Assignment 3 - Synchronization Mechanisms Lab Assignment

December 10, 2019

Student: Simon Jozsef-Gabriel

Group: 3.3 A

Speciality: CE

Contents

1 Problem 1 - Producer Consumer Problem

1.1 Problem Statement

Implement the producer consumer problem as follows:
a) using semaphores;
b) using monitors;
c) using locks.

The producer-consumer problem is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

1.2 Implementation

For this problem it was required to implement the problem in 3 separate ways. Because we had to use Java, I chose to work in Eclipse IDE because I am more familiar with this particular IDE. I chose to separate this problem in 3 packages, one for each implementation: semaphores, lock and monitors. Also, I created another package that holds the common parts, which are: the **Producer** class which is an extension of the **Thread** class, the **Consumer** class which is also an extension of the **Thread** class and an interface **PCQueue** where I declared two methods which represent the operations that the producer and the consumer have to execute: *enqueue* - which is the operation of the producer(add a value to the queue) and *dequeue* - which is the operation of the consumer(return the value from the head and then remove it). I chose to create this common package because I found it more easier to access some methods from the interface, and also to declare some variables used in the producer and consumer; and also I didn't wanted to have repetitive code. The pseudocode of these classes and of the interface are presented below:

```
interface PCQueue
  method void enqueue(int value - the value that will be added to the queue,
  int capacity - the maximum size of the queue); - the method that the producer
  will execute in order to produce(add) a value to the queue
  method int dequeue(); - the method that the consumer will execute in order
  to consume(return) the head of the queue and remove it
end interface

class Producer extends Thread
  PCQueue queue ← the queue where the producer will produce a number
  int maxSize ← the maximum size of the queue
```

```
    int count ← *counter for how many numbers were produced*
    function RUN()
        for $i \leftarrow 1, maxSize$ do
            Print(Producer produce i)
            queue.enqueue(i, maxSize)
            count ← count + 1
        end for
    end function
end class

class Consumer extends Thread
    PCQueue queue ← *the queue from where the consumer will consume a number*
    int maxSize ← *the maximum size of the queue*
    int count ← *counter for how many numbers were consumed*
    function RUN()
        for $i \leftarrow 1, maxSize$ do
            Print(Consumer consumed queue.dequeue())
            count ← count + 1
        end for
    end function
end class
```

After the implementation of the common package which contains the classes and the interface presented above, I started to implement the problem in 3 separate ways within 3 packages, each of them containing 2 classes: an **Application** class where is situated the main function and from where we run the code, and a **PCQueue** class which implements the interface presented above(**PCQueue** interface). I will explain and present them in 3 subsections.

1.2.1 Implementation with Semaphores

For this implementation it was required to solve the producer-consumer problem using semaphores. My idea was to create 2 semaphores, one with only one permit and the other with no permit, such that, the consumer will definitely not start first, because it could not consume(return) the head of an empty queue. The **Application** class which here is called **ApplicationSemaphores** contains the main function where all the declarations and initializations are made. At the beginning I initialized the *queue* variable which is of type **PCQueue**, with the constructor from **PCQueueSemaphores** class. After that, in order to have a maximum capacity for the queue, I initialized a variable called capacity which is a random number between 1 and 500. Only 500 because otherwise the output will be to wide and to hard to understand.

After these things, I initialized the two threads, **Producer** and **Consumer**, which will start and join after. The pseudocode of the main class is presented below:

```
class ApplicationSemaphores
  function MAIN($String[]args$)
      PCQueue queue ← new PCQueueSemaphores()
      int capacity ← randomNumberGenerator()
      Producer producer ← new Producer(queue, capacity)
      Consumer consumer ← new Consumer(queue, capacity)
      producer.start()
      consumer.start()
      producer.join()
      consumer.join()
  end function
end class
```

After this class, I implemented the **PCQueueSemaphores** class which implements the interface **PCQueue**. The 2 methods that need to be implemented are the *enqueue* - which is the operation of adding a value to the queue, and *dequeue* - which is the operation of removing the value from the head of the queue. The *dequeue* method returns the value removed, and is executed by the consumer, while the *enqueue* method is executed by the producer. In order to add and remove values from a queue, I declared a variable *queue* which is of type *Queue* - an interface implemented already in java.util library.

In the enqueue method, the producer will enter and aquire a permit in the *producerSemaphre* and if the queue didn't reached the maximum capacity will add the value sent by a parameter. After this, the producer will release the semaphore in order to let another producer come and produce another number. As for the dequeue method, the consumer will enter the method, acquire a permit in the *consumerSemaphore* and, if the queue is not empty, it will save the value from the head of the queue to a temporary variable and after that it will remove the value. At the end, before returning the value stored in the temporary variable, it will release the semaphoer, in order to let another consumer(thread) to acquire a permit. This class is presented in the pseudocode below:

```
class PCQueueSemaphores implements PCQueue
  Queue queue ← new LinkedList()
  Semaphore producerSemaphore ← new Semaphore(1)
  Semaphore consumerSemaphore ← new Semaphore(0)
  function ENQUEUE($number, capacity$)
      producerSemaphore.acquire()
      if $queue.size() < capacity$ then
          queue.add(number)
      end if
      consumerSemaphore.release()
  end function
  function DEQUEUE()
```

```
        consumerSemaphore.acquire()
        if !$queue.isEmpty()$ then
            temp ← queue.remove()
        end if
        producerSemaphore.release()
        return temp
    end function
end class
```

