**МІНІСТЕРСТВО НАУКИ ТА ОСВІТИ УКРАЇНИ**

**ОДЕСЬКА НАЦІОНАЛЬНА АКАДЕМІЯ ЗВ'ЯЗКУ ІМ. О. С. ПОПОВА**

**ЗВІТ**

**до лабораторного завдання №4**

**з дисципліни «Мережеве програмування»**

Виконав

студент групи ІПЗ-3.02

Писарев С. Д.

Перевірив

Викладач Вороний С. М.

Задание

Разработайте серверную часть программы из лабораторной работы No3 в двух вариантах так, чтобы в одном из них работа с сокетами была реализована по модели WSAAsyncSelect, а в другом – по модели WSAEventSelect.

**Вариант №7**

Клиентская программа оправляет на сервер текстовый файл из N строк (путь к файлу задается через пользовательский интерфейс клиента). Сервер подсчитывает общее количество слов длиной не менее пяти символов в строках файла. Это количество возвращает клиенту.

Листинг

Server.cpp

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#include <iostream>

#include <fstream>

#include <thread>

#include <WinSock2.h>

#include <ws2tcpip.h>

#include "Server.h"

#include "Helpers.h"

#pragma comment(lib, "Ws2\_32.lib")

using namespace std;

constexpr auto DEFAULT\_BUFLEN = 512;

const string filename = "fileToProcess.txt";

SOCKET ListenSocket = INVALID\_SOCKET;

SOCKET ClientSocket = INVALID\_SOCKET;

constexpr auto TIMEOUT\_SEC = 150;

constexpr auto TIMEOUT\_MSEC = 0;

#define WM\_SOCKET (WM\_USER + 1)

typedef struct \_SOCKET\_INFORMATION{

    CHAR Buffer[DEFAULT\_BUFLEN];

    WSABUF DataBuf;

    SOCKET Socket;

    DWORD BytesSEND;

    DWORD BytesRECV;

} SOCKET\_INFORMATION, \* LPSOCKET\_INFORMATION;

WSAEVENT EventArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

LPSOCKET\_INFORMATION SocketArray[WSA\_MAXIMUM\_WAIT\_EVENTS];

BOOL CreateSocketInformation(SOCKET s);

void FreeSocketInformation(DWORD Event);

DWORD EventTotal = 0;

int bytes\_read;

char buf[1024];

int fileSize = 0;

void OutputClientState(SOCKET\* socket, bool isConnected)

{

    int foreground\_color, background\_color;

    string state;

    struct sockaddr\_in sin;

    socklen\_t len = sizeof(sin);

    getsockname(\*socket, (struct sockaddr\*)&sin, &len);

    if (isConnected)

    {

        foreground\_color = FOREGROUND\_GREEN;

        background\_color = BACKGROUND\_GREEN;

        state = "connected";

    }

    else

    {

        foreground\_color = FOREGROUND\_RED;

        background\_color = BACKGROUND\_RED;

        state = "disconnected";

    }

    HANDLE hConsoleOutput = GetStdHandle(STD\_OUTPUT\_HANDLE);

    SetConsoleTextAttribute(hConsoleOutput, foreground\_color);

    cout << "The client with an IP address ";

    SetConsoleTextAttribute(hConsoleOutput, background\_color);

    cout << inet\_ntoa(sin.sin\_addr);

    SetConsoleTextAttribute(hConsoleOutput, foreground\_color);

    cout << " was " << state << " via " << htons(sin.sin\_port) << " port." << endl;

    SetConsoleTextAttribute(hConsoleOutput, FOREGROUND\_BLUE | FOREGROUND\_GREEN | FOREGROUND\_RED);

}

Server::Server(string& serverPort)

{

    MSG msg;

    DWORD Ret;

    SOCKADDR \_INInternetAddr;

    HWND Window;

    WSADATA wsaData;

    int iResult;

    // Initialize Winsock

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0) {

        printf("WSAStartup failed: %d\n", iResult);

        return;

    }

    SOCKET ClientSocket = INVALID\_SOCKET;

    struct addrinfo\* result = NULL, \* ptr = NULL, hints;

    ZeroMemory(&hints, sizeof(hints));

    hints.ai\_family = AF\_INET;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_protocol = IPPROTO\_TCP;

    hints.ai\_flags = AI\_PASSIVE;

