Readers and Writers Problem

Project Description:

Readers Writers Problem

This problem occurs when many threads of execution try to access the same shared resources at a time. There are N-readers to read data and K-Writers to write data to shared resources. Write Java Multi-threading program to solve the Readers and Writers Problem.

The solution must be free from Deadlock and starvation

Solution pseudocode:

1) First class:Main

1. Create instance from class Random
2. Create instance from class Read
3. Create instance from class Write
4. For loop to give number to the thread
5. Pass the instance to the Thread class Read if it's even and to Write if it's odd
6. w.start(); or r.start();

```
int num;

Random random=new Random();

Read read = new Read();

Write write = new Write();

for (int i = 0; i < 10; i++) {

num =  1+random.nextInt(10);

if ((num%2==0)){

Thread r=  new Thread(read);
```

```
r.setName("READ"+(i+1));

r.start();

}

else{

Thread w = new Thread(write);

w.setName("WRITE"+(i+1));

w.start();

} }
```

2) Second class: ReaderandWriter

There a shared class has the shared variables (Semaphores)

5 semaphores and 2 integers

```
static Semaphore mutex1 = new Semaphore(1);

static Semaphore mutex2 = new Semaphore(1);

static Semaphore readLock = new Semaphore(1);

 static Semaphore writeLock = new Semaphore(1);

 static int readCount = 0;

 static int writeCount = 0;
```

3)Third class:Reader process

1.The reader requests entry to the critical section

2.It takes (Acquire) the *readLock* and then take (Acquire) the *mutex1*lock to increase the *readCount*

3.If this reader is the first to enter then

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

5.It will leave (release) the *mutex1lock* indicating that any new reader may enter while others are currently reading lock and leave (release) *readLock*

6.It enters the critical section and perform reading and it takes (Acquire) the *mutex1lock* to decrease the *readCount*

7. It checks to see whether there are not anymore readers within and if there are, it leaves (release) the write*Lock* , indicating that the writer can now enter the critical region.
8. It will leave (release) the *mutex1lock*

```
While(true){
readLock.acquire();
            mutex1.acquire();
             readCount++;
             if (readCount == 1) {
             writeLock.acquire();
             }
                   mutex1.release();
                  ////////////////////////////////
                     Critical section
                  ////////////////////////////////
              readLock.release();
             mutex1.acquire();
            readCount--;
             if (readCount == 0) {
             writeLock.release();
             }
             mutex1.release();
}
```

4)Fourth class:Writer process

1. The writer requests entry to the critical section

2. It takes (Acquire) the *mutex2lock* to increase the *writeCount*

3. if this writer is the first to enter then

4. it takes (Acquire) the *readLock to prevent any other* readers from entering if any other writers are present.

5. It will leave (release) the *mutex2lock* indicating that any new writers may enter

6. Before entering the critical section it takes (Acquire) *writeLock*

7. It enters the critical section and perform writing and leave (release) the *writeLock* after writing

8. it takes (Acquire) the *mutex2lock* to decrease the *writeCount*

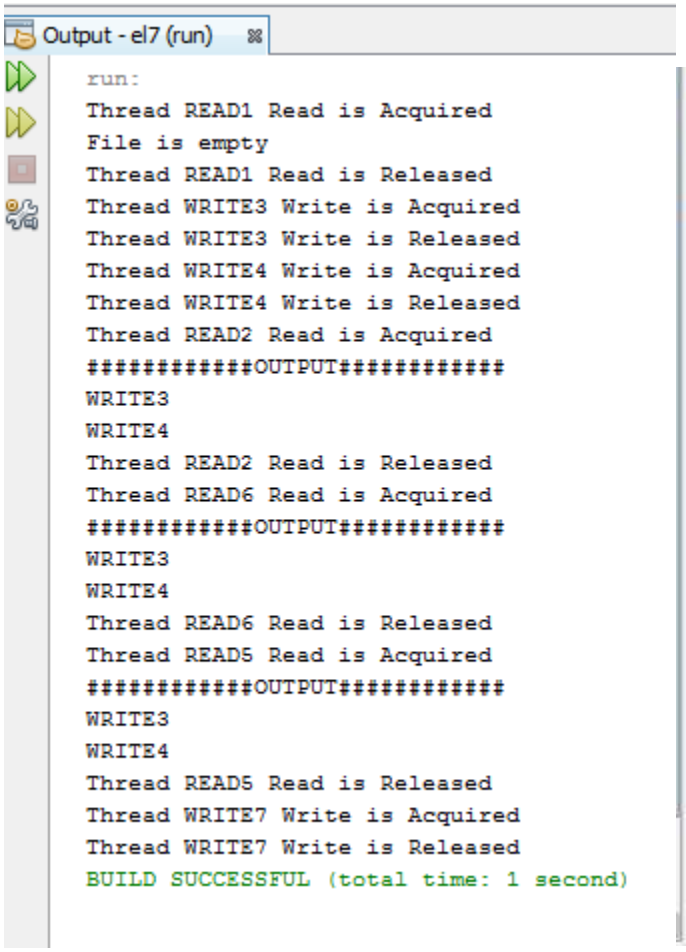
9. It checks to see whether there are not anymore writers within and if there are, it leaves (release) the read*Lock* , indicating that the readers can now enter the critical region.

