CMPS 455

Dr. Ashok Kumar

Carl Ferré

Nov 22, 2022

Project I: Overhaul

I, Carl Ferré, certify that this assignment has been done completely by myself.

Introduction

Leveraging more efficiency from computer programs has become all the more important as modern processors hold an increased number of cores. Three canonical concurrency problems were modeled for the purpose of learning about synchronization tools, specifically semaphores. Semaphores were used to conduct flow-control over threads, live program processes that execute in multitudes of instances, unless checked. Analysis and workflow notes regarding the Dining Philosophers problem, Post Office Simulation (Producer-Consumer problem), and Readers-Writers problem are detailed below.

Task I: Dining Philosophers

Task 1 was implemented by first studying the Dining Philosophers problem and understanding the constraints and assumptions. From there, I modeled the classes to reflect the data that would need to be passed between various objects as they spawned from the main program's control flow. I did my best to test modular pieces of code after not having programmed with Java for nearly 6 months. I found myself making progress with the implementation trying to observe best practices for constructors since I was not confident my threads were behaving as expected against the "global" behavior of my atomic integers. This made me curious to learn more about creational design patterns, like the Factory pattern. Once I was able to test the attemptEating function it was a lot easier to see how each thread behaved, independently of one another—yet the

debugger only allowed me to follow one of the threads, so I often had to conceptualize multiple concurrency scenarios.

1. Note the runtime in milliseconds for at least 3 different sets of parameters. Each time, vary the following: the random seed, number of philosophers, and number of meals. Record your results in a table, and include the parameters that you used each time. Explain your findings.

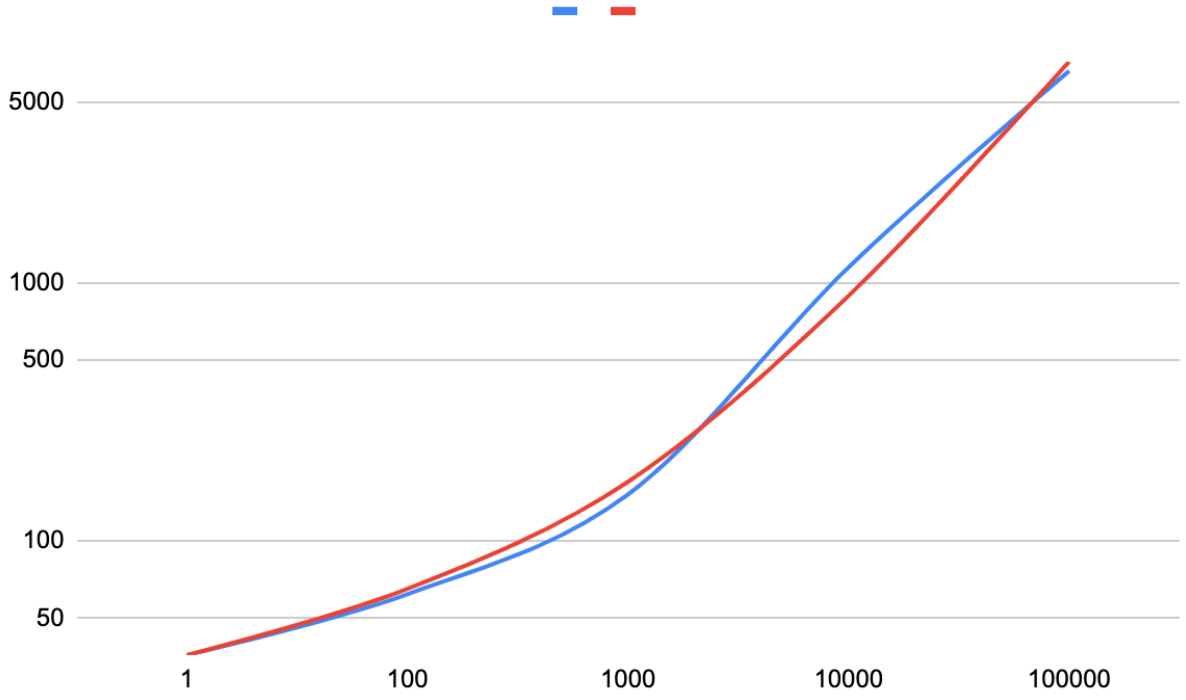
Random Seed	**# Philosophers**	**# Meals**	**Runtime**
ThreadLocalRandom.current().nextInt(3, 7);	2	1	36ms
ThreadLocalRandom.current().nextInt(3, 7);	2	100	62ms
ThreadLocalRandom.current().nextInt(3, 7);	2	1000	151ms
ThreadLocalRandom.current().nextInt(3, 7);	2	10000	1148ms
ThreadLocalRandom.current().nextInt(3, 7);	2	100000	6629ms

Random Seed	# Philosophers	# Meals	Runtime
ThreadLocalRandom.current().nextInt(3, 7);	2	1	36ms
ThreadLocalRandom.current().nextInt(3, 7);	4	100	65ms
ThreadLocalRandom.current().nextInt(3, 7);	8	1000	169ms
ThreadLocalRandom.current().nextInt(3, 7);	16	10000	890ms
ThreadLocalRandom.current().nextInt(3, 7);	32	100000	7198ms

Keeping the number of philosopher threads constant while increasing the order of magnitude of meals leads to an exponential resource depletion time. This is because two philosophers (two threads) are frequently competing amongst one another to avoid deadlock in order to access chopsticks, the shared resource. Interestingly, doubling the number of philosopher threads while the number of meals increases in magnitude, produces steadily matched runtimes.

