Grunnatriði stýrikerfa - dæmatímaverkefni 3

January 2016

English follows.

Afurðin

Verkefnið er að skrifa forrit í Java sem keyrir fallið Solver.findAndPrintSolution() 30 sinnum. Það gerir þetta á eftirfarandi vegu:

1. Í röð: ekki er kallað á tilvik fyrr en næsta á undan hefur skilað
2. Öll í einu í mismunandi þráðum. Nýr þráður er búinn til fyrir hvert tilvik
3. Ákveðinn fjölda í einu. Þræðir eru settir af stað gegnum thread pool af ákveðinni stærð (nemendur prófa sig áfram með stærð). Nýtt tilvik er ekki sett af stað fyrr en einhver þráður losnar í thread poolinu.

Að auki er skilað PDF skjali sem inniheldur þær viðbætur sem gerðar eru á grunnforritinu til að uppfylla atriði 1-3 í listanum á undan ásamt dæmi um úttak úr forritinu fyrir hvert atriði. Auk þess skuluð þið hafa í PDF skjalinu stutta lýsingu fyrir hvert atriða 1-3 á því hvernig og hvers vegna röð þess sem skrifað er út breytist milli aðferða, einnig í atriði 3 hvaða áhrif mismunandi fjöldi samhliða þráða hefur.

Forritstextinn sem þarf að fara í skjalið er það sem línunni
"Solver.findAndPrintSolution(Problematic.nextProblem());"
í grunnforritinu er skipt út fyrir, ásamt öllum þeim hjálparföllum sem skrifuð eru.

Að sækja og skila verkefni

Verkefnið er inni á myschool. Grunnforrit í Java er þar í ZIP skrá. Afpakkið verkefninu og opnið. Í eclipse gerið þið File-import-general-existing projects into workspace. Veljið síðan möppuna eða yfirmöppu hennar og importið verkefninu. Passið vel upp á að gera refactor-rename á projectið til að gefa því ykkar eigin nöfn.

Þegar þið eruð búin að vinna verkefnið, þjappið möppunni þá aftur og skilið inn í myschool skilakerfið ásamt PDS skjalinu með niðurstöðunum.

Verkefnið mega nemendur vinna tveir og tveir saman.

Vinnufyrirkomulag

Látið línuna **"findAndPrint..."** keyra **NUMBER_OF_PROBLEMS** sinnum.

Thread klasann er hægt að nota til þess að setja keyrslu af stað án þess að bíða eftir að henni ljúki. Nýtt tilvik af **Thread** er búið til með:

new Thread(Runnable runnable)

sem þýðir að þið þurfið að útfæra Interfaceið **Runnable**. Þið getið búið til sér klasa

class SomeRunnableThing implements Runnable

og útfært síðan fallið **public void run()** í honum. Fallið **run()** er það sem verður keyrt þegar kallað er á **someThread.start()** ef þráðurinn someThread var búinn til svona:

Thread someThread = new Thread(new SomeRunnableThing);

Það er líka hægt að búa þetta allt til bara þar sem þráðurinn er sendur af stað án þess að halda utan um neinar breytur, svona:

- new Thread(new Runnable() {
  - @Override
  - public void run() {
    - doStuff();
  - }
- }).start()

Hér var enginn klasi SomeRunnableThing búinn til, heldur er **run()** fallið overrideað "on the fly".

Það er meira að segja hægt að láta **run()** fallið tala við breytur í því scope sem þráðurinn er búinn til í, svo fremi þær breytur séu skilgreindar **final** og að þeim verði ekki breytt á meðan á keyrslu þráðarins stendur. T.d.

- void someFunction() {
  - final int myVariable = getSomeInt();
  - new Thread(new Runnable() {
    - @Override
    - public void run() {
      - doStuffWithVariable(myVariable);
    - }
  - }).start()
- }

Látið núna forritið ykkar setja af stað nýjan þráð fyrir hvert skipti sem á að kalla á **"findAndPrint..."**. Passið mjög vel upp á að örugglega sé kallað á sömu tilvikin af **Problem**. Það er ekki víst að niðurstöðurnar skrifist út í réttri röð, en passið mjög vel upp á t.d. að ekki séu fleiri en einn þráður að kalla á **Problematic.nextProblem()** í einu. Þá gæti random generatorinn skilað gildum í annarri röð en hann átti að gera. Kannski er betra að kalla á **Problematic.nextProblem()** utan þráðanna og koma upplýsingum um vandamálið inn í þræðina eða runnable-in. Mögulega er gott hérna að búa til hjálparfall sem skilar Runnable fyrir hvern þráð. Vinnið í þessu þangað til allar sömu lausnirnar skrifast út og þegar allt var keyrt í röð. Skoðið röðina sem úttakið skrifast út í. Er eitthvað hægt að lesa út úr því?

Í stað þess að setja af stað nýjan þráð fyrir hverja einustu keyrslu er hægt að halda utan um ákveðinn fjölda samhliða þráð í svokölluðu **thread pool**. Góð leið til að búa til thread pool er að nota aðgerðir sem boðið er upp á í Java pakkanum **java.util.concurrent**, t.d. **ExecutorService**, sem má búa til svona:

ExecutorService threadPool = Executors.newFixedThreadPool(POOL_SIZE);

þar sem **POOL_SIZE** er sá fjöldi þráða sem þið viljið að keyri samtímis.

