Dining Philosophers

This chapter discusses the famous Dijkstra's Dining Philosopher's problem. Two different solutions are explained at length.

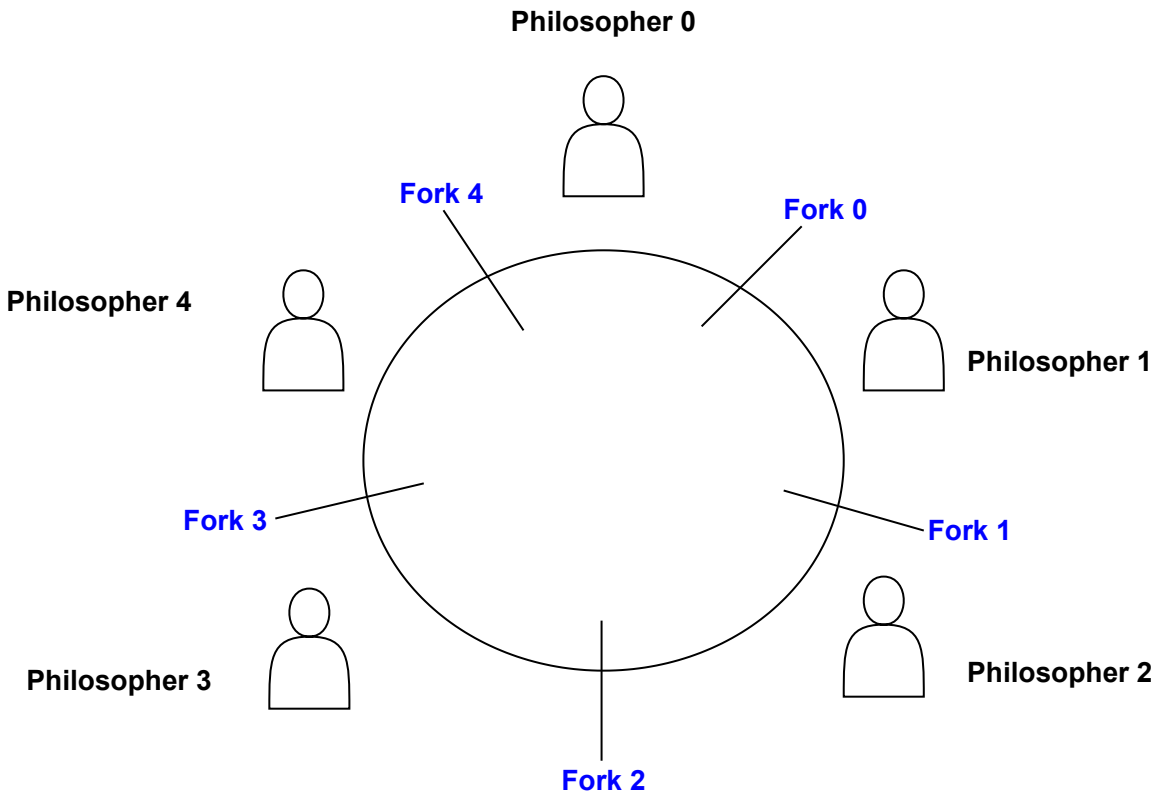
Problem

This is a classical synchronization problem proposed by Dijkstra.

Imagine you have five philosopher's sitting on a roundtable. The philosopher's do only two kinds of activities. One they contemplate, and two they eat. However, they have only five forks between themselves to eat their food with. Each philosopher requires both the fork to his left and the fork to his right to eat his food.

The arrangement of the philosophers and the forks are shown in the diagram.

Design a solution where each philosopher gets a chance to eat his food without causing a deadlock



Dining Philosophers

Solution

For no deadlock to occur at all and have all the philosopher be able to eat, we would need ten forks, two for each philosopher. With five forks available, at most, only two philosophers will be able to eat while letting a third hungry philosopher to hold onto the fifth fork and wait for another one to become available before he can eat.

Think of each fork as a resource that needs to be owned by one of the philosophers sitting on either side.