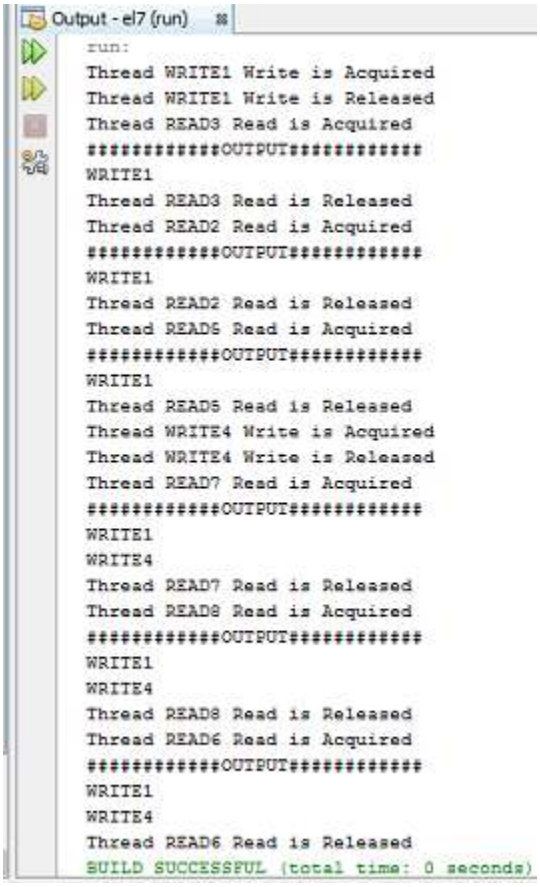
10. It will leave (release) the *mutex2lock*

```
While(true){
mutex2.acquire();
        writeCount++;
         if(writeCount==1){
          readLock.acquire();
         }
         mutex2.release();
        writeLock.acquire();
           ////////////////////////////////
                Critical section
           ////////////////////////////////
```

```
            writeLock.release();
          mutex2.acquire();
        writeCount--;
        if(writeCount==0){
         readLock.release();
        }
        mutex2.release();
}
```

Example of the run





Examples of Deadlock

When we don't leave (release) the readlock

1) The structure of a writer's process

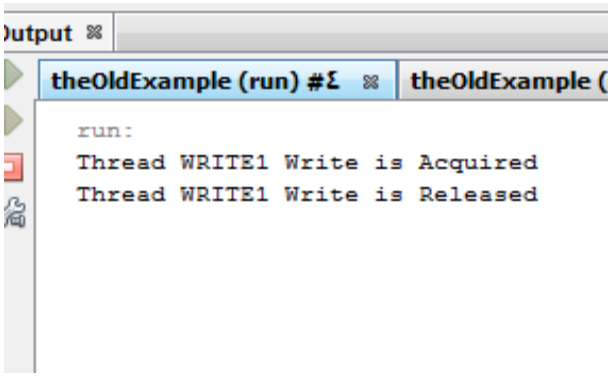
```
While(true){
r_mutex.acquire();
        w_mutex.acquire();
                                    ...
                            /* writing is performed */
                                    ...
        w_mutex.release();
}
```

