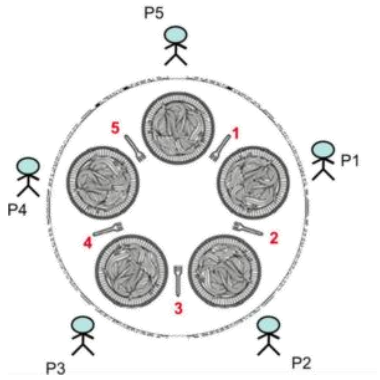
Lab 9 Dining Philosophers Problem

1. Introduction

The Dining Philosophers problem is one of the classic problems used to describe synchronization issues in a multi-threaded environment and illustrate techniques for solving them. Dijkstra first formulated this problem and presented it regarding computers accessing tape drive peripherals. The present formulation was given by Tony Hoare, who is also known for inventing the quicksort sorting algorithm. In this lab, we analyse this well-known problem and code a popular solution.

2. The Problem

The diagram below represents the problem. There are five silent philosophers (P1 - P5) sitting around a circular table, spending their lives eating and thinking. There are five forks for them to share (1 - 5) and to be able to eat, a philosopher needs to have forks in both his hands. After eating, he puts both of them down and the released forks can be picked by another philosopher who repeats the same cycle.



The goal is to come up with a scheme that helps the philosophers achieve their goal of eating and thinking without getting starved to death.

3. A Solution

An initial solution would be to make each of the philosophers follow the following protocol:

```
while(true) {
   // Initially, thinking about life, universe, and everything
   think();

   // Take a break from thinking, hungry now
   pick_up_left_fork();
   pick_up_right_fork();
   eat();
   put_down_right_fork();
   put_down_left_fork();

   // Not hungry anymore. Back to thinking!
}
```

As the above pseudo code describes, each philosopher is initially thinking. After a certain amount of time, the philosopher gets hungry and wishes to eat. At this point, he reaches for the forks on his either side and once he has got both of them, proceeds to eat. Once the eating is done, the philosopher then puts the forks down, so that the forks are available for the philosopher's neighbour.

4. Implementation

We model each of our philosophers as classes that implement the Runnable interface so that we can run them as separate threads. Each Philosopher has access to two forks on his left and right sides:

```
public class Philosopher implements Runnable {

   // The forks on either side of this Philosopher
   private Object leftFork;
   private Object rightFork;

   public Philosopher(Object leftFork, Object rightFork) {
      this.leftFork = leftFork;
      this.rightFork = rightFork;
   }

   @Override
   public void run() {
      // Yet to populate this method
   }

}
```

We also have a method that instructs a Philosopher to perform an action - eat, think, or acquire forks in preparation for eating:

```
public class Philosopher implements Runnable {

   // Member variables, standard constructor

   private void doAction(String action) throws InterruptedException {
         System.out.println( Thread.currentThread().getName() + " " + action);
         Thread.sleep(((int) (Math.random() * 100)));
   }

   // Rest of the methods written earlier
}
```

As shown in the code above, each action is simulated by suspending the invoking thread for a random amount of time, so that the execution order is not enforced by time alone.

Now, let us implement the core logic of a Philosopher.

To simulate acquiring a fork, we need to lock it so that no two Philosopher threads acquire it at the same time. To achieve this, we use the synchronized keyword to acquire the internal monitor of the fork object and prevent other threads from doing the same. We proceed with implementing the run() method in the Philosopher class now:

```
public class Philosopher implements Runnable {

  // Member variables, methods defined earlier

   @Override
   public void run() {
      try {
         while (true) {

            // thinking
            doAction( System.nanoTime() + ": Thinking");
            synchronized (leftFork) {
               doAction( System.nanoTime() + ": Picked up left fork");

               synchronized (rightFork) {

                  doAction( System.nanoTime() + ": Picked up right fork - eating");
                  // eating
                  doAction( System.nanoTime() + ": Put down right fork");
               }

               // Back to thinking
               doAction( System.nanoTime() + ": Put down left fork. Back to thinking");
            }
         }
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
         return;
      }
   }
}
```

This scheme exactly implements the one described earlier: a Philosopher thinks for a while and then decides to eat. After this, he acquires the forks to his left and right and starts eating. When done, he places the forks down. We also add timestamps to each action, which would help us understand the order in which events occur.

