Introduction to Java

CS9053

Thursday 6 PM – 8:30 PM

Prof. Dean Christakos

July 21st, 2023

Due: July 28th, 2023 11:59 PM

**Part I: Threading Number of Primes**

In the class CountPrimes, there are two static methods, isPrime(long val) and numPrimes(long from, long to). isPrime takes a value and returns is the number is prime. numPrimes takes a minimum and a maximum value and returns the number of primes within that interval.

Try it out by executing CountPrimes.numPrimes (0, 1\_000\_000). Then try CountPrimes.numPrimes (0, 10\_000\_000) and note that it takes longer.

This can run faster if you divide this process into pieces.

In the class ThreadedCountPrimes.java, write in the run() method something which will get the number of primes– implement the class however you choose – and in the main method, generate the threads, then combined the results of each thread to get the value you are looking for.

Compare the speed of the multithreaded division of labor with simply running CountPrimes.numPrimes (10\_000\_000, 20\_000\_000). See if the interval affects the result.

What is the best interval you get for dividing up the calculation of primes? Do you divide the count up by 10,000 or 1,000 or 100,000? How do the results differ?

1 Point extra credit if you make any additional improvements.

**Part II: Understanding Synchronization**

This assignment is a little different. Most of the code is already written. You have to understand it. We have three queues. A general purpose queue, and Integer Queue, and a String Queue.

At the start, you have a Producer, which randomly picks whether to put an Integer or a String on the general purpose queue. If it chooses to put an Integer on the queue, it picks a random Integer. If it chooses to put a String on the queue, it chooses a random String from the file in “data/words”. It runs very fast and stops putting data on the queue if there are 100 objects in the queue already but will start adding them again if the length of the queue falls beneath 100. The Producer puts data on the general purpose queue very fast.

The general purpose queue has four “middlemen” consumers: two Integer MiddleMan objects and two String MiddleMan objects, each of which polls the general purpose queue to see if their data type is available on the queue, and if it is, removes it from the queue and places it on their respective queue. However, it only places it on their respective queue if there are less than 10 objects in the queue already but will start adding them again if the length of the queue falls beneath 10. The MiddleMan objects get data off the General Purpose Queue and put it on their outgoings queues very fast, but not as fast as the Producer Object.

At the end we have two Consumer objects: one for the Integer queue and one for the String queue. Consumer objects will take two items off the queue and, if those items are numbers, add them together, and if those items are Strings (or another non-number), concatenate the String representations of those objects together. The Consumer objects consume very slowly

Consumer 2

Producer

Consumer 1

Integer queue

String queue

String

Middleman 2

String

Middleman 1

Integer

Middleman 2

Integer

Middleman 1

General purpose queue

This is all put together in the class MonitorQueues which connects everything together, starts the Producer, Consumer, and MiddleMan threads, and then monitors the queue sizes for any errors. Note that the GeneralPurposeQueue is a LinkedList while the Integer Queue and String Queue are ConcurrentLinkedQueue objects.

**Question 1:** Go to MiddleMan.java, line 24. You can see I have commented out the synchronized(in) block. The code has an if statement on line 25 that checks if there is an object available on the queue and, if so, if the type of object on the queue is of the correct class for the MiddleMan, using the isInstance method, implemented in the StringMiddleMan and IntegerMiddleMan subclasses.

In MonitorQueues.java, Run the application. The first thing you will see will be a NoSuchElementException or a NullPointerException (or both!). The compound conditional:

**if** ((in.peek() != **null**) && (isInstance(in.peek().getClass())))

Should return false immediately if in.peek() is null, and in.remove() should work fine as long as in.peek() shows there is something available on the queue.

**In the absence of synchronization, why does this happen? What sequence of events is occurring that causes these errors to be raised?**

**This happens because multiple threads are trying to access the same resource at the same time, similar to a deadlock situation, it’s called a race condition. This can lead to an object being null, NullPointerException, as one thread can remove the object while another isn’t aware that there is nothing there and attempts to grab the object. It can also lead to NoSuchElementException, as one thread can remove an object while the next object would be of a different type than what was expected from another thread.**

Ok, now uncomment the synchronized(in) block before proceeding to question 2.

**Question 2:** Stop the program if it’s still running and restart it. What you will see is the output of the two consumers, which is either the addition of two numbers of the concatenation of two strings. However, every so often an alter comes up that either the Integer Queue or the String Queue is too long. However, the alert that the General Purpose Queue is too long never comes up. If we go to Producer.java, line 42, we see the place where it puts a new object on the queue only if the size is less than the maximum queue size of 100, and the block isn’t synchronized.

**Even though the block isn’t synchronized, and even though the queue is a regular LinkedList that doesn’t support concurrency protection, why is there never a situation where the MAX\_QUEUE\_SIZE constraint is violated?**

**This happens because in MiddleMan, in the synchronized block, we are locking the generalPurposeQueue if there is no object to offer to the integer or string queue(s), this will grab an object from the generalPurposeQueue and nothing else will be done to that queue until that object is grabbed so none will be added to it as well.**

Now that this is done, you can choose to synchronize that block or not. It doesn’t matter too much, as you can see.

**Question 3:** We still get those messages “Alert. Queue [1 or 2] > 10. Shouldn’t Happen,” referring to the IntegerQueue and the StringQueue, however. We need to fix that. The MiddleMan objects put data on those Queues in the MiddleMan.run() method, once we have an outObj. Consumer objects get data off of those Queues in the code in Consumer.java in lines 26-44.

**Write code to fix these overflows. You may NOT change the DELAY timings to do so, which influences the rate at which data is produced or consumed on the queues. Explain why your modifications work.**

**I added 3 lines of code to MiddleMan, they are lines 43, 45, and 54. They are also commented with //added. These work because it will lock the integer or string queue(s) with the excess amount of objects and remove the last one.**