Fatima
Business School
SALIM HABIB UNIVERSITY

OPERATING SYSTEM COURSE PROJECT REPORT

GROUP MEMBERS:

BILAL NAEEM (F23CSC008)

SHAHJAHAN (F23CSC021)

COURSE INSTRUCTOR: MISS SUMRA KHAN

1. TITLE OF THE PROJECT

Threaded Chat Application in C Using Multithreading and Socket Programming

2. INTRODUCTION / OVERVIEW

Operating Systems play a vital role in managing system resources such as CPU, memory, processes, threads, and communication between processes. One of the most important concepts in modern operating systems is **multithreading**, which allows multiple tasks to execute concurrently within a single process. Another important concept is **Interprocess Communication (IPC)**, which enables processes to exchange data and coordinate their actions.

This project, *Threaded Chat Application in C*, is designed to demonstrate the practical implementation of multithreading and socket-based interprocess communication in a Linux environment. The application consists of a server and multiple clients that can communicate with each other in real time. The server uses POSIX threads (pthreads) to handle multiple clients simultaneously, ensuring efficient and responsive communication.

3. OBJECTIVES / GOALS

The main objectives of this project are:

- ✓ To understand the concept of multithreading and its implementation using the pthread library in C.
- ✓ To study socket programming and TCP-based client-server communication.
- ✓ To demonstrate Interprocess Communication (IPC) through network sockets.
- ✓ To build a real-time chat system that supports multiple simultaneous clients.
- ✓ To understand how an operating system schedules threads and manages concurrent execution.

4. LITERATURE REVIEW

Traditional single-threaded server applications can handle only one client request at a time, leading to delays and poor performance when multiple clients attempt to connect simultaneously. Research and studies on concurrent server architectures show that multithreaded servers significantly improve performance, responsiveness, and resource utilization.

Socket programming using TCP/IP is a widely adopted mechanism for network-based communication between processes. POSIX threads provide a standardized API for thread creation and synchronization in Unix-like operating systems. Many real-world applications such as chat servers, web servers, and online gaming systems use multithreading and socket programming to handle large numbers of clients efficiently.

This project draws inspiration from classical client-server models and concurrent programming techniques used in modern operating systems.

5. PROBLEM STATEMENT

In a single-threaded communication system, the server can handle only one client at a time. When multiple clients try to connect simultaneously, the server becomes unresponsive or introduces delays, making real-time communication impossible.

The problem is to design and implement a system that allows multiple clients to communicate with each other simultaneously without affecting system performance or responsiveness.

6. PROPOSED SOLUTION / METHODOLOGY

The proposed solution is to develop a **multithreaded chat application** using C on a Linux platform. The system follows a client-server architecture:

- ✓ The server creates a TCP socket, binds it to a specific port, and listens for incoming client connections.
- ✓ For every new client connection, the server creates a separate thread using pthread_create().
- ✓ Each thread independently handles communication with its assigned client.
- ✓ Messages received from one client are broadcast to all other connected clients.

This approach ensures concurrent execution, efficient resource utilization, and real-time communication.

7. MODULES / COMPONENTS

Module 1: Server Initialization

- ✓ Creates a TCP socket
- ✓ Binds the socket to a port number
- ✓ Listens for incoming client connections

Module 2: Client Connection Handler (Thread)

- ✓ A separate thread is created for each connected client
- ✓ Receives messages from the client
- ✓ Handles client disconnection

Module 3: Message Broadcasting System

- ✓ Broadcasts messages received from one client to all other connected clients
- ✓ Uses synchronization mechanisms to avoid race conditions

Module 4: Client Program

- ✓ Connects to the server using a TCP socket
- ✓ Sends user messages to the server
- ✓ Receives and displays messages from other clients

8. EXPECTED OUTPUT / RESULTS

The expected results include:

- ✓ Multiple clients can connect to the server simultaneously.
- ✓ Clients can exchange messages in real time.
- ✓ No blocking or delay occurs due to concurrent thread execution.
- ✓ Efficient utilization of system resources.
- ✓ Demonstration of multithreading, thread scheduling, and IPC concepts in an operating system.

