#include "stdafx.h"

#include "Winsock2.h"

#include "ErrorFunctions.h"

#include <string>

#include <list>

#include <time.h>

#include <iostream>

#define AS\_SQ 10

#define IP\_SERVER "192.168.134.8"

using namespace std;

SOCKET sS;

int serverPort;

char dllName[50];

char namedPipeName[10];

volatile long connectionCount = 0;

volatile long sayNoCount = 0;

volatile long successConnections = 0;

volatile long currentActiveConnections = 0;

HANDLE hAcceptServer, hConsolePipe, hGarbageCleaner, hDispatchServer, hResponseServer;

HANDLE hClientConnectedEvent = CreateEvent(NULL,

FALSE,

FALSE,

L"ClientConnected");;

DWORD WINAPI AcceptServer(LPVOID pPrm);

DWORD WINAPI ConsolePipe(LPVOID pPrm);

DWORD WINAPI GarbageCleaner(LPVOID pPrm);

DWORD WINAPI DispatchServer(LPVOID pPrm);

DWORD WINAPI ResponseServer(LPVOID pPrm);

CRITICAL\_SECTION scListContact;

enum TalkersCommand {

START, STOP, EXIT, STATISTICS, WAIT, SHUTDOWN, GETCOMMAND, LOAD\_LIB, UNLOAD\_LIB

};

volatile TalkersCommand previousCommand = GETCOMMAND;

#pragma region Contact

struct Contact // элемент списка подключений

{

enum TE { // состояние сервера подключения

EMPTY, // пустой элемент списка подключений

ACCEPT, // подключен (accept), но не обслуживается

CONTACT // передан обслуживающему серверу

} type;

enum ST { // состояние обслуживающего сервера

WORK, // идет обмен данными с клиентом

ABORT, // обслуживающий сервер завершился не нормально

TIMEOUT, // обслуживающий сервер завершился по времени

FINISH // обслуживающий сервер завершился нормально

} sthread;

SOCKET s; // сокет для обмена данными с клиентом

SOCKADDR\_IN prms; // параметры сокета

int lprms; // длина prms

HANDLE hthread; // handle потока (или процесса)

HANDLE htimer; // handle таймера

HANDLE serverHThtead;

char msg[50]; // сообщение

char srvname[15]; // наименование обслуживающего сервера

Contact(TE t = EMPTY, const char\* namesrv = "")

{

ZeroMemory(&prms, sizeof(SOCKADDR\_IN));

lprms = sizeof(SOCKADDR\_IN);

type = t;

strcpy(srvname, namesrv);

msg[0] = 0x00;

};

void SetST(ST sth, const char\* m = "")

{

sthread = sth;

strcpy(msg, m);

}

};

typedef list<Contact> ListContact;

#pragma endregion

ListContact contacts; // список подключений

bool GetRequestFromClient(char\* name, short port, SOCKADDR\_IN\* from, int\* flen);

bool AcceptCycle(int sq)

{

bool rc = false;

Contact c(Contact::ACCEPT, "EchoServer");

while (sq-- > 0 && !rc)

{

if ((c.s = accept(sS, (sockaddr\*)&c.prms, &c.lprms)) == INVALID\_SOCKET) прием соединения на сокете сервера

{

if (WSAGetLastError() != WSAEWOULDBLOCK)

throw SetErrorMsgText("accept:", WSAGetLastError());

}

else

{

rc = true; успешное принятие соединения

EnterCriticalSection(&scListContact); Защищенная критическая секция scListContact входит в состояние блокировки с помощью EnterCriticalSection, чтобы предотвратить конфликты доступа к общему списку контактов

contacts.push\_front(c); Созданный контакт c добавляется в начало списка контактов contacts

LeaveCriticalSection(&scListContact); Защищенная критическая секция scListContact покидает состояние блокировки

puts("contact connected");

InterlockedIncrement(&connectionCount); Счетчик connectionCount инкрементируется

}

}

return rc;

};

void openSocket() {

SOCKADDR\_IN serv; параметры сокета

sockaddr\_in clnt; параметры клиента

u\_long nonblk = 1; для установки неблокирующего режима сокета

if ((sS = socket(AF\_INET, SOCK\_STREAM, NULL)) == INVALID\_SOCKET) конструктор сокета

throw SetErrorMsgText("socket:", WSAGetLastError());

int lclnt = sizeof(clnt);

serv.sin\_family = AF\_INET;

serv.sin\_port = htons(serverPort);

