

T-444-USTY

3. Febrúar 2017

Lab 3

Thread Assignment

Teacher: Kári Halldórsson

Reykjavík University

Computer Science

Spring 2017

Students:

Hrafnkell Ívarsson

Raquelita Rós Aguilar

# **Requirements**

The program was suppose to be able to do the following three requirements when running the function Solve.findAndPrintSolution() thirty times:

**Requirement 1**

,,Sequentially: Don’t run the next instance until the one before has returned”

**Requirement 2**

,,All at once in separate threads. A new thread is created for each instance. “

**Requirement 3**  
,,A certain number at a time. Threads are run through a thread pool of a certain size (students can try different sizes). New instances aren’t run until a thread is free in the thread pool.”

# **Given with assignment**

Five classes were given with this assignment as base program, named:

* Problem.java
* Problematic.java
* Solver.java
* ThreadAssignment01Main.java
* Tile.java

# **Implemented**

As suggested in the project description a class was created, SomeRunnableThing.java which created a new instance of Thread. Inside this class the function public void run() was implemented which is executed when the start() function is called with a some thread. For each requirement a for-loop was created were three different functions was suppose to run NUMBER\_OF\_PROBLEMS times.

## **Code snippet 1 – Requirement 1**

|  |
| --- |
| System.***out***.println("Requirement 1");  **for**(**int** i = 0; i < ***NUMBER\_OF\_PROBLEMS***; i++){  Solver.*findAndPrintSolution*(Problematic.*nextProblem*());  } |

Here you can see a code snippet for a requirement 1, where Solver.findAndPrintSolution takes in some problem, solves it and returns the solution. Below you can find sample output for this part which always gave us the same result when testing.

|  |
| --- |
| Requirement 1  Problem: (557,160) - (110,177) - length: 464 - iterations: 217040  Problem: (461,73) - (99,155) - length: 444 - iterations: 204752  Problem: (615,254) - (394,470) - length: 437 - iterations: 188783  Problem: (296,109) - (34,279) - length: 432 - iterations: 243092  Problem: (85,493) - (61,440) - length: 77 - iterations: 11811  Problem: (263,29) - (336,608) - length: 652 - iterations: 329093  Problem: (279,152) - (618,465) - length: 652 - iterations: 383761  Problem: (276,214) - (2,334) - length: 394 - iterations: 263061  Problem: (113,34) - (489,423) - length: 765 - iterations: 342223  Problem: (60,307) - (350,354) - length: 337 - iterations: 149614  Problem: (393,490) - (64,8) - length: 811 - iterations: 406964  Problem: (493,167) - (222,265) - length: 369 - iterations: 183582  Problem: (422,584) - (107,589) - length: 320 - iterations: 125153  Problem: (216,64) - (525,344) - length: 589 - iterations: 304957  Problem: (626,79) - (148,152) - length: 551 - iterations: 200691  Problem: (303,611) - (328,206) - length: 430 - iterations: 191063  Problem: (313,267) - (603,279) - length: 302 - iterations: 181243  Problem: (232,148) - (381,22) - length: 275 - iterations: 133651  Problem: (423,194) - (398,320) - length: 151 - iterations: 45554  Problem: (119,429) - (111,580) - length: 159 - iterations: 48948  Problem: (13,309) - (113,149) - length: 260 - iterations: 74511  Problem: (279,35) - (623,21) - length: 358 - iterations: 147055  Problem: (413,605) - (248,223) - length: 547 - iterations: 260947  Problem: (130,454) - (548,561) - length: 525 - iterations: 296899  Problem: (297,432) - (392,501) - length: 164 - iterations: 53931  Problem: (436,382) - (324,618) - length: 348 - iterations: 213150  Problem: (359,475) - (441,372) - length: 185 - iterations: 68236  Problem: (299,273) - (344,252) - length: 66 - iterations: 8754  Problem: (431,263) - (150,389) - length: 407 - iterations: 270185  Problem: (553,90) - (379,333) - length: 417 - iterations: 160783 |

## **Code snippet 2 – Requirement 2**

|  |
| --- |
| **System**.***out***.println("Requirement 2");  **Thread**[] **threads** = **new** **Thread**[***NUMBER\_OF\_PROBLEMS***];  **for**(**int** **i** = 0; i < ***NUMBER\_OF\_PROBLEMS***; i++){  threads[i] = **new** Thread(**new** SomeRunnableThing(**Problematic**.*nextProblem*()));  threads[i].start();  }  **for**(**int** **i** = 0; i < ***NUMBER\_OF\_PROBLEMS***; i++){ //join the threads to get running time  **try** { threads[i].join(); }  **catch** (**InterruptedException** **e**) { e.printStackTrace(); } |

In requirement 2, each problem has it’s own thread, so they are all solved at once. This causes the results to come in a slightly different order with each run, but the shorter problems are solved first and the longer last.  
Afterwards, the threads are joined so running time can be calculated correctly.  
Each thread is of the class SomeRunnableThing that takes in a problem and runs Solver.findAndPrintSolution() for it. The class is shown below.

|  |
| --- |
| **public** **class** **SomeRunnableThing** **implements** Runnable{  **Problem** problem;    ***@Override***  **public** **void** **run**() {  **Solver**.*findAndPrintSolution*(problem);  }    **public** **SomeRunnableThing**(**Problem** p) {  problem = p;  }  } |

## **Code snippet 3 – Requirement 3**

## **Running times**

**Method**

Each version of the program was run 5 times and running times noted.

All runs were done on the same computer.

This method does not produce a reliable average, but can give a good reference to compare the different methods with.

**System Information**

Operating System: Linux Mint 18 Cinnamon 64-bit

Linux Kernel: 4.4.0-21

Processor: Intel Core i5-6500 3.20GHz x 4

**Requirement 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | avg |
| 23197 ms | 23237 ms | 23173 ms | 23088 ms | 23227 ms | 23184 ms |

This method is very unefficient and has by far the longest running times. However, the running times seem to deviate very little from the average.

**Requirement 2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | avg |
| 4122 ms | 3408 ms | 3530 ms | 3435 ms | 3930 ms | 3685 ms |

An impressive improvement over requirement 1, more than 6 times faster on average than the method used there. The running times seem to deviate from the average a little bit more though.

**Requirement 3 (with POOL\_SIZE = 10)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | avg |
| 3759 ms | 3827 ms | 3624 ms | 3867 ms | 3870 ms | 3790 ms |

Similar to requirement 2 in speed, but deviates less from the average.  
When trying different values of POOL\_SIZE it seemed the higher the number the faster the runtime. Lower numbers, however, seemed to cause less deviation from average running time.