1.2.2 Implementation with Monitors

Regarding this implemetation, the **Application** class, named here **ApplicationMonitors**, is similar with the one presented above, at the implementation with semaphores section, the only line that is changed being the initialization of the *queue* variable which here is as following:

PCQueue queue ← new PCQueueMonitors()

This means that the methods executed by the producer and the consumer will be the ones implemented in the **PCQueueMonitors** class, which is also an implementation of the **PCQueue** interface.

In this class, the two methods, *enqueue* and *dequeue* are synchronized which means that only one thread can use it and execute it, and also that we can use the methods *wait()* and *notifyAll()* which will determine a thread to wait until another thread will notify and awake the other thread which will continue with its job. This is used because the two threads, the producer and the consumer, can't execute the method till the end if some conditions are not applied like: for the producer, the queue must not be full, which means that if it is full it will wait until the consumer notifies it that he consumed a number and removed it from the queue, and the queue is not full anymore, notification that will awake the producer; and, for the consumer, the queue must not be null, and if it is it will wait until the producer will notify the consumer that he produced a number(added a number to the queue) and the queue is not null anymore, notification that will awake the consumer thread. All these are presented in the pseudocode below:

```
class PCQueueMonitors implements PCQueue
    Queue ← new LinkedList()
    function SYNCHRONIZED ENQUEUE($number, capacity$)
        while $queue.size() \geq capacity$ do
            wait()
        end while
        queue.add(number)
        notifyAll()
    end function
    function SYNCHRONIZED DEQUEUE()
        while $queue.isEmpty()$ do
```

```
            wait()
        end while
        temp ← queue.remove()
        notifyAll()
        return temp
    end function
end class
```

1.2.3 Implementation with Locks

For this implementation also, the **Application** class, named here **Application-Locks** is similar to the both from the above, except for the same initialization of the *queue*, which here is as follows:

```
    PCQueue queue ← new PCQueueLocks()
```

This means that the methods executed by the producer and the consumer will be the ones implemented in the **PCQueueLocks** class, which is also an implementation of the **PCQueue** interface.
In the class implementing the **PCQueue** interface, I have a queue, which is of type **Queue** and which is, in face, a linked list; a lock, which will help me acquire the lock and also release it, and a condition which is given by the lock using the method found in the implementation named *newCondition()*. All these will be appearing in the two method implemented by this class: *enqueue* and *dequeue*.
Regarding the first method, at first, the producer will acquire the lock, which will lead to the fact that another producer will not be able to use this method and execute it until the lock is released by the current producer. After this, while the queue size is greater or equal with the maximum capacity, the condition variable will help me make the thread wait for the signal transmitted by the consumer that he removed a value from the queue, so the producer is able to add another one. After producing the value, the producer will unlock the lock, which means that will release it.
In the second method, *dequeue*, the consumer will acquire the lock so that another consumer will not be able to execute the method. After this, if the queue is empty, the consumer will wait until the producer will transmit a signal that tells the consumer that the queue is not empty anymore, because the producer produced a value and added it to the queue, which means that the consumer will continue to consume from the head of the queue and remove the value after saving it to a temporary variable. At the end, it will unlock the lock, which means that it will release it and return the value.
The lock will be initialized with the constructor from the class **ReentrantLock** which is an implementation of the **Lock** interface and which means that if the same thread is trying to acquire the lock, it will. This class is presented in the pseudocode below:

```
class PCQueueLocks implements PCQueue
    Queue queue ← new LinkedList()
    Lock lock ← new ReentrantLock()
```

```
  Condition condition ← lock.newCondition()
  function ENQUEUE($number, capacity$)
      lock.lock()
      while $queue.size() \geq capacity$ do
          condition.wait()
      end while
      queue.add(number)
      condition.signal()
      lock.unlock()
  end function
  function DEQUEUE()
      lock.lock()
      while $queue.isEmpty()$ do
          condition.wait()
      end while
      temp ← queue.remove()
      condition.signal()
      lock.unlock()
      return temp
  end function
end class
```

1.3 Experimental Data

1.3.1 Semaphores

Test 1

For the capacity equal with 10 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Consumer consumed: 1
Producer produce: 3
Consumer consumed: 2
Producer produce: 4
Consumer consumed: 3
Producer produce: 5
Producer produce: 6
Consumer consumed: 4
Producer produce: 7
Consumer consumed: 5
Consumer consumed: 6
Consumer consumed: 7
Producer produce: 8
Producer produce: 9
Consumer consumed: 8
```

```
Producer produce: 10
Consumer consumed: 9
Consumer consumed: 10
Producer produced 10 elements!
Consumer consumed 10 elements!
Producer - Consumer finished!
```

Test 2
For the capacity equal with 117 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Producer produce: 3
Consumer consumed: 1
Consumer consumed: 2
Producer produce: 4
Consumer consumed: 3
Producer produce: 5
Producer produce: 6
Consumer consumed: 4
Consumer consumed: 5
Consumer consumed: 6
Producer produce: 7
................................
Producer produce: 113
Consumer consumed: 112
Producer produce: 114
Consumer consumed: 113
Consumer consumed: 114
Producer produce: 115
Producer produce: 116
Consumer consumed: 115
Producer produce: 117
Consumer consumed: 116
Consumer consumed: 117
Producer produced 117 elements!
Consumer consumed 117 elements!
Producer - Consumer finished!
```

Test 3 For the capacity equal with 99 we have the upcoming output: Producer - Consumer started!

```
Producer produce: 1
Producer produce: 2
Consumer consumed: 1
Consumer consumed: 2
Producer produce: 3
Producer produce: 4
Producer produce: 5
Consumer consumed: 3
Consumer consumed: 4
Producer produce: 6
Producer produce: 7
Consumer consumed: 5
Consumer consumed: 6
Consumer consumed: 7
Producer produce: 8
Producer produce: 9
Consumer consumed: 8
Consumer consumed: 9
..............................
Producer produce: 94
Consumer consumed: 92
Producer produce: 95
Consumer consumed: 93
Producer produce: 96
Consumer consumed: 94
Producer produce: 97
Consumer consumed: 95
Producer produce: 98
Consumer consumed: 96
Producer produce: 99
Consumer consumed: 97
Consumer consumed: 98
Consumer consumed: 99
Producer produced 99 elements!
Consumer consumed 99 elements!
Producer - Consumer finished!
```

Test 4
For the capacity equal with 60 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Consumer consumed: 1
Producer produce: 3
Consumer consumed: 2
Consumer consumed: 3
Producer produce: 4
Producer produce: 5
Producer produce: 6
Consumer consumed: 4
Consumer consumed: 5
Producer produce: 7
Producer produce: 8
Consumer consumed: 6
Consumer consumed: 7
Producer produce: 9
Consumer consumed: 8
Producer produce: 10
..............................
Producer produce: 56
Consumer consumed: 54
Consumer consumed: 55
Consumer consumed: 56
Producer produce: 57
Producer produce: 58
Consumer consumed: 57
Consumer consumed: 58
Producer produce: 59
Producer produce: 60
Consumer consumed: 59
Consumer consumed: 60
Producer produced 60 elements!
Consumer consumed 60 elements!
Producer - Consumer finished!
```

As we can see from the tests above, my implementation seems to be correct because both of the threads, the producer and the consumer, shared the same buffer and each are producing, respective consuming values from the queue. The implementation with the semaphores is, from what I observed, an implementation where both of the threads are taking round after round, meaning that after the producer produced a value, the consumer consumed it almost immediately, which means that the semaphore is working correctly.

1.3.2 Monitors

Test 1
For the capacity equal with 10 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Producer produce: 3
Producer produce: 4
Producer produce: 5
Producer produce: 6
Producer produce: 7
Producer produce: 8
Producer produce: 9
Consumer consumed: 1
Producer produce: 10
Consumer consumed: 2
Consumer consumed: 3
Consumer consumed: 4
Consumer consumed: 5
Consumer consumed: 6
Consumer consumed: 7
Consumer consumed: 8
Consumer consumed: 9
Consumer consumed: 10
Producer produced 10 elements!
Consumer consumed 10 elements!
Producer - Consumer finished!
```

Test 2
For the capacity equal with 143 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Producer produce: 3
Producer produce: 4
Producer produce: 5
Producer produce: 6
Producer produce: 7
Producer produce: 8
Producer produce: 9
Producer produce: 10
Producer produce: 11
Producer produce: 12
Producer produce: 13
Producer produce: 14
Consumer consumed: 1
Consumer consumed: 2
Producer produce: 15
Consumer consumed: 3
Consumer consumed: 4
................................
Consumer consumed: 138
Consumer consumed: 139
Consumer consumed: 140
Consumer consumed: 141
Consumer consumed: 142
Consumer consumed: 143
Producer produced 143 elements!
Consumer consumed 143 elements!
Producer - Consumer finished!
```

Test 3
For the capacity equal with 301 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Consumer consumed: 1
Consumer consumed: 2
Producer produce: 3
Producer produce: 4
Producer produce: 5
Producer produce: 6
Producer produce: 7
Producer produce: 8
Producer produce: 9
Producer produce: 10
Consumer consumed: 3
................................
Consumer consumed: 297
Consumer consumed: 298
Consumer consumed: 299
Consumer consumed: 300
Consumer consumed: 301
Producer produced 301 elements!
Consumer consumed 301 elements!
Producer - Consumer finished!
```

From here I saw that the implementation with monitors is giving very different outputs than the one with semaphores. The two threads are not working the same as in that implementation, and the producer produces much faster than there, which means that by the time the consumer start to consume the values from the head of the queue, the producer almost finished it's job and added all the values into the queue. This here shows more correctly the synchronization between the threads and also the concurrent computing.