When we don't leave (release) a writelock

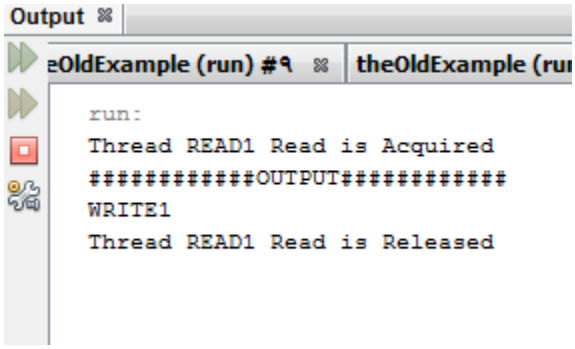
2) The structure of a reader's process

```
        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();
}
```



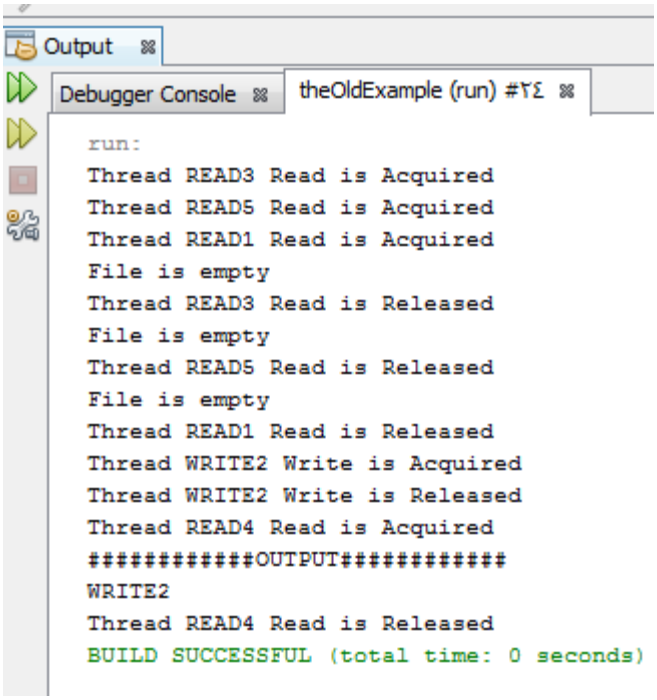
$ The writer entered the critical section and wrote in the file, and then he prevented the rest of the writers and the readers from entering the critical section



$ The reader entered the critical section and read from the file, and then he prevented the rest of the readers and the writers from entering the critical section.

How did solve Deadlock

- Data set (The shared file)
- Semaphore mutex initialized to 1 (controls access to read_count)
- Semaphore rw_mutex initialized to 1 (writer access)
- Integer read_count initialized to 0 (how many processes are reading object)



Examples of starvation:

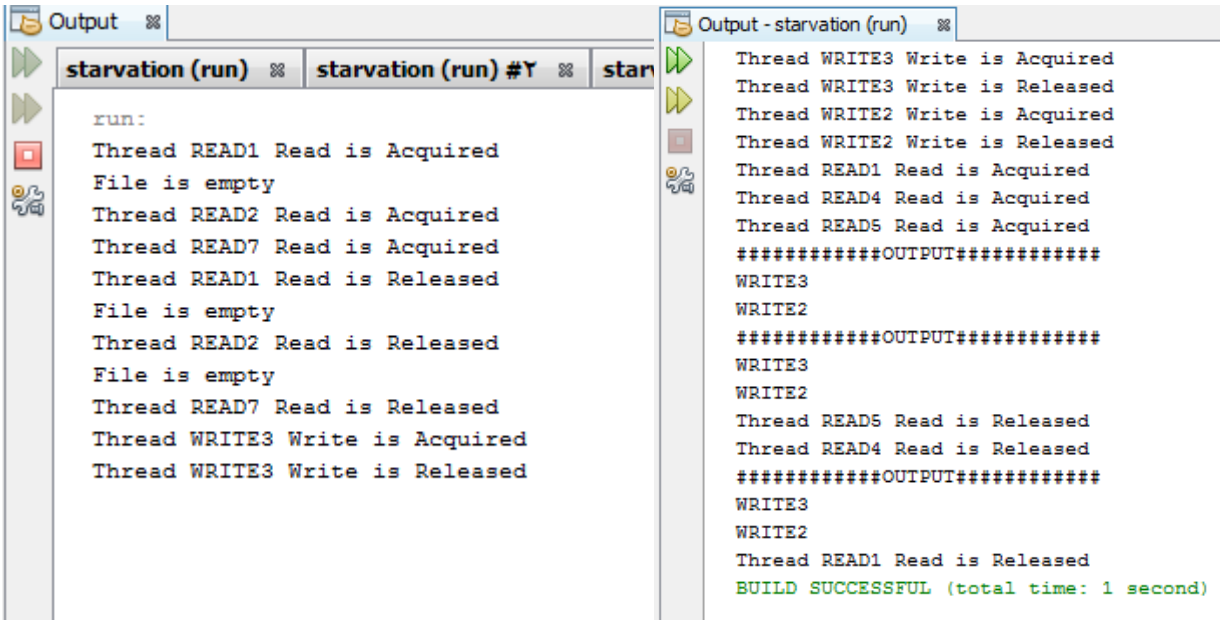
1) The structure of a writer's process

```
while (true) {
        rw_mutex.acquire();
                                ...
                /* writing is performed */
                                ...
        rw_mutex.release();
}
```

2) 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 mutex1 initialized to 1 (controls access to read_count)

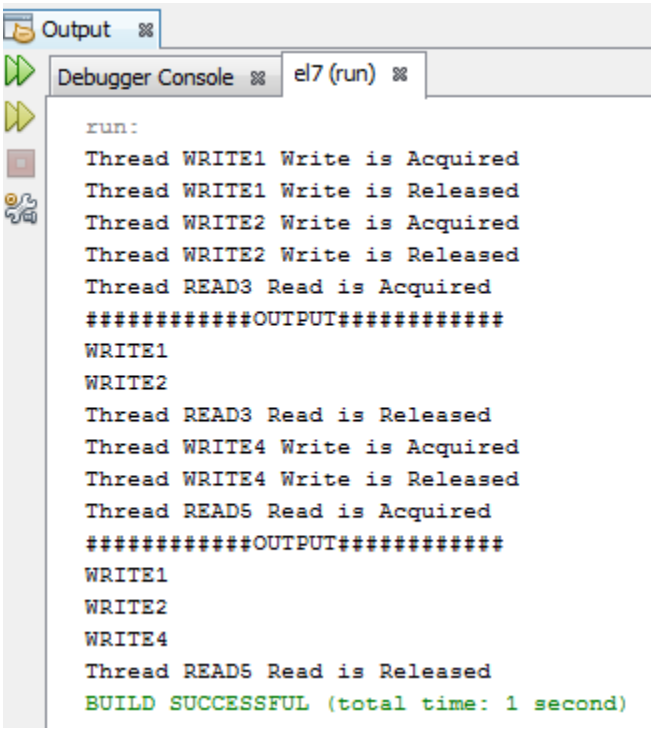
• Semaphore mutex2 initialized to 1 (controls access to write_count)

• Semaphore readLock initialized to 1 (reader access)

• Semaphore writeLock initialized to 1 (writer access)

