CS9053, Tuesday Section
April 21, 2020
Due April 28, 2020, 11:55 PM
Prof. Dean Christakos

Part I: Threading Count of Divisors

In the class CountDivisors, there are two static methods, countDivisors(long val) and maxDivisors(long from, long to). countDivisors takes a value and returns the number of divisors the value has. maxDivisors takes a minimum and a maximum value and returns the largest value with the largest number of divisors.

Try it out by executing CountDivisors.maxDivisors(0,1000). Then try CountDivisors.maxDivisors(0,100_000) and note that it takes longer.

If you try CountDivisors.maxDivisors(100_000, 200_000) it takes quite a while.

This can run faster if you divide this process pieces.

In the class ThreadedCountDivisors.java, write in the run() method something which will get the value with the largest number of divisors and that number of divisors– implement the class however you choose – and in the main method, generate the threads, then find the results of each thread that has the value you are looking for. (ie, the number that has the largest number of divisors).

Compare the speed of the multithreaded division of labor with simply running maxDivisors(100_000, 200_000). See if the interval affects the result.

There are two ways of doing this. One will get full credit, but the faster method will give you 1 point extra credit.

Note: You should return the largest number with the largest number of divisors. If two numbers have the same number of divisors, pick the larger. The output should be of the format:
```
<Number with largest number of divisors>=<number of divisors>
```
Eg: 10=2

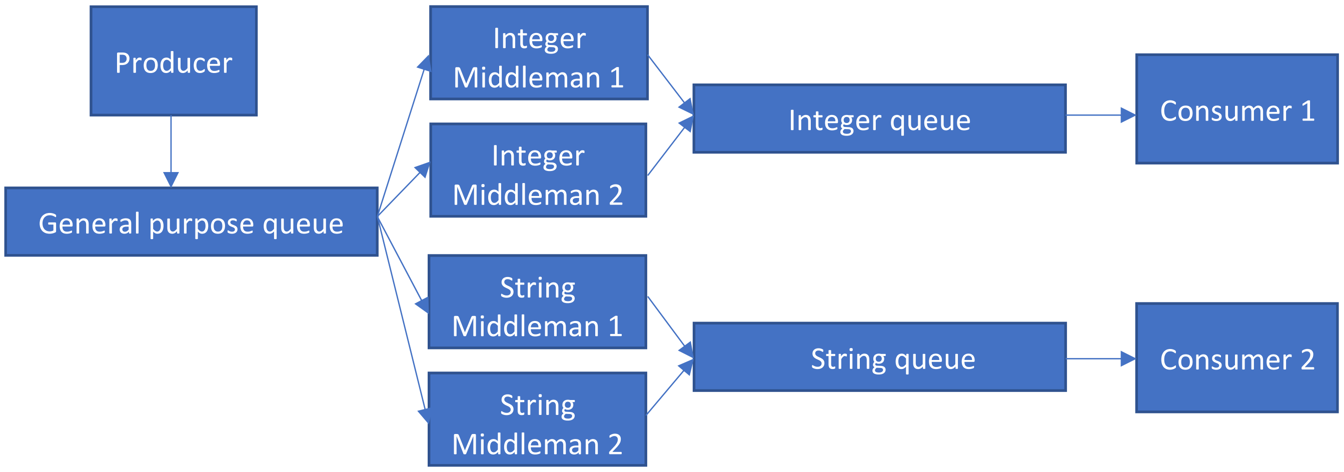
Part II: Understanding Synchronization

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



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?

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?

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.