Григорьев В.М.

Лабораторный практикум по программированию в сетях TCP/IP с помощью библиотеки Winsock операционных систем Windows

Днепропетровск 2009

УДК 681.3.07
Лабораторний практикум із програмування у мережах TCP/IP за допомогою бібліотеки Winsock операційних систем Windows.: Навч.-мет. посіб.// Григор'в В.М., 2009. – с.
Розроблено лабораторний практикум практикум із програмування у мережах TCP/IP за допомогою бібліотеки Winsock операційних систем Windows. Лабораторний практикум покликаний навчити студентів самостійно програмувати надійні мережеві додатки для роботи в операційних системами Windows. В ході практикуму докладно вивчається протокол TCP / IP і програмний інтерфейс для програмування сокетів. Наведено завдання для курсової роботи
Для студентів спеціальності ―Комп'ютерні системи та мережі‖ Дніпропетровського національного університету.
Затверджено на засіданні кафедри ЕОМ ФФЕКС ДНУ, протокол №7 від 1 грудня 2009р.
Навчальне видання
Віктор Михайлович Григор'єв

Содержание

Лабораторная работа №1. Введение в Winsock.

Winsock является стандартным прикладным программным интерфейсом (application programming interface (API)), позволяющим двум и более приложениям (вычислительным процессам) взаимодействовать или внутри одного вычислительного устройства или по сети. Winsock не является протоколом и является основным средством для программирования сетевых приложений для операционных систем Windows. Winsock предоставляет программный интерфейс для коммуникации процессов с помощью всех известных современных протоколов: TCP/IP версий 4 и 6, IPX/SPX, NetBIOS, ATM, Bluetooth и т.д. Winsock берёт своё начало от сокетов UNIX платформ клона BSD.

Основная цель разработки спецификации Winsock 2 — создать независимый от транспортного протокола программный интерфейс. К его неоспоримым преимуществам относится предоставление единого привычного интерфейса сетевого программирования для различных транспортов сети.

Теоретическая часть.

Характеристики протоколов

Протокол называют ориентированным на передачу сообщений (датаграмм), если для каждой команды записи он передает байты по сети в отдельном сообщении. Это также означает, что приемник получит данные в виде отдельного сообщения отправителя. Таким образом, приемник получит только одно сообщение за одну команду чтения, даже если передатчик передал несколько сообщений.

Протокол, не сохраняющий границы сообщений, обычно называют *протоколом, основанным на потоке.* Потоковая служба непрерывно передает данные: отправитель пишет в сеть произвольное количество данных, а получатель считывает столько данных, сколько имеется в наличии. На приемнике сетевой стек считывает данные из сети по мере их поступления и буферизует для целевого процесса. Когда процесс запрашивает данные, система возвращает максимально возможное количество данных, не переполняющее буфер, предоставленный клиентским вызовом. Сообщения можно передавать/принимать с помощью потока. Однако способ разделения потока на сообщения должен быть определено в прикладной программе. Это может быть либо специальный символ(ы) типа \n, либо сообщению может предшествовать заголовок, содержащий длину сообщения.

Различают протоколы, требующие и не требующие установления логического соединения. О наличии и отсутствии логического соединения говорят применительно к протоколам, а не применительно к наличию физического соединения. Иными словами, речь идет о способе передачи данных по физическому носителю, а не о самом физическом носителе: в обеих случаях между двумя абонентами существует сетевое соединение.

В протоколах, не требующих соединения, каждый пакет передается независимо от остальных. Тогда как протоколы, устанавливающие соединение, поддерживают информацию о состоянии, которая позволяет следить за последовательностью пакетов.

При работе с протоколом, не требующим соединения, информация пересылается в виде датаграмм, которые адресуется и посылается индивидуально. С точки зрения протокола каждая датаграмма - это независимая единица, не имеющая ничего общего с другими датаграммами, которыми обмениваются приложения. Такой протокол не обязательно надежен, то есть сеть предпримет все возвозможное для доставки каждой датаграммы, но нет гарантий, что ни одна не будет потеряна, задержана или доставлена не в том порядке.

Протоколы, требующие установления соединения, самостоятельно отслеживают состояние пакетов. Сохраняемая информация о состоянии позволяет протоколу обеспечить надежную доставку. Например отправитель запоминает, когда и какие данные послал, но они еще не подтверждены. Если подтверждение не приходит в течение определенного времени, отправитель повторяет передачу. Получатель запоминает, какие данные уже принял, и отбрасывает пакеты-дубликаты. Если пакет поступает не в порядке очередности, то получатель может «придержать» его, пока не придут логически предшествующие пакеты.

Как правило в протоколах, требующих соединения данные передаются потоком.

У типичного протокола, требующего наличия соединения, есть три фазы. Сначала устанавливается соединение между двумя приложениями. Затем происходит обмен данными. И, наконец, когда оба приложения завершили обмен данными, соединение разрывается.

Обычно такой протокол сравнивают с телефонным разговором, а протокол, не требующий соединения, - с отправкой почтовой открытки. Отправитель не знает, существует ли вообще получатель, или не помешает ли стихийное бедствие почтовой службе доставить послание. Все открытки оказываются самостоятельными сущностями. Каждая открытка имеет адрес назначения и доставляется адресату индивидуально. Каждая открытка обрабатывается на почте независимо от других. Почта не отслеживает историю переписки, то есть состояние последовательности открыток. Кроме того, не гарантируется, что открытки не затеряются, не задержатся и будут доставлены в правильном порядке.

При телефонном разговоре для начала набирается номер абонента. Он отвечает. Некоторое время идёт разговор. Можно попросить повторить непонятую фразу. Потом идёт прощание и вешается трубка. Так же обстоит дело и в протоколе, требующем соединения. В ходе процедуры установления соединения

одна из сторон связывается с другой, стороны обмениваются «приветствиями» (на этом этапе они «договариваются» о тех параметрах и соглашениях которым будут следовать далее), и соединение вступает в фазу обмена данными. Во время телефонного разговора звонящий знает своего собеседника. И перед каждой фразой не нужно снова набирать номер телефона - соединение установлено. Аналогично в фазе передачи данных протокола, требующего наличия соединения, не надо передавать свой адрес или адрес другой стороны. Эти адреса - часть информации о состоянии, хранящейся вместе с логическим соединением. Остается только посылать данные, не заботясь ни об адресации, ни о других деталях, связанных с протоколом. Как и в разговоре по телефону, каждая сторона, заканчивая передачу данных, информирует об этом собеседника. Когда обе стороны договорились о завершении, они выполняют строго определенную процедуру разрыва соединения.

В связи с многочисленными недостатками протоколов, не требующих соединения, возникает закономерный вопрос: зачем вообще нужен такой вид протоколов? Часто встречаются ситуации, когда для создания приложения использование именно такого протокола оправдано. Например, протокол без соединения может легко поддерживать связь одного хоста со многими и наоборот. Между тем протоколы, устанавливающие соединение, должны обычно организовать по одному соединению между каждой парой хостов. Впрочем, установление канала связи между двумя участниками и гарантия безошибочной с правильным порядком передачи данных влечет дополнительные издержки. С другой стороны, протокол, не требующих соединения, можно использовать для быстрой передачи коротких сообщений. Контроль за правильностью передачи можно возложить на приложения. Важно то, что протоколы, не требующие наличия соединения, - это фундамент, на котором строятся более сложные протоколы.

Internet Protocol (IP) широко используется в Интернете, поддерживается большинством ОС и применяется как в локальных (local area networks, LAN),так и в глобальных сетях (wide area networks, WAN. IP не требует установления соединения и не гарантирует доставку данных поэтому для передачи данных поверх IP используются два протокола более высокого уровня TCP (Transrmssion Control Protocol – протокол управления передачей) и UDP (User Datagram Protocol рпользовательский датаграммный протокол). Каждый TCP и UDP пакет инкапсулируются в поле данных IP-пакета.

В стеке TCP/IP протокол UDP работает без установки соединения, передаёт данные в виде датаграмм и не гарантирует их безошибочную передачу в правильном порядке. UDP может осуществлять передачу данных множеству адресатов и принимать данные от множества источников. Например, данные, отправляемые клиентом на сервер, передаются немедленно, независимо от того, готов ли сервер к приему. При получении данных от клиента, сервер не подтверждает их прием.

Протокол TCP работает с установкой соединения, передаёт данные в виде потока байт и гарантирует их безошибочную передачу в правильном порядке между двумя компьютерами. Когда приложения связываются по TCP, осуществляется виртуальное соединение исходного компьютера с целевым, после чего между ними возможен одновременный двунаправленный обмен данными. То есть в случае протокола TCP имеем полнодуплексную связь, как при телефонном разговоре: два связанных абонента могут разговаривать одновременно.

