CONTENTS

USCS3P01:USCS303 – OPERATING SYSTEM (OS) PRACTICAL_04

SYSTEM (OS) PRACTICAL – 04

PRACTICAL-04: PROCESS COMMUNICATION..

PRACTICAL DATE: 7TH AUGUST 2021

PRACTICAL AIM: PRODUCER-CONSUMER PROBLEM,RPI

Content:

Solution for Producer - Consumer Problem using shared memory and message passing.

Communication in Client-Server environment using Remote Method Invocation (RMI).

Process:

Producer - Consumer problem without threads and without synchronization.

Implement Remote method invocation (RMI).

Prior Knowledge:

Concept of shared memory, message passing, interfaces and remote method invocation.

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 communication in distributed systems.
- A less frequently used but no less valuable method is distributed shared memory.

PRODUCER-CONSUMER PROBLEM

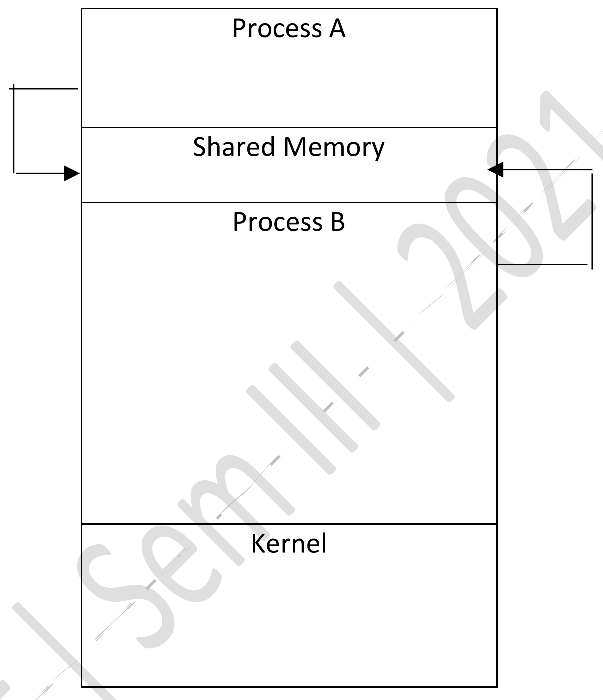
- In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of an application reads data from the shared object .

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

a) USING SHARED MEMORY

- Shared memory is memory that may be simultaneously accessed by multiple processes with an intent to provide communication among them or avoid redundant copies.

- Shared memory is an efficient means of passing data between processes.



1. Producer-Consumer Solution using Shared Memory

A) QUESTION: - Write a Java program for producer consumer problem using shared memory.

B) SOURCE CODE

File Name: P4_PC_SM_Buffer_YP.java

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

public interface P4_PC_SM_Buffer_YP
{
        //Producers call this method
        public void insert(String item);



        //Consumers call this method
        public String remove();
}//interface ends
```

File Name: P4_PC_SM_BufferImpl_YP.java

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

public class P4_PC_SM_BufferImpl_YP    implements
P4_PC_SM_Buffer_YP
{
        private static final int BUFFER_SIZE = 5;
        private String[] elements;
        private int in, out, count;

        public P4_PC_SM_BufferImpl_YP()  //constructor initializing the variable to initial
value
        {
                count = 0;
                in = 0;
                out = 0;
                elements = new String[BUFFER_SIZE];
        }//constructor ends
        // Producers 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.println("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.println("Item Consumed: " + item + " from position " + out +
"remaining total items" + count);
                return item;
        }//remove() ends
}//class ends
```

FILE NAME: P4_PC_SM_YP.JAVA

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

public class P4_PC_SM_YP
{
        public static void main(String[] args) {
                P4_PC_SM_BufferImpl_YP    bufobj    =    new
P4_PC_SM_BufferImpl_YP();

                System.out.println("\n==========PRODUCER    producing
ITEMS==========\n");
                bufobj.insert("Name: Yash Patil");
                bufobj.insert("CHMS: Batch-B2");
                bufobj.insert("PRN: 2020016400809191");
                bufobj.insert("USCSP301 - USCS303: OS Practical -P4");
                System.out.println("\n==========CONSUMER    consuming    the
ITEMS==========\n");
                String element = bufobj.remove();
                System.out.println(element);
                System.out.println(bufobj.remove());
                System.out.println(bufobj.remove());
                System.out.println(bufobj.remove());
        }//main ends
}//class ends
```

C) OUTPUT: -

USING MESSAGE PASSING

- Message passing is the basis of most inter-process communication in distributed systems.
- 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.
- Communication in the message passing paradigm, in its simplest form, is performed using the send() and receive() primitives. The syntax is generally of the form:

Send (receiver, message)

Receive (sender, message)

- The send() primitive requires the name of the destination process and the message data as parameters. The addition of the name of the sender as a parameter for 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.

2. Producer-Consumer Solution using Message Passing

A) QUESTION: - Write a java program for producer consumer problem using message passing.

B) SOURCE CODE

File Name: P4_PC_MP_Channel_YP.java

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

public interface P4_PC_MP_Channel_YP<E>
{
        // Send a message to the channel
        public void send(E item);
        // Receive a message from the channel
         public E receive();
}//interface ends
```

FILE NAME: P4_PC_MP_MESSAGEQUEUE_YP.JAVA

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

import java.util.Vector;
public class P4_PC_MP_MessageQueue_YP<E> implements P4_PC_MP_Channel_YP<E>
{
        private Vector<E> queue;
        public P4_PC_MP_MessageQueue_YP(){
        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
```

FILE NAME: P4_PC_MP_YP.JAVA

```
//Name: Yash Patil

// Batch: B2

// PRN: 2020016400809191

// Date: 06-08-2021

// Prac-04: Process Communication


import java.util.Date;

public class P4_PC_MP_YP

{

        public static void main(String args[])

        {

        // Producer and Consumer process

        P4_PC_MP_Channel_YP<Date> mailBox = new P4_PC_MP_MessageQueue_YP

<Date>();

        int i=0;

do

{

        Date message = new Date();

        System.out.println("Producer produced - " + (i+1) + ":" + message);

        mailBox.send(message);

        Date rightNow = mailBox.receive();

        if(rightNow != null)

        {

        System.out.println("Consumer consumed " + (i+1)+": " +rightNow);

        }

        i++;

        }while (i < 10);

        } // main ends

}// class ends
```

C) OUTPUT: -

REMOTE PROCDURE CALLS.

- Message passing leaves the programmers with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) relives this burden by increasing the level of abstraction and providing semantics similar to local procedure call.
- Syntax:
- The syntax of a remote procedure call is generally of the from:

  Call procedure_name(value_arguments;result_arguments)

- The client process block at the call() until the reply is received
- The remote procedure is the server processes which has already begun executing on a remote machine.
- It blocks at the receive() until it receives a message and parameters from the sender.
- The server then sends a reply() when it has finished its task.
- The syntax is as follows:

  receive procedure_name(in value_parameters; out result_parameters)

  reply(caller, result_parameters)

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

3. REMOTE METHOD INVOCATION (RMI) CALCULATOR.

A) QUESTION: - Write a java RMI program for adding, subtracting, multiply and dividing two numbers.

B) Steps:-

- ✓ **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.**
- ✓ **Step 2: Implementing 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.
- ✓ **Step 3: Creating the server**
  - 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. The first argument to the **rebind()** method is a string that names the server. Its second argument is a reference to an instance of **CalcServerImpl**.
- ✓ **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 then be directed to this object.
- ✓ **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_YP.class, P4_RMI_CalcServerImpl_YP_Stub.class** (if needed), and **P4_RMI_CalcServerIntf_YP.class** to a directory on the client machine.
  - Copy **CalcServerIntf.class, P4_RMI_CalcServerImpl_YP.class, P4_RMI_CalcServerImpl_YP_Stub.class** (if needed), and **P4_RMI_CalcServer_YP.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_YP
- ✓ **Step 9: Start the client**
  - The client code is started from the command line, as shown here:

    java P4_RMI_CalcClient_YP 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-02:**
      - ▪ **Start the Server:**
      - ▪ **Command:**

        java P4_RMI_CalcServer_YP
      - ▪ **Run the Client:**
      - ▪ **Command:**

        java P4_RMI_CalcClient_YP 127.0.0.1 15 5

C) SOURCE CODE

File Name: P4_RMI_CalcServerIntf_YP.java

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication


import java.rmi.*;
public interface P4_RMI_CalcServerIntf_YP 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
```

FILE NAME: P4_RMI_CALCSERVERIMPL_YP.JAVA

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication


import java.rmi.*;
import java.rmi.server.*;
public class P4_RMI_CalcServerImpl_YP extends UnicastRemoteObject implements
P4_RMI_CalcServerIntf_YP
{
        public P4_RMI_CalcServerImpl_YP() 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

//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

import java.net.*;
import java.rmi.*;
public class P4_RMI_CalcServer_YP
{
 public static void main(String args[])
{
            try
            {
            P4_RMI_CalcServerImpl_YP csi = new P4_RMI_CalcServerImpl_YP();
            Naming.rebind("CSB1",csi);
            } // try ends
            catch(Exception e)
        {
                        System.out.println("Exception: " + e);
            } // catch ends
    } // main ends
} // class ends
```

FILE NAME: P4_RMI_CALCSERVER_YP.JAVA

```
//Name: Yash Patil
// Batch: B2
// PRN: 2020016400809191
// Date: 06-08-2021
// Prac-04: Process Communication

import java.rmi.*;
public class P4_RMI_CalcClient_YP
{
                    public static void main(String args[])
                    {
                    try{
                    String CSURL="rmi://"+args[0]+"/CSB1";
                    P4_RMI_CalcServerIntf_YP CSIntf =
(P4_RMI_CalcServerIntf_YP) Naming.lookup(CSURL);
                    System.out.println("The first number is: " + args[1]);
                    int x = Integer.parseInt(args[1]);
                    System.out.println("The second number is: " + args[2]);
                    int y = Integer.parseInt(args[2]);
    System.out.println("=======Arithmetic Operations=======");
    System.out.println("Addition: "+x+"+"+y+"="+ CSIntf.add(x,y));
    System.out.println("Subtraction: "+ x + " - "+ y + " = " +CSIntf.subtract(x,y));
                    System.out.println("Multiplication: " + x + " * "+ y + " =
"+CSIntf.multiply(x,y));
                    System.out.println("Division: "+x+"/"+y+" = " +CSIntf.divide(x,y));
                    } // try ends
                    catch(Exception e){
```

```
                                        System.out.println("Exception: " + e);
                                   }// catch ends
  } // main ends
} // class ends
```

D) OUTPUT: -