To kick start the whole process, we write a client that creates 5 Philosophers as threads and starts all of them:

```
public class DiningPhilosophers {

   public static void main(String[] args) throws Exception {

      Philosopher[] philosophers = new Philosopher[5];
      Object[] forks = new Object[philosophers.length];

      for (int i = 0; i < forks.length; i++) {
         forks[i] = new Object();
      }

      for (int i = 0; i < philosophers.length; i++) {
```

```
            Object leftFork = forks[i];
            Object rightFork = forks[(i + 1) % forks.length];

            philosophers[i] = new Philosopher(leftFork, rightFork);

            Thread t = new Thread(philosophers[i], "Philosopher " + (i + 1));
            t.start();
        }
    }
}
```

We model each of the forks as generic Java objects and make as many of them as there are philosophers. We pass each Philosopher his left and right forks that he attempts to lock using the synchronized keyword.

Running this code results in an output similar to the following. Your output will most likely differ from the one given below, mostly because the sleep() method is invoked for a different interval:

```
Philosopher 1 8038014601251: Thinking
Philosopher 2 8038014828862: Thinking
Philosopher 3 8038015066722: Thinking
Philosopher 4 8038015284511: Thinking
Philosopher 5 8038015468564: Thinking
Philosopher 1 8038016857288: Picked up left fork
Philosopher 1 8038022332758: Picked up right fork - eating
Philosopher 3 8038028886069: Picked up left fork
Philosopher 4 8038063952219: Picked up left fork
Philosopher 1 8038067505168: Put down right fork
Philosopher 2 8038089505264: Picked up left fork
Philosopher 1 8038089505264: Put down left fork. Back to thinking
Philosopher 5 8038111040317: Picked up left fork
```

All the Philosophers initially start off thinking, and we see that Philosopher 1 proceeds to pick up the left and right fork, then eats and proceeds to place both of them down, after which ‘Philosopher 5’ picks it up.

5. The Problem with the above Solution: Deadlock

Though it seems that the above solution is correct, there is an issue of a deadlock arising. Here is a sample output that demonstrates the above issue: (tip: reduce the NUM_PHILOSOPHER if you cannot see a deadlock.)

```
Philosopher 1 8487540546530: Thinking
Philosopher 2 8487542012975: Thinking
Philosopher 3 8487543057508: Thinking
Philosopher 4 8487543318428: Thinking
Philosopher 5 8487544590144: Thinking
Philosopher 3 8487589069046: Picked up left fork
Philosopher 1 8487596641267: Picked up left fork
Philosopher 5 8487597646086: Picked up left fork
Philosopher 4 8487617680958: Picked up left fork
Philosopher 2 8487631148853: Picked up left fork
```

In this situation, each of the Philosophers has acquired his left fork, but cannot acquire his right fork, because his neighbour has already acquired it. This situation is commonly known as the 'circular wait' and is one of the conditions that results in a deadlock and prevents the progress of the system.

6. Resolving the Deadlock

As we saw above, the primary reason for a deadlock is the circular wait condition where each philosopher waits upon a fork that is being held by some other philosopher. Hence, to avoid a deadlock situation we need to make sure that the circular wait condition is broken. There are several ways to achieve this, the simplest one being the follows:

All Philosophers reach for their left fork first, except one who first reaches for his right fork. (Question: why this solution is correct?)

We implement this in our existing code by making a relatively minor change in code:

```
public class DiningPhilosophers {

  public static void main(String[] args) throws Exception {

    final Philosopher[] philosophers = new Philosopher[5];
    Object[] forks = new Object[philosophers.length];

    for (int i = 0; i < forks.length; i++) {
      forks[i] = new Object();
    }

    for (int i = 0; i < philosophers.length; i++) {
      Object leftFork = forks[i];
      Object rightFork = forks[(i + 1) % forks.length];

      if (i == philosophers.length - 1) {

        // The last philosopher picks up the right fork first
        philosophers[i] = new Philosopher(rightFork, leftFork);
      } else {
        philosophers[i] = new Philosopher(leftFork, rightFork);
      }

      Thread t = new Thread(philosophers[i], "Philosopher " + (i + 1));
      t.start();
    }
  }
}
```

The change is that we introduce the condition that makes the last philosopher reach for his right fork first, instead of the left. This breaks the circular wait condition and we can avert the deadlock.

Reference:

The above material is copied and adapted from http://www.baeldung.com/java-dining-philoshophers

Think: can you implement with semaphores? can you think of another way to resolve the circular wait?