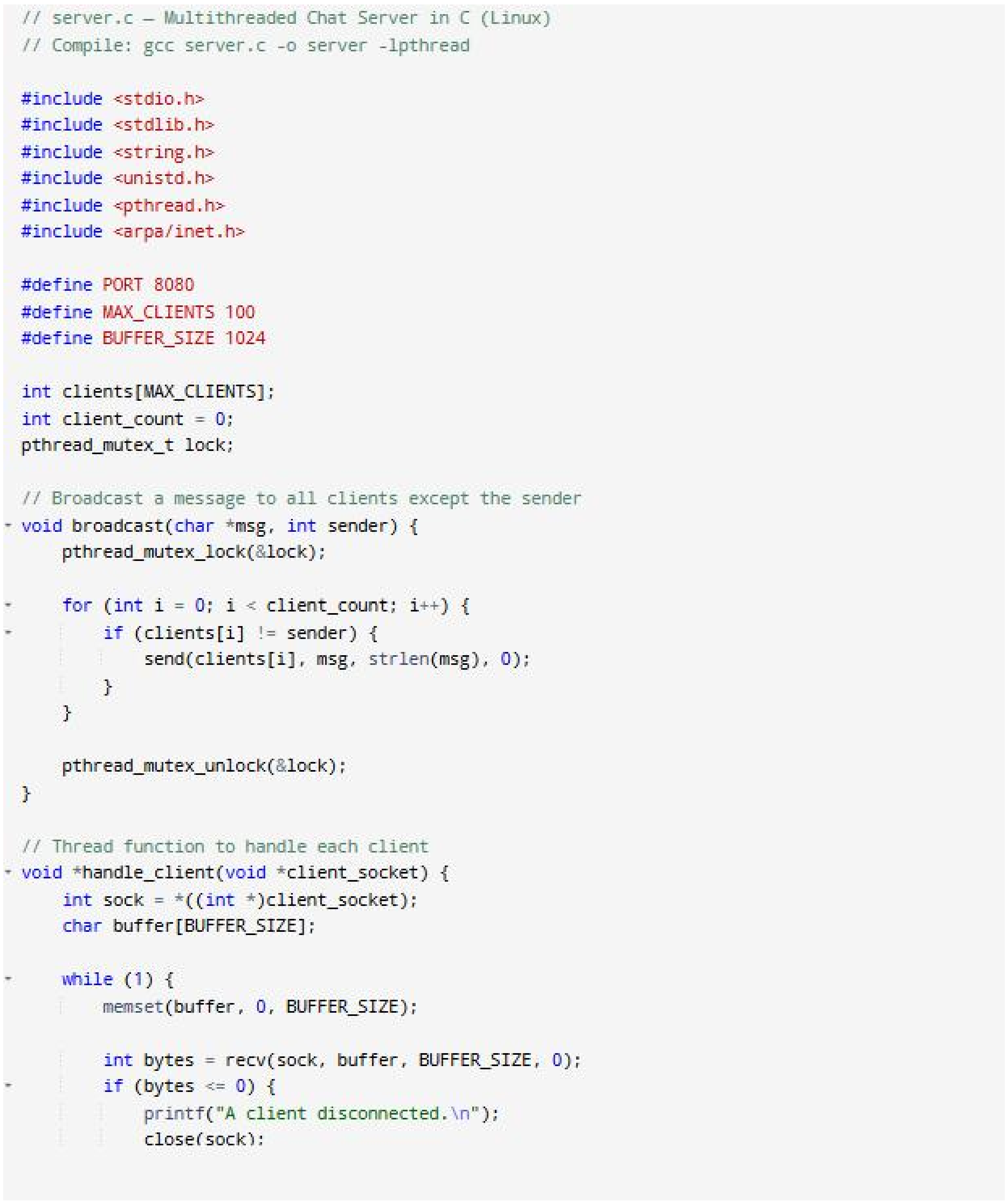
9. TOOLS AND TECHNOLOGIES

Programming Language: C
Operating System: Linux (Ubuntu / Kali Linux)
Libraries Used:
- ✓ pthread.h for multithreading
- ✓ sys/socket.h and arpa/inet.h for socket programming
- ✓ unistd.h and string.h for system calls and string operations

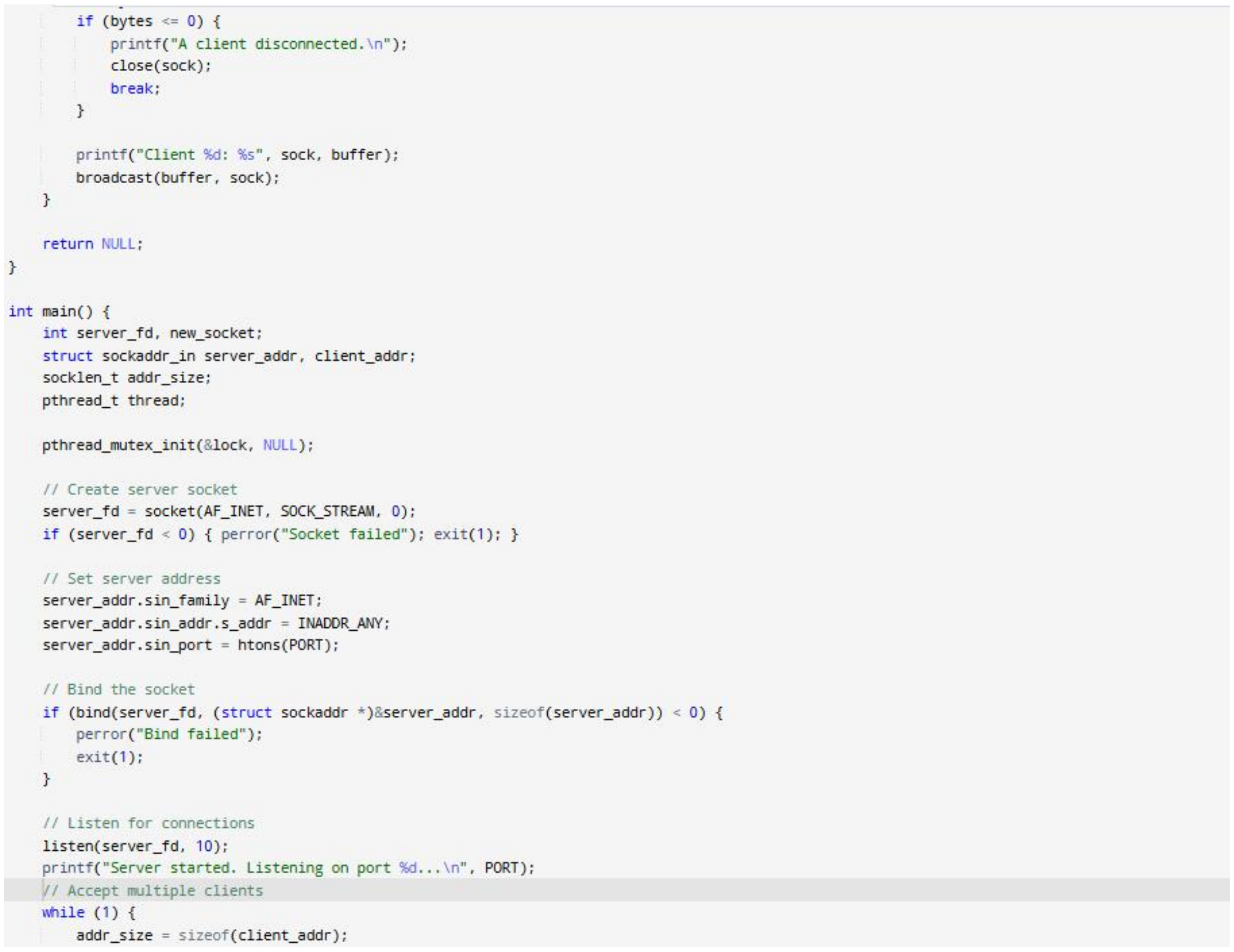
Compiler: GCC
Development Environment: Linux Terminal / VS Code / Any Linux-supported editor