Инициализация Winsock

Перед вызовом любой функции Winsock необходимо загрузить правильную версию библиотеки Winsock. Функция инициализации Winsock

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);
```

Первый параметр - версия библиотеки Winsock, которую необходимо загрузить. На современных платформах Win32 используется версия 2.2. Единственное исключение - Windows CE, поддерживающая только Winsock 1.1. Для загрузки версии Winsock 2.2 укажите значение 0x0202, либо макрос MAKEWORD(2, 2). Верхний байт определяет дополнительный номер версии, нижний - основной. Второй параметр - структура WSADATA, возвращаемая по завершении вызова. Она содержит информацию о версии Winsock, загруженной функцией WSAStartup. Функция WSAStartup при успехе возвращает ноль иначе она возвращает одну из ошибок WSASYSNOTREADY, WSAVERNOTSUPPORTED, WSAEINPROGRESS, WSAEPROCLIM, WSAEFAULT, значения которых определены в заголовочном файле winsock2.h и расшифрованы в приложении.

Вызов функции WSAStartup целесообразно окружить диагностической оболочкой, например

```
#include <winsock2.h>
int rc;
WSADATA      wsd;
if ((rc = WSAStartup(MAKEWORD(2,2), &wsd)) != 0)    {
myprintf("Ошибка WSAStartup: %s", encodeWSAGetLastError(rc));
return 1;
}
```

Здесь myprintf – наша функция форматного вывода кирилицы на консоль, работающая также, как и функция printf

```
#include <Windows.h>

int myprintf (const char *lpFormat, ... ) {
int nLen = 0;
int nRet = 0;
char ansiBuffer[512] ;
char oemBuffer[512] ;
va_list arglist ;
HANDLE hOut = NULL;
ZeroMemory(ansiBuffer, sizeof(ansiBuffer));
va_start(arglist, lpFormat);
nLen = lstrlen( lpFormat ) ;
nRet = wvsprintf( ansiBuffer, lpFormat, arglist );
if( nRet >= nLen || GetLastError() == 0 ) {
        hOut = GetStdHandle(STD_OUTPUT_HANDLE) ;
        CharToOem(ansiBuffer,oemBuffer);
        if( hOut != INVALID_HANDLE_VALUE )
        WriteConsole( hOut, oemBuffer, lstrlen(oemBuffer), (LPDWORD)&nLen, NULL ) ;
}
return nLen ;
}
```

Листинг 1. Функция myprintf

Наша функция encodeWSAGetLastError расшифровывает ошибки. Заготовка для этой функции имеет вид

```
char* encodeWSAGetLastError(int n){
switch(n){
case WSAEINTR: return " WSAEINTR: Прерванный вызов функции."; break;
case WSAEACCES: return " WSAEACCES: Доступ запрещен.";  break;

.....

default : return "Неизвестная ошибка"; break;
}
}
```

Листинг 2. Заготовка для функции encodeWSAGetLastError

Самостоятельно завершите эту функцию, используя расшифровку ошибок из приложения.

По завершении работы с библиотекой Winsock вызовите функцию WSACleanup для выгрузки библиотеки и освобождения ресурсов:

```
int WSACleanup (void);
```

Функция при ошибке возвращает SOCKET_ERROR, иначе – ноль. Для каждого вызова WSAStartup необходимо согласованно вызывать WSACleanup, так как каждый стартовый вызов увеличивает значение эталонного счетчика ссылок на загруженные Winsock DLL. Чтобы уменьшить значение счетчика, требуется равное количество вызовов WSACleanup.

Проверка и обработка ошибок

Проверка и обработка ошибок играют весомую роль при написании Winsock-приложения. Функции Winsock достаточно часто возвращают ошибки, но как правило, не критические - передачу информации можно продолжать. Большинство функций Winsock при ошибке вызова возвращают значение SOCKET_ERROR, но так происходит не всегда. При подробном рассмотрении API-вызовов мы обратим внимание на возвращаемые значения, соответствующие ошибкам. Константа SOCKET_ERROR на самом деле равна -1. Для получения более информативного кода ошибки, возникшей после одного из вызовов Winsock, задействуйте функцию

```
int WSAGetLastError (void);
```

Эта функция возвращает код последней ошибки. Если ошибка возникла при выполнении функции WSACleanup , функция WSAGetLastError может вернуть одно из следующих значений: WSANOTINITIALISED, WSAENETDOWN, WSAEINPROGRESS.

Всем кодам ошибок, возвращаемым WSAGetLastError, соответствуют стандартные константные значения. Они описаны в Winsock2.h и приведены в приложении. Константы, определенные для кодов ошибок директивой #define, обычно начинаются с префикса WSAE.

Вызов каждой функции Winsock следует обрамлять диагностической оболочкой, например

```
if (WSACleanup() == SOCKET_ERROR)
myprintf("Ошибка WSACleanup: %s", encodeWSAGetLastError(WSAGetLastError()));
```

Конструкции такого типа затрудняют чтение программы. Поэтому в [1] было предложено вместо оригинальных функций Winsock использовать функции обвёртки. Например для функции WSACleanup функция обвёртка может быть задана так

```
void vSACleanup(){
if (WSACleanup() == SOCKET_ERROR)
myprintf("Ошибка WSACleanup: %s", encodeWSAGetLastError(WSAGetLastError()));
}
```

Листинг 2. Обвёртка для функции WSACleanup

Приложение не модет вызывать функцию WSAGetLastError для определения кода ошибки, если вызов функции WSAStartup закончился неудачей. При неудачном вызове WSAStartup динамическая библиотека WS2_32.DLL, содержащая коды функций Winsock, не будет загружена и следовательно код функции WSAGetLastError будет недоступен. Из этого следует, что функцию обвёртку для функции WSAStartup можно не писать или написать обвёртку без использования вызова функции WSAGetLastError, например

```
void vSAStartup (){
int rc;
WSADATA       wsd;
if ((rc = WSAStartup(MAKEWORD(2,2), &wsd)) != 0)    {
myprintf("Ошибка WSAStartup: %s", encodeWSAGetLastError(rc));
return 1;
}
```

Сокеты Windows

Сетевой программный интерфейс Winsock основан на понятии сокета. Сокет — это описатель поставщика сетевого транспорта. Создание сокета позволит приложениям осуществлять сетевое подключение и обмен данными через него. В Win32, в отличие от UNIX/LINUX , сокет отличается от описателя файла, а потому представлен отдельным типом — SOCKET (переопредёлённое беззнаковое целое). Это означает, что в отличие от UNIX/LINUX, с сокетом нельзя обмениваться информацией с помощью файловых функций ввода и вывода типа read и write. Сокет создается функцией

```
SOCKET socket (int af, int type, int protocol).
```

Первый параметр — af, определяет семейство адресов протокола. Например, если вы хотите создать UDP- или TCP-сокет, подставьте константу AF_INET, чтобы сослаться на протокол IP версии 4. Второй параметр — type, это тип сокета для данного протокола. Он может принимать одно из следующих значений SOCK_STREAM, SOCK_DGRAM, SOCK_RAW и т.д. Третий параметр — protocol, указывает конкретный транспорт, если для данного семейства адресов и типа сокета существует несколько записей. С помощью команды **netsh winsock show catalog** можно получить поддерживаемые на компьютере семейства адресов, типы сокетов и протоколы. Эту же информацию можно получигь, вызвав утилиту Msinfo32. Далее в ней выбрать путь components -> network -> protocol.

При успешном вызове функция socket возвращает дескриптор, ссылающийся на новый сокет. Иначе возвращается значение INVALID_SOCKET и код ошибки может быть получен с помощью функциis WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEAFNOSUPPORT, WSAEINPROGRESS, WSAEMFILE, WSAENOBUFS, WSAEPROTONOSUPPORT, WSAEPROTOTYPE, WSAESOCKTNOSUPPORT.

Создать TCP-сокет можно так

```
SOCKET s = socket(AF_INET, SOCK_STREAM, 0)
```

Или так
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)

Создать UDP-сокет можно так

SOCKET s = socket(AF_INET, SOCK_ DGRAM, 0)

Или так

SOCKET s = socket(AF_INET, SOCK_ DGRAM, IPPROTO_ UDP)

Можно предложить следующую функцию обвёртку

```
Int Socket(int family, int type, int protocol)
{       int                 n;
if ( (n = socket(family, type, protocol)) == INVALID_SOCKET){
        myprintf("Ошибка socket %s ", encodeWSAGetLastError(WSAGetLastError()));
        exit(1)
}
return(n);
}
```

И создавать сокеты можно, например так

SOCKET s = Socket(AF_INET, SOCK_ DGRAM, IPPROTO_ UDP).

Сокеты, ориентированные на соединение, такие как SOCK_STREAM предоставляют полнодуплексное соединение и должны быть в соединённом состоянии до того пока все данные будет через них приняты или переданы. Соединение клиента с другим сокетом создаётся с помощью функции connect. После соединения данные передаются с помощью функции send и принимаются с помощью функции recv. В конце обмена данными следует закрыть сокет вызовом функции closesocket.

int closesocket(SOCKET s),

где s – дескриптор закрываемого сокета. При успешном вызове функция возвращает ноль. Иначе возвращается значение INVALID_SOCKET и код ошибки может быть получен с помощью функцииs WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAENOTSOCK, WSAEINPROGRESS, WSAEINTR, WSAEWOULDBLOCK. Можно предложить следующую функцию обвёртку

```
void closesocket(SOCKET s){
int                 n;
if ( closesocket (s) == INVALID_SOCKET)
myprintf ("Ошибка closesocket %s ", encodeWSAGetLastError( WSAGetLastError()));}
```

Сокеты TCP гарантируют, что данные не будут потеряны, дублированы или передадутся в неправильном порядке. Если данные не могут быть переданы за приемлемое время (например из-за недостатка памяти для организации буферов) соединение рассматривается как разорванное и все последующие попытки передачи данных потерпят неудачу с кодом ошибки WSAETIMEDOUT.

Сокеты UDP позволяют передавать и принимать данные от произвольных абонентов с помощью функций sendto и recvfrom. В случае необходимости сокет UDP могут быть соединён с другим сокетом UDP с помощью функции connect и датаграммы этому сокету могут быть отосланы с помощью функции send, а приняты с помощью функции recv.

Адресация в Winsock для в IP-сетей

Каждый протокол имеет свои правила образования семейств адресов и разрешения имен. Мы будем рассматривать лишь протокол TCP/IPv4. Для связи средствами Winsock важны механизмы адресации рабочих станций и серверов. Здесь рассматривается порядок разрешения адресов. И TCP, и UDP передают данные по IP, поэтому обычно говорят об использовании TCP/IP или UDP/IP. В Winsock для IP-соединений предусмотрено семейство адресов AF_INET, определенное в файлах Winsock.h и Winsock2.h.

При использовании IP сетевым интерфейсам сетевых устройств назначается один или более IP-адресов, состоящий из 32 бит, официально называемый IP-адресом версии 4 (IPv4). Для взаимодействия с сервером по TCP или UDP клиент должен указать IP-адрес сетевого интерфейса сервера и номер порта

службы. Чтобы прослушивать входящие запросы клиента, сервер также должен указать свой IP-адрес и номер порта. В Winsock IP-адрес и порт службы задают в структуре

