Assignments for week 3

Anders Degn Lapiki, Jacob Kjærulff Furberg, Jonas Ishøj Nielsen

03/09/2020

Contents

1 3-1 Bounded buffer 2
1.1 3.1.1 - green 2
1.2 3.1.2 - green 2
1.3 3.1.3 - yellow 2
1.4 3.1.4 - red 2
1.5 3.1.5 - red 2

2 3.2 count primes, TestCountFactors.java 3
2.1 3.2.1 - green 3
2.2 3.2.2 - Green 3
2.3 3.2.3 - green 3
2.4 3.2.4 - green 3
2.5 3.2.5 - yellow 3

3 3.3 histogram 3
3.1 3.3.1 - Green 3
3.2 3.3.2 - Green 3
3.3 3.3.3 - yellow 4
3.4 3.3.4 - yellow 4
3.5 3.3.5 - yellow 4
3.6 3.3.6 - red 4
3.7 3.4.1 - red 5
3.8 3.4.2 - red 5
3.9 3.4.3 - red 5

Course: Practical Concurrent and Parallel Programming, MSc CS (Autumn 2020)
KSPRCPP2KU

1 3-1 Bounded buffer

1.1 3.1.1 - green

The solution along with tests can be seen in `oneDotOneDotBoundedBuffer`.
We choose the thread safe collection synchronized list to store the elements and a semaphore to add the blocking bounded functionality. We chose to thread safe collection because multiple threads need to access it, meaning the list needs to have some kind of synchronization policy.

1.2 3.1.2 - green

We chose semaphores. This is mainly because of the example from the book in listing 5.14. The BoundedHashSet class is very similar to what we are trying the achieve. Semaphore here is a good choice, since we can use it to keep track of missing elements in the buffer. When reaching 0 the semaphore will block threads from acquiring until another thread has released.

1.3 3.1.3 - yellow

The solution along with tests can be seen in `oneDotOneDotBoundedBuffer`.
We use the standard factory pattern for the constructor, to avoid early publication.

1.4 3.1.4 - red

The FAIR parameter is a boolean which if true means that waiting threads are picked with a FIFO ordering. It is useful for example in a context where n threads each do x tasks and only m<n tasks can be done concurrently. Fair can be set to true to avoid threads only being allowed to start when all others have finished. For this reason we choose to set the Fair parameter to TRUE.

1.5 3.1.5 - red

Count of how many times it has been blocked is stored in 2 atomic longs. The first is incremented when insert is blocked, and the second when take is blocked. The count for take increments when remove is called and list is empty.
The count for how many times insert is blocked increments when availablePermits is 0 before trying to acquire. An alternative is our implementation with parameter 'waiting' set to false, where it increments if tryAcquire returns false. This second implementation will return a much higher value as it will continue to try until it succeeds, whereas the normal acquire only tries once.

2 3.2 count primes, TestCountFactors.java

2.1 3.2.1 - green

For range = 5.000.000 it took 12.059717100 seconds.

2.2 3.2.2 - Green

Written in `TestCountFactors.java`. We implemented MyAtomicInteger using synchronization. We discussed if using a binary semaphore would work, but we concluded that this would be the same as using the synchronized keyword.

2.3 3.2.3 - green

The solution still computes the correct result and time is reduced to 2.7194305 seconds.

2.4 3.2.4 - green

MyAtomicInteger can not be implemented without synchronization and only a volatile field. They keyword volatile will only be sufficient for the get method, since it only ensures visibility. It will therefore not be sufficient for the setAndGet.

2.5 3.2.5 - yellow

There is no noticeable change in the time it takes to execute. The field is declared final to make it immutable, which doesn't affect result as the field is thread confined.

3 3.3 histogram

3.1 3.3.1 - Green

The field counts became final. The increment, getCount and getTotal are the only functions that needs to be synchronized to ensure atomicity and visibility of total. The getCount was synchronized to ensure visibility.
GetSpan method isn't synchronized because counts is final and thus immutable.

3.2 3.3.2 - Green

```
0:         2
1:    348513
2:    979274
3:   1232881
4:   1015979
```

```
   5:    660254
   6:    374791
   7:    197039
   8:     98949
   9:     48400
  10:     23251
  11:     11019
  12:      5199
  13:      2403
  14:      1124
  15:       510
  16:       233
  17:       102
  18:        45
  19:        21
  20:         7
  21:         3
  22:         1
```

3.3 3.3.3 - yellow

We can remove all synchronized keywords by making the int array into an atomicInteger array. The total field was also made into an atomicInteger. After these changes using Histogram3 gives the same result as Histogram2

3.4 3.3.4 - yellow

Total is still an atomic integer, but counts is now a AtomicIntegerArray. After these changes using Histogram4 gives the same result as Histogram2

3.5 3.3.5 - yellow

Snapshots aren't affected by future increments, and thus doesn't endanger thread-safety.

- Histogram1: live view - return array
- Histogram2: snapshot - copy array in synchronized function
- Histogram3: snapshot - copy array in non-synchronized function
- Histogram4: snapshot - copy array in non-synchronized function

3.6 3.3.6 - red

The LongAdder contains multiple "cells" of datatypes similar to an AtomicLong. The amount of cells a LongAdders cantains, is equal to the amount of cores on the machines CPU. All values in the cells combined represent the value of the LongAdder. If multiple threads try to manipulate the LongAdder value, they can work on their own cell, and therefore allowing concurrent access and manipulation. Which improves performance if many threads are requesting the same resources.

3.7 3.4.1 - red

Neither thread reaches main's calculation.

3.8 3.4.2 - red

The problem is, that the lock is held by the Long which is a pseudo primitive class. When using "++" to manipulate the data, java creates a new Lock object (with a new reference). Therefor the reference to the lock holding object changes as count changes.

3.9 3.4.3 - red

This can be solved by either create an object which will work as the lock, or locking on the class' reflective object.