Phase 1: Build a thread system

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. 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 proj1 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.

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

1. An autograder 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 autograder can use, leaving slightly less room for you to be creative. Your code must strictly follow these interfaces (the documented `*Interface` classes).

In order for you to verify that your code is compatible with the autograder, there are some simple compatibility tests you can run before you submit.

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 source files. You will not be submitting this file (`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. You will not be submitting this file either. 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. 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 we provide.
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.

When you want to add source files to your project, simply add entries to your `Makefile`.

In this project,

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.

Design Reviews:

To make sure that you have a working design before you attempt to implement it, you should meet with a TA to review your design. These meetings will be held around ten days before the due date, and should take around 20-25 minutes. Please bring a copy of your design for the TA to read. In addition, you will turn in your final design document reflecting the actual implementation one day after the phase is due (might not change from the design review document if you design well).

Testing your files:

To submit your code, run the command `test-subm proj1-test` from within the `nachos` directory on the server `klwin00.ucmerced.edu`. It will compile the files and run the appropriate tests while giving you the results of each test. You are expected to create and test the program on your own. We will run the autograder using the same command, `test-subm`, so make sure it works on the server before submitting.

Tasks:

I. (5%, 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. (5%, 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. (10%, 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. (20%, 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). Your solution should work even if there are multiple `speak`ers and `listen`ers 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.

V. (35%, 125 lines) Implement priority scheduling in Nachos by completing the `PriorityScheduler` class. Priority scheduling is a key building block in real-time systems. Note that in order to use your priority scheduler, you will need to change a line in `nachos.conf` that specifies the scheduler class to use. The `ThreadedKernel.scheduler` key is initially equal to `nachos.threads.RoundRobinScheduler`. You need to change this to `nachos.threads.PriorityScheduler` when you're ready to run Nachos with priority scheduling.

Note that all scheduler classes extend the abstract class `nachos.threads.Scheduler`. You must implement the methods `getPriority()`, `getEffectivePriority()`, and `setPriority()`. You may optionally also implement `increasePriority()` and `decreasePriority()` (these are not required). In choosing which thread to dequeue, the scheduler should always choose a thread of the highest effective priority. If multiple threads with the same highest priority are waiting, the scheduler should choose the one that has been waiting in the queue the longest.

An issue with priority scheduling is *priority inversion*. If a high priority thread needs to wait for a low priority thread (for instance, for a lock held by a low priority thread), and another high priority thread is on the ready list, then the high priority thread will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is to have the waiting thread *donate* its priority to the low priority thread while it is holding the lock.

Implement the priority scheduler so that it donates priority, where possible. Be sure to implement `Scheduler.getEffectivePriority()`, which returns the priority of a thread after taking into account all the donations it is receiving.

Note that while solving the priority donation problem, you will find a point where you can easily calculate the effective priority for a thread, but this calculation takes a long time. To receive full credit for the design aspect of this project, you need to speed this up by caching the effective priority and only recalculating a thread's effective priority when it is possible for it to change.

It is important that you do not break the abstraction barriers while doing this part -- the Lock class does not need to be modified. Priority donation should be accomplished by creating a subclass of ThreadQueue that will accomplish priority donation when used with the existing Lock class, and still work correctly when used with the existing Semaphore and Condition classes.

VI. (25%, 150 lines) Now that you have all of these synchronization devices, use them to solve this problem. You will find condition variables to be the most useful synchronization method for this problem.

A number of Hawaiian adults and children are trying to get from Oahu to Molokai. Unfortunately, they have only one boat which can carry maximally two children or one adult (but *not* one child and one adult). The boat can be rowed back to Oahu, but it requires a pilot to do so.

Arrange a solution to transfer everyone from Oahu to Molokai. You may assume that there are at least two children.

The method Boat.begin() should fork off a thread for each child or adult. Your mechanism cannot rely on knowing how many children or adults are present beforehand, although you are free to attempt to determine this among the threads (i.e. you can't pass the values to your threads, but you are free to have each thread increment a shared variable, if you wish).

To show that the trip is properly synchronized, make calls to the appropriate BoatGrader methods every time someone crosses the channel. When a child pilots the boat from Oahu to Molokai, call ChildRowToMolokai. When a child rides as a passenger from Oahu to Molokai, call ChildRideToMolokai. Make sure that when a boat with two people on it crosses, the pilot calls the ...RowTo... method before the passenger calls the ...RideTo... method.

Your solution must have no busy waiting, and it must eventually end. Note that it is not necessary to terminate all the threads -- you can leave them blocked waiting for a condition variable. The threads representing the adults and children cannot have access to the numbers of threads that were created, but you will probably

need to use these number in begin() in order to determine when all the adults and children are across and you can return.

The idea behind this task is to use independent threads to solve a problem. You are to program the logic that a child or an adult would follow if that person were in this situation. For example, it is reasonable to allow a person to see how many children or adults are on the same island they are on. A person could see whether the boat is at their island. A person can know which island they are on. All of this information may be stored with each individual thread or in shared variables. So a counter that holds the number of children on Oahu would be allowed, so long as only threads that represent people on Oahu could access it.

What is not allowed is a thread which executes a "top-down" strategy for the simulation. For example, you may not create threads for children and adults, then have a controller thread simply send commands to them through communicators. The threads must act as if they were individuals.

Information which is not possible in the real world is also not allowed. For example, a child on Molokai cannot magically see all of the people on Oahu. That child may remember the number of people that he or she has seen leaving, but the child may not view people on Oahu as if it were there. (Assume that the people do not have any technology other than a boat!)

You will reach a point in your simulation where the adult and child threads believe that everyone is across on Molokai. At this point, you are allowed to do one-way communication from the threads to begin() in order to inform it that the simulation may be over. It may be possible, however, that your adult and child threads are incorrect. Your simulation must handle this case without requiring explict or implict communication from begin() to the threads.

Design Questions:

Please answer the following questions in your design document (due *one day after the code for this phase is due).*

7. *Why is it fortunate that we did not ask you to implement priority donation for semaphores?*
8. *A student proposes to solve the boats problem by use of a counter, AdultsOnOahu. Since this number isn't known initially, it will be started at zero, and incremented by each adult thread before they do anything else. Is this solution likely to work? Why or why not?*