**Operating Systems BankTellerSimulator Report**

**Serena McPherson 19765644**

**Main Function:**

My main function creates a Queue struct named c\_queue using the createQueue function, initialises a TellerTotals struct named totals using the initTellerTotals function, and opens a log file named "r\_log" using openLogFile function.

The function then creates a pthread\_t variable named customer\_th and an array of 4 pthread\_t variables named teller\_th, and initialises a loop variable i.

The function prints a message indicating that it is starting the threads, creates a new thread for the customer function using pthread\_create, and creates 4 new threads for the teller function using a for loop and pthread\_create.

The function then waits for the threads to finish executing using pthread\_join, and prints a message indicating that the threads have been joined.

Finally, the function frees the memory allocated for c\_queue using freeQueue function, totals using freeTellerTotals function, and the logfile using closeLogFile function, and returns 0 to indicate successful completion of the program.

The TellerTotals struct stores information about the active tellers, the number of customers they have served, and utilizes a mutex for synchronization during read and write operations. Similarly, the LogFile struct contains a FILE\* pointer to a file stream and also incorporates a mutex to ensure proper synchronization during write operations.

After initializing the parameters and shared memory, the main() function creates one thread for the customer() function and four threads for the teller() function using pthread\_create(). The customer() thread is responsible for reading from the c\_file.txt file and pushing customers to the queue, while the teller() threads are responsible for popping customers from the queue and serving them. All threads are provided with a void pointer to the previously defined Parameters struct.

**Customer()**

The Customer() function begins by converting the argument it received from void\* to Parameter\* and defining variables for later use. It then opens the c\_file.txt file (defined in macros.h) and enters a while loop that continues until it has read every line in the file. Inside the loop, Customer() repeatedly creates a customer object and reads its number and type from the file using fscanf(). Afterward, it sleeps for the specified customer\_interval, params->tc, before attempting to push the customer to the queue.

To push a customer to the queue, Customer() first needs to acquire a lock on the queue by using pthread\_mutex\_lock(&queue->lock). Once it obtains the lock, it checks if the queue is full. If the queue is full, customer() blocks and waits using pthread\_cond\_wait() for the queue to have space, signaled by the queue->not\_full condition that is triggered when a teller thread successfully pops a customer from the queue. However, if there is room in the queue, the customer threads proceed to insert the customer at the back of the queue using the insertStart() function (see appendix B for the LinkedList code). It then sets the customer's arrival time using localtime\_r() and logs the arrival of the customer to the log file using the LOG() variadic macro.

Once the customer has been pushed to the queue, Customer() unlocks the &queue->lock and signals the new\_customer condition to unblock a waiting teller. It then returns to the beginning of the while loop to process the next customer, unless it has reached the end of the file. In that case, it sets the boolean queue->incoming to FALSE, closes the file, and exits the function.

**Teller()**

Similar to the customer thread, a teller thread begins by converting the argument from `void\*` to `Parameter\*` and defining some basic variables. It then needs to determine which teller it is and assign the corresponding teller ID. To achieve this, it acquires a lock for the `TellerTotals` structure, increments the `num\_tellers`, sets the acquired value as its teller ID, and releases the lock. Additionally, it stores the start time as `start\_time`. If there are already four tellers, the thread will release the lock and terminate.

Once these initial steps are completed, the teller can start serving customers. Firstly, it locks the customer queue and checks if the `incoming` flag is set. If it is, the teller waits for the customer thread to signal on `new\_customer`. Once the teller resumes, or if the `incoming` flag was set to false, it exits the loop if the queue is empty. Otherwise, it proceeds to remove the last customer from the queue using the `removeLast()` function (refer to Appendix B), records the customer's response time, and releases the lock. Before releasing the lock, the teller signals `queue->not\_full`, which wakes up the customer thread if it was previously blocked due to a full queue. After releasing the lock, the teller thread writes to the log file and calls `teller\_serve()`, which causes the thread to sleep for a specified duration. After serving the customer, the teller records the customer's finish time, logs relevant information, and frees the customer. The teller then increments its tally before acquiring the queue lock and repeating the while loop.

When a teller thread exits the while loop, it releases `queue->lock` and signals `queue->new\_customer`. This means that if the `customer()` thread sets `incoming` to false and exits while a teller is still waiting on the `new\_customer` condition, another teller thread can unblock it by exiting and signaling on `new\_customer`. The unblocked teller can then observe that the queue is empty and exit as well.

Before a teller exits, it records its finish time, locks `totals->lock`, logs its final statistics, decrements `num\_tellers`, and releases the lock. If it is the last teller to exit, it also logs the final statistics of all the tellers before releasing the lock.

After the last teller exits, `main()` resumes and performs memory cleanup before exiting successfully.