Thread Pool Architecture

In part 1 we implemented a custom thread pool in a server and achieved server/client communication. In the question we were asked to implement only 'Query' for part1, rather we implemented both **'Buy' and 'Query'.** Below are the accepted requests that a client can make and the possible messages that the client can receive from the server.

Client Requests should be in this format:

Note: All these request messages are **not** case sensitive and we are also stripping any additional spaces.

1. **Buy Tux** 2. **Query Tux** 3. **Buy Whale** 4. **Query Whale**

Server responses :

"Stock over: 0"
You will see this response when there is no stock of the toy that you are trying to buy/query

"Number of items left: 3.0 Cost of the item: 10.99"
You will see this message when the client tries to query/buy an available item. If the client is buying it, we automatically reduce the count and print the items left along with the cost of the item. If the client is just querying the item, we show the current number of items along with its cost.

"Item not found: -1"
This message is seen when the client tries to buy any item other than whale and tux

"Invalid message entered"
You will see this message when the client sends an invalid request. Anything other than the above discussed message formats.

Architecture:

We have a Server class and a Client class. When we start the server, the main thread creates a new thread which is the dispatcher thread. This thread continuously listens to the client request and adds the client sockets to a dispatcher queue.The other part of the main function in the server initiates a threadpool and starts 5 threads that keep listening to the dispatcher queue.

We used two semaphores to utilize the dispatcher queue across the dispatcher thread and the other thread pool threads. Now we will discuss the use of these two semaphores and how they help us in the synchronization.

Semaphore sem = **new** Semaphore(0);

‘sem’ is a semaphore that is initialized to 0. We use this semaphore to make sure that the threads pull the sockets from the queue only when there are any sockets inside them. Initially when the server is started, all the threads will be started and are locked by this semaphore because there is nothing in the queue yet. We can avoid the busy wait of checking a null queue continuously by using this semaphore. We only release this semaphore after a socket is accepted at the dispatched thread. We acquire this semaphore before we start processing in the thread pool threads.

Semaphore sem_q = **new** Semaphore(1);

‘sem_q’ is a semaphore that is initialized to 1. We used this semaphore to make sure only one thread among the dispatcher thread and the 5 thread pool threads are modifying the queue. We acquire and release this semaphore in all the 6 threads before and after entering the critical region.

Two mutexes sem_whale and sem_tux are used to make changes to read/write the data related to tux and whale. I wanted the write/read operations on whale and tux to run concurrently. Hence I used two mutexes each for whale and tux.

We have two Client classes: **Client.java** and **ClientLoadTest.java**. Client.java is a simple client which asks you to provide the server’s host name. Then it establishes a socket connection with the server and you will be prompted to make a request. Enter the request message and you will receive the reply from the server. The socket connection will be closed from both the ends after this.

We can use ClientLoadTest.java to test various combinations of requests and their latencies. We basically create multiple client threads that make simultaneous requests to the server, in the main function of this class. Each client thread makes 100 requests to the server. Now you can run this class on the command line in various formats. Based on the parameters you pass, we perform different load tests. Here is how you can perform load tests:

Samples of how you can pass parameters:

java ClientLoadTest 128.119.243.168 query 2
java ClientLoadTest 128.119.243.168 buy 2
java ClientLoadTest 128.119.243.168 2

This command line takes three inputs ‘128.119.243.168’ - **hostname**, ‘query’ & ‘buy’- **method name** and ‘2’ - **the number of clients** you want to run in parallel. Now our main function creates two threads and they start sending the server 100 requests each. We compute the average latency for each request for all the clients that you started. That value will be printed at the end of the output. We also printed all the messages from client threads to server and the responses

from server. You can observe both the clients hitting the server when you take a look at the output. If you don't provide any method name we are sending random requests (both buy and query randomly).

Note: The method name parameter is not case sensitive. Also the order in which you pass these parameters doesn't matter.

i.e, "java ClientLoadTest.java 128.119.243.168 2 Query" will also work well

java ClientLoadTest 128.119.243.168 query
java ClientLoadTest 128.119.243.168 buy
java ClientLoadTest 128.119.243.168

When you don't pass num of clients parameter, we do the load test for **one client, two clients,three clients, four clients, five clients each**. We get the latencies for five different scenarios increasing the number of clients for each iteration. We store these latencies in a json file and use them to plot the graphs. As you can see, you can specify the requests you want to perform the load test on. If no method is given, we do the load tests on random requests.

A brief overview of our architecture: