Computer Networks Lab Assignment

Concurrent Server using TCP Socket

Name – Umang

Reg No – 20214199

Group – D

Program # 1

String Reversal Concurrent Server using TCP Sockets

Change the server program in the previous assignment from iterative to concurrent, so that more than one clients can be handled at a time.

A concurrent server is a server that waits on the welcoming socket and then creates a new thread or process to handle the incoming request. In this program use fork() system call to create new processes. There is a single process accepting incoming connections. After a connection is made, the server forks a copy of itself to process the request and then returns to accepting incoming connections. Modify your string reversal server code to fork a new process after accepting a new connection. The child process should handle the actual string reversal task, whereas the parent should continue to accept new connections.

Server Code –

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

#define PORT 50800
#define BUFFER_SIZE 512

// Function to reverse a string
void reverseString(char *str)
{
    int len = strlen(str);
    for(int i=0; i < len/2;i++)
    {
        char temp = str[i];
```

```
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
}

int main()
{
    int server_fd, client_socket;

    struct sockaddr_in server_addr, client_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    memset(&client_addr, 0, sizeof(client_addr));

    int addrlen = sizeof(client_addr);

    // Creating socket file descriptor
    if((server_fd=socket(AF_INET, SOCK_STREAM, 0))==0)
    {
        perror("socket failed");
        exit(1);
    }

    // Filling server information
    server_addr.sin_family = AF_INET; // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if(bind(server_fd,(struct sockaddr *)&server_addr,sizeof(server_addr)) < 0)
    {
        perror("bind failed");
        exit(1);
    }

    // Listen for incoming connections
    if(listen(server_fd, 3) < 0)
    {
        perror("listen");
        exit(1);
    }

    printf("Server listening on port %d...\n", PORT);

    // Accept incoming connections and fork to handle each client
    while(1)
    {
        // Accept the incoming connection
        if((client_socket = accept(server_fd, (struct sockaddr *)&client_addr,(socklen_t
*)&addrlen)) < 0)
        {
            perror("accept");
            exit(1);
        }
        printf("Connection established with client\n");
```

```
        // Fork to handle client connection
        pid_t pid = fork();
        if(pid == 0)
        {
            close(server_fd); // Close server socket in child process
            // child process handles client request

            char buffer[BUFFER_SIZE] = {0};

            // Receive message from client
            int valread;
            if((valread = read(client_socket, buffer, BUFFER_SIZE)) < 0)
            {
                perror("read");
                close(client_socket);
                exit(1);
            }
            printf("Received message from client: %s\n", buffer);

            // Reverse the message
            reverseString(buffer);

            // Send reversed message back to client
            send(client_socket, buffer, strlen(buffer), 0);
            printf("Reversed message sent to client: %s\n", buffer);

            // Close client socket and exit child process
            close(client_socket);
            exit(0);
        }
        else if(pid < 0)
        {
            // Error handling
            perror("fork");
            close(client_socket);
        }
        else
        {
            close(client_socket); // Close client socket in parent process
            // parent process remains open and listens for new client requests
        }
    }
    return 0;
}
```

Client Code –

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

```c
#define PORT 50800
#define BUFFER_SIZE 512

int main()
{
    int sock;

    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));

    char buffer[BUFFER_SIZE] = "MNNIT ALLAHABAD";

    // Creating socket file descriptor
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        {perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Filling server information
    server_addr.sin_family = AF_INET; // IPv4
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // Connect to the server
    if(connect(sock,(struct sockaddr *)&server_addr,sizeof(server_addr)) < 0)
    {
        perror("connect failed");
        exit(EXIT_FAILURE);
    }

    printf("Connected to server\n");

    // Send message to server
    send(sock, buffer, strlen(buffer), 0);
    printf("Message sent to server: %s\n", buffer);

    // Receive reversed message from server
    int valread;
    if((valread = read(sock, buffer, BUFFER_SIZE)) < 0)
    {
        perror("read");
        exit(EXIT_FAILURE);
    }
    buffer[valread] = '\0';
    printf("Reversed message received from server: %s\n", buffer);

    close(sock);

