Project 1: Threads

Fall 2019

~~**Due**: Wednesday, October 23, at 11:59pm~~
Due: Friday, October 25, at 11:59pm

The baseline Nachos implementation has an incomplete thread system. In this project, your job is to complete it, and then use it to solve synchronization problems.

Background

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 <number>` with a different `number` each time, calls to `KThread.yield` will be inserted at different places in the code.

You will be modifying source code files in the `threads` subdirectory, and compiling in the `proj1` subdirectory. There should be no busy-waiting in any of your solutions to this assignment. (The initial implementation of `Alarm.waitUntil` is an example of busy-waiting.)

Tasks

0. (0%) Be sure to have your group registered on the Google form so that we can create a course repo on github for you (see the pinned Piazza note). This step applies even if you are working on your own. We can only grade repos that were created for the course.

   Browse through the initial thread system implementation, starting with `KThread.java`. This thread system implements thread fork, thread completion, and semaphores for synchronization. It also provides locks and condition variables built on top of semaphores.

   Trace the execution path (by hand) for the startup test case provided. 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 will seem cryptic to you at first, but you will understand threads once you understand why the `TCB.contextSwitch` that gets called is different from the `TCB.contextSwitch` that returns.

Compile and run the baseline implementation of Nachos in the `proj1` directory:

```
% cd nachos/proj1
% make
% nachos
```

The output should be the same as in project 0 before your changes.

1. (20%) Complete the implementation of the `Alarm` class (except for `cancel`, which you will implement later) . A thread calls `waitUntil(long x)` to suspend its execution until wall-clock time has advanced to at least *now* + *x*. This method is useful for threads that operate in real time, such as 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 methods. `waitUntil` itself, though, is not limited to being called by one thread; any number of threads may call it and be suspended at any one time. If the wait parameter `x` is 0 or negative, return without waiting (do not assert).

   Note that only one instance of `Alarm` may exist at a time (due to a limitation of Nachos), and Nachos already creates one global alarm that is referenced via `ThreadedKernel.alarm`.

   Testing: Implement tests that verify that a thread waits (approximately) for its requested duration; if the wait parameter is 0 or negative, the thread does not wait; multiple threads waiting on the alarm are woken up at the proper times, and in the proper order. For examples and strategies for implementing tests, see the Testing section below.

2. (20%) Implement `KThread.join`, which synchronizes the *calling* thread with the completion of the *called* thread. As an example, if thread `B` executes the following:

```
KThread A = new KThread (...);
...
A.join ();
```

   we say that thread `B` joins with thread `A`. When `B` calls join on `A`, there are two possibilities. If `A` has already finished, then `B` returns immediately from join without waiting. If `A` has not finished, then `B` waits inside of join until `A` finishes; when `A` finishes, it resumes `B`. Often thread `B` is called the "parent" and `A` is called the "child" since a common pattern is for a thread that creates child threads to join on them to wait for them to finish. However, note that any thread can call `KThread.join` on another (it does not have to be a parent/child relationship).

Note that: (1) join does not have to be called on a thread. A thread should be able to finish successfully even if no other thread calls join on it; (2) A thread cannot join to itself. (The initial implementation already checks for this case and invokes Lib.assert when it happens. Keep this Lib.assert call in your code); (3) Join can be called on a thread at most once. If thread `B` calls join on `A`, then it is an error for `B` or any other thread `C` to call join on `A` again. Assert on this error.

Testing: Implement tests that verify if `B` calls join on `A` and `A` is still executing, `B` waits; if `B` calls join on `A` and `A` has finished executing, the `B` does not block; if a thread calls join on itself, Nachos asserts; if join is called more than once on a thread, Nachos asserts; one thread can join with multiple other threads in succession; independent pairs of threads can join with each other without interference.

3. (25%) Implement condition variables using interrupt enable and disable to provide atomicity. The class `Condition` is a sample implementation that uses semaphores, and your job is to provide an equivalent implementation in class `Condition2` by manipulating interrupts instead of using semaphores. Once you are done, you will have two alternative implementations that provide the exact same functionality. Examine the existing implementation of class `Semaphore` to guide you on how to manipulate interrupts for when you implement the methods of `Condition2`. For this part, you do not have to implement `sleepFor` (you will implement it in the next part).

   A thread must have acquired the lock associated with the condition variable when it invokes methods on the CV. The underlying implementation of the `Lock` class already has code to assert in these cases, but we recommend writing a test program that causes such an error so that you can see what happens.

   Testing: Implement tests that verify that `sleep` blocks the calling thread; `wake` wakes up at most one thread, even if multiple threads are waiting; `wakeAll` wakes up all waiting threads; if a thread calls any of the synchronization methods without holding the lock, Nachos asserts; `wake` and `wakeAll` with no waiting threads have no effect, yet future threads that `sleep` will still block (i.e., the `wake/wakeAll` is "lost", which is in contrast to the semantics of semaphores).

