USCS303 – OS: Practical – 04: Process Communication

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 mean for processes to co-operate and compete.

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.

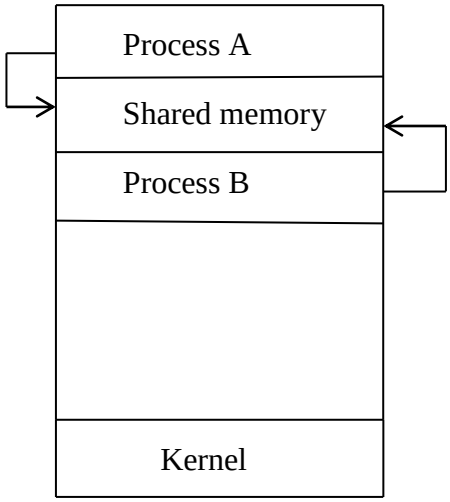
One example of a common producer/consumer relationship is print spooling. A word processor spools data to a buffer (typically a file) and that data is subsequently consumed by the printer as it prints the document. Similarly, an application that copies data onto compact discs places data in a fixed-size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

Using Shared Memory

Producer – Consumer Solution using Shared Memory

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

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



Question – 01: Write a java program for producer – consumer problem using shared memory.

Filename: P4_PC_SM_BufferImpl_ST.java

Code:

```
//Name:Sumit Telawane
//Batch:B1
//PRN: 2020016400825777
//Date:07 August  2021
//prac-04. : Process Communication


public class P4_PC_SM_BufferImpl_ST implements P4_PC_SM_Buffer_ST {
   private static final int BUFFER_SIZE = 5;
   private String[] elements;
   private int in,out,count;
 // constructor initializing the variable to initial value
   public P4_PC_SM_BufferImpl_ST () {
                count=0;
                in=0;
                out=0;
                elements=new String[BUFFER_SIZE];
   }
 // 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;
   }
}
```

Filename: P4_PC_SM_Buffer_ST.java

Code: //Name:Sumit Telawane

//Batch:B1

//PRN: 2020016400825777

//Date:07 August 2021

```
//prac-04. : Process Communication
public interface P4_PC_SM_Buffer_ST {
    // producers call this method
    public void insert (String item);
   // consumers call this method
    public String remove();
} // interface ends
```

Filename: P4_PC_SM_ST.java

Code:

```
//Name:Sumit Telawane
//Batch:B1
//PRN: 2020016400825777
//Date:07 August  2021
//prac-04. : Process Communication

public class P4_PC_SM_ST {
    public static void main(String[] args) {
        P4_PC_SM_BufferImpl_ST bufobj = new P4_PC_SM_BufferImpl_ST();
        System.out.println("\n==========PRODUCER producing the ITEMS
==========\n");
        bufobj.insert("Name: Sumit Telawane");
        bufobj.insert("CHMCS: Batch - B1");
        bufobj.insert("PRN: 2020016400825777");
        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());

    }

}
```

Output:

```
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P4\P4_PC_SM_ST>
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P4\P4_PC_SM_ST>javac P4_PC_SM_ST.java

D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P4\P4_PC_SM_ST>java P4_PC_SM_ST

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

Item produced: Name: Sumit Telawane at position 1 having total items 1
Item produced: CHMCS: Batch - B1 at position 2 having total items 2
Item produced: PRN: 2020016400825777 at position 3 having total items 3
Item produced: USCSP301 - USCS303: OS Practical-P4 at position 4 having total items 4

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

Item consumed: Name: Sumit Telawane from position 1 remaining total items 3
Name: Sumit Telawane
Item consumed: CHMCS: Batch - B1 from position 2 remaining total items 2
CHMCS: Batch - B1
Item consumed: PRN: 2020016400825777 from position 3 remaining total items 1
PRN: 2020016400825777
Item consumed: USCSP301 - USCS303: OS Practical-P4 from position 4 remaining total items 0
USCSP301 - USCS303: OS Practical-P4
```

Using Message Passing

Producer-Consumer Solution 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.

Filename: P4_PC_MP_MessageQueue_ST.java

Code: //Name:Sumit Telawane

//Batch:B1

//PRN: 2020016400825777

//Date:07 August 2021

//prac-04. : Process Communication

```
import java.util.Vector;
public class P4_PC_MP_MessageQueue_ST<E> implements P4_PC_MP_Channel_ST<E> {
   private Vector<E> queue;
 public P4_PC_MP_MessageQueue_ST(){
        queue = new Vector<E>();
   }
  // This implements a nonblocking send
   public void send(E item){
          queue.addElement(item);
   } // send() ends
 // This implements a nonblocking recieve
```

```
    public E recieve() {
        if(queue.size() == 0)
            return null;
        else
            return queue.remove(0);
    }
}
```

Filename: P4_PC_MP_Channel_ST.java

Code: //Name:Sumit Telawane

//Batch:B1

//PRN: 2020016400825777

//Date:07 August 2021

//prac-04. : Process Communication

```
public interface P4_PC_MP_Channel_ST<E> {
    // Send a message to the channel
    public void send(E item);

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

Filename: P4_PC_MP_ST.java

Code: //Name:Sumit Telawane

```
//Batch:B1

//PRN: 2020016400825777

//Date:07 August  2021

//prac-04. : Process Communication


import java.util.Date;


public class P4_PC_MP_ST {

        public static void main(String[] args){


            // procucer and consumer process

            P4_PC_MP_Channel_ST<Date> mailBox = new
P4_PC_MP_MessageQueue_ST<Date>();


            int i=0;


            do {

                Date message = new Date();

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

                mailBox.send(message);

                Date rightNow = mailBox.recieve();

                if (rightNow != null)

                        System.out.println("Producer produced - " + " : " + rightNow);

                   i++;

