Design and implement a network service that reliably handles concurrent client connections while ensuring graceful termination in response to external signals (e.g., SIGTERM, SIGINT). The service must maintain data consistency and avoid resource leaks throughout its lifecycle.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <arpa/inet.h>

#include <pthread.h>

#include <csignal>

#include <vector>

#include <mutex>

#include <algorithm>

std::vector<int> clients;

std::mutex clients\_mutex;

int server\_socket;

bool shutdown\_requested = false;

void\* handle\_client(void\* arg) {

int client\_socket = \*(int\*)arg;

delete (int\*)arg;

char buffer[1024];

while (true) {

ssize\_t bytes\_received = recv(client\_socket, buffer, sizeof(buffer) - 1, 0);

if (bytes\_received <= 0) break;

buffer[bytes\_received] = '\0';

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

if (send(client\_socket, buffer, bytes\_received, 0) == -1) {

std::cerr << "Failed to send data to client." << std::endl;

break;

}

}

std::cout << "Closing connection." << std::endl;

close(client\_socket);

std::lock\_guard<std::mutex> lock(clients\_mutex);

clients.erase(std::remove(clients.begin(), clients.end(), client\_socket), clients.end());

return nullptr;

}

void signal\_handler(int signum) {

std::cout << "Signal " << signum << " received, shutting down." << std::endl;

shutdown\_requested = true;

close(server\_socket);

}

int main() {

struct sockaddr\_in server\_addr;

struct sockaddr\_in client\_addr;

socklen\_t client\_len = sizeof(client\_addr);

signal(SIGTERM, signal\_handler);

signal(SIGINT, signal\_handler);

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

if (server\_socket == -1) {

std::cerr << "Failed to create socket." << std::endl;

return EXIT\_FAILURE;

}

server\_addr.sin\_family = AF\_INET;

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

server\_addr.sin\_port = htons(12345);

if (bind(server\_socket, (struct sockaddr\*)&server\_addr, sizeof(server\_addr)) == -1) {

std::cerr << "Failed to bind socket." << std::endl;

close(server\_socket);

return EXIT\_FAILURE;

}

if (listen(server\_socket, 5) == -1) {

std::cerr << "Failed to listen on socket." << std::endl;

close(server\_socket);

return EXIT\_FAILURE;

}

std::cout << "Server started on port 12345" << std::endl;

while (!shutdown\_requested) {

int\* client\_socket = new int;

\*client\_socket = accept(server\_socket, (struct sockaddr\*)&client\_addr, &client\_len);

if (\*client\_socket == -1) {

if (shutdown\_requested) break;

std::cerr << "Failed to accept connection." << std::endl;

delete client\_socket;

continue;

}

std::cout << "Accepted connection from " << inet\_ntoa(client\_addr.sin\_addr) << std::endl;

pthread\_t client\_thread;

if (pthread\_create(&client\_thread, nullptr, handle\_client, client\_socket) != 0) {

std::cerr << "Failed to create thread." << std::endl;

close(\*client\_socket);

delete client\_socket;

continue;

}

{

std::lock\_guard<std::mutex> lock(clients\_mutex);

clients.push\_back(\*client\_socket);

}

pthread\_detach(client\_thread);

}

{

std::lock\_guard<std::mutex> lock(clients\_mutex);

for (int client\_socket : clients) {

close(client\_socket);