10. CODE AND OUTPUT

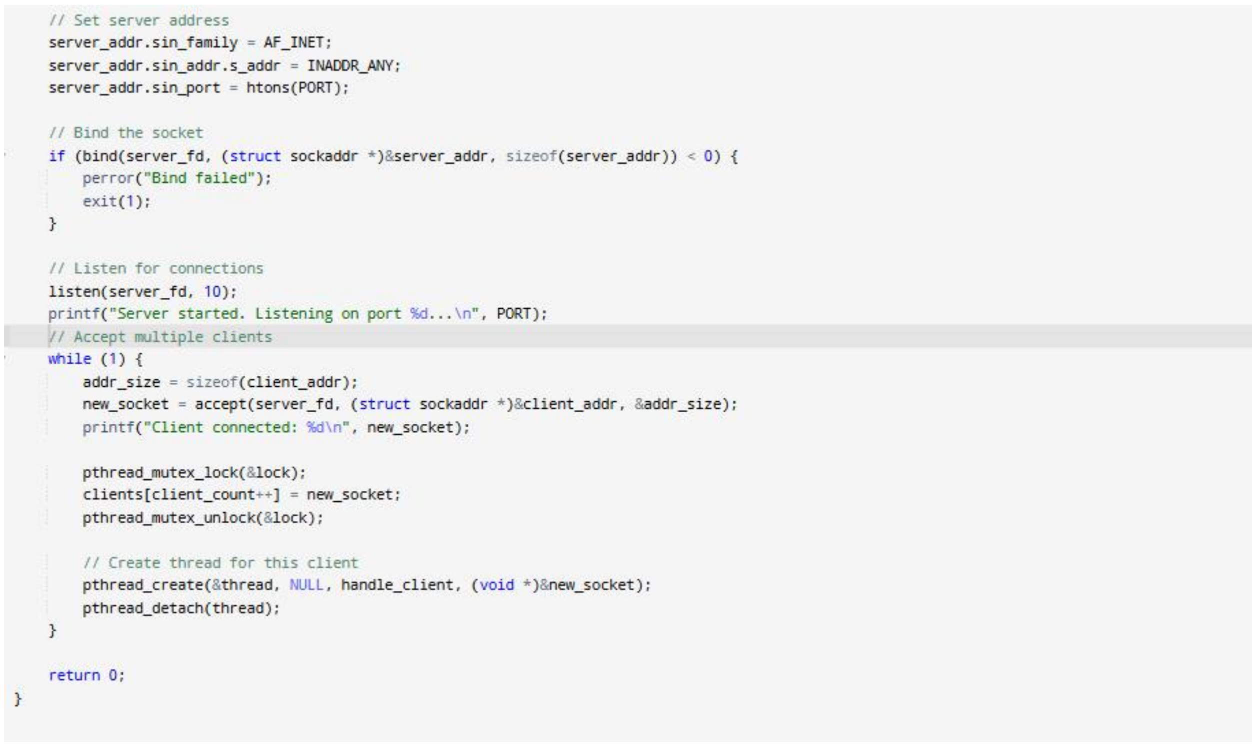
SERVER CODE



```c
// server.c — Multithreaded Chat Server in C (Linux)
// Compile: gcc server.c -o server -lpthread

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

#define PORT 8080
#define MAX_CLIENTS 100
#define BUFFER_SIZE 1024

int clients[MAX_CLIENTS];
int client_count = 0;
pthread_mutex_t lock;

// Broadcast a message to all clients except the sender
void broadcast(char *msg, int sender) {
    pthread_mutex_lock(&lock);

    for (int i = 0; i < client_count; i++) {
        if (clients[i] != sender) {
            send(clients[i], msg, strlen(msg), 0);
        }
    }

    pthread_mutex_unlock(&lock);
}

// Thread function to handle each client
void *handle_client(void *client_socket) {
    int sock = *((int *)client_socket);
    char buffer[BUFFER_SIZE];

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);

        int bytes = recv(sock, buffer, BUFFER_SIZE, 0);
        if (bytes <= 0) {
            printf("A client disconnected.\n");
            close(sock);
```



