EECS 111
System Software
Spring 2016
Project 3: Thread Scheduling

Due Date (May 17th, 2016 11:59 PM)

Instructions

Stock Nachos has a simple thread scheduling system – round robin scheduler. In this assignment, your job is to implement more sophisticated scheduling algorithm – priority scheduler.

The first step is to read and understand the partial thread system and scheduler we have written for you (use the archive file provided along the project). Learn how this scheduler puts the threads in a ready queue and how it selects a thread to be dispatched to the processor.

After installing the Nachos distribution, run the program nachos (in the proj3 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.

Grading

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

1. A grader program 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 grader can use, leaving slightly less room for you to be creative. Your code must strictly follow these interfaces (the documented *Interface classes).
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 non-java source files (useful in later projects). We will be using our own Makefile. (`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. We will also being using our own `nachos.conf` file. 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. Also do not add any classes to these packages. They will not be used during grading.
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 grader 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 non-java source files to your project, simply add entries to your `Makefile`.
In this project, you will **not** need to add any new files.

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.

Submission

In this project, you will modify couple of java files in threads folder. A dropbox folder is provided in EEE website. You need to compress all the files in threads folder into a single archive file (zip, rar, etc.) and upload it to **EECS111 Assignment 3** directory. Make sure the directory name and the zip file name are *`{Student ID}_A3`*. The deadline for uploading the files is the project deadline.

Task

I. (100%, 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. Priority should also be donated through thread joins.

   Priority Donation Implementation Details:

   1) A thread's effective priority is calculated by taking the max of the donor's and the recipient's priority. If thread A with priority 4 donates to thread B with priority 2, then thread B's effective priority is now 4. Note that thread A's priority is also still 4. A thread that donates priority to another thread does not lose any of its own priority. For these reasons, the term "priority inheritance" is in many ways a more appropriate name than the term "priority donation".
   2) Priority donation is transitive. If thread A donates to thread B and then thread B donates to thread C, thread B will be donating its new effective priority (which it received from thread A) to thread C.