**PROJECT: Multi-User Shared To-Do List Server-Client Application**

**Introduction**

In a multi-user environment, sharing tasks and ensuring proper task management can be challenging. This project showcases a system where multiple clients can connect to a server to get tasks and report their completion. The server assigns tasks from a shared list and logs completed tasks.

**Project Objective**

The primary objective of this project is to create a multi-client task management system that allows clients to request tasks from a server and report back when a task is completed. The server manages the task list and handles multiple clients simultaneously using multi-threading, ensuring efficient task distribution and tracking.

**System Design**

The system consists of two main components: the server and the client.

1. **Server:**
   * The server listens for incoming client connections on a specified port (8080).
   * It maintains a list of tasks that need to be completed.
   * The server handles multiple clients simultaneously using threads, where each thread manages a connected client.
   * When a client requests a task, the server assigns the next available task from the list.
   * The server also listens for notifications from clients about task completion and updates the task list accordingly.
2. **Client:**
   * The client connects to the server and can send two types of requests: "get\_task" to receive a task from the server and "complete\_task" to notify the server that a task has been completed.
   * The client interacts with the server through a simple command-line interface, allowing the user to request tasks and mark them as completed.

**Source code:**

// server.cpp

#include <iostream>

#include <string>

#include <vector>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

#include <pthread.h>

#define PORT 8080

#define MAX\_CLIENTS 3

std::vector<std::string> tasks = {"Design webpage", "Do backend", "Deploy website"};

std::vector<int> clients;

void\* handleClient(void\* arg) {

int clientSocket = ((int) arg);

std::string request;

char buffer[256];

while (true) {

read(clientSocket, buffer, 256);

request = buffer;

if (request == "get\_task") {

if (!tasks.empty()) {

std::string task = tasks.front();

tasks.erase(tasks.begin());

send(clientSocket, task.c\_str(), task.length(), 0);

std::cout << "Assigned task '" << task << "' to client " << clientSocket << std::endl;

} else {

send(clientSocket, "No tasks available", 17, 0);

}

} else if (request == "complete\_task") {

read(clientSocket, buffer, 256);

std::string task = buffer;

std::cout << "Task '" << task << "' completed by client " << clientSocket << std::endl;

} else {

std::cout << "Invalid request from client " << clientSocket << std::endl;

}

}

close(clientSocket);

return NULL;

}

int main() {

int serverSocket, clientSocket;

struct sockaddr\_in serverAddress, clientAddress;

socklen\_t clientLength = sizeof(clientAddress);

pthread\_t threads[MAX\_CLIENTS];

// Create server socket

serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (serverSocket < 0) {

std::cerr << "Error creating server socket" << std::endl;

return 1;

}

// Set server address

serverAddress.sin\_family = AF\_INET;

serverAddress.sin\_port = htons(PORT);

serverAddress.sin\_addr.s\_addr = INADDR\_ANY;

// Bind server socket to address

if (bind(serverSocket, (struct sockaddr\*) &serverAddress, sizeof(serverAddress)) < 0) {

std::cerr << "Error binding server socket" << std::endl;

return 1;

}

// Listen for clients

if (listen(serverSocket, MAX\_CLIENTS) < 0) {

std::cerr << "Error listening for clients" << std::endl;

return 1;

}

std::cout << "Server started. Waiting for clients..." << std::endl;

while (true) {

// Accept client connection

clientSocket = accept(serverSocket, (struct sockaddr\*) &clientAddress, &clientLength);

if (clientSocket < 0) {

std::cerr << "Error accepting client connection" << std::endl;

continue;

}

// Create new thread to handle client

pthread\_create(&threads[clients.size()], NULL, handleClient, &clientSocket);

clients.push\_back(clientSocket);

}

return 0;

}

// client.cpp

#include <iostream>

#include <string>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#define PORT 8080

int main() {

int clientSocket;

struct sockaddr\_in serverAddress;

// Create client socket

clientSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (clientSocket < 0) {

std::cerr << "Error creating client socket" << std::endl;

return 1;

}

// Set server address

serverAddress.sin\_family = AF\_INET;

serverAddress.sin\_port = htons(PORT);

inet\_pton(AF\_INET, "127.0.0.1", &serverAddress.sin\_addr);

// Connect to server

if (connect(clientSocket, (struct sockaddr\*) &serverAddress, sizeof(serverAddress)) < 0) {

std::cerr << "Error connecting to server" << std::endl;

return 1;

}

std::cout << "Connected to server. Type 'get\_task' to receive a task or 'complete\_task' to complete a task." << std::endl;

while (true) {

std::string request;

std::cout << "Enter request: ";

std::cin >> request;

if (request == "get\_task") {

send(clientSocket, "get\_task", 8, 0);

char buffer[256];

read(clientSocket, buffer, 256);

std::cout << "Received task: " << buffer << std::endl;

} else if (request == "complete\_task") {

std::string task;

std::cout << "Enter task to complete: ";

std::cin >> task;

send(clientSocket, "complete\_task", 13, 0);

send(clientSocket, task.c\_str(), task.length(), 0);

std::cout << "Task completed successfully." << std::endl;

