Documentation
Project #4

1.1/Pseudocode solution:

values from ReaderWritersProblem.class:

Semaphore readlock {initial value = 1}

Semaphore writelock {initial value = 1}

Int reacount {initial value = 0}

```
public class ReaderWritersProblem {
    static Semaphore readLock = new Semaphore(1);
    static Semaphore writeLock = new Semaphore(1);
    volatile static int readCount = 0;
}
```

values from pos.class:

Read read;

Write write;

Threads:

T1 = read

T2= write

T3= write

T4= read

T5=write

```
public class Pos {

 public static void main(String[] args) throws Exception {
        Read read = new Read();
        Write write = new Write();

        Thread t1 = new Thread( read);
        t1.setName("thread1");
        Thread t2 = new Thread( write);
        t2.setName("thread2");
        Thread t3 = new Thread( write);
        t3.setName("thread3");
        Thread t4 = new Thread( read);
        t4.setName("thread4");
        Thread t5 = new Thread( write);
        t5.setName("thread5");
        //Thread t6 = new Thread( write);
        //t6.setName("thread6");

        t1.start();
        t3.start();
        t2.start();
        t4.start();
        t5.start();
        //t6.start();
    }
}
```

Read class:

```
public class Read extends ReaderWritersProblem implements Runnable{
     public void run() {
            try {
                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount++;
                }
                if (readCount == 1) {
                    writeLock.acquire();
                }
                readLock.release();

                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING");

                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount--;
                }
                if(readCount == 0) {
                    writeLock.release();
                }
                readLock.release();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }

}
```

---

try {

wait(reader) ;

Increase read count by 1;

if readCount == 1 then wait(writer);

signal(reader);

// Reading is performed…

wait(reader);

```
        decrease readCount by1;
            If readCount == 0 then signal(writer);
        signal(reader);
    } catch (InterruptedException e) {
            print(e.getMessage());
```

write class:

```
public class Write extends ReaderWritersProblem implements Runnable  {
     public void run() {
            try {
                writeLock.acquire();
                System.out.println("Thread "+ Thread.currentThread().getName() + " is WRITING");
                Thread.sleep(2500);
                System.out.println("Thread "+ Thread.currentThread().getName() + " has finished WRITING");
                writeLock.release();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
}
```

```
try {
            wait(writer);
           //writing is performed
            signal(writer);
        } catch (InterruptedException e) {
              print(e.getMessage());
```

}

.....

1.2/ Examples of DeadLock:

- writer deadlock:

```
public class Read extends ReaderWritersProblem implements Runnable{
     public void run() {
            try {
                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount++;
                }
                if (readCount == 1) {
                    writeLock.acquire();
                }
                readLock.release();

                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING");

                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount--;
                }

                readLock.release();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
```

If the reader takes the writer's lock and does not release it, the writer enters a deadlock waiting for the lock to be released.

-Reader DeadLock:

```
public class Read extends ReaderWritersProblem implements Runnable{
     public void run() {
            try {
                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount++;
                }
                if (readCount == 1) {
                    writeLock.acquire();
                }


                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING");


                synchronized(ReaderWritersProblem.class) {
                    readCount--;
                }


            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }

}
```

If a reader takes the read lock and does not release it, other readers will be deadlocked waiting for the lock to be released.

1.3/How did solves the Deadlock:

- Writer deadlock:

The reader releases the writer lock.

```
public class Read extends ReaderWritersProblem implements Runnable{
     public void run() {
            try {
                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount++;
                }
                if (readCount == 1) {
                    writeLock.acquire();
                }
                readLock.release();

                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING");

                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount--;
                }
                if(readCount == 0) {
                    writeLock.release();
                }
                readLock.release();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }

}
```

- Reader deadlock:

The reader releases the read lock.

```
public class Read extends ReaderWritersProblem implements Runnable{
     public void run() {
            try {
                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount++;
                }
                if (readCount == 1) {
                    writeLock.acquire();
                }
                readLock.release();

                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING");

                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount--;
                }
                if(readCount == 0) {
                    writeLock.release();
                }
                readLock.release();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }

}
```

1.4 / Examples of starvation:

- writer starvation:

```
public class Read extends ReaderWritersProblem implements Runnable{
     public void run() {
            try {
                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount++;
                }
                if (readCount == 1) {
                    writeLock.acquire();
                }
                readLock.release();

                System.out.println("Thread "+Thread.currentThread().getName() + " is READING");
                Thread.sleep(1500);
                System.out.println("Thread "+Thread.currentThread().getName() + " has FINISHED READING");

                readLock.acquire();
                synchronized(ReaderWritersProblem.class) {
                    readCount--;
                }

                readLock.release();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
```

When the writer arrives and tries to do the write, it gets blocked. The writer must then wait until all readers have left. Once Reader B arrives, the two readers take turns leaving and re-entering. Since at least one reader is always in the system, the writer is blocked indefinitely, a situation known as starvation

1.5/ How did solves the starvation:

Writer starvation:

```
public synchronized void startWrite(){

        while(writing||readers>0){
                waitingWriters++}
        try{
           wait();}
        catch(interruptedException ex){
           waitingWriters--;
                           {
                             {
         waitingWriters--;
         writing=true;
                                 {
```

A condition is added when the number of readers reaches 0, the writer is allowed to enter.

- Explanation for real world application and how and how did apply the problem :

The Readers and Writers problem is useful for modeling processes which are competing for a limited shared resource. A practical example of a Readers and Writers problem is an airline reservation system consisting of a huge data base with many processes that read and write the data. Reading information from the data base will not cause a problem since no data is changed. The problem lies in writing information to the data base. If no constraints are put on access to the data base, data may change at any moment.