Jacob Tilmon
C00292879

CMPS 455 Project 1: Semaphores

Task 1: Dining Philosophers

For this task, I implemented the algorithm as was laid out in the Project 1 handout. Each “philosopher” was its own thread which ran the algorithm in its run() function. I also created an ArrayList which held all the chopsticks for the table. Each philosopher was given the reference to their specific left and right chopstick.

One of the major issues I ran into for this problem was the deadlocking that could occur if each philosopher only picked up their left chopsticks. In order to fix this, I ran a tryAcquire() instead of a standard acquire(). This way the philosopher would only try to pick up a chopstick when available. Additionally, it allowed me to create condidtion for a philosopher to “put down” their left chopstick if they could not pick up their right chopstick. This way deadlock was avoided.

Task 1 Question 1

P: 10 M: 50 Seed: 1024	P: 5 M: 10 Seed: 985	P: 25 M: 100 Seed: 2006
4049.136 ms	50.831 ms	80.648 ms

One interesting thing to note is that tests with an odd number of philosophers run much faster than ones with an even number of philosophers. I’m not entirely sure what causes this, however output shows that there was a lot of conflict trying to grab chopsticks. I think an even number makes it more likely that philosophers grab each other’s chopsticks.

Task 1 Question 2

P: 10 M: 50 Seed: 1024	P: 5 M: 10 Seed: 985	P: 25 M: 100 Seed: 2006
43.134 ms	36.369 ms	3063.81 ms

Interestingly, adding a wait inbetween grabbing either chopstick dramatically sped up the first case while dramatically slowing down the last case. The first task was sped up as more philosophers had to put down chopsticks as they were unable to eat without both but im not sure why the third case sped up. Perhaps it was overwhelming to my laptop’s processor but I’m not entirely sure.

Task 2: Post Office Simulation

For this task I implemented the algorithm as listed in the handout in each MailPerson’s run() method. Rather than use a two-dimensional array as suggested, I created my own object called “Mailbox”. Each Mailbox has methods protected by semaphores to add and remove mail from the mailbox. The contents

of the mailboxes is represented by a Stack data structure. The reason I chose a Stack was due to my difficulty to get correct indexing with an ArrayList. Additionally, It does not matter in what order the MailPerson recieves their mail as they will continue to receive mail until their mailbox is empty.

Task 2 Question 1

There wasn't as much deadlock for me in task 2 as there was in task 1 as by this point I had a decent groove. I believe me using Mailbox objects defenetly helped me avoid major frustration as I was able to focus on fixing individual methods instead of one whole run() method. It was different in the beginning in that some threads would just stop as they were trying to put mail in full mailboxes that wouldn't get read, that problem was resolved when I started using a Stack for the actual messages.

Task 3: Readers-Writers

This task was defenetly the hardest for me. I initially re-created the standard Readers-Writers problem without trying to follow the pattern of N readers, 1 writer, repeat. From here I struggled greatly trying to get the pattern of Readers-Writers correct. Eventualy I figured out that I could treat it a bit like a producer consumer problem. I implemented semaphores that had N for readers and 1 for the writer, and ony readers could release the writer semaphore and vice versa.

Task 3 Question 1

For this question there was a lot more conditions involving semaphores. It was defenetly tricky managing many conditions such as when there were no more writers, no more readers, etc.

Task 3 Question 2

If N becomes larger, more readers will go through before a single writer would. This would more than likely lead to writers being starved out until all readers are finished. After that point, each writer would have to finish one my one which would take a greater amount of time than smaller groups of readers.