    // Resolve the local address and port to be used by the server

    iResult = getaddrinfo(NULL, serverPort.c\_str(), &hints, &result);

    if (iResult != 0) {

        printf("getaddrinfo failed: %d\n", iResult);

        WSACleanup();

        return;

    }

    // Create a SOCKET for the server to listen for client connections

    ListenSocket = socket(result->ai\_family, result->ai\_socktype, result->ai\_protocol);

    if (ListenSocket == INVALID\_SOCKET) {

        printf("Error at socket(): %ld\n", WSAGetLastError());

        freeaddrinfo(result);

        WSACleanup();

        return;

    }

    if (CreateSocketInformation(ListenSocket) == FALSE)

        printf("CreateSocketInformation() failed!\n");

    else

        printf("CreateSocketInformation() is OK lol!\n");

    if (WSAEventSelect(ListenSocket, EventArray[EventTotal - 1], FD\_ACCEPT | FD\_CLOSE) == SOCKET\_ERROR)

    {

        printf("WSAEventSelect() failed with error %d\n", WSAGetLastError());

        return;

    }

    else

        printf("WSAEventSelect() is pretty fine!\n");

    /\*InternetAddr.sin\_family = AF\_INET;

    InternetAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

    InternetAddr.sin\_port = htons(PORT);\*/

    // Setup the TCP listening socket

    // Without a double colon behind the bind function it won't work. It means that it will use the bind func from the global namespace.

    iResult = ::bind(ListenSocket, result->ai\_addr, (int)result->ai\_addrlen);

    if (iResult == SOCKET\_ERROR) {

        printf("bind failed with error: %d\n", WSAGetLastError());

        freeaddrinfo(result);

        closesocket(ListenSocket);

        WSACleanup();

        return;

    }

    //freeaddrinfo(result);

    auto CtrlHandler = [](DWORD fdwCtrlType)

    {

        SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

        closesocket(ListenSocket);

        WSACleanup();

        return FALSE;

    };

    SetConsoleCtrlHandler(CtrlHandler, true);

    if (listen(ListenSocket, SOMAXCONN) == SOCKET\_ERROR) {

        printf("Listen failed with error: %ld\n", WSAGetLastError());

        closesocket(ListenSocket);

        WSACleanup();

        return;

    }

    DWORD Event;

    WSANETWORKEVENTS NetworkEvents;

    SOCKET Accept;

    LPSOCKET\_INFORMATION SocketInfo;

    DWORD Flags;

    DWORD RecvBytes;

    DWORD SendBytes;

    while (TRUE)

    {

        cout << "\n";

        // Wait for one of the sockets to receive I/O notification and

        if ((Event = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA\_INFINITE, FALSE)) == WSA\_WAIT\_FAILED)

        {

            printf("WSAWaitForMultipleEvents() failed with error %d\n", WSAGetLastError());

            return;

        }

        else

            printf("WSAWaitForMultipleEvents() is pretty damn OK!\n");

        if (WSAEnumNetworkEvents(SocketArray[Event - WSA\_WAIT\_EVENT\_0]->Socket,

            EventArray[Event - WSA\_WAIT\_EVENT\_0], &NetworkEvents) == SOCKET\_ERROR)

        {

            printf("WSAEnumNetworkEvents() failed with error %d\n", WSAGetLastError());

            return;

        }

        else

            printf("WSAEnumNetworkEvents() should be fine!\n");

        if (NetworkEvents.lNetworkEvents & FD\_ACCEPT)

        {

            if (NetworkEvents.iErrorCode[FD\_ACCEPT\_BIT] != 0)

            {

                printf("FD\_ACCEPT failed with error %d\n",

                    NetworkEvents.iErrorCode[FD\_ACCEPT\_BIT]);

                break;

            }

            if ((Accept = accept(SocketArray[Event - WSA\_WAIT\_EVENT\_0] -> Socket, NULL, NULL)) == INVALID\_SOCKET)

            {

                printf("accept() failed with error %d\n", WSAGetLastError());

                break;

            }

            else

                printf("accept() should be OK!\n");

            if (EventTotal > WSA\_MAXIMUM\_WAIT\_EVENTS)

            {

                printf("Too many connections - closing socket...\n");

                closesocket(Accept);

                break;

            }

            CreateSocketInformation(Accept);

            if (WSAEventSelect(Accept, EventArray[EventTotal - 1], FD\_READ | FD\_WRITE | FD\_CLOSE) == SOCKET\_ERROR)

            {

                printf("WSAEventSelect() failed with error %d\n", WSAGetLastError());

                return;

            }

            else

                printf("WSAEventSelect() is OK!\n");

            printf("Socket %d got connected...\n", (int)Accept);

        }

        // Try to read and write data to and from the data buffer if read and write events occur

        if (NetworkEvents.lNetworkEvents & FD\_READ || NetworkEvents.lNetworkEvents & FD\_WRITE)

        {

            if (NetworkEvents.lNetworkEvents & FD\_READ && NetworkEvents.iErrorCode[FD\_READ\_BIT] != 0)

            {

                printf("FD\_READ failed with error %d\n", NetworkEvents.iErrorCode[FD\_READ\_BIT]);

                break;

            }

            else

                printf("FD\_READ is OK!\n");

            if (NetworkEvents.lNetworkEvents & FD\_WRITE && NetworkEvents.iErrorCode[FD\_WRITE\_BIT] != 0)

            {

                printf("FD\_WRITE failed with error %d\n",

                    NetworkEvents.iErrorCode[FD\_WRITE\_BIT]);

                break;

            }

            else

                printf("FD\_WRITE is OK!\n");

            SocketInfo = SocketArray[Event - WSA\_WAIT\_EVENT\_0];

            // Read data only if the receive buffer is empty

            if (SocketInfo->BytesRECV == 0)

            {

                SocketInfo->DataBuf.buf = SocketInfo->Buffer;

                SocketInfo->DataBuf.len = DEFAULT\_BUFLEN;

                Flags = 0;

                if (WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &RecvBytes, &Flags, NULL, NULL) == SOCKET\_ERROR)

                {

                    if (WSAGetLastError() != WSAEWOULDBLOCK)

                    {

                        printf("WSARecv() failed with error %d\n", WSAGetLastError());

                        FreeSocketInformation(Event - WSA\_WAIT\_EVENT\_0);

                        return;

                    }

                }

                else

                {

                    printf("WSARecv() is working!\n");

                    cout << "bytes\_read = " << RecvBytes << endl;

                    SocketInfo->BytesRECV = RecvBytes;

                    // ���������� ������ ������� �������

                    // Receive until the peer shuts down the connection.

                    string actualData(SocketInfo->DataBuf.buf, SocketInfo->DataBuf.buf + SocketInfo->BytesRECV);

                    if (fileSize == 0)

                    {

                        fileSize = stoi(actualData);

                        SocketInfo->BytesRECV = 0;

                        continue;

                    }

                    string fileData;

                    fileData.append(actualData);

                    fileSize -= RecvBytes;

                    if (fileSize == 0)

                    {

                        // A file receiving is complete.

                        // Writes file contents in binary to a new file.

                        fstream myfile(filename, ios::binary | ios::out | ios::trunc);

                        myfile.write(fileData.c\_str(), fileData.length());

                        myfile.close();

                        myfile.open(filename, ios::in);

                        string word;

                        int numOfWordsGTEQFiveLetters = 0;

                        // Reads the file word by word.

                        while (myfile >> word)

                        {

                            if (word.length() >= 5)

                            {

                                numOfWordsGTEQFiveLetters++;

                            }

                        }

                        string valueToSend = to\_string(numOfWordsGTEQFiveLetters);

                        SocketInfo->DataBuf.buf = (char\*)valueToSend.c\_str();

                        SocketInfo->DataBuf.len = valueToSend.length();

                        SendBytes = valueToSend.length();

                        if (WSASend(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &SendBytes, 0, NULL, NULL) == SOCKET\_ERROR)

                        {

                            if (WSAGetLastError() != WSAEWOULDBLOCK)

                            {

                                printf("WSASend() failed with error %d\n", WSAGetLastError());

                                FreeSocketInformation(Event - WSA\_WAIT\_EVENT\_0);

                                return;

                            }

                            // A WSAEWOULDBLOCK error has occurred. An FD\_WRITE event will be posted

                            // when more buffer space becomes available

                        }

                        else

                        {

                            printf("WSASend() is fine! Thank you...\n");

                            SocketInfo->BytesRECV = 0;

                        }

                        myfile.close();

                        fileData.clear();

                    }

                }

            }

        }

        if (NetworkEvents.lNetworkEvents & FD\_CLOSE)

