Project 3 Report: Code Design and Implementation Choices

Group Members: Dylan Monge, Maayan Israel, Brittany Flores, Marvin Sevilla, Carlos Castrillo

Project Overview:

Project #2 aimed to have our team implement the PriorityScheduler Class. The function of this class is to schedule processes based on a priority and perform donation and restoration of priorities to solve the problem of starvation. Starvation occurs when a high priority process cannot run because it needs a resource held by a lower priority process.

Code Design:

In this code, our team is implementing a method called `waitForAccess`, which is part of a scheduling algorithm in a multi-threaded system. First, we ensure that interrupts are disabled using `Lib.assertTrue(Machine.interrupt().disabled())`. Then, we retrieve the `ThreadState` object associated with the thread passed as an argument. We add this thread to a priority queue (`pq`). Next, we enter a while loop where we compare the effective priorities of the current owner thread (`owner`) and the waiting thread (`threadWaiting`). If the owner's effective priority is lower than the waiting thread's effective priority and `transferPriority` is enabled, we update the owner's effective priority to the maximum of its current priority and the waiting thread's priority. We also add the waiting thread to the owner's list of threads that have donated priority. If the owner is waiting for another thread (`owner.waitingFor`), we update its effective priority recursively. This loop continues until the owner has a higher or equal effective priority to the waiting thread, or until the owner is not waiting for another thread. Finally, we call `waitForAccess` recursively on the waiting thread to continue the priority donation chain.

our team had to implement the `nextThread` method. We ensure interrupts are disabled using `Lib.assertTrue(Machine.interrupt().disabled())`. We then handle priority restoration, a crucial aspect of our scheduling mechanism. If the current owner thread has threads that have donated priority (`owner.listOfThreadsThatHaveDonated`), we remove the highest priority donor from this list and update the owner's effective priority accordingly. If no threads have donated priority, we revert the owner's effective priority to its original priority. Finally, we return the next thread to be scheduled by dequeuing it from the priority queue (`pq`). This design efficiently manages thread priorities and ensures that threads with higher effective priorities are scheduled first, facilitating a fair and effective scheduling strategy.

Implementation Choices:

In our implementation of the `waitForAccess` method, our team opted to use a priority queue (`pq`) to efficiently manage thread priorities. We chose a while loop to dynamically adjust priorities based on thread interactions, ensuring fairness in scheduling. Priority donation is implemented by updating the effective priority of the owner thread (`owner`) when a waiting thread has a higher priority. Recursion is utilized to propagate priority donation through the thread dependency chain, ensuring that threads with higher effective priorities are scheduled first.

Similarly, in the `nextThread` method, we utilize a priority queue to efficiently select the next thread for execution based on their effective priorities. We ensure interrupts are disabled before proceeding. Priority restoration is handled here, where if the current owner thread has received priority donations, we update its effective priority accordingly. If no priority donations have occurred, we revert the owner's effective priority to its original value. This design choice enables our scheduling algorithm to prioritize threads effectively, ensuring fairness and efficiency in thread execution.

Conclusion:

In conclusion, we found that the implementations were pretty. We struggled to implement the traversing of the queues. We also struggled to implement the reverse traversing involved in restoration. Ultimately, our code encountered a runtime error while running our program.