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.

My Problem

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.

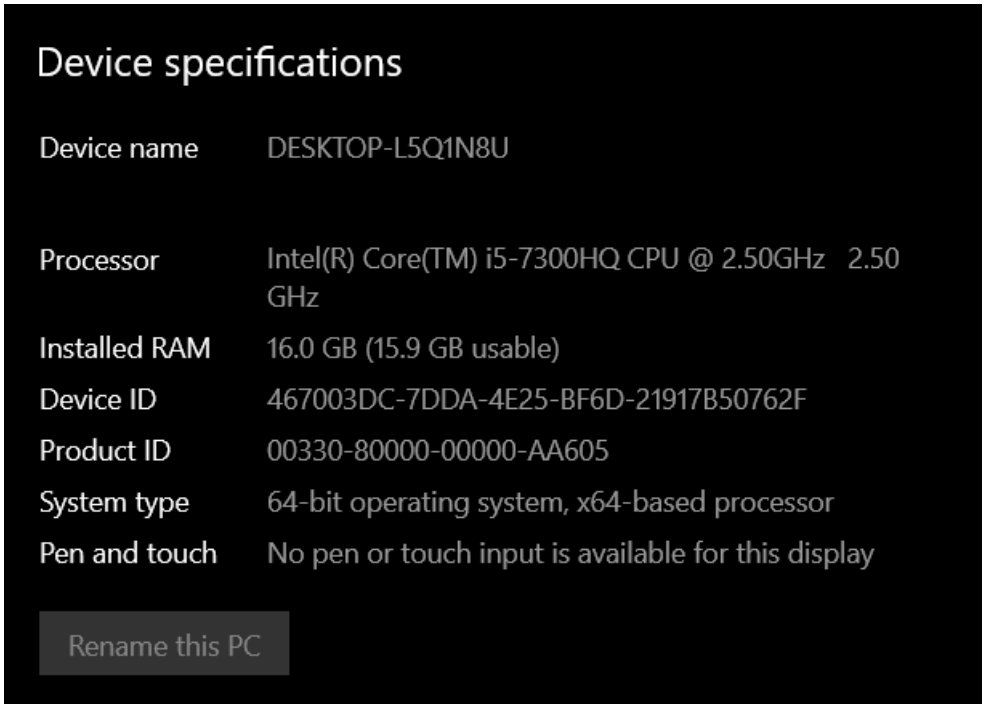
We must keep track of the quantity of each product, the amount of money (initially zero), and the list of bills, corresponding to sales. Each bill is a list of items and quantities sold in a single operation, and their total price.

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 – Median Time: 548ms

Hardcoded Variables:

```
private final static int THREAD_COUNT = 100;
private final static int PRODUCT_PRICE_LIMIT = 1000;
private final static int PRODUCT_QUANTITY_LIMIT = 10000;
private final static int 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 – Median Time: 1323ms

Hardcoded Variables:

```
private final static int THREAD_COUNT = 100;
private final static int PRODUCT_PRICE_LIMIT = 1000;
private final static int PRODUCT_QUANTITY_LIMIT = 10000;
private final static int NUMBER_OF_CHECKS = 10;
```

Lock Granularity:

- The whole inventory has one single product mutex
- The bill list has a mutex of its own.

Test 1:

Number of runs: 10 – Median Time: 2012ms

Hardcoded Variables:

```
private final static int THREAD_COUNT = 100;
private final static int PRODUCT_PRICE_LIMIT = 1000;
private final static int PRODUCT_QUANTITY_LIMIT = 10000;
private final static int NUMBER_OF_CHECKS = 10;

this.numberOfRuns = 10000; //each thread simulates 10000 customers
```

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.

Here's a bottom-up review of how my program works, starting from our domain and making our way up to the main service.

In my solution, each product object has its own mutex.

```
...


public class Product implements Serializable {
    private String name;
    private int price;
    private int quantity;
    private final ReentrantLock productMutex;


    public Product(String name, int price, int quantity) {
        this.name = name;
        this.price = price;
        this.quantity = quantity;
        this.productMutex = new ReentrantLock();
    }

    ...

    public boolean decreaseQuantity(){
        this.productMutex.lock();
        if(this.quantity <= 0){
            this.productMutex.unlock();
            return false;
        }
        else{
            this.quantity--;
        }
        this.productMutex.unlock();
        return true;
    }

    ...
}
```

Each of our products takes its name from a predefined list of product names.

```
private final List<String> NAMES_OF_PRODUCTS = List.of("Laptop", "Mouse",
"Telefon",
        "Mouse Pad", "Scaun", "Tastatura", "Cabluri");
```

Each Bill is a map of a product name maped to a number of bought items.

```
...

public class Bill implements Serializable {
    private Map<String, Integer> wantedProducts;
```

```
    public Bill() {
        wantedProducts = new HashMap<>();
    }

    public void addPurchasedProduct(String productName){
        if(!wantedProducts.containsKey(productName)){
            wantedProducts.put(productName, 1);
            return;
        }
        wantedProducts.put(productName, wantedProducts.get(productName) +
1);
    }

    public Map<String, Integer> getWantedProducts() {
        return wantedProducts;
    }

    public void setWantedProducts(Map<String, Integer> wantedProducts) {
        this.wantedProducts = wantedProducts;
    }

    ...
}
```

When we actually buy a product, we simply call

```
public Boolean buyProduct(String name) {
    return this.inventory.get(name).decreaseQuantity();
}
```

When we do this, it locks the mutex for that respective product

```
public boolean decreaseQuantity(){
    this.productMutex.lock();
    if(this.quantity <= 0){
        this.productMutex.unlock();
        return false;
    }
    else{
        this.quantity--;
    }
    this.productMutex.unlock();
    return true;
}
```

and then checks if that product is still available. If it is, then it decreases the quantity, unlocks the mutex and returns true.

If the product is not available, then the method unlocks the mutex and returns false. In this case, the product is not added to the current bill for the current customer (if the customer asks for bread, and you don't have bread anymore, you won't bill the customer for bread).