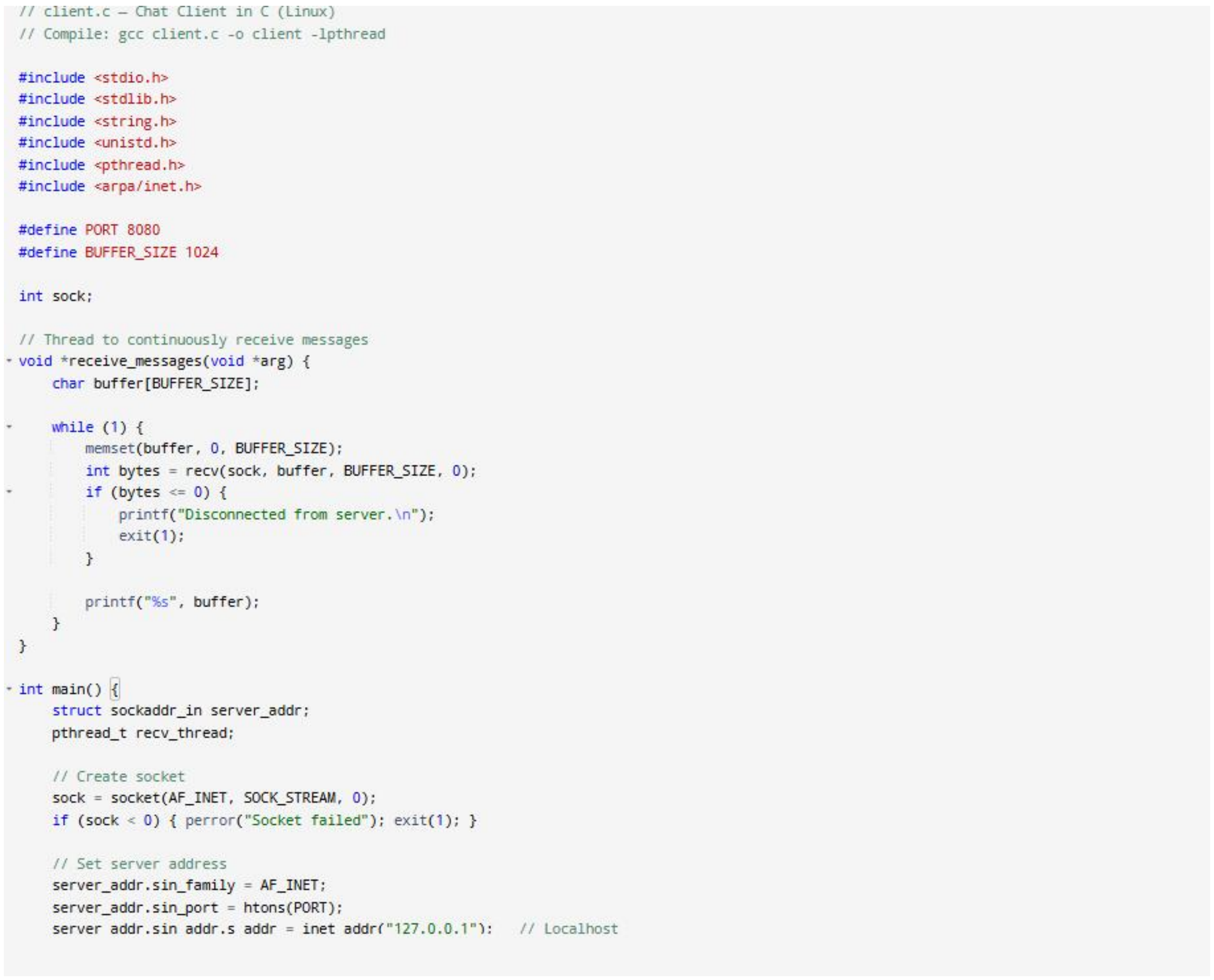
```c
        if (bytes <= 0) {
            printf("A client disconnected.\n");
            close(sock);
            break;
        }

        printf("Client %d: %s", sock, buffer);
        broadcast(buffer, sock);
    }

    return NULL;
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size;
    pthread_t thread;

    pthread_mutex_init(&lock, NULL);

    // Create server socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) { perror("Socket failed"); exit(1); }

    // Set server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

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

    // Listen for connections
    listen(server_fd, 10);
    printf("Server started. Listening on port %d...\n", PORT);
    // Accept multiple clients
    while (1) {
        addr_size = sizeof(client_addr);
```



