PCPP **IT University, E2020**

Exercises week 5

Last update 2020-09-17

Goal of the exercises

The goals of these exercises supports the topics of chapter 6 and 8 in *Goetz*. That is, to give you practical experience in designing software that utilizes threads to make the overall system *efficient* and responsive.

Exercise 5.1 This exercise is based on the program ExecutorAccountExperiments.java. It generates a number of transactions to move money between accounts. Each transaction simulate transaction time by sleeping 50 milliseconds. The transactions are randomly generated, but ensures that the source and target accounts are not the same.

Green

1. Use the nanoSecond timer to verify that the time it takes to run the program is proportional to the transaction time. There is statement which should be removed from the code to get reasonable measurements.
2. Change the main program so two threads generate and execute transactions. This will potentially make the system incorrect. Where is the error located? *Hint: Goetz chap 10 introduces the idea of lock-ordering. You can use the ID of the account instead of hashCode() as he propose in listing 10.3.*
3. The number of threads one can create depends on the OS. How many are you able to make on your JVM? How do you find out?
4. On page 205[1] Goetz writes: "Whenever you see code of the form: new Thread(runnable).start() and you think you might at some point want a more flexible execution policy, seriously consider replacing it with the use of an Executor."

   Change the program to use a the executor framework instead of raw threads. Make it use a fixed size thread pool.
5. Is there a way to tell when the program finishes? Why/ Why not?

Yellow

6. Ensure that the executor shuts down after all tasks has been executed, and make the main thread print out how long it took from start of the executor to the executor has successfully shut down.
7. How many threads are needed in the fixed size thread pool to obtain best performance? Try to vary the sleep time from its current value of 50 down to 1, in steps 50,40,30,20,10,1. How

Red

8. On page 282 there is a formula for computing the optimal pool size. Does that make sense in this case where there is a sleep involved? Do the number fit with the results from the previous item?

Exercise 5.2 *Yellow*

Use the lecture's example in file TestCountPrimesTasks.java to count prime numbers using tasks and the executor framework instead of explicitly creating threads for each concurrent activity.

1. Using a CachedThreadPool as executor, measure the time consumption as a function of the number of tasks used to determine whether a given number is a prime. The Mark7 method is relevant.
2. Use a WorkStealingPool and repeat the experiments.

[1] From time to time I refer to pagenumbers. They might be off a bit, as I am getting them from my kindle.

3. Use Excel or gnuplot or Google Docs online or some other charting package to make graphs of the execution time as a function of the number of tasks. Do this for the executors you have tried.
4. Reflect and comment on the results; are they plausible? How do they compare with the performance results obtained using explicit threads in last week's exercise? Is there any reasonable relation between the number of threads and the number of cores in the computer you ran the benchmarks on? Any surprises?

Red

5. Java 8 has a class java.util.concurrent.atomic.LongAdder that is a kind of AtomicLong. It may perform much better than LongCounter and AtomicLong when many threads add to it often and its actual value is inspected only rarely. Also, it should perform well in case locking is especially slow on the hardware being used. Replace the use of LongCounter with LongAdder, and rerun the measurements.

Exercise 5.3 This exercise is about fetching a bunch of web pages. This activity is heavy on input-output and the latency (delay) involved in requests and responses sent over a network. In contrast to the previous exercises, fetching a webpage does not involve much computation; it is an input-output bound activity rather than a CPU-bound activity.

File TestDownload.java contains a declaration of a method `getPage(url, maxLines)` that fetches at most `maxLines` lines of the body of the webpage at `url`, or throws an exception.

Yellow

1. First, run that code to fetch and print the first 10 lines of `www.wikipedia.org` to see that the code and net connection works.
2. Now write a sequential method `getPages(urls, maxLines)` that given an array `urls` of webpage URLs fetches up to `maxLines` lines of each of those webpages, and returns the result as a map from URL to the text retrieved. Concretely, the result may be an immutable collection of type Map<String,String>. You should use neither tasks nor threads at this point.

   File TestDownload.java contains such a list of URLs; you may remove unresponsive ones and add any others you fancy.

   Call `getPages` on a list of URLs to get at most 200 lines from each, and for each result print the URL and the number of characters in the corresponding body text — the latter is a sanity check to see that something was fetched at all.
3. Use the Timer class — **not** the Mark6 or Mark7 methods — for simple wall-clock measurement (described in the *Microbenchmarks* lecture note from week 4) to measure and print the time it takes to fetch these URLs sequentially. Do not include the time it takes to print the length of the webpages.

   Perform this measurement five times and report the numbers. Expect the times to vary a lot due to fluctuations in network traffic, webserver loads, and many other factors. In particular, the very first run may be slow because the DNS nameserver needs to map names to IP numbers.

   (In principle, you could use Mark6 or Mark7 from the *Microbenchmarks* note to more accurately measure the time to load webpages, but this is probably a bad idea. It would run the same web requests many times, and this might be regarded as a denial-of-service attack by the websites, which could then block requests from your network).
4. Now create a new parallel version `getPagesParallel` of the `getPages` method that creates a separate task (not thread) for each page to fetch. It should submit the tasks to an executor and then wait for all of them to complete, then return the result as a Map<String,String> as before.

   The advantage of this approach is that many webpage fetches can proceed in parallel, even on a single-core computer, because the webservers out there work in parallel. In the old sequential version, the first request will have to fully complete before the second request is even initiated, and so on; meanwhile the CPU and the network sits mostly idle, wasting much time.

Red

5. Are you able to show (by timing or other means) a difference in performance between a WorkStealingPool, a CachedThreadPool, or a FixedSizePool. Make sure that the executor is allocated only once, for instance as a static field in the class; do not allocate a fresh executor for each call to `getPagesParallel`.

   Call it as in the previous questions, measure the wall-clock time it takes to complete, and print that and the list of page lengths. Repeat the measurement five times and compare the results with the sequential version. Discuss results, in particular, why is fetching 23 webpages in parallel not 23 times faster than fetching them one by one?