PF3: Assignment 3

Due on December 10, 2018 at 8:30am

Prof. Walter Binder

A. Romanelli

Problem 1

Assuming that we have a program which is parallelizable for the 90%, it means that our total time on a single processor will be:

$$T_1 = T_s + \frac{T_p}{n} = \frac{1}{10}T_1 + \frac{9}{10}T_1$$

1. Let's imagine that it takes 1s for the whole program to run, that means that $T_s = \frac{1}{10}s$ and $T_p = \frac{9}{10}s$, then if we scale this code up to 256 cores, we get:

$$T_{256} = T_s + \frac{T_p}{256} \Rightarrow T_{256} = \frac{1}{10}T_1 + \frac{9}{10}\frac{T_1}{256} = \frac{256T_1 + 9T_1}{2560} = \frac{265}{2560}T_1 = \frac{53}{512}T_1$$

This means that our final speedup w.r.t T_1 will be:

$$S_{256} = \frac{T_1}{T_{256}} = \frac{T_1}{\frac{53}{512}T_1} = \frac{512}{53} = 9.\overline{6603773584905}$$

If we further scale up the number of cores by a factor of 2 and 4 respectively we'll get:

$$T_{512} = \frac{512T_1 + 9T_1}{5120} = \frac{521}{5120}T_1$$

$$T_{1024} = \frac{1024T_1 + 9T_1}{10240} = \frac{1033}{10240}T_1$$

$$S_{512} = \frac{T_1}{T_{512}} = \frac{T_1}{\frac{521}{5120}T_1} = \frac{5120}{521} = 9.\overline{827255278310940499040307101727447216890595009596928 9}$$

$$S_{1024} = \frac{T_1}{T_{1024}} = \frac{T_1}{\frac{1033}{10240}T_1} = \frac{10240}{1033} \approx 9.912875121006776379477250726040658276863504356243949...\text{(period 1032)}$$

As the number of cores $n \to \infty$ the T_n is given by:

$$\lim_{n\to\infty}\left(T_s + \frac{T_p}{n}\right) = T_s$$

Thus the time needed to run the sequential part of the program will be the fastest possible time, which would be a speed up w.r.t T_1 of:

$$S_\infty = \frac{T_1}{T_\infty} = \frac{T_1}{\frac{1}{10}T_1} = 10$$

Problem 2

1. Check the attached code for the solution to this point.

2. In the case in which the array size is less than $2^{13} = 8192$, there's not going to be any substantial difference, since it will run very similarly as the `mergeSort` implementation. In the case in which the program has only one physical core at disposal, the parallelized version will result slower, as it will have an overhead due to the thread instantiation without gaining any of the benefits of concurrency programming.

Problem 3

1. (a) The first `synchronized` at line 15 is synchronized on the class "LogManager"
   (b) The second at line 20 is as well synchronized on the class "LogManager"
   (c) The third at line 38 is synchronized on the Map object `allSimpleLogs`
   (d) The fourth at line 56 is synchronized on the Map object `allSecurityLogs`
   (e) The fifth at line 62 is synchronized on the object invoking the instance method, instantiated from the class "SecurityLog"
   (f) The last at line 67 is syncrhonized again on the objected invoking the method, which is instantiated from the class "SecurityLog"

2. 1) The reason why the HashMap is still not thread safe is that, despite using the synchronization block, there are still methods which access or modify the map without synchronized blocks, thus it's thread unsafe.
   2) There are two ways to solve the problem:
      i. Wrap the parts of the program which interact with the HashMap object into synchronized blocks on the HashMap as well. This would work but it wouldn't be ideal as every time we utilise the HashMap we'd need to wrap the code into a synchronized block.
      ii. We could define a ConcurrentHashMap class with synchronized methods or just use the already defined class by importing it from the `java.util.concurrent` library.

3. 1) Just like in the previous example, this HashMap also fails to use the synchronization block each time a method is invoked upon it and thus it cannot be guaranteed to not throw a ConcurrentOperationException.
   2) The method is not thread safe for the HashMap because the method `SecurityLog.updateSecurityLogByID` is synchronised on the object on which is invoked, which is not the HashMap but rather an instantiation of class `SecurityLog`, thus this can be invoked by any thread whilst another thread is operating on the HashMap.
   3) Just like before, we can prevent this behaviour by wrapping the body of the methods `LogManager.getSecurityLogsByID` and `SecurityLog.updateSecurityLogID` into blocks synchronized on the `allSecurityLogs` HashMap object. Alternatively we could have used the `ConcurrentHashMap` class instead of `HashMap`.

4. The two methods cannot run concurrently because they will require the lock on the same object, that is the object on which they are invoked, an instantiation of the class "SecurityLog" and thus when invoked at the same time they will require the lock on the same object and consequently they will run sequentially.

5. Since we now the method `SecurityLog.addTextEntry` doesn't require the lock on the same object as `SecurityLog.updateSecurityLogByID`, they will be able to run concurrently.

Problem 4

Out of the 6 original synchronized blocks, given the introduction of `ConcurrentHashMap` and its thread safeness, we can remove all of the previous synchronizations exception made for the one applied on the method `SecurityLog.addTextEntry`, which still needs to be synchronized since ArrayList is not thread safe.

Problem 5

I will denote $T1$ and $T2$ as the two threads and $T1_1$ as the first line of the first thread.

(a) A possible result for x would be 2, which would be given by the following execution: $T2_1, T1_1, T1_2, T2_2$, whereas another execution would yield $x = 1$, in the case in which this was the program execution: $T1_1, T1_2, T2_1, T2_2$.
There are no other possible values for x wand the other (valid) permutations of the execution yield the same results.

(b) In this case the order of the operation does not influence the outcome since they perform the exact same operation and the operation is addition. Because of the commutativity of addition, the order in which we perform the operation doesn't influence the result. Thus our x would be incremented by 10 for each of the two iterations, thus yielding $x = 20$ for any possible execution combination.