Lab Threads

10th of September, 2021

Getting started

There are both *group* and *individual* tasks in this project. Individual tasks are the responsibility of a single group member. Group tasks are the shared responsibility of members in the group.

Your group should have a `git` repository containing the `NachOS` code base and including Dr. Dymacek in your group.

Understanding threads

The first assignment is about implementing a working thread system and solving several synchronization problems. You will need to read and *understand* the `NachOS` implementation in `threads` directory.

To start you should compile `NachOS` by running `make` from inside the `threads` directory. After compiling run the `./nachos` programming. Read through `threads/threadtest.cc` for an example test case.

When you trace a threads 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.
- `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

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 or structs. **Also, there should be no busy-waiting in any of your solutions to this assignment**.

Group Task

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.

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 re-acquire the lock it gave up. 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, 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.

You will build test cases to demonstrate that your implementations work. You will need to include appropriate random seeds as well.

Individual Task 1

Create new files named `pipe.h` and `pipe.cc` that implement a *"pipe message passing system"* as follows:

In `pipe.h` build a struct or class named `Pipe` and add prototypes for two functions, `void Send(int msg)` and `void Receive(int* msg)`. Then, in `pipe.cc`, implement functions for send and receive, using condition variables. Messages are sent on *"pipes"*, which allow senders and receivers to synchronize with each other. `Send(msg_in)` atomically waits until `Receive(msg_pointer)` is called on the same pipe, and then copies msg_in into the buffer pointed to by `msg_ pointer`. 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 pipe.

You should also implement a `Close()` function in your `Pipe` structure. After the pipe has been *"closed"* neither `Send` or `Receive` should block. Future calls to `Send` should do nothing and future calls to `Receive` should fill the buffer with `-1`.

Individual Task 2

Implement `Thread::Join` in `NachOS` . Add an argument to the thread constructor that says whether or not a `Join` will ever be called on this thread. Then create a `Join` function.

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.

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

Individual Task 3

Create two new files `alarm.h` and `alarm.cc`. 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.

Handing in

Create a README.txt file which describes what each group member has done. You should include which files have been changed and why. If you were unable to get something to work, explain the problem. The documentation is an important part of the project. You will need to demo your code for me after the due date.

This lab will be due on 09/22/2021 at 11:59pm.

This assignment is adapted from one by Dr. Robert Marmorstein and 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.