**Reader Writer Problem**

**Project Description:**

A classical Reader-Writer Problem is a situation where a data structure can be read and modified simultaneously by concurrent threads. Only one Writer is allowed access the critical area at any moment in time. When no Writer is active any number of Readers can access the critical area. To allow concurrent threads mutually exclusive access to some critical data structure, a mutually exclusive object, or mutex, is used. As an implementation mutex is a special case of a more generic synchronisation concept – semaphore. A simple image of a semaphore is a positive number which allows increment by one and decrement by one operations. If a decrement operation is invoked by a thread on a semaphore whose value is zero, the thread blocks until another thread increments the semaphore and the program must be free from the starvation and the deadlock.

**Solution pseudocode:**

1) First class:Main

1- Create instance from class Read

2- Create instance from class Write

3- For loop to give number to the thread

4- Create instance from class Random

5- Pass the instance to the Thread class Read if it's even and to Write if it's odd

**The code :**

public class Main {

public static void main(String[] args) {

ExecutorService pool = Executors.newCachedThreadPool();

SharedResource Resource = new SharedResource();

int reader = 1;

int writer = 1;

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

if(ThreadLocalRandom.current().nextInt(2) == 0) {

pool.execute(new ReaderThread(Resource, "Reader-" + reader++));

}

else {

pool.execute(new WriterThread(Resource, "Writer-" + writer++));

}

}

pool.shutdown();

}

}

2) Second class:SharedResource

This is an array contain the number of places that we can write in :

**The code :**

public class SharedResource{

private final HashMap<Integer,String> places;

SharedResource()

{

this.places = new HashMap<>();

for(int i=1;i<=25;i++) this.places.put(i,"0");

}

public HashMap<Integer, String> Getplaces() {

return this.places;

}

public int placeCount() {

return this.places.size();

}

}

3)Third class:Reader Thread

1.The reader requests entry to the critical section

2.It takes (Acquire) the in and then take (Acquire) the read mutex and increase the readCount

3.If this reader is the first to enter then

4.It takes (Acquire) the writeLock to prevent any other writers from entering if any other reader is present.

5.It will leave (release) the read mutex indicating that any new reader may enter while others are currently reading lock

6. it will leave(release) in

7.It enters the critical section and perform reading and it takes (Acquire) the reader mutex to decrease the readCount

8. It checks to see whether there are not anymore readers within and if there are, it leaves (release) the writeLock , indicating that the writer can now enter the critical region.

9. It will leave (release) the read mutex

**The code :**

public class ReaderThread implements Runnable{

private final SharedResource Resource;

private final String name;

Semaphore readerMutex;

Semaphore writerMutex;

Semaphore in ;

AtomicInteger readersCount;

public ReaderThread(SharedResource Resource, String name) {

this.Resource = Resource;

this.name = name;

in = new Semaphore(1);

readerMutex = new Semaphore(1);

writerMutex = new Semaphore(1);

readersCount = new AtomicInteger(0);

}

public void run() {

final String threadName = name + "(" + Thread.currentThread().getName() + ")";

boolean running = true;

while (running) {

try {

for (int i=1; i<Resource.placeCount();i++) {

Thread.sleep(ThreadLocalRandom.current().nextInt(3000));

read(threadName);

}

}

catch (InterruptedException e) {

running = false;

}

}

}

public void read(String name) throws InterruptedException

{

in.acquire();

readerMutex.acquire(); //only one reader could modify writerMutex

if (readersCount.incrementAndGet() == 1) {

//the first reader that enters the critical section blocks all writers or waits until the writer has finished

writerMutex.acquire();

}

readerMutex.release();

in.release();

//[Critical Section]

String time = "Time: " + LocalTime.now() +"\n";

System.out.println(time + name + " observe the resource :");

HashMap<Integer,String> s = Resource.Getplaces() ;

s.forEach((k,v) -> System.out.printf("Seat No " + k + ": " + v+ " "));

System.out.println();

System.out.println("-------------------------------------------------------------------------------");

System.out.println();

//[/Critical Section]

readerMutex.acquire();

if(readersCount.decrementAndGet() == 0) {

//the last reader that leaves the critical section unblocks all writers

writerMutex.release();

}

readerMutex.release();

}

}

4)Fourth class:Writer thread

1. The writer requests entry to the critical section

2. It takes (Acquire) the in

3. It takes (Acquire) the WriterMutex

4. it is entering the critical section and perform writing and leave (release) the writeLock after writing

5. It will leave (release) the WriterMutex

**The code :**

public class WriterThread implements Runnable {

private final SharedResource Resource;

private final String name;

Semaphore writerMutex;

Semaphore in ;

public WriterThread(SharedResource Resource, String name) {

this.Resource = Resource;

this.name = name;

in = new Semaphore(1);

writerMutex = new Semaphore(1);