4. (10%) Since threads waiting on condition variables typically do so in a loop, checking whether the situation they are waiting for is true after waking up, it is usually safe to wake up threads waiting on condition variables at any time. As a result, condition variables implemented in many programming languages also support a "scheduled wait" operation where threads can wait on a condition variable with a timeout. Implement the `sleepFor` method of `Condition2` and the `cancel` method of `Alarm` to provide this functionality (modifying other methods as necessary). With `sleepFor(x)`, a thread is woken up and returns either because another thread has called `wake` as with `sleep`, or the timeout `x` has expired.

   Testing: Implement tests that verify that a thread that calls `sleepFor` will timeout and return after `x` ticks if no other thread calls `wake` to wake it up; a thread that calls `sleepFor` will wake up and return if another thread calls `wake` before the timeout expires; `sleepFor` handles multiple threads correctly (e.g., different timeouts, all are woken up with `wakeAll`).

5. (25%) Implement the `Rendezvous` class to provide a mechanism for threads to exchange values, using locks and condition variables to manage concurrency. `Rendezvous` provides similar functionality as the rendezvous system call from the Plan 9 operating system. The same functionality is also provided by the Exchanger class in Java (but you cannot use Exchanger in your implementation).

   Rendezvous has just one method, `exchange`. The first thread `A` to call exchange with value `X` will block waiting for another thread `B`. When thread `B` calls exchange with value `Y`, it will unblock `A` and the threads will exchange values: value `Y` will be returned to thread `A`, and value `X` will be returned to thread `B`.

   `exchange` also takes a `tag` argument. Different integer tags are used as different, parallel synchronization points (i.e., threads synchronizing on different tags do not interact with each other). The same tag can also be used repeatedly for multiple exchanges. Note that there can be different instances of `Rendezvous`, each of which would synchronize threads completely independently of each other.

   Tip: Implement `Rendezvous` in stages. When starting, ignore tags and assume threads are synchronizing on the same tag. First implement synchronizing two threads exchanging values, and test. Then extend your implementation to many threads exchanging values on the same tag, and test. Then handle multiple tags, and then multiple instances of Rendezvous.

   Testing: Implement tests that verify that: a thread only returns from `exchange` when another thread synchronizes with it; `exchange` returns the exchanged values from the threads properly; many threads can call `exchange` on the same tag, and `exchange` will correctly pair them up and exchange their values; threads exchanging values on different tags operate independently of each other; threads exchanging values on different instances of Rendezvous operate independently of each other.

Testing

It is your responsibility to implement your own tests to thoroughly exercise your code to ensure that it meets the requirements specified for each part of the project. Testing is an important skill to develop, and the Nachos projects will help you to continue to develop that skill. You can add calls to testing code in `ThreadedKernel.selfTest`, and add class-specific code in `selfTest` methods of each class.

As a testing strategy, first start with simple tests and then implement more complicated tests. When something goes wrong with a simple test, it is easier to pinpoint what aspect of your implementation has a bug. When something goes wrong with a more complicated test, it is more difficult to determine where the bug may be unless you can rule out all the causes that your simple tests have shown to already be correct. We also strongly recommend implementing tests as separate methods, rather than making changes to just one or a few methods. Rather than making a change to an existing test to evaluate new functionality, copy the test into a new method and make the change. That way, your earlier tests are always there in case you need to use them again. You can comment out calls to previous tests so that you can concentrate on one test at a time.

To help you get jumpstarted on testing, here are a handful of example test programs across the various problems:

- A simple test for Alarm
- A simple test for Join
- A simple test for Condition2, and a more complicated test for Condition2
- A simple test for sleepFor
- A simple test for Rendezvous

Keep in mind that Nachos has a number of command line arguments, two of which are particularly useful for debugging this project:

- Invoking `nachos -d t` will display thread-related debugging information, such as context switches and thread state changes. You can add your own debugging output for this flag using `Lib.debug(dbgThread, ...)`.
- Invoking `nachos -s <number>` with different `numbers` will change when context switches happen.

During the project period, you can also use Gradescope to run a snapshot of your code on the sample tests that we have given. See the following Piazza post describing how to use Gradescope to compile and run the sample tests. **Important:** We will be using Gradescope as the grading platform. Before the deadline, you must submit your code to Gradescope at least once to initialize the grading system for your project.

Code Submission

As a final step, create a file named README in the `proj1` directory. The README file should list the members of your group and provide a short description of what code you wrote, how well it worked, how you tested your code, and how each group member contributed to the project. The goal is to make it easier for us to understand what you did as we grade your project in case there is a problem with your code, not to burden you with a lot more work. Do not agonize over wording. It does not have to be poetic, but it should be informative.

We will use a snapshot of your Nachos implementation in your github repository as it exists at the deadline, and grade that version. (Even if you have made changes to your repo after the deadline, that's ok, we will use a snapshot of your code at the deadline.) **Important:** We will be using Gradescope as the grading platform. Before the deadline, you must submit your code to Gradescope at least once. See the instructions in the Testing section above.

Cheating

You can discuss concepts with students in other groups, but do not cheat when implementing your project. Cheating includes copying code from someone else's implementation, or copying code from an implementation found on the Internet. See the main project page for more information.

We will manually check and also run code plagiarism tools on submissions and multiple Internet distributions (if you can find it, so can we).

---

voelker@cs.ucsd.edu