Concurrent and Distributed Systems

Laboratory Assignment 4

December 22, 2019

Student: Marcu Andrei Cristian

Computers and Information Technology

CEN 3.2 A

3rd Year

Problems statements

Problem 1 (50p)
Implement the dining philosophers problem using channels

Problem 2 (50p)
Implement the readers-writers problem using semaphores as follows:
- More readers can read at the same time the same shared resource as long as there is no writer with writing access to the same shared resource
- A writer will get exclusive access to the shared resource (no other writer and/or reader can have access at the same time to the same shared resource)
- There is no starvation for readers or writers

Both problems are implemented using Java.

Problem1: The dining philosophers problem has the next statement. We have n philosophers and n forks around a circular table.Both philosophers and forks are threads. 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. Philosophers communicate with the fork[i] through the channel[i], fork[i+1] through the channel[i+1] etc. Channels are composed of 2 methods (send and receive). When a message is RECEIVED it means that the adjacent forks are not in use and the philosopher can pick them up. After the philosopher finishes eating he'll SEND a message to the next philosopher.

Problem2: The readers-writers is a problem like producer consumer. The readers can read only after a writer has finished writing. I've implemented the problem wih 3 semaphores, one for the writer, one for the reader and one which is common for both of them(I've called it orderSemaphore) which keeps the order of the writers and readers, avoiding starvation. I have two methods, read and write. The read method uses all 3 semaphores, the readerSemaphore is acquired and released multiple times so many readers can read at the same time. The order semaphore is acquired to avoid starvation. The writer semaphore is acquired and released when the last reader finished reading. The write method uses the writerSemaphore and orderSemaphore. The writer Semaphore is locked when the writer starts writing and unlocked when it finished. The orderSemaphore is used to avoid starvation.

Source code for Dinning Philosophers with channels

```
//---------------------------Channel.java---------------------------


public class Channel<T> {
       private T chan = null;
       int ready = 0;

       public synchronized void send(T mes) throws InterruptedException
           {
              // save the message in channel
              chan = mes;
              ++ready;
              notifyAll();
              while(chan != null) {
                     wait();// wait until the message is received,
                         after that the channel becomes null
              }
       }


       public synchronized T receive() throws InterruptedException {

              while(ready == 0) {
                     wait();
              }
              --ready;
              T tmp = chan; // we save the message
              chan = null;
              notifyAll();// notify all the senders
              return(tmp);
       }
}
```

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

public class Fork extends Thread {

              private Boolean isReceived;
              private int i;
              private ArrayList<Channel<Boolean>> forks;

              public Fork(int i, ArrayList<Channel<Boolean>> forks) {
                     this.i = i;
                     this.forks = forks;
```

```
		}
		public void run() {

			while(true) {

				try {
					(forks.get(i)).send(true);
					isReceived =
					    (forks.get(i)).receive();
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
}
```

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

public class Philosopher extends Thread {
	private Boolean isReceived;
	private String name;
	private int leftFork;
	private int rightFork;
	private ArrayList<Channel<Boolean>> forks;


	public Philosopher(int leftFork, int rightFork, String name,
	    ArrayList<Channel<Boolean>> forks2) {
		this.leftFork = leftFork;
		this.rightFork = rightFork;
		this.forks = forks2;
		this.name = name;
	}

	public void run() {

		while(true) {


			try {
				System.out.println(name + " is thinking");
				Thread.sleep(2000);
				isReceived =
				    (forks.get(leftFork)).receive();
				isReceived =
				    (forks.get(rightFork)).receive();
				System.out.println(name + " is eating");
				Thread.sleep(2000);
				(forks.get(leftFork)).send(true);
```

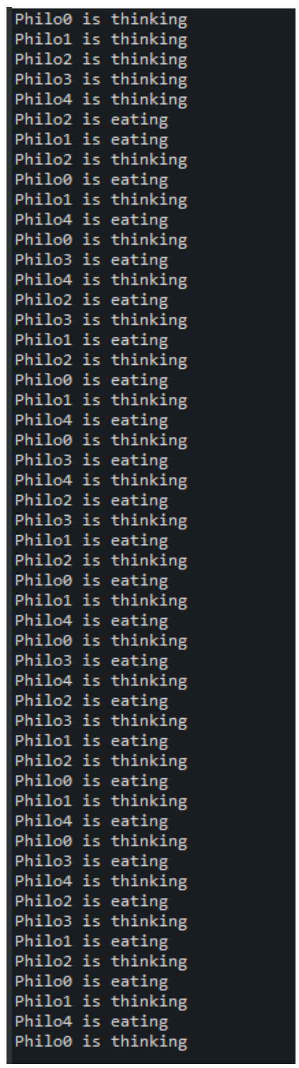
```
                        (forks.get(rightFork)).send(true);
                } catch (InterruptedException e) {

                        e.printStackTrace();
                }

        }
    }
}
```

