**CS 4385 - Concurrency and Distributed Systems**

**Project: Dining Philosophers Problem**

**Authors: Jecsan Blanco and M. Rachel Van Pelt**

**Report**: Your project report should discuss details of your logic/functions and how your solution has met/violated the constraints and correctness properties. Any assumptions made, problems encountered, sources referred should all be included in the report  
  
**Assumptions**:  
1. A valid parsable integer will be passed to the program.

2. Readme instructions will be read for program compilation and execution.

**The main programs**

1. **Main** – the main program for the algorithm 6.10
2. **MainV2** – the main program for algorithm 6.12

The main programs are both responsible for initializing all the philosophers and the forks used by the philosophers, the number of philosophers depends on the number passed via the command line (by default the number is set to 5). The main programs are identical besides one is using Philosopher2 for algorithm 6.12

**The Philosopher class (threads):**The program represents what a philosopher does based on the constraints:

 Philosophers either eat or think - 5 by default, or more if given, and each use the eat and think methods.  
 They must have two forks to eat - forced to wait for left and right forks  
 Can only use forks on either side of their plate - forced to use id and id + 1 forks (left and right)  
 Cannot forcefully obtain a fork (no preemption) – the fork class insures only one philosopher can pick it up.

1. **Philosopher** - an implementation of a philosopher using algorithm 6.10
2. **Phiosopher2** - an implementation of a philosopher using algorithm 6.12

The only difference between these two classes is that in **Philosopher2,** if it is the highest id of philosopher trying to eat, they will try to pick up the right fork first instead of the left fork. Since most of the code is the same, we utilized class inheritance to only change what was needed on **Philosopher2**.

