Lab Session 2 – Report
Concurrency and Parallelism

-Submitted by
Cyril Naves & Athip Ponna

Exercise 1: Semaphore

Source Code:

Exo1.java:

```java
package masterInt.CandP.exo1;

public class Exo1 {

	public static void main(String[] args) {
		Philosopher[] philosophers = new Philosopher[5];
		Fork[] forks = new Fork[5];

		System.out.println("begin initializing the resources and users...");

		for (int i = 0; i < 5; i++) {

			forks[i] = new Fork(i);
		}

		for (int i = 0; i < 4; i++) {
			philosophers[i] = new Philosopher(i, forks[i], forks[i + 1]);
		}
		philosophers[4] = new Philosopher(4, forks[0], forks[4]);

		Thread[] threads = new Thread[10];
		for (int i = 0; i < 5; i++) {
			threads[i] = new Thread(philosophers[i]);
			threads[i].start();
		}
	}

}
```

Philosopher.java

```java
package masterInt.CandP.exo1;

import java.util.Random;


public class Philosopher implements Runnable {

	private int _id;
	private Fork _leftFork;
	private Fork _rightFork;

	public Philosopher(int id, Fork leftFork, Fork rightFork) {
		// TODO Auto-generated constructor stub
		this._id = id;
		this._leftFork = leftFork;
		this._rightFork = rightFork;
	}

	@Override
	public void run() {
		// TODO Auto-generated method stub
		try {
			while (true) {
				this._leftFork.get();;
				System.out.println(String.format("p%s get fork %s",
this._id,this._leftFork.get_id()));
```

```
                                this._rightFork.get();
                                System.out.println(String.format("p%s get fork %s",
this._id,this._rightFork.get_id()));
                                eat();
                                this._leftFork.put();
                                this._rightFork.put();
                                think();
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

        public void eat() {
                System.out.println(String.format("philosopher %s start to eat.", this._id));
                try {
                        Random r = new Random(50);
                        Thread.sleep(r.nextInt(3000));
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                System.out.println(String.format("philosopher %s finish eating.", this._id));
        }

        public void think() {
                System.out.println(String.format("philosopher %s start to think.", this._id));
                try {
                        Random r = new Random(100);
                        Thread.sleep(r.nextInt(3000));
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                System.out.println(String.format("philosopher %s finish thinking.", this._id));
        }

}
```

Fork.java

```
package masterInt.CandP.exo1;

import java.util.concurrent.Semaphore;

public class Fork {

        private int _id;
        private boolean _isAvailable;

        private final Semaphore available = new Semaphore(1,true);

        public int get_id() {
                return _id;
        }

        public void set_id(int _id) {
                this._id = _id;
        }

        public boolean is_Available() {
                synchronized (this) {
                        return _isAvailable;
                }
        }

        public void set_Available(boolean _isAvailable) {
            synchronized (this) {
                  this._isAvailable = _isAvailable;
            }
      }

      public Fork(int id) {
            // TODO Auto-generated constructor stub
            this._id = id;
            this._isAvailable = true;
      }

      //synchronized

      public synchronized void get() throws InterruptedException
      {
            while(!this.is_Available())
                  wait();
            this.set_Available(false);
      }

      public synchronized void put()
      {
            this.set_Available(true);
            notifyAll();
      }

      //semaphore version
      public void get_s() throws InterruptedException
      {
            this.available.acquire();
      }

      public void put_s()
      {
            this.available.release();
      }

}
```

Output:

```
begin initializing the resources and users...
p2 get fork 2
p3 get fork 3
p1 get fork 1
p0 get fork 0
p3 get fork 4
philosopher 3 start to eat.
philosopher 3 finish eating.
philosopher 3 start to think.
p2 get fork 3
philosopher 2 start to eat.
philosopher 2 finish eating.
p1 get fork 2
philosopher 2 start to think.
philosopher 1 start to eat.
philosopher 3 finish thinking.
p3 get fork 3
p3 get fork 4
philosopher 3 start to eat.
philosopher 1 finish eating.
philosopher 1 start to think.
p0 get fork 1
philosopher 0 start to eat.
philosopher 2 finish thinking.
philosopher 3 finish eating.
p2 get fork 2
```

```
philosopher 3 start to think.
p2 get fork 3
philosopher 2 start to eat.
philosopher 0 finish eating.
philosopher 0 start to think.
p4 get fork 0
p4 get fork 4
philosopher 4 start to eat.
philosopher 1 finish thinking.
p1 get fork 1
philosopher 2 finish eating.
philosopher 2 start to think.
p1 get fork 2
philosopher 1 start to eat.
philosopher 4 finish eating.
philosopher 4 start to think.
philosopher 3 finish thinking.
p3 get fork 3
p3 get fork 4
philosopher 3 start to eat.
philosopher 1 finish eating.
philosopher 0 finish thinking.
philosopher 1 start to think.
p0 get fork 0
p0 get fork 1
philosopher 0 start to eat.
philosopher 3 finish eating.
philosopher 3 start to think.
philosopher 2 finish thinking.
p2 get fork 2
p2 get fork 3
philosopher 2 start to eat.
philosopher 4 finish thinking.
philosopher 0 finish eating.
philosopher 0 start to think.
p4 get fork 0
p4 get fork 4
philosopher 4 start to eat.
philosopher 1 finish thinking.
p1 get fork 1
philosopher 2 finish eating.
philosopher 2 start to think.
p1 get fork 2
philosopher 1 start to eat.
philosopher 4 finish eating.
philosopher 4 start to think.
philosopher 3 finish thinking.
p3 get fork 3
p3 get fork 4
philosopher 3 start to eat.
philosopher 0 finish thinking.
p0 get fork 0
philosopher 1 finish eating.
philosopher 1 start to think.
p0 get fork 1
```

