PCPP **IT University of Copenhagen, F2024**

Exercises week 3

Last update: 2024/09/08

These exercises aim 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 *only* 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 from the Java library specifically designed for concurrent access are *not* allowed in this exercise.

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). In this exercise, your task is to use monitors to implement other synchronization primitives.

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.