    return 0;
}
```

For each accepted new connection, the file descriptor needs to be closed in two places:

- In the child process: After handling the client connection, the child process should close the client socket file descriptor.
- In the parent process: After forking a child process to handle the client connection, the parent process should close the client socket file descriptor.

These two closings ensure that the socket file descriptor is properly closed in both the parent and child processes, preventing any resource leaks.

Program #2

Do the same modification in the server code CalcServerTCP.c written for Program # 2 in Lab Assignment 6 to create a concurrent sever that can handle multiple clients at the same time. In this program use POSIX threads to create concurrency. Test your code by connecting multiple clients at the same time.

Server code –

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

#define BUFFER_SIZE 512

float evaluateExp(char* exp)
{
    float a,b;
    char op;
    sscanf(exp,"%f %c %f",&a,&op,&b);

    switch(op)
    {
        case '+':return a+b;
        break;
        case '-':return a-b;
        break;
        case '*':return a*b;
        break;
```

```
        case '^':return pow(a,b);break;
        case '/':
            if(b==0)
            {
                printf("Cannot divide by zero.\n");
                return -1;
            }
            else return a/b;
        break;
        default:
            printf("Invalid operator.\n");
    }

    return -1;
}

void *client_handler(void *arg) {
    int client_socket = *((int *)arg);
    char buffer[BUFFER_SIZE] = {0};

    // Handle client messages
    while(1)
    {
        // Receive message from client
        int valread;
        if((valread = read(client_socket, buffer, BUFFER_SIZE)) < 0)
        {
            printf("Read operation failed.\n");
            close(client_socket);
            pthread_exit(NULL);
        }
        if(valread == 0)
        {
            printf("Client disconnected.\n");
            break;
        }

        printf("Received from client: %s\n", buffer);

        float res = evaluateExp(buffer);
        sprintf(buffer, "%.5f", res);

        // Send the result back to client
        printf("Sending to client: %s", buffer);
        send(client_socket, buffer, strlen(buffer), 0);
    }

    printf("Client closed connection.\n");
    // Close the socket for this client
    close(client_socket);
    pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf("Usage: %s <port>\n", argv[0]);
        return 1;
    }
    int port = atoi(argv[1]);

    int server_fd, new_socket;
    struct sockaddr_in server_addr, client_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    memset(&client_addr, 0, sizeof(client_addr));

    int addrlen = sizeof(client_addr);
    char buffer[BUFFER_SIZE] = {0};

    // Creating socket file descriptor
    if((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        printf("Socket creation failed.\n");exit(1);
    }

    // Filling server information
    server_addr.sin_family = AF_INET; // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port);

    // Bind the socket with the server address
    if(bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        printf("Bind operation failed.\n");
        exit(1);
    }

    // Listen for incoming connections
    if(listen(server_fd, 3) < 0)
    {
        printf("Server starts to listen.\n");
        exit(1);
    }

    printf("Server listening on port %d...\n", port);

    // Accept incoming connections and handle each client in a separate threadwhile(1)
    {
        // Accept the incoming connection
        if((new_socket = accept(server_fd, (struct sockaddr *)&client_addr, (socklen_t
*)&addrlen)) < 0)
        {
            printf("Accept operation failed.\n");
            exit(1);
```

```
        }
        printf("Connection established with client.\n");

        // Create a new thread to handle client connection
        pthread_t thread;
        if(pthread_create(&thread, NULL, client_handler, &new_socket) != 0)
        {
            printf("Failed to create thread.\n");
            close(new_socket);
        }

        // Detach the thread so its resources are automatically cleaned up
        if(pthread_detach(thread) != 0)
        {
            printf("Failed to detach thread.\n");
            close(new_socket);
        }
    }

    return 0;
}
```

Client code –

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

#define BUFFER_SIZE 512

int main(int argc, char *argv[])
{
    if(argc!=3)
    {
        printf("Usage: %s <server_ip> <port>\n", argv[0]);
        return 1;
    }
    char *server_ip=argv[1];
    int port=atoi(argv[2]);

    int sock;
    struct sockaddr_in server_addr;
    memset(&server_addr,0,sizeof(server_addr));

    char buffer[BUFFER_SIZE],copy[BUFFER_SIZE];

    // creating socket file descriptor
    if((sock=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        printf("Socket creation failed.\n");
        exit(1);
```

```
    }

    // filling server information
    server_addr.sin_family=AF_INET; // IPv4
    server_addr.sin_port=htons(port);

    if(inet_pton(AF_INET,server_ip,&server_addr.sin_addr)<1)
    {
        printf("Invalid IPv4 address.\n");
        exit(1);
    }

    // connect to the server
    if(connect(sock,(struct sockaddr *)&server_addr,sizeof(server_addr))<0)
    {
        printf("Connection failed.\n");
        exit(1);
    }

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

    // keep sending and receiving messages until user closes the process
    printf("Enter an expression in the following format:\noperand1 operator
operand2\nValid operators are + - * / ^.\nTo quit, enter -1.\n");
    while(1)
    {
        fgets(buffer, BUFFER_SIZE, stdin);
        copy=buffer;

        // remove newline character
        buffer[strcspn(buffer,"\n")]='\0'; // assuming newline occurs only once at the
very end

        // check if user wants to quit
        if(strcmp(buffer,"-1")==0)
        {
            printf("Bye!\n");
            break;
        }

        // send message to server
        send(sock,buffer,strlen(buffer),0);

        printf("Message sent to server: %s\n",buffer);

        // receive response from server
        int valread;
        if((valread=read(sock,buffer,BUFFER_SIZE))<0)
        {
            printf("Read operation failed.\n");
            exit(1);
        }

        buffer[valread]='\0';
```

```
        printf("ANS: %s = %s\n",copy,buffer);
    }
    // close the socket
    close(sock);

    return 0;
}
```