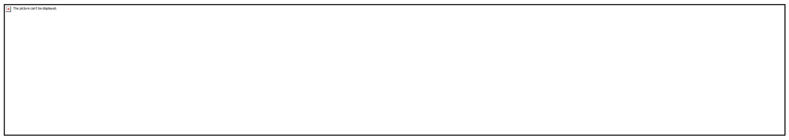


```
/*
Name: Rohit Saini
Erp: 1032200897
Panel: C
RollNo: PC-41 */
```

TITLE:

Write a program to implement Echo server using socket programming

AIM:

To implement Client-Server architecture as echo server using

1. Socket programming
2. Multi-threading

OBJECTIVE:

To understand the concept of socket programming, multi-threading and echo servers.

THEORY:

Most inter process communication uses the client server model. One of the two processes, the client, typically to make a request for information .The system calls for establishing a connection for the client and server are as follows:

Client side:

1. Socket ()
2. Connect ()
3. Read ()
4. Write ()

Server Side:

1. Socket ()
2. Bind ()
3. Listen ()
4. Accept ()
5. Read ()
6. Write()

SOCKET TYPES:

When a socket is created , the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of same type and in the same domain. There are two widely used domains:

1. UNIX domain
2. internet domain

There are two widely used socket types.

1. Stream sockets
2. Datagram sockets

Stream sockets treat communication as continuous stream of characters, while datagram sockets have to read the entire message at once. Each uses its own communication protocol. Stream sockets use TCP , which is reliable , stream oriented protocol and datagram sockets are UDP , which is unreliable and message oriented .

SYSTEM CALL FORMATS:

1. int Socket(int family , int type , int protocol)

Type: f Socket type
SOCK_DGRAM------- UDP
SOCK_STREAM ----- TCP

Protocol: Type = 0------------- TCP
Type = 1------------- UDP

The above call returns socket descriptor.

2. int Bind(int SOCK_FD,struct sockaddr *myadddr , int addrlen)
use:
Binding socket with the address.

3. Listen (int SOCK_FD, int backlog)
Where
Backlog = number of requests that can be queued up to the server .

4. Accept (int SOCK_FD, struct sockaddr *peer, int *addrlen)
Where
int *adddrlen = length of the structure

5. Connect (int SOCK_FD, struct sock addr *servaddr , socklen_t addrlen)

RANGE OF PORTS:
1. Wellknown ports: 0-1023
2. Registered: 1024 – 59151

(Controlled by IANA)

3. Dynamic (Ephemeral): 49152-65535

Reserved port in UNIX is any port < 1024

MULTITHREADED ECHO SERVER:

Multiple clients request for service to the same server. Server creates threads to serve the clients. Threads are light weight processes. It provides concurrency. On the server side, process keeps listening to the requests made by clients. As soon as the connection has to be made with clients, server creates the threads.

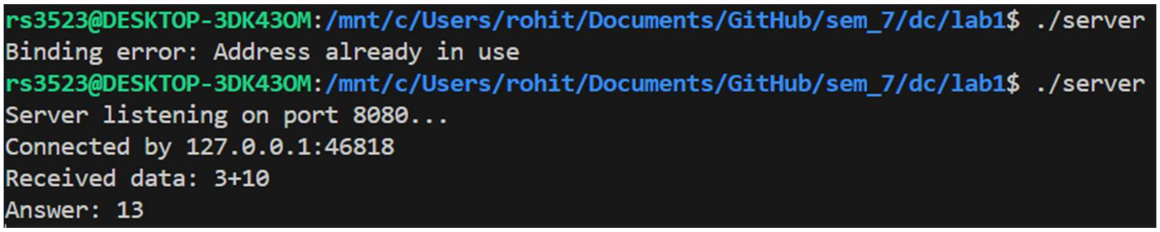
INPUT:

3+4

OUTPUT:

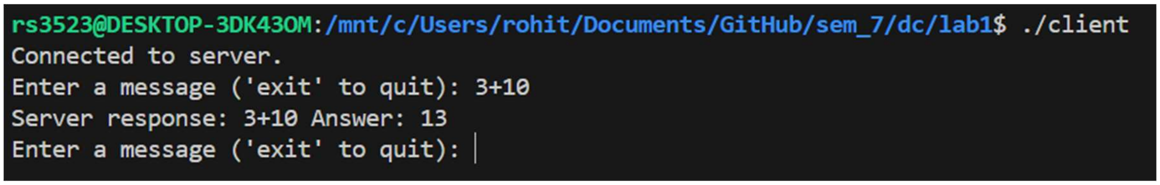
3+4 Answer=7

Server:



```
rs3523@DESKTOP-3DK43OM:/mnt/c/Users/rohit/Documents/GitHub/sem_7/dc/lab1$ ./server
Binding error: Address already in use
rs3523@DESKTOP-3DK43OM:/mnt/c/Users/rohit/Documents/GitHub/sem_7/dc/lab1$ ./server
Server listening on port 8080...
Connected by 127.0.0.1:46818
Received data: 3+10
Answer: 13
```

