# Concurrent and Distributed Systems

Laboratory Assignment 3

December 12, 2019


Student: Marcu Andrei Cristian

Computers and Information Technology

CEN 3.2 A

3rd Year

# Problems statements

**Problem 1** (50p)
Implement the producer consumer problem as follows:
a) using semaphores
b) using monitors
c) using locks

**Problem 2** (50p)
Implement the dining philosophers problem as follows:
a) using semaphores
b) using monitors
c) using locks

Both problems are implemented using Java.

**Problem1:** The producer-consumer problem has two threads. The producer respectively the consumer. The producer is adding data to a queue until it is full, and the consumer is taking data from that queue unless it is empty. The idea behind the problem is that the consumer should go to sleep when the queue is empty and the producer should go to sleep when the queue is full. Three similar implementations are required. I've tried to keep the code as close as I could to the other solutions, changing only the synchronization method( locks, monitors and semaphores ).

**Problem2:** The dining philosophers problem has the next statement. We have n philosophers and n forks around a circular table. A philosopher needs 2 forks to eat( the left one and the right one ) . Implement a method to synchronize this process. The idea behind the solution is that each philosopher has two adjacent forks. We use locks/semaphores/monitors to block two philosophers from using the same fork. I've also tried to keep the code as close as I could to the other solutions, changing only the synchronization method( locks, monitors and semaphores ).

## Common code for all the Producer-Consumer implementations

```
//---------------------------Consumer.java---------------------------

public class Consumer extends Thread{
	PCQ pcq;
	Consumer(PCQ pcq)
	{
		this.pcq = pcq;
	}
	public void run()
	{
		try {
			pcq.cons();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

```
//---------------------------Producer.java---------------------------

public class Producer extends Thread {
	PCQ pcq;
	Producer(PCQ pcq)
	{
		this.pcq = pcq;
	}
	public void run()
	{
		try {
			pcq.prod();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

```
//---------------------------Main.java---------------------------

public class Main {
	public static void main(String args[]) throws
	    InterruptedException
	{
		PCQ pcq = new PCQ();
```

```
            Thread producer = new Thread(new Producer(pcq));
            Thread consumer = new Thread(new Consumer(pcq));
            producer.start();
            consumer.start();
            producer.join();
            consumer.join();
        }
}
```

## Solution Producer-Consumer with Semaphores

The solution I've implemented, presented by our teacher at the laboratory uses two semaphores, one for the producer and one for the consumer. When we the producer adds a number in the queue it acquires the producer semaphore blocking it until it is released. It is released when the consumer gets the value. The same goes for the consumer semaphore. It is acquired when we consume a value and released when we produce another. Next I'll present my source code. The Consumer.java, Producer.java and Main.java are the same for all the solutions.

```
//----------------------------PCQ.java----------------------------

import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.*;
public class PCQ {
        Semaphore semP;
        Semaphore semC;
        Queue<Integer> queue;
        int queueCapacity = 1;
        Random rand = new Random();
        int i;
        PCQ(){
                semP = new Semaphore(1);
                semC = new Semaphore(1);
                queue = new LinkedList<>();
        }
        public void prod() throws InterruptedException
        {
                for(int it = 0 ; it < 10 ; it++)
                {
                        try {
                        semP.acquire();
                        }
                        catch (InterruptedException e)
                        {
```

```
                }
                if(queue.size() < queueCapacity)
                {
                        i = rand.nextInt(100);
                        queue.add(i);
                        System.out.println("Producer added " + i);
                }
                semC.release();
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
    }
    public void cons() throws InterruptedException
    {
        for(int it = 0 ; it < 10 ; it++)
        {
                try {
                        semC.acquire();
                }
                catch(InterruptedException e)
                {
                }
                if(!queue.isEmpty())
                        System.out.println("Consumer removed " +
                            queue.remove());
                semP.release();
                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
    }
}
```

## Experiments and results for producer-consumer with semaphores

Both the producer and the consumer will do 10 steps. ( 10 numbers add, 10 numbers consumed ). The queue capacity in the examples is 1.

