Threads and Synchronization

In this project module you will work on implementing some features required for proper process synchronization in Nachos. Nachos simulates multi-processing using Java threads for their execution and are represented in the simulation by the KThread class, with the state of each thread represented as an instance of the TCB class. Running in programs in Nachos can be done either by subclassing the AutoGrader class, or by running COFF binaries (MIPS). For this project, you will do the former.

I have linked to this post a Nachos tutorial PDF that was nicely put together by folks at McMaster University. It would be a good idea to read some of this file to understand how Nachos works, at least at a high level. It also explains what the different command line options for Nachos are.

As you work on this project keep in mind the following:

If for some reason you set up Nachos to work outside of Eclipse in an Unix environment and build it with Make, DO NOT modify the Makefile except to add sources (if you do not understand what this is, you don't have to worry about it).

Only modify nachos.conf according to the project's specifications. For this project, basically DO NOT modify it, except to change the scheduler.

DO NOT modify any classes outside the nachos.threads package, and only those required for the project.

DO NOT add any new packages to your project.

DO NOT modify the API for the methods the auto grader uses. Basically, don't mess with Nachos API.

DO NOT directly use Java threads (java.lang.Thread class). Use only Nachos threads that are managed by TCB objects.

DO NOT use the synchronized keyword in your code. If you need to enforce mutual exclusion, use locks and semaphores provided by Nachos.

DO NOT use Java File objects (java.io.File class).

Before you start the project it would be a good idea to get acquainted with the Nachos system, by running the self test, and tracing its execution. The self test is the program Nachos runs when your run it in its default state (like we did in class). Upon booting, Nachos initializes the system, allocates the specified auto grader instance, and runs it (by default, the AutoGrader class). Right now, it simply runs a couple of tests and exits. Here are the tasks you must complete:

Task 1: Condition Variables (10%)

Implement condition variables directly by using interrupt enable and disable to provide atomicity. The implementation of condition variables included in the system uses semaphores to achieve this, so your implementation MUST NOT use semaphores (however, you may use locks). Your new implementation must reside on nachos.threads.Condition2.

Task 2: KThread (10%)

Implement KThread.join() using condition variables. Keep in mind that implement this functionality you may add code to / modify other parts of the KThread class. But you must not break existing functionality.

Task 3: Alarms (20%)

Complete the implementation of the Alarm class, by implementing the waitUntil(long x) method using condition variables. 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.

Task 4: Communicator (60%)

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.

***

To help you check your work I have attached a zip file with several auto grader classes. Check the tutorial file for instructions on how to run other auto grader classes. You may add other tests to the auto graders if you want (or even create your own auto graders). You should also change random seed when testing.

Submission

Submit a zip file with your solution. Your submission should include source code and a written report with a detailed discussion of your design and implementation choices. The final grade will be 40% for your written report, and 60% for the correctness of your implementation.

This project is worth 100 points.