ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 312E
Computer Operating Systems

Homework 2

Ali Emre Kaya
150210097
kayaemr21@itu.edu.tr

SPRING 2025

Contents

1 Challenges

1.1 Randomness

For some reason that I still don't fully understand, the expression `rand() % 2` was returning only 1 for a while, and then it started returning only 0. I thought this might be because the `rand()` function is based on time and the threads were running simultaneously. However, I also created randomness mechanisms based on different factors and independent timings, but I still couldn't achieve a very smooth distribution of randomness. The best result I obtained was with the randomness approach I am currently using.

1.2 Retrying Mechanism

If there are a sufficient number of products that have not yet been purchased in other threads, I really had a hard time writing the mechanism to try and buy again. First, I wanted to check all the threads and look at their values, but I decided that I couldn't do this directly, so I kept the data in a large array as a copy of the threads. If the thread couldn't find enough products in stock, it would filter the threads that had the products it wanted in the large array, and then it would sort them and try again with a new thread when it could get enough products in the shortest time.

Sorting may not be the most efficient solution. The reason for this is that after sorting, I chose a retry time according to what was finished in the shortest time, if even one of these threads I chose completed, my retry would be in vain and I would have to create a new thread again.

1.3 Some Libraries

The libraries that will and will not be used are specified in the assignment file, but I used default libraries such as $< unistd.h >$, $< string.h >$, I think these are not prohibited.

1.4 One C File

I guess I should have done the homework in a single C file. But cramming this project into something like this would have reduced readability and made it difficult for me to develop, so I proceeded by dividing it into smaller pieces.

- src - `main.c` is the runable, main file. `readHelper.c` help to read the txt file. `threadHelper.c` especially help to implement retry mechanism. `utils.c` generally have some log functions.
- include - has 2 header file, one of the main request struct, and second one for helping to create a retry request.
- data - is include `input.txt` and `info.log`.
- bin - for binary files.
- .devcontainer - determine the docker environment.

2 Questions

2.1 How did you ensure that no more than two customers could make a payment at the same time?

First, I defined a global variable, and then I wrote a code that will check it and loop. Before entering the loop, this code checks if there is an available cashier. If there is an available cashier, it directly enters the loop. Otherwise, it waits for a cashier to become available.

If there is no cashier available, the thread puts itself to wait with `pthread_cond_wait(&payment_cond, &payment_mutex)`. When one of the other threads that is busy with the cashier finishes and notifies the system using `pthread_cond_signal(&payment_cond)`, the waiting thread wakes up and proceeds to the cashier. In order to prevent the woken thread from harming the running program, I do the wake-up process while mutex is locked and I wake up my sleeping thread with mutex unlock, because waking it up locks the program, there is no need for this feature at the moment, I unlock it as soon as it wakes up.

The semaphore method could be used to keep track of the number of cashiers. A structure could be created by starting the semaphore from the number of cashiers and decreasing as the cashiers were busy and increasing when they were available.

2.2 What strategy did you use to manage reservation timeouts, and how did your implementation ensure thread safety during stock access?

I made my Implementation reliable with mutexes, if there were threads that would work on a common area, I would directly lock the area and make the other thread wait, thus preventing race conditions. Also, instead of running threads unnecessarily, I tried to increase efficiency by putting them to sleep and waking them up when the time came. For example, the address I kept the stocks was the same for each thread, they all had to work on a single address, and I tried to create a reliable mutex structure especially in these areas.

I managed the reservation timeout by creating a thread structure that dies when the reservation timeout expires. As long as the thread timeout is not up, it can make a request to the system and search for an available cashier, but when its timeout is up and it cannot find a suitable cashier, it is terminated.

My mechanism for checking stock availability was as follows

- First check if there are enough items in the current stock, and if there are enough items, buy them directly from stock.
- If there are not enough items in stock, check the current working threads: sort the threads trying to get the desired product according to their closest expiration times and start counting by adding the ones that finished in the closest time. When you reach enough items, create a new thread after that many seconds. (This newly created thread will work as a retry thread and adjust its logs accordingly.)

- If suitable conditions are met, the newly formed retry thread can re-run and continue trying to get the desired products.

2.3 Under what conditions does a product request thread terminate?

Situations where threads are terminated can be listed as follows:

- Timeout - If the determined time for a thread has passed and the thread has not received its product within this time period, the thread is terminated.
- Success - If the thread has successfully received its product, the thread is terminated.
- Globally Insufficient Stock - If there is insufficient product in stock, the thread is terminated.
- Globally Sufficient Stock - If there are enough items in the carts of other threads, the thread dies and creates a child to try again.

In order to avoid memory leaks and any stray threads in the system, I emptied the used, dynamically allocated areas properly with `free(arg);`, and also safely terminated the threads for a safe thread structure `pthread_exit(NULL);`

REFERENCES