1.3.3 Locks

Test 1
For the capacity equal with 10 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Producer produce: 3
Producer produce: 4
Producer produce: 5
Producer produce: 6
Producer produce: 7
Producer produce: 8
Producer produce: 9
Producer produce: 10
Consumer consumed: 1
Consumer consumed: 2
Consumer consumed: 3
Consumer consumed: 4
Consumer consumed: 5
Consumer consumed: 6
Consumer consumed: 7
Consumer consumed: 8
Consumer consumed: 9
Consumer consumed: 10
Producer produced 10 elements!
Consumer consumed 10 elements!
Producer - Consumer finished!
```

Test 2

For the capacity equal with 122 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Consumer consumed: 1
Producer produce: 3
Consumer consumed: 2
Producer produce: 4
Consumer consumed: 3
Producer produce: 5
Producer produce: 6
Producer produce: 7
Producer produce: 8
Consumer consumed: 4
Producer produce: 9
Consumer consumed: 5
.................................
Consumer consumed: 116
Consumer consumed: 117
Consumer consumed: 118
Consumer consumed: 119
Consumer consumed: 120
Consumer consumed: 121
Consumer consumed: 122
Producer produced 122 elements!
Consumer consumed 122 elements!
Producer - Consumer finished!
```

Test 3

For the capacity equal with 312 we have the upcoming output:

```
Producer - Consumer started!
Producer produce: 1
Producer produce: 2
Consumer consumed: 1
Producer produce: 3
Consumer consumed: 2
Producer produce: 4
Consumer consumed: 3
Producer produce: 5
Producer produce: 6
Producer produce: 7
...............................
Consumer consumed: 305
Consumer consumed: 306
Consumer consumed: 307
Consumer consumed: 308
Consumer consumed: 309
Consumer consumed: 310
Consumer consumed: 311
Consumer consumed: 312
Producer produced 312 elements!
Consumer consumed 312 elements!
Producer - Consumer finished!
```

From what I have seen here, the implementation with locks is similar with both of the above implementations. In some cases the producer is producing faster than the consumer consumes, like in the monitors case, and in some cases both of them are executing the commands almost the same, like in the semaphores case. As I seen from the tests, I think that this implementation, also, is working correctly and is showing how the threads work in synchronization and how they are sharing the same buffer(queue).

1.4 Conclusions

In my opinion and from the tests I have made for this problem and for all 3 required implementations, the way that my program works is correct and it shows exactly what the problem required to be shown, how two threads(the producer and the consumer) are working in a concurrent computing, how they synchronize with each other, in a way that both of them are not using the buffer is the same time, even if they share it. Also, it shows that regarding the fact that the queue can be empty(in the case of the consumer) or full(in the case of the producer), the two threads will synchronize with each other and let the other thread know what is happening and when the queue is not empty and not full either.

Working at this problem helped me a lot with understanding the synchronization mechanism applied to this problem and also how they work in a concurrent computing program where threads are running in parallel and not sequential.

Regarding the Java language, I found out how the locks, monitors and semaphores are running and how were implemented, as well as how to use them correctly.

2 Problem 2 - The Dining Philosophers Problem

2.1 Problem Statement

Implement the dining philosophers problem as follows:
a) using semaphores;
b) using monitors;
c) using locks.

The dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. The problem states that five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think. This problem is known as **Resource Starvation** in concurrent computing. Alongside this, 2 major problems can occur: **Deadlock** which is a state in which each member of a group is waiting for another member(philosopher in this case), including itself, to take action, such as sending a message or more commonly releasing a lock; and **Mutual Exclusion** which means that one thread of execution never enters its critical section at the same time that another concurrent thread of execution enters its own critical section, which refers to an interval of time during which a thread of execution accesses a shared resource, such as shared memory(forks in out case).

The main point of this problem is to resolve these major problems in a faster, easier and best way.

2.2 Implementation

Regarding this problem, there were required 3 separate ways of implementing it: with semaphores, with monitors and with locks. We had to implement this problem in such way that the 3 major issues will all be solved and never occur during running the program. Fortunately, I think that I managed to solve those

issue as much as I could, and neither starvation, neither deadlock and neither mutual exclusion didn't appear or appeared to rarely to be a problem. Because I chose to implement the problem in Eclipse IDE, I separated the 3 implementations in 3 separate packages, each representing an implementation(semaphores, monitors, locks). Each package contain an **Application** class which is the main class, and a **Fork** class which implements the **Fork** interface. The interface and the **Philosopher** class are located in another package, named **diningPhilosophers**, where the common classes, operations and interfaces are located.
In the **Fork** interface, I have declared the two operation that each philosopher(thread) can and have to execute, namely: the method pickUp - which is the method where the philosopher will pick up the fork, and putDown - where the philosopher will put down the fork. I chose to use an interface for declaring the forks because I thought that is more easier to understand what is the fork exactly and what are the main operations(method) it has incorporated. This interface is presented in a psedocode below:

```
interface Fork
    boolean pickUp(String side - represents the side where the fork is situated,
    right of left, and used for printing the message, int philosopherIndex -
    represents the index of the philosopher, meaning the philosopher that picked
    up the fork) - is the method where the philosopher will try or succeed to pick
    up the fork
    void putDown(String side - representing what is the type of the fork, left or
    right, int philosopherIndex - representing the number of the philosopher)
    - is the method where the philosopher will put down the fork and sometimes
    notify or signal the other threads that the fork is not used
end interface
```

