Project 1: Threads
CSE-150-01

Overview

Stock Nachos implementation has an incomplete thread system. In this project, your job is to complete it, and then use it to solve several 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 this 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. More specifically, the only package you will modify is `nachos.threads`, so do not add any source files to any other package. Before adding any source code, please read **Adding Source Code and Grading**.

Also, 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.

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.)

Design Document and Review

You must make sure that you have a working design before you attempt to implement tasks 1 – 6 listed below (see **Project 1 Initial Design Document** assignment on CatCourses for more details). To make sure you have a working plan, you will be meet with your TA to review your design. These meetings will be held during your lab session around ten days before the due date for the final code and should take around 20–30 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 the day when your final code is due (might not change from the design review document if you design well).

Tasks

1. **(0%)** Complete all steps listed in the file **Task 0** to set up server access for your group.

   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.

2. **(10%)** 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: (a) 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; (b) 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); (c) 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; (d) A thread must finish executing normally whether or not it is joined.

3. **(10%)** 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 (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. 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.

4. **(15%)** Complete the implementation of the `Alarm` class by implementing the `waitUntil(long x)` method. 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`.

5. **(25%)** Implement synchronous send and receive of one-word messages (also known as Ada-style rendezvous), using condition variables (*do not 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 are using more than one lock, you are making things too complicated.

6. **(40%)** 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. It takes in the initial number of adults on Oahu, the initial number of children on Oahu, and a BoatGrader object that we assign to our static BoatGrader object.

   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 can access to the numbers of threads that were created.

   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 the people on Oahu. That child may remember the number of people that they have 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!)

Code Submission and Testing

You may use the `scp` command to upload your code to the course server `klwin00.eng.ucmerced.edu`. You will be submitting your entire `nachos` directory and its contents to the server. Note that you only need to upload the (uncompiled) source code to the server (it will be compiled when autograded so `.class` files need not be submitted). If you have multiple submissions, clearly place your final code in an obviously named directory, like `P1Final`, so the TA knows what to grade. It is your team's responsibility to maintain an organized directory structure in the server and make sure there are no accidental erasures of prior versions of code which may be useful.

Autograder Testing:

At any time, after copying your nachos directory to the server, you may run the command `test-subm proj1-test` from outside the same directory as the submitted nachos directory to test your code with a subset of the test-cases that your code will be put through during final testing. For example, if your nachos directory is placed inside the folder `P1`, you may navigate to (or `cd` into) `P1` and run the `test-subm` command. It will compile the files and run the appropriate tests while giving you the results of each test. We will run the autograder using the same command, `test-subm` (but with a larger set of test-cases), so make sure it works on the server.