PCPP IT University of Copenhagen, F2022

Exercises week 4

Last update: 2022/09/22

Goal of the exercises

The goals of this week's exercises are:

- Identify sources of errors in concurrent use of a non thread-safe data structure.
- Design and implement tests to discover errors in concurrent access to non thread-safe data structures.

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/week12exercises/app/src/main/java/testingconcurrency/`). 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 `HashMap`.
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/testingconcurrency/`).

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 http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashSet.java. 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 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.

Exercises week 3

Last update: 2022/09/15

Goal of the exercises

The goal of these exercises is to give you practical experience in designing and implementing *thread-safe* classes.

Note: Remember that we use a concrete definition of thread-safe class (see slides).

Exercise 3.1 A *Bounded Buffer* is a data structure that can only hold a *fixed* set of elements. In this exercise, you must implement such a data structure which allow elements to be inserted into the buffer by a number of *producer* threads, and taken by a number of *consumer* threads.

The specification of the bounded buffer is as follows:

- If a producer thread tries to insert an element into and the buffer is full, the thread must be blocked until an element is taken by another thread.
- If a consumer thread tries to take an element from and the buffer is empty, the thread must be blocked until an element is inserted.

Note that this is an instance of the producer-consumer problem we discussed in the lecture.

In this exercise, your task is to build a simple version of a Bounded Buffer. Section 5.3 in *Goetz* talks about how to use `BlockingQueue`. Also, `java.util.concurrent.BlockingQueue` class implements a bounded buffer as described above.

Your bounded buffer must implement the interface `BoundedBufferInterface.java` (see exercises code folder):

```
interface BoundedBufferInteface<T> {
  public T take() throws Exception;
  public void insert(T elem) throws Exception;
}
```

Of course, your class also must include a constructor which takes as parameter the size of the buffer.

Note that the methods in the interface allow for throwing `Exception`. We do this to not impose constraints on the implementation of the methods. In your implementation, please write the concrete exception your code throws, if any.

The buffer you design, must follow a FIFO queue policy for inserting and taking elements. Regarding the state of your class, you are allowed to use non thread-safe collections from the Java library that implement the `Queue` interface, such as `LinkedList<T>`. This may save some time in the implementation, but do not feel obliged to use these libraries. It is also allowed to use primitive Java arrays. In summary, any collections designed for concurrent access are not allowed.

Mandatory

1. Implement a class `BoundedBuffer<T>` as described above using *only* Java `Semaphore` for synchronization—i.e., Java `Lock` or intrinsic locks (`synchronized`) cannot be used.
2. Explain why your implementation of `BoundedBuffer<T>` is thread-safe. Hint: Recall our definition of thread-safe class, and the elements to identify/consider in analyzing thread-safe classes (see slides).
3. Is it possible to implement `BoundedBuffer<T>` using Barriers? Explain your answer.

   Challenging

4. One of the two constructors to Semaphore has an extra parameter named `fair`. Explain what it does, and explain if it matters in this example. If it does not matter in this example, find an example where it does matter.

Exercise 3.2 Consider a `Person` class with attributes: id (`long`), name (`String`), zip (`int`) and address (`String`). The `Person` class has the following functionality:

- It must be possible to change zip and address together.
- It is not necessary to be able to change name—but it is not forbidden.
- The id cannot be changed.
- It must be possible to get the values of all fields.
- There must be a constructor for `Person` that takes no parameters. When calling this constructor, each new instance of Person gets an id one higher than the previously created person. In case the constructor is used to create the first instance of `Person`, then the id for that object is set to 0.
- There must be a constructor for `Person` that takes as parameter the initial id value from which future ids are generated. In case the constructor is used to create the first instance of `Person`, the initial parameter must be used. For subsequent instances, the parameter must be ignored and the value of the previously created person must be used (as stated in the previous requirement).

Mandatory

1. Implement a thread-safe version of `Person` using Java intrinsic locks (`synchronized`). Hint: The `Person` class may include more attributes than those stated above; including `static` attributes.
2. Explain why your implementation of the `Person` constructor is thread-safe, and why subsequent accesses to a created object will never refer to partially created objects.
3. Implement a main thread that starting several threads the create and use instances of the `Person` class.
4. Assuming that you did not find any errors when running 3. Is your experiment in 3 sufficient to prove that your implementation is thread-safe?

Exercise 3.3 Perhaps you have noticed that Monitors are the most general synchronization primitive (that we have seen). And other primitives can be implemented using them.

Challenging

1. Implement a Semaphore thread-safe class using Java `Lock`. Use the description of semaphore provided in the slides.
2. Implement a (non-cyclic) Barrier thread-safe class using your implementation of Semaphore above. Use the description of Barrier provided in the slides.