1/ Define the *fork* to be a class with only a boolean state: ”available” or ”currently used” and the associated getters and setters.
2/ Implements the dining philosophers problem in a way that avoid deadlock.
3/ Have you *synchronized* the accesses to the forks?
4/ Change the implementation in order to use the class **java.util.concurrent.Semaphore**.

Summary of Dining Philosophers Problem with Synchronization and Semaphore:

1) In Dining Philosophers Problem the forks are created through fork class which are 5 in no
2) Then for each of philosphers the fork are assigned in(0,1) (1,2), (2,3),(3,4) and (4,5) pairs of forks assigned to each of the philosopher.
3) Then each philosopher thread and implementation of Philosopher Class implementing Runnable Interface is made to run and get the fork and release the fork alternately through synchronized methods in Fork Class
4) Alternatively each philosopher when eating gets the fork only when is available and then thinks when the availability of the fork is true. A wait () method is used for a thread to wait until the fork is available thereby avoiding deadlock
5) **Synchronization of the access to the Forks:** Synchronized keyword enables to create a synchronized instance method of get and put of fork thereby when in a particular object instance only a single thread can execute and when another tries to get the fork then it is put in blocking until the existing thread is complete
6) A Semaphore Implementation which is a thread synchronization construct which is used to guard resources and send signal between threads is achieved by created a counter with value 1 and then it is then alternate between acquire() and release() method to deal with the resources of a thread.

```
private final Semaphore available = new Semaphore(1,true);
…


public void get_s() throws InterruptedException
{
       this.available.acquire();
}

public void put_s()
{
       this.available.release();
}
```

```
Output:begin initializing the resources and users...
```

```
p0 get fork 0
p1 get fork 1
p2 get fork 2
p3 get fork 3
p3 get fork 4
philosopher 3 start to eat.
philosopher 3 finish eating.
philosopher 3 start to think.
p2 get fork 3
philosopher 2 start to eat.
philosopher 2 finish eating.
p1 get fork 2
philosopher 2 start to think.
philosopher 1 start to eat.
philosopher 3 finish thinking.
p3 get fork 3
p3 get fork 4
philosopher 3 start to eat.
philosopher 1 finish eating.
philosopher 1 start to think.
```

Exercise 2: Readers writers with priority

Let us consider a table of 1000 bytes that will simulate the behavior of a memory. The access to this memory are the following:

1. There are three writers that are reading data from text files and writing them in memory at he next available slot.
2. There are three readers that read the next available data in the memory and write them in (another) text files.
3. The writers have priority over the readers. Of course, when the memory is full, writers cannot write anymore.
4. The three readers as well as the three writers have there own hierarchy of priorities.
5. The three readers as well as the three writers alternate between sleeping for a random time and writing 100 bytes inthe memory.

Source:

Exo2.java

```
package masterInt.CandP.exo2;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.concurrent.Semaphore;

public class Exo2 {
public static void main(String[] args) {

        final int READERS = 3;
        final int WRITERS = 3;
        Memory memory = new Memory();

        Semaphore inputS = new Semaphore(1);
        Semaphore outputS = new Semaphore(1);
        File file =null;
        FileInputStream in = null;

        /*int[] priorityTable = new int[3];
        for(int p : priorityTable)
                p=0;*/
        //System.out.println(new File(".").getAbsolutePath());

        try {
                file = new File("src\\masterInt\\CandP\\exo2\\input.txt");
                in = new FileInputStream(file);
        } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        for (int i = 0; i < READERS; i++) {
                new Reader(memory,outputS).start();
                Thread.currentThread().setPriority(6+i);
        }
        for (int i = 0; i < WRITERS; i++) {
                new Writer(memory,inputS,in).start();
                Thread.currentThread().setPriority(i+1);

        }

}
}
```