```
struct sockaddr_in
{
short sin_family;
unsigned short sin_port;
struct in_addr sin_addr;
char sin_zero[8];
}
```

Поле sin_family должно быть равно AF_ INET: этим Winsock сообщают об использовании семейства адресов IP. Поле sin_port задает, какой коммуникационный порт TCP или UDP будет использован. Приложения должны быть очень внимательны при выборе порта, поскольку некоторые доступные порты зарезервированы для использования популярными службами: такими, как File Transfer Protocol (FTP) и Hypertext Transfer Protocol (HTTP). Эти порты обслуживаются и распределяются центром Internet Assigned Numbers Authority (IANA), их описание см. в RFC 1700. По сути, номера портов делят на три категории: стандартные, зарегистрированные и динамические и (или) частные порты. Диапазоны портов:

0-1023 - управляются IANA и зарезервированы для стандартных служб;
1024-49151 - зарегистрированы IANA и могут использоваться процессами и программами;
49152-65535 - являются динамическими и (или) частными.

Во избежание накладок с портами, уже занятыми системой или другим приложением, ваша программа должна выбирать зарегистрированные порты в диапазоне 1024-49151. Порты 49152-65535 также можно задействовать свободно - с ними не связаны никакие стандартные службы. Если ваше приложение попытается выбрать порт, уже занятый другим приложением на узле, то система вернет ошибку WSAEADDRINUSE.

Поле sin_addr структуры sockaddr_in имеет тип

```
struct in_addr {
unsigned long s_addr
}
```

То есть sin_addr структуры sockaddr_in хранит IP-адрес в 4-байтном виде с типом unsigned long int с сетевым порядком байт. В зависимости от того, как это поле использовано, оно может представлять и локальный, и удаленный IP-адрес. И наконец, поле sin_zero играет роль простого заполнителя, чтобы структура sockaddr_in по размеру равнялась структуре sockaddr, определяемой так

```
struct sockaddr {
unsigned short sa_family;  // семейство адресов: константа AF_xxx
char sa_data[14]; ].  // адрес, специфичный для протокола
};
```

Роль этого заполнителя состоит в создании универсальной структуры адреса сокета. Структуры адреса сокета *всегда* передаются по ссылке при передаче в качестве аргумента для любой функции сокета. Но функции сокета, принимающие один из этих указателей в качестве аргумента, должны работать со структурами адреса сокета из *любого* поддерживаемого семейства протоколов. Проблема в том, как объявить тип передаваемого указателя. Для ANSI C решение простое: void* является указателем на неопределенный (универсальный)тип. Но функции сокетов существовали до появления ANSI C, и в 1982 году было принято решение определить *универсальную* структуру адреса сокета как struct sockaddr.

Функции сокетов определяются таким образом, что их аргументом является указатель на универсальную структуру адреса сокета, например, как показано в прототипе функции

```
int bind(  SOCKET s,  const struct sockaddr* name,  int namelen);
```

При этом требуется, чтобы для любых вызовов этих функций указатель на структуру адреса сокета, специфичную для протокола, был преобразован в указатель на универсальную структуру адреса сокета. Например:

```
struct sockaddr_in serv // структура адреса сокета IPv4
//
// заполняем serv
```

```
//
bind(s, (struct sockaddr *) &serv,  sizeof(serv));
```

Если мы не выполним преобразование (struct sockaddr *), компилятор C сгенерирует предупреждение “Передается указатель несовместимого типа”. С точки зрения разработчика приложений универсальная структура адреса сокета используется только для преобразования указателей на структуры адресов конкретных протоколов.

IP-адреса обычно задают в точечной нотации: a.b.c.d. Здесь каждая буква представляет десятичное число в диапазоне 0-255 для каждого байта и задается слева направо. Полезная вспомогательная функция inet_addr преобразует IP-адрес из точечной нотации в 32-битное длинное целое без знака:

```
unsigned long inet_addr(const char FAR *cp);
```

Поле cp является строкой, заканчивающейся нулевым символом, здесь задается IP-адрес в точечной нотации. Заметьте, что эта функция в качестве результата возвращает IP-адрес, представленный 32-битным числом с сетевым порядком следования байт (network-byte order). При ошибке, если неверно задан адрес, функция возвращает INADDR_NONE.

Сопутствующая функция inet_ntoa преобразует сетевой адрес (Ipv4) в форме struct in_addr в точечную нотацию

```
char* FAR inet_ntoa(  struct  in_addr in);
```

При ошибке функция возвращает NULL.

Существует специальный IP-адрес INADDR_ANY, позволяющий серверному приложению слушать клиента через любой сетевой интерфейс на несущем компьютере. Обычно приложения сервера используют этот адрес, чтобы привязать сокет к локальному интерфейсу для прослушивания соединений. Если на компьютере несколько сетевых адаптеров, то этот адрес позволит отдельному приложению получать отклики от нескольких интерфейсов.

Порядок байт

Разные процессоры представляют числа в одном из двух порядков байт big-endian или little-endian Например, процессоры Intel x86 представляют многобайтные числа, следуя от менее значимого к более значимому байту (little-endian). Если номер порта и IP-адрес хранятся в памяти компьютера как многобайтные числа, они представляются в системном порядке (host-byte-order). Стандарты Интернета требуют, чтобы многобайтные значения представлялись от старшего байта к младшему (в порядке bigendian), что обычно называется сетевым порядком (network-byte order). Есть целый ряд функций для преобразования многобайтных чисел из системного порядка в сетевой и обратно. Например, две следующих API функции преобразуют числа из системного порядка в сетевой

```
u_long htonl(u_long hostlong),
u_short htons(u_short hostshort);
```

Параметр hostlong функции htonl - четырехбайтное число с системным порядком следования байт. Функция htonl возвращает число с сетевым порядком. Параметр hostshort функции htons является двухбайтным числом с системным порядком. Функция htons возвращает число, как двухбайтное значение с сетевым порядком.

Следующие две функции решают обратную задачу переставляют байты из сетевого порядка в системный

```
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

А сейчас продемонстрируем, как создать структуру sockaddr_in при помощи уже описанных функций inet_addr и htons. Рекомендуется обнулять поля этой структуры перед использованием с помощью макроса ZeroMemory(адрес, размер).

```
sockaddr_in InternetAddr;
ZeroMemory(InternetAddr, sizeof(InternetAddr));
int nPort = 5150;
InternetAddr.sin_family = AF_INET;
// Преобразование адреса 87.250.251.11 из десятично-точечной нотации в
// 4-байтное целое число и присвоение результата sm_addr
```

```
InternetAddr.sin_addr.s_addr = inet_addr("87.250.251.11”);
// Переменная nPortld хранится в системном порядке. Преобразование
// nPortld к сетевому порядку и присвоение результата sin_port.
InternetAddr.sin_port = htons(nPort);
```

Листинг 3. Заполнение структуры sockaddr_in

Разрешение имен

Для подключения к узлу по IP Winsock-приложение должно знать IP-адрес этого узла, сложный для запоминания пользователем. Большинство людей более охотно обращаются к компьютерам при помощи легко запоминаемых имен узлов, а не IP-адресов. В Winsock предусмотрено функция для преобразования имени в IP-адрес. Функции gethostbyname отыскивают сведения об узле, соответствующие его имени.

API-функция gethostbyname определена так

```
struct hostent* FAR gethostbyname(  const char* name);
```

Параметр name представляет имя узла. IP-адресс точечной нотации недопустим. При успешном выполнении функции возвращается указатель на структуру hostent, которая хранится в системной памяти. Приложение не должно полагать, что эти сведения непременно статичны Поскольку эта память обслуживается системой, Приложение не должно освобождать память для возвращенной структуры. При ошибке функция возвращает NULL и функция WSAGetLastError может возвратить одну из следующих ошибок WSANOTINITIALISED, WSAENETDOWN, WSAHOST_NOT_FOUND, WSATRY_AGAIN, WSANO_RECOVERY, WSANO_DATA, WSAEINPROGRESS, WSAEFAULT, WSAEINTR

Рассмотрим структуру hostent

```
Struct hostent
{
   char FAR *       h_name;
   char FAR * FAR * h_aliases;
   short          h_addrtype;
   short          h_length;
   char FAR * FAR * h_addr_list;
};
```

Поле h_name является официальным именем узла. Если в сети используется доменная система имен (Domain Name System, DNS), в качестве имени сервера будет возвращено полное имя домена (Fully Qualified Domain Name, FQDN). Если в сети применяется локальный файл узлов (hosts, lmhosts) – это первая запись после IP-адреса. Поле h_aliases - массив, завершающийся нулем (null-terminated array) дополнительных имен узла. Поле h_addrtype представляет возвращаемое семейство адресов. Поле h_length определяет длину в байтах каждого адреса из поля h_addr_list. Поле h_addr_list - массив, завершающийся 0 и содержащий IP-адреса узла. Узел может иметь несколько IP-адресов. Каждый адрес в этом массиве представлен в сетевом порядке. Обычно приложение использует первый адрес из массива. Впрочем, при получении нескольких адресов, приложение должно выбирать адрес случайным образом из числа доступных, а не упорно использовать первый. Следующая программа выводит все адреса сайта yandex.ru

```
#include <winsock2.h>
#include <stdio.h>
main()
{          char                              **pptr;
struct hostent      *hptr;
struct in_addr sin_addr;
WSADATA      wsd;
WSAStartup(MAKEWORD(2,2), &wsd);
hptr = gethostbyname("yandex.ru");
pptr = hptr->h_addr_list; // Список IP-адресов хоста yandex.ru
for ( ; *pptr != NULL; pptr++){// перебираем все IP-адреса
          CopyMemory(&sin_addr, *pptr, hptr->h_length);//Копируем IP-адрес в структуру sin_addr
          printf("\taddress: %s\n",inet_ntoa(sin_addr));//Преобразуем адрес в точечную нотацию
          }
}
```

Листинг 4. Программа, которая выводит все адреса сайта yandex.ru

Сравните результат работы программы с результатом выполнения утилиты

nslookup yandex.ru

87.250.251.11, 93.158.134.11, 213.180.204.11, 77.88.21.11

Таким образом если использовать не IP-адресс, а имя узла в листинге 3 вместо строки
InternetAddr.sin_addr.s_addr = inet_addr("87.250.251.11”);
Следует использовать конструкцию типа

```
struct hostent      *hptr = gethostbyname("yandex.ru");
CopyMemory(&InternetAddr.sin_addr, hptr ->h_addr_list[0], hptr->h_length);
```

Мы здесь используем 0-й IP-адрес хоста yandex.ru.

