

Week 9 Assignment: Implementing Multithreading in the Parking System Server for

ICT 4315-1 Object-Oriented Methods & Paradigms II

Jack Hermanson

University of Denver University College

June 1, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

Introduction

This paper discusses the basic implementation of multithreading in the parking system's server so that it can simultaneously handle multiple requests. A developer may choose to add concurrency like this to enhance an application's performance and prevent queues from building up unnecessarily. For the parking system, this addition of concurrency could allow multiple entrances to a single parking lot to handle requests at the same time. For example, let's say there is an entrance to a parking lot on University Blvd, and another entrance on Evans Ave. Two cars are attempting to scan in at the same time. Without concurrency, one entrance would have to wait for the other's request to complete before continuing. With concurrency, they can both be handled at the same time. One thing to consider, however, is that multiple threads accessing the same memory location can lead to corrupted or overwritten data. To prevent that, special care must be taken to use data structures that are designed for concurrency and thread safety.

Starting Code: No Concurrency

The starting code (i.e., the state of the code after the previous assignment) used a single thread for the server. It listened for requests and handled them one at a time. If one request was in progress, the next one would have to wait. To illustrate this, I added a delay of 5 seconds in the processor method, which helps clearly demonstrate that one must complete before the next begins. Here is how that looks with timing information added:

```
Server is running on port 8000
[19:47:29] Handling request on thread: HTTP-Dispatcher
Delaying 5s
[19:47:34] Request has been completed on thread HTTP-Dispatcher
Took 5.086456125 seconds
[19:47:34] Handling request on thread: HTTP-Dispatcher
Delaying 5s
[19:47:39] Request has been completed on thread HTTP-Dispatcher
Took 5.004289958 seconds
[19:47:39] Handling request on thread: HTTP-Dispatcher
Delaying 5s
[19:47:44] Request has been completed on thread HTTP-Dispatcher
Took 5.007840084 seconds
```

As you can see, the first request is handled at 19:47:29, then 5 seconds later it completes. The next request is handled for 5 seconds, then it completes. Finally, the last request is handled for 5 seconds, then it completes. Each request takes about 5 seconds, and the total time between request 1 and request 3 is 15 seconds. This screenshot also shows that each request was handled on the same thread “HTTP-Dispatcher.” Request 3 started at the same time as request 1, but it had to wait an extra 10 seconds for the requests before it to finish.

Implementing Concurrency on the Server

To implement concurrency, I specified that the server should use up to 4 threads by calling the `setExecutor` method on the `HttpServer` with a parameter of a fixed thread pool of 4 threads (Tutorials Point 2025). This tells the server to create a fixed-sized pool of 4 threads and use those to handle requests. The screenshot below shows this new code as well as the commented-out previous code.

```
// Create server instance.  
HttpServer server = HttpServer.create(new InetSocketAddress(port: 8000), backlog: 0);  
server.createContext(path: "/command", new PostHandler());  
// server.setExecutor(null); // default - single-threaded  
server.setExecutor(Executors.newFixedThreadPool(nThreads: 4)); // Allow 4 threads  
System.out.println("Server is running on port 8000");  
server.start();
```

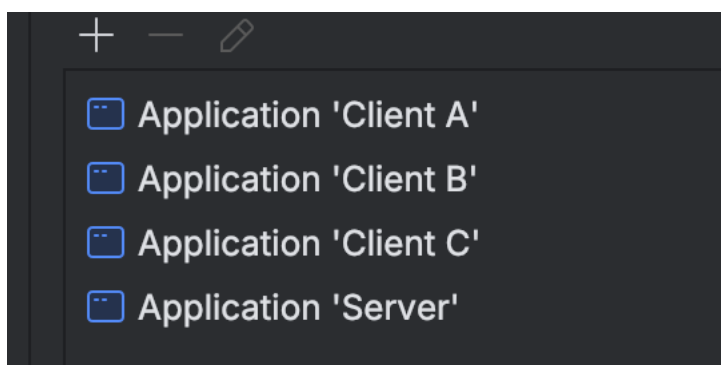
In the PostHandler class, I modified the handle method to give us more information.

First, I use System.nanoTime() to record a moment in time when the request begins its processing. At the end of the method's execution, I use System.nanoTime() again to compute the difference between the start and end time. This helps us verify exactly how long it takes each request.

I also added code that instructs the program to print the time that it started processing the request as well as the time when it completed. This helps us verify *when* a request was handled so that we can see if it happened at the same time as any other requests.

Verifying Concurrency and Timing Request Processing

With the addition of timing information to our code, we can now verify that simultaneous requests are processed concurrently. I had 3 requests go out to the server at the same time while I monitored the output from the server. Here are the configuration and output with concurrency:



```
Server is running on port 8000
[20:07:29] Handling request on thread: pool-1-thread-1
Delaying 5s
[20:07:29] Handling request on thread: pool-1-thread-2
Delaying 5s
[20:07:30] Handling request on thread: pool-1-thread-3
Delaying 5s
[20:07:34] Request has been completed on thread pool-1-thread-1
Took 5.086413833 seconds
[20:07:34] Request has been completed on thread pool-1-thread-2
Took 5.008009209 seconds
[20:07:35] Request has been completed on thread pool-1-thread-3
Took 5.004258416 seconds
```

As you can see, the requests are handled at almost exactly the same time. The slight discrepancy between the third request and the first two is just the result of my IDE taking some time to start up each client individually. The first request is handled on thread “pool-1-thread-1”, and it takes 5 seconds. However, the second and third requests are handled on their own respective threads, also taking 5 seconds each. Because each request was handled by its own thread, the program executed much more quickly and did not require any request to wait for a prior request to complete. Requests 1, 2, and 3 were all processed by the same program at the same time, just on different threads.

