CS 307 – Operating Systems

Fall 2019

Homework 2

Barriers and Dining Philosophers Problem Using Java Threads

Date Assigned: 21.10.2019
Due Date Time: 29.10.2019 at 23:55 (sharp, according to server's time)
Late Policy: For every 10 minutes you miss 1 point will be deducted. (Details at the end)

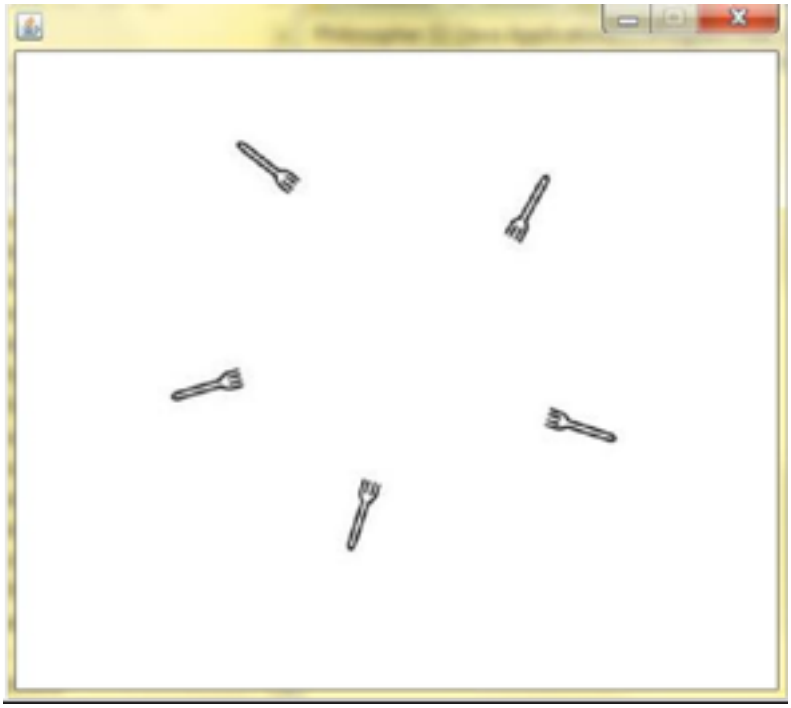
Introduction

In this project, you are going to use Java threads and semaphores to implement a solution for a modified version of famous Dining Philosophers problem. The description of the famous Dining Philosophers problem is available in course slides. In addition to the Dining Philosophers problem, you will implement a barrier for philosophers, so that they can wait each other before they start dining. You will make use of a GUI class (provided to you) to visualize your solution.