Если мы хотим создать программу, понимающую и IP-адреса и имена, то можно воспользоваться следующей конструкцией

```
Int iPort = 5001; //порт
//либо
char  szServer[128] = "yandex.ru";
//либо
//char  szServer[128] = "87.250.251.11";
struct sockaddr_in server;
ZeroMemory(server, sizeof(server r));
struct hostent    *host = NULL;
server.sin_family = AF_INET;
   server.sin_port = htons(iPort);
   server.sin_addr.s_addr = inet_addr(szServer);
   if (server.sin_addr.s_addr == INADDR_NONE)
   {
    host = gethostbyname(szServer);
    CopyMemory(&server.sin_addr, host->h_addr_list[0], host->h_length);
   }
```

Листинг 5. Универсальный способ формирования структуры sockaddr_in

Существует независимый от протокола способ преобразования имени в адреса, реализованный в функции getaddrinfo. Разберитесь с ним самостоятельно.

Установления TCP-соединения

Сразу после создания сокета в нем отсутствует информация об адресах оконечных точек (ему не присвоен ни локальный, ни удаленный адрес). Такой сокет называют несвязанным. В серверном приложении для указания адреса локальной оконечной точки вызывается функция bind. Этот вызов принимает параметры, в которых задаются дескриптор сокета и адрес оконечной точки. В протоколах TCP/IP для задания адреса оконечной точки используется структура sockaddr_in, которая включает и IP-адрес, и номер порта протокола. В сервере функция bind используется в основном для указания общепринятого номера порта, через который он будет принимать запросы на установление соединения.

Сразу после создания сокет не является ни активным (т.е. готовым для использования клиентом), ни пассивным (т.е. готовым для использования сервером) до тех пор, пока в приложении не будут осуществлены дальнейшие действия. Серверы с установлением логического соединения вызывают функцию listen, чтобы перевести сокет в пассивный режим и подготовить его для приема входящих запросов на установление соединения. Большая часть серверных программ состоит из бесконечного цикла, в котором принимается очередной входящий запрос на установление соединения, выполняется его обработка, а затем происходит возврат к выполнению операции приема очередного входящего соединения. Даже если обработка каждого входящего соединения занимает только несколько миллисекунд, может оказаться, что новый запрос на установление соединения поступит в тот момент, когда сервер занимается обработкой текущего запроса. Для обеспечения того, чтобы не был потерян ни один запрос на установление соединения, сервер должен передать функции listen параметр, который указывает операционной системе, что запросы на установление соединения, поступающие в сокет, нужно ставить в очередь. Поэтому один параметр вызова функции listen указывает сокет, который должен быть переведен в пассивный режим, а другой —размер очереди, предназначенной для этого сокета. Следует отметить, что это не максимальное число

одновременных соединений с данным портом, а лишь максимальное число частично установленных соединений, ожидающих в очереди, пока приложение их примет.

При работе с сокетами TCP после вызова в серверной программе функций socket (для создания сокета), bind (для указания адреса локальной оконечной точки) и listen (для перевода сокета в пассивный режим) вызывается функция accept для извлечения из очереди следующего входящего запроса на установление соединения. Параметр функции accept указывает сокет, из очереди которого должен быть принят запрос на соединение. Функция accept создает новый сокет для каждого нового запроса на соединение и возвращает дескриптор нового сокета в вызывающую прогамму. В сервере новый сокет используется только для нового соединения, а первоначальный сокет служит для приема следующих запросов на соединение. Сразу после приема запроса на соединение сервер может передавать данные через новый сокет с помощью функций send и recv. Номер локального порта для этого сокета такой же, как и для прослушивающего сокета. Обратите внимание, что оба сокета имеют один и тот же номер локального порта. Это нормально, поскольку TCP-соединение полностью определяется четырьмя параметрами - локальным адресом, локальным портом, удаленным адресом и удаленным портом. Поскольку удаленные адрес и порт для этих двух сокетов различны, то ядро может отличить их друг от друга. Закончив использование нового сокета, сервер его закрывает.

Для формирования запроса на установление соединения сокета клиента с сокетом сервера используется функция connect, в которой указывается IP-адрес и номер порта протокола сервера, на котором должен существовать пассивный сокет, привязанный к адресу и порту. Система сама назначает адрес и порт для клиентского сокета. Однако можно самому привязать клиентский сокет к конкретному адресу и порту клиента с помощью функции bind, что делается крайне редко в случае наличия на клиенте фаервола с закрытыми портами. Если сокет предназначен для работы по протоколу UDP, то функция connect записывает в память указанный адрес удалённой оконечной точки, но не передает по этому адресу никаких дейтаграмм. Если же сокет предназначен для работы по протоколу TCP, то функция connect устанавливает соединение с помощью трехстороннего квитирования (рукопожатия):

1. Сервер должен быть подготовлен для того, чтобы принять входящее соединение. Обычно это достигается путем вызова функций socket, bind и listen и называется пассивным открытием.

2. Клиент выполняет активное открытие, вызывая функцию connect. Это заставляет клиент TCP послать сегмент SYN j (от слова synchronize —синхронизировать), чтобы сообщить серверу начальный порядковый номер j данных, которые клиент будет посылать по соединению.

3. Сервер должен подтвердить получение клиентского сегмента SYN, а также послать свой собственный сегмент SYN k, содержащий начальный порядковый номер k для данных, которые сервер будет посылать по соединению. Сервер посылает SYN k и ACK j+1 — подтверждение приема (от слова acknowledgement) клиентского SYN j — в виде единого сегмента.

4. Клиент должен подтвердить получение сегмента SYN k от сервера, посылая ему пакет ACK k+1.

Для подобного обмена нужно как минимум три пакета, поэтому он называется трехэтапным рукопожатием TCP (TCP three-way handshake).

Повседневной аналогией установления соединения TCP может служить система телефонной связи. Функция socket эквивалентна включению используемого телефона. Функция bind дает возможность другим узнать ваш телефонный номер, чтобы они могли позвонить вам. Функция listen включает звонок, и вы можете услышать, когда происходит входящий звонок. Функция connect требует, чтобы мы знали чей-то номер телефона и могли до него дозвониться. Функция accept — аналогия ответа на входящий звонок. Идентифицирующие данные, возвращаемые функцией accept (это IP-адрес и номер порта клиента), аналогичны телефонному номеру звонящего по телефону. Однако имеется отличие, состоящее в том, что функция accept возвращает идентифицирующие данные клиента только после того, как соединение установлено, а во время телефонного звонка после определения номера телефона звонящего мы можем выбрать, отвечать на звонок или нет. Если используется система DNS , она предоставляет услуги, аналогичные телефонной книге. Действие функции getsockbyname аналогично отысканию номера телефона по конкретному имени, а действие функции getsockbyaddr (здесь не рассматривается) можно сравнить с отысканием имени но упорядоченному списку телефонных номеров.

Информацию об истинном адресе и порте локального соединённого сокета sc можно получить так

```
struct sockaddr_in ca;
int caSize = sizeof(ca);
ZeroMemory(ca, caSize);
getsockname(sc, (struct sockaddr *)&ca, &caSize);
printf("Адрес:%s Порт:%d\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
```

Информацию об истинном адресе и порте удалённого сокета к которому подсоединился локальный сокета sc можно получить так

```
struct sockaddr_in sa;
int saSize = sizeof(sa);
ZeroMemory(sa, saSize);
```

```
getsockname(sc, (struct sockaddr *)&sa, &saSize);
printf("Адрес:%s Порт:%d\n",inet_ntoa(sa.sin_addr),ntohs(sa.sin_port));
```