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

import java.util.ArrayList;

public class Main {
public static void main(String[] args) throws InterruptedException {
        int philosophersNumber = 5; // no of philosopohers = no
            of forks
        ArrayList<Channel<Boolean>> forks = new
            ArrayList<Channel<Boolean>>();
        Fork[] forksArray = new Fork[philosophersNumber];
        Philosopher[] philosophers = new
            Philosopher[philosophersNumber];

        for(int i = 0 ; i < 5 ; ++i) {
                forks.add(new Channel<Boolean>());
        }


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

        for (int i = 0; i < philosophersNumber; i++) {
        philosophers[i] = new Philosopher(i, (i + 1) %
            philosophersNumber, "Philo" + i, forks);
        philosophers[i].start();
    }

        Thread.sleep(5000); // the time is in ms. after this time
            we stop the execution


        for(Philosopher philosopher: philosophers)
        {
                philosopher.join();
        }
```

```
        }
}
```

Experiments and results for Dinning Philosophers with channels

1. Random example 1 with 5 philosophers and 5 forks



```
Philo0 is thinking
Philo1 is thinking
Philo2 is thinking
Philo3 is thinking
Philo4 is thinking
Philo2 is eating
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo4 is eating
Philo0 is thinking
Philo3 is eating
Philo4 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo4 is eating
Philo0 is thinking
Philo3 is eating
Philo4 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo4 is eating
Philo0 is thinking
Philo3 is eating
Philo4 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo4 is eating
Philo0 is thinking
Philo3 is eating
Philo4 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo4 is eating
Philo0 is thinking
```

2. Random example 2 with 5 philosophers and 5 forks

```
Philo0 is thinking
Philo1 is thinking
Philo2 is thinking
Philo3 is thinking
Philo4 is thinking
Philo1 is eating
Philo3 is eating
Philo0 is eating
Philo1 is thinking
Philo2 is eating
Philo3 is thinking
Philo4 is eating
Philo0 is thinking
Philo1 is eating
Philo2 is thinking
Philo3 is eating
Philo4 is thinking
Philo0 is eating
Philo1 is thinking
Philo2 is eating
Philo3 is thinking
Philo4 is eating
Philo0 is thinking
Philo1 is eating
Philo2 is thinking
Philo3 is eating
Philo4 is thinking
Philo0 is eating
Philo1 is thinking
Philo2 is eating
Philo3 is thinking
Philo4 is eating
Philo0 is thinking
Philo1 is eating
Philo2 is thinking
Philo3 is eating
Philo4 is thinking
Philo0 is eating
Philo1 is thinking
Philo2 is eating
Philo3 is thinking
Philo4 is eating
Philo0 is thinking
Philo1 is eating
Philo2 is thinking
Philo3 is eating
Philo4 is thinking
Philo0 is eating
Philo1 is thinking
Philo2 is eating
Philo3 is thinking
Philo4 is eating
Philo0 is thinking
```

3. Random example 3 with 4 philosophers and 4 forks

```
Philo0 is thinking
Philo1 is thinking
Philo2 is thinking
Philo3 is thinking
Philo2 is eating
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo3 is eating
Philo0 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo3 is eating
Philo0 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo3 is eating
Philo0 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo3 is eating
Philo0 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo3 is eating
Philo0 is thinking
Philo2 is eating
Philo3 is thinking
Philo1 is eating
Philo2 is thinking
Philo0 is eating
Philo1 is thinking
Philo3 is eating
Philo0 is thinking
Philo2 is eating
Philo3 is thinking
```

Conclusions Producer-Consumer 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

Source code for readers-writers problem with semaphores

```
//---------------------------ReadersWriters.java---------------------------

public class ReadersWriters {
	private int readCount = 0;
	private Semaphore writerSemaphore = new Semaphore(1);
	private Semaphore readerSemaphore = new Semaphore(1);
	private Semaphore orderSemaphore = new Semaphore(1);

	public void write(String writer) {


		// Obtaining access to the resource

		try {
			orderSemaphore.acquire();
			writerSemaphore.acquire();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		orderSemaphore.release();