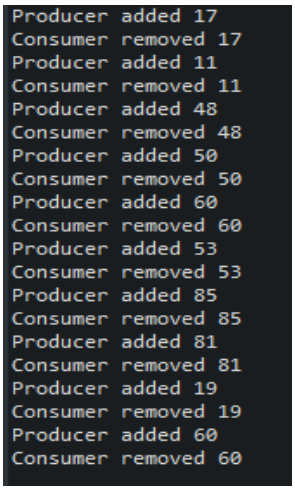
1. Random example 1

```
Producer added 66
Consumer removed 66
Producer added 24
Consumer removed 24
Producer added 17
Consumer removed 17
Producer added 1
Consumer removed 1
Producer added 2
Consumer removed 2
Producer added 20
Consumer removed 20
Producer added 6
Consumer removed 6
Producer added 43
Consumer removed 43
Producer added 9
Consumer removed 9
Producer added 67
Consumer removed 67
```

2. Random example 2

```
Producer added 63
Consumer removed 63
Producer added 50
Consumer removed 50
Producer added 63
Consumer removed 63
Producer added 21
Consumer removed 21
Producer added 85
Consumer removed 85
Producer added 47
Consumer removed 47
Producer added 96
Consumer removed 96
Producer added 32
Consumer removed 32
Producer added 57
Consumer removed 57
Producer added 69
Consumer removed 69
```

3. Random example 3




## Solution Producer-Consumer with Monitors

We use the synchronized keyword in order that 2 threads can't join the block at the same time. When the queue is empty the consumer waits, when the queue is full the producer waits. When a number is added to the queue the producer notifies, when a number is consumed from the queue the consumer notifies.

```
//---------------------------PCQ.java----------------------------

import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;

public class PCQ {
	int queueCapacity = 5;
	Random rand = new Random();
	int i;
	Queue<Integer> queue;
	PCQ(){
		queue = new LinkedList<>();
	}
	public void prod() throws InterruptedException
	{
		for(int it = 0 ; it < 10 ; it++)
		{
			synchronized(this)
			{
```

```
                        while(queue.size() == queueCapacity)
                                wait();
                        i = rand.nextInt(100);
                        System.out.println("Producer added " + i);
                        queue.add(i);
                        notify();
                        Thread.sleep(500);
                }
        }
    }

    public void cons() throws InterruptedException
    {
        for(int it = 0; it < 10; it++)
        {
                synchronized (this)
                {
                        while(queue.isEmpty())
                                wait();
                        System.out.println("Consumer removed " +
                            queue.remove());
                        notify();
                        Thread.sleep(2000);
                }
        }
    }
}
```

## Experiments and results for producer-consumer with monitors

Both the producer and consumer will do 10 steps. The queue capacity I've used for this example is 5.

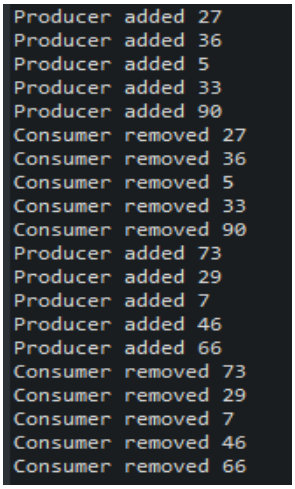
1. Random example 1

```
Producer added 31
Producer added 39
Producer added 51
Producer added 72
Producer added 61
Consumer removed 31
Consumer removed 39
Consumer removed 51
Consumer removed 72
Consumer removed 61
Producer added 2
Producer added 55
Producer added 35
Producer added 15
Producer added 39
Consumer removed 2
Consumer removed 55
Consumer removed 35
Consumer removed 15
Consumer removed 39
```

2. Random example 2

```
Producer added 31
Producer added 73
Producer added 18
Consumer removed 31
Consumer removed 73
Consumer removed 18
Producer added 98
Producer added 37
Producer added 66
Producer added 89
Producer added 58
Consumer removed 98
Consumer removed 37
Consumer removed 66
Consumer removed 89
Consumer removed 58
Producer added 89
Producer added 7
Consumer removed 89
Consumer removed 7
```

3. Random example 3



## Solution Producer-Consumer with Locks

We use lock.lock() and lock.unlock() methods to keep the block available only to one thread. We also use two conditions which are the replacement for monitor methods( wait, await, etc). The conditions are for the cases when queue is full and empty.