Приведём спецификации рассмотренных функций

bind

Функция bind связывает локальный адрес с сокетом. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int bind(SOCKET s, const struct sockaddr* name,  int namelen);
```

Все параметры входные. s – связываемый сокет, name – адрес, назначаемый сокету, namelen - длина в байтах параметра name. В случае успеха функция возвращает ноль, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEACCES, WSAEADDRINUSE, WSAEADDRNOTAVAIL, WSAEFAULT, WSAEINPROGRESS, WSAEINVAL, WSAENOBUFS, WSAENOTSOCK. Расшифровку ошибок смотри в приложении или в MSDN.

Функция используется на несоединённых сокетах перед вызовом функций accept, connect или listen. Она используется либо с сокетами ориентированными на соединение (потоки) либо с сокетами не ориентированными на соединение (датаграммы). Если приложение не заботится о назначаемом локальном адресе, следует полю sa_data структуры name назначить константу ADDR_ANY. Это позволяет Winsock использовать для привязки с сокету приемлемый локальный сетевой адрес, что особенно полезно для хостов со многими сетевыми интерфейсами. Если в структуре name задан нулевой порт, то система назначает сокету порт динамически из диапазона 1024 - 5000. Приложение может использовать функци getsockname после вызова bind для получения адреса и порта назначенного сокету. Если адрес равен INADDR_ANY, getsockname возвратит адрес только после соединения сокета, так как хост может иметь несколько адресов. Привязка клиента к порту отличному от нулевого может привести к конфликту с приложениями которые используют этот порт. Поэтому использовать функцию bind на клиенте следует только при истинной необходимости.

Если сокет имеет опции SO_EXCLUSIVEADDR или SO_REUSEADDR то опции сокета можно устанавливать только до вызова функции bind .

Пример

```
 SOCKET ListenSockett = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
 sockaddr_in service;
ZeroMemory(service, sizeof(service));
 service.sin_family = AF_INET;
 service.sin_addr.s_addr = inet_addr("127.0.0.1");
 service.sin_port = htons(27015);
bind( ListenSocket,    (SOCKADDR*) &service,    sizeof(service));
```

listen

Функция listen переводит сокет в состояние прослушивания входящих соединений. Сокет переводится в пассивный режим в котором запросы на входящее соединение подтверждаются и помещаются в очередь. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int listen( SOCKET s,   int backlog);
```

Здесь s – связанный, несоединённый сокет, backlog – максимальная длина очереди ожидающих соединений. Если он установлен в SOMAXCONN, то нижлежащий сервис-провайдер установит для сокета s максимально возможную приемлемую величину backlog. Неприемлемое значение параметра заменяется ближайшим правильным значением. Истинное значение backlog получить нельзя. В случае успеха функция возвращает ноль, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEADDRINUSE, WSAEINPROGRESS, WSAEINVAL, WSAEISCONN, WSAEMFILE, WSAENOBUFS, WSAENOTSOCK, WSAEOPNOTSUPP. Расшифровку ошибок смотри в приложении или в MSDN.

Чтобы принимать соединения, сокет надо вначале привязать к локальному адресу с помощью функции bind. После вызова функции listen сокет может принимать соединения с помощью функции accept. Серверные сокеты, ориентированные на соединение используются с функцией listen. Функция listen обычно используется серверами, имеющими одновременно несколько запросов на соединение. Если клиент натолкнётся на переполненную очередь запросов, то он получит ошибку WSAECONNREFUSED.

Последующий вызов listen для того же сокета ничего не меняет. Так установка параметра backlog в ноль не сбрасывает очередь соединений в ноль.

accept

Функция accept допускает попытку входного соединения на сокете. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

SOCKET accept(SOCKET s, struct sockaddr* addr, int* addrlen);

Здесь s это дескриптор сокета, переведенного функцией listen в состояние прослушивания. Соединение реально осуществляется с новым сокетом, который возвращает эта функция accept. addr это указатель на буфер в котором функция возвращает адрес принятого входного соединения. Формат этого адреса определяется используемым семейством адресов. addrlen это указатель на целое, которое содержит длину структуры, на которую указывает параметр addr. Целое, на которое указывает addrlen, до вызова должно содержать длину области памяти, на которую указывает addr. При вызврате это целое содержит истинную длину в байтах возвращаемого адреса. Для параметров addr и addrlen можно указать NULL, тогда функция не возвратит адресной информации. В случае успеха функция возвращает значение типа SOCKET. Это значение является дескриптором нового сокета на котором осуществляются реальное соединение. Иначе функция возвращает SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок WSANOTINITIALISED, WSAECONNRESET, WSAEFAULT, WSAEINTR, WSAEINVAL, WSAEINPROGRESS, WSAEMFILE, WSAENETDOWN, WSAENOBUFS, WSAENOTSOCK, WSAEOPNOTSUPP, WSAEWOULDBLOCK. Расшифровку ошибок смотри в приложении или в MSDN.

Функция accept берёт первое соединение из очереди отложенных соединеий на сокете s. Затем она создаёт и возвращает новый сокет. Этот сокет будет обслуживать реальное соединение. Вычислительный процесс блокируется на функции accept до тех пор пока очередь отложенных соединений на сокете s пуста. Приняв соединение, функция accept разблокирует процесс, возвращая новый сокет. Сокет s остаётся открытым и продолжает слушать запросы на новое соединение. То есть функцию accept следует вызывать циклически для одного и того же сокета s.

Если предварительно перевести сокет в неблокирующее состояние, то вычислительный процесс не блокируется на функции accept при пустой очереди отложенных соединений, но функция accept установит ошибку WSAEWOULDBLOCK. Функция accept используется с сокетами, ориентированными на соединение такими как SOCK_STREAM.

Пример

```
SOCKET n;
struct sockaddr_in CA;
int CALen = sizeof(CA);
ZeroMemory(CA, CALen);
n =  accept(s, (struct sockaddr *) &CA, &CALen));
```

connect

Функция connect устанавливает соединение к указанному сокету. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

int connect(SOCKET s, const struct sockaddr* name, int namelen);

Параметр s определяет неподсоединённый сокет, name это указатель на структуру sockaddr в которую надо поместить адрес сокета к которому необходимо подсоединиться, namelen - длина этого адреса в байтах. В случае успеха функция возвращает ноль, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEADDRINUSE, WSAEINTR, WSAEINPROGRESS, WSAEALREADY, WSAEADDRNOTAVAIL, WSAEAFNOSUPPORT, WSAECONNREFUSED, WSAEFAULT, WSAEINVAL, WSAEISCONN, WSAENETUNREACH, WSAEHOSTUNREACH, WSAENOBUFS, WSAENOTSOCK, WSAETIMEDOUT, WSAEWOULDBLOCK, WSAEACCES. Расшифровку ошибок смотри в приложении или в MSDN.

Если сокет блокирующий, то connect блокируется на период установки соединения и возвращаемое значение показывает успех либо ошибку попытки соединения. Для неблокирующего сокета попытка соединения не может быть завершена немедленно и функция возвращает SOCKET_ERROR, а функция WSAGetLastError вернёт ошибку WSAEWOULDBLOCK. Пока на неблокирующем сокете соединение не завершится все последующие вызовы connect будут завершаться с кодом ошибки WSAEALREADY. Если соединение успешно завершится все последующие вызовы connect будут завершаться с кодом ошибки WSAEISCONN. Однако не рекомендуется делать многочисленные вызовы connect для проверки завершения

соединения. В этом случае возможно использовать функцию select для определения завершения попытки соединения путём проверки можно ли в сокет писать.

Если попытка соединения не удалась приложение может снова вызвать connect для того же сокета.

Если сокет не связан с локальным адресом функцией bind, то после вызова connect система связывает сокет, назначая ему локальный адрес и порт.

После успешного завершения функции connect сокет готов для приёма и передачи данных.

Для сокетов не ориентированных на соединение, например типа SOCK_DGRAM, connect просто определяет адрес назначения по умолчанию для дальнейшего использования для обмена данными функций send и recv вместо sendto и recfrom. Любая датаграмма полученная от адреса отличного от адреса назначения будет отброшена. Для установления соединения для сокетов ориентированных на соединение, например типа SOCK_STREAM выполняется трёхшаговое рукопожатие.

Пример использования функции connect

```
SOCKET ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  struct sockaddr_in serverService;
ZeroMemory(serverService, sizeof(serverService));
  serverService.sin_family = AF_INET;
  serverService.sin_addr.s_addr = inet_addr( "123.156.241.13" );
  serverService.sin_port = htons( 27015 );
connect( ConnectSocket, (struct sockaddr *) &servertService, sizeof(serverService) );
```

recv

Функция recv принимает данные от соединённого или связанного сокета. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int recv(SOCKET s, char* buf, int len, int flags);
```

Параметр s ределяет дескрипор сокета, buf – буфер для входящих данных, len – длину буфера в байтах, flags – флаг, определяющий поведение функции, например при значении MSG_PEEK данные копируются в буфер функции recv, но остаются в системном буфере. Для обычного приёма следует использовать 0. Если не произошла ошибка, то функция возвращает число принятых байт, которых может быть меньше, чем len, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAENOTCONN, WSAEINTR, WSAEINPROGRESS, WSAENETRESET, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEINVAL, WSAECONNABORTED, WSAETIMEDOUT, WSAECONNRESET. Расшифровку ошибок смотри в приложении или в MSDN.

Функция читает входящие данные и на сокетах ориентированных на соединение и на сокетах не ориентированных на соединение. В первом случае сокет должен быть предварительно соединён, а во втором связан. Локальный адрес сокета должен быть известен. Для серверных приложений можно использовать явное связывание с помощью функции bind или неявное с помощью accept. Явное связывание не рекомендуется для клиентских приложений. Сокеты на клиенте связываются неявно функциями connect или sendto.

Recv принимает сообщения только от адреса, совпадающего с удалённым адресом сокета s. Остальные сообщения отбрасываются без уведомления.

На сокетах ориентированных на соединение recv принимает все доступные данные, вплоть до размера буфера приёма. На сокетах не ориентированных на соединение данные извлекаются из первого сообщения в очереди. Если данных в сообщении больше, чем размер буфера функции recv, то для протокола UDP оставшиеся данные отбрасываются, а для протокола TCP могут быть забраны следующим вызовом recv. Если сообщение в очереди короче, чем буфер, то ошибок не возникает, recv забирает всё сообщение в буфер и возвращает длину сообщения

Если на блокирующем сокете сокете нет доступных данных, то recv блокируется в ожидании данных. Если на не блокирующем сокете нет доступных данных, то recv не блокируется и возвращается код ошибки WSAEWOULDBLOCK. Следует использовать функцию select для орпеделения наличия доступных для чтения данных.

Если сокет ориентирован на соединение, данных в сети нет и удалённая сторона правильно закрыла соединение, то recv возвратит 0. Если соединение сброшено, то recv установит ошибку WSAECONNRESET.

send

Функция send отсылает данные на соединённый сокет. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int send(SOCKET s, const char* buf, int len, int flags);
```

Параметр s задаёт соединённый сокет, buf - указатель на буфер данных для передачи, len – длина в байтах буфер данных (для UDP сокетов параметр len не может превосходить максимального значения, которое можно получить вызвав getsockopt с парамертом SO_MAX_MSG_SIZE (65507 для windows 2003)), flags в большинсве случаях устанавливают в 0. Если не произошла ошибка, то функция возвращает число переданных байт, которых может быть меньше, чем len, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEACCES, WSAEINTR, WSAEINPROGRESS, WSAEFAULT, WSAENETRESET, WSAENOBUFS, WSAENOTCONN, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEHOSTUNREACH, WSAEINVAL, WSAECONNABORTED, WSAECONNRESET, WSAETIMEDOUT. Расшифровку ошибок смотри в приложении или в MSDN.

