Table of Contents

USCSP301: USCS303 – Operating Systems (OS)

USCS303 - OS: Practical - 04: Process Communication

Date: 06/08/2021

Aim Solution for Producer - Consumer Problem using shared memory and message passing. Communication in Client - Server environment using Remote Method Invocation (RMI).

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.

Message passing and remote procedure calls are the most common methods of interprocess communication in distributed systems.

A less frequently used but no less valuable method is distributed shared memory.

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

Producer - Consumer Problem

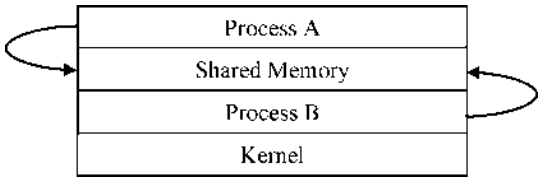
In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of an application reads data from the shared object.

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 bums the data onto the compact disc.

Solving Producer - Consumer Problem 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.



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

Source Code - 01
File Name - P4_PC_SM_Buffer_SS.java public interface P4_PC_SM_Buffer_SS
{
public void insert(String item); //Producers call this method public String remove();
//Consumers call this method } //interface ends

File Name: P4_PC_SM_BufferImpl_SS.java public class P4_PC_SM_BufferImpl_SS
implements
P4 PC SM Buffer SS
{
private static final int
BUFFER SIZE = 5 private
Stringf] elements;
private int in, out, count;
public P4_PC_SM_BufferImpl_SS() //co

count = 0;
in = 0;
out = 0;

elements = new StringfBUFFER SIZE];
} //constructor ends
public void insert(String item) // Producers call this method
<
while(count == BUFFER SIZE); //do nothing as there is no free space
//add an item to the buffer
elementsfin] = item;
... (in + 1) % BUFFER SIZE;
++count;
System.out.println("Item Produced: " + item + " at position " + in + " having total items " +
count);
} //insert()ends
public String remove() //Consumers call this method
{
in = (in + 1) % BUFFER SIZE;
String item;
while(count == 0); //do nothing as there is nothing to consume

//remove an item from the buffer

```
            item = elementsfout];
            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_SS.java

```
        public static void main(String[] args)
        {
        P4_PC_SM_BufferImpl_SS    bufobj    =    new
        P4_PC_SM_BufferImpl_SS();
        System.out.println("\n=========  PRODUCER
        producing  ITEMS  =====  bufobj.insert("Name:
        Subrat Chitrasen Sahu");
        bufobj ,insert("CHMCS: Batch-B2");
        bufobj.insertf'PRN: 2020016400833692");
        bufobj.insert("USCS303 - OS: Practical - 04: Process Communication");
        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
```

Solvong
producer
consumer
problem 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.

Question - 02

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

Source Code - 02

File Name: P4_PC_MP_Channel_SS.java

```
public interface P4_PC_MP_Channel_SS<E> {
    public void send(E item); // Send a message to the channel public E receive(); // Receive a message from the channel } //interface ends
```

File Name: P4_PC_MP_MessageQueue_SS.java

```
import java.util. Vector;
public class P4_PC_MP_MessageQueue_SS<E> implements P4 PC J
{
    private Vector<E> queue;
    public P4_PC_MP_MessageQueue_SS()
    {
    public void send(E item) // This implements a
        queue. addEl em ent(item);
    } // send() ends
    public E receive() // This implements a non-blocking receive
        if(queue.size() ==
            return null.
        else
            return queue.remove(O); } // receive()
} // class ends
```

File Name: P.J MP SS.java

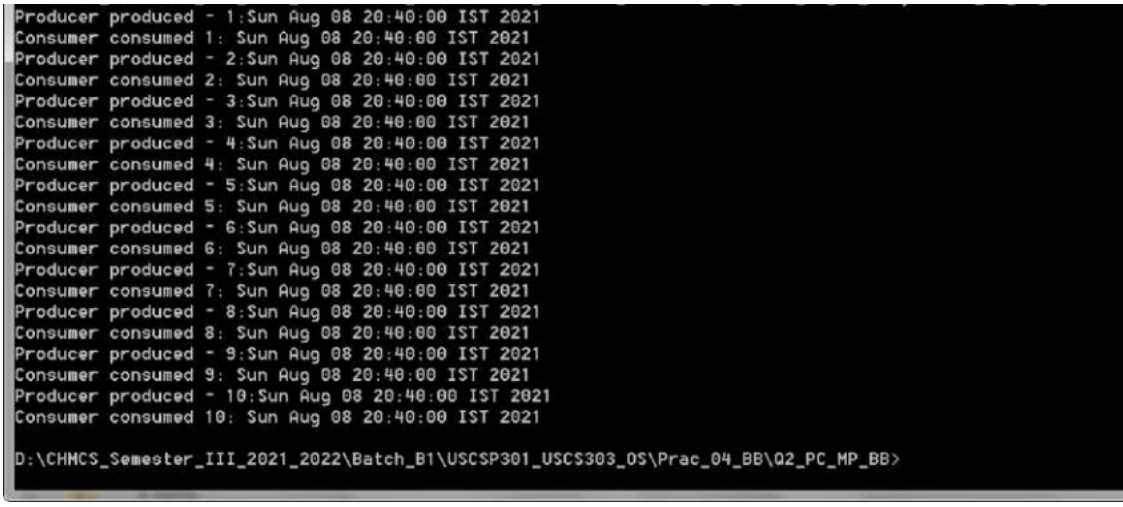
```
import j ava.util .Date;
public class P4PCMPSS
{
    public static void main(String args[])
    {
        // Producer and Consumer process
        P4_PC_MP_Channel_SS<Date> mailBox = new
    P4_PC_MP_MessageQueue_SS<Date>();
int i=0;

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
```

Output 2:



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.

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

call procedure_name(value_arguments; result arguments)

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

The remote procedure is the server processes which has already begun executing 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 the call() 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 the server stub 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.

Question - 03

Write a java RMI program for adding, subtracting, multiplying and dividing two numbers.

Steps for writing RMI

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 Remote Exception.

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 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 "CSBO". 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 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_SS.class, P4_RMI_CalcServerImpl_SS_Stub.class (if needed), and P4_RMI_CalcServerIntf_SS.class to a directory on the client machine.

Copy CalcServerintf.class, P4_RMI_CalcServerImpl_SS.class, P4_RMI_CalcServerImpl_SS_Stub.class (if needed), P4_RMI_CalcServer_SS.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_SS

Step 9: Start the Client

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

java P4_RMI_CalcClient_SS 127.0.0.1

Terminal - 01:
Compile all the .java files
C ommand: j avac * .j ava
Start the RMI registry:
Command: start rmiregistry
Start the Server:
Command: java P4_RMI_CalcServer_SS
Terminal - 02:
Run the Client:
Command: java P4_RMI_CalcClient_SS 127.0.0.1 15 5

Source Code - 03
File Name: P4_RMI_CalcServerIntf_SS.jav;

15 5

```
import java.rmi.*;
public interface P4_RMI_CalcServerIntf_SS extends Remote (
    int add(int a, int bjthrows RemoteException; int subtract(int a, int bjthrows RemoteException; int multiply(int a, int bjthrows RemoteException; int divide(int a, int bjthrows RemoteException; } //interface ends import java.rmi.p;
import j ava. rmi. server. *;
public class P4_RMI_CalcServerImpl_SS extends
UnicastRemoteObject implements
P4_RMI_CalcServerIntf_SS
{
    public P4_RMI_CalcServerImpl_SS()throws RemoteException {

    }
public int add(int a, int bjthrows 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
```

File Name: P4_RMI_CalcServer_SS.java

```
import java.net.*;
import java.rmi.*;
public class P4_RMI_CalcServer_SS
{
```

```
public static void main(String args[])

{try




P4_RMI_CalcServerImpl_SS csi = new P4_RMI_CalcServerImpl_SS();

Naming.rebind("CSB1", csi);

}//try ends

catch(Exception e)

{

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

}//catch ends

}//main ends

}//class ends
```

File Name: P4_RMI_CalcClient_SS.java

```
import java.rmi.*;

public class P4_RMI_CalcClient_SS

{

public static void main(String args[])

try
{
        String CSURL = "rmi://"+args[0]+"/CSB1";
        P4_RMI_CalcServerIntf_SS CSIntf = (P4_RMI_CalcServerIntf_SS)
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 + " = " +
```

```
C SIntf.subtract(x,y));
        System.out.println("Addition: " + x + " * " +
CSIntf.multiply(x,y));
        System.out.println("Addition: " + x +
CSIntf.divide(x,y));
}//try ends
catch(Exception e)
{
        }//main ends
}//class ends
```

Output - 03