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

if (bind(sS, (LPSOCKADDR)&serv, sizeof(serv)) == SOCKET\_ERROR) привязка сокета с его параметрами

throw SetErrorMsgText("bind:", WSAGetLastError());

if (listen(sS, SOMAXCONN) == SOCKET\_ERROR) сервер на прослушку

throw SetErrorMsgText("listen:", WSAGetLastError());

if (ioctlsocket(sS, FIONBIO, &nonblk) == SOCKET\_ERROR) установка сокета в неблокирущий режим

throw SetErrorMsgText("ioctlsocket:", WSAGetLastError());

}

void closeSocket() {

if (closesocket(sS) == SOCKET\_ERROR)

throw SetErrorMsgText("closesocket:", WSAGetLastError());

}

void CommandsCycle(TalkersCommand& cmd)

{

int sq = 0;

while (cmd != EXIT)

{

switch (cmd)

{

case START: cmd = GETCOMMAND; команда старт

if (previousCommand != START) {

sq = AS\_SQ;

puts("Start command");

openSocket();

previousCommand = START;

}

else puts("start already in use");

break;

case STOP: cmd = GETCOMMAND;

if (previousCommand != STOP) {

sq = 0;

puts("Stop command");

closeSocket();

previousCommand = STOP;

}

else puts("stop already in use");

break;

case WAIT: cmd = GETCOMMAND;

sq = 0;

puts("Wait command\n" \

"socket closed for waiting other clients");

closeSocket();

while (contacts.size() != 0);

printf("size of contacts %d\n", contacts.size());

cmd = START;

previousCommand = WAIT;

puts("socket is open");

break;

case SHUTDOWN:

sq = 0;

puts("SHUTDOWN command\n" \

"........shutting down...........");

closeSocket();

while (contacts.size() != 0);

printf("size of contacts %d\n", contacts.size());

cmd = EXIT;

break;

case GETCOMMAND: cmd = GETCOMMAND;

break;

};

if (cmd != STOP) {

if (AcceptCycle(sq))

{

cmd = GETCOMMAND;

SetEvent(hClientConnectedEvent);

}

else SleepEx(0, TRUE);

}

};

};

DWORD WINAPI AcceptServer(LPVOID pPrm)

{

DWORD rc = 0;

WSADATA wsaData;

try

{

if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)

throw SetErrorMsgText("Startup:", WSAGetLastError());

CommandsCycle(\*((TalkersCommand\*)pPrm));

if (WSACleanup() == SOCKET\_ERROR)

throw SetErrorMsgText("Cleanup:", WSAGetLastError());

}

catch (string errorMsgText)

{

printf("\n%s", errorMsgText.c\_str());

}

puts("shutdown acceptServer");

ExitThread(rc); завершение потока

}

TalkersCommand set\_param(char\* param) {

if (!strcmp(param, "start")) return START;

if (!strcmp(param, "stop")) return STOP;

if (!strcmp(param, "exit")) return EXIT;

if (!strcmp(param, "wait")) return WAIT;

if (!strcmp(param, "shutdown")) return SHUTDOWN;

if (!strcmp(param, "statistics")) return STATISTICS;

if (!strcmp(param, "getcommand")) return GETCOMMAND;

if (strstr(param, "UNLOAD\_LIB")) return UNLOAD\_LIB;

if (strstr(param, "LOAD\_LIB")) return LOAD\_LIB;

}

typedef void\* (\*FUNCTION)(char\*, LPVOID);

FUNCTION ts;

volatile bool is\_load\_library = false;

std::list<HMODULE> list\_of\_dlls;

std::list<FUNCTION> list\_of\_functions;

HMODULE st;

SOCKET sSUDP;

DWORD WINAPI ConsolePipe(LPVOID pPrm)

{

DWORD rc = 0;

char rbuf[100];

DWORD dwRead, dwWrite;

HANDLE hPipe; хранение дескриптора именованного канала

try

{

char namedPipeConnectionString[50];

sprintf(namedPipeConnectionString, "\\\\.\\pipe\\%s", namedPipeName);

if ((hPipe = CreateNamedPipeA(namedPipeConnectionString,

PIPE\_ACCESS\_DUPLEX,

PIPE\_TYPE\_MESSAGE | PIPE\_WAIT,

1, NULL, NULL,

INFINITE, NULL)) == INVALID\_HANDLE\_VALUE)

throw SetPipeError("create:", GetLastError()); создание именнованого канала

if (!ConnectNamedPipe(hPipe, NULL)) ожидание подключения к каналу

throw SetPipeError("connect:", GetLastError());