}

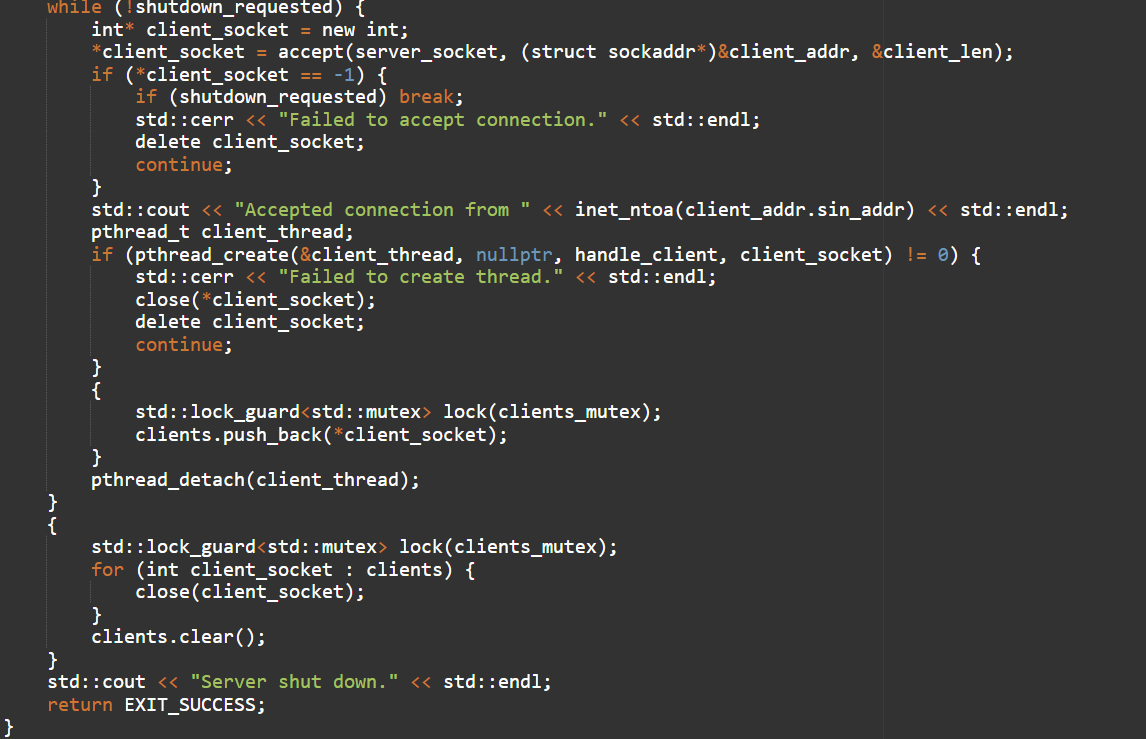
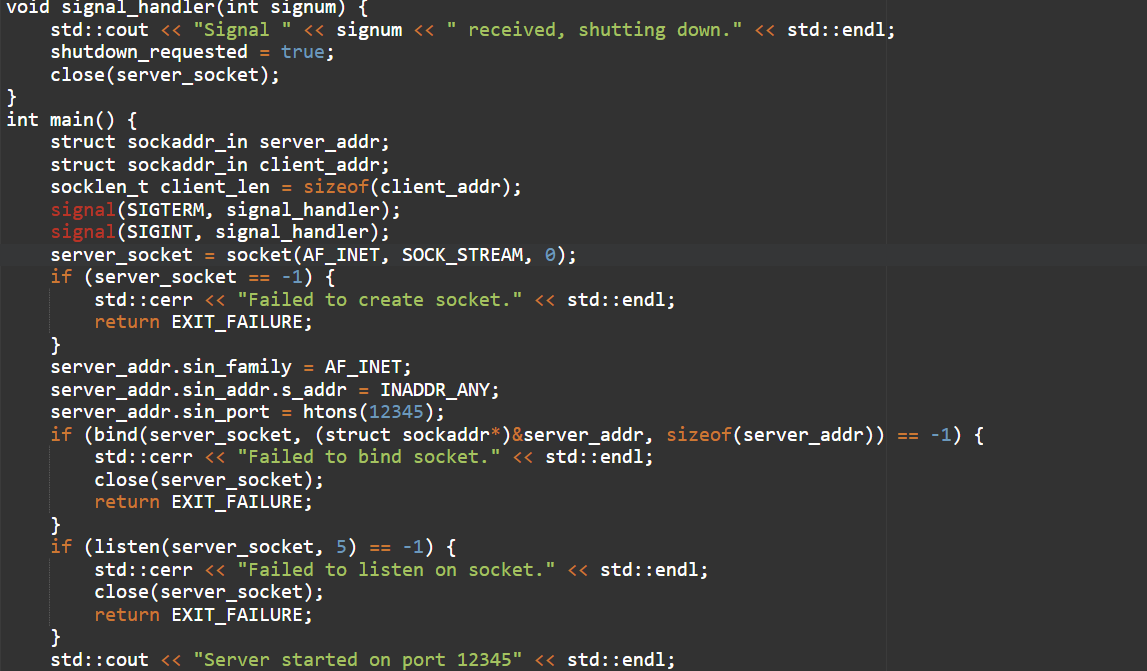
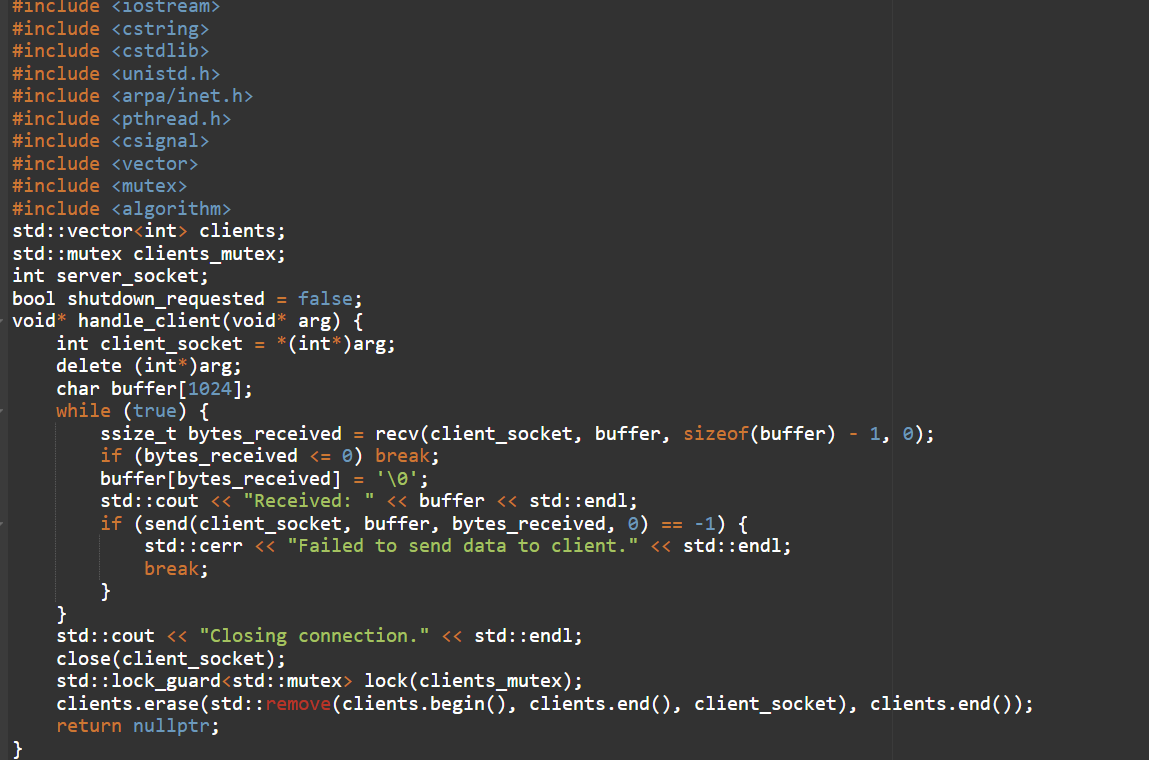
clients.clear();