---

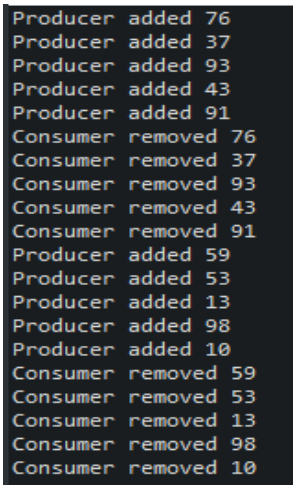
```
//----------------------------PCQ.java----------------------------

import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PCQ {
	int queueCapacity = 5;
	Random rand = new Random();
	int i;
	Queue<Integer> queue;
	Lock lock;
	Condition notFull;
	Condition notEmpty;
	PCQ(){
		queue = new LinkedList<>();
		lock = new ReentrantLock();
		notFull = lock.newCondition();
		notEmpty = lock.newCondition();
	}
	public void prod() throws InterruptedException
	{
		for(int it = 0 ; it < 10 ; it++)
		{
			lock.lock();
			try {
				while(queue.size() == queueCapacity)
					notEmpty.await();
				i = rand.nextInt(100);
				System.out.println("Producer added " + i);
				queue.add(i);
				Thread.sleep(500);
				notFull.signalAll();
			}
			finally {
				lock.unlock();
			}
		}
	}
```

```
        public void cons() throws InterruptedException
        {
                for(int it = 0; it < 10; it++)
                {
                        lock.lock();
                        try {
                                while(queue.isEmpty())
                                        notFull.await();
                                System.out.println("Consumer removed " +
                                    queue.remove());
                                Thread.sleep(2000);
                                notEmpty.signalAll();
                        }
                        finally {
                                lock.unlock();
                        }
                }
        }
}
```

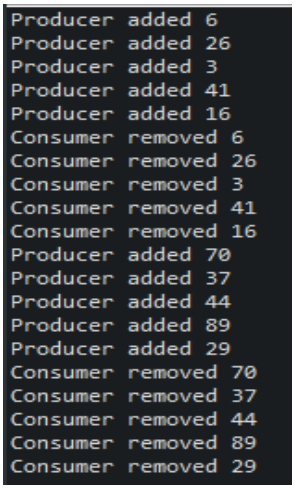
## Experiments and results for producer-consumer with locks

Both the producer and consumer will do 10 steps. The queue capacity I've used for this example is 5.

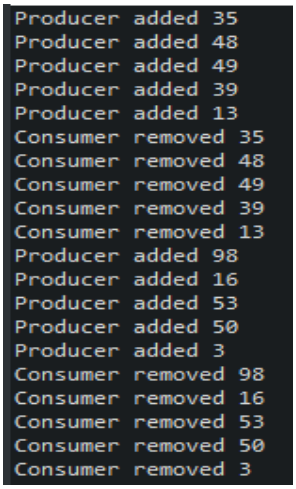
1. Random example 1




2. Random example 2



3. Random example 3




## Conclusions Producer-Consumer problem

The main problem we need to avoid is the deadlock. That's why we need to check everytime if the queue is full or empty. All the synchronization methods(locks, monitors, semaphores) have the same result, and the same efficiency. The reason we've implemented all is just for learning. I've started from the code presented at the laboratory and implemented the other synchronization methods on the same example.

## Common code for all the Dinning Philosophers implementations

```
//----------------------------Philosopher.java----------------------------

public class Philosopher extends Thread {
       public int philosopherId;
    private Fork leftFork;
    private Fork rightFork;
    public boolean stop = false;
    int hasEat = 0;

    public Philosopher(int id, Fork leftFork , Fork rightFork ) {
      this.philosopherId = id;
      this.leftFork = leftFork;
      this.rightFork = rightFork;
    }

    @Override
    public void run() {

      try {
        while (!stop) {
          think();
          if (leftFork.pickUp("Philosopher-" + philosopherId, "left")) {
            if (rightFork.pickUp("Philosopher-" + philosopherId,
                "right")) {
              eat();
              rightFork.putDown("Philosopher-" + philosopherId, "right");
            }
            leftFork.putDown("Philosopher-" + philosopherId, "left");
          }
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    private void think() throws InterruptedException {
      System.out.println("Philosopher-" + philosopherId + " is
          thinking");
      Thread.sleep(1000);
    }

    private void eat() throws InterruptedException {
      System.out.println("Philosopher-" + philosopherId + " is eating");
      hasEat++;
      Thread.sleep(1000);
```

```
    }
}
```

```
//----------------------------Main.java----------------------------


public class Main {

        public static void main(String[] args) throws
             InterruptedException {
                int philosophersNumber = 5;

            Philosopher[] philosophers = null;

              philosophers = new Philosopher[philosophersNumber];
              Fork[] forks = new Fork[philosophersNumber];

              for (int i = 0; i < philosophersNumber; i++) {
                  forks[i] = new Fork(i);
              }

              for (int i = 0; i < philosophersNumber; i++) {
                philosophers[i] = new Philosopher(i, forks[i], forks[(i +
                     1) % philosophersNumber]);
                philosophers[i].start();
              }
              Thread.sleep(10000);
              for (Philosopher philosopher : philosophers) {
                philosopher.stop = true;
                philosopher.join();
              }
              for (Philosopher philosopher2 : philosophers)
                  System.out.println("Philosopher-" +
                       philosopher2.philosopherId + " has eat " +
                       philosopher2.hasEat);

        }
}
```