TalkersCommand& param = \*((TalkersCommand\*)pPrm);

while (param != EXIT) {

puts("Connecting to Named Pipe Client ...");

ConnectNamedPipe(hPipe, NULL);

while (ReadFile(hPipe, rbuf, sizeof(rbuf), &dwRead, NULL)) чтение данных из канала

{

printf("main client message: %s\n", rbuf);

param = set\_param(rbuf);

if (param == LOAD\_LIB)

{

is\_load\_library = true;

EnterCriticalSection(&scListContact); вход в критическую секцию

list\_of\_dlls.push\_front(LoadLibraryA(strstr(rbuf, "Win"))); загрузка библиотеки

list\_of\_functions.push\_front((FUNCTION)GetProcAddress(list\_of\_dlls.front(), "SSS")); получение адреса функции и добавление в начало списка

LeaveCriticalSection(&scListContact); выход из критической секции

}

else if (param == UNLOAD\_LIB)

{

is\_load\_library = false;

EnterCriticalSection(&scListContact);

list\_of\_dlls.pop\_front(); удаление первого элемента списка

list\_of\_functions.pop\_front();

LeaveCriticalSection(&scListContact);

}

if (param == STATISTICS)

{

char sendStastistics[200];

sprintf(sendStastistics, "\nStatistics\n"\

"count of connectings : %d\n" \

"count of denides: %d\n" \

"success end: %d\n" \

"count of active connections : %d\n" \

"", connectionCount, sayNoCount, successConnections, contacts.size());

WriteFile(hPipe, sendStastistics, sizeof(sendStastistics), &dwWrite, NULL); запись в канал

}

if (param != STATISTICS)

WriteFile(hPipe, rbuf, strlen(rbuf) + 1, &dwWrite, NULL);

if (param == EXIT || param == SHUTDOWN) {

break;

}

}

DisconnectNamedPipe(hPipe); отключение клиента от канала

if (param == EXIT || param == SHUTDOWN) {

break;

}

}

}

catch (string ErrorPipeText)

{

printf("\n%s", ErrorPipeText.c\_str());

return -1;

}

CloseHandle(hPipe);

puts("shutdown ConsolePipe");

ExitThread(rc);

}

DWORD WINAPI GarbageCleaner(LPVOID pPrm)

{

DWORD rc = 0;

while (\*((TalkersCommand\*)pPrm) != EXIT) {

int listSize = 0;

int howMuchClean = 0;

if (contacts.size() != 0) {

for (auto i = contacts.begin(); i != contacts.end();) {

if (i->type == i->EMPTY) {

EnterCriticalSection(&scListContact);

if (i->sthread == i->FINISH)

InterlockedIncrement(&successConnections);

if (i->sthread == i->ABORT || i->sthread == i->TIMEOUT)

InterlockedIncrement(&sayNoCount);

i = contacts.erase(i);

howMuchClean++;

listSize = contacts.size();

LeaveCriticalSection(&scListContact);

}

else ++i;

}

}

}

puts("shutdown garbageCleaner");

ExitThread(rc);

}

void CALLBACK ASWTimer(LPVOID Prm, DWORD, DWORD) {

Contact\* contact = (Contact\*)(Prm);

printf("ASWTimer is calling %p\n", contact->hthread);

TerminateThread(contact->serverHThtead, NULL); принудительное завершение потока

send(contact->s, "TimeOUT", strlen("TimeOUT") + 1, NULL);

EnterCriticalSection(&scListContact);

CancelWaitableTimer(contact->htimer); отмена ожидающего таймера

contact->type = contact->EMPTY;

contact->sthread = contact->TIMEOUT;

LeaveCriticalSection(&scListContact);

}

DWORD WINAPI DispatchServer(LPVOID pPrm)

{

DWORD rc = 0;

TalkersCommand& command = \*(TalkersCommand\*)pPrm;

while (command != EXIT)

{

if (command == STOP) continue;

WaitForSingleObject(hClientConnectedEvent, INFINITE); Ожидание события hClientConnectedEvent. Поток блокируется до тех пор, пока событие не будет сигнализировано.