Memory.java:

```
package masterInt.CandP.exo2;

import java.util.Random;
import java.util.concurrent.Semaphore;

public class Memory {
private int writers; // number of active readers

private final int MEMORYSIZE = 1000;
byte[] tab = new byte[MEMORYSIZE];

private int writeIndex = 0;
private int readIndex = 0;

private Semaphore memoryS = new Semaphore(1);

public Memory() {
		this.writers = 0;
		this.writeIndex = 0;
		this.readIndex = 0;
}

public synchronized byte[] read(int number) {
		while (this.writers != 0) {
				try {
						this.wait();
				} catch (InterruptedException e) {
				}
		}
		int size =100;
		//int size = new Random(10).nextInt(1000);
		byte[] buffer = new byte[size];
		System.out.println("Reader " + number + " read " +size + " from memory");
		// read from memory

		int temp = readIndex;
		if (readIndex + 100 > writeIndex) {
				try {
						this.wait();
				} catch (InterruptedException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
				}
		}
		readIndex = (readIndex + 100);

		for (int i = 0; i < 100; i++) {
				buffer[i] = tab[(temp++) % MEMORYSIZE];
		}

		this.notifyAll();
		return buffer;
}

public synchronized void write(int number, byte[] bytes) {
		synchronized (this) {
				this.writers++;
		}

		int i = 0;
		while (true) {
				if (i == bytes.length)
						break;

				while (writeIndex - MEMORYSIZE >= readIndex) {
```

```
				try {
					this.wait();
				} catch (InterruptedException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
			}
			tab[(writeIndex++) % MEMORYSIZE] = bytes[i++];
		}

		synchronized (this) {
			this.writers--;
			if (this.writers == 0) {
				this.notifyAll();
			}
		}
}
}
```

Reader.java

```
package masterInt.CandP.exo2;

import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.concurrent.Semaphore;

public class Reader extends Thread {
private static int readers = 0; // number of readers

private int number;
private Memory memory;
Semaphore output;

public Reader(Memory memory,Semaphore s) {
	this.memory = memory;
	this.number = Reader.readers++;
	this.output = s;
}

public void run() {
	while (true) {
		final int DELAY = 5000;
		try {
			Thread.sleep((int) (Math.random() * DELAY));
		} catch (InterruptedException e) {
		}

		try {
			this.output.acquire();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		byte[] bytes = this.memory.read(this.number);


		writeToOutput(bytes);
		this.output.release();
	}
}

public void writeToOutput(byte[] bytes) {

	System.out.println("Reader " + number + " write to output.");
	BufferedWriter out = null;
	try {
		out = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("src\\\\masterInt\\\\CandP\\\\exo2\\\\output.txt", true)));
		out.write(new String(bytes));
	} catch (Exception e) {
		e.printStackTrace();
	} finally {
		try {
			out.close();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}

}
```

Writer.java

```
package masterInt.CandP.exo2;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Random;
import java.util.concurrent.Semaphore;

public class Writer extends Thread {
private static int writers = 0; // number of writers

private int number;
private Memory memory;
Semaphore input;
FileInputStream fis;

public Writer(Memory memory,Semaphore s,FileInputStream fis) {
	this.memory = memory;
	this.number = Writer.writers++;
	this.input = s;
	this.fis= fis;
}

public void run() {
	while (true) {

		try {
			input.acquire();
		} catch (InterruptedException e1) {
			// TODO Auto-generated catch block
			e1.printStackTrace();
		}
		byte[] bytes = readFromInput();
```

```
                final int DELAY = 5000;
                try {
                        Thread.sleep((int) (Math.random() * DELAY));
                } catch (InterruptedException e) {
                }

                if(bytes==null)
                {
                        System.out.println("Writer " + number + "stop!");
                        input.release();
                        break;
                }
                this.memory.write(this.number,bytes);
                input.release();
        }
}

public byte[] readFromInput()
{


    int length = 0;
    //int size=new Random(10).nextInt(1000);
    int size =100;
    byte[] bytes = new byte[size];
    System.out.println("Writer " + number + " read " +size + " byte from input");
    try {
                length = this.fis.read(bytes);
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
    if(length >0)
    {
        //System.out.println("Writer " + number + "\r\n\r\n"+new String(bytes));
        return bytes;
    }
    else
    {
        System.out.println("Writer " + number + "empty input");
        return null;
    }
}
}
```

Output:

```
Writer 1 read 100 byte from input
Reader 2 read 100 from memory
Writer 1 read 100 byte from input
Reader 2 write to output.
Writer 1empty input
Reader 1 read 100 from memory
Writer 1stop!
Writer 2 read 100 byte from input
Writer 2empty input
Writer 2stop!
Writer 0 read 100 byte from input
Writer 0empty input
Writer 0stop!
```

1/ Implement the system as described

2/ Let the readers and writers work on a random size (≤ 1000) packet of data instead of 100.
3/ Consider the following extra requirement: Reader *i* reads only data from Writer *i* (with $i \in \{1,2,3\}$).

Summary of Readers Writers Implementation with priority:

1) Here a Memory Class with the required bytes is assigned which is of the value 1000 and then the required operation of read and write in a synchronized method instance is created.
2) Reader and writer Classes which implements the runnable interface is with the required semaphore to acquire and release resource of the memory.
3) Also when the threads are instantiated, writer threads are then set a priority higher (6 to 8)than reader threads (1 to 3) just to maintain hierarchy of thread each having their own priority hierarchy level.
4) When each of the thread related to reader and writer is executed then semaphore is alternated between reading and writing by acquiring and releasing the resource.
5) Thread for the current execution is made to sleep for definite time period to avoid the deadlock scenario between the reader and writer threads.