**Multithreading.**

Students Name

Institutional Affiliation

Course

Instructors Name

Date

**Task 1**

The reason for this is because Java is an object-oriented programming language. Java provides built-in functionality for concurrent threading of execution. Java's implementation of multithreading makes it possible for several sections to be processed in parallel. In reality, a Java thread is just a lightweight process that runs in the background. I will be using it for my project.

**Task 2**

The idea of switching contexts is fundamental to the practice of threading. The CPU consults a hardware timer in order to ascertain when a particular thread's timeslice will have reached its conclusion. When the timer detects that the timeslice has completed, it sends a signal to the processor, which then stores all of the information that is necessary for the active thread on a stack. The information is then transferred by the processor from the stack into a preset data structure that is referred to as a context structure. When the processor has to switch back to a thread that was previously running, it moves all of the relevant information from the context structure that is connected with the thread onto the stack. Changing one's perspective entirely is referred to as context flipping.

Java allows for multitasking via the use of threads. In comparison to multitasking based on processes, thread-based multitasking has a number of benefits, some of which are listed below:

* The address space is shared by all of the threads.
* Switching between different threads often does not incur any additional costs.
* Communication across threads often does not incur any additional costs.

Runnable Interface

Developing a class that conforms to the Runnable interface is the simplest method for producing a thread in your program.

A class only has to implement a single method known as run() in order to conform to the requirements of the Runnable interface, which is specified as follows:

public void run( )

public class MyClass implements Runnable {

public void run(){

System.out.println("MyClass running");

}

}

**Task 3**

Java offers a remedy for the problem of interprocess communication by using distributed computing in conjunction with the Internet Protocol (IP) loopback network (that is, by making use of the 127.0.0.0/8 network address family). This indicates that communication may take place between different processes running on the same computer at the same time. This is accomplished via the use of Java's standard socket API and takes the form of socket programming that makes use of several network protocols (such as TCP and UDP) It is possible to encapsulate messages into network packets and then transport those packets as required over the IP stack. By utilizing the loopback network to send messages to applications that are executing in distinct JVMs, an instance of interprocess communication (IPC) is created, the communication method of Java's TCP/UDP network support is inefficient.

By using Remote Method Invocation (RMI), methods in an object may be invoked even if they are being executed in a different JVM. Classes that use Remote Method Invocation (RMI) basically operate as a collection of JVMs that may exchange messages with one another. A packetized version of the method's arguments is transmitted over the network to another JVM, which then executes the remote method with the appropriate parameters. Since the calling process is unaware of the status of the callee, it is the responsibility of the object to which the remote method belongs to provide appropriate safeguards. When implementing a loopback network, RMI may be utilized to facilitate communication between the two ends. Common Object Request Broker Architecture (CORBA) is a comparable technology that also uses remote calling of methods to create a distributed solution; it also supports several languages The Java Message Service (JMS) is another kind of distributed communication that allows programs to talk to one another, although these high-level distributed techniques do solve IPC, they do so at the expense of performance.

**Task 4 client side**

package client;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.Socket;

import java.net.UnknownHostException;

public class Client {

// initialize socket and input output streams

private Socket socket = null;

private DataInputStream input = null;

private DataOutputStream out = null;

// constructor to put ip address and port

public Client(String address, int port) {

try {

socket = new Socket(address, port);

System.out.println("connected");

System.out.println("hello world");

input = new DataInputStream(System.in);

// sends output to the socket

out = new DataOutputStream(socket.getOutputStream());

} catch (UnknownHostException u) {

System.out.println(u);

} catch (IOException i) {

System.out.println(i);

}// string to read message from input

String line = "";

// keep reading until "Over" is input

while (!line.equals("Over")) {

try {

line = input.readLine();

out.writeUTF(line);

} catch (IOException i) {

System.out.println(i);

}

}

// close the connection

try {

input.close();

out.close();

socket.close();

} catch (IOException i) {

System.out.println(i);

}

}

public static void main(String args[]) {

Client client = new Client("127.0.0.1", 5000);

}

}

**Task 5**

package server;

import java.io.BufferedInputStream;

import java.io.DataInputStream;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

public class Server {

//initialize socket and input stream

private Socket socket = null;

private ServerSocket server = null;

private DataInputStream in = null;

public Server(int port) {

try {

server = new ServerSocket(port);

System.out.println("Server started");

System.out.println("Waiting for a client ...");

socket = server.accept();

System.out.println("Client accepted");

in = new DataInputStream(new BufferedInputStream(socket.getInputStream()));

String line = "";

while (!line.equals("Over")) {

try {

line = in.readUTF();

System.out.println(line);

} catch (IOException i) {

System.out.println(i);

}

}

System.out.println("Closing connection");

socket.close();

in.close();

} catch (

IOException i) {

System.out.println(i);

}

}

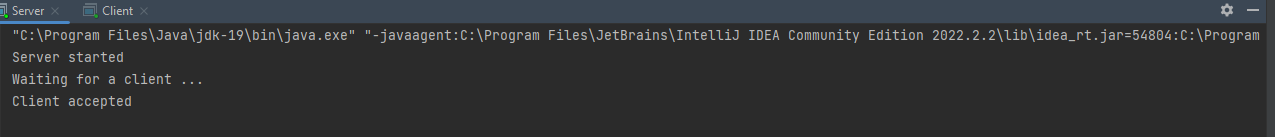
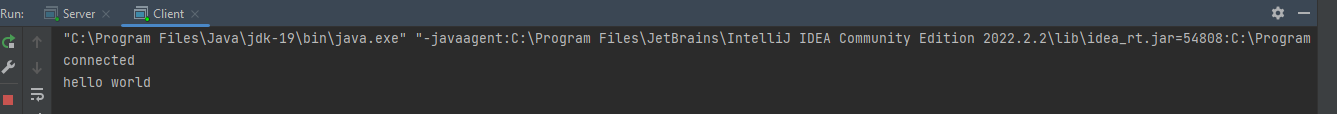
public static void main(String args[]) {

Server server = new Server(5000);

}

}

Task 6



After you have finished setting both the client and the server end, you may begin executing the program on the server side. After that, you will need to submit the request while running the client-side software. The server will react as quickly as possible after the request has been submitted from the client's end. You may also run these applications by typing their commands into a command prompt or terminal window. On the other hand, given that Eclipse and Intellij have developed quite a few useful capabilities, you can easily run any of these applications on a console.