Exercises week 6

Last update 2023/09/29

Exercise 6.1 This exercise is based on the program `AccountExperiments.java` (in the exercises directory for week 6). 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 transaction time.
2. Now consider the version in `ThreadsAccountExperimentsMany.java` (in the directory `exercise61`).

   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 6)

Exercise 6.2 Use the code in file `TestCountPrimesThreads.java` (in the exercises directory for week 6) 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 6.3 A histogram is a collection of bins, each of which is an integer count. The span of the histogram is the number of bins. In the problems below a span of 30 will be sufficient; in that case the bins are numbered 0...29.

Consider this Histogram interface for creating histograms:

```
interface Histogram {
  public void increment(int bin);
  public float getPercentage(int bin);
  public int getSpan();
}
```

Method call `increment(7)` will add one to bin 7; method call `getCount(7)` will return the current count in bin 7;method call `getPercentage(7)` will return the current percentage of total in bin 7; method `getSpan()` will return the number of bins; method call `getTotal()` will return the current total of all bins.

There is a non-thread-safe implementation of Histogram1 in file SimpleHistogram.java. You may assume that the `dump` method given there is called only when no other thread manipulates the histogram and therefore does not require locking, and that the span is fixed (immutable) for any given Histogram object.

Mandatory

1. Make a thread-safe implementation, class `Histogram2` implementing the interface `Histogram`. Use suitable modifiers (`final` and `synchronized`) in a copy of the `Histogram1` class. This class must use at most one lock to ensure mutual exclusion.

   Explain what fields and methods need modifiers and why. Does the `getSpan` method need to be synchronized?

2. Now create a new class, `Histogram3` (implementing the `Histogram` interface) that uses lock striping. You can start with a copy of `Histogram2`. Then, the constructor of `Histogram3` must take an additional parameter `nrLocks` which indicates the number of locks that the histogram uses. You will have to associate a lock to each bin. Note that, if the number of locks is less than the number of bins, you may use the same lock for more than one bin. Try to distribute locks evenly among bins; consider the modulo operation `%` for this task.

3. Now consider again counting the number of prime factors in a number `p`. Use the `Histogram2` class to write a program with multiple threads that counts how many numbers in the range 0...4 999 999 have 0 prime factors, how many have 1 prime factor, how many have 2 prime factors, and so on. You may draw inspiration from the `TestCountPrimesThreads.java`.

   The correct result should look like this:

   ```
     0:        2
     1:   348513
     2:   979274
     3:  1232881
     4:  1015979
     5:   660254
     6:   374791
     7:   197039
     8:    98949
     9:    48400
   ... and so on
   ```

   showing that 348 513 numbers in 0...4 999 999 have 1 prime factor (those are the prime numbers), 979 274 numbers have 2 prime factors, and so on. (The 2 numbers that have 0 prime factors are 0 and 1). And of course the numbers in the second column should add up to 5 000 000.

   Hint: There is a class `HistogramPrimesThread.java` which you can use a starting point for this exercise. That class contains a method `countFactors(int p)` which returns the number of prime factors of `p`. This might be handy for the exercise.

.

4. Finally, evaluate the effect of lock striping on the performance of question 6.3. Create a new class where you use `Mark7` to measure the performance of `Histogram3` with increasing number of locks to compute the number of prime factors in 0. . . 4 999 999. Report your results and comment on them. Is there a increase or not? Why?