Exercises week 4

Last update: 2024/09/14

Goal of the exercises

These exercises aim to give you practical experience on:

- Identifying sources of errors in concurrent executions of data structures and synchronization primitives.
- Designing and implementing tests to discover and trigger errors in concurrent access to data structures and synchronization primitives.

The tests in this exercise must be written using JUnit 5, see lecture 4.

For convenience, we recall here the Gradle command for running JUnit 5 tests:

```
$ gradle cleanTest test --tests <package>.<test_class>
```

Exercise 4.1 We have repeatedly argued that regular (non thread-safe) Java collections are not adequate for use in concurrent programs. We have even seen how to discover interleavings that show that these collections are non thread-safe. However, how likely is it that those interleavings appear? Can they even occur? In this exercise, your task is to develop some tests that will trigger undesired interleavings when using a non thread-safe data structure in a concurrent environment.

We look into a set of collections implementing the interface `ConcurrentIntegerSet` (see the directory `code-exercises/week04exercises/app/src/main/java/exercises04/`). The interface defines the two most basic operations on sets: `add` and `remove`. Additionally, it defines a `size` method which is useful when writing tests. The file contains three different implementations of the interface:

1. `ConcurrentIntegerSetBuggy`, this class uses an underlying `HashSet` and binds the interface method calls directly to the equivalent operations in the `HashSet`.
2. `ConcurrentIntegerSetSync`, this is exactly as `ConcurrentIntegerSetBuggy`. You will modify so that it to fix the concurrency issues found by some tests you will develop.
3. `ConcurrentIntegerSetLibrary`, this class uses an underlying `ConcurrentSkipListSet` which is already thread-safe.

Note also that there is a skeleton for writing tests in the `ConcurrentSetTest.java` in the tests directory of the Gradle project for the exercises (`app/src/test/java/exercises04/`).

Before you start with the exercise, here are some (possibly) useful hints regarding testing concurrent programs:

- Remember that interleavings appear non-deterministically, so it might be helpful to run the test several times. This can easily be done in JUnit 5 with `@RepeatedTest()`. Depending on how you design your tests, it is possible that you need to run your tests more than 5000 times. However, no more than 10000 executions should be necessary.
- To cover as many interleavings as possible, it is useful to run the tests with increasing number of threads. Typically, slightly more threads than processors should suffice. For instance, in an 8 core machine, running tests with 16 threads should ensure that the execution of threads interleaves.
- To minimize the chance of threads executing sequentially, it is helpful to start all threads at the same time. You may `CyclicBarrier` to make sure that all threads start the execution (almost) at the same time. The same barrier may be used to wait until all threads have terminated their execution.

Mandatory

1. Implement a functional correctness test that finds concurrency errors in the `add(Integer element)` method in `ConcurrentIntegerSetBuggy`. Describe the interleaving that your test finds.

   Note I: Remember that, by definition, sets do not have repeated elements. In fact, if, in a sequential test, you try to insert twice the same element, you will notice that the second insertion will no suceed and `add` will return false.

   Note II: Remember that the execution of the `add()` method in `HashSet` is not atomic.

   Hint I: Even if many threads try to add the same number, the size of the set must be at most 1. Think of an assertion related to the size of the set that would occur if the same element is added more than once.

2. Implement a functional correctness test that finds concurrency errors in the `remove(Integer element)` method in `ConcurrentIntegerSetBuggy`. Describe the interleaving that your test finds.

   Note: Remember that the execution of the `remove()` method in `HashSet` is not atomic.

   Hint: The method `size()` may return values smaller 0 when executed concurrently. This fact should useful in thinking of an assertion related to the size of the set that would occur if an element is removed more than once.

3. In the class `ConcurrentIntegerSetSync`, implement fixes to the errors you found in the previous exercises. Run the tests again to increase your confidence that your updates fixed the problems. In addition, explain why your solution fixes the problems discovered by your tests.

4. Run your tests on the `ConcurrentIntegerSetLibrary`. Discuss the results.

   Ideally, you should find no errors—otherwise please submit a bug report to the maintainers of Java concurrency package :)

5. Do a failure on your tests above prove that the tested collection is not thread-safe? Explain your answer.

6. Does passing your tests above prove that the tested collection is thread-safe (when only using `add()` and `remove()`)? Explain your answer.

Challenging

You have probably noticed that we have not tested the `size()` method in any of the implementations above. Now we turn our attention to this method.

7. It is possible that `size()` returns a value different than the actual number of elements in the set. Give an interleaving showing how this is possible.

   Hint: In `HashMap`, `size()` is a constant time function that returns the value of a `size` field defined in the class. This field is increased or decreases when adding or removing elements in the set. See the implementation in https://github.com/openjdk/jdk17/blob/4afbcaf55383ec2f5da53282a1547bac3d099e9d/src/java.base/share/classes/java/util/HashSet.java#L182. For `ConcurrentIntegerSetSync`, the javadoc says that *“ the size method is not a constant-time operation. Because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal.”* See https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html.

8. Is it possible to write a test that detects the interleaving you provided in 7? Explain your answer.

Exercise 4.2 In this exercise, we focus on testing whether a Semaphore implementation works according to its specification. In particular, we will test the Semaphore implementation in the file `SemaphoreImp.java` in `app/src/main/java/exercises04/`.

As specification for the semaphore, we use Herlihy’s description on how a semaphore class behaves (see Herlihy, Section 8.5, page 194). Concretely, we focus in this property of the specification,

> "Each semaphore has a *capacity* that is determined when the semaphore is initialized. Instead of allowing only one thread at a time into the critical section, a semaphore allows at most c threads, where c is its capacity."

Mandatory

1. Let *capacity* denote the `final` field `capacity` in `SemaphoreImp`. Then, the property above does not hold for `SemaphoreImp`. Your task is to provide an interleaving showing a counterexample of the property, and explain why the interleaving violates the property.

   Note: If a thread successfully acquires the semaphore—i.e., it executes `acquire()` and does not block—then we consider that it has entered its critical section.

   Hint I: Consider only interleavings that involve method calls to `acquire()` and `release()`. Note that field accesses cannot occur in interleavings because all fields are defined as `private`.

   Hint II: The operations executed by the main test thread—i.e., the thread that starts the threads for testing—are also part of interleavings. You may use the main thread to execute method calls in the tested semaphore before starting the threads that try to enter the critical section.

2. Write a functional correctness test that can trigger the interleaving you describe in 1. Explain why your test triggers the interlaving.

   Note: The note and hints in 1 also apply to this exercise.

Exercise 4.3 Now we turn our attention to the readers and writers monitor you implemented in Week 2. I repeat here the specification of the problem:

- Several reader and writer threads want to access a shared resource.
- Many readers may access the resource at the same time as long as there are no writers.
- At most one writer may access the resource if and only if there are no readers.

Now consider the following modification for the second above:

- **At most 5 readers** may access the resource at the same time as long as there are no writers.

Recall that we discussed this modification in class.

Challenging

1. Implement the above modification in your *fair* reader-writer monitor from week 2. That is, it should allow at most 5 readers accessing the resource.

   Note: This exercise is not difficult, but it is a pre-requisite for the following one.

2. Write a functional correctness test that checks whether there are ever more than 5 readers accessing the resource.