Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 9: Beyond Locks
Spring Semester 2020

Assigned on: **20.04.2020** Due by: **27.04.2020**

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 proof / disproof the correctness of a lock.

In the last problem we will look at basic definitions required to understand the Java Memory Model

Since this is a theoretical exercise, you are not required to download or complete any code. Simply write a text file with your answers.

Dining Philosophers

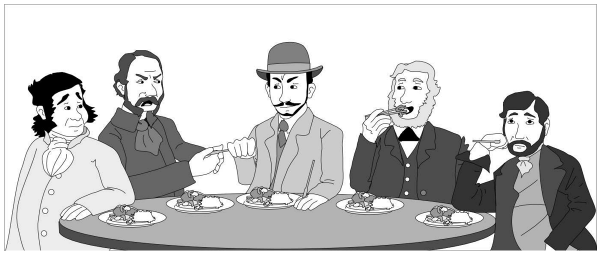


Figure 1: Dining Philosophers Problem

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:

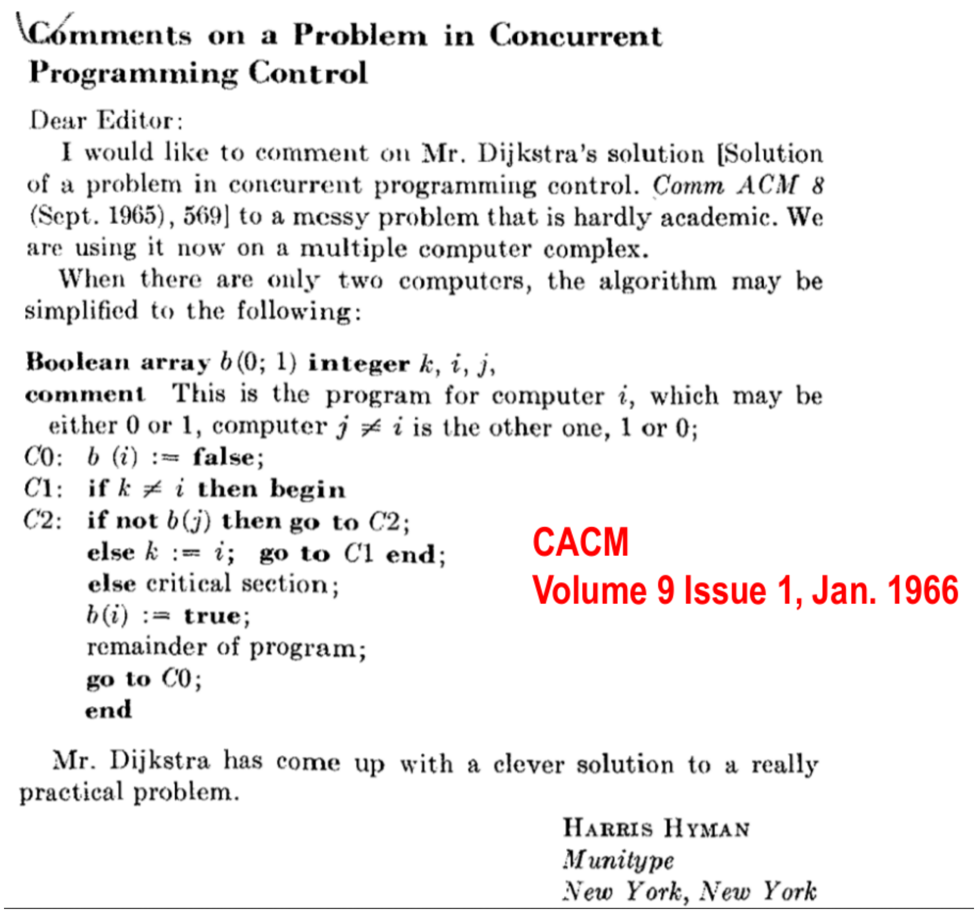
a) Given the pseudo-code above, is there a possibility where the philosophers would starve to death (i.e., a deadlock occurs)? Describe how you can reach this scenario.

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?

Better than Dijkstra?

In 1965 Edgar Dijkstra published an N-thread lock in the journal **Communications of the ACM (Vol. 5 / Nr. 9)** which prompted the response of a reader reproduced below:



Comments on a Problem in Concurrent Programming Control

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control. *Comm ACM 8* (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

```
Boolean array b(0; 1) integer k, i, j,
comment  This is the program for computer i, which may be
  either 0 or 1, computer j ≠ i is the other one, 1 or 0;
C0:  b (i) := false;
C1:  if k ≠ i then begin
C2:  if not b(j) then go to C2;
     else k := i;  go to C1 end;
     else critical section;
     b(i) := true;
     remainder of program;
     go to C0;
     end
```

CACM
Volume 9 Issue 1, Jan. 1966

Mr. Dijkstra has come up with a clever solution to a really practical problem.

Harris Hyman
Munitype
New York, New York

Assuming all variables are volatile and so no access to any variable is reordered, is the proposed lock correct? Use any of the techniques you have seen in the lecture to proof your answer.

The Java Memory Model

In this question we will look at basic definitions required to understand the formal specification of the Java Memory Model.

Transitive Closure

Below shows a table of direct flights between cities. Each row is a starting point, the columns possible destinations. An X in the table means there is a direct flight between the staring point to the destination, unmarked cells indicate there is no direct flight. Other starting points or destinations do not exist.

From / To	Aachen	Bern	Chemnitz	Dresden	Erfurt	Frankfurt	St. Gallen	Hamburg
Aachen	X						X	
Bern	X	X					X	
Chemnitz	X		X	X			X	
Dresden			X	X			X	
Erfurt					X			
Frankfurt						X		
St. Gallen	X	X	X	X			X	
Hamburg	X							X

Formally, this table describes a relation. What is the transitive closure of this relation (expressed as a table)? What does the transitive closure of this relation tell you in your own words?

Programm Order

In the following (Java) code, list all pairs of statements which are in program order:

```
a = 5;              // S1
b = 3;              // S2
if (a + 1 > c) {    // S3
 b = a + 2;         // S4
} else {
 b = a * 2;         // S5
}
```

Synchronization Actions

Which of the following are not *synchronization actions* in the Java Memory Model?

a) Acquiring a lock

b) Releasing a lock

c) Reading a volatile variable

d) Writing a volatile variable

e) Writing a non-volatile variable

f) Reading a non-volatile variable

g) Printing to the screen

Submission

In order to receive feedback for your exercises, 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 write them on paper and bring them to the next exercise session where the solutions will be discussed.

- **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**.
  - You should see a dialog "Configure Git Repository". Here, next to the Repository input field click on **Create...**
  - Select a root git directory or your projects 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 **assignment9**.
  - 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 **assignment9**.
  - 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-PPROG20/`*`<nethz-username>`*`.git`
  - Click **Next**
  - Keep the default values and click **Next**
  - An authentication dialog should appear. Fill in your nethz username and password and click **OK**.
  - Click **Finish** 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-PPROG20/`*`<nethz-username>`*