Успешное завершение функции не означает, что переданные данные были хотя бы частично приняты. Оно означает лишь, что данные отосланы в сеть. Если транспортная система не может выделить затребованного объёма len буфера buf, send будет заблокирован до тех пор пока сокет не будет пареведен в неблокирующий режим. На неблокирующих TCP сокетах число отправляемых байт может быть между 1 и числом, определяемом возможностями клиентской и серверной операционной системами по выделению буферов. Можно использовать функцию select для определения момента, когда можно отослать больше данных.

Заметим, что отсылать 0 данных допустимо.

recvfrom

Функция recvfrom получает датаграмму и возвращает адрес источника.

```
int recvfrom(SOCKET s, char* buf, int len, int flags, struct sockaddr* from, int* fromlen);
```

Параметр s задаёт связанный сокет, buf - буфер для входящих данных, len - длина буфера в байтах, flags – флаг, определяющий поведение функции, например при значении MSG_PEEK данные копируются в буфер функции recv, но остаются в системном буфере. Для обычного приёма следует использовать 0; from – необязательный указатель на структуру sockaddr в которой будет возвращен адрес источника, fromlen необязательный указатель на размер в байтах адреса источника. Если не произошла ошибка, то функция возвращает число принятых байт, которых может быть меньше len, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEFAULT, WSAEINTR, WSAEINPROGRESS, WSAEINVAL, WSAEISCONN, WSAENETRESET, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAETIMEDOUT, WSAECONNRESET. Расшифровку ошибок смотри в приложении или в MSDN.

Функция recvfrom читает входящие данные как на сокетах, ориентированных на соединение, так и на сокетах, не ориентированных на соединение и захватывает адресс сокета от которого данные были посланы. Обычно эту функцию используют на сокетах, не ориентированных на соединение. Локальный адрес сокета должен быть известен. Для серверных приложений это обычно осуществлякся явным вызовом функции bind. Явное связывание не приветствуется для клиентских приложений и привязка сокета к локальному адресу осуществляется неявно с помощью функции sendto.

На сокетах, ориентированных на поток, таких как SOCK_STREAM, вызов recvfrom возвращает все доступные данные вплоть до указанного размера буфера. На сокетах, ориентированных на соединение параметры from и fromlen игнорируются.

На сокетах, ориентированных на сообщения данные извлекаются из первого сообщения в очереди вплоть до размера указанного буфера. Если сообщение в очереди длиннее, чем буфер, то буфер заполняется начальной частью сообщения, recvfrom генерирует сообщение WSAEMSGSIZE. Для протокола UDP оставшиеся в сообщении данные теряются. Если сообщение в очереди короче, чем буфер, то ошибок не возникает, recvfrom забирает всё сообщение в буфер и возвращает длину сообщения

Если параметр from не равен нулю и сокет не ориентирован на соединение (например SOCK_DGRAM), сетевой адрес отправителя данных копируется в структуру sockaddr, на которую указывает from. Величина на которую указывает fromlen устанавливается в размер этой структуры и при возврате изменяется, показывая истинный размер адреса, сохранённого в структуре sockaddr.

Если на блокирующем сокете нет входящих данных, то recvfrom блокируется и ждёт появления данных. Если сокет не блокирующий, то возвращается ошибка WSAEWOULDBLOCK. Для определения, когда на сокете появятся данные можно использовать функцию select.

Если сокет ориентирован на соединение, данных в сети нет и удалённая сторона правильно закрыла соединение, то recvfrom возвратит 0. Если соединение сброшено, то recv установит ошибку WSAECONNRESET.

```
sockaddr_in R, S;
ZeroMemory(R, sizeof(R));
```

```
ZeroMemory(S, sizeof(S));
char RecvBuf[1024];
int  BufLen = 1024;
int Ssize = sizeof(S);
SOCKET s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
R.sin_family = AF_INET;
R.sin_port = htons(27015);
R.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (SOCKADDR *) &R, sizeof(R));
recvfrom(s,  RecvBuf,  BufLen,  0,  (SOCKADDR *)&S,  & Ssize);
```

sendto

Функция sendto отсылает данные по указанному назначению. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int sendto(SOCKET s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen);
```

Парметр s определяет сокет, клотоый может быть уже и соединён, buf – буфер, содержащий данные для передачи, len – длина буфера в байтах (для UDP сокетов параметр len не может превосходить максимального значения, которое можно получить вызвав getsockopt с парамертом SO_MAX_MSG_SIZE (65507 для 2003)), flags в большинсве случаях устанавливают в 0, to – необязательный указатель на структуру, которая содержит адрес целевого сокета, tolen – размер адреса в байтах. Если не произошла ошибка, то функция возвращает число переданных байт, которых может быть меньше, чем len, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEACCES, WSAEINVAL, WSAEINTR, WSAEINPROGRESS, WSAEFAULT, WSAENETRESET, WSAENOBUFS, WSAENOTCONN, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSAEMSGSIZE, WSAEHOSTUNREACH, WSAECONNABORTED, WSAECONNRESET, WSAEADDRNOTAVAIL, WSAEAFNOSUPPORT, WSAEDESTADDRREQ, WSAENETUNREACH, WSAEHOSTUNREACH, WSAETIMEDOUT Расшифровку ошибок смотри в приложении или в MSDN.

Параметр to может быть любым действительным адресом, включая широковещательные. Если сокет не связан, то при вызове sendto WinSock назначает ему локальный адрес и уникальный порт, осуществляя тем самым неявную привязку сокета. Этот адрес и порт можно узнать с помощью функции getsockname.

Успешный вызов sendto не означает, что данные были успешно доставлены. Обычно sendto используется на сокетах, не ориентированных на соединение для пересылки датаграмм к указанному сокету, определённому в парамере to. Если вызвать sendto на сокете, не ориентированных на соединение, который был предварительно подсоединён к определённому адресу, то параметр to, указанный в вызове sendto перекрывает этот адрес один раз для текущего вызова sendto. На сокете, ориентированных на соединение параметры to и tolen игнорируются и вызов sendto эквивалентен вызову send.

Пример

```
 sockaddr_in R;
ZeroMemory(R, sizeof(R));
 char Buf[1024];
 int BufLen = 1024;
 SOCKET s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
 R.sin_family = AF_INET;
 R.sin_port = htons(27015);
 R.sin_addr.s_addr = inet_addr("123.456.789.1");
 sendto(s,  Buf,  BufLen,  0,  (struct sockaddr*) &R,  sizeof(R));
```

Если транспортная система не может выделить затребованного объёма len буфера buf, sendto будет заблокирован до тех пор пока сокет не будет пареведен в неблокирующий режим. На неблокирующих TCP сокетах число отправляемых байт может быть между 1 и числом, определяемом возможностями клиентской и серверной операционной системами по выделению буферов. Можно использовать функцию select для определения момента, когда можно отослать больше данных.

Заметим, что отсылать 0 данных допустимо.

Примеры сетевых приложений.

Приведены листинги простейших TCP и UDP клиентов и серверов. Исходные тексты вспомогательных функций myprintf и WSAGetLastError приведены выше в листингах 1 и 2.

Пример простого TCP-сервера

Этот пример иллюстрирует как разработать простой TCP сервер, который принимает запрос на соединение от клиента на порту 5150, принимает от него данные и закрывается. Это консольное приложение, которое печатает диагностические сообщения о соединении и приёме данных. Для запуска приложения набирите в коммандной строке tcpserver.exe. Используйте проект tcpserver из решения lab1 из архива src.

Для работы с консольными приложениями рекомендуется использовать файловый менеджер **FAR.**

```
#include <winsock2.h>
#include <stdio.h>
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);

void main(void){
  WSADATA            wsaData;
  SOCKET             ListeningSocket;
  SOCKET             NewConnection;
 struct sockaddr_in        ServerAddr;
 struct sockaddr_in         ClientAddr;
  int               ClientAddrLen;
  int               Port = 5150;
  int               Ret;
  char               DataBuffer[1024];
ZeroMemory(ServerAdd, sizeof(ServerAdd));
ZeroMemory(ClientAddr, sizeof(ClientAddr));

  // Инициализация Winsock версии 2.2

  if ((Ret = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0)
  {
    // Так как Winsock не загрузился мы не можем использовать WSAGetLastError
    // для определения ошибки. Код ошибки возвращает сама функция WSAStartup
    myprintf("WSAStartup ошибка %s\n", encodeWSAGetLastError(Ret));
    return;
  }

