Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 3: Multi-threading
Spring Semester 2024

Assigned on: **06.03.2024**

Due by: **(Wednesday Exercise) 11.03.2024**
(Friday Exercise) 13.03.2024

Overview

This week's assignment is about multi-threaded programs in Java.

- Download the ZIP file named `assignment3.zip` on Moodle.
- Import the project in Eclipse: Click on *File* in the top-menu, then select *Import*. In the dialog, select *Existing Projects into Workspace* under the *General* directory, then click on Next. In the new dialog, select the radiobox in front of *Select archive file* to import a ZIP file. Then, click Browse on the right side of the text-box to select the ZIP file you just downloaded on Moodle (`assignment3.zip`). After that, you should see assignment3 as a project under *Projects*. Click Finish.
- If you have done everything correctly, you should now have a project named assignment3 in your *Package Explorer*.

Task 1 – Parallel Counting

Description: In this exercise we will implement a `Counter` that allows counting the number of times a given event occurred. We will start with the following interface of the `Counter`:

```
public interface Counter {
    public void increment();
    public int value();
}
```

As can be seen, the counter has only two methods – `increment` that increases the `Counter` value by one and `value` that returns the current value of the `Counter`. We will use the `Counter` to monitor the progress of executing multiple threads that perform the following loop:

```
for (int i = 0; i < numIterations; i++) {
  ... // perform some work

  counter.increment();
}
```

For simplicity, in this assignment no actual work will be performed and each thread will only increment the counter `numIterations` times. The implementation of the above loop is already provided to you in the `NativeThreadCounter` class. In what follows we will study several different approaches that implement the `Counter` such that it can be safely used by multiple threads.

JUnit Tests: We provide JUnit tests to check the basic functionality of your solution.

Notice: Java libraries not included in the assignment project are not allowed. Do not rename any of the provided methods in the assignment (you can create additional classes and methods).

Tasks

A) To start with, implement a sequential version of the `Counter` in `SequentialCounter` class that does not use any synchronization. That is, the counter simply increments an integer value by one. We already provide code in `taskASequential` method that runs a single thread that increments the counter. Inspect the code and understand how it works. Verify that the `SequentialCounter` works properly when used with a single thread (the test `testSequentialCounter` should pass). Now run the code in `taskAParallel` which creates several threads that all try to increment the counter at the same time. Notice how the expected value of counter at the end of the execution is not what we would expect. Discuss why this is the case.

B) To fix this issue, implement a different thread safe version of the `Counter` in `SynchronizedCounter`. In this version use the standard primitive type **`int`** but synchronize the access to the variable by inserting **`synchronized`** blocks. Run the code in `taskB`.

C) Whenever the `Counter` is incremented, keep track which thread performed the increment (you can print out the thread-id to the console). Can you see a pattern in how the threads are scheduled? Discuss what might be the reason for this behaviour.

D) Implement a `FairThreadCounter` that ensures that different threads increment the `Counter` in an round-robin fashion. In round-robin scheduling the threads perform the increments in circular order. That is, two threads with ids 1 and 2 would increment the value in the following order $1, 2, 1, 2, 1, 2$, etc. You should implement the scheduling using the `wait` and `notify` methods. Can you think of an implementation that does not use `wait` and `notify` methods? Extend your implemenation to work with arbitrary number of threads (instead of only 2) that increment the counter in round-robin fashion.

E) Implement a thread safe version of the `Counter` in `AtomicCounter`. In this version we will use an implementation of the **`int`** primitive value, called `AtomicInteger`*, that can be safely used from multiple threads. Run the code in `taskE` that should now produce correct results even with multiple threads.

F) Compare the `AtomicCounter` and `SynchronizedCounter` implementations by measuring which one is faster. Observe the differences in the CPU load between the two versions. Can you explain what is the cause of different performance characteristics?

G) Implement a thread that measures the execution progress. That is, create a thread that observes the values of the `Counter` during the execution and prints them to the console. Make sure that the thread is properly terminated once all the work is done.

Submission

You need to submit your code to the Git repository. You will find detailed instructions on how to install and set-up Eclipse for use with Git in Exercise 1.

Once you have completed the skeleton, commit it to Git by following the steps described below. For the questions that require written answers, please commit a PDF or a text file to the repository.

*https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

- **Check-in your project for the first time**
  - Right click your created project called **assignment3**.
  - In the menu go to **Team**, then click **Share Project**.
  - You should see a dialog "Configure Git Repository". Here, next to the Repository input field click on **Create...**
  - Select your already existing repository that you have created in Execise 1. Note for all your assignments you should use the same directory.
  - click **Finish**.
- **Commit changes in your project**
  - Now that your project is connected to your git repository, you need to make sure that every time you change your code or your report, at the end you commit your changes and send (push) them to the git server.
  - Right click your project called **assignment3**.
  - In the menu go to **Team**, then click **Commit...**.
  - In the Comment field, enter a comment that summarizes your changes.
  - In the Files list, select all the files that you changed and want them to be committed. This typically includes all the Java files but not necessarily all the files (e.g., you dont have to commit setting files of our eclipse installation).
  - Then, click on **Commit** to store the changes locally or **Commit and Push** to also upload them to the server. Note that in order to submit your solution you need to **both** commit and push your changes to the server.
- **Push changes to the git server**
  - Right click your project called **assignment3**.
  - In the menu go to **Team**, then click **Push Branch 'master'**. Note if this is not your fist push you can also use **Push to Upstream** to speed up the process.
  - A new dialog appears, now fill in for the URL field:
    `https://gitlab.inf.ethz.ch/COURSE-PPROG24/pprog-<nethz-username>.git`
  - Click **Preview**
  - An authentication dialog should appear. Fill in your nethz username and password and click **Log in**.
  - Click **Push** to confirm your changes. Note that eclipse might ask for authentication again.
- **Browse your repository online**
  - you can access and browse the files in your repository online on GitLab at:
    `https://gitlab.inf.ethz.ch/COURSE-PPROG24/pprog-<nethz-username>`