PCPP - Assignment 2

Dallas Trent Maxwell (dalm@itu.dk)
Martin Conradsen (conr@itu.dk)
Mohammad Toheed Sharif (mosh@itu.dk)

September 20, 2021

Exercise 3.1

The results are included in the files *ex1_out_mac.txt* and *ex1_out_mac_m1.txt*. We did the experiments on two different MacBooks, one of which is equipped with the Apple M1 chip, resulting in significantly faster execution of the programs. The computer with the M1 chip will be referred to as *M1*, while the other will be referred to as *M2*.

The results for Mark1 are quite implausible; despite M1 being significantly faster, M2 achieves a faster time (0,004 vs. 0,006 seconds). The results for Mark2 through Mark6 do seem plausible, however. M1 consistently achieves a time of about 2,7 ns after a few runs, while M2 hovers around a time of about 20 ns. Both machines achieve faster times than the ones presented in the *Microbenchmarks* note, with the M1 machine being much faster in all cases, which is unsurprising.

For Mark7, the results are quite inconsistent. None of the two machines were consistently faster than the other, with M1 computing e.g. `pow` and `exp` faster than M2, while M2 computed `log` and `atan` faster than M1. The inconsistence in these results are quite surprising, and we believed that M1 would be faster in all cases. Again, all of the times are faster than the ones in the *Microbenchmarks* note, but the variance between the different running times are very similar.

Exercise 3.2

1. For `hashCode()`, the results are quite strange: the time measurements are faster between 2 and 8 threads, but with 16 threads, the execution suddenly becomes 4 times slower than with 8 threads, and than even slower with 32 threads. After that, however, the measurements are almost always faster, until they reach about 3,5 ns with 131072 and more threads. `Point creation`, `Thread's create` and `Uncontended lock` behave in much the same way. With `Thread's work`, the results are almost always faster with more threads, until we reach 32 threads, after which not a lot of improvement is seen. The time measurements for `Thread create start` are seemingly random, while the ones for `Thread create start join` behave similarly to `Thread's work`, where not a lot of improvement is seen after reaching 32 threads.

2. The results are included in the file *ex2_out.txt*. They are plausible and are all very similar to the last times reported when running Mark6. Most of the time measurements are similar to the ones shown in the lecture, with some of them being faster. The time for running `Thread create start`, however, is much slower than the one shown in the lecture (114132,3 ns on our machine vs. 47000 ns in the lecture), which is surprising.

Exercise 3.3

Mandatory

1. The results are included in the file *ex3_out.txt*.

2. The results seem plausible, since running the program with 1 thread results in a similar time measurement as running `countSequential`, and using 2 threads is much faster. However, after reaching 3 threads, the results are seemingly random with a number of threads between 4 and 32. It does not seem like there is any relation between the number of cores on the system and the performance of the threads, since the system has 8 cores, but the results with 5 vs. 8 threads are very similar. That is surprising, and we would expect the performance to improve until reaching 8 threads, after which we would expect the measurements to be much the same.

3. The results are included in the file *ex3_out_atomiclong.txt*. The measurements for AtomicLong are similar to the ones before until we reach 6 threads, after which the results are consistently faster until we reach 9 threads. With 10 and up to 14 threads, the results are still faster than with LongCounter, while 15 and more threads produce very similar results as those with LongCounter.

Generally, one should use built-in classes and methods when they exist, since these are most likely optimized for the programming language, and chosen by the developers to be the most efficient among the alternatives.

Challenging

4. With the changes, the results are still very similar to those from using AtomicLong, but the improvement is very insignificant on our hardware. We achieved the fastest time yet with 10 threads after doing the changes, which is surprising, since we wouldn't expect the results to improve after 8 threads on an 8-core system.

Exercise 3.4

The results are included in the file *ex4_out.txt*. Incrementing a normal `int` takes around 0,9 ns, while incrementing a `volatile int` takes around 7,0 ns. The `volatile int` being slower to increment is plausible and unsurprising, since it's flushed to main memory, while the normal `int` stays in the registers, where the value is much faster to fetch.

Exercise 4.1

Mandatory

1. See *BoundedBuffer.java* for implementation.

2. The class is thread-safe because we ensure that no race conditions happen during execution of method calls or field access, and because the internal list is never published. Anything that happens before calling a `release()`-method happens-before anything that happens after a successful `acquire()`-call.

3. We do not believe that it is possible to achieve the same functionality using Barriers. This is due to not being able to distinguish between a publisher and consumer for the barrier to break down.

Exercise 4.2

1. See *Person.java* for implementation.

2. Our implementation of the `Person` class is thread safe because (1) all the `get`-methods and the `changeAddress`-method are synchronized, which means that a thread is not able to read the wrong value or zip code (in the case of thread 1 reading the address while thread 2 is currently changing it), and (2) because the variable `idCounter` that is in charge of handling the ID assigned to each Person is implemented as the thread-safe `AtomicLong` (as well as being static), which ensures that the variable is shared across all instances of Person, as well as being synchronized to prevent the variable from being changed and/or read by multiple threads at the same time.

3. See *Person.java* for implementation.

4. No. No amount of experiments can ensure that a class or other implementation is thread-safe.