

NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY (AN AUTONOMOUS INSTITUTION)

(AFFILIATED TO VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELGAUM, APPROVED
BY AICTE & GOVT.OF KARNATAKA)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM PROJECT REPORT ON

READER'S WRITER'S PROBLEM USING CLIENT SERVER ARCHITECTURE

GUIDE:

Mrs. Chaitra H.V

(Associative Professor,
Department of CSE)

Mrs. Sharmila Shanthi Sequeira

(Assistant Professor, Department

TEAM MEMBERS:

- | | |
|---------------------|------------|
| 1. PRATHIK KUMAR MP | 1NT15CS120 |
| 2. RAJATH B | 1NT15CS130 |
| 3. ROSHAN BADRINATH | 1NT15CS140 |
| 4. TEJAS M | 1NT15CS194 |

NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution Affiliated to VTU Belgaum)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



CERTIFICATE

This is to certify that the project report entitled

READER'S WRITER'S PROBLEM USING CLIENT SERVER ARCHITECTURE

is an authentic record of the project carried out by

TEAM MEMBERS

1. PRATHIK KUMAR MP	1NT15CS120
2. RAJATH B	1NT15CS130
3. ROSHAN BADRINATH	1NT15CS140
4. TEJAS M	1NT15CS194

PROJECT GUIDE:

.....

(Mrs. Chaitra H.V)

.....

(Mrs. Sharmila Shanthi Sequeira)

HEAD OF THE DEPARTMENT:

.....

(Dr. Thippeswamy MN)

ACKNOWLEDGMENT

The satisfaction that we have completed our project gives us immense pleasure and happiness. This project would have been incomplete without mentioning names of the people who have rightly guided.

We would like to convey our heartfelt thanks to **Dr. H.C. Nagaraj**, Principal and NMIT for giving us opportunity to embark on this project.

We are grateful to **Dr. Thippeswamy MN**, Head of the department, Computer Science for his invaluable guidance and support.

We would also like to thank our subject teacher and project guide **Prof. Chaitra HV** and **Prof. Sharmila Shanthi Sequeira** for their valuable guidance, assistance, support and constructive suggestion for betterment of the project.

We are also indebted to our friends for their continued moral and material support throughout the course of the project.

ABSTRACT

In computer science, the Readers Writers Problem is a classic inter-process communication and synchronization problem between multiple operating system processes. The problem is analogous to that of multiple readers reading the file and a single writer writing into the file.

The analogy is based on a single file in a system. The file can be written by a single writer. If a writer is writing into a file, no other writer can write, no other reader can read. If a reader is reading the contents of the file, no other writer can write into the file. Multiple readers can read the file as the contents of the file are not changed.

Analyzing the above analogy should ensure system functions correctly. It will prevent the occurrences of deadlock and starvation. In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

TABLE OF CONTENTS

Sl No.	Topics	Page No
1	Introduction	1
	a. Reader Writer Problem using client server architecture	3
2	Concepts used for correlating theoretical concept to practical implementation a. Semaphores b. Threads and FIFO Implementation	5
3	Design of the system and implementation a. Code b. Output	6
4	Conclusion	14
5	Bibliography	15

INTRODUCTION

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

The code for the writer process looks like this:

```
while(TRUE) {  
    wait(w);  
    /*perform the  
write operation */  
    signal(w);  
}
```

The code for the reader process looks like this:

```
while(TRUE) {  
    wait(m);    //acquire lock  
    read_count++;  
    if(read_count == 1)  
        wait(w);  
    signal(m); //release lock  
    /* perform the  
       reading operation */  
    wait(m);    // acquire lock  
    read_count--;  
    if(read_count == 0)  
        signal(w);  
    signal(m); // release lock  
}
```

Code Explained:

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Concepts used for Correlating theoretical concept to Practical implementation

SEMAPHORES:

In programming, especially in Unix systems, semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Semaphores are commonly use for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication (IPC). The C programming language provides a set of interfaces or "functions" for managing semaphores.