}

@Override

public void run() {

final ThreadLocalRandom localRandom = ThreadLocalRandom.current();

final String threadName = name + "(" + Thread.currentThread().getName() + ")";

final long threadId = Thread.currentThread().getId();

boolean running = true;

while(running) {

try {

if (localRandom.nextInt(0, 9) < 7) {

// Write

Thread.sleep(localRandom.nextInt(3000));

Write(

threadName,

localRandom.nextInt(1, 25),

threadId);

}

else {

// Delete

Thread.sleep(localRandom.nextInt(3000));

Delete(

threadName,

localRandom.nextInt(1, 25),

threadId);

}

}

catch (InterruptedException e) {

running = false;

}

}

}

public void Write(String name, int placeNo, long customerId) throws InterruptedException

{

in.acquire();

writerMutex.acquire();

//[Critical Section]

String time = "Time: " + LocalTime.now() + "\n";

System.out.println(time + name + " tries to Write " + placeNo + "\n");

Thread.sleep(ThreadLocalRandom.current().nextInt(1000));

if (Resource.Getplaces().containsKey(placeNo)) {

if(Resource.Getplaces().get(placeNo) == "0") {

System.out.println(time + name + " write done " + placeNo + " successfully\n");

Resource.Getplaces().put(placeNo,String.valueOf(customerId));

}

else {

System.out.println(time + name + " could not write " + placeNo + " the place is occupied\n");

}

}

//[Critical Section]

writerMutex.release();

in.release();

}

public void Delete(String name, int placeNo, long customerId) throws InterruptedException

{

in.acquire();

writerMutex.acquire();

//[Critical Section]

String time = "Time: " + LocalTime.now() +"\n";

System.out.println(time + name + " tries to delete number " + placeNo + "\n");

Thread.sleep(ThreadLocalRandom.current().nextInt(1000));

if (Resource.Getplaces().containsKey(placeNo)) {

if(Resource.Getplaces().get(placeNo) == String.valueOf(customerId)) {

System.out.println(time + name + " number deleted " + placeNo + "\n");

System.out.println();

Resource.Getplaces().put(placeNo, "0");

}

else {

System.out.println(time + name + " could not delete number " + placeNo + "\n");

}

}

//[Critical Section]

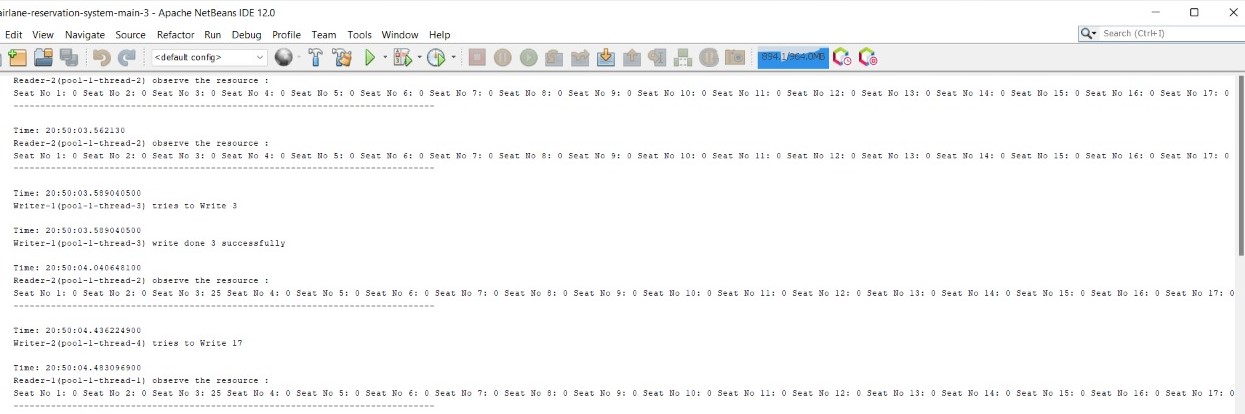
writerMutex.release();

in.release();

}

}

**Example of the run :**





There is no Dead lock will occur in the reader writer problem except if we change the structure of the reader writer problem to that :

When we don’t leave (release) the readlock :

While(true)

{

r\_mutex.acquire();

w\_mutex.acquire();

/\* writing is performed \*/

w\_mutex.release();

}

When we don’t leave (release) a writelock :

While(true)

{

mutex.acquire();

readCount++;

if (readCount == 1) {

w\_mutex.acquire();

}

mutex.release();

r\_mutex.acquire();

/\* reading is performed \*/

r\_mutex.release();

mutex.acquire();

readCount--;

mutex.release();

}

We can solve this problem by :

1. Semaphore ReaderMutex initialized to 1
2. Semaphore WriterMutex initialized to 1

Examples of starvation:

1. The structure of a writer’s process

while (true) {

rw\_mutex.acquire();

/\* writing is performed \*/

rw\_mutex.release();