  // Создаём новый TCP сокет для приёма запросов на соединение от клиентов.

  if ((ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))
     == INVALID_SOCKET)
  {
    myprintf("ошибка socket %d\n", encodeWSAGetLastError(WSAGetLastError()));
    WSACleanup();
    return;
  }

  // Заполняем  struct sockaddr_in ServerAddr, которая скажет функции bind, что мы хотим
  // слушать соединения на всех интерфейсах (INADDR_ANY), используя порт 5150.
  // Мы преобразуем порядок байт  из системного в сетевой (htons и htonl)

  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(Port);
  ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);

// Функция  bind привязывает адресную информацию, определённую  в ServerAddr к сокету  ListeningSocket

  if (bind(ListeningSocket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr))
     == SOCKET_ERROR)
  {
    myprintf("Ошибка bind %s\n", encodeWSAGetLastError(WSAGetLastError()));
    closesocket(ListeningSocket);
    WSACleanup();
    return;
```

```
        }

// Делаем сокет пасивным для прослушивания (приёма) запросов на TCP-соединение от клиентов.
// Длина очереди запросов  на соединение равна 5
        if (listen(ListeningSocket, 5) == SOCKET_ERROR)
        {
          myprintf("Ошибка listen %s\n", encodeWSAGetLastError(WSAGetLastError()));
          closesocket(ListeningSocket);
          WSACleanup();
          return;
        }
      myprintf("Ожидаем соединение на порту %d.\n", Port);
      ClientAddrLen = sizeof(ClientAddr);
        // Принимаеи новое соединение, когда оно возникнет
        if ((NewConnection = accept(ListeningSocket, (struct sockaddr *) &ClientAddr,
                        &ClientAddrLen)) == INVALID_SOCKET)
        {
          myprintf("Ошибка accept  %s\n", encodeWSAGetLastError(WSAGetLastError()));
          closesocket(ListeningSocket);
          WSACleanup();
          return;
        }


        myprintf("Успешно соединились с %s:%d.\n",
             inet_ntoa(ClientAddr.sin_addr), ntohs(ClientAddr.sin_port));

// Далее мы можем снова ожидать на сокете ListeningSocket новые соединения снова, вызывая accept
// и/или начать передавать и принимать данные на сокете NewConnection.
// ДЛя простоты остановим прослушивание, закрывая сокет ListeningSocket.
//Можно начинать приём и предачу данных на сокете NewConnection.

        closesocket(ListeningSocket);

        // Для простоты ограничимся только приёмом
        myprintf("Ждём данные для получения..\n");

        if ((Ret = recv(NewConnection, DataBuffer, sizeof(DataBuffer), 0))
           == SOCKET_ERROR)
        {
          myprintf("Ошибка recv %s\n", encodeWSAGetLastError(WSAGetLastError()));
          closesocket(NewConnection);
          WSACleanup();
          return;
        }
        // Делаем из полученных данных строку языка С
      DataBuffer[Ret] = '\0';

        myprintf("Успешно получено %d байтов в сообщении  %s.\n", Ret, DataBuffer);
        myprintf("Закрываем соединение с клиентом.\n");
        closesocket(NewConnection);

        // Выгружаем Winsock

        WSACleanup();
       myprintf("Нажмите Enter для завершения.\n");
        getchar();
      }
```

Листинг 6. Пример простого TCP-сервера tcpserver*)

Пример простого TCP-клиента

Этот пример иллюстрирует, как разработать простой TCP клиент, который может послать сообщение "Привет" к TCP серверу, который слушает порт 5150. Эта программа выводит диагностические сообщения о ходе соединения и пересылки данных. Используйте проект tcpclient из решения lab1 из архива src.

Программа запускается из командной строки с указанием IP-адреса сервера:
tcpclient.exe <server IP address>

```
#include <winsock2.h>
#include <stdio.h>
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
void main(int argc, char **argv)
{
  WSADATA            wsaData;
  SOCKET             s;
  struct sockaddr_in         ServerAddr, ca;
  int                Port = 5150;
  int                Ret;
  char *msg = "Привет";
int lmsg = strlen(msg);
  int caSize = sizeof(ca);
ZeroMemory(ServerAdd, sizeof(ServerAdd));
ZeroMemory(ca, sizeof(caSize));



  if (argc <= 1)
  {
    myprintf("Использование: tcpclient <Server IP address>.\n");
    return;
  }

  // Инициализация Winsock версии 2.2

  if ((Ret = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0)
  {
    // Так как Winsock не загрузился мы не можем использовать WSAGetLastError
    // для определения ошибки. Код ошибки возвращает сама функция WSAStartup

    myprintf("WSAStartup ошибка %d\n", encodeWSAGetLastError(Ret));
    return;
  }

  // Создадим новый сокет для соединения этого TCP клиента

  if ((s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))
     == INVALID_SOCKET)
  {
    myprintf("socket ошибка %d\n", encodeWSAGetLastError(WSAGetLastError()));
    WSACleanup();
    return;
  }

  // Заполним в переменной ServerAddr типа  struct sockaddr_in адресную информацию о сервере,
  //к которому мы хотим подключится. Сервер слушает порт 5150. IP-адрес сервера задаётся
  //в командной строке

  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(Port);
  ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);

  // Соединим сокет s с сервером
```

```
    myprintf("Попытка соединения к %s:%d...\n",
          inet_ntoa(ServerAddr.sin_addr), htons(ServerAddr.sin_port));

    if (connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr))
       == SOCKET_ERROR)
    {
      myprintf("Ошибка connect %s\n", encodeWSAGetLastError(WSAGetLastError()));
      closesocket(s);
      WSACleanup();
      return;
    }

    myprintf("Наше соединение успешно.\n");
//  Информацию об истинном адресе и порте локального соединённого сокета sc можно получить так

   getsockname(s, (struct sockaddr *)&ca, &caSize);
   myprintf("Локальный адрес:%s и порт:%d сокета клиента\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));

    // Теперь мы можем обмениваться данными с сервером через сокет  s
    // Мы просто передадим серверу сообщение "Привет"

    myprintf("Попытка передать сообщение Привет.\n");

    if ((Ret = send(s, msg, lmsg, 0)) == SOCKET_ERROR)
    {
      myprintf("Ошибка send %s\n", encodeWSAGetLastError(WSAGetLastError()));
      closesocket(s);
      WSACleanup();
      return;
    }

    myprintf("Успешно передано %d байтов сообщения  %s.\n", Ret, msg);

    myprintf("Закрываем соединение с сервером.\n");

    closesocket(s);

    //  Выгружаем Winsock
    WSACleanup();
     myprintf("Нажмите Enter для завершения.\n");
    getchar();
}
```

Листинг 7. Пример простого TCP-клиента tcpclient *)

Построим клиент и сервер (см. Практическую часть). Вначале запускаем сервер tcpserver, и далее в отдельной консоли запускаем клиент на том же компьютере, что и сервер

tcpclient 127.0.0.1.

В консоли сервера имеем

Ожидаем соединение на порту 5150.
Успешно соединились с 127.0.0.1:1101.
Ждём данные для получения..
Успешно получено 6 байтов в сообщении Привет.
Закрываем соединение с клиентом.
Нажмите Enter для завершения.

В консоли клиента имеем

Попытка соединения к 127.0.0.1:5150...
Наше соединение успешно.
Локальный адрес:127.0.0.1 и порт:1101 сокета клиента

Попытка передать сообщение Привет.
Успешно передано байтов сообщения Привет.
Закрываем соединение с сервером.
Нажмите Enter для завершения.

Пример простого UDP-приёмника

Это пример простого UDP приёмника, ожидающего датаграммы на порту 5150. Это консольное приложение, которое печатает диагностические сообщения о приёме данных. Используйте проект udpreceiver из решения lab1 из архива src. Для запуска приложения набирите в коммандной строке udpreceiver.exe.

```
#include <winsock2.h>
#include <stdio.h>
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
void main(void)
{
  WSADATA            wsaData;
  SOCKET             ReceivingSocket;
  struct sockaddr_in         ReceiverAddr, SenderAddr;
  int                Port = 5151;
  char               ReceiveBuf[1024];
  int                BufLength = 1024;
  int                SenderAddrSize = sizeof(SenderAddr);
  int                Ret;

ZeroMemory(ReceiverAddr, sizeof(ReceiverAddr));
ZeroMemory(SenderAddr, sizeof(SenderAddr));

// Инициализация Winsock версии 2.2

  if ((Ret = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0)
  {
   // Так как Winsock не загрузился мы не можем использовать WSAGetLastError
    // для определения ошибки. Код ошибки возвращает сама функция WSAStartup
    myprintf("Ошибка WSAStartup %s\n", encodeWSAGetLastError(Ret));
    return;
  }


 // Создаём новый сокет для приёма датаграмм

  if ((ReceivingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))
     == INVALID_SOCKET)
  {
    myprintf("Ошибка socket %s\n", encodeWSAGetLastError(WSAGetLastError()));
    WSACleanup();
    return;
  }

  // Заполнение стуктуры  struct sockaddr_in, чтобы уведомить функцию  bind,
  // что мы хотим получать датаграммы со всех интерфейсов, используя порт 5150.

