Programming Fundamentals 3 **2022**

Instructor: Prof. Walter Binder **TA:** Matteo Basso, Andrea Adamoli, Marco Tereh

---

Assignment 2

Due date: 22 November 2022, 10:00 a.m.

For questions regarding this assignment, please contact **Marco Tereh** via email: `marco.tereh@usi.ch`.

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

Environment Setup

The exercises presented in this assignment are available as a single maven project in the attached folder: **/Ex**. Before proceeding, you are advised to properly set the `JAVA_HOME` variable and you must install maven following the official installation guide (you may also use brew for MacOS).

Compile and Run the Projects

To compile the projects, you should use the command: `mvn compile`. For instance:

1. Open your terminal
2. Navigate to the root folder of the project, **/Ex**
3. Compile the source files:

   ```
   mvn compile
   ```

4. To run the project please follow the instructions provided for each exercise

Note: please remember that races are non-deterministic such that you should **run the implementations multiple times**.

Exercise 1 - Bounded Buffer - Semaphore, Wait/Notify, ReentrantLock (3 points)

A *restaurant* receives orders from *clients*. Each order is processed by a *chef*. A Java program collects the orders of the clients and queues them to be processed by a chef following a First In First Out (FIFO) policy. The `Restaurant` interface models the behavior of such a program. It allows both clients to add new orders using the `receive()` method and chefs to retrieve them using the `cook()` method. A program implementing the `Restaurant` interface serves as temporary storage for orders while they wait to be processed, storing them in a shared queue. This allows both clients to create orders without knowing (having a reference to) any chef and chefs to get new orders without the need for interacting with clients.

```
1  public interface Restaurant {
2      void receive(Order order) throws InterruptedException;
3      Order cook() throws InterruptedException;
4  }
```

To run the implementations that you will develop to complete this exercise, please use the commands:

```
mvn compile
mvn exec:java -Dexec.mainClass="shared.Main"
```

While solving this exercise, you are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested. You are only allowed to change what is strictly required below.

`NaiveRestaurant` is a naive implementation of `Restaurant` capable of handling at most `size` outstanding orders. It receives orders from clients using `receive()` and retrieves them to chefs using `cook()`. Pending orders are stored in a shared `Queue`. If the `NaiveRestaurant` receives an order while its queue is full, it blocks and continuously tries to reinsert the order in the queue until it succeeds. If `cook()` is called while its queue is empty, it blocks and continuously tries to retrieve an order until it succeeds.

```
1  public class NaiveRestaurant implements Restaurant{
2      private final int size;
3      private final Queue<Order> queue = new LinkedList<>();
4      public NaiveRestaurant(int size) {
5          this.size = size;
6      }
7      @Override
8      public void receive(Order order) throws InterruptedException {
9          while (queue.size() >= size) {
10             if (Thread.interrupted()) throw new InterruptedException();
11         }
12         queue.add(order);
13     }
14     @Override
15     public Order cook() throws InterruptedException {
16         while (queue.size() == 0) {
17             if (Thread.interrupted()) throw new InterruptedException();
18         }
19         return queue.poll();
20     }
21 }
```

Questions

1. Consider class `NaiveRestaurant`. If you run it, you should get an output similar to one of the following examples. You can ignore the other implementation at the moment.

   ```
   NaiveRestaurant failed: 3 orders have been cooked multiple times and 12
   orders have not been fulfilled.
   ```

   or

   ```
   NaiveRestaurant failed: no activity for 10 seconds; liveness hazard suspected.
   Remaining threads interrupted.
   ```

   Some orders have been lost, others were served multiple times and some chefs or clients got stuck forever. The previous issues can arise because methods `receive()` and `cook()` (of class `NaiveRestaurant`) are not thread safe implementations. Due to the lack of proper synchronization, the scheduling or interleaving of threads can result in races.

   Modify class `BoundedSemaphoreRestaurant`[1] using Java `semaphores` to prevent races. You must use the methods `Semaphore.acquire()` and `Semaphore.release()`.

   Make sure to remove the inefficient *busy waiting* (i.e., the inefficient loop that continuously checks the condition, consuming CPU cycles) in methods `receive()` and `cook()`.

   To test your implementation, please compile the sources and run the program via maven. A correct solution produces the following output:

   ```
   BoundedSemaphoreRestaurant successfully processed all orders.
   ```

2. Modify the `BoundedWaitNotifyRestaurant` class using the `synchronized` keyword to prevent races. Use the methods `Object.wait()` and `Object.notify()` or `Object.notifyAll()` to avoid the inefficient busy waiting.

   To test your implementation, please compile the sources and run the program via maven. A correct solution produces the following output:

   ```
   BoundedWaitNotifyRestaurant successfully processed all orders.
   ```

3. Modify the `BoundedReentrantLockRestaurant` class using a `ReentrantLock` and `Conditions` to prevent races and the inefficient busy waiting. Relevant methods are `ReentrantLock.lock()`, `ReentrantLock.unlock()`, `ReentrantLock.newCondition()`, `Condition.await()` and `Condition.signal()`.

   To test your implementation, please compile the sources and run the program via maven. A correct solution produces the following output:

   ```
   BoundedReentrantLockRestaurant successfully processed all orders.
   ```

[1] The classes that you need to modify to complete exercise 1 are located in folder: **Ex/src/main/java/ex1**

Exercise 2 - Unbounded Buffer - Semaphore, Wait/Notify, ReentrantLock (3 points)

In the previous exercise you were required to implement a restaurant with a maximum capacity, i.e., once the shared queue was full, the restaurant could not accept any new orders. In this exercise you will create implementations of `Restaurant` which are *unbounded*, i.e., they can immediately add any newly received order, but may still block in `cook()` if the queue is empty.

