Exercises week 1

Last update: 2021/08/29

Practical info before you start

Abbreviations

The following abbreviations are used in the exercise sheets:

- "Goetz" means Goetz *et al.*, *Java Concurrency in Practice*, Addison-Wesley 2006.
- "Herlihy" means Herlihy *et al.*, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2020.

The exercises are a way for you to get a practical understanding of the material. In addition, they serve as the outset for bi-weekly feedback.

Oral Feedback

Feedback is given orally. Please see the document in our Github repository (https://github.itu.dk/jst/PCPP2021-public/blob/master/general-info/assignment-submissions-and-oral-feedback.md) regarding how to book an oral feedback session, and how feedback sessions are conducted.

Mandatory/Challenging Exercises

Exercises are labelled as *mandatory* or *challenging*. For a submission to be considered accepted, all mandatory exercises must be successfully completed (and approved). Furthermore, the code of mandatory exercises shown during oral sessions, must compile and run. Code that is close to correct, but that does not compile and run will not be accepted; note again that this is only for mandatory assignments.

We acknowledge that different students strive to strike different balances between the different classes and between study and other aspects of their lives. Consequently, we include *challenging* exercises. These exercises are *optional*, they are not required for acceptance. These exercises are for students who would like not only to get the major part of the syllabus, but also have a keen interest in all the finer details and subtle concepts. Completing the challenging exercises will increase your chances of obtaining high marks in the exam.

Groups should be composed by students at the same ambition level.

Installing JDK/Gradle and Running Exercise Code

In this course, we will use the Gradle tool and Java JDK 8 or higher for the exercises (and lectures). We have provided a guide on how to set up the programming environment we will use int he course. The guide is in our Github repository (https://github.itu.dk/jst/PCPP2021-public/blob/master/general-info/guide-using-gradle-for-exercises.md). Please follow the guide to ensure that you set up the programming environment in your machine. The guide provides information for different operating systems. If you have any trouble on getting things running please contact us asap, e.g., by attending exercise sessions or posting in the Questions and Answer Forum in LearnIT.

Exercise 1.1 Consider the code in the file `TestLongCounterExperiments.java`. Note that this file contains a class, `LongCounter`, used by two threads:

```
class LongCounter {
  private long count = 0;
  public void increment() {
    count = count + 1;
  }
  public long get() {
    return count;
  }
}
```

Mandatory

1. The `main` method creates a LongCounter object. Then it creates and starts two threads that run concurrently, and each increments the `count` field 10 million times by calling method `increment`.

   What output values do you get? Do you get the expected output, i.e., 20 million?

2. Reduce the `counts` value from 10 million to 100, recompile, and rerun the code. It is now likely that you get the expected result (200) in every run.

   Explain how this could be. Would you say that it is guaranteed that the output is always 200?

3. The `increment` method in LongCounter uses the assignment

   ```
   count = count + 1;
   ```

   to add one to `count`. This could be expressed also as `count += 1` or as `count++`.

   Do you think it would make any difference to use one of these forms instead? Why? Change the code and run it. Do you see any difference in the results for any of these alternatives?

4. Set the value of `counts` back to 10 million. Use Java `ReentrantLock` to ensure that the output of the program equals 20 million. Explain why your solution is correct, and why no other output is possible.

   Note: In your explanation, please use the concepts and vocabulary introduced during the lecture, e.g., critical sections, interleavings, race conditions, mutual exclusion, etc.

   Note II: The notes above applies to all exercises asking you to explain the correctness of your solution.

   Hint: When solving this exercise, try to avoid locking unnecesary code. For instance, think whether the code in the method `get()` need to be part of the critical section.

Challenging

5. Decompile the methods increment from above to see the byte code in the three versions (as is, +=, ++). The basic decompiler is `javap`. The decompiler takes as input a (target) `.class` file. In Gradle projects, `.class` files are located in the directory `app/build/classes/`—after compiling the `.java` files. The flag `-c` decompiles the code of a class. Does the output of `javap` verify or refuse the explanation you provided in part 3.?

