Programming Fundamentals 3 **2020**

Instructor: Prof. Walter Binder **TA:** Eduardo Rosales, Filippo Schiavio, Matteo Basso

Assignment 3 (8 points)

Due date: 22 November 2020, 08:00 a.m.

For questions regarding this assignment, please contact Matteo Basso via email: matteo.basso@usi.ch.

This assignment contributes 8% to the overall grade. Please follow strictly the submission instructions written at the end of the assignment.

Environment Setup

The exercise presented in this assignment is available as maven project in the attached folder: **/Ex1**. Before proceeding, you are advised to properly set the `JAVA_HOME` variable and you must install maven following the official installation guide.

Run and Test the Projects

To compile, run, and test the projects, you should use the same instructions provided in assignment 2, but properly changing the package name to `assignment3` along with providing the proper class name as a parameter. For instance:

```
# compile the java source
mvn compile
# run the main class using the ShopSynchronized implementation class
mvn exec:java -Dexec.mainClass="assignment3.Main"
-Dexec.args="ShopWaitNotify"
# test the ShopWaitNotify implementation class
mvn test -Dtest=MainTest#ShopWaitNotify
```

Note: please remember that races are non-deterministic such that you should run the tests multiple times. A single failure indicates a wrong implementation. Please note also that tests are useful only to check whether errors occur. They do not ensure that an exercise was solved correctly.

If you need further details or you encounter errors, please refer to the Java Tools lecture recordings that include a detailed explanation of maven and a step by step guide to install and run it, including running tests and changing the class name passed as a parameter to a given execution.

Exercise 1 - Unbounded Buffer - Wait/Notify, ReentrantLock, Semaphore (8 points)

A shop buys second-hand products from sellers and sells them again to its customers. The shop starts its activity with an empty warehouse, and sells products following a First In First Out (FIFO) policy. The following is the Java implementation of class `Product`. Each product has a unique ID which is assigned when the product is created.

```java
import java.util.concurrent.atomic.AtomicLong;

public final class Product {
  private static final AtomicLong currentId = new AtomicLong();
  private final long id;

  public Product() {
    this.id = currentId.getAndIncrement();
  }

  public long getId() {
    return id;
  }
}
```

A shop must implement the `Shop` interface. In this way, it is possible to easily test multiple and different implementations of a shop.

```java
public interface Shop {
  public void buy(Product product) throws InterruptedException;
  public Product sell() throws InterruptedException;
}
```

`Shop1Impl` is an initial Java implementation of interface `Shop`. It buys products from sellers, using method `buy(..)`, and sells them to customers, using method `sell()`. Products are stored and retrieved from the shop warehouse, which is implemented using a `Queue`. For simplicity, a customer always buys the first available product.

```java
public final class ShopImpl implements Shop {
  private final Queue<Product> warehouse = new LinkedList<Product>();

  public void buy(Product product) throws InterruptedException {
    processOrder();
    warehouse.add(product);
  }

  public Product sell() throws InterruptedException {
    Product productToSell;
    processOrder();
    while(warehouse.size() == 0) {
      // waiting for a product
    }
    productToSell = warehouse.poll();
    processOrder();
    return productToSell;
  }

  private static void processOrder() throws InterruptedException {
    // simulating waiting time
    Thread.sleep(100);
  }
}
```

Sellers are uniquely identified with an ID. For simplicity, sellers are able to sell a single product to the shop calling its method buy(..).

```java
public final class Seller implements Runnable {
  private final long id;
  private final Shop shop;

  public Seller(long id, Shop shop) {
    this.id = id;
    this.shop = shop;
  }

  @Override
  public void run() {
    try {
      shop.buy(new Product());
    } catch(InterruptedException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Customers are uniquely identified with an ID. For simplicity, customers are able to buy a single product from the shop calling its method sell().