            }while(i<10);

        }

}
```

Output:

```
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P4\P4_PC_MP_ST>javac P4_PC_MP_ST.java

D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P4\P4_PC_MP_ST>java P4_PC_MP_ST
Producer produced - 1 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 2 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 3 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 4 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 5 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 6 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 7 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 8 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 9 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
Producer produced - 10 : Tue Aug 24 10:29:49 IST 2021
Producer produced -  : Tue Aug 24 10:29:49 IST 2021
```

Remote Method Invocation

Remote Procedure Calls

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.

Syntax: The syntax of a remote procedure call is generally of the form:

Call procedure_name(value_arguments; result_arguments)

Steps for writing RMI Program:

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 arguments 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 "CSB0". 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 the type **CalcServerIntf**. 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_ST.class, P4_RMI_CalcServerImpl_ST_Stub.class,** (if needed), and **P4_RMI_CalcServerIntf_ST.class** to a directory on the client machine.

Copy **CalcServerIntf.class, P4_RMI_CalcServerImpl_ST_Stub.class,** (if needed), and **P4_RMI_CalcServerIntf_ST.class** to a directory on the servermachine.

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_ST

Step 9: Start the Client

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

Java P4_RMI_CalcClient_ST 127.0.0.1_15_5

Filename: P4_RMI_CalcServerImpl_ST.java

Code:

```
//Name:Sumit Telawane
//Batch:B1
//PRN: 2020016400825777
//Date:07 August  2021
//prac-04. : Process Communication


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


public class P4_RMI_CalcServerImpl_ST extends UnicastRemoteObject implements
P4_RMI_CalcServerIntf_ST {
        public P4_RMI_CalcServerImpl_ST() 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 mutliply(int a, int b) throws RemoteException {
```

```
                return a*b;
        }
        public int divide(int a, int b) throws RemoteException {
                return a/b;
        }
}
```

Filename: P4_RMI_CalcServerIntf_ST.java

Code: //Name:Sumit Telawane

//Batch:B1

//PRN: 2020016400825777

//Date:07 August 2021

//prac-04. : Process Communication

```
import java.rmi.*;

public interface P4_RMI_CalcServerIntf_ST extends Remote {
        int add(int a, int b) throws RemoteException;
        int subtract(int a, int b) throws RemoteException;
        int mutliply(int a, int b) throws RemoteException;
        int divide(int a, int b) throws RemoteException;
} // interface ends
```

Filename: P4_RMI_CalcServer_ST.java

Code: //Name:Sumit Telawane

```
//Batch:B1

//PRN: 2020016400825777

//Date:07 August  2021

//prac-04. : Process Communication


import java.net.*;

import java.rmi.*;


public class P4_RMI_CalcServer_ST {

	public static void main(String[] args) {

		try {

			P4_RMI_CalcServerImpl_ST csi = new
P4_RMI_CalcServerImpl_ST();

			Naming.rebind("CSB0", csi);

		} // try ends

		catch (Exception e) {

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

		} // catch ends

	}

}
```

Filename: P4_RMI_CalcClient_ST.java

Code: //Name:Sumit Telawane

//Batch:B1

//PRN: 2020016400825777

//Date:07 August 2021

//prac-04. : Process Communication

```
import java.rmi.*;


public class P4_RMI_CalcClient_ST {
	public static void main(String[] args) {
		try {
			String CSURL = "rmi://" + args[0] + "/CSB0";
			P4_RMI_CalcServerIntf_ST CSIntf = (P4_RMI_CalcServerIntf_ST)
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("Mutliplication: " + x + " - " + y + " = " +
CSIntf.mutliply(x,y));
			System.out.println("Division: " + x + " / " + y + " = " +
CSIntf.divide(x,y));
		} // try ends
		catch(Exception e) {
			System.out.println("Exception: " + e);
```

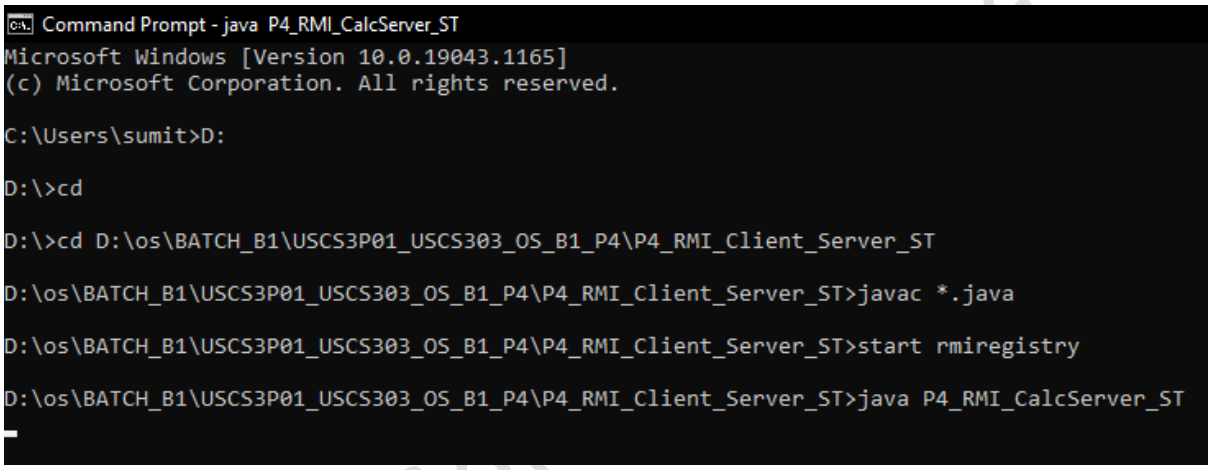
```
                } // catch ends

        } // main ends

} // class ends
```

Output:

Server



Client