Documentation

Problem Statement

Supermarket inventory

Due: week 4.

Each student will fully solve a single problem. However, looking at how to solve the others is highly recommended.

Goal

The goal of this lab is to refresh the knowledge regarding threads and mutexes.

The programs to be written will demonstrate the usage of threads to do non-cooperative work on shared data. The access to the shared data must be protected by using mutexes.

Common requirements

1. The problems will require to execute a number of independent operations, that operate on shared data.
2. There shall be several threads launched at the beginning, and each thread shall execute a lot of operations. The operations to be executed are to be randomly choosen, and with randomly choosen parameters.
3. The main thread shall wait for all other threads to end and, then, it shall check that the invariants are obeyed.
4. The operations must be synchronized in order to operate correctly. Write, in a documentation, the rules (which mutex what invariants it protects).
5. You shall play with the number of threads and with the granularity of the locking, in order to asses the performance issues. Document what tests have you done, on what hardware platform, for what size of the data, and what was the time consumed.

1. Supermarket inventory:

There are several types of products, each having a known, constant, unit price. In the begining, we know the quantity of each product.

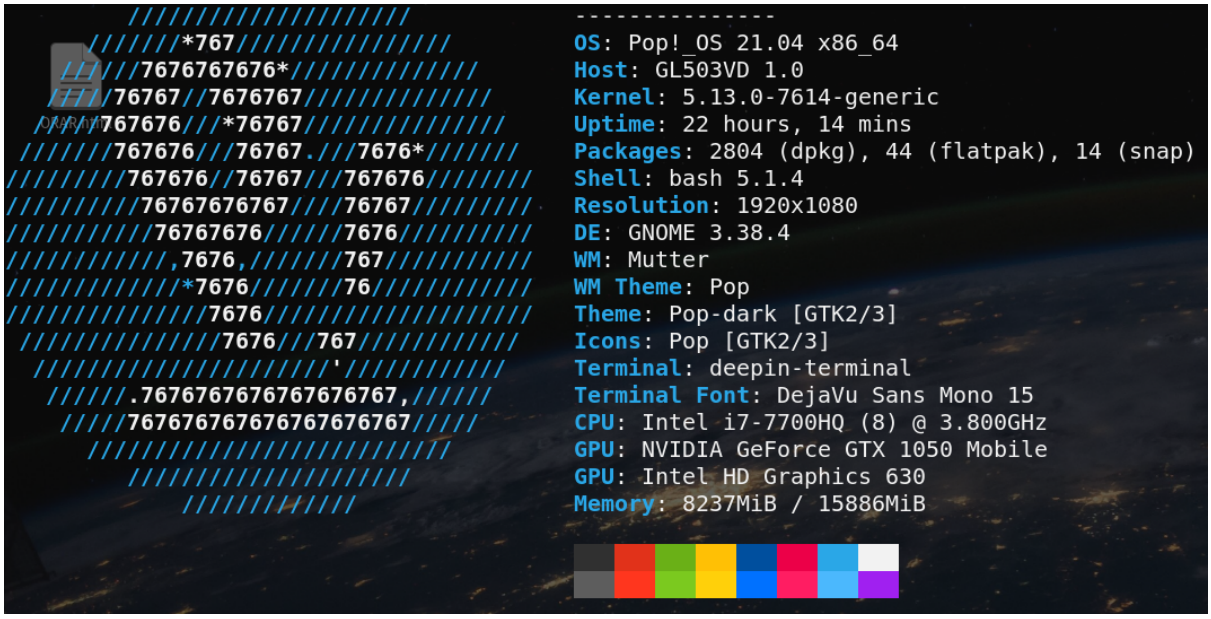
We must keep track of the quantity of each product, the amount of money (initially zero), and the list of bills, corresponding to sales. Each bill is a list of items and quantities sold in a single operation, and their total price.

We have sale operations running concurrently, on several threads. Each sale decreases the amounts of available products (corresponding to the sold items), increases the amount of money, and adds a bill to a record of all sales.

From time to time, as well as at the end, an inventory check operation shall be run. It shall check that all the sold products and all the money are justified by the recorded bills.

Hardware Platform

My Laptop with the following specifications:



Testing done

Test 1:

Number of runs: 10 – Average Time: 336.5 ms

Hardcoded Variables:

```
THREAD_COUNT = 100;
PRODUCT_PRICE_LIMIT = 1000;
PRODUCT_QUANTITY_LIMIT = 10000;
NUMBER_OF_CHECKS = 10;
```

Lock Granularity:

- Each product has its own mutex.
- The bill list has a mutex of its own.

Test 2:

Number of runs: 10 – Average Time: 50s

Hardcoded Variables:

```
THREAD_COUNT = 100000;
PRODUCT_PRICE_LIMIT = 1000;
PRODUCT_QUANTITY_LIMIT = 10000;
NUMBER_OF_CHECKS = 10;
```

Lock Granularity:

- Each product has its own mutex.
- The bill list has a mutex of its own.

Test 3:

Number of runs: 10 – Average Time: 49ms

Hardcoded Variables:

```
THREAD_COUNT = 4;
PRODUCT_PRICE_LIMIT = 1000;
PRODUCT_QUANTITY_LIMIT = 10000;
NUMBER_OF_CHECKS = 10;
```

Lock Granularity:

- Each product has its own mutex.
- The bill list has a mutex of its own.

Test 4:

Number of runs: 10 – Median Time: 2540ms

Hardcoded Variables:

```
THREAD_COUNT = 100;
PRODUCT_PRICE_LIMIT = 1000;
PRODUCT_QUANTITY_LIMIT = 10000;
NUMBER_OF_CHECKS = 10;
```

Lock Granularity:

- The WHOLE SHOP has one single mutex
-

Conclusion

Controlling access on one single resource at a time has its own overhead cost, but it’s a worthwile tradeoff in the long run.