}

std::cout << "Server shut down." << std::endl;

return EXIT\_SUCCESS;

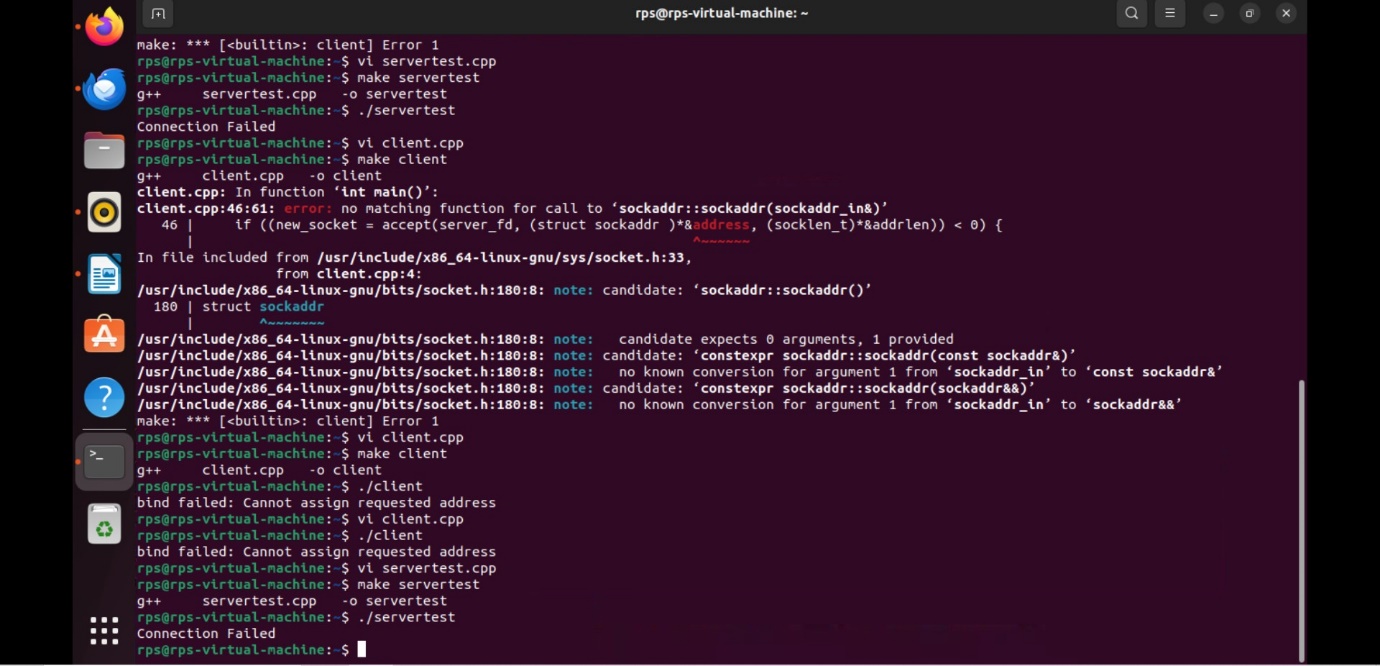
}



Example 1:



Example 2:



Example 3:

