Programming Fundamentals 3 **2021**

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

---

Assignment 2

Due date: 22 November 2021, 08:00 a.m.

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

This assignment contributes 12% to the overall grade. Please follow strictly the submission instructions written at the end of the assignment.

Environment Setup

The two 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.

Compile and Test the Projects

To compile and test the project, you should use the same instructions provided in assignment 1, but properly changing the package name to `assignment2` along with providing the specific class name as a parameter. For instance:

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

   ```
   mvn compile
   ```

4. Test the `BoundedCallCenterSemaphore` implementation:

   ```
   mvn test -Dtest=MainTest#BoundedCallCenterSemaphore
   ```

Please note that, in contrast to assignment 1, **the maven project of this assignment does not contain a `Main` class and hence it is not possible to execute the command `mvn exec:java`**.

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 also note that tests are useful only to check whether errors occur. They do not ensure that an exercise was solved correctly.

Note: please note that after solving correctly the assignment, two tests will still fail. This behaviour is expected.

If you need further details or you encounter errors, please refer to the Java Tools lecture recordings that includes a detailed explanation of maven and a step by step guide to install and run it, including running tests.

Introduction

A *call center* receives calls from several *callers*. Each call is answered by an *operator* in the call center. A program marks each incoming call as a "pending call" and informs the caller that an operator will answer as soon as possible. The call center starts its activity with no pending calls, and operators answer pending calls following a First In First Out (FIFO) policy. The following is the Java implementation of class `Call`. Each call has a unique ID which is assigned when the call is created.

```java
public final class Call {
  private static final AtomicLong currentId = new AtomicLong();
  private final long id = currentId.getAndIncrement();

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

A call center must implement the `CallCenter` interface. In this way, it is possible to easily test different implementations of a call center. An implementation of `CallCenter` must implement a queue of pending calls. Call centers receive calls from callers, using method `receive(..)`, and operators answer to such calls using method `answer()`.

```java
public interface CallCenter {
  void receive(Call call) throws Exception;
  Call answer() throws InterruptedException;
}
```

Concrete implementations of the `CallCenter` interface will be presented later in the context of the two exercises in this assignment.

For simplicity, assume that `Callers` are able to call the call center only a single time. To do so, callers invoke method `receive(..)` of the call center.

```java
public final class Caller implements Runnable {
  private final CallCenter callCenter;

  public Caller(long id, CallCenter callCenter) {
    this.callCenter = callCenter;
  }

  @Override
  public void run() {
    try {
      callCenter.receive(new Call());
    } catch(Exception ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

For simplicity, assume that `Operators` are able to answer only a single call. To do so, operators invoke the method `answer()` of the call center.

```java
public final class Operator implements Runnable {
  private final CallCenter callCenter;
  private Call call;

  public Operator(CallCenter callCenter) {
    this.callCenter = callCenter;
  }

  @Override
  public void run() {
    try {
      this.call = callCenter.answer();
    } catch(InterruptedException ex) {
      throw new RuntimeException(ex);
    }
  }

  public Call getCall() {
    return this.call;
  }
}
```

Please note that `Callers` represent producer threads that put new items into the shared queue of pending calls, while `Customers` represent consumer threads that retrieve items from the shared queue. The total number of threads is given by the sum of the number of callers and customers. Each producer and consumer is able to either produce or consume only a single entry.

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: **/Ex**.

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 in this document.

Exercise 1 - Bounded Buffer - Semaphore, Wait/Notify, ReentrantLock (4.5 points)

The `BoundedCallCenter` interface represents a `CallCenter` capable of handling at most a number of pending calls equal to `MAX_NUMBER_OF_PENDING_CALLS`. Implementations of `BoundedCallCenter` must implement a shared queue of pending calls whose size cannot be greater than `MAX_NUMBER_OF_PENDING_CALLS`.

```java
public interface BoundedCallCenter extends CallCenter {
  static final int MAX_NUMBER_OF_PENDING_CALLS = 10;
}
```

`BoundedCallCenterImpl` is an initial Java implementation of interface `BoundedCallCenter`. It receives calls from callers, using method `receive(..)`, and operators answer them, using method `answer()`. Pending calls are stored and retrieved from the shared queue of pending calls, which is implemented using a `Queue`. If the call center receives a call while the queue of pending calls is full, the call center would continuously retry to insert the call into the queue until a slot is available. For simplicity, an operator always answers the first available call.

```java
public final class BoundedCallCenterImpl implements BoundedCallCenter {
  private final Queue<Call> pendingCalls = new LinkedList<Call>();

  public void receive(Call call) throws Exception {
    // simulating call retrival
    while(pendingCalls.size() >= MAX_NUMBER_OF_PENDING_CALLS) {
    }
    pendingCalls.add(call);
  }

  public Call answer() throws InterruptedException {
    // waiting for a call
    while(pendingCalls.size() == 0) {
    }
    return pendingCalls.poll();
  }
}
```

Please note that `BoundedCallCenter` represents a bounded buffer problem where the buffer is the queue of pending calls. We remark that `Callers` represent producer threads while `Operators` represent consumer threads. The buffer has a size limit equal to `MAX_NUMBER_OF_PENDING_CALLS`.

Questions

1. Consider the implementation provided in the `BoundedCallCenterImpl` class. The following is a sample output obtained after running
   `mvn test -Dtest=MainTest#BoundedCallCenterImpl`:

   ```
   [ERROR] Failures:
   [ERROR]   MainTest.BoundedCallCenterImpl:80->test:61
             Unique call number 21 has been answered 2 times.
             expected:<1> but was:<2>
   [ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
   ```

   This means that a single call, identified by its unique ID, has been answered multiple times by different operators, which must not happen. The error can be identified in the `receive(..)` and `answer()` methods of the `BoundedCallCenterImpl` 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 `pendingCalls` shared mutable field.

   Locate class `BoundedCallCenterSemaphore` (a renamed copy of class `BoundedCallCenterImpl`). Modify this class using Semaphores to avoid the races produced in methods `BoundedCallCenterSemaphore.receive(..)` and `BoundedCallCenterSemaphore.answer()` when multiple threads access the `pendingCalls` field concurrently. You are only allowed to use methods Semaphore.acquire() and Semaphore.release(). In addition, remove the inefficient busy waiting (i.e., the inefficient loop that continuously checks the condition, consuming CPU cycles) in method `BoundedCallCenterSemaphore.receive(..)`:

   ```java
       // simulating call retrival
       while(pendingCalls.size() >= MAX_NUMBER_OF_PENDING_CALLS) {
       }
   ```

   and the inefficient busy waiting in method `BoundedCallCenterSemaphore.answer()`:

   ```java
       // waiting for a call
       while(pendingCalls.size() == 0) {
       }
   ```

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

2. Locate class `BoundedCallCenterWaitNotify` (a renamed copy of class `BoundedCallCenterImpl`). Modify this class using **`synchronized` blocks/methods** to avoid the races produced in methods `BoundedCallCenterWaitNotify.receive(..)` and `BoundedCallCenterWaitNotify.answer()` when multiple threads access the `pendingCalls` field concurrently. In addition, remove the inefficient busy waiting in methods `BoundedCallCenterWaitNotify.receive(..)` and `BoundedCallCenterWaitNotify.answer()` using Object.wait() and Object.notifyAll().

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

3. Locate class `BoundedCallCenterReentrantLock` (a renamed copy of class `BoundedCallCenterImpl`). Modify this class using ReentrantLock to avoid the races produced in methods `BoundedCallCenterReentrantLock.receive(..)` and `BoundedCallCenterReentrantLock.answer()` when multiple threads access the `pendingCalls` field concurrently. You must use one ReentrantLock and one Condition. You are only allowed to use methods ReentrantLock.lock(), ReentrantLock.unlock(), ReentrantLock.newCondition(), Condition.await(), and Condition.signal(). In addition, remove the inefficient busy waiting in methods `BoundedCallCenterWaitNotify.receive(..)` and `BoundedCallCenterWaitNotify.answer()`.