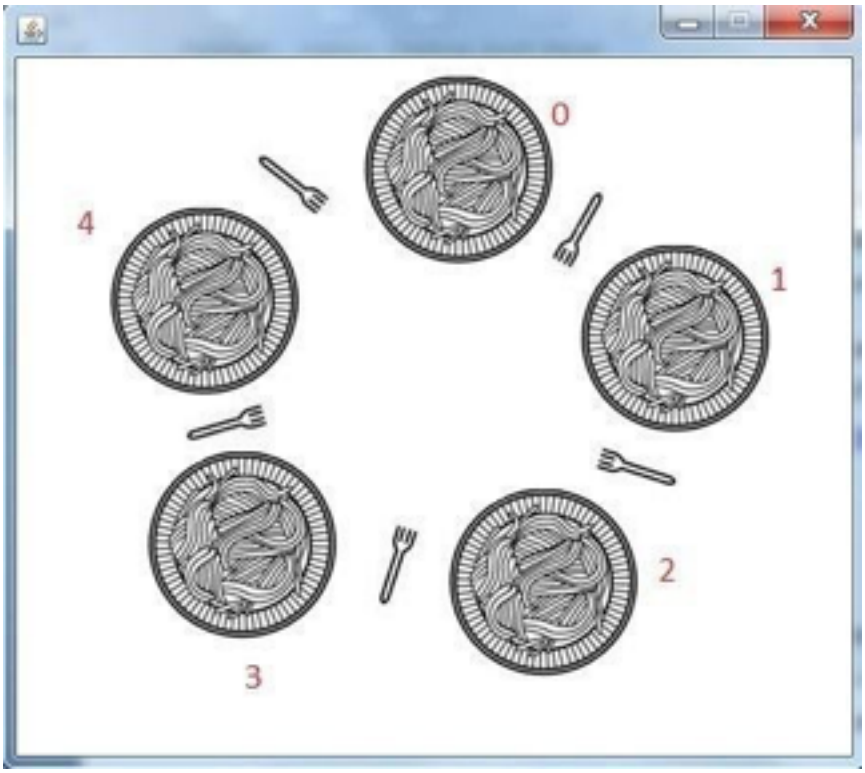
GUI and Related Functions

The partial program that we provide on SUCourse has all necessary functions to initialize and update the GUI. You do not need to and you should not change GUI-related code. If you really think something needs to be changed, let us know first.

This is what you should be getting when you download and run the code directly without making any changes:



This is the screen you should be getting when all philosophers are sitting around the table and start dining:



The numbers next to the plates indicate which philosopher that plate belongs to. In your implementation, your philosophers should also be numbered from 0 to 4 (NOT 1 to 5) so that the GUI works properly.

The functions that are given to you are:

`PutPlate_GUI(int i)`: This function should be called when a philosopher arrives to the table. It will put i'th philosopher's plate and the plate will be yellow. Yellow plate means that the philosopher is waiting for other philosophers to start dining.

`StartDining_GUI(int i)`: This function should be called when all philosophers are sitting around the table and start dining. It will cause the i'th philosopher's plate to turn black&white. White plate means that the philosopher is not waiting anymore for other philosophers to arrive (meaning that they have all arrived) and he can start dining.

`Hungry_GUI(int i)`: This function should be called when a philosopher is hungry. It will cause the i'th philosopher's plate to turn red. It will then put the thread to sleep for 0.5 seconds. This helps us visualize that the philosopher's state is first set to hungry and then he takes the forks and starts eating.

`ForkTake_GUI(int i)`: This function should be called when a philosopher takes the forks that are next to his plate. The forks next to the i'th philosopher's plate will turn red to indicate that they are being used.

`Eating_GUI(int i)`: When a philosopher starts eating, call this function to turn the plate of the i'th philosopher to blue. Each philosopher eats for three seconds when this function is called.

`StopEating_GUI(int i)`: This function should be called when a philosopher stops eating. It will cause the plate of the i'th philosopher to go back to black&white.

`ForkPut_GUI(int i)`: After a philosopher is done eating, he puts the forks back on the table. When this function is called, the forks next to the i'th philosopher's plate will become black&white.

Task & Implementation Details

Here's a short description of the problem: We have five forks, five plates and our five philosophers. At the beginning, philosophers are not at the table and they need to walk towards table. One philosopher can walk faster or slower than another philosopher, because of that philosophers must walk towards the table for a random amount of time, between 1 to 10 seconds. After reaching to the table, a philosopher must put his plate on the table and must wait for other philosophers. When five philosophers sit around the table, they can start dining.

You will have five Java threads, one for each philosopher. While a philosopher waits other philosophers, the thread of the philosopher must be blocked until other philosophers arrive to the table. To achieve this, you must implement a Barrier for 5 threads.

When dining, these philosophers can either think or eat. In order to eat, they need two forks, the one on their left and the one on their right. If the forks are available, they start eating; otherwise they need to wait until both forks are available. Due to different scenarios, as discussed in class, these philosophers may be deadlocked or they might starve. We need an algorithm to avoid these situations.

You need to implement an algorithm so that no philosopher starves and they are never deadlocked. You should make use of semaphores and mutexes in order to achieve this.