The value of the semaphore S is the number of units of the resource that are currently available. The P operation wastes time or sleeps until a resource protected by the semaphore becomes available, at which time the resource is immediately claimed. The V operation is the inverse: it makes a resource available again after the process has finished using it. One important property of semaphore S is that its value cannot be changed except by using the V and P operations.

A simple way to understand wait (P) and signal (V) operations is:

wait: If the value of semaphore variable is not negative, decrement it by 1. If the semaphore variable is now negative, the process executing wait is blocked (i.e., added to the semaphore's queue) until the value is greater or equal to 1. Otherwise, the process continues execution, having used a unit of the resource.

signal: Increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue.

The following is an implementation of this semaphore (where the value can be greater than 1):

```
wait(Semaphore s){  
  
    while (s==0); /* wait until s>0 */  
  
    s=s-1;  
  
}
```

```
signal(Semaphore s){  
  
    s=s+1;  
  
}
```

```
Init(Semaphore s , Int v){  
  
    s=v;  
  
}
```

Threads & FIFO Implementation

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Operating systems schedule threads either preemptively or cooperatively. On multi-user operating systems, preemptive multithreading is the more widely used approach for its finer grained control over execution time via context switching. However, preemptive scheduling may context switch threads at moments unanticipated by programmers therefore causing lock convoy, priority inversion, or other side-effects. In contrast, cooperative multithreading relies on threads to relinquish control of execution thus ensuring that threads run to completion. This can create problems if a cooperatively multitasked thread blocks by waiting on a resource or if it starves other threads by not yielding control of execution during intensive computation.

Another low-overhead paging algorithm is the **FIFO (First-In, First-Out)** algorithm. As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises. For this reason, FIFO in its pure form is rarely used.

CODE:**Server.java:**

```
import java.net.*;
import java.io.*;
import java.util.concurrent.Semaphore;

public class Server {
    public static void main(String[] args) throws Exception {
        try {
            ServerSocket server = new ServerSocket(8888);
            int counter = 0;
            System.out.println("Server has started..");
            while (true) {
                counter++;
                Socket client = server.accept();
                System.out.println(">> " + "Client No:" + counter + " started!");
                ReaderWriter rw = new ReaderWriter(client, counter);
                rw.start();
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

class ReaderWriter extends Thread {
    public static Semaphore wrt = new Semaphore(1);
    public static Semaphore mutex = new Semaphore(1);
    private String clM = "", srM = "", tc, line;
    Socket client;
    int clientNo;
    public static int readcount=0;
```

```
ReaderWriter(Socket inSocket, int counter) {
    client = inSocket;
    clientNo = counter;
}

public void run() {
    try {
        DataInputStream inStream = new DataInputStream(client.getInputStream());
        DataOutputStream outStream = new DataOutputStream(client.getOutputStream());
        while (!clM.equals("exit")) {
            clM = inStream.readUTF();
            System.out.println("Client " + clientNo);
            if(clM.equals("reader")) {
                tc = "reader";
                mutex.acquire();
                readcount++;
                System.out.println(readcount);
                if(readcount==1) wrt.acquire();
                mutex.release();
                outStream.writeUTF("Reader Ready");
                outStream.flush();
                outStream.writeUTF("Contents of file :");
                outStream.flush();
                srM="";
                System.out.println("Reader..!!");
                FileReader fr = new FileReader("file.txt");
                BufferedReader bfr = new BufferedReader(fr);
                while((line = bfr.readLine()) != null) {
                    srM += line;
                    srM += "\n";
                }
                bfr.close();
                outStream.writeUTF(srM);
                outStream.flush();
            } else if(clM.equals("writer")) {
```

```
        tc = "writer";
        wrt.acquire();
        srM = "Writer Ready";
        outStream.writeUTF(srM);
        outStream.flush();
        srM = inStream.readUTF();
        System.out.println(srM);
        FileWriter fw = new FileWriter("file.txt");
        BufferedWriter bfw = new BufferedWriter(fw);
        bfw.write(srM);
        bfw.close();
    } else {
        System.out.println(clM);
    }
}

if(tc.equals("reader")) {
    System.out.println(tc);
    mutex.acquire();
    readcount--;
    if(readcount==0) wrt.release();
    mutex.release();
} else if(tc.equals("writer")) {
    System.out.println(tc);
    wrt.release();
} else {
    System.out.println(tc);
}

inStream.close();
outStream.close();
client.close();
} catch (Exception ex) {
    System.out.println(ex);
} finally {
    System.out.println("Client -" + clientNo + " exit!! ");
}
```

```
}  
}  
}
```

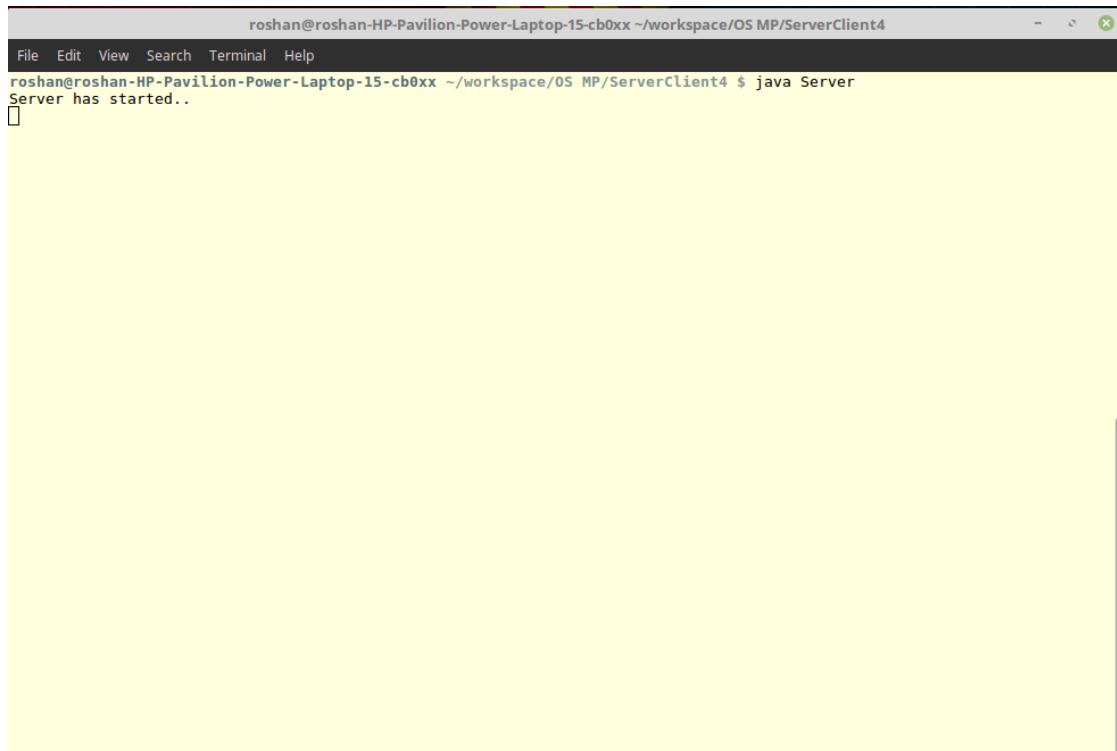
Client.java

```
import java.net.*;  
import java.io.*;  
  
public class Client {  
    public static void main(String[] args) throws Exception {  
        try {  
            Socket socket = new Socket("192.168.43.56", 8888);  
            DataInputStream inStream = new DataInputStream(socket.getInputStream());  
            DataOutputStream outStream = new DataOutputStream(socket.getOutputStream());  
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
            String clM = "", srM = "";  
            StringBuilder txt = new StringBuilder("");  
            System.out.println("Reader/writer :");  
            while (!clM.equals("exit")) {  
                clM = br.readLine();  
                outStream.writeUTF(clM);  
                outStream.flush();  
                if (clM.equals("reader")) {  
                    System.out.println("Busy..!");  
                    srM = inStream.readUTF();  
                    System.out.println(srM);  
                    srM = inStream.readUTF();  
                    System.out.println(srM);  
                    srM = inStream.readUTF();  
                    System.out.println(srM);  
                } else if (clM.equals("writer")) {  
                    System.out.println("Busy..!");  
                    srM = inStream.readUTF();
```

```
System.out.println(srM);
System.out.println("Enter Text ( enter * in new line to send )");
clM = br.readLine();
while(!(clM.equals("*"))) {
    txt.append(clM);
    txt.append("\n");
    clM = br.readLine();
}
clM = txt.toString();
outStream.writeUTF(clM);
outStream.flush();
}else {
    System.out.println("Bye");
}
}
outStream.close();
outStream.close();
socket.close();
} catch (Exception e) {
    System.out.println(e);
}
}
}
```

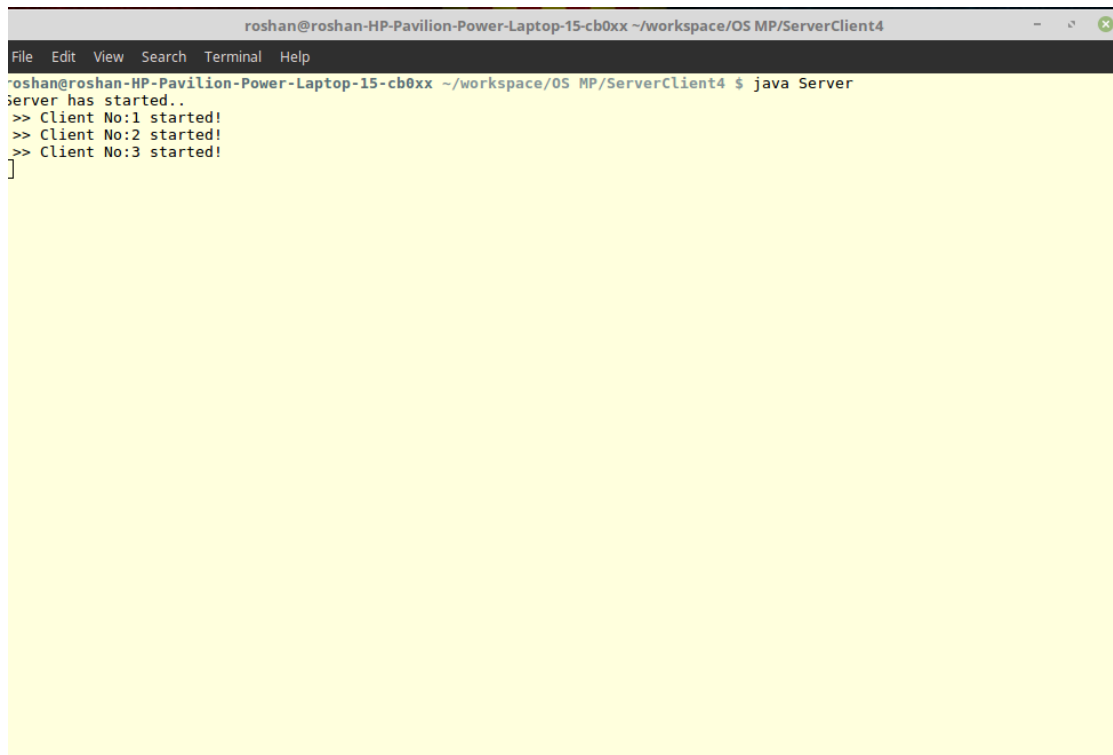
OUTPUT:

1. Starting the Server

A terminal window titled 'roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4'. The terminal shows the command 'java Server' being executed, followed by the output 'Server has started..' and a cursor on the next line.

```
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4
File Edit View Search Terminal Help
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $ java Server
Server has started..
█
```

2. Multiple Clients

A terminal window titled 'roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4'. The terminal shows the command 'java Server' being executed, followed by the output 'server has started..' and three lines of client status: '>> Client No:1 started!', '>> Client No:2 started!', and '>> Client No:3 started!'. The prompt returns to the shell.

```
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4
File Edit View Search Terminal Help
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $ java Server
server has started..
>> Client No:1 started!
>> Client No:2 started!
>> Client No:3 started!
█
```


3. Client as Writer



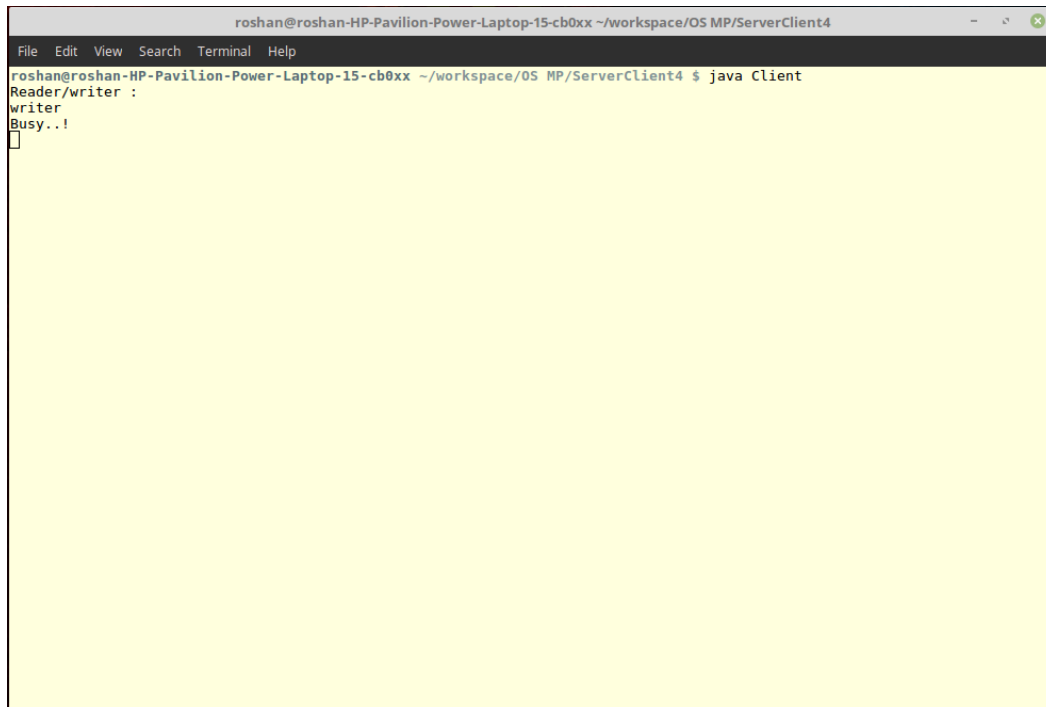
```
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4
File Edit View Search Terminal Help
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $ java Client
Reader/writer :
writer
Busy..!
Writer Ready
Enter Text ( enter * in new line to send )
These are contents of
the file
*
exit
Bye
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $
```

4. Client as Reader



```
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4
File Edit View Search Terminal Help
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $ java Client
Reader/writer :
reader
Busy..!
Reader Ready
Contents of file :
These are contents of
the file
exit
Bye
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $
```

