Project 2: Thread Scheduler and Synchronization

Task Overview

In this project, we continue to work in the Nachos **threads** directory

(1) In **boat.java**, use Nachos synchronization tools to solve a problem involving shared use of a boat to get people from one island to another. Persons are to be represented by individual threads (of KThread type). A person can be either an adult or a child (two types of threads).

(2) In **PriorityScheduler.java**, implement priority scheduling for Nachos. Your solution must implement priority donation to address the problem of priority inversion.

Submission Requirements

For this project, you will submit to Harvey the Nachos **threads** directory and a project **report**. Your project should, at a minimum, describe your approach in solving each one of the tasks, and how you overcome the difficulties you encountered during debugging. The report should be at least one page, well written and organized.

Important Note

Nachos is an incomplete OS for instructional purpose. As such, elements of Nachos are gradually modified or added by different projects.

In project 0, which is a warm-up project, we run the original codes in a local IDE. You should have traced how the messages are printed and which java file(s) can print them.

In project 1, you are required to complete KThread.java, alarm.java, communicator.java and condition2.java.

In **project 2**, the tasks you work on will be partially based upon your finished code for project 1. Specifically, task 1 will need to use the available synchronization tools from Nachos, which includes lock, semaphore, condition2, etc. Task 2's auto-grader will use the join() method from project 1 to test your PriorityScheduler.java.

Therefore, **you should continue to use the threads folder submitted for project 1 to finish your project 2 tasks.**

Scheduler Use

In project 2, task 1 and 2 need to use different schedulers. Task 1 uses RoundRobinScheduler and task 2 uses PriorityScheduler. Note that the Docker Auto-grader for project 2 already sets the schedulers for you. For auto-grading, you only need to download the Docker image for papama/cs3053 p2, then map your local threads folder to /root/nachos/threads papama/cs3053:p2. However, if you want to write your own test code, which needs to load information from nachos.conf in folder proj2, remember to set the scheduler properly.

Task 1 - Synchronization

Now that you have all the synchronization devices implemented by project 1 and the original Nachos, use them to complete boat.java. You will find condition variables to be the most useful synchronization method for this task:
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. Assume that there are at least two children.

The method Boat.begin() should fork off a thread for each child and 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 cannot 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 numbers 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 situation. 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!)

The one exception to these rules is that you may use the number of people in the total simulation to determine when to terminate. This number must only be used for this purpose.

Task 2 – Priority Scheduling

Implement priority scheduling in Nachos by completing the PriorityScheduler.java. Priority scheduling is a key building block in real-time systems.

Priority in nachos ranges from 0 to 7 (integer values), with 7 being the maximum priority. Note that all scheduler classes extend the abstract class nachos.threads.Scheduler. You must implement the methods:
- nextThread and pickNextThread in the inner class PriorityQueue
- getEffectivePriority, setPriority, waitForAccess and acquire in the inner class ThreadState

In choosing which thread to dequeue from a PriorityQueue-typed queue, your implementation should always choose a thread of the highest effective priority.

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

When implementing your priority scheduler, make sure the priority is donated by the waiting thread wherever possible. The implementation of the getEffectivePriority method in the ThreadState inner class should return the priority of a thread after taking into account all the donations it is receiving.