2. Make the philosophers yield between attempting to pick up the left and right chopsticks. Run the same tests you ran in the previous question, and record the results. Were the results the same or different? Why? Undo any changes made to accommodate this question before submitting your assignment.

Adding a randomized yield time of 3-6 cycles between left and right chopstick acquisition added a linear padding of milliseconds less than the order of magnitude of the runtime. In other words, the results were close to the same though marginally increased. This is because yield() is a suggestion to the CPU scheduler to level the relative progress made between threads that would otherwise over-utilize the CPU.



Task II: Post Office Simulation

1. Did you experience any deadlock when testing this task? How was it different from Task 1?

In the post office simulation, each mailer in the directory holds an inbox with a given message capacity (number of slots) represented by the number of available semaphore permits. As this is a model of the producer-consumer problem, deadlock would occur if a mailer were successful “reading” a message when the inbox was empty, thus introducing an incorrect semaphore permit count. In my case, I always checked the inbox size in order to iterate through messages as permits are acquired, so I did not encounter deadlock issues.

Task III: Readers Writers Problem

1. *How did this task differ from the previous synchronization tasks?*

   In this task, a single Thread class was used to model both reader and writer threads in order to avoid Java thread ownership issues. I found that to be the most difficult aspect of testing my logic at first since I had originally designed for a solution that used Reader Thread and Writer Thread classes. By instantiating each thread with a type, I was able to include the logic for both types in the same run() override method. This allowed each thread an opportunity to vie for access to the file, while still ensuring they behaved according to their respective laws as readers and writers. This problem also required the use of two Semaphores, one for readers and one for writers. Each functioned to control the amount of readers and then writers that would be permitted to serially perform their duty in the file without breaking the validity of the data in the file.

2. *What kind of problems do you see when N is very large (i.e. high priority is given to readers)?*

    When a high priority is given to readers, this leads to a writer starvation problem. Hence, serializing the number of readers that are permitted access to the file before allowing writers is critical to ensuring no processes are left hanging, i.e. waiting with virtually no access to a resource.

Task V: Report Summary

1. *In your own words, explain how you implemented each task. Did you encounter any bugs? If so, how did you fix them? If you failed to complete any tasks, list them here and briefly explain why.*

    During this re-implementation I was much more confident in my employment of semaphores. Completing readers-writers and the post office simulation was a much easier task not only from a solution design standpoint, but also in terms of working with Java. The longest issue I spent trying to resolve was using two different Thread classes for readers-writers.

    For each of the three problems I used a parent container to pass user input to smaller class modules that would be instantiated in the specified number. This trickle down of data provided each class with data unique to that instance, such as an id, and allowed them access to global variables held by the parent container. I whiteboarded solutions with an understanding of what data structures would be used in advance and made sure to rethink or change them as necessary once coding.

Dining Philosophers required an understanding of how to prevent deadlock. For this I switched the last philosopher's left and right chopstick assignments such that the last two philosophers would not wait for a chopstick that would never be dropped.

The Post Office simulation provided I thought through how I would model the varying number of letters in each person's mailbox slots. Deciphering how to use classes for each of the moving parts of the requirement meant taking an object-oriented design approach.

Readers-Writers was trickier for me because I initially wanted to use a single lock to control the passage of control from readers to writers, and readers again. When I had difficulty testing the logic for this it quickly became difficult to test since I had multiple locks around multiple counter variables. Ultimately I realized it was possible to solve this using only one large while loop for both thread types that observed just two respective semaphores.

2. *What data structures and algorithms did you use for each task?*

Dining Philosopher: Semaphores, and arrays specifically, in order to initialize chopstick assignment the way that I had done. I would potentially refactor this to use simpler logic by giving each philosopher a left and right chopstick assignment based on the total number rather than ever utilizing the array.

Post Office: The post office simulation was fun because I got to use a stack to contain a map of <integer, string> detailing the sender and message. In order to avoid using both an inbox message and outgoing message map entry variable I think next

time I would create a message class that would ubiquitously contain a senderId, recipientId, and string message.

ReadersWriters: I resisted the urge to have an actual string/buffer/file to read/write to in order to concretize this problem example. Instead, I used count variables to demonstrate access granted/revoked from threads as they carried out their operations. In this problem I also learned that calling Thread.yield() was not surefire to avoiding deadlock, but I was not able to determine why the sleeping thread would not be awoken by its counterpart performing the opposite read/write operation thusly calling a semaphore release(). I believe this is because Thread.notify() is required to signal the thread to proceed from a yield state.