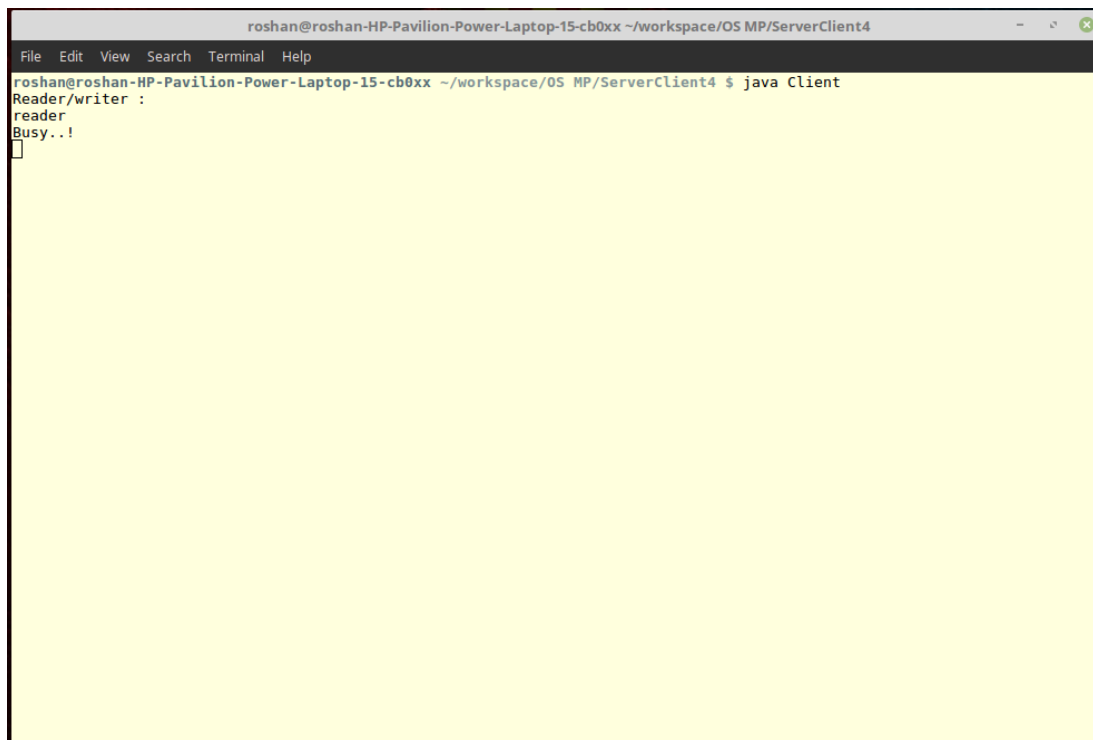
5. Writer waiting



```
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4
File Edit View Search Terminal Help
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $ java Client
Reader/writer :
writer
Busy..!
□
```

A terminal window titled "roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4" with a menu bar (File, Edit, View, Search, Terminal, Help). The command "java Client" has been executed. The output shows "Reader/writer :", followed by "writer" on the next line, then "Busy..!" on the next line, and a small square symbol "□" on the final line.

6. Reader waiting



```
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4
File Edit View Search Terminal Help
roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4 $ java Client
Reader/writer :
reader
Busy..!
□
```

A terminal window titled "roshan@roshan-HP-Pavilion-Power-Laptop-15-cb0xx ~/workspace/OS MP/ServerClient4" with a menu bar (File, Edit, View, Search, Terminal, Help). The command "java Client" has been executed. The output shows "Reader/writer :", followed by "reader" on the next line, then "Busy..!" on the next line, and a small square symbol "□" on the final line.

CONCLUSION:

The Readers Writers Problem enables pre-planned service to all the clients who use the file system. The server system also counts the number of clients that are connected at a given instant of time. The client server architecture provides a file sharing system which removes the need to create resources from scratch. The synchronization provided by semaphores prevents deadlock state in the system. The code which we provided obtains the solution to the above specified problem.

BIBLIOGRAPHY

1. Operating System Concepts 7th Edition by
Abraham Silberchatz,
Peter Baer Galvin,
Greg Gagne
2. www.tutorialspoint.com