**README - COSC 315 Mini Project 2**

Tanvir Kang - 35470160

Carson Perreaux - 85322311

Zachary Maludzinski - 32950164

**Java**

**Design Explanation**

There was a choice to use three distinct classes in order to take advantage of the object oriented nature of Java, a class for requests with their unique attributes, a master(main) class, and a class of slave threads. This provided an easy framework to create request objects on a random basis, create slave objects on initialization, and allowed the master to only contain code relevant to itself. In order to implement synchronization, the choice to use synchronized methods was quite obvious along with a synchronized linkedlist as the implementation of the queue interface. Whenever a slave or master wants to access the linked list, they must use a synchronized method, along with using a synchronized list, this prevents concurrency errors and prevents threads from taking requests out of order.

**How to Compile and Run**

Note: A “assignment.jar” file has already been created for your convenience however the following steps will describe how on a UNIX machine create and execute the files.

1. Move the “Main” folder located in the “Java” folder to a location of your choosing.
2. Open the terminal, and navigate into the Main/Source directory containing the .java files
3. Execute the following command “javac -d ../classes \*.java”
   1. This creates “className.class” objects in the classes folder
4. Enter “cd ../classes" to navigate to the classes folder.
5. Enter “jar -cvmf manifest.txt assignment.jar pkg/\*.class”
   1. This will create a .jar file called assignment.jar, using the classes in pkg folder
6. To run this program, enter the following command “java -jar assignment.jar”
7. Close the terminal if you wish to exit the program.

**Functions and Usage**

**void getInput()**

Get Input Prompts the user for input, where N is the number of slave threads and size of the input buffer, and M is the maximum number of seconds any one request can take. Both N and M are positive integers.

**void createSlaves()**

Creates a new array of SlaveThread objects, as well as a new Thread array that creates the runnable slave instances, both of these arrays are of length N, where N is specific by the input from the user with the getInput() function. createSlaves returns the program flow back to the main function.

**void generateRequests()**

This is the master thread that generates requests. It first calls the startSlaves() function, then runs a do-while loop, where it creates a request and checks the queue by calling checkQueue(). If checkQueue() returns true, then add(request) is called. If checkQueue() returns false, then wait(request) is called, and the master Thread is put to sleep for 1 second.

**void startSlaves()**

This will start executing the slave threads in a first-in, first-out basis, until there are no more threads to start.

**void wait(Request r)**

This will cause the main Thread to sleep for 500 milliseconds, then check the queue to see if it is full. If it’s full, it will sleep for another 500 milliseconds and repeat this process until the queue is not full. Once the queue is not full, it will add the request to the queue and return execution back to the calling function.

**void add(Request r)**

Adds the request, passed in as the parameter, to the queue.

**boolean checkQueue()**

checkQueue looks to see if the queue is full by checking if the size of the queue is less than N, specified by the input from the getInput() function. The function returns a boolean value back to the calling function. If the queue is full, it will return false. If the queue is not full, then it will return true.

**Int getN()**

Returns the value of N, which is given from the input from the user with the getInput() function.

**Comments and Experience**

When we began our implementation, we read about Threads and Synchronization on Oracle. When reading articles online, a lot of people were talking about using thread pools, but after a little more research, we found that using arrays worked great and kept the code simple and concise. The most difficult part was learning how to compile and execute Java projects without Eclipse, this was a unique challenge with the command line and involved hours of learning and troubleshooting. Overall this experience was quite challenging, however very rewarding as it did not “hold our hand” and expected students to design, form, and implement their own unique solutions.

**C**

**Design Explanation**

This implementation uses structs and a global queue array for most arguments. This choice was made largely for ease of use. Since both request length and id can be contained in a single struct, it allows for handy and easy variable retrieval. A global array was used for the queue simply to make development easier. Specifically, an array was used because C has no native queue data structure. The functions push and pop were created to add “queue-like” functionality to the array. Synchronization is achieved through the use of semaphores within the consumer function. A thread must lock the semaphore before popping a request off the request queue. The semaphore is then released. Throughout testing, this achieves consistent synchronization.

**How to Compile and Run**

1. Move the C\_Implementation folder to your desired directory.
2. Navigate to the C\_Implementation folder in the terminal.
3. Type the command “make”. This will create your executable.
4. Type ./finalMake to execute the newly created executable.
5. The program will prompt the user for desired N and M before beginning.
6. Close the terminal if you wish to terminate the program.

**Functions and Usage**

**struct request pop(int size)**

This function effectively “pops” from the globally accessible queue array of requests and returns the request. It removes the first request struct from the queue and moves every other request struct forward one place in the array to “fill in” the empty space. This function must take in the size of the array and returns the “popped” request struct.

**int push(struct request item, int size)**

This function takes in a request struct and size of the globally accessible queue array of requests. The request struct is then added to the end of the queue of request structs. If the array is full of “active” requests and the addition fails, the function returns 0. If the addition to the end of the queue succeeds, the function returns 1.

**void init(int size)**

This function effectively “empties” the globally accessible queue array of request structs. It achieves this by setting the length of every request to 0 and the id to -1. This initializes the array to be used for the push and pop functions.

**void \*consumer(void \* tArgs)**

This function is the function to be executed by the consumer threads. It takes in a pointer to a struct containing thread arguments. Currently, the only argument contained in the tArgs struct is size, which indicates the user-specified size of the globally accessible queue array of requests.

**Comments and Experience**

I felt like implementing this problem in C involved a lot of head-scratching. The lack of high-level language features like error-handling and queues presented unique problems during the process. I feel as though the amount of effort involved in developing in C was fairly high, with a few moments of frustration and deadlock. However, compiling and exporting with C seems to be much easier than Java, which is one positive side. As well, the direct memory interaction with C allows for some more creative solutions.