Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 9: Beyond Locks
Spring Semester 2017

Assigned on: **25.04.2017** Due by: **02.05.2017**

Overview

In this exercise we will look at concepts beyond locks.

First, we will look at a well known theoretical exercise for locking in order to acquire a deeper understanding for one of the key problems: Deadlocks.

Then we will try to implement Lock conditions and try to familiarize ourselves with Java's Monitor implementation. This exercise was modeled very closely on an old exam question.

In the last problem we will explore Semaphores and Barriers. In the example code we will see how they work and afterwards will implement our own versions of them.

Getting Prepared

- Download the ZIP file named `assignment9.zip` on the course website.
- 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 from the website (`assignment9.zip`). After that, you should see assignment9 as a project under *Projects*. Click Finish.
- If you have done everything correctly, you should now have a project named assignment9 in your *Package Explorer*.

Dining Philosophers

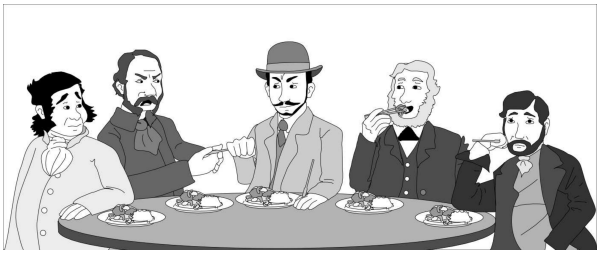

Figure 1: Dining Philosophers Problem

The Dining Philosophers Problem was proposed by Dijkstra in 1965. The problem consists of a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti (see also Figure 1). Five philosophers, do nothing but think for a certain amount of time and eat a very difficult kind of spaghetti which requires two forks to eat. You can think of a philosopher as a thread, executing the following piece of pseudo-code:

```
while(true) {
    think();
    acquire_fork_on_left_side();
    acquire_fork_on_right_side();
    eat();
    release_fork_on_right_side();
    release_fork_on_left_side();
}
```

Assuming that the philosophers know how to think and eat, the methods to pick up the forks and put down the forks again need to satisfy the following constraints:

- Only one philosopher can hold a fork at a time and is permitted to use only the forks to his or her immediate right and left.
- A philosopher needs to acquire both forks to his left and right before he can start to eat.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

In your report, answer the following questions:

a) Given the pseudo-code above, is there a possibility where the philosophers would starve to death (i.e., a deadlock occurs)? Describe how you can reach this scenario and provide a dependency graph for your argument. What property does the dependency graph satisfy in case of a deadlock?

b) Propose a way to implement the routine above so that it does no longer suffer from a deadlock. Explain why your code is now deadlock free.

c) Assuming five philosophers and five forks (as pictured in Figure 1), what is the possible maximum and minimum amount of philosophers eating concurrently if they all want to eat, given your implementation. Describe both scenarios. In case your minimum is not equal to the maximum, is there a way to improve this?

Monitors, Conditions and Bridges

You are in charge of a bridge. A structural engineer tells you, that there is a problem and the bridge can currently carry only a very limited load. Thus you came up with the following regulation: Only either 3 cars or one truck may be on the bridge at each moment. To see how this impacts traffic you run a Java simulation.

a) In `BridgeMonitor` you are given a code skeleton for this. Finish the implementation using a Java object as monitor. All Java objects allow you to use `wait()`, `notify()` and `notifyAll()`.

b) In `BridgeCondition` you are given a similar code skeleton. Finish this implementation using a Lock and it's condition interface.

Hint: Both classes `BridgeMonitor` and `BridgeCondition` have a main method that starts a run with test data.

Semaphores and Databases

A Database application allows - for performance reasons - only 10 concurrent users. In the lecture you have seen the semaphore as a tool to enforce such conditions. The Database program also uses a backup algorithm, that works as follows:

- For all currently logged in users run Part A of the algorithm locally.
- Once all users finished Part A, run Part B for all users locally.
- Once all users finished Part B locally, run the last part of the algorithm on the Database server.

As a tool to enforce such conditions as needed here you have seen Barriers in the lecture.

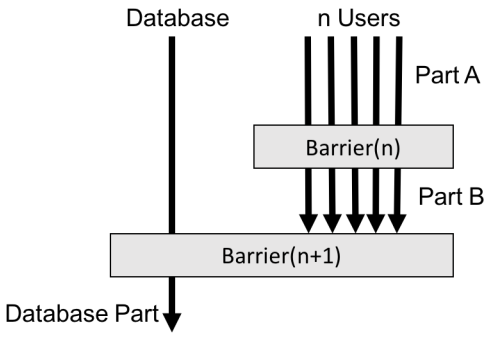

Figure 2: Backup Algorithm

In `DatabaseJava` you can see a reference implementation of this. It uses Java's `Semaphore` and `CyclicBarrier`. Your task is to implement `MySemaphore` and `MyBarrier` as drop-in replacements for those classes.

a) Implement your own semaphore in `MySemaphore`.Your implementation should be based on a Monitor.

b) Implement your own barrier in `MyBarrier`.Your implementation should be based on a Monitor. Compare this implementation with the implementation using Semaphores from the Lecture slides. You will find that Monitors make it a lot easier.

c) Explain the different ways to implement Semaphores (those shown in Lecture and Exercise session) with their pros and cons.

You can test both implementations by running the main methods in `DatabaseJava` and `DatabaseMyCustom`. If you have implemented `MySemaphore` correctly they should behave very similarly.

Submission

In order for us to grade your exercises and give you feedback, you need to submit your code to the Subversion repository. You will find detailed instructions on how to install and set-up Eclipse for use with Subversion in Exercise 1.

Once you have completed the skeleton, commit it to SVN in a directory named `assignment9` by following the steps described below. The questions that require written answers should all be recorded in a single file named `report.pdf` and placed in the base directory of your project (i.e., in folder `assignment9`).

- **Check-in your project for the first time**
  - Right click your created project called **assignment9**.
  - In the menu go to **Team**, then click **Share Project**.
  - In the dialog that now appears, select **SVN** as a repository type, then click **Next**.
  - In case you have submitted Exercise 1, choose **Use existing repository location** and select the pre-defined URL in the dialog that should look like this
    https://svn.inf.ethz.ch/svn/vechev/pprog17/students/NETHZ_USERNAME
    Click Finish. Otherwise follow the steps in Exercise 1 to set-up a repository location.
- **Commit changes in your project**
  - Now that your project is connected with the SVN server, you need to make sure that every time you change your code or your report, at the end you submit it to the SVN server as well.
  - Right click your project called **assignment9**.
  - In the menu go to **Team**, then click **Commit**.
  - In the Comment field, enter a comment that summarizes your changes.
  - Then, click on **Ok**.