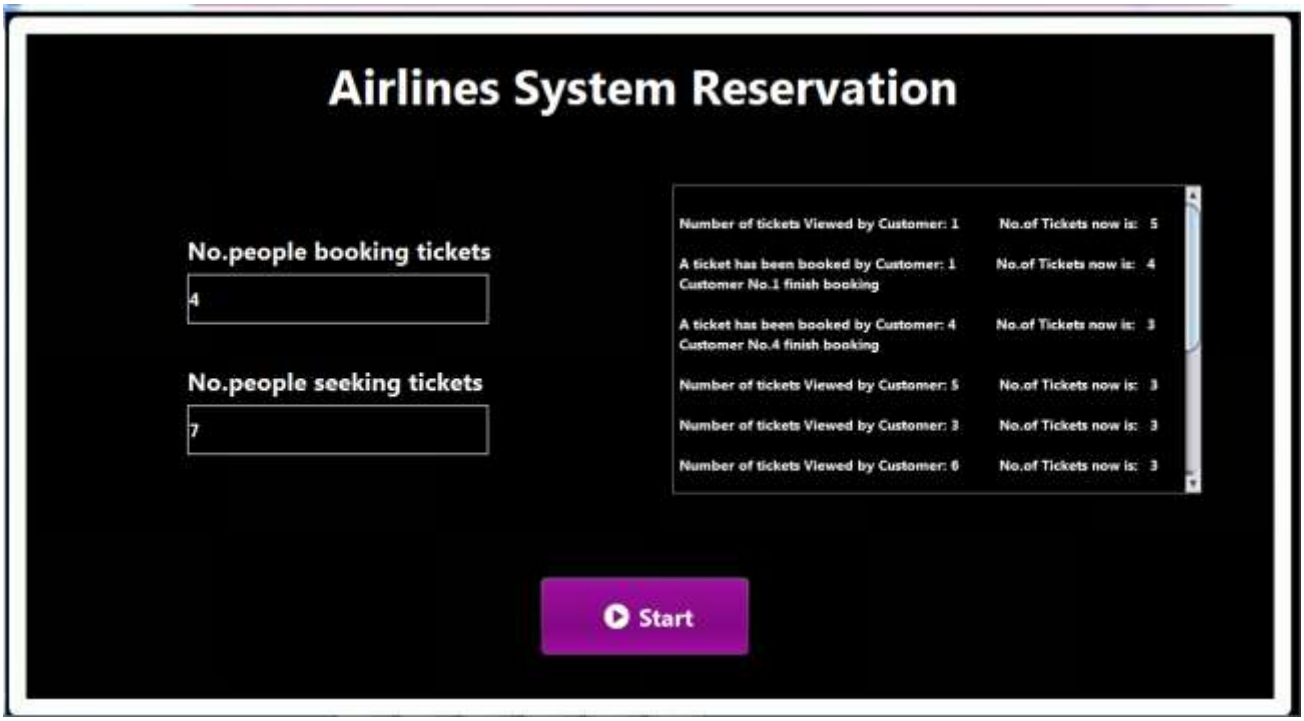
• Integer read_count initialized to 0 (how many processes are reading object)

• Integer write_count initialized to 0 (how many processes are writing object)