As for the **Philosopher** class, here is where are located the movements or the actions that each philosopher has to do: *eat, think and the run* method specific for every **Thread** class extension. In the *eat* method, the philosopher(thread) will "eat" for 300 milliseconds, or any other time, chose by the one who is running the program, meaning that the thread will sleep after printing that the philosopher eats. In the *think* method, is the same approach, but the philosopher will think in this time. The *run* method is where all this methods and operations are called, and while there is remainingFood, or while the rounds haven't finished, because each philosopher will have a number of rounds between 10-20, he will think and try to eat, after he picked up both the left and the right fork, in this order. There are several fields for this class representing: the left fork, the right fork, the remingFood counter(rounds), the counter for the number of times each philosopher eats(is initialized with 0) and the index of the philosopher. The class is represented in pseudocode below:

```
class Philosopher extends Thread
    Fork leftFork ← this is the left fork and is initialized in the main class
    Fork rightFork ← the right fork, also initialized in the main class
    int remainingFood ← number of rounds, is between 10-20
    int count ← 0
    int index ← the number representing the philosopher
    function EATING()
        Print(Philosopher index eating!)
        Thread.sleep(300)
    end function
    function THINKING()
        Print(Philosopher index thinking!)
        Thread.sleep(300)
    end function
    function RUN()
        for $i \leftarrow 0, remainingFood$ do
            thinking()
            if $leftFork.pickUp("Left", index)$ then
                if $rightFork.pickUp("Right", index)$ then
                    eating()
                    rightFork.putDown("Right", index)
                end if
                leftFork.putDown("Left", index)
            end if
        end for
    end function
end class
```

After the implementation of the common package which contains the classes and the interface presented above, I started to implement the problem in 3 separate ways within 3 packages, each of them containing 2 classes: an **Application** class where is situated the main function and from where we run the code, and a **Fork** class which implements the interface presented above(Fork interface). I will explain and present them in 3 subsections.

2.2.1 Implementation with Semaphores

For this implementation, it was required to solve the dining philosophers problem using semaphores. In order to succeed, I chose to declare just one semaphore, just with one permit, which will mean that at a moment in time, just a single philosopher can pick up a fork from all 5 forks. But, for trying to solve the starvation issue, I used the method *tryAcquire* which has two parameters, one with a number, and the other is the time unit in which the thread will wait. I chose to use this method, incorporated in the semaphore interface specific to Java, because If another thread(philosopher) will not release a semaphore permit in the amount of time, the thread will not longer wait and will go on thinking again until he will be able to pick up the forks. And I think that this implementation is solving the deadlock and mutual exclusion issues as well. The implementation of the **Fork** interface here is located in the **SemaphoreFork** class, where the methods: *pickUp and putDown* are implemented with the help of a semaphore.

Regarding the main class, the **Application** class, named here **Application-Semaphore** where all the variables needed are initialized and where the threads start and join, which means that here is where the program is starting. These two classes are presented in the pseudocode below:

```
class ApplicationSemaphore
  function MAIN($String[]args$)
      Philosopher[] philosophers ← new Philosopher[5]
      Fork[] forks ← new Fork[philosophers.length]
      int rounds ← given by a random generator, between 10-20
      for $i \leftarrow 0, forks.length$ do
          forks[i] ← new SemaphoreFork(i+1)
      end for
      for $i \leftarrow 0, philosophers.length$ do
          Fork leftFork ← forks[i]
          Fork rightFork ← fork[(i+1)  mod forks.length]
          if $i = philosophers.length - 1$ then
              philosophers[i] ← new Philosopher(rightFork, leftFork, i+1, rounds)
          else
              philosophers[i] ← new Philosopher(leftFork, rightFork, i+1, rounds)
          end if
          philosophers[i].start()
      end for
      for $i \leftarrow 0, philosophers.length$ do
          philosophers[i].join()
      end for
      Print for each philosopher how much he ate
  end function
end class
```

As we can see here, I initialized the variable in different ways, not every time, the philosopher$_i$ has the left fork actually the left fork, because the last philosopher always has its left fork as the right fork, this is used for the deadlock issue and is solving it correctly. Now i will be presenting the **SempahoreFork** class, which I have explained at the beginning of this subsection:

```
class SemaphoreFork implements Fork
  Semaphore semaphore ← new Semaphore(1)
  int index ← the index of the fork
  function PICKUP($forkSide, philosopherIndex$)
      if $semaphore.tryAcquire(10, TimeUnit.MILLISECONDS$ then
          Print(forkSide fork index picked up by philosopher indexPhilosopher)
          return true
      end if
```

```
        return false
    end function
    function putDown($forkSide, philosopherIndex$)
        semaphore.release()
        Print(forkSide fork index released by philosopher indexPhilosopher)
    end function
```

