Dining Philosophers

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

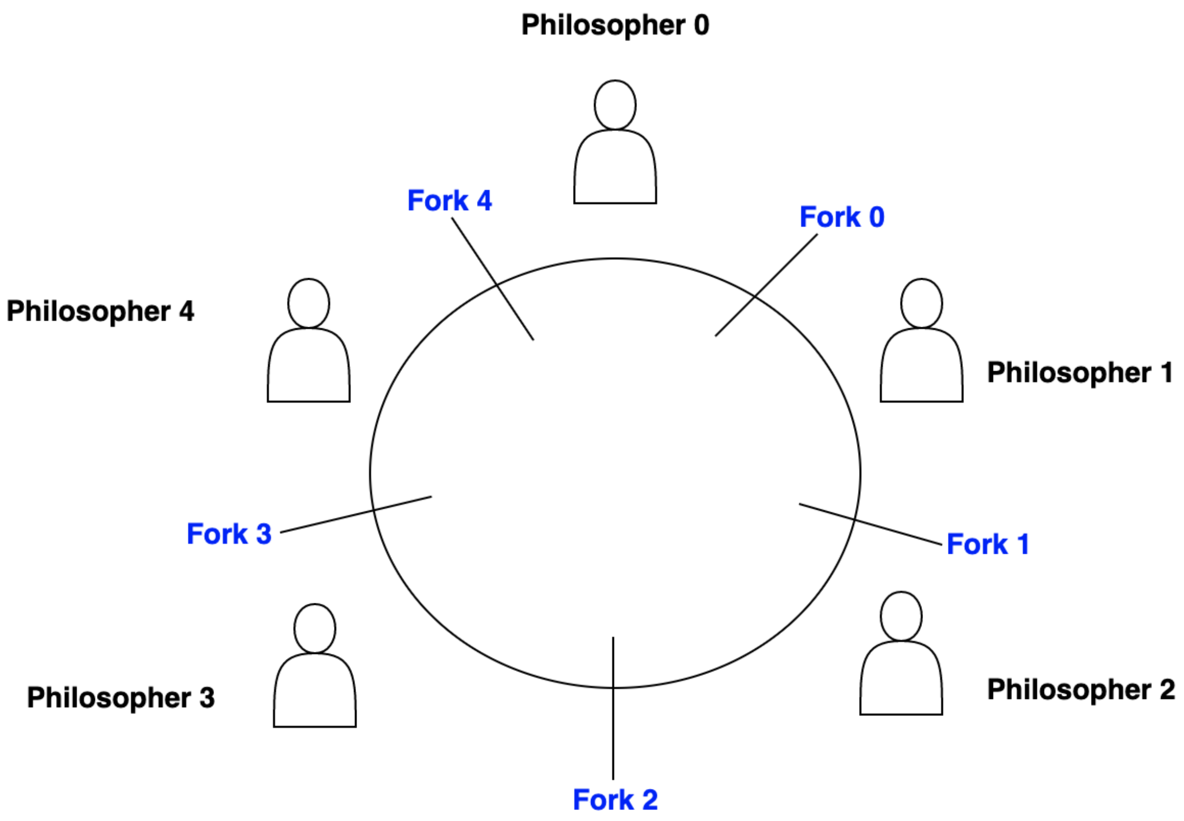
Dining Philosophers

This is a classical synchronization problem proposed by Dijkstra.

Imagine you have five philosophers 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.



Solution

For no deadlock to occur at all and have all the philosophers 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 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:

```
class DiningPhilosopherProblem

  def initialize()
    @forks = Array.new(5)
    @forks[0] = CountingSemaphore.new(1)
    @forks[1] = CountingSemaphore.new(1)
    @forks[2] = CountingSemaphore.new(1)
    @forks[3] = CountingSemaphore.new(1)
    @forks[4] = CountingSemaphore.new(1)
  end

  def lifeCycleOfAPhilosopher(id)
    while true
      contemplate()
      eat(id)
    end
  end

  def contemplate()
    sleepFor = rand()
    sleep(sleepFor)
  end

  def eat(id)
  end
end
```

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 takes in an ID parameter.