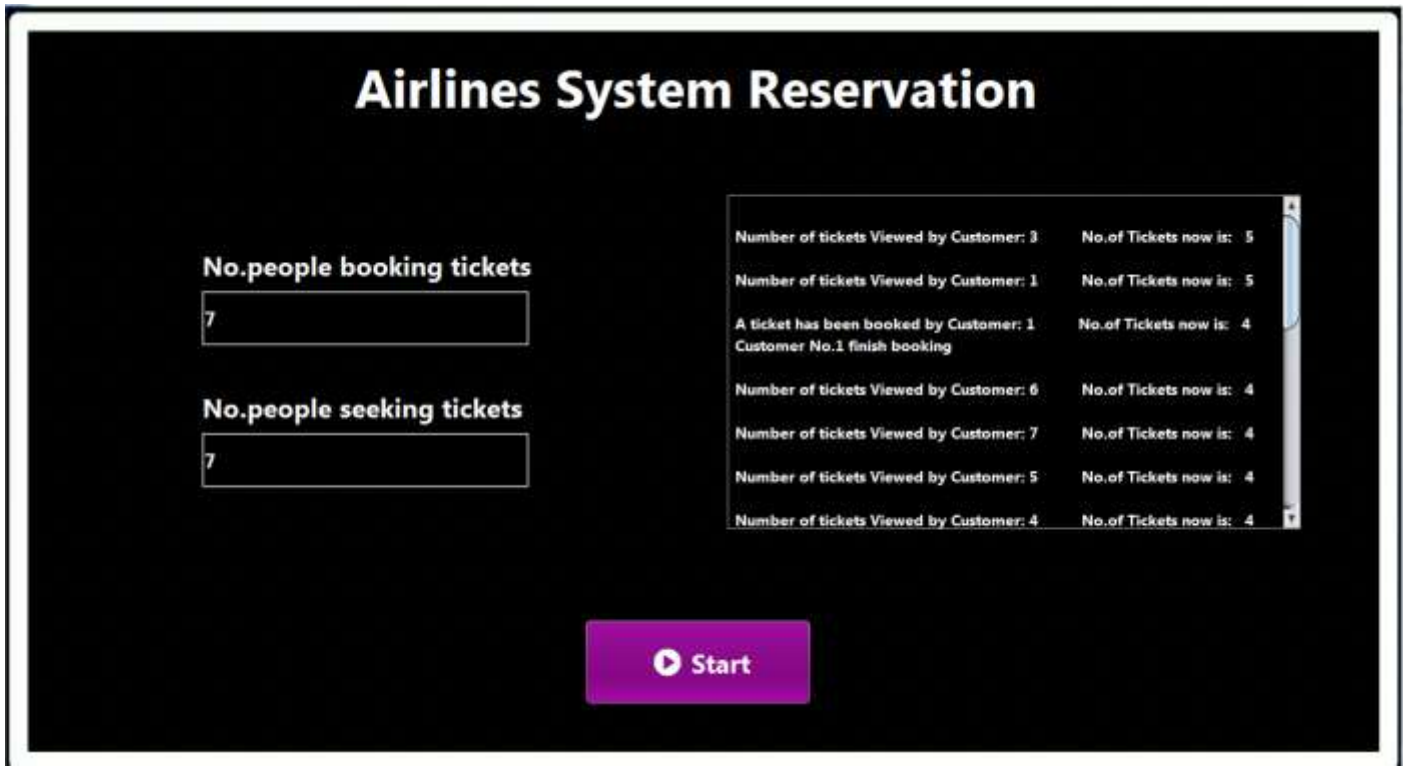


Explanation for real world application and how did apply the problem

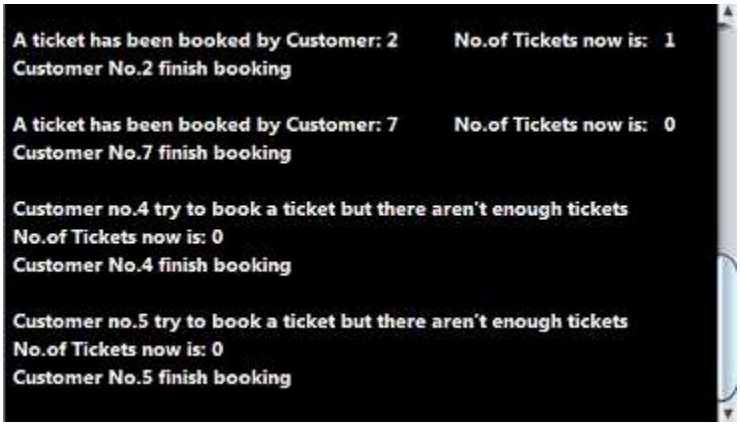
- We choose the Airline systems there are users who wants to book ticket (write) and who wants to seek (read)
- The main idea when someone try to book there is no others can see or book too (only one writer) and more than one can read in same time .
- and when someone try to (book or see) don't prevent 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 5 tickets and there's a number of people try to book (write) we assume that they = 7 and the people who is see (read) = 7




Number of tickets Viewed by Customer: 2 No.of Tickets now is: 4
A ticket has been booked by Customer: 3 No.of Tickets now is: 3
Customer No.3 finish booking
A ticket has been booked by Customer: 6 No.of Tickets now is: 2
Customer No.6 finish booking
A ticket has been booked by Customer: 2 No.of Tickets now is: 1
Customer No.2 finish booking
A ticket has been booked by Customer: 7 No.of Tickets now is: 0
Customer No.7 finish booking


A ticket has been booked by Customer: 2 No.of Tickets now is: 1
Customer No.2 finish booking
A ticket has been booked by Customer: 7 No.of Tickets now is: 0
Customer No.7 finish booking
Customer no.4 try to book a ticket but there aren't enough tickets
No.of Tickets now is: 0
Customer No.4 finish booking
Customer no.5 try to book a ticket but there aren't enough tickets
No.of Tickets now is: 0
Customer No.5 finish booking