Challenges

One challenge that I ran into was making sure each `ParkingLot` instance could handle concurrency. Each `ParkingLot` had a `HashMap` of permits to timestamps, so the lot could track which permit was scanned and when. That works fine when the program is single-threaded, but there could be issues when the program is run on multiple threads. For example, one thread

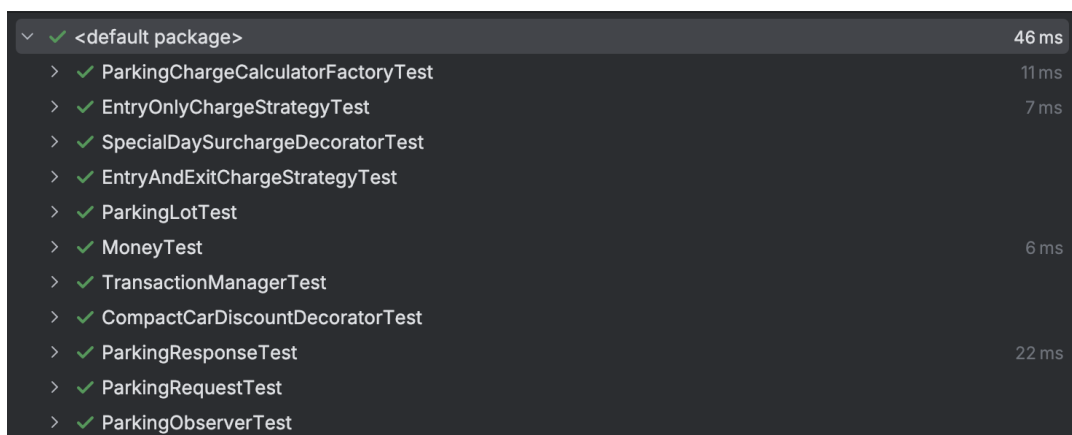
could be attempting to add a new item to that HashMap at some arbitrary memory location, while another thread attempts to do the exact same thing, creating a race condition where we cannot guarantee which of the threads will succeed in writing to the hash map.

One method that I thought of was using a Boolean “lock” reference that each thread would read from. While lock is true, it would do nothing, but as soon as it is false, the thread can safely set the lock to true, modify the parking lot’s data, and then set it back to false when it is done so other threads can do what they need to do. This also applies to reading from the HashMap, as theoretically one thread could attempt to read data while another is still actively writing to it.

Fortunately, this complexity is handled for us in the ConcurrentHashMap class, which allows “multiple threads to read and write data simultaneously, without the need for locking the entire map” (GeeksForGeeks 2025). This class was created to handle this exact situation; multiple threads need to access and modify the map concurrently.

Program Compiles and Tests Pass

The program builds successfully and all unit tests pass, indicating that the changes did not break anything.



A screenshot of a test runner interface with a dark background. It shows a list of 12 unit tests, each preceded by a green checkmark and a right-pointing arrow. The tests are grouped under a collapsed package header '<default package>'. The total execution time for the package is 46 ms, and individual test times are listed on the right. The tests are: ParkingChargeCalculatorFactoryTest (11 ms), EntryOnlyChargeStrategyTest (7 ms), SpecialDaySurchargeDecoratorTest, EntryAndExitChargeStrategyTest, ParkingLotTest, MoneyTest (6 ms), TransactionManagerTest, CompactCarDiscountDecoratorTest, ParkingResponseTest (22 ms), ParkingRequestTest, and ParkingObserverTest.

✓ <default package>	46 ms
> ✓ ParkingChargeCalculatorFactoryTest	11 ms
> ✓ EntryOnlyChargeStrategyTest	7 ms
> ✓ SpecialDaySurchargeDecoratorTest	
> ✓ EntryAndExitChargeStrategyTest	
> ✓ ParkingLotTest	
> ✓ MoneyTest	6 ms
> ✓ TransactionManagerTest	
> ✓ CompactCarDiscountDecoratorTest	
> ✓ ParkingResponseTest	22 ms
> ✓ ParkingRequestTest	
> ✓ ParkingObserverTest	

References

GeeksforGeeks. 2025. "ConcurrentHashMap in Java." *GeeksforGeeks*. Last modified February 14,

2025. <https://www.geeksforgeeks.org/concurrenthashmap-in-java/>.

Tutorials Point. 2025. "Java Concurrency – newFixedThreadPool Method." *Tutorials Point*. Last

modified March 25, 2025.

https://www.tutorialspoint.com/java_concurrency/concurrency_newfixedthreadpool.htm.