## Solution Producer-Consumer with Semaphores

The solution I've implemented uses one semaphore for each fork. This semaphore locks fork when it's picked up and releases it when it's put down. This way two philosophers can't pick the same fork at the same time.

```
//----------------------------Fork.java----------------------------
```

```
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Fork {
       Semaphore sem = new Semaphore(1);
    private final int ForkId;

    public Fork(int id) {
      this.ForkId = id;
    }

    public boolean pickUp(String name, String where) throws
        InterruptedException {
      if (sem.tryAcquire()) {
        System.out.println(name + " picked up " + where + " fork-" +
            ForkId);
        return true;
      }
      return false;
    }

    public void putDown(String name, String where) {
        sem.release();
      System.out.println(name + " put down " + where + " fork-" +
          ForkId);
    }
}
```

## Experiments and results for Dinning Philosophers with semaphores

The code will run for 5 seconds and will have 2 philosophers. More experiments with bigger number of philosophers/time will be found in the Experiments and data results folder

1. Random example 1

```
Philosopher-0 is thinking
Philosopher-1 is thinking
Philosopher-0 picked up left fork-0
Philosopher-1 picked up left fork-1
Philosopher-0 put down left fork-0
Philosopher-1 picked up right fork-0
Philosopher-0 is thinking
Philosopher-1 is eating
Philosopher-0 is thinking
Philosopher-1 put down right fork-0
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-1 picked up left fork-1
Philosopher-0 picked up left fork-0
Philosopher-1 put down left fork-1
Philosopher-0 picked up right fork-1
Philosopher-1 is thinking
Philosopher-0 is eating
Philosopher-1 is thinking
Philosopher-0 put down right fork-1
Philosopher-0 put down left fork-0
Philosopher-0 is thinking
Philosopher-1 picked up left fork-1
Philosopher-1 put down left fork-1
Philosopher-0 picked up left fork-0
Philosopher-0 picked up right fork-1
Philosopher-1 is thinking
Philosopher-0 is eating
Philosopher-1 picked up left fork-1
Philosopher-0 put down right fork-1
Philosopher-0 put down left fork-0
Philosopher-1 put down left fork-1
Philosopher-0 has eat 2
Philosopher-1 has eat 1
```

2. Random example 2

```
Philosopher-1 is thinking
Philosopher-0 is thinking
Philosopher-1 picked up left fork-1
Philosopher-0 picked up left fork-0
Philosopher-1 put down left fork-1
Philosopher-0 picked up right fork-1
Philosopher-1 is thinking
Philosopher-0 is eating
Philosopher-0 put down right fork-1
Philosopher-1 picked up left fork-1
Philosopher-0 put down left fork-0
Philosopher-1 picked up right fork-0
Philosopher-0 is thinking
Philosopher-1 is eating
Philosopher-0 is thinking
Philosopher-1 put down right fork-0
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-1 picked up left fork-1
Philosopher-0 picked up left fork-0
Philosopher-1 put down left fork-1
Philosopher-0 picked up right fork-1
Philosopher-1 is thinking
Philosopher-0 is eating
Philosopher-1 is thinking
Philosopher-0 put down right fork-1
Philosopher-0 put down left fork-0
Philosopher-1 picked up left fork-1
Philosopher-1 picked up right fork-0
Philosopher-1 is eating
Philosopher-1 put down right fork-0
Philosopher-1 put down left fork-1
Philosopher-0 has eat 2
Philosopher-1 has eat 2
```