```
Boolean transactionSucceeded =
        this.repository.buyProduct(wantedProduct.getProductName());
if(transactionSucceeded){
    boughtItems.add(wantedProduct.getProductName());
}
```

Each thread is inside a class that implements the Runnable interface.

The driver method for each thread is this one:

```
@Override
public void run() {
    for(int runNumber=0; runNumber<numberOfRuns; runNumber++){
        List<WantedProduct> wantedProductList =
generateListOfWantedProducts();
        List<String> boughtItems = new ArrayList<>();
        for(WantedProduct wantedProduct: wantedProductList){
            for(int i=0; i<wantedProduct.getQuantity(); i++){
                Boolean transactionSucceeded =

this.repository.buyProduct(wantedProduct.getProductName());
                if(transactionSucceeded){
                    boughtItems.add(wantedProduct.getProductName());
                }
                else{
                    break;
                }
            }
        }
        if(boughtItems.size() > 0){
            this.createAndAddBillOfBoughtItems(boughtItems);
        }
    }
}
```

It basically generates a list of wanted products (the customer walks into our shop asking for 10 laptops, 3 keyboards and 2 chairs.

For each of this wanted products, we will add as many products as we can sell to the bill (if the customer asks for 10 laptops, but we only have 7, we will bill him for 7 laptops)

We only bill the customer if he actually received anything (if the customer asks for 10 laptops and we have none, we won't create a separat bill for that customer).

The way we generate the list of wanted products is this one

```
private List<WantedProduct> generateListOfWantedProducts() {
    List<WantedProduct> wantedProductList = new ArrayList<>();

    for(String productName: namesOfProducts){
        WantedProduct wantedProduct =
                new WantedProduct(productName, random.nextInt(10));
        wantedProductList.add(wantedProduct);
    }
```

```
    return wantedProductList;
}
```

The main method creates a hardcoded number of threads, stop them after a while, checks the inventory for inconsistencies, then resumes the execution. In total, there's a hardcoded number of checks done for each run.

```
public void run(){
        ProductInventoryRepository repository = new
ProductInventoryRepository();
        populateRepository(repository);
        ProductInventoryRepository repositoryClone =
(ProductInventoryRepository) Copy.deepCopy(repository);

        List<Thread> threads = new ArrayList<>();

        long start = System.currentTimeMillis();

        for(int numberOfRuns=0; numberOfRuns<NUMBER_OF_CHECKS;
numberOfRuns++){
            for(int i=0; i < THREAD_COUNT; i++){
                ProductRunnable productRunnable = new
ProductRunnable(repository, NAMES_OF_PRODUCTS);
                Thread thread = new Thread(productRunnable);
                threads.add(thread);
                thread.start();
            }

            for(Thread thread: threads){
                try {
                    thread.join(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            checkBillsAndInventory(repository, repositoryClone);

        }

        checkBillsAndInventory(repository, repositoryClone);

        long finish = System.currentTimeMillis();
        long timeElapsed = finish - start;
        System.out.println(timeElapsed);

        System.out.println(repository);
    }
```

The main way we check our repository is this one. We basically reconstruct our repo based on the bills and we check with the initial repository that was created at startup, simulating the real way shops do it.

```
    private void checkBillsAndInventory(ProductInventoryRepository
repository,
                                        ProductInventoryRepository
```

```
repositoryClone) {
        ProductInventoryRepository auxiliaryRepository =
(ProductInventoryRepository) Copy.deepCopy(repository);
        for(Bill bill: auxiliaryRepository.getBillList() ){
            for(String boughtProduct: bill.getWantedProducts().keySet()){
                auxiliaryRepository.restockProduct(boughtProduct,
bill.getWantedProducts().get(boughtProduct));
            }
        }

        for(String productInInventory:
auxiliaryRepository.getInventory().keySet()){
            if
(auxiliaryRepository.getInventory().get(productInInventory).getQuantity()
!= repositoryClone.getInventory().get(productInInventory).getQuantity()){
                throw new RuntimeException(String.format("Product
quantities do NOT match! %s %s - %s %s",
                        productInInventory,

auxiliaryRepository.getInventory().get(productInInventory).getQuantity(),
                        productInInventory,

repositoryClone.getInventory().get(productInInventory).getQuantity()));
            }
        }

        System.out.println("Final Profit:" + repository.getMoney());
    }
```

This is the way we populate our repo at startup

```
private void populateRepository(ProductInventoryRepository repository) {
    for(String productName: NAMES_OF_PRODUCTS){
        int productPrice = random.nextInt(PRODUCT_PRICE_LIMIT);
        int productQuantity = random.nextInt(PRODUCT_QUANTITY_LIMIT);
        Product productToAdd = new Product(productName, productPrice,
productQuantity);
        repository.addProduct(productToAdd);
    }
}
```

And all the hardcoded values are set in the service.

```
private final static int THREAD_COUNT = 100;
private final static int PRODUCT_PRICE_LIMIT = 1000;
private final static int PRODUCT_QUANTITY_LIMIT = 10000;
private final static int NUMBER_OF_CHECKS = 10;
```

The main class simply creates a new service and runs it

```
public class Main {
    public static void main(String[] args) {
        ProductService productService = new ProductService();
        productService.run();
}}
```