Concurrency Assignment

Your task is to extend and improve an existing "solution" to the classic Dining Philosophers problem, originally posed by Dijkstra over 40 years ago. The problem is described in exercises 13.28 and 13.29 (pages 694–5) in the text. In contrast to some of the assignments that have been given in prior years, the challenge this year is primarily one of correct concurrent execution, rather than parallel speedup.

Five philosophers sit around a circular table. In the center is a large communal plate of spaghetti. Each philosopher repeatedly thinks for a while and then eats for a while, at intervals of their own choosing. On the table between each pair of adjacent philosophers is a single fork. To eat, a philosopher requires both adjacent forks: the one on the left and the one on the right. Because they share a fork, adjacent philosophers cannot eat simultaneously.

Source code for the program is in the file `Dining.java`, which you can view in, and save from, your browser. Once you have created your own copy, you can compile it with

```
javac Dining.java
```

and run it with

```
java Dining
```

You will see that it creates an appropriate (though very primitive) graphical display. The 5 philosophers are represented as large colored blobs: blue when thinking, red when waiting, green when eating. The 5 forks are represented as small black blobs: halfway between philosophers when on the table, adjacent to the appropriate philosopher when in use. Note that when you first start up the program, several seconds will elapse before anything appears to happen—i.e., before the first philosopher decides to stop thinking and start eating.

What you'll probably notice right away is that things are not at all synchronized. Nothing in the current code prevents a philosopher from "taking" an unavailable fork. Your task is to add appropriate synchronization to force philosophers to wait when one or both of their forks are unavailable. At the very least, your solution must be *correct*

(no philosopher eats without both forks), *deadlock-free* (there is no situation in which every philosopher is blocked—e.g., because each has picked up their right-hand fork and is waiting for the left), *decentralized* (no philosopher accesses any data other than its own state and that of the neighboring forks), and *starvation-free* (no philosopher waits forever). Ideally, your code should maximize concurrency (as many hungry philosophers as possible are permitted to eat simultaneously) and be *fair* (over the long run, all philosphers get to eat equally often).

To facilitate grading, you are required to add code that accepts an optional `-v` command-line argument and, if that argument is present, prints a log of state transitions to standard output. Specifically, it should print messages of the form

`Philosopher` i `thinking`
`Philosopher` j `eating`
`Philosopher` k `waiting`
`...`

where i, j, and $k \in \{1, 2, 3, 4, 5\}$. Take care to make sure that these messages are output in the order that the events actually happened. So, for example,

```
Philosopher 1 eating
Philosopher 2 eating
```

would be incorrect: Philosopher 2 can't start eating until Philosopher 1 has transitioned back to thinking (and has put down their fork).

When coding up your solution, you must ensure that the philosophers run in "quasi-parallel." Specifically, you may not eliminate any of the calls to `sleep` or `yield`; these calls allow the implementation to switch between threads, even on a single-core version of Java with limited preemption.

Be sure to follow all the rules on the Grading page. As with all assignments, use the turn-in script: `~cs254/bin/TURN_IN`. Put your write-up in a `README.txt` or `README.pdf` file in the directory in which you run the script. Be sure to describe any features of your code that the TA might not immediately notice.

Documentation for Java

is available on-line:

- Tutorials at Oracle. You may be particularly interested in the Concurrency Trail.
- Java Language Specification.

Extra credit suggestions:

1. Implement alternative anti-deadlock strategies. Consider both deadlock *prevention* (designing the program so it never gets into a deadlock situation) and deadlock *recovery* (detecting deadlock and breaking it).
2. Explore generalizations of the dining problem. Consider, for example, a general graph in which a node that decides to "eat" must acquire all its outgoing edges. For an elegant solution, see the 1984 article by Chandy and Misra.
3. Draw prettier graphics.
4. (Ambitious) Implement a *transactional memory* system that allows you to write simply

```
txBegin();
    // grab forks
txEnd();
```

   and trust that the underlying implementation makes the code in the middle atomic while simultaneously avoiding deadlock and maximizing concurrency. Your implementation will need to employ some sort of *speculation*. See, for example, the DSTM system of Herlihy, Luchangco, Moir, and Scherer.

Trivia Assignment

Before the beginning of class on **Monday, November 11**, *each student* should complete the T5 trivia assignment found on Blackboard.

MAIN DUE DATE:

Wednesday November 27, at 12:00 noon; no extensions.