}

1. The structure of a reader’s process

While(true){

mutex.acquire();

read\_count++;

if (read\_count == 1)

rw\_mutex.acquire();

mutex.release();

/\* reading is performed \*/

mutex.acquire();

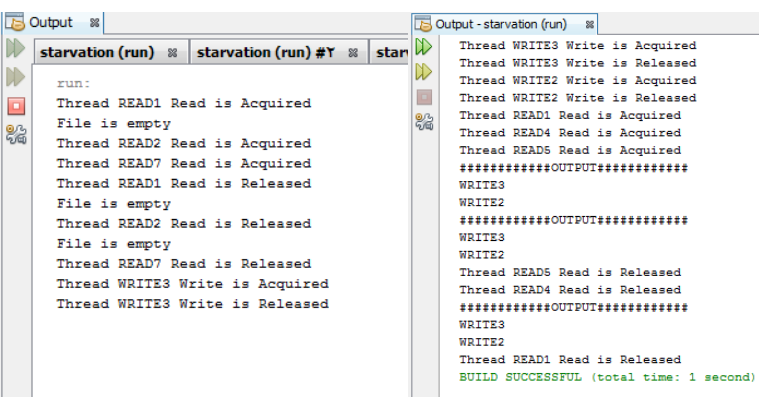
read count--;

if (read\_count == 0)

rw\_mutex.release();

mutex.release();

}



**Readers-Writers Problem Variations**

• First variation – no reader kept waiting unless the writer has permission to use a shared object (Writer will starve)

• the Second variation – once a writer is ready, it performs the write ASAP. In other words, if a writer is waiting to access the object, no new readers may start reading. (Reader will starve)

• Both may have starvation leading to even more variations

**How did solve starvation**

• Data set (The shared file)

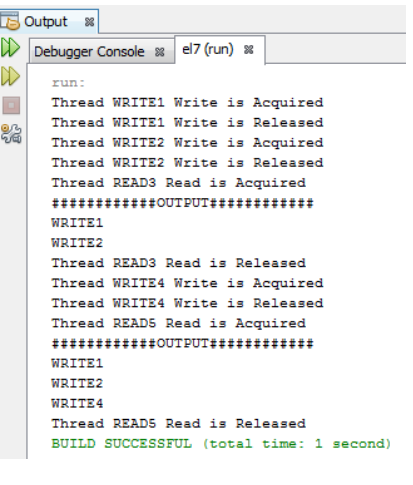
• Semaphore in initialized to 1

• Semaphore readMutex initialized to 1 (reader access)

• Semaphore writeMutex initialized to 1 (writer access)

• Integer read\_count initialized to 0 (how many processes are reading object)

**Explanation :**

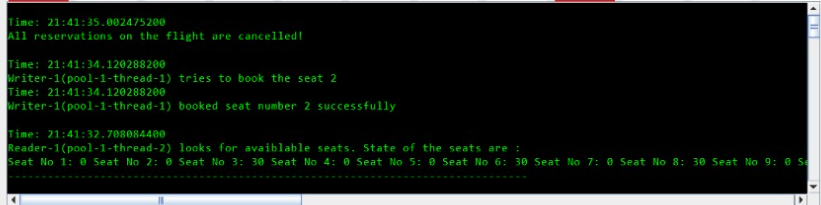


**Explanation for real world application and how did apply the** **problem** - We choose the Airline systems there are users who wants to book ticket ( write ) and who wants to seek ( read ) - The main idea when someone try to book there is no others can see or book too (only one writer) and more than one can read in same time . - and when someone try to (book or see) don’t prevent other people who try to (book or see) to do this which is mean deadlock - and when someone try to book don't take all the booking requests after him and let the seeking requests till the end --- and when someone try to see don't take all the seeking requests after him and let the booking requests till the end which is mean starvation - We have 5 tickets and there's a number of people try to book ( write ) we assume that they = 4 and the people who is see ( read ) = 7 - Like this

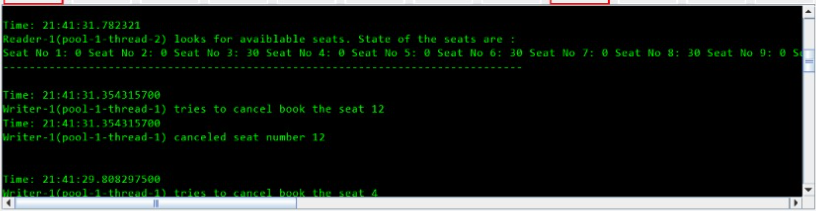
-We have 24 seat shared resources and reader and writer threads try to access it.



-Writer 1 (thread 1) try to book seat 2 and reader 1 (thread 2) try to observe the available seat



-Writer 1 (thread 1) try to cancel seat 12



-Writer 1 (thread 1) try to book seat 3 but it’s already booked