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

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:

```
class DiningPhilosopherProblem

  def initialize()
    @forks = Array.new(5)
    @forks[0] = CountingSemaphore.new(1)
    @forks[1] = CountingSemaphore.new(1)
    @forks[2] = CountingSemaphore.new(1)
    @forks[3] = CountingSemaphore.new(1)
    @forks[4] = CountingSemaphore.new(1)
  end

  def lifeCycleOfAPhilosopher(id)
    while true
      contemplate()
      eat(id)
    end
  end

  def contemplate()
    sleepFor = rand()
    sleep(sleepFor)
  end


  def eat(id)
    # acquire the left fork first
    @forks[id].acquire()

    # acquire the right fork second
    @forks[(id + 1) % 5].acquire()

    # eat to your heart's content
    p "Philosopher #{id} is eating"

    # release forks for others to use
    @forks[id].release()
    @forks[(id + 1) % 5].release()
  end
end
```

If you run the above code, it'll eventually end up in a deadlock at some point. Realize if all the philosophers simultaneously grab their right fork,

none would be able to eat. Below we discuss a couple of ways in which this deadlock can be avoided and reach 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 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.

```
class DiningPhilosopherProblem

  def initialize()
    @forks = Array.new(5)
    @forks[0] = CountingSemaphore.new(1)
    @forks[1] = CountingSemaphore.new(1)
    @forks[2] = CountingSemaphore.new(1)
    @forks[3] = CountingSemaphore.new(1)
    @forks[4] = CountingSemaphore.new(1)

    @maxDiners = CountingSemaphore.new(4)
  end

  def lifeCycleOfAPhilosopher(id)
    while true
      contemplate()
      eat(id)
    end
  end

  def contemplate()
    sleepFor = rand()
    sleep(sleepFor)
  end


  def eat(id)

    @maxDiners.acquire()

    # acquire the left fork first
    @forks[id].acquire()

    # acquire the right fork second
    @forks[(id + 1) % 5].acquire()
    # eat to your heart's content
    p "Philosopher #{id} is eating"

    # release forks for others to use
    @forks[id].release()
    @forks[(id + 1) % 5].release()

    @maxDiners.release()
  end
end
```

In the code widget below, we slightly tweak the above solution so that our simulation runs for a few seconds. Infinite looping will cause the code widget to throw a timeout exception. For the limited time the test runs, one can see all philosophers take turns to eat food without any deadlock.

main.rb

semaphore.rb

```
class CountingSemaphore

  def initialize(maxPermits)
    @maxPermits = maxPermits
    @givenOut = 0
    @monitor = Monitor.new
    @condVar = @monitor.new_cond()
  end

  def acquire()
    @monitor.enter()

    while @givenOut == @maxPermits
      @condVar.wait()
    end

    @givenOut += 1
    @condVar.broadcast()

    @monitor.exit()
  end

  def release()
    @monitor.enter()

    while @givenOut == 0
      @condVar.wait()
    end

    @givenOut -= 1
    @condVar.broadcast()

    @monitor.exit()
  end
end
```

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 it's a circle. In our solution, we select the philosopher with id=3 as the left-handed philosopher.

```
def acquireForksForLeftHandedPhilosopher(id)
  @forks[(id + 1) % 5].acquire()
  @forks[id].acquire()
end

def acquireForksForRightHandedPhilosopher(id)
  @forks[id].acquire()
  @forks[(id + 1) % 5].acquire()
end

def eat(id)

  # We randomly selected the philosopher with
  # id 3 as left-handed. All others must be
```

```
    # right-handed to avoid a deadlock.
    if id == 3
      acquireForksForLeftHandedPhilosopher(3)

    else
      acquireForksForRightHandedPhilosopher(id)
    end

    # eat to your heart's content
    p "Philosopher #{id} is eating"

    # release forks for others to use
    @forks[id].release()
    @forks[(id + 1) % 5].release()
  end
```

As in the previous solution, we tweak the given solution so that it runs the simulation for a few seconds and the code widget doesn't time out.

main.rb

semaphore.rb

```
class CountingSemaphore

  def initialize(maxPermits)
    @maxPermits = maxPermits
    @givenOut = 0
    @monitor = Monitor.new
    @condVar = @monitor.new_cond()
  end

  def acquire()
    @monitor.enter()

    while @givenOut == @maxPermits
      @condVar.wait()
    end

    @givenOut += 1
    @condVar.broadcast()

    @monitor.exit()
  end

  def release()
    @monitor.enter()

    while @givenOut == 0
      @condVar.wait()
    end

    @givenOut -= 1
    @condVar.broadcast()

    @monitor.exit()
  end
end
```