} else {

std::cout << "Invalid request. Please try again." << std::endl;

}

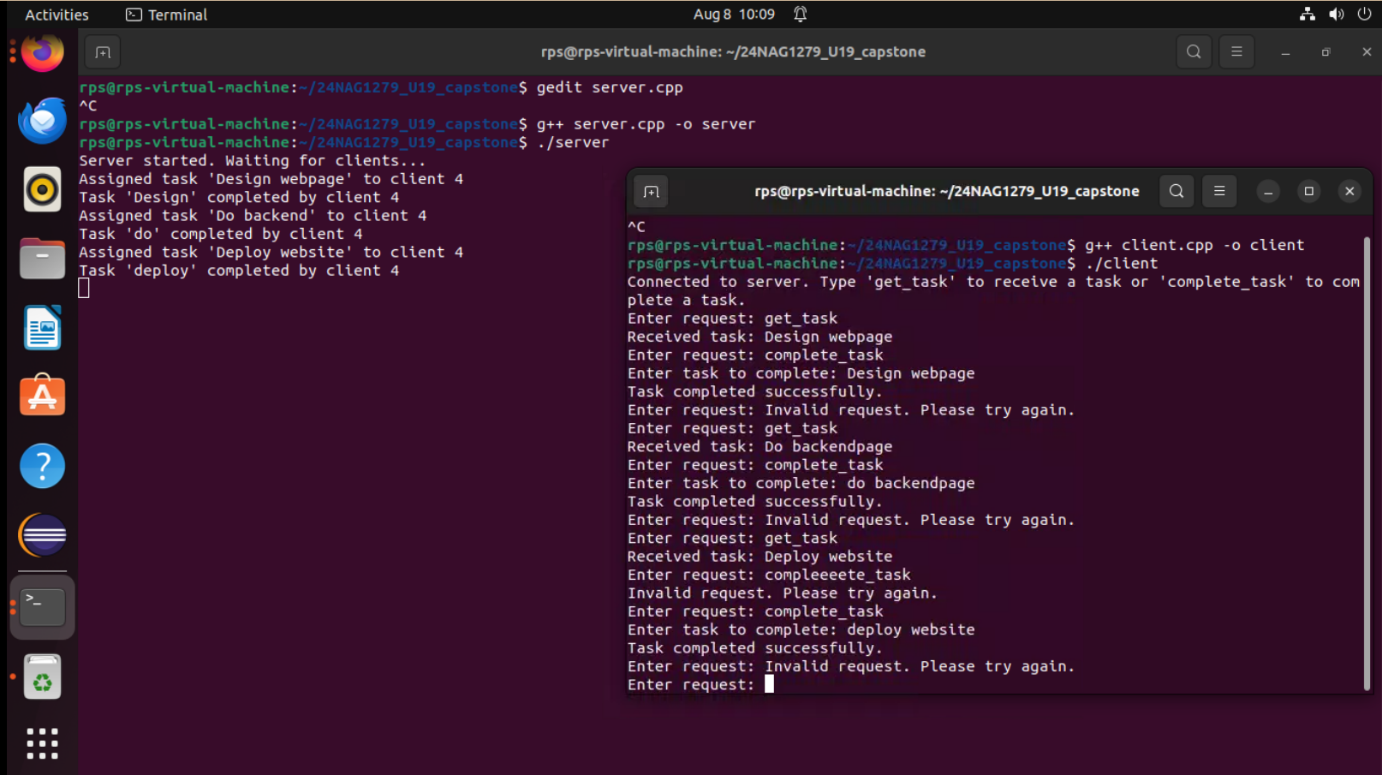
}

close(clientSocket);

return 0;

}

**Output:**

****

**Future Enhancement**

Several potential enhancements could be made to improve and expand the functionality of this system:

1. **Persistent Storage:** Implementing persistent storage for the task list, such as a database or file system, would allow the server to save and reload tasks across restarts.
2. **Task Prioritization:** Introducing priority levels for tasks would enable the server to assign more critical tasks before others, improving task management.
3. **Task Assignment History:** Tracking which client completed which task and maintaining a log would allow for better monitoring and auditing.
4. **Scalability:** Enhancing the server to handle a larger number of clients efficiently and distribute tasks across multiple servers would make the system more scalable.
5. **Security:** Adding authentication and encryption mechanisms to secure the communication between clients and the server would make the system more robust against unauthorized access.
6. **GUI Interface:** Developing a graphical user interface (GUI) for both the server and client would improve usability and make the system more user-friendly.

**Conclusion**

This project demonstrates the basics of a multi-client task management system using a client-server architecture. It highlights the importance of synchronization and multi-threading in handling multiple clients simultaneously. The system effectively distributes tasks to clients and manages task completion notifications. With further enhancements, this basic model could evolve into a more comprehensive and scalable task management solution for real-world applications.

USER ID -24NAG1279\_U19