Operating Systems Assignment Report: Customer Queue

Justyn Mahen, Student ID: 21029112

This report for the 2023 semester one Operating Systems assignment report explains the shared resources/variables, shared data structures, threads, mutual exclusion, and synchronization in the program 'Customer Queue' that I have been tasked with implementing. It will also cover any cases where my program is not working properly according to the assignment brief.

Throughout the program, there are 2 shared global variables that are used. These are int total_customers; and pthread_mutex_t mutex; that can be found in the cq.c file. They are also declared in the teller.h file as extern int total_customersl; and extern pthread_mutex_t mutex;. The first shared global variable int total_customers; is a shared variable that records the total number of customers that have been served by the implemented tellers. The other shared global variable is a mutex that reinforces that when accessing shared resources, there is a mutual exclusion.

There are also 2 shared data structures that are evident in the program. These are the c_queue and _teller structures. The _teller structure can be found in teller.h which contains a thread ID, a customer number counter to count how many customers are served, a pointer to the shared c_queue and a teller termination flag. The c_queue structure is in queue.h and is for the FIFO queue where customers are queuing up to be served by the tellers.

Threads play a very important role in the program. They are what enable the customer and tellers to work as desired. In the main function that is implemented in cq.c, 1 customer thread and 4 teller threads are created. The customer thread is responsible for running the function customer() which reads and enqueues customers from c_file.txt into the shared data structure c_queue. The 4 teller threads that are created each use the teller() function to serve customers by accessing the shared customer queue and dequeuing them from it then updating the count of the total number of customers served.

My program achieves mutual exclusion by using pthread_mutex_unlock and lock functions. They are used in critical sections in functions in which shared variables and data structures are accessed. An example of this would be in the teller() function's critical sections when the shared variable total_customers and shared data structure c_queue are accessed.

Overall synchronization is achieved in my implementation of the program using the shared c_queue and mutex. The mutex makes certain that only 1 teller thread can update the count of the total number of served customers or access the queue at a time. Using the mutex implemented, the 4 teller threads update the count of the total number of customers served and their own individual counter of how many customers they each have served in a mutually exclusive way.

There is 1 case where my implementation of the program customer queue is not working accordingly to the assignment brief. This case is that my program does not take command line arguments. I have chosen not to implement this because when I attempted implementing this, the program will only run correctly for a short amount of time before it suddenly only takes in customers and does nothing else. The program will take the command line arguments if it is implemented using argc, argv and atoi in the main function however when I try to assign the command line argument variables to what they are meant to do such as determine the size of c_queue, represent the time duration to serve customers and enqueue them, the program will have the issue when after a while it only ever enqueues customers and nothing else will happen in such that the tellers don't serve them and the customers will be infinitely trapped in the queue. This is why I have chosen to not

implement command line arguments and have chosen to hard code the length of c_queue, the periodic enqueue time and the times it takes to for tellers to serve withdrawals, deposits and information requests. To change these, the user would have to change the hard coded values to what they want. If command line arguments were to implemented, a programmer would have to first use argc, argv and atoi in the main function then make appropriate changes to the code to ensure that the inputs assigned to the command line arguments are actually being used and work appropriately.

Besides the command line inputs, to my understanding all other things are working as intended. The outputs to r_log and statistics printed to the terminal are correct. The screenshots below are samples of the outputs: