Contents

SMT CHANDIBAI HIMATHMAL MASUKHANI COLLEGE
USCSP301_USCS303 :Operating System(OS) Practical-04

Practical-04: PROCESS COMMUNICATION

(a)Practical Date: 07 August 2021.

(b)Practical Aim: Producer- Consumer Problem, RMI

(c)Process Communication:
Processes often need to communicate with each other.

This is complicated in distributed systems by the fact that the communicating processes may be on different workstations.

Inter-process communication provides a means for processes to cooperate and compete.

Message passing and **remote procedure calls** are the most common methods of inter-process communicationin distributed systems.

(d)Producer-Consumer Problem:
In a **producer/consumer relationship**, the **producer** portion of an application generates data and stores it in a shared object, and the **consumer** portion of an application reads data from the shared object.

One example of a common producer/consumer relationship is print spooling. A word processor pool data to a buffer (typically a file) and the data is subsequently consumed by the printer as it prints the document. Similarly, an application that copies data onto compact discs paces data ina fixed-size buffer that is emptied as the CD-RW drive burns the data onto the compact disc.

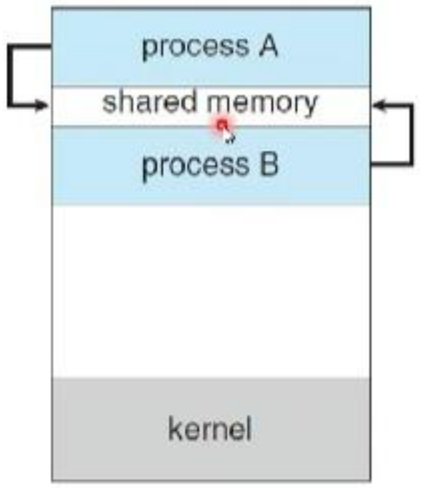
(e)Using Shared Memory

1. Producer - Consumer Solution using Shared Memory

(i) Shared memory is memory that may be simultaneously accessed by multiple processes with an intent to provide communication among them or avoid reduntant copies.

(ii) Shared memory is an efficient means of passing data between processes.

SMT CHANDIBAI HIMATHMAL MASUKHANI COLLEGE



QUESTION 1:

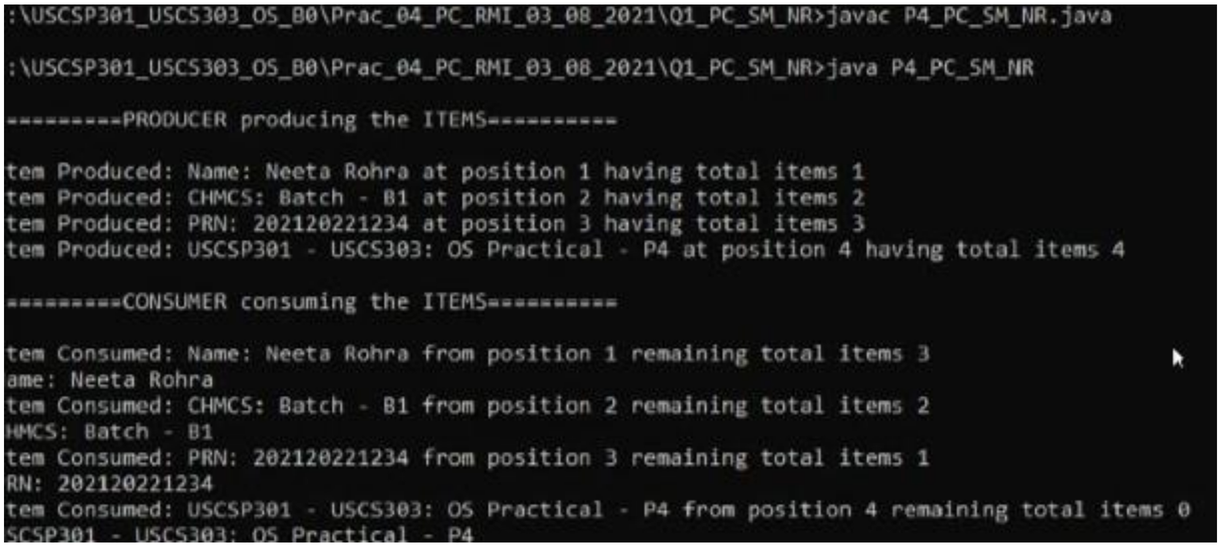
Write a java program for producer-consumer problem using shared memory.

```
//Name: Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


public clas P4_PC_SM_BufferImpl_RS implements P4_PC_SM_Buffer_RS

{

    private static final int BUFFER_SIZE = 5;
    private String[] elemnts;

    private int in, out, count;

public P4_PC_SM_BufferImpl_RS()        //  constructor initializing the variables to initial value

{

    count=0;

    in=0;

    out=0;

    elements=new String[BUFFER_SIZE];

} //  constructor ends

// Producer call this method

public void insert(String item)

{

        while(count==BUFFER_SIZE)

                ;//do nothing as there is no free space

        // add an item to the buffer

        elements[in]=item;

        in=(in+1)% BUFFER_SIZE;

        ++COUNT;

        System.out.printIn("Item Produced:"+item+"at position"+in+"having total items"+count);

}//insert() ends

// Consumers call this method

public String remove()

{

         String item;

         while(count==0)
```

```
                ;// do nothing as there is nothing to consume

          // remove an item from the buffer

          item= elements[out];

          out=(out+1)% BUFFER_SIZE;

          --count;

          System.out.printIn("Item Consumed:"+item+"from position"+out+"remaining total
items"+count);

          return item;

          }// remove() ends

}// class ends


//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


public class P4_PC_SM_RS

{

        public static void main(Sting args[]){

                P4_PC_SM_BufferImpl_RS            bufobj          =              new
P4_PC_SM_Bufferimpl_RS();

              System.out.printIn("\n===========PRODUCER producing the
ITEMS===========\n");

              bufobj.insert("Name:Ritika Sahu");

              bufobj.insert("CHMCS:Batch-B1");
```

```
                bufobj.insert("PRN:2020016400783543");

                bufobj.insert("USCSP301_USCS303:OS Practical-P4");

        System.out.printIn("\n=========CONSUMER consuming the ITEMS===========\n");

String element=bufobj.remove();

System.out.printIn(element);

System.out.printIn(bufobj.remove());

System.out.printIn(bufobj.remove());

System.out.printIn(bufobj.remove());

}// main ends

}// class ends
```

OUTPUT:



```
:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q1_PC_SM_NR>javac P4_PC_SM_NR.java

:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q1_PC_SM_NR>java P4_PC_SM_NR

=========PRODUCER producing the ITEMS==========

tem Produced: Name: Neeta Rohra at position 1 having total items 1
tem Produced: CHMCS: Batch - B1 at position 2 having total items 2
tem Produced: PRN: 202120221234 at position 3 having total items 3
tem Produced: USCSP301 - USCS303: OS Practical - P4 at position 4 having total items 4

=========CONSUMER consuming the ITEMS==========

tem Consumed: Name: Neeta Rohra from position 1 remaining total items 3
ame: Neeta Rohra
tem Consumed: CHMCS: Batch - B1 from position 2 remaining total items 2
HMCS: Batch - B1
tem Consumed: PRN: 202120221234 from position 3 remaining total items 1
RN: 202120221234
tem Consumed: USCSP301 - USCS303: OS Practical - P4 from position 4 remaining total items 0
SCSP301 - USCS303: OS Practical - P4
```

(f) Using Message Passing

2. Producer - Consumer solution using Message Passing

(i) Message passing is the basis of most inter-process communication in distributed systems.

(ii) It is at the lowest level of abstraction and requires the application programmer to be able to identify the destination process, the message, the source process and the data types expected from these processes.

(iii) Communication in the message passing paradigm, in the simplest form, is performed using the send() and receive() primitives. The syntax is generally of the form:

send(receiver, message)

receive(sender, message)

(iv) The send() primitive requires the name of the destination process and the message data as parameters. The addition of the name of the senderas a parameter of the send() primitive would enable the receiver to acknowledge the message. The receive() primitive requires the name of the anticipated sender and should provide a storage buffer for the message.

QUESTION 2:

Write a java program for producer - consumer problem using message passing.

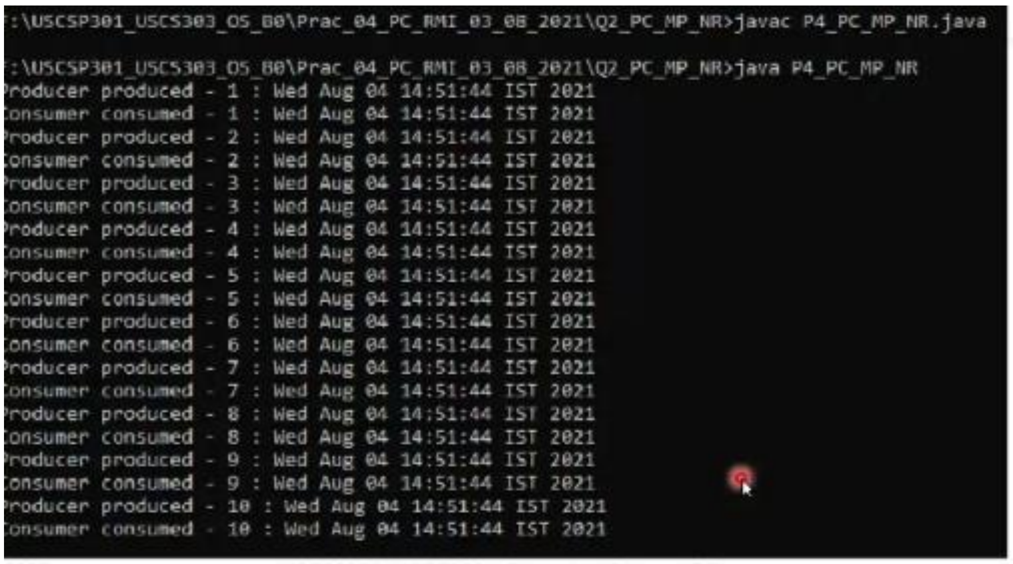
```
//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


public interface P4_PC_MP_Channel_RS<E>

{

        // Send a message to the channel

        public void send(E item);
```

```
            //Receive a message from the channel
           public E receive();
}// interface ends

//Name:Ritika Sahu
//Batch : B1
//PRN:2020016400783543
//Date: 07 August, 2021.
//Practical 4: Process communication

import java.util.Vector;
public class P4_PC_MP_MessageQueue_RS_<E> implements P4_PC_MP_Channel_RS<E>
{
     private Vector<E> queue;

     public P4_PC_MP_MessageQueue_RS(){
            queue = new Vector<E>();
}

//This implements a nonblocking send
public void send(E item){
        queue.addElement(item);
}// send() ends
```

```
//This implements a nonblocking receive

public E receive(){

       if(queue.size()==0)

             return null;

       else

            return.queue.remove(0);

     }//receive() ends

}// class ends


//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


import java.util.Date;

public class P4_PC_MP_RS

{

           public static void main(String args[])

           {

                   //Producer and Consumer process

                   P4_PC_MP_Channel_rs<Date>  mailBox = new
P4_PC_MP_Messagequeue_RS<Date>(i);

                   int i=0;

                   do
```

```
        {
            Date message = new Date();
            System.out.printIn("Producer produced-"(i+1)+":"+message);
            mailBox.send(message);
            Date rightNow = mailBox.receive();
            if(rightNow !=null)
            {
                System.out.printIn("Consumer consumed-"+(i+1)+":"+rightNow);
            }
             i++;
         }while(i<10);
     }// main ends
}//class ends
```

OUTPUT:



```
:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q2_PC_MP_NR>javac P4_PC_MP_NR.java

:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q2_PC_MP_NR>java P4_PC_MP_NR
roducer produced - 1 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 1 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 2 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 2 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 3 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 3 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 4 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 4 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 5 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 5 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 6 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 6 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 7 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 7 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 8 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 8 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 9 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 9 : Wed Aug 04 14:51:44 IST 2021
roducer produced - 10 : Wed Aug 04 14:51:44 IST 2021
onsumer consumed - 10 : Wed Aug 04 14:51:44 IST 2021
```

(g) Remote Procedure Calls

(i) Message passing leaves the programmer with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) relieves this burden by increasing the level of abstraction and providing semantics similar to a local procedure call.

(ii) The syntax of a remote procedure call is generally of the form:

call procedure_name(value_arguments;result_arguments)

(iii) The client process blocks at the **call()** until the reply is received.

(iv) The remote procedure is the server processes which has already begun executing on a remote machine.

(v) It blocks at the **receive()** until it receives a message and parameters from the sender.

(vi) The syntax is as follows:

receive procedure_name(in value_parameters;out result_parameters)

reply(caller, result_parameters)

(vii) In the simplest case, the execution of the call() generates a client stub which marshals the arguments into a message and sends the message to the servr machine. On the server machine the server is blocked awaiting te message. On receipt of the message the server tub is generated and extracts the parameters from the message and passes the parameters and control to the procedure. The results are returned to the client with the same procedure in reverse.

3. Remote Method Invocation (RMI) Calculator

Step 1: Creating the Remote interface

This file defines the remote interface that is provided by the server.It contains four methods that accepts two **integer** arguments and returns their sum, difference, product and quotient. All remote interfaces must extend the **Remote** interface, which is part of java.rmi. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException.**

//Name:Ritika Sahu

SMT CHANDIBAI HIMATHMAL MASUKHANI COLLEGE

```
//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


import java.rmi.*;

public interface P4_RMI_CalcServerInt_RS extends Remote

{

   int add(int a, int b) throws RemoteException;

   int subtract(int a, int b) throws RemoteException;

   int  multiply(int a, int b) throws RemoteException;

   int divide(int a, int b) throws RemoteException;

}// interface ends
```

Step 2: Implementation the Remote Interface

This file implements the remote interface. The implementation of all the four methods is straight forward. All remote methods must extend **UnicastRemoteObject** , which provides functionality that is needed to make objects available from remote machines.

```
 //Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


import java.rmi.*;
import java.rmi.server.*;

public calss P4_RMI_CalcServerImpl_RS extends UnicastremoteObject implements
P4_RMI_CalcServerintf_RS

{

public P4_RMI_CalcServerImpl_RS() throws RemoteException{

}

public int add(int a, int b) throws RemoteException

{

   return a+b;

}

public int subtract(int a, int b) throws RemoteException

{

   return a-b;

}

public int multiply(int a, int b) throws RemoteException

{

   return a*b;

}

public int divide(int a, int b) throws RemoteException

{

   return a/b;

}

}// class ends
```

Step 3: Creating the Server

SMT CHANDIBAI HIMATHMAL MASUKHANI COLLEGE

This file contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the **rebind()** method of the **Naming** class (found in **java.rmi**). that method associates a name with an object reference. This first argument to the **rebind()** method is a string that names a server .Its second argument is a reference to an instance of **CalcServerImpl.**

```
 //Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


import java.net.*;

import.java.rmi.*;

public class P4_RMI_CalcServer_RS

{

  public static void main(String args[])

      {

               try

                 {

      P4_RMI_CalcServerImpl_RS csi=new P4_RMI_CalcServerImpl_RS();

       Naming.rebind("CSB1",csi);

            }// try ends

             catch(Exception e) {

System.out.printIn("Exception:"+e);

}// catch ends

}// main ends
}// class ends
```

Step 4: Creating the Client

This file implements the client side of this distributed application. It accepts three command-line arguments.The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be operated.

The application begins by forming a string that follows the URL syntax. This URL uses the rmi protocol. The string includes the IP address or name of the server and the string"CSB1". The program then invokes the **lookup()** method of the **Naming** class. This method accepts one argument, the rmi URL, and returns a reference to an object of type **CalcServerInf** . All remote method invocations can be directed to this object.

```
 //Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 07 August, 2021.

//Practical 4: Process communication


import java.rmi.*;

public class P4_RMI_CalcClient_RS

{

     public static void main(String args[])

     {

      try{

               String CSURL="rmi://"+args[0]+"/CSB1";

               P4_RMI_CalcServerIntf_CSIntf = (P4_RMI_CalcServerIntf_RS)
Naming.lookup(CSURL);

               System.out.printIn("The first number is:"+args[1]);
```

```
		int x=Integer.parseInt(args[1]);

		System.out.printIn("The second number is:"+args[2]);

		int y=Integer.parseInt(args[2]);

		System.out.printIn("=========Arithmetic Operations========");

		System.out.printIn("Addition:"+x+"+"+y+"="+ CSIntf.add(x,y));

		System.out.printIn("Subtraction:"+x+"-"+y+"="+CSIntf.subtract(x,y));

		System.out.printIn("Multiplication:"+x+"*"+y+"="+CSIntf.multiply(x,y));

		System.out.printIn("Division:"+x+"/"+y+"="+CSIntf.dividet(x,y));

}// try ends

catch(Exception e){

		System.out.printIn("Exception:"+e);

}//catch ends

}// main ends

}// class ends
```

Step 5: Manually generate a stub, if required

Prior to Java 5,stubs needed to be built manually by using rmic.This step is not required for modern versions of Java. However, if we work in a legacy environment, then we can use the rmic compiler, as shown here, to build a stub.

rmic CalcServerImpl

Step 6: Install Files on the Client and Server Machines

Copy **P4_RMI_CalcClient_RS.class, P4_RMI_CalcServerImpl_RS_Stub.class (if needed),** and **P4_RMI_CalcServerIntf_RS.class** to a directory on the client machine.

Copy **CalcServerIntf,class, P4_RMI_CalcServerImpl_RS.class, P4_RMI_CalcServerImpl_RS_Stub.clas**s (if needed), and **P4_RMI_CalcServer_RS.class** to a directory on the server machine.

Step 7: Start the RMI Registry on the Server Machine.

The JDK provides a program called rmiregistry, which executes on the server machine. It maps names to object references. Start the RMI Registry form the command line, as shown here: **start rmiregistry.**

When this command returns, a new window gets created. Leave this window open until we are done experimenting with the RMI example.

Step 8: Start the Server

The server code is started from the command line, as shown here:

```
Java P4_RMI_CalcServer_RS
```

Step 9:Start the Client

The client code is started from the command line, as shown here:

```
Java P4_RMI_CalcClient_RS   127.0.0.1  15  5
```

FOR EXECUTION:

Open 2 Command Terminals:

Terminal-01:

Compile all the .java files

Command:

```
Javac *.java
```

Start the RMI registry:

Command:

```
Start rmiregistry
```

Terminal-01:

Start the Server:

Command:

Java P4_RMI_CalcServer_RS

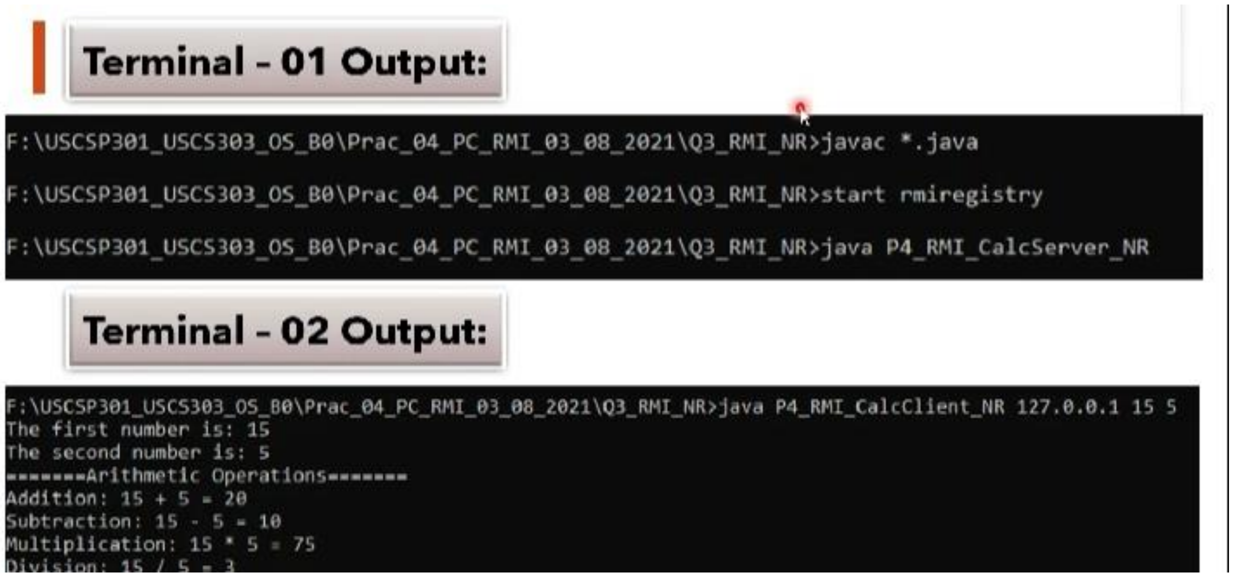
Terminal -02:

Run the Client:

Command:

Java P4_RMI_CalcClient_RS 127.0.0.1 15 5

OUTPUT:



```
F:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q3_RMI_NR>javac *.java

F:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q3_RMI_NR>start rmiregistry

F:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q3_RMI_NR>java P4_RMI_CalcServer_NR
```

```
F:\USCSP301_USCS303_OS_B0\Prac_04_PC_RMI_03_08_2021\Q3_RMI_NR>java P4_RMI_CalcClient_NR 127.0.0.1 15 5
The first number is: 15
The second number is: 5
=======Arithmetic Operations=======
Addition: 15 + 5 = 20
Subtraction: 15 - 5 = 10
Multiplication: 15 * 5 = 75
Division: 15 / 5 = 3
```