2.2.2 Implementation with Monitors

Regarding this implementation I will start the explanation by saying that the main class, **ApplicationMonitors** class, is similar to the one from the above(the semaphore implementation), the only thing being changed being the initialization of every fork which here is done like this:

```
    for $i \leftarrow 0, forks.length$ do
        forks[i] ← new MonitorsFork(i+1)
    end for
```

And as you can see from here, the new forks will get the implementation of the **Fork** interface from the class named here **MonitorsForks** where the methods are implemented with monitors and by using synchronized blocks and wait() and notifyAll() methods from the Java module.
The synchronized blocks are representing something like a place where just only one thread(philosopher) can access at a specific moment in time, which means that the problem with the concurrent computing is for sure solved, and this threads are not working sequential. In the methods implemented here: *pickUp and putDown*, I used a variable added as a field, *isUsed* that will let me know if the specific fork which a specific philosopher is trying to pick up, is used or not. If the fork is used, as the variable is saying so, the thread will wait for 100 milliseconds or until another thread puts down the fork and notify all other threads that the fork is not used anymore, the variable *isUsed* is also set to false. If after waiting the variable *isUsed* is still true, the philosopher will not wait anymore and go through another round thinking again at first and trying again to pick up the fork.
In my opinion and for what I have observed from the tests, the deadlock, starvation and mutual exclusion issues are all solved or partially solved as the program works correctly. The class is presented in pseudocode below:

```
class MonitorsFork implements Fork
    int index ← the index of the fork
    boolean isUsed ← false
    function synchronized pickUp($forkSide, philosopherIndex$)
        if $isUsed$ then
            wait(100)
        end if
        if !$isUsed$ then
            Print(forkSide fork index picked up by philosopher indexPhilosopher)
            isUsed ← true
        return true
      end if
      return false
    end function
    function SYNCHRONIZED PUTDOWN($forkSide, philosopherIndex$)
      Print(forkSide fork index released by philosopher indexPhilosopher)
      isUsed ← false
      notifyAll()
    end function
end class
```

2.2.3 Implementation with Locks

For this implementation also, the **ApplicationLocks** class, which is the main class, is changed in the same location, meaning that the initialization of the forks is made differently. The forks now have another implementation and this time is with locks. The change looks like this:

```
  for $i \leftarrow 0, forks.length$ do
    forks[i] ← new LocksFork(i+1)
  end for
```

As for the class that implements the interface named **Fork**, the two methods that I had to implement, namely: *pickUp and putDown*, here are implemented using locks which will help me solve the main issues of this problem. The locks will allow a philosopher(thread) to acquire it and to block it, which will mean that another thread will not be able to pick up the fork or it will try for 10 milliseconds, time in which another thread maybe will unlock the lock, which means that it will release it and the thread will succeed in acquire it and use the fork for eating. If in those 10 milliseconds, the thread will not succeed in picking up the fork and will return to thinking again until it will try again to pick the fork up. The lock used is of type ReentrantLock which means that if the same thread will try to access the locked area, or the locked fork in this case, it will succeed. The class is presented below:

```
class LocksFork implements Fork
    int index ← the index of the fork
    Lock lock ← new ReentrantLock()
    function PICKUP($forkSide, philosopherIndex$)
      if $lock.tryLock(10, TimeUnit.MILLISECONDS)$ then
        Print(forkSide fork index picked up by philosopher indexPhilosopher)
        return true
      end if
      return false
    end function
    function PUTDOWN($forkSide, philosopherIndex$)
      Print(forkSide fork index released by philosopher indexPhilosopher)
      lock.unlock
    end function
end class
```

2.3 Experimental Data

2.3.1 Semaphores

Test 1

```
For number of rounds equal with 13 Dining Philosopher starts!
Philosoper 1 thinking!
Philosopher 2 thinking!
Philosopher 3 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Left fork 4 picked up by philosopher 4
Left fork 3 picked up by philosopher 3
Left fork 2 picked up by philosopher 2
Left fork 1 picked up by philosopher 1
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Left fork 2 released by philosopher 2
Philosopher 2 thinking!
Left fork 3 released by philosopher 3
Philosopher 3 thinking!
Left fork 1 released by philosopher 1
Philosopher 1 thinking!
Left fork 1 picked up by philosopher 5
................................
Philosopher 4 eating!
Right fork 3 released by philosopher 2
Left fork 2 released by philosopher 2
Left fork 1 picked up by philosopher 5
Left fork 1 released by philosopher 5
Right fork 5 released by philosopher 4
Left fork 4 released by philosopher 4
Philosopher 1 ate 5 times!
Philosopher 2 ate 6 times!
Philosopher 3 ate 6 times!
Philosopher 4 ate 7 times!
Philosopher 5 ate 6 times!
Dining Philosophers problem stopped!
```

