Exercise 6.1

1. Changing the number of transactions changes the execution time in a proportional way to the transaction time
   in the example we have 49 extra transaction with 50ms of transaction time

```
Execution time low transaction      62541152,5 ns  280260,08          4
Execution time high transaction   3151432270,0 ns 34647896,03          2
```

2. Min and max are both necessary to avoid deadlock, in this way the order of acquiring the 2 locks is always the same, so no deadlock can happen.
   without them in this situation:
   t1(transfer(a,b))
   t2(transfer(b,a))
   if those threads run at the same time the 2 threads would get stuck with the second solution because t1 would lock a while t2 would lock b and they both will wait for one another to release the lock they got.
3. see code, with the print we can see that all threads are active.
4. see code, we shut down the pool only after waiting for all futures to arrive

Exercise 6.2

1.

```
countSequential              25042085,0 ns 1787884,98         16
countParallelN  1            25783095,6 ns 1649764,64         16
countParallelNLocal  1       25534258,1 ns 2186744,23         16
countParallelN  2            15368419,4 ns 1002215,09         32
countParallelNLocal  2       15877266,9 ns 1308906,85         16
countParallelN  3            11334962,8 ns  565117,97         32
countParallelNLocal  3       11754054,1 ns  914596,26         32
countParallelN  4            10374803,8 ns  939565,75         32
countParallelNLocal  4       10818771,9 ns 1068850,35         32
countParallelN  5             9537958,1 ns  250425,69         32
countParallelNLocal  5        9735455,3 ns  285901,78         32
countParallelN  6             9073134,4 ns 1170023,09         32
countParallelNLocal  6        8905104,7 ns  721698,79         32
countParallelN  7             7941704,4 ns  362712,44         32
countParallelNLocal  7        8718870,0 ns  675756,55         32
countParallelN  8             7915131,9 ns  323059,00         32
countParallelNLocal  8        7729986,3 ns  473294,11         64
countParallelN  9             7948170,9 ns  770679,22         64
countParallelNLocal  9        8709226,3 ns  806528,81         32
countParallelN 10             8414465,6 ns 1118634,00         32
countParallelNLocal 10        7864776,9 ns  497567,99         64
countParallelN 11             8289201,6 ns 1110011,22         32
countParallelNLocal 11        8098263,9 ns  972678,76         64
countParallelN 12             8263220,0 ns 1036368,72         32
countParallelNLocal 12        8355210,2 ns  784326,37         64
countParallelN 13             8110432,2 ns  455820,15         64
countParallelNLocal 13        8026380,6 ns  657715,72         32
countParallelN 14             7274170,5 ns  205924,18         64
countParallelNLocal 14        8473235,5 ns 1063132,96         64
countParallelN 15             7275818,9 ns  347009,15         64
countParallelNLocal 15        8094543,4 ns  889933,71         32
countParallelN 16             8563643,1 ns 1056427,77         32
countParallelNLocal 16        9264241,3 ns 1015120,71         32
```

   We can see that countParallelN initially improves a lot until it reaches 8 threads. Above 8 there is no longer such a difference, that is because the hardware we used to run the program has 8 CPUs.
   We can also notice that countParallelN 1 and countParallelNLocal 1 is slower that the sequential one, that's because of the overhead of creating and starting a thread.
   We noticed something unexpected: the countParallelNLocal performs worse than countParallelN, we expected the opposite since more accesses to the same shared variable are needed in the countParallelN.

2.

```
countSequential              24048208,1 ns 2034687,38          16
countParallelN  1            29579266,9 ns 3098168,29          16
countParallelNLocal  1       26331780,0 ns 3808903,09          16
countParallelN  2            17671170,6 ns 3032918,34          16
countParallelNLocal  2       15853601,6 ns 1314200,43          32
countParallelN  3            11710747,5 ns  712467,19          32
countParallelNLocal  3       11550599,7 ns  819759,37          32
countParallelN  4            10375557,2 ns  739271,59          32
countParallelNLocal  4        9684395,6 ns 1268880,96          32
countParallelN  5             9823319,1 ns  566827,73          32
countParallelNLocal  5        9560764,7 ns  302377,20          32
countParallelN  6             8503671,3 ns  171865,92          32
countParallelNLocal  6        8529694,1 ns  276059,26          32
countParallelN  7             8169442,8 ns  847621,78          64
countParallelNLocal  7        7784128,1 ns  133415,33          32
countParallelN  8             7358947,7 ns  279422,15          64
countParallelNLocal  8        7217139,8 ns  109568,75          64
countParallelN  9             7160320,8 ns  163965,56          64
countParallelNLocal  9        7124589,5 ns   97299,86          64
countParallelN 10             7280400,9 ns  228216,90          64
countParallelNLocal 10        7242953,9 ns  182899,22          64
countParallelN 11             7339524,2 ns  226343,74          64
countParallelNLocal 11        7950156,9 ns  846497,27          32
countParallelN 12             8436368,1 ns 1715831,03          64
countParallelNLocal 12        8144254,1 ns  949924,04          32
countParallelN 13             7399579,1 ns  178586,31          64
countParallelNLocal 13        7318348,6 ns  293618,11          64
countParallelN 14             7305887,3 ns  431018,53          64
countParallelNLocal 14        7224492,0 ns  257230,81          64
countParallelN 15             7030589,7 ns  218390,09          64
countParallelNLocal 15        7055915,5 ns  274765,94          64
countParallelN 16             7036966,9 ns  297177,90          64
countParallelNLocal 16        7244491,3 ns  527266,60          64
```

See TestCountPrimesThreadsFuture.java for implementation with Future.
As we expect we can see that when the number of threads exceeds the number of cores the execution time does not increase as in the previous case. The performances in this case are better compared to the version with threads; this is because if a pool of executors is used threads are reused to perform more tasks, reducing the overhead of creating and destroying threads.

Exercise 6.3

1. see Histogram2 for the implementation. We use the synchronized modifier on increment getCount and getPercentage to ensure one thread can access histogram data at time.
   getSpan does not need synchronization since we are told that the number of span is a constant value
2. see Histogram3.java for implementation
3. see HistogramPrimesThreads.java for implementation
4.

```
> Task :app:run
HistogramPrimesThreads with 8 thread and1 locks    4051550220,0 ns 255988309,52          2
HistogramPrimesThreads with 8 thread and2 locks    3855917485,0 ns 174422401,58          2
HistogramPrimesThreads with 8 thread and3 locks    3842269515,0 ns 197041132,56          2
HistogramPrimesThreads with 8 thread and4 locks    3867901460,0 ns 195471330,44          2
HistogramPrimesThreads with 8 thread and5 locks    3645858940,0 ns 172346001,04          2
HistogramPrimesThreads with 8 thread and6 locks    3737202350,0 ns 308111952,50          2
HistogramPrimesThreads with 8 thread and7 locks    3665925865,0 ns 116658981,59          2
```

As we expected we can see an improvement using lock stripping with an increasing number of locks since if we use more locks it's less probable that we have to wait to acquire the lock for the bean we need.