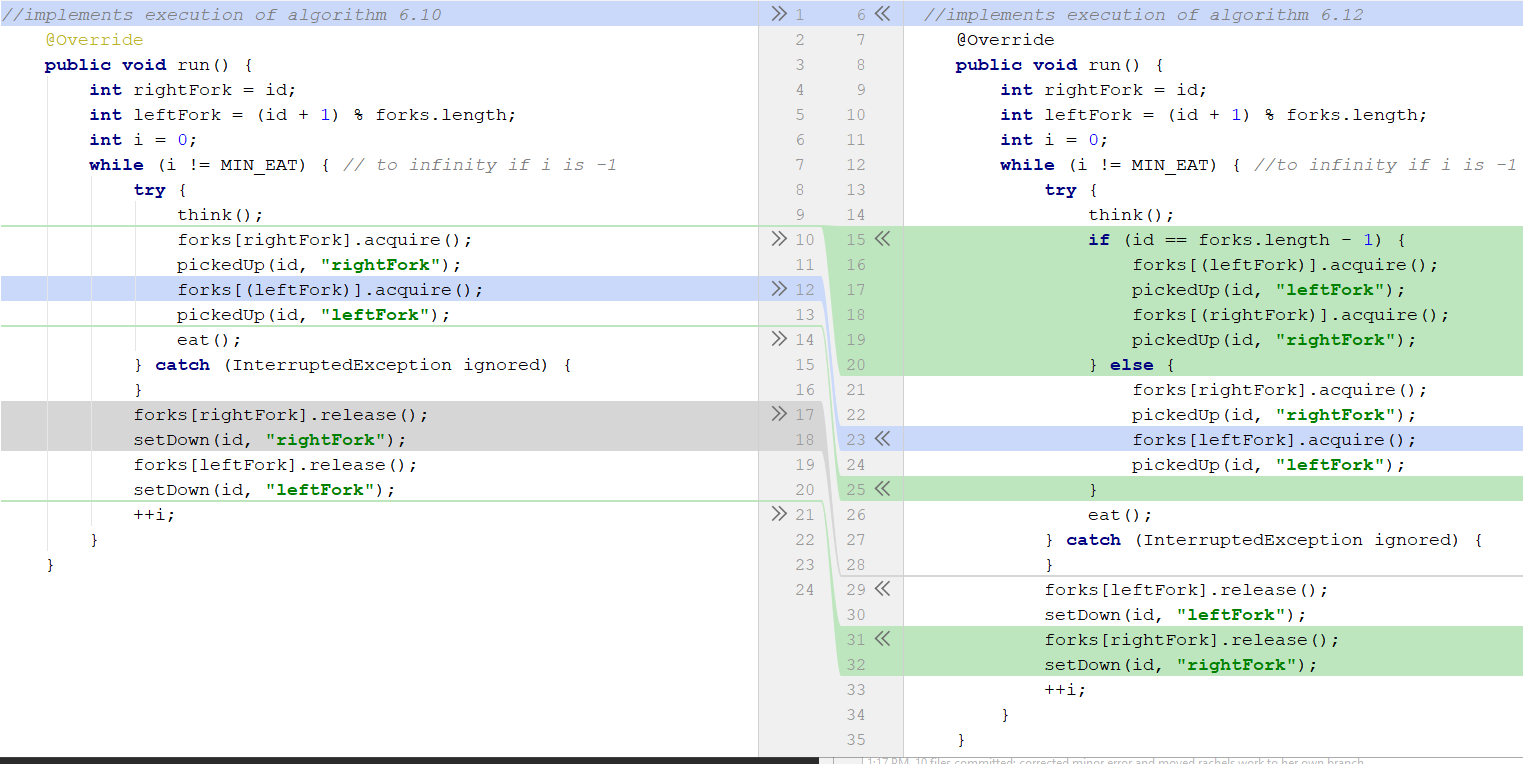


Figure Philosopher (LEFT) , Philosopher2 (right)

**The Forks class (semaphores):**

1. **Fork** – a renamed semaphore so we could use the same terminology as in the problem.

Used for synchronization and insurance of mutual exclusion between philosophers. This is just a renamed semaphore.

**Correctness properties:**

Philosophers cannot share a fork, satisfying mutual exclusion

Freedom from deadlock, where all philosophers pick up a left fork waiting to pick up a right fork

Freedom from starvation, where one or more philosophers do not get a chance to eat

**Problems encountered:**

**Algorithm 6.10**: is susceptible to deadlock.

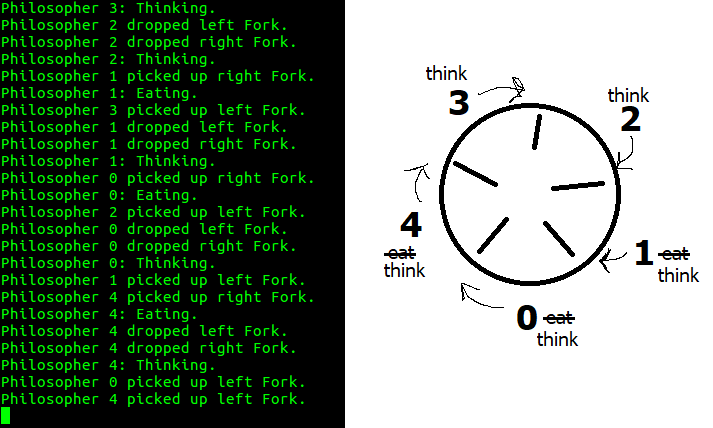


Figure Deadlock 1

In the scenario where every philosopher picks up their left fork everyone would be waiting for the right fork causing a deadlock. This can be shown to happen on the image above by running the program multiple times but can also be forced by setting the eat and think delays to 0 wait.

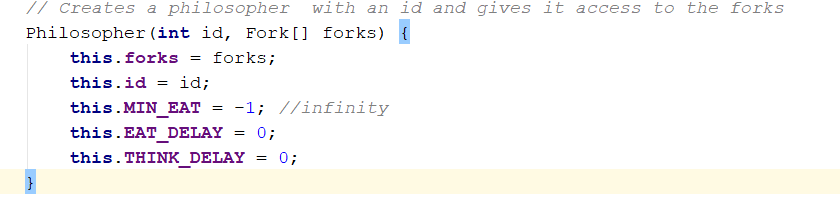
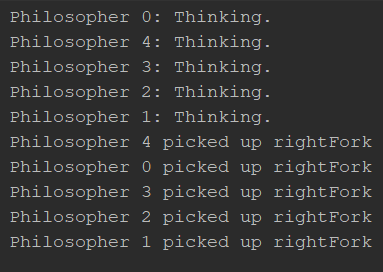


Figure deadlock 2

The setup for **Philosopher2** (algorithm 6.12) seeks to correct this by avoiding everyone from picking up the left forks at the same time. If the last philosopher in the circle always tries the right fork first, instead of the left the problem of deadlock is resolved. Having the last philosopher wait for the right fork first breaks the circular dependency.

**Sources**:

**Makefile** sample: https://www.cs.swarthmore.edu/~newhall/unixhelp/javamakefiles.html

**Insight Into Semaphores:**

https://www.geeksforgeeks.org/semaphore-in-java/

The textbook, chapter 6