Client:



```
rs3523@DESKTOP-3DK43OM:/mnt/c/Users/rohit/Documents/GitHub/sem_7/dc/lab1$ ./client
Connected to server.
Enter a message ('exit' to quit): 3+10
Server response: 3+10 Answer: 13
Enter a message ('exit' to quit):
```

CONCLUSION:

Echo server is implemented using sockets and multithreading.

PLATFORM:

Linux

LANGUAGE:

C language.

CODE:

//Client

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
```

```
int main()
{
    int client_socket;
    struct sockaddr_in server_addr;
    char buffer[1024];

    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1)
    {
        perror("Socket creation error");
        return 1;
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    if (connect(client_socket, (struct sockaddr
*)&server_addr, sizeof(server_addr)) == -1)
    {
        perror("Connection error");
        return 1;
    }

    printf("Connected to server.\n");

    while (1)
    {
        printf("Enter a message ('exit' to quit): ");
        fgets(buffer, sizeof(buffer), stdin);

        // Remove trailing newline
        buffer[strlen(buffer) - 1] = '\0';
        if (send(client_socket, buffer, strlen(buffer), 0)
== -1)
        {
            perror("Send error");
            break;
        }

        if (strcmp(buffer, "exit") == 0)
        {
            break;
        }

        ssize_t bytes_received = recv(client_socket,
buffer, sizeof(buffer), 0);
        if (bytes_received == -1)
        {
            perror("Receive error");
            break;
        }

        printf("Server response: %.*s\n",
(int)bytes_received, buffer);
    }
    close(client_socket);
    return 0;
}

/*
gcc client.c -o client -Wall
*/
//Server Code
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```
#include <pthread.h>

int evaluateExpression(const char *expression)
{
    int num1, num2;
    char operator;
    sscanf(expression, "%d%c%d", &num1, &operator, &
num2);

    switch (operator)
    {
    case '+':
        return num1 + num2;
    default:
        printf("Unsupported operator: %c\n", operator);
        return 0;
    }
}

int add(const char *data)
{
    int result = evaluateExpression(data);
    // printf("Result: %d\n", result);
    return result;
}

void *clientHandler(void *clientSocketPtr)
{
    int client_socket = *((int *)clientSocketPtr);
    char buffer[1024];

    while (1)
    {
        ssize_t bytes_received = recv(client_socket,
buffer, sizeof(buffer), 0);
        if (bytes_received == -1)
        {
            perror("Receive error");
            break;
        }

        if (bytes_received == 0 || strcmp(buffer, "exit")
== 0)
        {
            break;
        }

        printf("Received data: %.*s\n",
(int)bytes_received, buffer);

        int ans = add(buffer);
        printf("Answer: %d\n", ans);

        char ansStr[20];
        snprintf(ansStr, sizeof(ansStr), " Answer: %d",
ans);
        strcat(buffer, ansStr);

        send(client_socket, buffer, strlen(buffer), 0);
    }

    close(client_socket);
    return NULL;
}

int main()
{
    int MAX_CLIENTS = 1;
    int server_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1)
    {
        perror("Socket creation error");
        return 1;
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(8080);

    if (bind(server_socket, (struct sockaddr
*)&server_addr, sizeof(server_addr)) == -1)
    {
        perror("Binding error");
        return 1;
    }

    if (listen(server_socket) == -1)
    {
        perror("Listening error");
        return 1;
    }

    printf("Server listening on port 8080...\n");

    int connectedClients = 0;
    while (connectedClients < MAX_CLIENTS)
    {
        newSocket = accept(server_socket, (struct sockaddr
*)&client_addr, &client_len);
        if (newSocket == -1 && connectedClients <
MAX_CLIENTS)
        {
            perror("Client connection failed");
            exit(EXIT_FAILURE);
        }

        connectedClients++;
        printf("Connected by %s:%d\n",
inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));

        pthread_t thread;
        int result = pthread_create(&thread, NULL,
clientHandler, &newSocket);
        if (result != 0)
        {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }

        pthread_detach(thread);
    }

    close(server_socket);