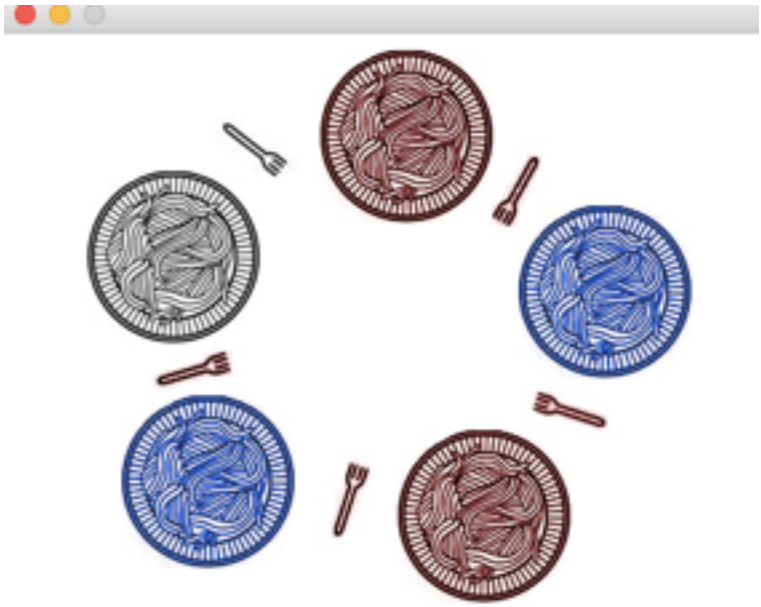
Each philosopher thinks for a random amount of time, between 0 to 10 seconds. The thinking time should vary for each philosopher at each iteration, so you need to make use of random number generators. After that, he becomes hungry and decides to eat. When a philosopher is hungry, his plate should turn red. If he can eat, then he starts eating; otherwise he waits in a hungry state (he doesn't go back to thinking!). When he is eating, his plate should turn blue After he finishes eating, he will go back to thinking. While the philosopher is thinking, his plate should be black and white. This procedure goes on forever. For the duration of the program, you should also pay attention to the forks; the ones that are picked up and being used should be red, while the idle ones should stay black and white. Note that a philosopher does not pick any forks up while he is hungry, forks should only be picked up when the philosopher is allowed to eat.

We have given you a starting point, a partial implementation of the Philosopher class that currently only initializes the GUI. You may decide to continue from that point, or use a different approach. Notice that there may be multiple ways to solve the problem, but in any case, you will need a main function that creates the GUI and the threads.

DO NOT:

- try to use some built-in type as your semaphores. (such as “typedef int semaphore”) Java has its own semaphore class, and its documentation is available on the Web and in the recitation notes. Make use of Java’s semaphore class.
- implement a partially working or wrong solution. (such as the case in which only one philosopher can eat, but there are enough forks on the table for two)
- totally forget about the GUI and use other means of output (for example System.out.println(“..”) to print on the console). While this approach might be handy if you want to debug your program, we will not consider these sorts of output when we grade your program. What happens in the background should be accurately conveyed on the GUI.

Here is a screenshot of an instance of what the final product should look like:



Submission Guides:

Solutions should be submitted in a zip achieve, name your zip achieve as: ***YourNameSurname_ID_hw2.zip*** and submit to **SUcourse**.

Note that, your system time and SUcourse server’s time may not be synchronized so do not wait the last minutes to submit your solution. Only the solutions in the SUcourse system will be graded. Other submissions, such as emailing to instructor or assistants, will not be graded.

You will be graded on the correctness of your solution (as observed from the GUI) and your use of semaphores. You shouldn’t use unnecessary semaphores/mutexes, or mutexes to lock GUI actions. You should definitely avoid using other means of aligning your threads (for example, you could make philosopher 1 sleep while philosopher 2 is eating... but this is VERY wrong). All of these dirty tricks may actually make your code look like it has “somehow finally started working” but in fact they will cause your solution to be invalid!

Late Policy:

You are allowed to late submit your homework. Our policy for late submission depends on how late you submit. For every 10 minutes you are late, 1 point from your grade will be deducted. Below you can find the table for the point deduction system:

Submission Time	Deduction points	Max Grade
23:56 - 00:05	1	99
00:06 - 00:15	2	98
00:16 - 00:25	3	97
00:26 - 00:35	4	96
00:36 - 00:45	5	95
00:46 - 00:55	6	94
00:56 - 01:05	7	93
01:06 - 01:15	8	92
…	…	…
15:46 - 15:55	96	4
15:56 - 16:05	97	3
16:06 - 16:15	98	2
16:16 - 16:25	99	1
16:26 - 16:35	100	0