Let's try to model the problem in code before we even attempt to find a solution. Each fork represents a resource that two of the philosophers on either side can attempt to acquire. This intuitively suggests using a semaphore with a permit value of 1 to represent a fork. Each philosopher can then be thought of as a thread that tries to acquire the forks to the left and right of it. Given this, let's see how our class would look like.

```
public class DiningPhilosophers {
```

```
    // This random variable is used for test purporses only
    private static Random random = new Random(System.currentTimeMilli
s());
    // Five semaphore represent the five forks.
    private Semaphore[] forks = new Semaphore[5];

    // Initializing the semaphores with a permit of 1
    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    // Represents how a philosopher lives his life
    public void lifecycleOfPhilosopher(int id) throws InterruptedExce
ption {

        while (true) {
            contemplate();
            eat(id);
        }
    }

    // We can sleep the thread when the philosopher is thinking
    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));
    }

    // This method will have the meat of the solution, where the
    // philosopher is trying to eat.
    void eat(int id) throws InterruptedException {
    }
}
```

That was easy enough. Now think about the eat method, when a philosopher wants to eat, he needs the fork to the left and right of him. So:

- Philosopher A(0) needs forks 4 and 0
- Philosopher B(1) needs forks 0 and 1

- Philosopher C(2) needs forks 1 and 2
- Philosopher D(3) needs forks 2 and 3
- Philosopher E(4) needs forks 3 and 4

This means each thread (philosopher) will also need to tell us what ID it is before we can attempt to lock the appropriate forks for him. That is why you see the `eat()` method take in an ID parameter.

We can programmatically express the requirement for each philosopher to hold the right and left forks as follows:

`forks[id]`
`forks[(id+4) % 5]`

So far we haven't discussed deadlocks and without them the naive solution would look like the following:

```
public class DiningPhilosophers {

    private static Random random = new Random(System.currentTimeMilli
s());

    private Semaphore[] forks = new Semaphore[5];
    private Semaphore maxDiners = new Semaphore(4);

    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedExce
ption {

        while (true) {
            contemplate();
            eat(id);
        }
    }
```

```
    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));

    }

    void eat(int id) throws InterruptedException {

        // acquire the left fork first
        forks[id].acquire();

        // acquire the right fork second
        forks[(id + 4) % 5].acquire();

        // eat to your heart's content
        System.out.println("Philosopher " + id + " is eating");

        // release forks for others to use
        forks[id].release();
        forks[(id + 4) % 5].release();

    }
}
```

If you run the above code eventually, it'll at some point end up in a deadlock. Realize if all the philosophers simultaneously grab their left fork, none would be able to eat. Below we discuss a couple of ways to avoid this deadlock and arrive at the final solution.

Limiting philosophers about to eat

A very simple fix is to allow only four philosophers at any given point in time to even try to acquire forks. Convince yourself that with five forks and four philosophers deadlock is impossible, since at any point in time, even if each philosopher grabs one fork, there will still be one fork left that can be acquired by one of the philosophers to eat. Implementing this solution requires us to introduce another semaphore with a permit of 4 which guards the logic for lifting/grabbing of the forks by the philosophers. The code appears below.

```
public class DiningPhilosophers {
```

```
    private static Random random = new Random(System.currentTimeMilli
s());

    private Semaphore[] forks = new Semaphore[5];
    private Semaphore maxDiners = new Semaphore(4);

    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedExce
ption {

        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));
    }

    void eat(int id) throws InterruptedException {
        // maxDiners allows only 4 philosphers to
        // attempt picking up forks.
        maxDiners.acquire();

        forks[id].acquire();
        forks[(id + 1) % 5].acquire();
        System.out.println("Philosopher " + id + " is eating");
        forks[id].release();
        forks[(id + 1) % 5].release();

        maxDiners.release();
    }
}
```

Ordering of fork pick-up

Another solution is to make any one of the philosophers pick-up the left fork first instead of the right one. If this gentleman successfully acquires the left fork then, it implies:

- The philosopher sitting next to the left-handed gentleman can't acquire his right fork, so he's blocked from eating since he must first pick up the fork to the right of him (already held by the left-handed philosopher). The blocked philosopher's left fork is free to be picked up by another philosopher.
- The left-handed philosopher may acquire his right fork implying no deadlock since he already picked up his left fork first. Or if he's unable to acquire his right fork, then the gentleman previous to the left-handed philosopher in an anti-clockwise direction will necessarily have had acquired both his right and left forks and will eat. Again, not resulting in a deadlock.