ResetEvent(hClientConnectedEvent); Сброс события hClientConnectedEvent в несигнальное состояние.

while (true) { получение команд от клиентов

for (auto i = contacts.begin(); i != contacts.end(); i++) {

if (i->type == i->ACCEPT) {

char serviceType[10];

if (recv(i->s, serviceType, sizeof(serviceType), NULL) < 1)

continue;

cout << "New command - " << serviceType << endl;

strcpy(i->msg, serviceType);

if (!strcmp(i->msg, "close")) {

if ((send(i->s, "echo: close", strlen("echo: close") + 1, NULL)) == SOCKET\_ERROR)

throw SetErrorMsgText("send:", WSAGetLastError());

i->sthread = i->FINISH;

i->type = i->EMPTY;

continue;

}

if (strcmp(i->msg, "Echo") && strcmp(i->msg, "Time") && strcmp(i->msg, "Random")) {

if ((send(i->s, "ErrorInquiry", strlen("ErrorInquiry") + 1, NULL)) == SOCKET\_ERROR)

throw SetErrorMsgText("send:", WSAGetLastError());

i->sthread = i->ABORT;

i->type = i->EMPTY;

if (closesocket(i->s) == SOCKET\_ERROR)

throw SetErrorMsgText("closesocket:", WSAGetLastError());

} else {

i->type = i->CONTACT;

i->hthread = hAcceptServer;

i->serverHThtead = ts(serviceType, (LPVOID) & (\*i));

i->htimer = CreateWaitableTimer(0, FALSE, 0);

LARGE\_INTEGER Li;

int seconds = 60;

Li.QuadPart = -(10000000 \* seconds);

SetWaitableTimer(i->htimer, &Li, 0, ASWTimer, (LPVOID) & (\*i), FALSE); Установка значения ожидаемого таймера i->htimer с помощью функции SetWaitableTimer. При истечении времени, будет вызвана функция ASWTimer, а в качестве аргумента ей будет передан указатель на структуру i.

SleepEx(0, TRUE); Вызов функции SleepEx для приостановки выполнения потока. В данном случае, поток будет ожидать, пока не произойдет истечение времени таймера или не будет сигнализировано другое событие.

}

}

}

Sleep(200);

}

}

puts("shutdown dispatchServer");

ExitThread(rc);

}

bool PutAnswerToClient(char\* name, sockaddr\* to, int\* lto) {

char msg[] = "You can connect to server ";

if ((sendto(sSUDP, msg, sizeof(msg) + 1, NULL, to, \*lto)) == SOCKET\_ERROR)

throw SetErrorMsgText("sendto:", WSAGetLastError());

return false;

}

bool GetRequestFromClient(char\* name, short port, SOCKADDR\_IN\* from, int\* flen)

{

SOCKADDR\_IN clnt; Создание переменной clnt типа SOCKADDR\_IN, которая будет хранить информацию о клиенте

int lc = sizeof(clnt);

ZeroMemory(&clnt, lc);

char ibuf[500];

int lb = 0;

int optval = 1;

int TimeOut = 10;

setsockopt(sSUDP, SOL\_SOCKET, SO\_BROADCAST , (char\*)&optval, sizeof(int)); Установка опции SO\_BROADCAST для сокета sSUDP с помощью функции setsockopt. Опция позволяет отправлять и принимать широковещательные сообщения.

setsockopt(sSUDP, SOL\_SOCKET, SO\_RCVTIMEO , (char\*)&TimeOut, sizeof(TimeOut)); - Установка опции SO\_RCVTIMEO для сокета sSUDP с помощью функции setsockopt. Опция устанавливает время ожидания приема данных.

while (true) {

if ((lb = recvfrom(sSUDP, ibuf, sizeof(ibuf), NULL, (sockaddr\*)&clnt, &lc)) == SOCKET\_ERROR) return false;

ibuf[lb] = '\0';

cout << ibuf << endl;

if (!strcmp(name, ibuf)) {

\*from = clnt;

\*flen = lc;

return true;

}

puts("\nbad name");

}

}

DWORD WINAPI ResponseServer(LPVOID pPrm)

{

DWORD rc = 0;

WSADATA wsaData;

SOCKADDR\_IN serv;

if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0)

throw SetErrorMsgText("Startup:", WSAGetLastError());

if ((sSUDP = socket(AF\_INET, SOCK\_DGRAM, NULL)) == INVALID\_SOCKET)

throw SetErrorMsgText("socket:", WSAGetLastError());

serv.sin\_family = AF\_INET;

serv.sin\_port = htons(serverPort);

