Lab 2: Building a Thread System
CMSC 442
Due: Friday, Sept. 29 by 11:59:59pm

(Note: This lab, and all the other Nachos labs were derived from the original NachOS projects by Tom Anderson at UC Berkeley. They have been modified to fit our lab environment and changes in the compilation software since NachOS was originally published.)

Step 0. Getting Started

The next three or four labs in this course will be group projects that require you to make improvements to a rudimentary operating system called NachOS. NachOS was originally developed by Tom Anderson at U.C. Berkeley. Most of the materials I will provide you were developed by him in the early nineties. NachOS isn't designed to run on Intel-based computers like the ones in the lab – it requires a machine that uses a MIPS-based assembly language – so to use NachOS, we deploy a virtual machine image. Fortunately, NachOS comes with tools to make running the VM relatively easy.

You should start your work by downloading the nachos code to some place in your home directory. You can do this by typing:

```
git clone gitosis:groupX nachos
```
(replace groupX with your actual "group" login name)

Some of the tasks in this project will be "group" projects that all of you are responsible for. I will also assign a set of "individual" tasks which a single member of your group will take the lead on. You can and should help each other with your individual tasks, but points for that task will be weighted much more heavily into the grade of the "owner" of the task.

I have tried to make the individual tasks roughly equal in difficulty, but it is inevitable that some will be easier or harder than others. I leave it to you to decide how to make this work out fairly in your groups.

It is important that you complete ALL the individual tasks. I have tried to make the individual tasks less critical to the proper functioning of the future projects than the group tasks, but in a few places there are still going to be some dependencies on previous work. Furthermore, given the nature of operating system code, errors in the implementation of an individual task can cause the entire operating system to crash or exhibit undefined behavior in some circumstances. In particular, Task 3 of this lab is critical for several subsequent projects, so make sure you get it right. This shouldn't be just one group member's responsibility. Use "pair programming", "code review", and other good software engineering practices to ensure that more than one set of eyes has looked at every line of code you turn in.

Step 1. Understanding threads

In this assignment, we give you part of a working thread system: your job is to complete it and then 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, using gdb) for the simple test case we provide in the *threads/threadtest.cc* file. The main function in *threads/main.cc* calls the function *ThreadTest()*, so this is probably a good place to start. As you trace through the execution of the thread system, you will encounter many other files and data structures used by NachOS. Take time to explore them and get a feel for what they do.

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, **a different 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 NachOS threads, you will get bizarre results if you try to use gdb to step through a call to SWITCH!)

The files for this assignment are:

main.cc, threadtest.cc	– a simple test of our thread routines (and which contains the "main" function).
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.cc	– assembly language magic for starting up threads and context switching between them.
interrupt.h, interrupt.cc	– manage enabling and disabling of interrupts as part of the machine emulation.
timer.h, timer.cc	– emulate a clock tick that periodically causes an interrupt to occur.
stats.h	– collect interesting statistics

The files you will probably spend the most time editing are synch.cc and thread.cc, but you will need to understand the other files and how they relate to each other as well. You are going to need to spend lots of time **reading code** and trying to understand it. Do not underestimate this part of the project – it will probably take **six to eight hours** to even make a good start reading through the project code. I highly recommend you take notes as you read through the code.

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 multiple times with the same arguments, it will do exactly the same thing each time. However, if you invoke "./nachos -rs #", replacing "#" 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 using different random seeds (that is, different numbers with the -rs flag).

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[100];"). 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.

Nachos is written in C++. Most real operating systems (including both Linux and Windows) are written in C, rather than in C++. This reduces the amount of wasted time and space imposed by using a high-level programming language. Although the solutions to these labs 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.

Step 2. Tasks

Implement each of the following tasks. The first task should be done as a group and you will probably need to get it working before you can implement the remaining tasks. That doesn't mean you can't sit down and think through how you will implement them while you are working on task one, though.

The three individual tasks should be assigned to a particular member of your group. You can all work on them, but one person in your group is responsible for THAT task and their grade on the project will largely depend on successful implementation of that task. I leave it up to you to decide who will take which task – I will not play referee in your internal group politics.