Test 2

```
For number of rounds equal with 16 Dining Philosopher starts!
```

```
Philosopher 1 thinking!
Philosopher 2 thinking!
Philosopher 3 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Left fork 4 picked up by philosopher 4
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Left fork 3 picked up by philosopher 3
Left fork 1 picked up by philosopher 5
Left fork 2 picked up by philosopher 2
Left fork 2 released by philosopher 2
Left fork 1 picked up by philosopher 1
Left fork 1 released by philosopher 5
Philosopher 5 thinking!
.................................
Philosopher 1 eating!
Right fork 4 released by philosopher 3
Left fork 3 released by philosopher 3
Right fork 2 released by philosopher 1
Left fork 1 released by philosopher 1
Philosopher 1 ate 9 times!
Philosopher 2 ate 7 times!
Philosopher 3 ate 8 times!
Philosopher 4 ate 8 times!
Philosopher 5 ate 8 times!
Dining Philosophers problem stopped!
```

Test 3

```
For number of rounds equal with 10 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Philosopher 3 thinking!
Philosopher 2 thinking!
Left fork 3 picked up by philosopher 3
Left fork 1 picked up by philosopher 5
Right fork 5 picked up by philosopher 5
Philosopher 5 eating!
Left fork 4 picked up by philosopher 4
Left fork 2 picked up by philosopher 2
Left fork 3 released by philosopher 3
Philosopher 3 thinking!
.................................
Left fork 1 released by philosopher 1
```

```
Right fork 3 picked up by philosopher 2
Philosopher 2 eating!
Right fork 5 released by philosopher 4
Left fork 4 released by philosopher 4
Right fork 3 released by philosopher 2
Left fork 2 released by philosopher 2
Philosopher 1 ate 5 times!
Philosopher 2 ate 6 times!
Philosopher 3 ate 4 times!
Philosopher 4 ate 5 times!
Philosopher 5 ate 5 times!
Dining Philosophers problem stopped!
```

Test 4

```
For number of rounds equal with 15 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 2 thinking!
Philosopher 4 thinking!
Philosopher 3 thinking!
Philosopher 5 thinking!
Left fork 3 picked up by philosopher 3
Left fork 4 picked up by philosopher 4
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Left fork 1 picked up by philosopher 1
Left fork 2 picked up by philosopher 2
Philosopher 5 thinking!
................................
Right fork 5 picked up by philosopher 5
Left fork 4 released by philosopher 4
Philosopher 5 eating!
Right fork 3 released by philosopher 2
Left fork 2 released by philosopher 2
Right fork 5 released by philosopher 5
Left fork 1 released by philosopher 5
Philosopher 1 ate 4 times!
Philosopher 2 ate 9 times!
Philosopher 3 ate 8 times!
Philosopher 4 ate 8 times!
Philosopher 5 ate 9 times!
Dining Philosophers problem stopped!
```

As we can see from the tests above, the implementation with semaphores is not satisfying the three major issues correctly and all the time, mainly the resource starvation issue. For some tests, it can be seen that some philosophers eat much less than others and almost not at all comparing with the others, but at least I tried to solve this and I think I succeed in solving the other two pretty well.

2.3.2 Monitors

Test 1

```
For number of rounds equal with 10 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 3 thinking!
Philosopher 4 thinking!
Philosopher 2 thinking!
Philosopher 5 thinking!
Left fork 2 picked up by philosopher 2
Left fork 4 picked up by philosopher 4
Left fork 1 picked up by philosopher 1
Right fork 5 picked up by philosopher 4
Left fork 3 picked up by philosopher 3
Philosopher 4 eating!
Left fork 1 released by philosopher 1
................................
Right fork 3 picked up by philosopher 2
Philosopher 2 eating!
Left fork 1 released by philosopher 5
Right fork 5 released by philosopher 4
Left fork 4 released by philosopher 4
Right fork 3 released by philosopher 2
Left fork 2 released by philosopher 2
Philosopher 1 ate 5 times!
Philosopher 2 ate 6 times!
Philosopher 3 ate 5 times!
Philosopher 4 ate 6 times!
Philosopher 5 ate 5 times!
Dining Philosophers problem stopped!
```

Test 2

```
For number of rounds equal with 17 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 2 thinking!
Philosopher 3 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Left fork 2 picked up by philosopher 2
Left fork 3 picked up by philosopher 3
Right fork 4 picked up by philosopher 3
Left fork 1 picked up by philosopher 1
```

```
Philosopher 3 eating!
Left fork 2 released by philosopher 2
Right fork 2 picked up by philosopher 1
Philosopher 1 eating!
................................
Left fork 4 picked up by philosopher 4
Left fork 1 released by philosopher 5
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Right fork 5 released by philosopher 4
Left fork 4 released by philosopher 4
Philosopher 1 ate 8 times!
Philosopher 2 ate 7 times!
Philosopher 3 ate 8 times!
Philosopher 4 ate 9 times!
Philosopher 5 ate 9 times!
Dining Philosophers problem stopped!
```

Test 3

```
For number of rounds equal with 19 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Philosopher 3 thinking!
Philosopher 2 thinking!
Left fork 1 picked up by philosopher 5
Right fork 5 picked up by philosopher 5
Philosopher 5 eating!
Left fork 4 picked up by philosopher 4
Left fork 3 picked up by philosopher 3
Left fork 2 picked up by philosopher 2
Philosopher 1 thinking!
Left fork 4 released by philosopher 4
Philosopher 4 thinking!
................................
Right fork 4 picked up by philosopher 3
Philosopher 3 eating!
Left fork 2 released by philosopher 2
Right fork 5 released by philosopher 5
Left fork 1 released by philosopher 5
Right fork 4 released by philosopher 3
Left fork 3 released by philosopher 3
Philosopher 1 ate 9 times!
Philosopher 2 ate 9 times!
Philosopher 3 ate 10 times!
Philosopher 4 ate 10 times!
Philosopher 5 ate 10 times!
Dining Philosophers problem stopped!
```

As we can see from these tests, there have not been any issue appearing in the output, but when I've done more tests, there were cases when the starvation issue has occurred or even deadlock, but as in the implementation with semaphores, this implementation is pretty close to its correct form.