		// Writing into the resource

		try {
			System.out.println(writer + " is writing");
			Thread.sleep(1000);
			System.out.println(writer + " has stopped
			    writing");
		} catch (InterruptedException e) {
			e.printStackTrace();
		}


		writerSemaphore.release();

	}


	public void read(String reader) {

		// Obtaining access to the queue and the readCount
		    variable

		try {
			orderSemaphore.acquire();
			readerSemaphore.acquire();
```

```
} catch (InterruptedException e) {
        e.printStackTrace();
}


// First reader obtains access to the resource
// so that writers are blocked

if(readCount == 0) {
        try {
                writerSemaphore.acquire();
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}

// Reader starts reading
readCount++;

orderSemaphore.release();
readerSemaphore.release();

// Reading the resource
try {
        System.out.println(reader + " is reading");
        Thread.sleep(1000);
        System.out.println(reader + " has stopped
            reading");
} catch (InterruptedException e1) {
        e1.printStackTrace();
}

// Obtaining access to modify readCount

try {
        readerSemaphore.acquire();
} catch (InterruptedException e) {
        e.printStackTrace();
}

// Reader stops reading
readCount--;


// Releasing access if the current reader is the last
    reader

if(readCount == 0) {
        writerSemaphore.release();
}
```

```
                readerSemaphore.release();
        }
}
```

```
//----------------------------Reader.java----------------------------

public class Reader extends Thread {
        ReadersWriters rw = new ReadersWriters();
        String name;
        Reader(ReadersWriters rw, String name)
        {
                this.rw = rw;
                this.name = name;
        }
        public void run() {
                for(int i = 0 ; i < 10 ; i++)
                {
                        rw.read(name);
                }
    }

}
```

```
//----------------------------Writer.java----------------------------


public class Writer extends Thread {
        ReadersWriters rw = new ReadersWriters();
        String name;
        Writer(ReadersWriters rw, String name)
        {
                this.rw = rw;
                this.name = name;
        }
        public void run() {
                for(int i = 0; i < 10; i++)
                {
                        rw.write(name);
                }
        }
}
```

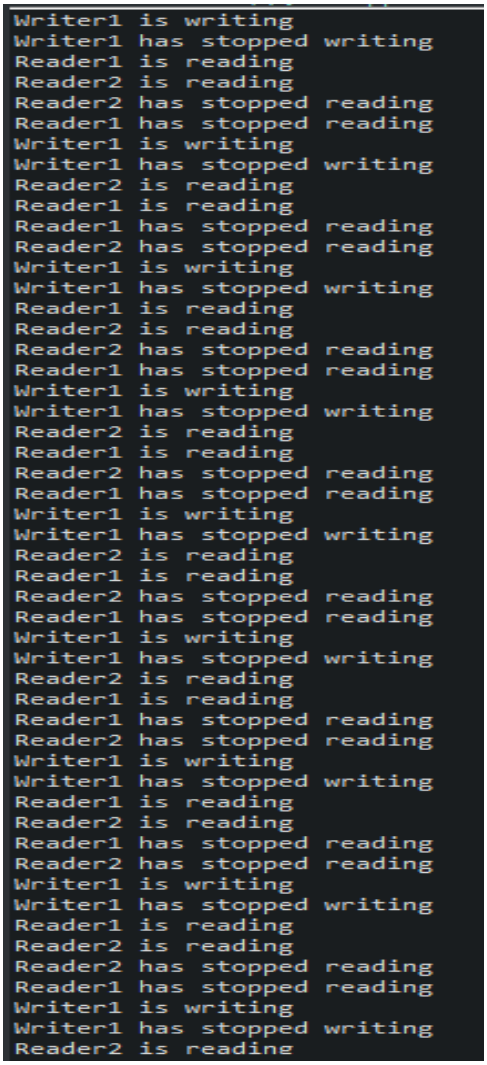
```
//----------------------------Main.java----------------------------

public class Main {
```

```
public static void main(String[] args) {
	ReadersWriters rw = new ReadersWriters();
	int readersNumber = 2;
	int writersNumber = 1;
	int i;


	Reader[] readers = new Reader[readersNumber + 1];


	// Initializing the readers

	for(i = 1 ; i <= readersNumber ; i++) {
		readers[i] = new Reader(rw, "Reader" + i);
	}

	Writer[] writers = new Writer[writersNumber + 1];