        {

            if (NetworkEvents.iErrorCode[FD\_CLOSE\_BIT] != 0)

            {

                printf("FD\_CLOSE failed with error %d\n",

                    NetworkEvents.iErrorCode[FD\_CLOSE\_BIT]);

                break;

            }

            else

                printf("FD\_CLOSE is OK!\n");

            printf("Closing socket information %d\n", SocketArray[Event - WSA\_WAIT\_EVENT\_0]->Socket);

            FreeSocketInformation(Event - WSA\_WAIT\_EVENT\_0);

        }

    }

    return;

}

BOOL CreateSocketInformation(SOCKET s)

{

    LPSOCKET\_INFORMATION SI;

    if ((EventArray[EventTotal] = WSACreateEvent()) == WSA\_INVALID\_EVENT)

    {

        printf("WSACreateEvent() failed with error %d\n", WSAGetLastError());

        return FALSE;

    }

    else

        printf("WSACreateEvent() is OK!\n");

    if ((SI = (LPSOCKET\_INFORMATION)GlobalAlloc(GPTR, sizeof(SOCKET\_INFORMATION)))

        == NULL)

    {

        printf("GlobalAlloc() failed with error %d\n", GetLastError());

        return FALSE;

    }

    else

        printf("GlobalAlloc() for LPSOCKET\_INFORMATION is OK!\n");

    // PrepareSocketInfo structure for use

    SI->Socket = s;

    SI->BytesSEND = 0;

    SI->BytesRECV = 0;

    SocketArray[EventTotal] = SI;

    EventTotal++;

    return(TRUE);

}

void FreeSocketInformation(DWORD Event)

{

    LPSOCKET\_INFORMATION SI = SocketArray[Event];

    DWORD i;

    closesocket(SI->Socket);

    GlobalFree(SI);

    if (WSACloseEvent(EventArray[Event]) == TRUE)

        printf("WSACloseEvent() is OK!\n\n");

    else

        printf("WSACloseEvent() failed miserably!\n\n");

    // Squash the socket and event arrays

    for (i = Event; i < EventTotal; i++)

    {

        EventArray[i] = EventArray[i + 1];

        SocketArray[i] = SocketArray[i + 1];

    }

    EventTotal--;

}

ClientServer.cpp

#undef UNICODE

#define WIN32\_LEAN\_AND\_MEAN

#include <iostream>

#include <stdio.h>

#include <algorithm>

#include <iterator>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <boost/program\_options.hpp>

#include "Client.h"

#include "Server.h"

#include "ClientServer.h"

#include "Helpers.h"

#pragma comment(lib, "Ws2\_32.lib")

#pragma comment (lib, "Mswsock.lib")

#pragma comment (lib, "AdvApi32.lib")

using namespace std;

namespace po = boost::program\_options;

int main(int argc, char\* argv[])

{

    string PORT, IP\_ADDRESS;

    try {

        po::options\_description desc("Options");

        desc.add\_options()

            ("help,h", "Produce a help message.")

            ("server,S", "The program will start as a server.")

            ("client,C", "The program will start as a client.")

            ("ip,I", po::value< string >(&IP\_ADDRESS), "Provides an IP address.")

            ("port,P", po::value< string >(&PORT), "Provides a port.")

            ;

        po::variables\_map vm;

        po::store(po::parse\_command\_line(argc, argv, desc), vm);

        po::notify(vm);

        if (vm.count("help")) {

            cout << desc << "\n";

            return 0;

        }

        cout << "IP: " << IP\_ADDRESS << endl

             << "Port: " << PORT << endl << endl;

        //SetConsoleCtrlHandler(CtrlHandler, true);

        if (vm.count("server")) {

            cout << "The program was started as a server." << "\n";

            new Server(PORT);

        }

        else if (vm.count("client"))

        {

            cout << "The program was started as a client." << "\n";

            new Client(PORT, IP\_ADDRESS);

        }

    }

    catch (std::exception& e) {

        cerr << "error: " << e.what() << "\n";

        return 1;

    }

    catch (...) {

        cerr << "Exception of unknown type!\n";

    }

    return 0;

}

Client.cpp

#undef UNICODE

#define WIN32\_LEAN\_AND\_MEAN

#include <iostream>

#include <fstream>

#include <WinSock2.h>

#include <ws2tcpip.h>

#include <thread>

#include "Client.h"

#include "Helpers.h"

#pragma comment(lib, "Ws2\_32.lib")

#pragma comment (lib, "Mswsock.lib")

#pragma comment (lib, "AdvApi32.lib")

using namespace std;

constexpr auto DEFAULT\_BUFLEN = 512;

Client::Client(string& PORT, string& IP\_ADDRESS)

{

    WSADATA wsaData;

    int iResult;

    SOCKET ConnectSocket = INVALID\_SOCKET;

    struct addrinfo\* result = NULL,

                   \* ptr = NULL,

                     hints;

    // Initializes Winsock.

