Project 1: Threads

Spring 2018

~~Due: Wednesday, April 25, at 11:59pm~~
Due: Friday, April 27, 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. You can add new classes to your project as you see fit; the code we provide you is a skeleton for the project. As described on the main project page, though, be careful to only add any additional classes to the packages (directories) permitted.

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. 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` (the "parent") joins with thread `A` (the "child"). 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`.

Note that 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.

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.)

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 a parent calls join on a child and the child is still executing, the parent waits; if a parent calls join on a child and the child has finished executing, the parent 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 parent thread can join with multiple child threads in succession; independent pairs of parent/child threads can join with each other without interference.

3. (30%) 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`.

   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. (30%) An online MOBA game company would like your assistance. They would like you to implement a game matching class that groups together players of the same ability into fixed-sized groups to play matches with each other. Your task is to implement the class `GameMatch` using `Lock` and `Condition` to synchronize player threads into groups. A `GameMatch` is created specifying the number of players `N` required for a game match. You may assume that `N` will always be greater than `0`.

   Each player is a separate thread that invokes the `play` method to join a match with a parameter indicating their ability level. The parameter `ability` can have one of three values: `abilityBeginner`, `abilityIntermediate`, and `abilityExpert`. `play` blocks a calling thread until `N` player threads of the *same* `ability` have called `play`. At that point, all threads of that ability

waiting for a match should continue and return from `play` (but not threads of other abilities). Subsequent threads of that ability calling `play` will then wait to form a group for the next match. `play` returns `-1` if it is invoked with an unknown ability.

The return value of `play` is the match number. The first match returned by `play` has match number `1`, and every subsequent match returned by `play` increments the match number by one, independent of ability. No two matches should have the same match number, match numbers should be strictly monotonically increasing, and there should be no gaps between match numbers. Different instances of a `GameMatch` have their own independent match number history.

Tip: Implement `GameMatch` in stages. First implement correct synchronization behavior for one ability, ignoring tracking match numbers. Then extend your implementation to handle all abilities. Then track match numbers.

Testing: Implement tests that verify that `play` only returns when `N` threads of the same ability have called it; `play` can create matches of threads with different abilities; threads do not return from `play` unless they are matched; multiple `GameMatch` instances do not interfere with each other; `play` returns match numbers as specified; `play` returns an error if invoked with an unknown ability.

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 GameMatch

Code Submission

You do not have to do anything special to submit your project. 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.)

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