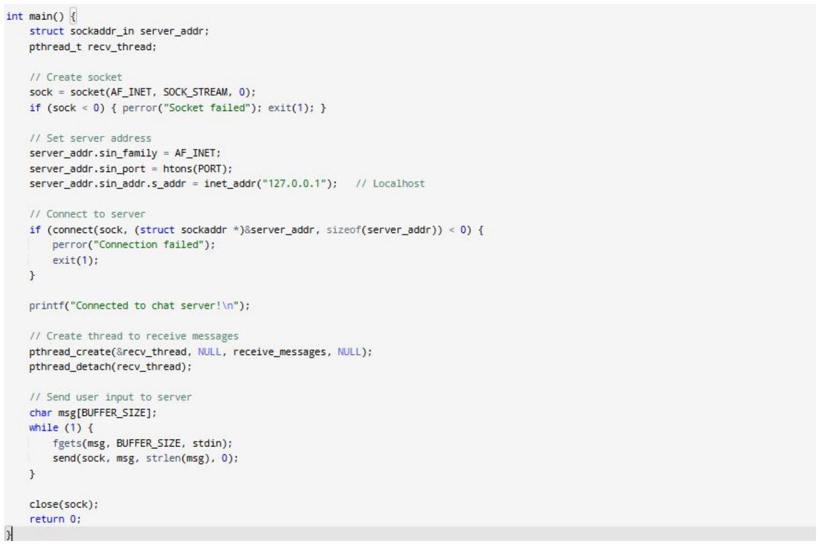
```c
    // Set server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

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

    // Listen for connections
    listen(server_fd, 10);
    printf("Server started. Listening on port %d...\n", PORT);
    // Accept multiple clients
    while (1) {
        addr_size = sizeof(client_addr);
```

```c
        new_socket = accept(server_fd, (struct sockaddr *)&client_addr, &addr_size);
        printf("Client connected: %d\n", new_socket);

        pthread_mutex_lock(&lock);
        clients[client_count++] = new_socket;
        pthread_mutex_unlock(&lock);

        // Create thread for this client
        pthread_create(&thread, NULL, handle_client, (void *)&new_socket);
        pthread_detach(thread);
    }

    return 0;
}
```

CLIENT CODE

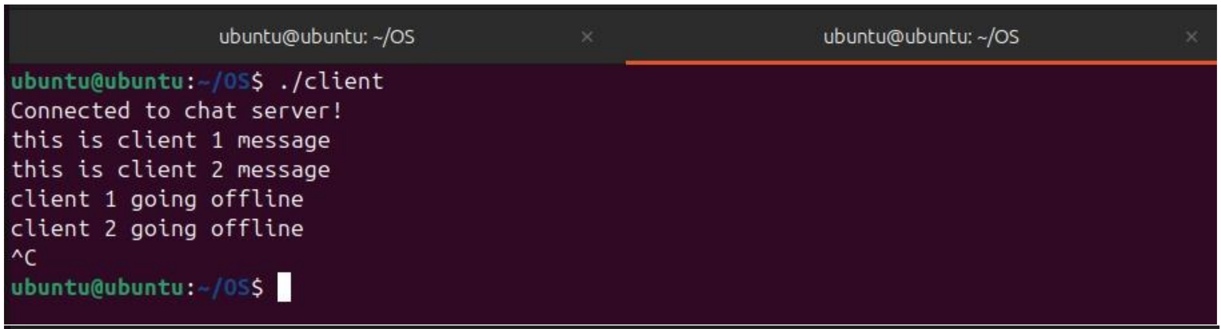


```c
// client.c — Chat Client in C (Linux)
// Compile: gcc client.c -o client -lpthread

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

#define PORT 8080
#define BUFFER_SIZE 1024

int sock;

// Thread to continuously receive messages
void *receive_messages(void *arg) {
    char buffer[BUFFER_SIZE];

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);
        int bytes = recv(sock, buffer, BUFFER_SIZE, 0);
        if (bytes <= 0) {
            printf("Disconnected from server.\n");
            exit(1);
        }

        printf("%s", buffer);
    }
}

int main() {
    struct sockaddr_in server_addr;
    pthread_t recv_thread;

    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) { perror("Socket failed"); exit(1); }

    // Set server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");   // Localhost
```