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

Exercise 2 - Unbounded Buffer - Semaphore, Wait/Notify, ReentrantLock (7.5 points)

`UnboundedCallCenterImpl` is a Java implementation of interface `CallCenter` whose shared queue of pending calls has no size limit. In contrast to the previous exercise, the call center is always able to receive and store calls into the queue of pending calls, requiring no busy waiting in method `receive(..)`. However, busy waiting is still present in method `answer()` to retrieve the first call.

```java
public final class UnboundedCallCenterImpl implements CallCenter {
  private final Queue<Call> pendingCalls = new LinkedList<Call>();

  public void receive(Call call) throws Exception {
    pendingCalls.add(call);
  }

  public Call answer() throws InterruptedException {
    // waiting for a call
    while(pendingCalls.size() == 0) {
    }
    return pendingCalls.poll();
  }
}
```

Please note that `UnboundedCallCenterImpl` implements an **unbounded buffer**, i.e., a buffer with no size limit.

Questions

1. Similarly to the previous exercise, the implementation provided in the `UnboundedCallCenterImpl` class lacks proper synchronization. As a result, races may occur based on the scheduling or interleaving of multiple concurrent threads. The following is a sample output obtained after running `mvn test -Dtest=MainTest#UnboundedCallCenterImpl`:

   ```
   [ERROR] Failures:
   [ERROR]   MainTest.UnboundedCallCenterImpl:100->test:61
             Unique call number 14 has been answered 2 times.
             expected:<1> but was:<2>
   [ERROR] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
   ```

   To solve this issue, it is necessary to synchronize the accesses to the `pendingCalls` shared mutable field.

   Locate class `UnboundedCallCenterSemaphore` (a renamed copy of class `UnboundedCallCenterImpl`). Modify this class using Semaphores to avoid the races produced in methods `UnboundedCallCenterSemaphore.receive(..)` and `UnboundedCallCenterSemaphore.answer()` when multiple threads access the `pendingCalls` field concurrently. You are only allowed to use methods Semaphore.acquire() and Semaphore.release(). In addition, remove the inefficient busy waiting (shown below) in method `UnboundedCallCenterSemaphore.answer()`.

   ```java
       // waiting for a call
       while(pendingCalls.size() == 0) {
       }
   ```

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

   Hint: `UnboundedCallCenterImpl` 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.

2. Locate class `UnboundedCallCenterWaitNotify` (a renamed copy of class `UnboundedCallCenterImpl`). Modify this class using **`synchronized` blocks/methods** to avoid the races produced in methods `UnboundedCallCenterWaitNotify.receive(..)` and `UnboundedCallCenterWaitNotify.answer()` when multiple threads access the `pendingCalls` field concurrently. In addition, remove the inefficient busy waiting in method `UnboundedCallCenterWaitNotify.answer()` using Object.wait() and Object.notify().

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

3. Locate class `UnboundedCallCenterReentrantLock` (a renamed copy of class `UnboundedCallCenterImpl`). Modify this class using ReentrantLock to avoid the races produced in methods `UnboundedCallCenterReentrantLock.receive(..)` and `UnboundedCallCenterReentrantLock.answer()` when multiple threads access the `pendingCalls` field concurrently. You must use one ReentrantLock and one Condition. You are only allowed to use methods ReentrantLock.lock(), ReentrantLock.unlock(), ReentrantLock.newCondition(), Condition.await(), and Condition.signal(). In addition, remove the inefficient busy waiting in method `UnboundedCallCenterWaitNotify.answer()`.

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

4. Consider a situation where `NUMBER_OF_CALLERS` is very high in comparison to `NUMBER_OF_OPERATORS`, 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 lead the application to crash due to an error at runtime? 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. While the bounded buffer in `BoundedCallCenterWaitNotify` must be necessarily implemented using `Object.notifyAll()`, the unbounded buffer in `UnboundedCallCenterWaitNotify` can be properly implemented using either `Object.notifyAll()` or `Object.notify()`. Explain the reason why an unbounded buffer can be correctly implemented using `Object.notify()`.

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 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>.<Surname>.<Name><.zip|.tar.gz>