```java
public final class Customer implements Runnable {
  private final long id;
  private final Shop shop;
  private Product product;

  public Customer(long id, Shop shop) {
    this.id = id;
    this.shop = shop;
  }

  @Override
  public void run() {
    try {
      this.product = shop.sell();
    } catch(InterruptedException ex) {
      throw new RuntimeException(ex);
    }
  }

  public long getId() {
    return this.id;
  }

  public Product getProduct() {
    return this.product;
  }
}
```

Recalling the bounded buffer problem, please note that Sellers represent producer threads that put new items into a shared buffer, i.e., the warehouse, while Customers represent consumer threads that retrieve items from the shared buffer. Note that the buffer is **unbounded**, i.e., it has no size limit. The total number of threads is given by the sum of the number of sellers and customers. Each of them is able to either produce or consume a single entry.

A test scenario has been implemented in the `Main` class. It allocates a `Shop`, several `Customers`, and `Sellers` making them use the shared buffer. At the end of the execution, the program performs some sanity checks to ensure the correctness of the implementation, printing errors (if any) on the standard error output. In addition, the test scenario prints the number of products sold by the shop and the execution time that the simulation took on the standard output.

```java
public final class Main {
  private static final int NUMBER_OF_SELLERS = 1000;
  private static final int NUMBER_OF_CUSTOMERS = 1000;

  public static void main(String[] args) throws Exception  {
    if (NUMBER_OF_SELLERS < NUMBER_OF_CUSTOMERS) {
      System.err.println(
        "- ERROR: NUMBER_OF_SELLERS must be equal to" +
        "or greater than NUMBER_OF_CUSTOMERS."
      );
      System.exit(1);
    }

    Class<? extends Shop> shopClass = null;

    String shopClassName = args.length > 0
      ? args[0]
      : "ShopImpl";
    try {
      shopClass = Class.forName("assignment3." + shopClassName)
                              .asSubclass(Shop.class);
    } catch (ClassNotFoundException ex) {
      System.err.println(
        "- ERROR: the provided command line parameter is not valid. " +
        "Cannot find a class named \"" + shopClassName + "\""
      );
      System.exit(2);
    }

    List<Thread> threads = new ArrayList<Thread>(
      NUMBER_OF_SELLERS + NUMBER_OF_CUSTOMERS
    );
    Shop shop = shopClass
                  .getDeclaredConstructor()
                  .newInstance();
    Seller[] sellers = new Seller[NUMBER_OF_SELLERS];
    Customer[] customers = new Customer[NUMBER_OF_CUSTOMERS];

    for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++) {
      customers[i] = new Customer(i, shop);
      threads.add(new Thread(customers[i]));
    }

    for (int i = 0; i < NUMBER_OF_SELLERS; i++) {
      sellers[i] = new Seller(i, shop);
      threads.add(new Thread(sellers[i]));
    }

    Collections.shuffle(threads);

    long startTime = System.nanoTime();
    for (Thread thread : threads) {
      thread.start();
    }

    for (Thread thread : threads) {
      thread.join();
    }
    long endTime = System.nanoTime();

    // assert that the same product has been sold only once
    Map<Product, Integer> duplicates = new HashMap<Product, Integer>();
    for (int i = 0; i < customers.length; i++) {
      Customer customer = customers[i];
      Product product = customer.getProduct();

      if (product != null) {
        int numberOfOccurences = duplicates.getOrDefault(product, 0);
        duplicates.put(product, numberOfOccurences + 1);
      }
    }

    boolean errors = false;
    for(Map.Entry<Product, Integer> entry : duplicates.entrySet()) {
      Product product = entry.getKey();
      int times = entry.getValue();

      if (times > 1) {
        errors = true;
        System.err.println(
          "- ERROR: Unique product number " + product.getId() +
          " has been sold " + times + " times."
        );
      }
    }

    System.out.println(
      "\nNumber of products sold: " + duplicates.size() +
      "\nExecution time: " + (
        TimeUnit.MILLISECONDS.convert(endTime-startTime, TimeUnit.NANOSECONDS)
      ) + " ms"
    );