3. Random example 3

```
Philosopher-0 is thinking
Philosopher-1 is thinking
Philosopher-1 picked up left fork-1
Philosopher-0 picked up left fork-0
Philosopher-1 put down left fork-1
Philosopher-0 picked up right fork-1
Philosopher-1 is thinking
Philosopher-0 is eating
Philosopher-1 is thinking
Philosopher-0 put down right fork-1
Philosopher-0 put down left fork-0
Philosopher-0 is thinking
Philosopher-1 picked up left fork-1
Philosopher-0 picked up left fork-0
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-0 picked up right fork-1
Philosopher-0 is eating
Philosopher-0 put down right fork-1
Philosopher-1 picked up left fork-1
Philosopher-1 picked up right fork-0
Philosopher-1 is eating
Philosopher-0 put down left fork-0
Philosopher-0 is thinking
Philosopher-1 put down right fork-0
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-0 picked up left fork-0
Philosopher-0 picked up right fork-1
Philosopher-0 is eating
Philosopher-0 put down right fork-1
Philosopher-1 picked up left fork-1
Philosopher-0 put down left fork-0
Philosopher-1 picked up right fork-0
Philosopher-1 is eating
Philosopher-1 put down right fork-0
Philosopher-1 put down left fork-1
Philosopher-0 has eat 3
Philosopher-1 has eat 2
```

## Solution Dining Philosophers with Monitors

We use the synchronized keyword in order that 2 threads can't join the block at the same time. So only one philospoher can pick the fork

---

```
//----------------------------Fork.java----------------------------

public class Fork {
    private final int ForkId;
    static boolean state = false;

    public Fork(int id) {
      this.ForkId = id;
    }

    public synchronized boolean pickUp(String name, String where) throws
        InterruptedException {
      if (state == false) {
        System.out.println(name + " picked up " + where + " fork-" +
            ForkId);
        state = true;
        return true;
      }
```

```
      return false;
    }

    public synchronized void putDown(String name, String where) {
      System.out.println(name + " put down " + where + " fork-" +
          ForkId);
      state = false;
    }
}
```

---

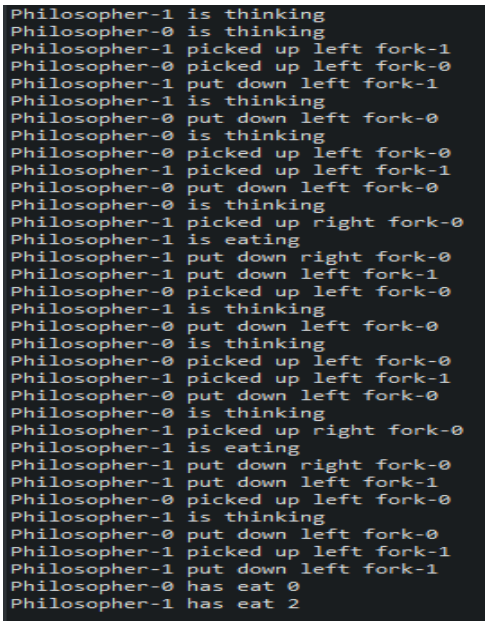
## Experiments and results for Dining Philosophers with monitors

The code will run for 5 seconds and will have 2 philosophers. More experiments with bigger number of philosophers/time will be found in the Experiments and data results folder

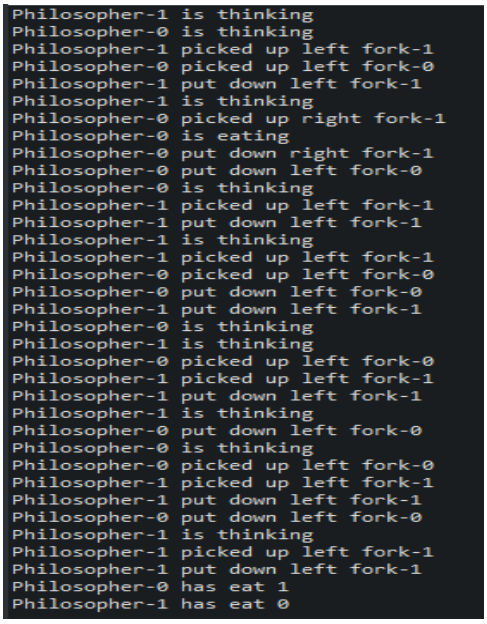
1. Random example 1

```
Philosopher-1 is thinking
Philosopher-0 is thinking
Philosopher-0 picked up left fork-0
Philosopher-1 picked up left fork-1
Philosopher-0 put down left fork-0
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-0 is thinking
Philosopher-0 picked up left fork-0
Philosopher-1 picked up left fork-1
Philosopher-0 put down left fork-0
Philosopher-1 put down left fork-1
Philosopher-0 is thinking
Philosopher-1 is thinking
Philosopher-1 picked up left fork-1
Philosopher-0 picked up left fork-0
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-0 picked up right fork-1
Philosopher-0 is eating
Philosopher-0 put down right fork-1
Philosopher-0 put down left fork-0
Philosopher-1 picked up left fork-1
Philosopher-0 is thinking
Philosopher-1 put down left fork-1
Philosopher-1 is thinking
Philosopher-0 picked up left fork-0
Philosopher-1 picked up left fork-1
Philosopher-0 put down left fork-0
Philosopher-1 picked up right fork-0
Philosopher-1 is eating
Philosopher-1 put down right fork-0
Philosopher-1 put down left fork-1
Philosopher-0 has eat 1
Philosopher-1 has eat 1
```