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    if (iResult != 0)

    {

        printf("WSAStartup failed with error: %d\n", iResult);

        return;

    }

    ZeroMemory(&hints, sizeof(hints));

    hints.ai\_family = AF\_UNSPEC;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_protocol = IPPROTO\_TCP;

    // Resolves a server address and port.

    iResult = getaddrinfo(IP\_ADDRESS.c\_str(), PORT.c\_str(), &hints, &result);

    if (iResult != 0)

    {

        printf("getaddrinfo failed with error: %d\n", iResult);

        WSACleanup();

        return;

    }

    // Attempt to connect to an address until one succeeds

    for (ptr = result; ptr != NULL; ptr = ptr->ai\_next)

    {

        // Creates a SOCKET for connecting to a server.

        ConnectSocket = socket(ptr->ai\_family,

                               ptr->ai\_socktype,

                               ptr->ai\_protocol);

        if (ConnectSocket == INVALID\_SOCKET)

        {

            printf("socket failed with error: %ld\n", WSAGetLastError());

            WSACleanup();

            return;

        }

        // Connects to a server.

        iResult = connect(ConnectSocket, ptr->ai\_addr, (int)ptr->ai\_addrlen);

        if (iResult == SOCKET\_ERROR)

        {

            closesocket(ConnectSocket);

            ConnectSocket = INVALID\_SOCKET;

            continue;

        }

        break;

    }

    freeaddrinfo(result);

    if (ConnectSocket == INVALID\_SOCKET)

    {

        printf("Unable to connect to server!\n");

        WSACleanup();

        return;

    }

    string input;

    char recvbuf[DEFAULT\_BUFLEN];

    int recvbuflen = DEFAULT\_BUFLEN;

    while (true)

    {

        string path;

        cout << "Type the path to a file." << endl;

        cin >> path;

        // Reads contents of the file.

        ifstream in(path);

        string contents((istreambuf\_iterator<char>(in)),

            istreambuf\_iterator<char>());

        // Sends the size of the file contents and then the contents itself.

        string contentsSize = to\_string(contents.length());

        send(ConnectSocket, contentsSize.c\_str(), contentsSize.length(), 0);

        iResult = send(ConnectSocket, contents.c\_str(), contents.length(), 0);

        if (iResult == SOCKET\_ERROR)

        {

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED);

            printf("The server was shutdown.");

            CorrectClosing(ConnectSocket);

            return;

        }

        else

        {

            // Receives the answer from a server.

            iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);

            string actualData(recvbuf, recvbuf + iResult);

            // Nicely displays the answer.

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_BLUE);

            cout << "The num of words 5 letters or more in the file: ";

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), BACKGROUND\_BLUE);

            cout << actualData << endl << endl;

            SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

        }

    }

}

Helpers.cpp

#include <vector>

#include <iterator>

#include <string>

#include <ws2tcpip.h>

using namespace std;

struct SocketData

{

    SOCKET socket;

    string ip;

    string port;

};

// A helper function to simplify the main part.

template<class T>

ostream& operator<<(ostream& os, const vector<T>& v)

{

    copy(v.begin(), v.end(), ostream\_iterator<T>(os, " "));

    return os;

}

void CorrectClosing(SOCKET socket)

{

    // Sets the console attributes to default values.

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);

    closesocket(socket);

    // Should be called at the end of the program and be paired with WSAStartup.

    WSACleanup();

}

void getIP(sockaddr socketInfo, char\* ip)

{

    sockaddr\_in\* sin = reinterpret\_cast<sockaddr\_in\*>(&socketInfo);

    inet\_ntop(AF\_INET, &sin->sin\_addr, ip, INET\_ADDRSTRLEN);

}

Скриншоты выполнения программы