	// Initializing the writers

	for(i = 1 ; i <= writersNumber ; i++) {
		writers[i] = new Writer(rw, "Writer" + i);
	}


	// Starting the threads execution

	for(i = 1 ; i <= writersNumber ; i++) {
		writers[i].start();
	}

	for(i = 1 ; i <= readersNumber ; i++) {
		readers[i].start();
	}


	for(i = 1 ; i <= readersNumber ; i++) {
		try {
			readers[i].join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	for(i = 1 ; i <= writersNumber ; i++) {
		try {
			writers[i].join();
		} catch (InterruptedException e) {
			e.printStackTrace();
```

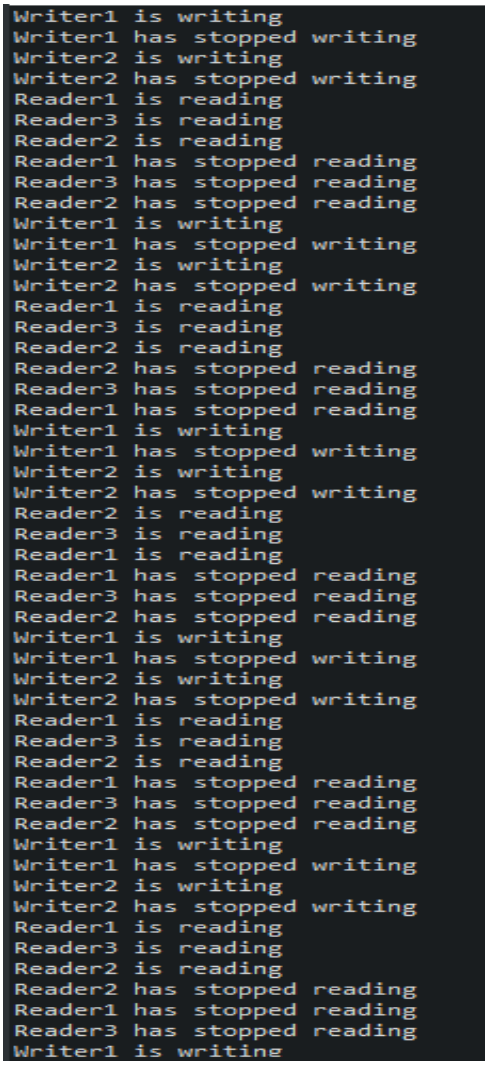
```
                    }
                }

            }

}
```

Experiments and results for readers-writers with semaphores

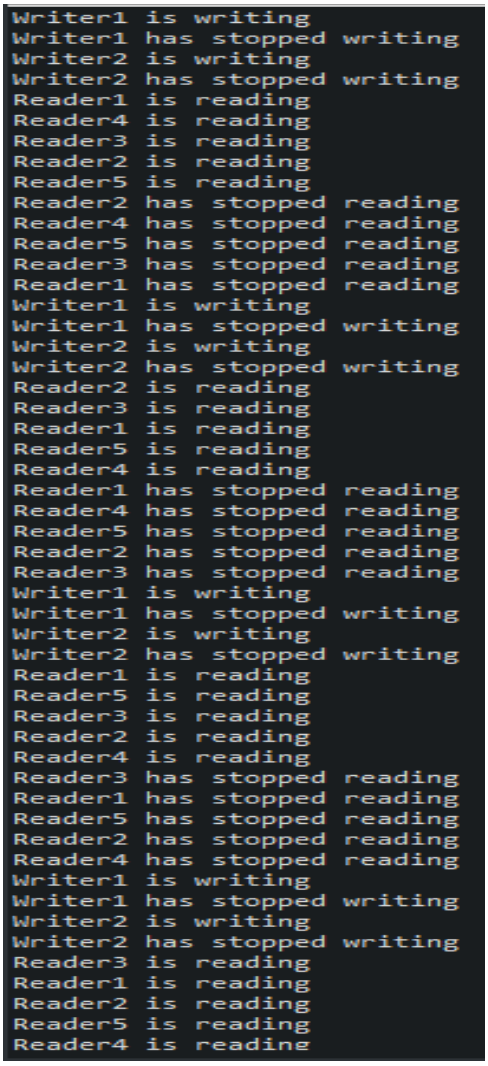
Both the writers and readers will do 10 steps.
1. Random example 1 with 2 readers 1 writer



2. Random example 2 with 3 readers 2 writers



3. Random example 3 with 5 readers 2 writers



Conclusions Readers Writers 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 accessed 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/Readers