Daniel Metrejean
C00402116
February 22, 2022
CMPS 455 – Kumar
Project 1: Synchronization

1 Dining Philosophers

1. (and 2 I guess)

Inputs (meals, philosophers)	**Time (no `yield()`)**	**Time (`yield()`)**
(5, 5)	0.7378 seconds	1.5516 seconds
(50, 50)	1.8090 seconds	2.9089 seconds
(500, 500)	2.1280 seconds	3.1525 seconds

Obviously, the runtime increases as the number of threads increases. I found in my tests (of a very small sample, only three executions) that the growth of time is *less than linear*. That is, it seems to take less time proportionately when the inputs are much larger. In addition, when yield() is included between picking up the chopsticks the time increases; in two cases out of three it actually almost doubles! This is simply because the active thread will be put back in the ready queue and overall there will be more downtime. I feel like with this limited test, there is little that we can say about the performance.

2 Post Office Simulation

1. If anything, I actually had more issues with deadlock in problem 1, when all the philosophers would pick up the left chopstick at the same time and then none of them could pick up the right one. Still, philosophers is a much simpler algorithm to implement. They will only be going through a list of tasks and the programmer need only coordinate how they will do so. In the Post Office Simulation, there is added complexity as the threads actually communicate with each other, sending messages back and forth vis a vis a mailbox array.

3 Readers-Writers Problem

1. I found this one to be the hardest. Although the readers' and writers' algorithms themselves were very simple, I had trouble in getting them to go in the right order (where N readers -> 1 writer -> N readers…). I ended up needed some internal logic inside of the readers and writers threads that would determine whether or not a reader or writer was already in area.
2. I found that my code behaves a little unpredictably when N is large. I suspect it has to do with all N of the readers trying to update readcount at the same time. The writers end up

all going at the end of the execution. To be quite honest, with big input parameters it is very difficult to even know what is happening thanks to JVM's scheduler. It seems that my outputs are all out of order, but the execution completes without problems.

Report Questions

1.
    a. For the **dining philosophers**, I used semaphores and barriers, both of which fall under the same Java class Semaphore. To start, for all the tasks, once command line input has been received, main will call the proper function for the specified problem. I have main(args) and DiningPhilosophers() which is called from main in the switch statement. In addition, I also have the class Philosopher which extends Thread. DiningPhilosophers() is where P and M inputs are gotten from the user and the threads are actually instantiated. From there, the threads will be instructed to start() and all of the remaining logic is in the overridden run() function within the Philosopher class. This involves checking for the sitBarrier (keeps any from getting greedy before everyone has sat down) picking up and putting down (acquire() and release()) the chopsticks and decrementing the number of meals remaining. There is also a leaveBarrier so that they don't leave while somebody is still eating.
    b. For the **post office simulation**, there are a couple of semaphores as well as String arrays. I created PostOfficePerson which again extends Thread. After calling PostOfficeSimulation() in main and getting input, a 2D String[][] array is created to house the messages that will be sent. PostOfficePersons are created as well and start(). The people will first do a while loop: while there are unread messages, read them and set the mailbox content to be empty, freeing up space for a new one. Next, if messagesLeft has not been decremented to 0, they will choose a random person recipient and a random message and put it in their mailbox, decrementing messagesLeft as they go. Lastly, even if messagesLeft is 0, there still may be unread messages, so I have a small block of code at the bottom of run() that checks every mailbox slot and sets a Boolean. Then I encapsulated the whole thing in another while loop which checks for this Boolean value, meaning, even if messagesLeft is 0, people will still be able to continue entering and checking their mail, just not sending messages.
    c. This one actually took me the longest, which may be surprising (idk it seems simple to me theoretically). For this one, I have two classes Reader extends Thread and Writer extends Thread. That way I could just have their own run() functions and not worry about `if type == reader {code} else if type == writer {code}.` So in main I call ReadersWriters() and get input for R, W and N. I want N readers to go first, so I instantiate R readers, W writers *and* a barrier. After calling start() on all of them, I release the barrier which is the first condition in the readers' run. As we know from class, next they increment readcount behind a protective semaphore and "do some reading". After finishing, if readcount == 0, meaning they are the last of N readers, it will release another barrier which is the first condition in Writer. It goes back and forth like this, writer releasing the barrier after one writer, and reader releasing barrierW after N

readers until there are no more of either. Then the rest of whichever will continue and finish their queue.