Eftir þetta má kalla annað hvort á fallið **threadPool.execute(Runnable)** eða **threadPool.submit(Runnable)**. Submit skilar Future, sem hægt er að láta skila þegar þráðurinn hefur lokið keyrslu, en ef það er ekki ástæða til að bíða eftir hverjum þræði fyrir sig, heldur bara því að allir þræðir hafi klárað má gera eftirfarandi:

- try {
  - threadPool.shutdown();
  - threadPool.awaitTermination(5, TimeUnit.MINUTES);
- } catch (InterruptedException e) {
  - e.printStackTrace();
- }

Þarna má líka breyta tímanum sem beðið er, en almennt ætti hann að vera nógu langur til að bíða eftir öllum eðlilegum keyrslum. fallið **threadPool.shutdown()**, slekkur ekki á þráðunum, heldur setur af stað þessa bið sem bíður í þessu tilfelli í 5 mínútur áður en þræðirnir eru stöðvaðir.

Prófið ykkur nú áfram með magn samhliða þráða og sjáið hvort og þá hvaða áhrif talan hefur. Þegar allar þessar þrjár aðferðir eru örugglega að skila lausnum við sömu vandamálum, og sömu stærð lausna þá er verkefninu lokið.

Lesið vel lýsingar á öllum föllum sem þið notið. Hvenær sem þið eruð ekki viss um hvað skal gera næst, finnið leið til að skrifa út eitthvað sem gæti hjálpað ykkur, t.d. innihald breyta. Notið líka debuggerinn! Geriði ykkar besta til að fylgja þessum vísbendingum og klára verkefnið á þann hátt.

Góða skemmtun!

The Product

Write a program in Java that runs the function Solver.findAndPrintSolution() 30 times. It will do it in the following three ways:

1. Sequentially: Don't run the next instance until the one before has returned
2. All at once in separate threads. A new thread is created for each instance.
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.

You must also return a PDF document containing the additions you made to the base program to implement parts 1-3 above as well as a sample output for each part. The PDF should also include a short description for each of the parts of how and why the order of solutions written out changes. Also include a little in part 3 on what effect the number of concurrent threads had.

The program code needed in the PDF document is what the line:

"Solver.findAndPrintSolution(Problematic.nextProblem());"

in the base program is replaced with, as well as any helper functions that are written.

Getting and returning the assignment

The project is on myschool. There is a ZIP archive with the base program. Unzip it and open. In Eclipse you can do File-import-general-existing projects into workspace. Then pick the folder and import the project. Make sure you do refacter-rename on the project to change it to your own names.

Once you have finished the project, re-archive the folder and return it into the myschool project system, along with the result PDF.

Students can work on this assignment in groups of up to two.

Process

Run the line **"findAndPrint..." NUMBER_OF_PROBLEMS** times.

The **Thread** class can be used to run a function without waiting for it to finish. A new instance of **Thread** is made so::

new Thread(Runnable runnable)

which means that you have to implement the Interface **Runnable**. You can make a separate class

class SomeRunnableThing implements Runnable

and then implement the function **public void run()** in it. The function **run()** is what will be executed when you call **someThread.start()** if the thread someThread was made so:

Thread someThread = new Thread(new SomeRunnableThing);

You can also make all this where the thread is started without keeping track of classes or variables, like so:

- new Thread(new Runnable() {
  - @Override
  - public void run() {
    - doStuff();
  - }
- }).start()

Here we made no class SomeRunnableThing , but instead overrode the **run()** function on the fly.

You can even have your **run()** function use variables in the scope that the Thread was started in, as long as those variables are declared **final** and that they can not be changed in any way after the thread is started. Example:

- void someFunction() {
  - final int myVariable = getSomeInt();
  - new Thread(new Runnable() {
    - @Override
    - public void run() {
      - doStuffWithVariable(myVariable);
    - }
  - }).start()
- }

Now have your program start a new thread for every single instance of **"findAndPrint..."**. Make sure that you are definitely calling with the same instances of **Problem**. You can't be sure that the results are printed in the same order, but make sure that no two threads are calling **Problematic.nextProblem()** at the same time. Then the random generator could return values in a different order from that intended. It may be better to call **Problematic.nextProblem()** outside the threads and get the information on the problem into the threads or runnables, possibly by using a helper function that creates a Runnable for each thread. Work on it until all the same problems and results are being printed as when everything was run sequentially. Look at the order in which the results are printed. How can you explain this order?

Instead of creating a separate thread for every single execution, you can keep track of a specific number of concurrent threads in a **thread pool**. A good way to make a thread pool is using classes and operations available in the **java.util.concurrent** libraries, such as **ExecutorService**, which can be made like this:

ExecutorService threadPool = Executors.newFixedThreadPool(POOL_SIZE);
where **POOL_SIZE** is the number of threads you wish to run at a time.

Next you can either call the function **threadPool.execute(Runnable)** or **threadPool.submit(Runnable)**. Submit returns a Future, which can be made to return when the thread has finished, but if there is no reason to wait for each thread individually, but only for the entire set to have finished you can do the following:

- try {
  - threadPool.shutdown();
  - threadPool.awaitTermination(5, TimeUnit.MINUTES);
- } catch (InterruptedException e) {
  - e.printStackTrace();
- }

You can also change the waiting time, but it should in general be long enough for normal execution to finish. The function **threadPool.shutdown()**, does not immediately shut down the threads, but starts a wait that, in this case, waits five minutes before shutting the down.

Try different numbers of concurrent threads and see what effect the number has. When all these three methods of running the solver on multiple problems are definitely returning solutions for the same problems and finishing properly, this assignment is over.

Read well the descriptions of all functions you use. Whenever you're not sure what to do next, find a way to print something to the screen that might help you, for example contents of variables. Also use the debugger! Follow those leads and finish this assignment through exploring.

Have a good time!