridge.

   (a) Write the procedures *ArriveBridge* and *ExitBridge* (the *CrossBridge* procedure should just print out a debug message), using locks and condition variables. *ArriveBridge* must not return until it safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses). *ExitBridge* is called to indicate that the caller has finished crossing the bridge; *ExitBridge* should take steps to let additional cars cross the bridge. This is a lightly-travelled rural bridge, so you do not need to guarantee fairness or freedom from starvation.

(b) In your solution, if a car arrives while traffic is currently moving in its direction of travel across the bridge, but there is another car already waiting to cross in the opposite direction, will the new arrival cross *before* the car waiting on the other side, *after* the car on the other side, or is it impossible to say? Explain briefly.

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 we'll need it for backward compatibility.) You may assume that there are a fixed, small number of priority levels – for this assignment, you'll only need two levels. Let the value zero denote a high priority process and let the value one denote a lower priority process.

   Can changing the relative priorities of the producers 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 the same priority, but the consumer is at higher priority? (Submit your answers along with output from the execution of test cases that verify the answer to these questions.)

4. Nachos is a user-level thread system. Understanding the details of Nachos thread management will help you to understand how a user-level thread system can be written and the difference between user-level and kernel-level threads. Toward that end, answer the following questions about Nachos threads. Put the answers in your README.

   (a) Explain what causes Nachos time to advance.

   (b) A Nachos thread executes more code than the routine passed in to Fork(). What other routines in the Nachos code are executed by a thread? Explain.

   (c) Describe how a context-switch between threads takes place on the 386 architecture. What causes a context switch? Where is the state for a currently executing thread saved when it is switched out?

5. An operating system should never fail. If you've ever been frustrated by an operating system locking up, then you understand why this is unacceptable. This means that you must test your code to ensure it operates correctly under all foreseeable circumstances. In this assignment, convince me that your implementations of locks and condition variables and priority scheduling are correct and robust by submitting test code that exercises the implementations. Your README should enumerate the tests that you ran and give a brief description of each test. If you must make assumptions during the implementation, then describe these in the README. The README should also contain answers to any questions posed in the numbered problem statements above.

Notes

The test code for this project uses the ThreadTest() interface. That is, the code for each test follows the following pattern.

```
void ThreadTest(){
---
TEST CODE
---
}
```

The code for the test is placed in a file named `aTest.cc`. The file is then linked to `threadtest.cc`. Nachos is compiled and run and the test executes. Hence, do not change the default behavior for invocation of

`ThreadTest()`.

Put your bridge implementation and test code into a file named `bridge.cc` and invoke it via the `ThreadTest()` routine. Your bridge code will be run with the following sequence of commands.

```
cd threads;
mv threadtest.cc threadtest.orig.cc
ln -s threadtest.cc bridge.cc
cd ..
make
threads/nachos -rs X
```

Points may be deducted if this sequence of commands does not successfully execute your bridge implementation.