PCPP Assignment 2

Group name: Thread Heresy
Real names : Nedas Surkus, Niclas Abelsen,
Github username: nesu, niab

September 2022

Contents

Exercise 3.1

1

The implementation can be found in the file `BoundedBuffer.java`.
A manual test of this can be found in `BoundedBufferTest.java`

2

All the member variables of the `BoundedBuffer` is private and final which ensures that these cannot escape upon construction or changed directly on publication. However, the class state, that is the variable `linkedListQueue`, can be changed by other threads using the methods `take` and `insert`. That means these methods contains critical sections and must be mutually excluded using a mutex.

3

Implying we will not use semaphores, it will not be possible to implement `BoundedBuffer` using barriers. Barriers are used to ensure that all threads await until they reach a certain point. While they can be instantiated after opened, it does not guarantee that just one thread access the section after the barrier. As this is a continues process where producer threads will produce, and consumer threads will consume, barriers will eventually force a deadlock or ensure that either the producers or customers starve by preventing producers from producing until certain amount of consumer threads have executed. Or vice versa.

Exercise 3.2

1

The implementation can be found in the file `Person.java`.

2

The person constructor is implemented by locking out the class behind a class lock which makes it impossible to access the class before the constructor has finished executing. Each Person class holds a private static variable called `seed` which functions as the initial id value. This seed is initialized only once as a static variable and can be accessed whenever necessary. Since the `seed` is assigned only once if it has not been initialized already, we can ensure that the class will not have a access to partially created object.

3

The implementation can be found in the file `PersonTest.java`

4

Any experiment is not sufficient in ensuring that your implementation is thread safe as the thread-safe implementation can be implemented incorrectly and give out "predictable" data but would crash under heavy load or produce unwanted results. Additionally, it is safe to assume that human error and bias can play a part in the implementation as the developer is trying to implement code that produces a specific result rather than trying to break and test the system. (or lacks the oversight to produce errors). Thus any kind of testing is inherently not safe and cant be 100% trusted to ensure that the application is thread safe.

Exercise 4.1

1

The implementation can be found in the file `ConcurrentSetTest.java`

Hashset is not thread safe. Internally it works similarly to hashmap. Meaning it dynamically allocates a specific amount of space that is used to store key sets of arrays. Whenever we put a value. At certain points the hashset will call a function to resize its size to fit any other potential new adds and does this to increase performance. This specific function recreates the array and while that happens. The array is empty. Thus, adding multiple values at the same time causes interleaving as multiple threads are encountering an empty hash map and adding values to it without waiting for it to resize and repopulate. This can cause values to just disappear and thus causes the test to fail.

In our code there are two operations independent of each other. An atomic integer that increments and then gets. And then once we have the atomic integer value, we attempt to insert it into the hashset. The atomic integer method is atomic. The hashset add method is not. Thus when try to add, we might be adding to a stale version of the hashset that has not finished resizing.

2

Remove in hashset has the same problem as mentioned in part 1. Just reverse. The amount of allocated memory decreases dynamically. While hashset does this, it is possible that remove will attempt to remove on an empty array. Thus the value is not removed once the hashset is repopulated

3

We add synchronize on add, remove, and size methods. Utilizing synchronize, we can ensure that Java locks the methods and that only one class at a time can access said methods. When one class has access, hashset has enough time to resize itself dynamically. Thus values are repopulated before we insert or remove the next value.

4

ConcurrentIntegerSetLibrary utilises ConcurrentSkipListSet which is thread safe. Thus interleaving cannot happen.[1]

5 & 6

Failed tests proves the presence of bugs. But successful tests do not prove lack of bugs. We cannot use test to prove that our class is thread safe as there might be other things that can cause the thread to be unsafe. As well, tests are only as good as the person writing them and human mistakes happen, or a test might not cover every single edge case. Thus, failing tests can show us that there are bugs in the system, but if a test succeeds – it does not mean that our code is bug free / thread safe.

[1] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html`