2.3.3 Locks

Test 1

```
For number of rounds equal with 10 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 3 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Philosopher 2 thinking!
Left fork 1 picked up by philosopher 1
Right fork 2 picked up by philosopher 1
Philosopher 1 eating!
Left fork 3 picked up by philosopher 3
Right fork 4 picked up by philosopher 3
Philosopher 3 eating!
Philosopher 5 thinking!
................................
Philosopher 2 eating!
Left fork 4 picked up by philosopher 4
Right fork 5 released by philosopher 5
Left fork 1 released by philosopher 5
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Right fork 3 released by philosopher 2
Left fork 2 released by philosopher 2
Right fork 5 released by philosopher 4
Left fork 4 released by philosopher 4
Philosopher 1 ate 4 times!
Philosopher 2 ate 4 times!
Philosopher 3 ate 4 times!
Philosopher 4 ate 5 times!
Philosopher 5 ate 4 times!
Dining Philosophers problem stopped!
```

Test 2

```
For number of rounds equal with 14 Dining Philosopher starts!
```

```
Philosopher 1 thinking!
Philosopher 2 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Philosopher 3 thinking!
Left fork 1 picked up by philosopher 1
Right fork 2 picked up by philosopher 1
Philosopher 1 eating!
Left fork 4 picked up by philosopher 4
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
...............................
Philosopher 3 thinking!
Right fork 2 released by philosopher 1
Left fork 1 released by philosopher 1
Right fork 5 released by philosopher 4
Left fork 3 picked up by philosopher 3
Left fork 4 released by philosopher 4
Right fork 4 picked up by philosopher 3
Philosopher 3 eating!
Right fork 4 released by philosopher 3
Left fork 3 released by philosopher 3
Philosopher 1 ate 6 times!
Philosopher 2 ate 5 times!
Philosopher 3 ate 8 times!
Philosopher 4 ate 7 times!
Philosopher 5 ate 5 times!
Dining Philosophers problem stopped!
```

Test 3

```
For number of rounds equal with 16 Dining Philosopher starts!
Philosopher 1 thinking!
Philosopher 3 thinking!
Philosopher 2 thinking!
Philosopher 4 thinking!
Philosopher 5 thinking!
Left fork 1 picked up by philosopher 1
Right fork 2 picked up by philosopher 1
Philosopher 1 eating!
Left fork 3 picked up by philosopher 3
Left fork 4 picked up by philosopher 4
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Philosopher 5 thinking!
...............................
```

```
Right fork 5 picked up by philosopher 4
Philosopher 4 eating!
Right fork 2 released by philosopher 1
Left fork 1 released by philosopher 1
Right fork 5 released by philosopher 4
Left fork 4 released by philosopher 4
Philosopher 1 ate 8 times!
Philosopher 2 ate 7 times!
Philosopher 3 ate 7 times!
Philosopher 4 ate 8 times!
Philosopher 5 ate 8 times!
Dining Philosophers problem stopped!
```

Regarding this implementation, from the tests from the above and also from the several tests I've made, I observed that this implementation is the most correct and is succeeding in solving the issued the most correctly and also in giving right outputs and presenting the concurrent computing of this problem.

2.4 Conclusions

Working at this project I managed to understand more about concurrent computing and about how this works in Java programming language. I also succeed in understanding more deeply the concepts of resource starvation, deadlock and mutual exclusion in a concurrent computing program.

As it results from the results from the above, I didn't managed to entirely solve the problem and its requirements, but I tried to and managed to reach far enough in my opinion. I think that a great target will be for me to completely succeed in solving this problem and in understanding all the concepts that it represents.

I am really thankful that I had to work on this problem because I managed to enlarge my knowledge regarding Java programming language, concurrent programs and programming in general.

3 References

1) https://stackoverflow.com/

2) https://en.wikibooks.org/wiki/LaTeX

3) https://tex.stackexchange.com

4) https://www.sharelatex.com

5) https://en.wikipedia.org/wiki/Dining_philosophers_problem

6) Concurrent and Distributed Systems classes and laboratories

7) https://en.wikipedia.org/wiki/Producer-consumer_problem

8) https://dzone.com/articles/reentrantlock-and-dining-philo

9) https://www.baeldung.com/java-dining-philoshophers

10) https://www.geeksforgeeks.org/producer-consumer-solution-using-semaphores-java/

11) https://www.geeksforgeeks.org/queue-interface-java/

12) https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores