CMPS 455 Project 1

Victoria Huval C00480584
Chante Wiltz C00037332

Task 1.

1. Test trial with random parameters

Trial	Philosopher	Meals	Time (MS)
1	5	20	2764.986584
2	8	8	637.388834
3	5	10	1490.219791
4	10	5	440.654125
5	16	16	796.733125
6	64	64	853.418083

I found that philosophers with more meals will take longer in time due to longer wait times in chopstick holding. I also noticed that with more philosophers and less meals the time will go by faster due to less wait time on chopstick holding. Larger numbers in both philosophers and meals will result in longer wait times.

Thread.yield() parameters

Trial	Philosopher	Meals	Time (MS) with .yield()
1	5	20	2829.928
2	8	8	847.288208
3	5	10	1454.264625
4	10	5	492.47025
5	16	16	794.6135
6	64	64	904.501041

2. I noticed that the results between the two tables were about the same in terms of time. Some of the trials were slower (Trials 1,2,4,6) in time while some ran faster (trials 3,5) by a little bit. Due to the placement of Thread.yield() being directly in the middle of when a philosopher picks up a left chopstick, yielding, then picking up the right one, this gives a moment for the philosopher to

pause while acquiring chopsticks. This may be the reasoning as to why some trials are slower in speed than others.

Trial	Philosopher	Meals	Time (MS) with .yield()
1	1	4	1546.872667
2	8	4	1995.895958
3	16	4	1952.383167
4	64	4	1951.032417

3. This modified version of code represents maximum parallelism because none of the threads have to wait on another thread. The modified code allows the philosophers to be able to have their own chopsticks so that they don't have to use any other philosophers chopsticks. Being able to have the meals remaining counter set to each private philosopher so that no thread has to read or write over another thread's meal counter. Removing the barriers that were in place allowed for each philosopher to eat immediately and to leave immediately after finishing eating without being dependent on another philosopher.

Task 2.

1. In the Reader-Writer problem I was focused on a single shared resource rather than multiple shared resources. Unlike the Dining Philosophers and their chopsticks, I was allowed to implement the instance of multiple readers sharing the resource at a time. To prevent a deadlock situation, I used a 'master' semaphore (coordTurn) paired with a simple logic statement (handleReaders variable) within the reader threads' join for loop. This strategy allowed for the access pattern of N reading agents then 1 coordinating agent along with the 'fizzling out' of the reader threads once all the writers had accessed the resource.

2. When N is very large, there becomes an increased risk of deadlock. I have seen that this is the case because there will most likely be a situation where there are not enough readers to make a full round (N readers = 1 round). This was an issue for me especially. Upon reaching the last round, the system would be unable to reach the join() function.

Task 4.

Dining Philosophers Problem

For the dining philosopher problem, I started by initializing each semaphores: A semaphore array chops to represent the chopsticks for the philosophers, arriveCount for the arrival count of philosophers entering, arrivalDone for the final arrival for all P philosophers that are at the table, finishedCount for counting the philosophers that are finished eating, releaseAll for releasing all of the philosophers from the table, printMutex for print formatting the philosophers ID, totalMeals for the total amount of meals(used to increment when philosopher is eating), numPhilosophers for the number of philosophers, and numMeals for the number of meals. Then I created a static class Philosopher that extended Thread. Inside this class, I created an ID tracker as well as mealsRemaining counter that decreased as each meal was being eaten. A constructor for the philosopher id. A random that is used for a methods randCycle() and simulateCycles() which is used to generate random 3 to 6 cycles of eating. I also made a left() and right() function to determine if a chopstick is a left chopstick or a right one.

Within the run() function, I ran a while loop for mealsRemaining to be greater than 0, signifying when there are no more meals left for the philosophers to eat. In the loop, the logic I wrote counted starts with the philosophers arriving (arrivalCount.release()), for the philosophers to wait until all of them arrive (arrivalDone.acquire()), and for all of the philosophers to arrive finally (arrivalDone.release()). For the case if there were one philosopher for the chopsticks to be acquired, simulated each cycle, the totalMeals counter increments, mealsRemaining decrements, and release chopsticks until all of the remaining philosophers are done eating. I wrote the logic to check whether each Philosopher has a left or right chopstick in order for the left chopstick to be picked up first while waiting for their opportunity for the right one to be available. Acquire while picking up chopstick and releasing while dropping chopstick. Within the main() function, i wrote new semaphores for the arrival and departure barriers as well as new chopsticks assigned to each of the philosophers. A for loop made to assign each ID to each specific philosopher, an arrival loop to count the philosophers that have arrived at a given index, and a leave loop to count the philosophers that have left the table at a given index.

I have noticed during my code that the only bugs that I have encountered were my meals. I noticed that my meal count would multiply the number of philosophers and number of meals instead of the provided number of meals that I have put in. I fixed this by taking the total number of meals in the ID constructor for the mealsRemaining, and divided it among each philosopher so that it would print out the appropriate amount of meals that were put in. Overall, the program ran very smoothly.

Readers-Writer Problem

For this problem, I began by initializing my core variables and semaphores: completedReaders, totalReaderDone, readerSpots, coordTurn, mutex, and printMutex. These served as the backbone for tracking available reader slots, the number of readers accessing the resource, when the writer could access the resource, and orderly printing of each thread’s actions to the console. From there, I created two static classes that extended Thread: Reader and Writer. In each class, I assigned a specific ID to each thread, with a ‘R’ for readers and a ‘W’ for writers. After assigning IDs, I defined a “work cycle” function for each thread using Thread.sleep() to simulate reading or writing. Each class also had a mutexLog function responsible for safely displaying each thread's actions. Before implementing the process, I wrote the main function to collect user input for the number of coordinator agents (W), the number of reader agents (R), and the maximum number of simultaneous readers (N). I used these inputs to create threads and store them in two arrays for easier access. The most important parts of the main function were the handleReaders and handleWriters variables, which contained logic used to prevent deadlocks.

The main logic was handled in the run() functions of the Reader and Writer classes. For the Reader class, each thread acquired a reader slot with readerSpots.acquire(), simulated reading with readCycle(), updated the shared counter of completed readers, and released the slot. I then checked if all N readers were finished; if so, the coordTurn.release() semaphore signaled a writer to acquire the resource. For the Writer class, each thread acquired the writer slot with coordTurn.acquire(), simulated writing with cycleWrite(), and then signaled the readers that it was their turn by releasing N readerSpots in a loop.