Assignments for week 2

Anders Degn Lapiki, Jacob Kjærulff Furberg, Jonas Ishøj Nielsen

03/09/2020

Contents

0.1 2.1.1 - green

The JDBC connection object is by nature confined to connection thread for request duration so connections will not create race conditions.
The designed class can be seen in $2_1 Main.java$.

0.2 2.1.2 - green

The tests can also be seen in 2.1.Main.java and below.

```
public static void main(String[] args) {
    int n = 100000;

    ConcurrentLinkedQueue<Person> persons = new
       ConcurrentLinkedQueue<>();
    ConcurrentLinkedQueue<Thread> threads = new
       ConcurrentLinkedQueue<>();
    for (int i = 0; i < n; i++) {
        Thread t1 = new Thread(() -> {
            persons.add(Person.generate());
        });
        threads.add(t1);
    }
    for (Thread thread : threads) {
        thread.start();
    }
    try {
        for (Thread thread : threads) {
            thread.join();
        }
    }
    catch (InterruptedException exn) {
        System.out.println("Some thread was
           interrupted");
    }

    HashMap<Long, Boolean> hm = new HashMap<>();
    for (Person person : persons) {
        hm.put(person.id, true);
    }
    System.out.println(hm.keySet().size());

}
```

The last lines of the main method tests whether or not the n persons have unique id's. On 15 run through with 100.000 persons, 100.000 unique ids was found.

0.3 2.1.3 - green

No the test only shows that the id generator don't suffer from race conditions, it doesn't test any other field's potential for race conditions.

1 2.2

1.1 2.2.1 - green, does non-synchronized run forever?

It runs forever, as expected, since the field's value is stored in the threat's cache, which without synchronization it isn't told to update.

1.2 2.2.2 - green, does synchronized run forever?

It runs now, as expected.

1.3 2.2.3 - green, does non-synchronized get run forever?

It runs forever.
Reason is that the value contains stale-data, until TestMutableInteger.class releases lock, which it may first do when main has executed, which happens after t.join(). Without a synchronised get there isn't any guarantee that the set happens before the get.

1.4 2.2.4 - yellow, non-synchronized, field volatile value?

It runs now, because value becomes visible(but not atomic). Volatile is enough as the writes don't depend on its current value(also only one class ever updates value. The variable also doesn't participate in invariants with other state variables and there isn't any reason for locking to be required for any other reason while the variable is being accesses.

2 2.3 - TestLocking1

2.1 2.3.1 - green, simplest natural way to make synchronised

Class becomes thread safe by making each function synchronised.
A slightly better version would make size, get, add and set synchronised while in toString(). Instead of referencing size and items, get them in an synchronized block and store them in separate fields, so the string building can be done not synchronized.

2.2 2.3.2 - green, how does it scale?

The class doesn't scale well as each function call must wait for the other to finish before it can acquire lock.
Most of the functions will also take a longer time, so not only can only one threat work at a time but it will also take a longer time for the thread to finish.

2.3 2.3.3 - yellow

It wouldn't secure thread safety. Because there are multiple functions working as modify-write this will suffer from happens-before race condition. The read operations aren't guaranteed to return up to date data as no locks related to writing is used synchronising the reads.

3 2.4

3.1 2.4.1 - yellow

To ensure that totalSize is correctly maintained in a multi-threaded environment, there must be created a static synchronized method for incrementing totalSize. Line 56 should look like this:

```
public static synchronized void incSize() {
    totalSize++;
}
```

To ensure visibility the function totalSize should also be synchronized, and since it is static the lock is held by DoubleArrayList.class.

3.2 2.4.2 - yellow

The constructor needs to be synchronized which is done by making the constructor private and creating a synchronized factory function like seen below.

```
private DoubleArrayList() {
   allLists.add(this);
 }

 public synchronized static DoubleArrayList
    newInstance() {
   DoubleArrayList object = new DoubleArrayList();
   return object;
 }
```

The allLists function also needs to be synchronized, which since it and newInstance are both static is done by adding the synchronized keyword.

4 2.5 - red

The file 2_5_Main.java contains the code for the tests. Because of JVM it was difficult to get clear answers from the tests. Which for loop was run first had a lot of effect on the results, and how many times the loops where run also had a great effect. The results somewhat stabilized when we ran the loops for as many iterations as was feasible for our machines.

We got the following result for how long on average updating volatile and non-volatile longs.

- Volatile: 60 nanoseconds, nonvolatile: 39 nanoseconds.
- Volatile: 58 nanoseconds, nonvolatile: 40 nanoseconds.
- Volatile: 59 nanoseconds, nonvolatile: 40 nanoseconds.

It can be seen from the results that volatile longs seems to be a lot slower when updated when compared to nonvolatile fields, which makes sense. The update to volatile seems to take on average 48.8% more time, according to our data.

5 2.6 - red

A class contains states detailing how and if the class have been initialized[1]. Each class also has a unique initialization lock ensuring only one thread can initialize the class at a time.

The important lock is the initialization lock for the ResourceHolder. The only purpose of ResourceHolder is to initialize Resource. JVM does not initialize ResourceHolder before it is used. Because it is a static class it will be fully initialized before being released. And since the only way to get to resource is though ResourceHolder, Resource is guaranteed to first be published only after initialization.
Had instead of ResourceHolder there only been a field, there wouldn't be any gurantee that the object referenced had been initialized.

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html