While solving this exercise, you are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested. You are only allowed to change what is strictly required below.

Questions

1. Modify class `UnboundedSemaphoreRestaurant`[2] using `Semaphore` to prevent races and the inefficient busy waiting.

   To test your implementation, please compile the sources and run the program via maven. A correct solution produces the following output:

   ```
   UnboundedSemaphoreRestaurant successfully processed all orders.
   ```

2. Modify `UnboundedWaitNotifyRestaurant` using `synchronized` and `wait()`/`notify()` to prevent races and the inefficient busy waiting.

   To test your implementation, please compile the sources and run the program via maven. A correct solution produces the following output:

   ```
   UnboundedWaitNotifyRestaurant successfully processed all orders.
   ```

3. Modify `UnboundedReentrantLockRestaurant` using `ReentrantLock` and `Condition` to prevent races and the inefficient busy waiting.

   To test your implementation, please compile the sources and run the program via maven. A correct solution produces the following output:

   ```
   UnboundedReentrantLockRestaurant successfully processed all orders.
   ```

4. Consider what happens if there are many more clients (*producers*) than chefs (*consumers*), causing a large number of orders to be received but few to be processed. In such a situation, the unbounded implementations in this exercise would eventually crash. Explain why.

5. `BoundedWaitNotifyRestaurant` has to be implemented using `notifyAll()`, but `notify()` is enough for `UnboundedWaitNotifyRestaurant`. Explain why.

[2] The classes that you need to modify to complete exercise 2 are located in folder: **Ex/src/main/java/ex2**

Exercise 3 - Digital Library - ReentrantReadWriteLock (2 points)

Consider a simple Java application modeling the operation of a digital library. There are three kinds of users of this application:

- **Readers**: Pick a book and read it.
- **Browsers**: Browse the index to find books they might be interested in.
- **Librarians**: Add new books to the library.

The library has a large number of books that is used simultaneously by all types of users. However, readers and browsers act much more frequently than librarians, therefore changes to the index are very rare.

The `Library` interface[3] models such digital library.

```
1  public interface Library {
2      /**
3       * Returns the current size of this library
4       *
5       * @return the size of this library
6       */
7      int getSize();
8
9      /**
10      * Returns a book from this library
11      *
12      * @param title the title of book to be obtained
13      *
14      * @return the book with the title provided or
15      *         null if the book is not available
16      */
17     Book getBook(String title);
18
19     /**
20      * Returns the list of all book titles in this library
21      *
22      * @return a list of the book titles in this library
23      */
24     Set<String> getTitles();
25
26     /**
27      * Returns a random book from this library, or null if the library is empty
28      *
29      * @return a randomly selected book from this library
30      */
31     Book getRandomBook();
32
33     /**
34      * Adds a book to the library
35      *
36      * @param book the book to be added to the library
37      */
38     void addBook(Book book);
39 }
```

[3]The classes that you need to modify to complete exercise 3 are located in folder: **Ex/src/main/java/ex3**

In earlier exercises you used `ReentrantLock` for synchronization, which enforces mutual exclusion over all accesses to critical sections guarded by it, however, that is often stricter than necessary. Although it is usually not possible to allow two writers or a reader and a writer to execute concurrently, multiple readers *can* occur in parallel without compromising data integrity. In cases where data structures are mostly read and rarely modified, *read-write* locks permit efficient access to multiple readers or to a single writer (but not both).

Questions

1. Implement the methods marked `TODO` in `LibraryImpl`. Use the `ReentrantReadWriteLock`'s[4] `read lock` and `write lock` to allow `getSize()`, `getTitles()`, `getBook()`, `getRandomBook` to execute concurrently while still ensuring thread safety when `addBook()` is called.

   To run your implementation, use the commands:

   ```
   mvn compile
   mvn exec:java -Dexec.mainClass="ex3.Main"
   ```

   If your implementation is correct it should print

   ```
   All threads terminated
   ```

   and nothing else. If the output includes other messages before that line, your implementation is incorrect (please see the message itself for more details).

   While solving this exercise, you are **not allowed** to change the architecture of the system, add or remove any logic that is not explicitly requested. You are only allowed to change what is strictly required above.

2. Would the advantage of the `ReentrantReadWriteLock` still be useful even if writes occur more frequently than reads? Why/Why not?

[4] You are not required to use any fairness policy but to use the default (nonfair) policy of `ReentrantReadWriteLock`.

SUBMISSION INSTRUCTIONS

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

- Assignments can only be done individually or in the groups (of up to 3 members) **registered through iCorsi**. Duplicates between groups are not graded.
- Assignments must be submitted via **iCorsi**. Do not send anything by email.
- **Only one member of the group should submit the assignment**. If (by mistake) multiple members of the same group submit an assignment, we will grade only the latest submission per group.
- Solutions to theoretical questions must be presented inside a (single) PDF or plain text file.
  In that document you must clearly specify the name of each member of the group and the question you are addressing.
- Please **do not submit scanned documents using handwritten text**.
- In addition, you must attach the whole maven project that you have modified 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>.<.zip|.tar.gz>