Grant Slape

Object Oriented Programming and Design

Programming Assignment 2

11/20/17

This was a fun problem, and my first project dealing with concurrency. The initial queue is processed by the initial checker which places a passenger in queue A or queue B randomly. Checker A and Checker B both process to Queue C, and Checker C processes to the completed queue.

My initial thoughts on this problem are that this is clearly a problem that is solvable with concurrent threads for each Checker. The most important thing when dealing with concurrency is thread safety. I did some research on thread-safe data structures in the Collections Framework, and it seemed like LinkedBlockingQueues were the best tool for my task. I wrapped LinkedBlockingQueue<Person> with my own PersonQueue. We need to send a “busy” signal if Queue C is already busy with the other queue, and we do want the queued-up operation to perform.

I also wanted the simulation to be configurable, mainly because I don’t have the time to sit and watch each checker take up to fifteen minutes to process each person ☺. I abstracted the time concept to “ticks” and allow the user to specify passenger count, maximum ticks for all passengers to be queued, and the value of a tick in milliseconds. This allows me much more configurability when testing, and allows me to easily test for race conditions and other concurrency issues by setting a high passenger count and a tick of 1ms. The largest, (and most compact) set that I have tried is 1,000,000 passengers distributed across 1,000,000 ticks with a tick value of 1ms. This took about 3832 seconds primarily because my expedition algorithm only looks at the previous length and sees if it has increased. If it has, we reduce processing time by 1 tick. If it is decreasing we adjust by 1 tick back towards the base line processing time.

After I designed my architecture and started implementing everything, I realized that I needed an initial checker to process the initial passenger pool. However, this Checker follows completely different rules then Checkers A, B, and C. Thus, I needed a Checker interface, and needed an initial checker that will take a sorted set of timestamps to process at instead of sleeping for random ticks.

The largest problem I had was using the right syntax for implementing interfaces, and extending existing classes. I am a little unfamiliar with inheritance in Java and learned through some trial and error. I honestly didn’t have any problems with null pointers or other common concurrency issues.

From here to improve this project I would remove the datetimes from the base person object and extend the person object so that there is an analytical person object that does contains the timestamps. Then we can set up an Observer/Observable pattern and the Checkers can notify their observers that they have processed a person. The observer can then find the person and update their data. This way if something breaks during execution we don’t lose all the analytical data, and it removes strictly unnecessary data from the Person object that is processed by the Checkers.

Some final notes: I’ve provided my first name and last name files. They are stored as .txt files with a new line separating each distinct name, however for the sake of paper and readability I have compressed this into five columns to turn in. Also, it is interesting to examine the execution times as the different variables are changed. When the tick value is changed, the execution times changes linearly as expected. This also occurs with the passenger count, given reasonable other parameters. The interesting part is that changing the initial distribution range of passenger seems to cause execution time to increase logarithmically. After a certain point, the execution time is determined by the distribution time.

TEST RUNS:

1000000 passengers, 1000000 ticks distribution, 1ms tick

START: 2017/11/21 08:16:45.491

END: 2017/11/21 09:20:37.639

TIME ELAPSED: 3832 seconds

1000000 passengers, 2000000 ticks distribution, 1ms tick

START: 2017/11/21 10:40:57.607

END: 2017/11/21 12:24:09.346

TIME ELAPSED: 6191 seconds

100000 passengers, 100000 ticks distribution, 1ms tick

START: 2017/11/21 12:53:31.159

END: 2017/11/21 13:01:33.425

TIME ELAPSED: 482 seconds

100000 passengers, 200000 ticks distribution, 1ms tick

START: 2017/11/21 13:11:47.373

END: 2017/11/21 13:22:19.198

TIME ELAPSED: 631 seconds

100000 passengers, 300000 ticks distribution, 1ms tick

START: 2017/11/21 14:23:40.333

END: 2017/11/21 14:36:33.289

TIME ELAPSED: 772 seconds

10000 passengers, 10000 ticks distribution, 1ms tick

START: 2017/11/21 16:33:01.166

END: 2017/11/21 16:33:45.021

TIME ELAPSED: 43 seconds

10000 passengers, 20000 ticks distribution, 1ms tick

START: 2017/11/21 16:36:06.974

END: 2017/11/21 16:37:13.642

TIME ELAPSED: 66 seconds

10000 passengers, 30000 ticks distribution, 1ms tick

START: 2017/11/21 16:37:41.968

END: 2017/11/21 16:38:53.682

TIME ELAPSED: 71 seconds

10000 passengers, 40000 ticks distribution, 1ms tick

START: 2017/11/21 16:41:20.677

END: 2017/11/21 16:42:30.580

TIME ELAPSED: 69-72 seconds

**^ logarithmic on tick distribution**

50 passengers, 10 ticks distribution, 1000ms tick

START: 2017/11/21 16:45:50.065

END: 2017/11/21 16:51:28.273

TIME ELAPSED: 338 seconds