EECS 111
System Software
Spring 2016
Project 2: Threading System

Due Date (May 3rd, 2016 11:59 PM)

Instructions

Stock Nachos has an incomplete thread system. In this assignment, 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 (use the archive file provided along the project). This thread system implements thread fork, thread completion, and semaphores for synchronization. It also provides locks and condition variables built on top of semaphores.

After installing the Nachos distribution, run the program nachos (in the proj2 subdirectory) for a simple test of our code. This causes the methods of `nachos.threads.ThreadedKernel` to be called in the order listed in `threads/ThreadedKernel.java`:

1. The `ThreadedKernel` constructor is invoked to create the Nachos kernel.
2. This kernel is initialized with `initialize()`.
3. This kernel is tested with `selfTest()`.
4. This kernel is finally "run" with `run()`. For now, `run()` does nothing, since our kernel is not yet able to run user programs.

Trace the execution path (by hand) for the simple test cases 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 `TCB.contextSwitch()`, that thread stops executing, and another thread starts running. The first thing the new thread does is to return from `TCB.contextSwitch()`. We realize this comment will seem cryptic to you at this point, but you will understand threads once you understand why the `TCB.contextSwitch()` that gets called is different from the `TCB.contextSwitch()` that returns.

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 `KThread.yield()` (causing the scheduler to choose another thread to run) anywhere in your code where interrupts are enabled, and your code should still be correct. 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 `KThread.yield()` to be called on your behalf in a repeatable (but sometimes 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 -s <some-long-value>`", with a different number each time, calls to `KThread.yield()` will be inserted at different places in the code.

You are encouraged to add new classes to your solution as you see fit; the code we provide you is not a complete skeleton for the project. Also, there should be no busy-waiting in any of your solutions to this assignment.

Grading

Your project code will be automatically graded. There are two reasons for this:

1. A grader program can test your code a lot more thoroughly than a TA can, yielding more fair results.
2. An autograder can test your code a lot faster than a TA can.

Of course, there is a downside. Everything that will be tested needs to have a standard interface that the grader can use, leaving slightly less room for you to be creative. Your code must strictly follow these interfaces (the documented *Interface classes).
Since your submissions will be processed by a program, there are some very important things you must do, as well as things you must not do. For all of the projects in this class...

1. Do not modify `Makefile`, **except** to add non-java source files (useful in later projects). We will be using our own Makefile. (`javac` automatically finds source files to compile, so we don't need you to submit `Makefile`).
2. Only modify `nachos.conf` according to the project specifications. We will also being using our own `nachos.conf` file. Do not rely on any additional keys being in this file.
3. Do not modify any classes in the `nachos.machine` package, the `nachos.ag` package, or the `nachos.security` package. Also do not add any classes to these packages. They will not be used during grading.
4. Do not add any new packages to your project. All the classes you submit must reside in the packages we provide.
5. Do not modify the API for methods that the grader uses. This is enforced every time you run Nachos by `Machine.checkUserClasses()`. If an assertion fails there, you'll know you've modified an interface that needs to stay the way it was given to you.
6. Do not directly use Java threads (the `java.lang.Thread` class). The Nachos security manager will not permit it. All the threads you use should be managed by TCB objects (see the documentation for `nachos.machine.TCB`).
7. Do not use the `synchronized` keyword in any of your code. We will `grep` for it and reject any submission that contains it.
8. Do not directly use Java `File` objects (in the `java.io` package). In later projects, when we start dealing with files, you will use a Nachos file system layer.

When you want to add non-java source files to your project, simply add entries to your `Makefile`.
In this project, you will **not** need to add any new files.

1. The only package you will submit is `nachos.threads`, so don't add any source files to any other package.
2. The `autograder` will not call `ThreadedKernel.selfTest()` or `ThreadedKernel.run()`. If there is any kernel initialization you need to do, you should finish it before `ThreadedKernel.initialize()` returns.
3. There are some mandatory `autograder` calls in the `KThread` code. Leave them as they are.

Submission

In this project, you will modify couple of java files in threads folder. A dropbox folder is provided in EEE website. You need to compress all the files in threads folder into a single archive file (zip, rar, etc.) and upload it to **EECS111 Assignment 2** directory. Make sure the directory name and the zip file name are *{Student ID}_A2*. The deadline for uploading the files is the project deadline.

Task

I. (20%, 5 lines) Implement `KThread.join()`. Note that another thread does not have to call `join()`, but if it is called, it must be called only once. The result of calling `join()` a second time on the same thread is undefined, even if the second caller is a different thread than the first caller. A thread must finish executing normally whether or not it is joined.

II. (20%, 20 lines) Implement condition variables directly, by using interrupt enable and disable to provide atomicity. We have provided a sample implementation that uses semaphores; your job is to provide an equivalent implementation without directly using semaphores (you may of course still use locks, even though they indirectly use semaphores). Once you are done, you will have two alternative implementations that provide the exact same functionality. Your second implementation of condition variables must reside in class `nachos.threads.Condition2`.

III. (25%, 40 lines) Complete the implementation of the Alarm class, by implementing the `waitUntil(long x)` method. A thread calls `waitUntil` to suspend its own execution until time has advanced to at least now + `x`. This is useful for threads that operate in real-time, for example, for blinking the cursor once per second. There is no requirement that threads start running immediately after waking up; just put them on the ready queue in the timer interrupt handler after they have waited for at least the right amount of time. Do not fork any additional threads to implement `waitUntil()`; you need only modify `waitUntil()` and the timer interrupt handler. `waitUntil` is not limited to one thread; any number of threads may call it and be suspended at any one time.

IV. (35%, 40 lines) Implement synchronous send and receive of one word messages (also known as Ada-style rendezvous), using condition variables (don't use semaphores!). Implement the Communicator class with operations, `void speak(int word)` and `int listen()`. `speak()` atomically waits until `listen()` is called on the same Communicator object, and then transfers the word over to `listen()`. Once the transfer is made, both can return. Similarly, `listen()` waits until `speak()` is called, at which point the transfer is made, and both can return (`listen()` returns the word). This means that neither thread may return from `listen()` or `speak()` until the word transfer has been made. Your solution should work even if there are multiple speakers and listeners for the same Communicator (note: this is equivalent to a zero-length bounded buffer; since the buffer has no room, the producer and consumer must interact directly, requiring that they wait for one another). Each communicator should only use exactly one lock. If you're using more than one lock, you're making things too complicated.