Formal Lab 8 (Formal Lab)

The dining philosophers' problem

Deadline is April 5th at 11:59 PM. Please submit on Brightspace.

Objective

Your objective is to successfully complete the dining philosophers' simulation program associated with this lab. To achieve that, you need to implement a mutual exclusion scheme so that a set of resources can be shared by multiple tasks (philosophers). Also, you will implement various schemes that can potentially achieve deadlock avoidance.

Introduction

The dining philosophers' problem can be summarized as follows: there are 5 philosophers sitting around a table on which are laid out 5 plates and 5 chopsticks. A philosopher can do two things, either think, or eat. When she/he thinks, a philosopher does not need the chopsticks. When she/he gets hungry she/he wants to eat; then she/he needs to pick up the two chopsticks disposed on each side of his plate. When she/he has picked up both chopsticks, she/he can eat. When she/he has finished eating, the philosopher will put back the chopsticks and will start to think, until she/he gets hungry again.

Current Simulation

A simulation for the dining philosophers' problem is implemented in Java (support files). The Graphical User Interface (GUI) related classes are already implemented. Also, the behavior of a philosopher is implemented in the `Philosopher` class. Nonetheless, the `take` and `release` methods of the `Chopstick` class are not implemented.

In the given program, the philosophers, just like the chopsticks, are numbered from 0 to 4 starting with the top (red) and going in a clockwise direction (trigonometric reverse) (see **Figure 1**). The colors used are mapped to the numeric IDs as follows: 0 to red, 1 to blue, 2 to green, 3 to yellow and 4 to white. The black color means that the philosopher is thinking. The free chopsticks are also colored in black and the chopsticks in use have the color of the philosopher who is holding them.

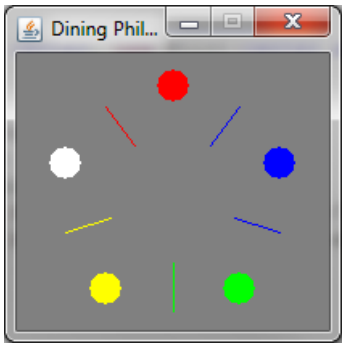

Figure 1 – Dining Philosophers Application

Part A – Basic Implementation

Exercise 1 – Chopstick Class Implementation (25 Points)

Provide an implementation of the `Chopstick` class. You should implement a mutual exclusion scheme in order to allow the philosophers to effectively use the chopsticks as shared resources. In order to achieve that, you must use the `wait` and `notify` methods of the Object class. Study these methods in the following Javadoc: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Part B – Deadlock Avoidance

Implement each of these scenarios separately based on the original code created in **Part A**. **Note that in this section, you will modify the `Choptstick` class, `Philosopher` class, or both, depending on the behavior you are trying to achieve.**

Exercise 2 - Using Semaphores (25 Points)

2.1) Modify the code of **Part A** to allow only four philosophers to eat at the same time. This concept can be easily implemented using a counting semaphore to limit the number of philosophers which can start taking chopsticks at the same time to four. This way, at least one philosopher will be able to eat, and later release the chopsticks, then allowing at least another to eat, etc.

2.2) Discuss whether the deadlock would be avoided in this case. Run the modified program and observe its behavior. Describe your observations.

Exercise 3 - Using Limited Wait (25 Points)

3.1) Modify the code of **Part A** to allow the philosophers to wait only X milliseconds for a chopstick (where X is a constant integer). If X milliseconds pass and the philosopher still did not get the chopstick, she/he gives up. Also, if she/he is currently holding a chopstick, she/he releases it.

3.2) Discuss whether the deadlock would be avoided in this case. Run the modified program and observe its behavior. Describe your observations.

Exercise 4 – Odd Takes Left, Even Takes Right (25 Points)

4.1) Modify the code of **Part A** to allow only the philosophers with odd IDs to grab the left chopstick first, while the others grab the right one first.

4.2) Discuss whether the deadlock would be avoided in this case. Run the modified program and observe its behavior. Describe your observations.

What to Submit

Submit a .zip file containing the following directory structure

- Part A (folder)
    - Exercise 1 (folder)
        - Java files
- Part B (folder)

    - Exercise 2 (folder)
        - Java files
    - Exercise 3 (folder)
        - Java files
    - Exercise 4 (folder)
        - Java files
- A document containing answers to questions **2.2**, **3.2**, and **4.2**.

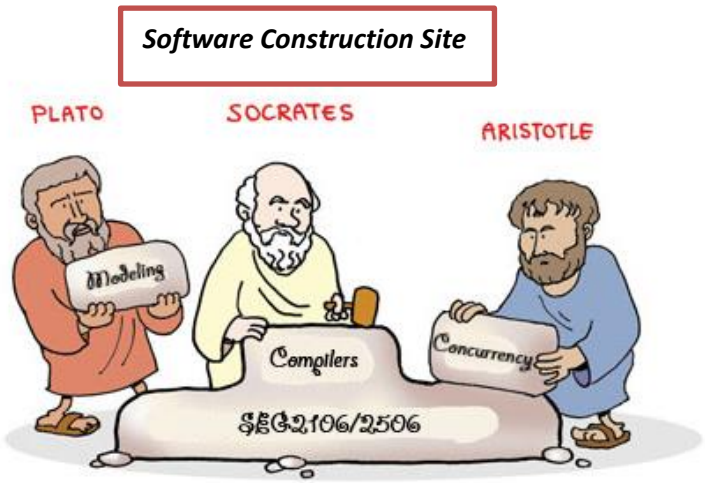

Figure 2 – Irrefutable proof that the dining philosophers have put together the material of this course