    if (errors) {
      System.exit(3);
    }
  }
}
```

Please note that you **do not need** to copy the source code from this document. The corresponding maven project is attached to this assignment in the folder: **/Ex1**.

You are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested in this document to solve the exercise.

You can test different behaviors changing the constants defined on top of the `Main` class (`NUMBER_OF_SELLERS` and `NUMBER_OF_CUSTOMERS`). However, please make sure that **`NUMBER_OF_SELLERS` is always equal to or greater than `NUMBER_OF_CUSTOMERS`**.

1.1 Questions

1. Consider the implementation provided in the `ShopImpl` class. The following is a sample output obtained after running `Main.main(..)`:

   ```
   - ERROR: Unique product number 780 has been sold 2 times.
   - ERROR: Unique product number 399 has been sold 2 times.
   - ERROR: Unique product number 688 has been sold 2 times.
   - ERROR: Unique product number 516 has been sold 2 times.
   - ERROR: Unique product number 914 has been sold 2 times.
   - ERROR: Unique product number 844 has been sold 2 times.
   - ERROR: Unique product number 87 has been sold 2 times.
   Number of products sold: 560
   Execution time: 734 ms
   ```

   This means that a single product, identified by its unique ID, has been sold multiple times to different customers, which must not happen. The error can be identified in the `buy(..)` and `sell()` methods of the `ShopImpl` class. Since the implementation lacks proper synchronization, races may happen based on the scheduling or interleaving of multiple concurrent threads. In particular, it is necessary to synchronize the accesses to the `warehouse` shared mutable field.

   Locate class `ShopSemaphore` (a renamed copy of class `ShopImpl`). Modify this class using Semaphores to avoid the races produced in methods `ShopSemaphore.buy(..)` and `ShopSemaphore.sell()` when multiple threads access the `warehouse` field concurrently. In addition, remove the inefficient busy waiting (shown below) in method `ShopWaitNotify.sell()`.

   ```java
       while(warehouse.size() == 0) {
         // waiting for a product
       }
   ```

   Update, compile, run, and test the implementation providing `ShopSemaphore` as parameter.

   Hint: you would need to read the documentation of methods Semaphore.acquire() and Semaphore.release().

   Hint: `Shop` implements an unbounded buffer. You need to check and reason about the difference between a bounded and an unbounded buffer, modifying the solution provided in slide 9 of the slide set: L4 - Semaphores and Monitors. Reason also about how many semaphores you would strictly need given the unbounded buffer.

2. Locate class `ShopWaitNotify` (a renamed copy of class `ShopImpl`). Modify this class using **`synchronized` blocks** to avoid the races produced in methods `ShopWaitNotify.buy(..)` and `ShopWaitNotify.sell()` when multiple threads access the `warehouse` field concurrently. In addition, remove the inefficient busy waiting in method `ShopWaitNotify.sell()` using Object.wait() and Object.notify().

   Please make sure that the synchronized blocks to be completed enclose only what is strictly needed, i.e., do not use synchronization for code portions that do not require it.

   Update, compile, run, and test the implementation providing `ShopWaitNotify` as parameter.

3. Locate class `ShopReentrantLock` (a renamed copy of class `ShopImpl`). Modify this class using the class ReentrantLock to avoid the races produced in methods `ShopReentrantLock.buy(..)` and `ShopReentrantLock.sell()` when multiple threads access the `warehouse` field concurrently. In addition, remove the inefficient busy waiting in method `ShopReentrantLock.sell()`.

   Update, compile, run, and test the implementation providing `ShopReentrantLock` as parameter.

   Hint: you should use one ReentrantLock and one Condition. You would also need to check the documentation of methods ReentrantLock.lock(), ReentrantLock.unlock(), ReentrantLock.newCondition(), Condition.await(), and Condition.signal().

4. Consider a situation where the `NUMBER_OF_SELLERS` is very high in comparison to `NUMBER_OF_CUSTOMERS`, such that a lot of producers frequently put data into the `Queue` but almost no consumers remove such data. In such a condition, what problem can arise at execution time? Theoretically explain the causes of the issue.

   Hint: remember that the buffer used in this exercise is unbounded, i.e., it has no size limit.

5. Consider the solutions provided in the previous questions that allow customers to wait until a product is available. Supposing `NUMBER_OF_SELLERS` to be less than `NUMBER_OF_CUSTOMERS`, what would be the behavior of the implementation? Theoretically explain the causes of the issue.

SUBMISSION INSTRUCTIONS

(please notice that the following instructions are mandatory. Submissions that do not comply, will not be graded):

- Assignments are individual work. Duplicates are not graded.
- Assignments must be submitted via **iCorsi**. Do not send anything by email.
- Solutions to theoretical questions must be presented inside a (single) PDF or plain text file named:
  assignment<AssignmentNumber>.<Surname>.<Name><.pdf|.txt>
  In that document you must clearly specify your name and the question you are addressing. Please do not submit scanned documents using handwritten text.
- In addition, you must attach the Java sources that you have completed to solve the assignment. Submissions without Java source code or with Java source code that does not compile will not be graded.
- All the produced files must be collected into a single archive file (.zip or .tar) named:
  assignment<AssignmentNumber>.<Surname>.<Name><.zip|.tar.gz>