It doesn't matter which philosopher is chosen to be left-handed and made to pick up his left fork first instead of the right one since its a circle. In our solution, we select the philosopher with id=3 as the left-handed philosopher

```
public class DiningPhilosophers2 {

    private static Random random = new Random(System.currentTimeMilli
s());

    private Semaphore[] forks = new Semaphore[5];

    public DiningPhilosophers2() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }
```

```
    public void lifecycleOfPhilosopher(int id) throws InterruptedExce
ption {


        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(500));
    }

    void eat(int id) throws InterruptedException {

        // We randomly selected the philosopher with
        // id 3 as left-handed. All others must be
        // right-handed to avoid a deadlock.
        if (id == 3) {
            acquireForkLeftHanded(3);
        } else {
            acquireForkForRightHanded(id);
        }

        System.out.println("Philosopher " + id + " is eating");
        forks[id].release();
        forks[(id + 1) % 5].release();
    }

    void acquireForkForRightHanded(int id) throws InterruptedExceptio
n {
        forks[id].acquire();
        forks[(id + 1) % 5].acquire();
    }

    // Left-handed philosopher picks the left fork first and then
    // the right one.
    void acquireForkLeftHanded(int id) throws InterruptedException {
        forks[(id + 1) % 5].acquire();
        forks[id].acquire();
    }
}
```

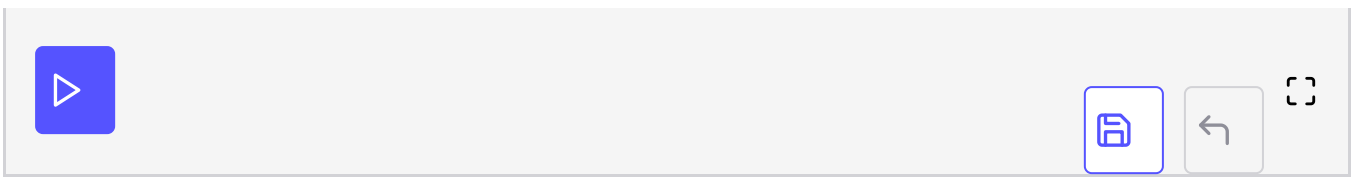
Complete Code 1 | Solution

Complete Code 1st Solution

Below is the code for the first solution we discussed, along with a test. The philosopher threads are perpetual so the widget execution times out. For the limited time the test runs, one can see all philosopher's take turns to eat food without any deadlock.

```
import java.util.Random;
import java.util.concurrent.Semaphore;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        DiningPhilosophers.runTest();
    }
}

class DiningPhilosophers {

    private static Random random = new Random(System.currentTimeMillis());

    private Semaphore[] forks = new Semaphore[5];
    private Semaphore maxDiners = new Semaphore(4);

    public DiningPhilosophers() {
        forks[0] = new Semaphore(1);
        forks[1] = new Semaphore(1);
        forks[2] = new Semaphore(1);
        forks[3] = new Semaphore(1);
        forks[4] = new Semaphore(1);
    }

    public void lifecycleOfPhilosopher(int id) throws InterruptedException {

        while (true) {
            contemplate();
            eat(id);
        }
    }

    void contemplate() throws InterruptedException {
        Thread.sleep(random.nextInt(50));
    }

    void eat(int id) throws InterruptedException {
        maxDiners.acquire();

        forks[id].acquire();
        forks[(id + 1) % 5].acquire();
        System.out.println("Philosopher " + id + " is eating");
        forks[id].release();
        forks[(id + 1) % 5].release();
```

```
        maxDiners.release();
    }


    static void startPhilosoper(DiningPhilosophers dp, int id) {
        try {
            dp.lifecycleOfPhilosopher(id);
        } catch (InterruptedException ie) {

        }
    }

    public static void runTest() throws InterruptedException {
        final DiningPhilosophers dp = new DiningPhilosophers();

        Thread p1 = new Thread(new Runnable() {

            public void run() {
                startPhilosoper(dp, 0);
            }
        });

        Thread p2 = new Thread(new Runnable() {

            public void run() {
                startPhilosoper(dp, 1);
            }
        });

        Thread p3 = new Thread(new Runnable() {

            public void run() {
                startPhilosoper(dp, 2);
            }
        });

        Thread p4 = new Thread(new Runnable() {

            public void run() {
                startPhilosoper(dp, 3);
            }
        });

        Thread p5 = new Thread(new Runnable() {

            public void run() {
                startPhilosoper(dp, 4);
            }
        });

        p1.start();
        p2.start();
        p3.start();
        p4.start();
        p5.start();

        p1.join();
        p2.join();
        p3.join();
        p4.join();
        p5.join();
    }
}
```



Dining Philosopher Solution