    return 0;
}

/*
gcc server.c -o server -Wall
 */
```

FAQs

1. Give the differences between UDP and TCP protocols
2. How does the accept system call work in socket programming
3. What is the advantage of using threads in socket programming

DS Lab 1

Q1. Give the difference between UDP and TCP protocols?

Ans. UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) are both transport layer protocols used for communication over networks, but they have distinct differences:

Feature	UDP (User Datagram Protocol)	TCP (Transmission Control Protocol)
1. Connection Type	connection less	connection-oriented.
2. Reliability	unreliable	Reliable
3. Order	No guaranted order	Guaranteed Order
4. Flow Control	No flow control mechanism	Implement flow control
5. Error Detection	Limited error checking	Extensive error checking

Q2. How the accept system call works in socket programming?

Ans. In socket programming, the 'accept' system call is used by server to accept the incoming ~~to~~ client connections. It's typically used with TCP protocol, which operates in a connection-oriented manner. Here's how the accept system call works.

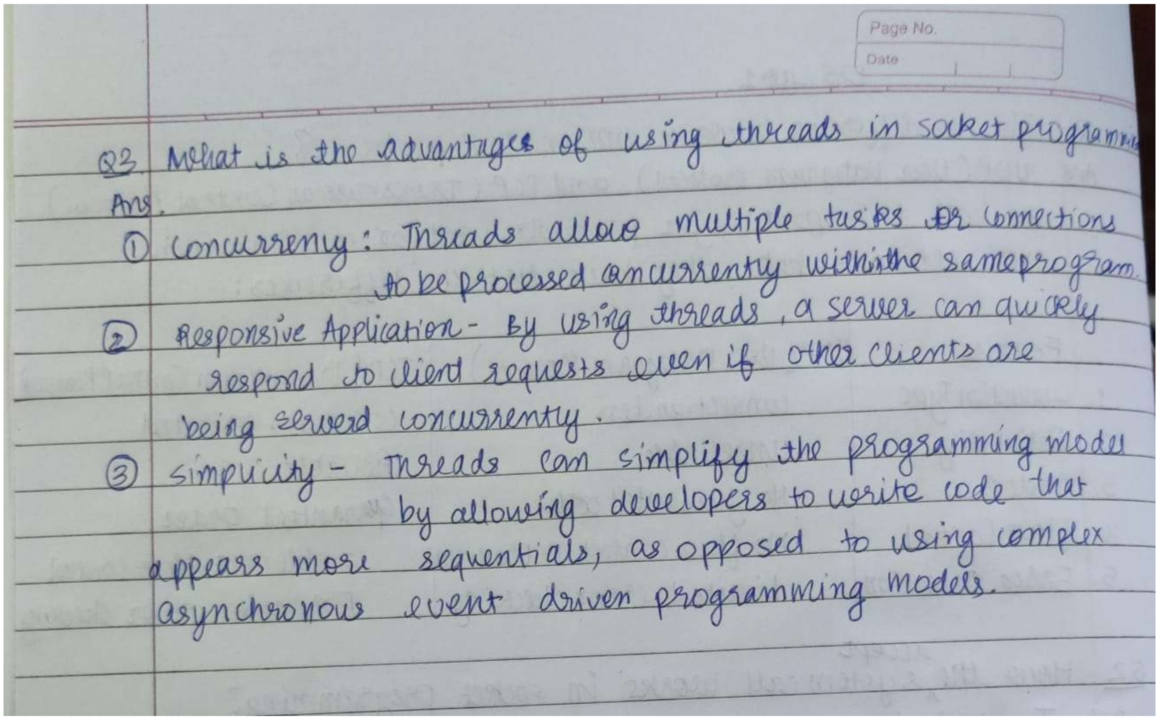
1. Server Setup: the server creates a socket, binds it to ~~b~~ a specific address and port, and then enters a listening state using the listen system call.

2. Blocking Call: The server application executes the accept system call. This blocks the server's execution until a client connection request arrives.

3. Client connection: when client tries to establish the connection with server, the server OS detect the incoming connection request and places it in queue.

4. Acceptance: when accept is executed, it extracts the first connection request from the queue and create new socket dedicated to that client connection.

5. Return: The accept call returns the new socket descriptor to the server application, allowing the server to send and receive data from the client using this socket.



Page No.

Date

Q3. What is the advantages of using threads in socket programm

Ans.

① concurreny: Threads allow multiple tasks or connections to be processed concurrenty withinthe sameprogram.

② Responsive Application- By using threads, a server can quickly respond to client requests even if other clients are being serverd concurrently.

③ simplicity - Threads can simplify the programming model by allowing developers to write code that appears more sequentials, as opposed to using complex asynchronous event driven programming models.