Project 2: Threads and Synchronization

Readings: Chapter 3, 4.1, Nachos Tutorial
Due date: Feb 13th, 2015
TA: Ala Shaabana (shaabaa)

January 21, 2015

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.

For all of the projects in this class,

(1) Do not modify Makefile, except to add source files.

(2) Only modify nachos.conf according to the project specifications.

(3) Do not modify any classes in the `nachos`.machine package, the `nachos`.ag package, or the `nachos`.security package. You will not be submitting any of these classes.

(4) Do not add any new packages to your project. All the classes you submit must reside in the packages provided.

(5) Do not modify the API for methods that the autograder 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.

1 Getting ready

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, and semaphores for synchronization. It also provides locks and condition variables built on top of semaphores. Take a look at the source code of *Condition.java* and answer the following questions:

(1) Does the code implement mesa style or hoare style monitors? Justify your answer.

(2) In class, we learn that the value of semaphore S is the number of units of the resource that are currently available. What are the possible values of the semaphore `waiter` in *Condition.java*?

(3) In `sleep()`, why is the lock released before `waiter.P()` and acquired after? When will `conditionLock.acquire()` be called?

(4) Why is `waitQueue` needed? Can we just use a single semaphore?

(5) In `sleep()` and `wake()`, a shared object `waitQueue` will be accessed. What prevents it being accessed by two threads at the same time?

2 Tasks

(1) (10%) 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). 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`.

(2) (10%) Implement `KThread.join()` using **condition variable** Note that another thread does not have to call join(), but if it is called, it must be called only once. A thread must finish executing normally whether or not it is joined.

(3) (20%) Complete the implementation of the `Alarm` class, by implementing the `waitUntil(long x)` method using **condition variable**. 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.

(4) (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.

[*Hints: Pay attention to the use of mesa style monitor, namely, you cannot assume an awaken thread is immediately executed*]

3 Testing

Test your codes with the tester classes in *ag/* directory. Check the *readme* file for the purpose of individual testers. You can modify the tester codes to include more test cases. Your codes will be tested automatically with a more comprehensive set of testers by the TAs. For testing, you can change the random seed by including "-s seed" in the command line (where seed is an integer).

4 Submission

You should **commit** your codes through the SVN repository. *Make sure the directory structure is intact.* Under the current nachos project directory, include a file called "report.pdf" (e.g, nachos/proj2/report.pdf for this project). In the report, you should include the following two parts:

1. Answers to the questions in the project description.
2. Key data structures used and a block diagram of your implementation (limited to 2 pages).

Grade of each project is divided as 20% for Q&A, 40% for key data structures/block diagram on the implementation and 40% for the correctness of your implemention with the respective percentage for each task.