Exercises week 9

Last update 2024/10/23

Exercise 9.1 This exercise is based on the program `AccountExperiments.java` (in the exercises directory for week 9). 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.

Mandatory

1. Use Mark7 (from Benchmark.java in the `benchmarking` package) to measure the execution time and verify that the time it takes to run the program is proportional to the number of transactions: `NO_TRANSACTION`.

2. Now consider the version in `ThreadsAccountExperimentsMany.java` (in the directory `exercises09`).

   Consider these four lines of the `transfer`:

   ```
   Account min = accounts[Math.min(source.id, target.id)];
   Account max = accounts[Math.max(source.id, target.id)];
   synchronized(min){
     synchronized(max){
   ```

   Explain why the calculation of `min` and `max` are necessary? Eg. what could happen if the code was written like this:

   ```
   Account s= accounts[source.id];
   Account t = accounts[target.id];
   synchronized(s){
     synchronized(t){
   ```

   Run the program with both versions of the code shown above and explain the results of doing this.

3. Change the program in `ThreadsAccountExperimentsMany.java` to use a the executor framework instead of raw threads. Make it use a ForkJoin thread pool. For now do not worry about terminating the main thread, but insert a print statement in the `doTransaction` method, so you can see that all executors are active.

4. Ensure that the executor shuts down after all tasks has been executed.

   Hint: See slides for suggestions on how to wait until all tasks are finished.

   .

Challenging

5. Use Mark8Setup to measure the execution time of the solution that ensures termination.

   Hint: Be inspired by `PoolSortingBenchmarkable.java` (in the `code-lecture` directory for week 9)

Exercise 9.2 In the lecture it was shown how to estimate the maximal speed-up for Quicksort using up-to 8 threads (processors/cores) and that this estimate was consistent with Amdahls law.

1. Use the same method to estimate the maximum speed-up for Quicksort on hardware with 16 and 32 cores.
2. Estimate the maximum speed-up for sorting 1.000.000 and 10.000.000 numbers with Quicksort on hardware with 16 and 32 cores.

Challenging

3. Can Amdahls law be used to estimatre the maximum speed-up for the prime-counting algorithm?

Exercise 9.3 Use the code in file `TestCountPrimesThreads.java` (in the exercises directory for week 9) to count prime numbers using threads.

Mandatory

1. Report and comment on the results you get from running `TestCountPrimesThreads.java`.
2. Rewrite `TestCountPrimesThreads.java` using Futures for the tasks of each of the threads in part 1. Run your solutions and report results. How do they compare with the results from the version using threads?

Exercise 9.4 This exercise is a continuation of the histogram exercise in week05 (exercise 5.1).

Mandatory

1. Use the benchmarking tools introduced in week08 to compare the running times for `CasHistogram` and a monitor implementation (see file `HistogramLocks` in the code-exercises folder for Week 9).

   You may use the skeleton in the file `...exercise94/TestCASLockHistogram` in the code-exercises folder for Week 9).

   What implementation performs better? The monitor implementation or the CAS-based one?

   Is the result you got expected? Explain why.