2. Random example 2




3. Random example 3



## Solution Dining Philosophers with Locks

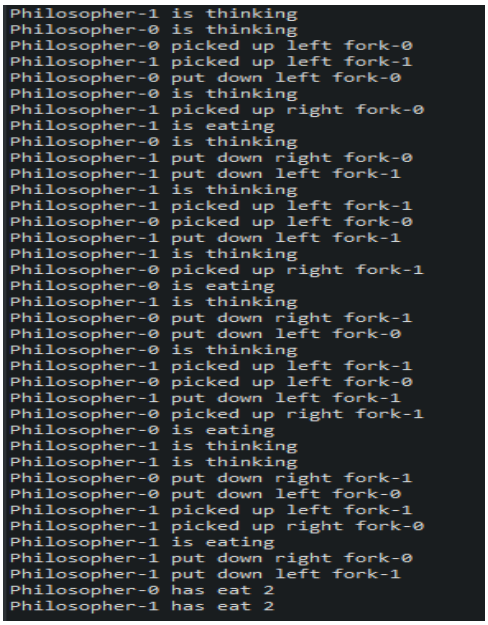
We use lock() method when the fork is picked up and we release it when it's put down.

```
//---------------------------Fork.java----------------------------

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Fork {
    Lock lock = new ReentrantLock();
    private final int ForkId;

    public Fork(int id) {
      this.ForkId = id;
    }

    public boolean pickUp(String name, String where) throws
        InterruptedException {
      if (lock.tryLock()) {
        System.out.println(name + " picked up " + where + " fork-" +
            ForkId);
        return true;
      }
      return false;
    }

    public void putDown(String name, String where) {
        lock.unlock();
      System.out.println(name + " put down " + where + " fork-" +
          ForkId);
    }
}
```

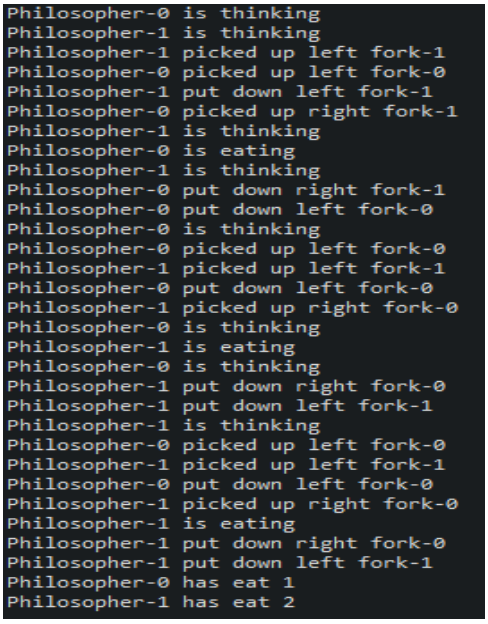
## Experiments and results for producer-consumer with locks

Both the producer and consumer will do 10 steps. The queue capacity I've used for this example is 5.

1. Random example 1




2. Random example 2




3. Random example 3


## Conclusions Dining Philosophers problem

The main problem we need to avoid is the deadlock. Each fork has a synchronization method( doesn't matter if it's a lock/monitor or semaphore because all of them are doing the same thing ). Each fork can be accesed only by one thread at a moment of time, this way two philosophers can't pick the same fork. I've learnt more about deadlocks and concurrent programming synchronization methods

## References

https://en.wikipedia.org/wiki/Dining$_p hilosophers_p roblem$
$https : //docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html$