6. Extend the `LongCounter` class with a `decrement()` method which subtracts 1 from the `count` field without using locks. Change the code in `main` so that `t1` calls `decrement` 10 million times, and `t2` calls `increment` 10 million times, on a `LongCounter` instance.

   What should the expected output be after both threads have completed?

   Use `ReentrantLock` to ensure that the program outputs 0. Explain why your solution is correct, and why no other output is possible.

7. Explain, in terms of the *happens-before* relation, why your solutions for part 4. and 6. produce the expected output.

Exercise 1.2 Consider this class, whose `print` method prints a dash "-", waits for 50 milliseconds, and then prints a vertical bar "|":

```
class Printer {
  public void print() {
    System.out.print("-");
    try { Thread.sleep(50); } catch (InterruptedException exn) { }
    System.out.print("|");
  }
}
```

Mandatory

1. Write a program that creates a Printer object `p`, and then creates and starts two threads. Each thread must call `p.print()` forever. Note: You can easily run your program using the gradle project for the exercises by placing your code in the directory `week01exercises/app/src/main/java/exercises01/` (remember to add `package exercises01;` as the first line of your Java files).

   You will observe that, most of the time, your program print the dash and bar symbols alternate neatly as in `-|-|-|-|-|-|-|`. But occasionally two bars are printed in a row, or two dashes are printed in a row, creating small "weaving faults" like those shown below:

   ```
   -|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-||--|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-||--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   -|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
   ```

2. Describe an interleaving where this happens.

3. Use Java `ReentrantLock` to ensure that the program outputs the expected sequence `-|-|-|-|....`

   Compile and run the improved program to see whether it works. Explain why your solution is correct, and why it is not possible for incorrect patterns, such as in the output above, to appear.

Challenging

4. Explain, in terms of the *happens-before* relation, why your solution for part 3 produces the correct output.

Exercise 1.3 Imagine that due to the COVID-19 pandemic Tivoli decides to limit the number of visitors to 15000. To this end, you are asked to modify the code for the turnstiles we saw in the lecture. The file `CounterThreads2Covid.java` includes a new constant `MAX_PEOPLE_COVID` equal to 15000. However, the two threads simulate 20000 people entering to the park, so unfortunately some people will not get in `:'(`.

Mandatory

1. Modify the behaviour of the `Turnstile` thread class so that that exactly 15000 enter the park; no less no more. To this end, simply add a check before incrementing `counter` to make sure that there are less than 15000 people in the park. Note that the class does not use any type of locks. You must use `ReentrantLock` to ensure that the program outputs the correct value, 15000.

2. Explain why your solution is correct, and why it always output 15000.

Exercise 1.4 In *Goetz* chapter 1.1, three motivations for concurrency is given: resource utilization, fairness and convenience. In the note about concurrency there is an alternative characterization of the different motivations for concurrency:

- **Exploitation** of multiprocessors. The motivation here is to exploit that the computer has multiple cores (or we have access to a number of physical computers). A good exploitation scheme makes it easy to write programs to make efficient use of true parallelism.
- **Concealed** parallelism. The motivation here is to make it possible for several programs to share some resources in a manner where each can act if they had sole ownership.
- **Intrinsic** parallelism. The real world is intrinsically parallel. Computers who interact with the real world need to deal with this. The motivation here is to make it easy to write programs which responds (in time) to input sensors or other connected devices.

Mandatory

1. Compare the categories in the concurrency note (https://github.itu.dk/jst/PCPP2021-public/blob/master/week01/ConcurrencyNotes/concurrency.pdf and Goetz, try to find some examples of systems which are included in the categories of Goetz, but not in those in the concurrency note, and vice versa (if possible - if not possible, argue why).
2. Find examples of 3 systems in each of the categories in the Concurrency note which you have used yourself (as a programmer or user).