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

if (bind(sSUDP, (LPSOCKADDR)&serv, sizeof(serv)) == SOCKET\_ERROR)

throw SetErrorMsgText("bind:", WSAGetLastError());

SOCKADDR\_IN some\_server;

int serverSize = sizeof(some\_server);

SOCKADDR\_IN from;

int lc = sizeof(from);

ZeroMemory(&from, lc);

int numberOfClients = 0;

while (\*(TalkersCommand\*)pPrm != EXIT)

{

try

{

if (GetRequestFromClient("Hello", serverPort, &from, &lc))

{

printf("\nconnected Client: %d, port: %d, address: %s", ++numberOfClients, htons(from.sin\_port), inet\_ntoa(from.sin\_addr));

PutAnswerToClient("Hello", (sockaddr\*)&from, &lc);

}

}

catch (string errorMsgText)

{

printf("\n%s", errorMsgText.c\_str());

}

}

if (closesocket(sSUDP) == SOCKET\_ERROR)

throw SetErrorMsgText("closesocket:", WSAGetLastError());

if (WSACleanup() == SOCKET\_ERROR)

throw SetErrorMsgText("Cleanup:", WSAGetLastError());

ExitThread(rc);

}

int main(int argc, char\* argv[])

{

if (argc == 2) {

serverPort = atoi(argv[1]);

}

else if (argc == 3) {

serverPort = atoi(argv[1]);

strcpy(dllName, argv[2]);

}

else if (argc == 4) {

serverPort = atoi(argv[1]);

strcpy(dllName, argv[2]);

strcpy(namedPipeName, argv[3]);

}

else {

serverPort = 2000;

strcpy(dllName, "Win32Project1.dll");

strcpy(namedPipeName, "BOX");

}

printf("server port %d\n", serverPort);

st = LoadLibraryA(dllName); Загружается динамическая библиотека

ts = (HANDLE(\*)(char\*, LPVOID))GetProcAddress(st, "SSS");

volatile TalkersCommand cmd = START;

InitializeCriticalSection(&scListContact);

hAcceptServer = CreateThread(NULL, NULL, AcceptServer,

(LPVOID)&cmd, NULL, NULL); поток для принятия соединений от клиентов, вызывается функция AcceptServer

hConsolePipe = CreateThread(NULL, NULL, ConsolePipe,

(LPVOID)&cmd, NULL, NULL); поток для обработки команд с консоли и через именованный канал (pipe), вызывается функция ConsolePipe

hGarbageCleaner = CreateThread(NULL, NULL, GarbageCleaner,

(LPVOID)&cmd, NULL, NULL); поток для очистки списка контактов, вызывается функция GarbageCleaner.

hDispatchServer = CreateThread(NULL, NULL, DispatchServer,

(LPVOID)&cmd, NULL, NULL); поток для обработки запросов от клиентов и отправки их на соответствующие обработчики, вызывается функция DispatchServer

hResponseServer = CreateThread(NULL, NULL, ResponseServer,

(LPVOID)&cmd, NULL, NULL); поток для обработки ответов от клиентов, вызывается функция ResponseServer

SetThreadPriority(hGarbageCleaner, THREAD\_PRIORITY\_BELOW\_NORMAL);

SetThreadPriority(hDispatchServer, THREAD\_PRIORITY\_NORMAL);

SetThreadPriority(hConsolePipe, THREAD\_PRIORITY\_NORMAL);

SetThreadPriority(hResponseServer, THREAD\_PRIORITY\_NORMAL);

SetThreadPriority(hAcceptServer, THREAD\_PRIORITY\_HIGHEST);

WaitForSingleObject(hAcceptServer, INFINITE); Ожидается завершение работы потоков

CloseHandle(hAcceptServer);

WaitForSingleObject(hConsolePipe, INFINITE);

CloseHandle(hConsolePipe);

WaitForSingleObject(hGarbageCleaner, INFINITE);

CloseHandle(hGarbageCleaner);

TerminateThread(hDispatchServer, 0); Потоки hDispatchServer и hResponseServer принудительно завершаются

puts("shutdown dispatchServer");

TerminateThread(hResponseServer, 0);

puts("shutdown responseServer");

CloseHandle(hDispatchServer);

CloseHandle(hResponseServer);

DeleteCriticalSection(&scListContact); Удаляется критическая секция

FreeLibrary(st); Выгружается динамическая библиотека

return 0;

};