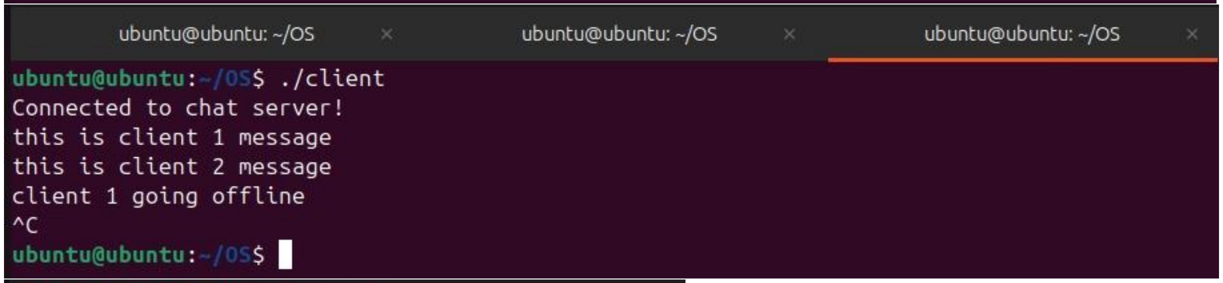


```c
int main() {
    struct sockaddr_in server_addr;
    pthread_t recv_thread;

    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) { perror("Socket failed"); exit(1); }

    // Set server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");   // Localhost
```

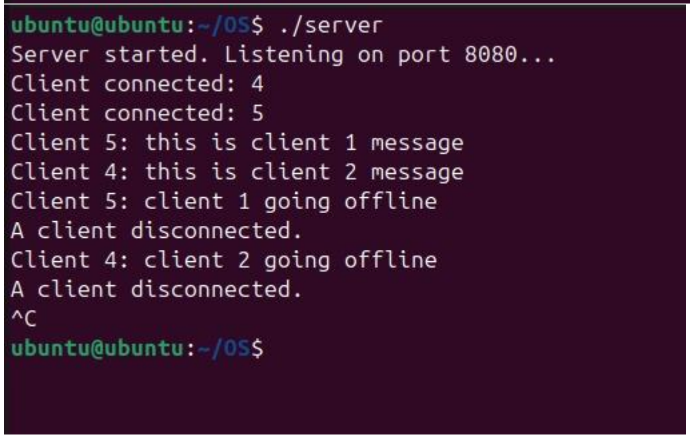
```c

    // Connect to server
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection failed");
        exit(1);
    }

    printf("Connected to chat server!\n");

    // Create thread to receive messages
    pthread_create(&recv_thread, NULL, receive_messages, NULL);
    pthread_detach(recv_thread);

    // Send user input to server
    char msg[BUFFER_SIZE];
    while (1) {
        fgets(msg, BUFFER_SIZE, stdin);
        send(sock, msg, strlen(msg), 0);
    }

    close(sock);
    return 0;
}
```



```
ubuntu@ubuntu:~/OS$ ./client
Connected to chat server!
this is client 1 message
this is client 2 message
client 1 going offline
client 2 going offline
^C
ubuntu@ubuntu:~/OS$
```



```
ubuntu@ubuntu:~/OS$ ./client
Connected to chat server!
this is client 1 message
this is client 2 message
client 1 going offline
^C
ubuntu@ubuntu:~/OS$
```



```
ubuntu@ubuntu:~/OS$ ./server
Server started. Listening on port 8080...
Client connected: 4
Client connected: 5
Client 5: this is client 1 message
Client 4: this is client 2 message
Client 5: client 1 going offline
A client disconnected.
Client 4: client 2 going offline
A client disconnected.
^C
ubuntu@ubuntu:~/OS$
```