Task 1 (To be completed by the entire group):

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 it should not take you very much code to implement either locks or condition variables. If your code starts getting messy, you've probably gone wrong somewhere!

A lock is like a "mutex" semaphore, except that it can be in only one of two states: locked or unlocked. Exactly one thread can "hold" the lock at a time and no other thread can acquire the lock until the thread that holds it has unlocked it. If a thread which doesn't hold the lock tries to acquire the lock, it is put to sleep. When a thread releases a lock, exactly one waiting thread (if any are available) should be woken up and immediately acquire the lock. If no threads are waiting, the lock should be left open.

Unlike semaphores, releasing a lock multiple times does not allow multiple threads into the critical section. Locks can (and should) be implemented using semaphores. You may need other variables or data structures as well. Please read the comments in *threads/synch.h* and be sure you understand them.

A "condition variable" is another synchronization construct that can be implemented with semaphores (or, really, with locks that use semaphores). Threads can wait on a condition, signal that they are done with a condition, or send a broadcast that wakes up all threads waiting on the condition.

Condition variables are used to protect code without creating a deadlock. When a thread that holds a lock does something that might cause it to go to sleep, it "waits" on a condition. This allows it to "give up" the lock temporarily and then go to sleep. To wake it up, another thread will "signal" the condition. This causes one of the threads waiting on the condition to wake up and immediately try to re-acquire the lock it gave up. Once it acquires the lock, it proceeds.

A "broadcast" behaves like a "signal" except that it wakes up ALL threads waiting on the condition. Only one of them will successfully acquire the lock (the others will sleep, waiting on the lock), but this is handled by the "lock" implementation, not the "condition" implementation.

You must implement locks and conditions in such a way that:

1. The description above is followed.
2. If used correctly, no deadlocks occur.
3. If used correctly, no two threads can ever enter a critical section protected by the lock.

Task 2 (To be implemented by a single member of the group):

Create new files named “comm.h” and “comm.c” that implement a “channel-based message passing system” as follows:

In comm.h add prototypes for two functions, void Send(int port, int msg) and void Receive(int port, int* msg). Then, in comm.cc, implement functions for send and receive, using condition variables. Messages are sent on “ports”, which allow senders and receivers to synchronize with each other. You may assume that port number is in the range 0 to 255.

Send(port, msg_in) atomically waits until another thread calls Receive(port, msg_pointer) is called on the same port, and then copies msg_in into the Receive buffer (pointed to by the msg_pointer parameter). Once the copy is made, both functions can return.

Similarly, Receive waits until Send is called by another thread, 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. This is very similar to the “leaders-followers” semaphore pattern. A good way to implement this is to create a “Port” class to handle these operations and make a global array of 256 port objects in comm.c.

Task 3 (To be implemented by a single member of the group):

Implement Thread::Join in Nachos. The Join function causes the currently running thread to wait for some other thread (passed as a parameter to the Join function) to terminate. The other thread should be a child of the currently running thread, created by the “Fork” function.

Your solution should properly delete the thread control block when the thread finishes, whether or not Join is to be called, and whether or not the forked thread finishes **before** the Join is called. To prevent your system from hanging if a thread is never joined, you should add a boolean argument named “joinable” to the thread constructor that indicates whether or not Join will ever be called on this thread. You may assume that all “joinable” threads eventually get “Join” called on them in my test cases. Set the default value of this argument to “false”.

Getting this task correct is essential to the proper working of your future NachOS projects, so be sure you test it carefully.

Task 4 (To be implemented by a single member of the group):

Create two new files “alarm.h” and “alarm.cc” and add them to the Makefile in the threads/ directory. In these files, 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 (see the file “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 approximately the right amount of time.

Step 3. Documentation

In your base nachos directory, I would like you to create a file named LAB2-README.TXT that contains:

Your group name, the names of each member of your group, and a description of the contributions of each member (in paragraph form).

Your description should be detailed, explaining which files you changed and which task they implemented. If you were unable to get part of a task working, you should be explicit about that and explain the problem in detail.

This documentation is a significant part of your grade. Don't skimp on it. Be detailed.

Submit as usual by uploading a tarball to the course web site at http://marmorstein.org/~robert/submit/