  ReceiverAddr.sin_family = AF_INET;
  ReceiverAddr.sin_port = htons(Port);
  ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

  // Ассоциируем адресную информацию с сокетом

  if (bind(ReceivingSocket, (struct sockaddr *)&ReceiverAddr, sizeof(ReceiverAddr))
     == SOCKET_ERROR)
  {
    myprintf("Ошибка  bind %s\n", encodeWSAGetLastError(WSAGetLastError()));
```

```
    closesocket(ReceivingSocket);
    WSACleanup();
    return;
  }

  myprintf("Мы готовы получить 1 диаграмму с любого интерфейса на порту %d...\n",
       Port);

  // Здесь мы можем получить 1 диаграмму на наш связанный сокет

  if ((Ret = recvfrom(ReceivingSocket, ReceiveBuf, BufLength, 0,
     (struct sockaddr *)&SenderAddr, &SenderAddrSize)) == SOCKET_ERROR)
  {
    myprintf("Ошибка  recvfrom %s\n", encodeWSAGetLastError(WSAGetLastError()));
    closesocket(ReceivingSocket);
    WSACleanup();
    return;
  }
ReceiveBuf[Ret] = '\0';
  myprintf("Мы успешно получили %d байт сообщения %s от адреса %s:%d.\n", Ret, ReceiveBuf,
       inet_ntoa(SenderAddr.sin_addr), ntohs(SenderAddr.sin_port));


  // По окончании приёма датаграммы закроем сокет

  closesocket(ReceivingSocket);

//  Выгружаем Winsock

  WSACleanup();
myprintf("Нажмите Enter для завершения.\n");
getchar();

}
```

Листинг 8. Пример простого UDP приёмника udpreceiver

Пример простого UDP передатчика

Это пример простого UDP передатчика, отсылающего сообщение UDP приёмнику, ожидающему датаграммы на порту 5150. Это консольное приложение, которое печатает диагностические сообщения о передаче данных. Используйте проект udpsender из решения lab1 из архива src. Для запуска приложения набирите в коммандной строке udpsender.exe <receiver IP address>

```
#include <winsock2.h>
#include <stdio.h>
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);

void main(int argc, char **argv)
{
  WSADATA            wsaData;
  SOCKET             SendingSocket;
 struct sockaddr_in         ReceiverAddr;
  int                Port = 5151;
  int                Ret;
  char *msg = "Привет";
int lmsg = strlen(msg);
ZeroMemory(ReceiverAddr, sizeof(ReceiverAddr));
  if (argc <= 1)
  {
    myprintf("Использование: udpsender <receiver IP address>.\n");
    return;
  }

// Инициализация Winsock версии 2.2
```

```
        if ((Ret = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0)
        {
          // Так как Winsock не загрузился мы не можем использовать WSAGetLastError
          // для определения ошибки. Код ошибки возвращает сама функция WSAStartup

          myprintf("Ошибка WSAStartup %s\n", encodeWSAGetLastError(Ret));
          return;
        }

        // Создаём новый сокет для отсылки датаграмм

        if ((SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))
           == INVALID_SOCKET)
        {
          myprintf("Ошибка socket%s\n", encodeWSAGetLastError(WSAGetLastError()));
          WSACleanup();
          return;
        }
// Заполним в переменной  ReceiverAddr типа  struct sockaddr_in адресную информацию о
//получателе дптпграмм. Получатель ждёт датаграммы на порту 5150. IP-адрес получателя задаётся
        //в командной строке

        ReceiverAddr.sin_family = AF_INET;
        ReceiverAddr.sin_port = htons(Port);
        ReceiverAddr.sin_addr.s_addr = inet_addr(argv[1]);

        // Отсылка датаграммы  получателю

        if ((Ret = sendto(SendingSocket,msg , lmsg, 0,
           (struct sockaddr *)&ReceiverAddr, sizeof(ReceiverAddr))) == SOCKET_ERROR)
        {
          myprintf("Ошибка sendto %s\n", encodeWSAGetLastError(WSAGetLastError()));
          closesocket(SendingSocket);
          WSACleanup();
          return;
        }

        // По окончании отсылки датаграммы приложение закрывает сокет

        myprintf("Успешно отослано %d байт сообщения  %s к %s:%d.\n", Ret, msg,
             inet_ntoa(ReceiverAddr.sin_addr), htons(ReceiverAddr.sin_port));

        closesocket(SendingSocket);

      //  Выгружаем Winsock

        WSACleanup();
      myprintf("Нажмите Enter для завершения.\n");
      getchar();
      }
```

Листинг 9. Пример простого UDP передатчика udpsender

Вначале запускаем UDP приёмник udpreceiver, и далее в отдельной консоли запускаем UDP передатчик udpsender 127.0.0.1. В консоли приёмника имеем

```
Мы готовы получить 1 диаграмму с любого интерфейса на порту 5151...
Мы успешно получили 6 байт сообщения Привет от адреса 127.0.0.1:2605.
Нажмите Enter для завершения.
```

В консоли передатчика имеем

Успешно отослано 6 байт сообщения Привет к 127.0.0.1:5151.
Нажмите Enter для завершения.

Практическая часть
Работа в Visual Studio 2008
Построение проектов

Откроем Visual Studio 2008.

1. File->New->project
2. В окне New project выберем Other Project Types->Visual Studio Solutions->Blank Solution
3. В поле Name зададим имя Solution, например Lab1
4. Используя кнопку Browse, зададим папку для Solution
5. Нажмём Ok. Solution Lab1 появится в Solution Explorer
6. Нажмём правую кнопку мыши на имени нашего Solution Lab1 и выберем Add-> New Project
7. Выберем Visual C++->Win32->Console Application
8. В поле Name зададим имя нашему проекту, например TcpServer
9. Проконтролируем в поле Location с помощью кнопки Browse, чтобы проект находился в папке нашего Solution Lab1
10. Нажав Ok, попадём в Win32 Application Wizard. Нажмём в нём Next
11. Проверим, чтобы радиокнопки Application Type, указывали на Console Application
12. Уберём галочку в Precompiled headers
13. Установим галочку в Empty Project и нажмём Finish.
14. Проект TcpServer появится в в Solution Explorer
15. Нажмём правую кнопку мыши на имени нашего проекта TcpServer -> Source Files и выберите Add->New Item
16. В окне Add New Item выберите Visual C++->Code->C++ file
17. В поле Name введите имя файла с расширением .c (не .cpp), например TcpServer.c
18. Проконтролируем в поле Location с помощью кнопки Browse, чтобы TcpServer.c находился в папке нашего проекта TcpServer
19. Введём в окно TcpServer.c текст программы, например, скопировав его из текста лабораторных работ (Листинг 6.)
20. Нажмём правую кнопку мыши на имени нашего проекта TcpServer и выберите Property
21. В окне Property Pages выберите Configuration Properties->General->Caracter Set-> Use Multi-Byte Character Set
22. Если приложение требует аргументов командной строки, как, например TcpClient или UDPReceiver, то в окне Property Pages выберите Configuration Properties->Debugging->Command Arguments и введите аргументы командной строки, например 127.0.0.1
23. Задайте проекту режим компоновки с библиотеками Winsock: в окне Property Pages выберите Configuration Properties->Linker->Input->Additional Dependendencies и введите имена Winsock библиотек: ws2_32.lib mswsock.lib
24. По умолчанию установлен режим отладки Debug Win32, проверьте
25. Нажмём правую кнопку мыши на имени нашего проекта TcpServer и выберите Build
26. Исправьте ошибки. Должно быть сообщение об успешном построении проекта
27. Повторим шаги 6 – 26 для других проектов текущего Solution. Так для Lab1 добавим проекты TcpClient, UDPReceiver, UDPSender из листингов 7, 8 и 9.
28. Можно построить или перестроить все проекты Solution Lab1, нажав правой кнопкой на имени Solution Lab1 и выбрав build Solution или rebuild Solution

Отладка проектов

Перед началом работы с помощью команда netstat –an убедитесь, что используемый порт 5150 не занят.

В начале проект надо сделать активным в Solution, нажав на его имени правую кнопку и выбрав Set as StartUp project. Клавиша F10 задаёт пошаговое выполнение без захода в вызываемые функции, а клавиша F11 задаёт пошаговое выполнение с заходом в вызываемые функции. Щелчок мыши левее оператора устанавливает/снимает контрольную точку. Клавиша F5 выполняет программу до следующей контрольной точки, или до конца. Подведя курсор мыши к переменной, можно увидеть её значение. Это можно сделать и с помощью пунктов контекстного меню add watch и quick watch. Отладка останавливается выбором пункта меню debugging->stop debugging.

Запустить программу (.exe) из командной строки можно либо набрав её имя с аргументами через старт->пуск, либо запустив командный процессор cmd.rxe, либо запустив из менеджера файлов типа Far. Исполняемые файлы находятся в подпапке debug или release папки Solution Lab1. Важно отметить

необходимость наличия в конце программ макроса getchar, без него консольное окно программы после завершения программы просто исчезнет.

Программы работают только в парах: пара TcpClient, TcpServer и пара UDPReceiver, UDPSender. Поэтому целесообразно запустить два экземпляра программ из пары. Можно обе программы запускать из командной строки, можно одну запускать в командной строке, а вторую в режиме отладки в Visual Studio 2008 и можно обе программы запускать в режиме оладки в Visual Studio 2008. Рассмотрим последний вариант более подробно для пары TcpClient, TcpServer.

Клиент и сервер TCP

Запустим Visual Studio 2008 с активным проектом TcpServer (не забудьте Set as StartUp project) и установим контрольные точки на вызовах функций accept и recv. Запустим второй экцемпляр Visual Studio 2008 с активным проектом TcpClient. Игнорируем предупреждающее сообщение. Установим контрольные точки на вызовах функций connect и send. Не забудьте указать аргумент командной строки 127.0.0.1.

Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpClient и нажмём F5. Программа остановится на установленной контрольной точке на функции connect. Подведя курсор к переменным s, msg, Port, Ret, lmsg, caSize, argv[1], посмотрите их значения. Обратите внимание на значения полей структуры ServerAddr. В консольном окне появится

Попытка соединения к 127.0.0.1:5150

Нажимая F11 или F10, дойдите до функции closesocket(s). В консольном окне появится

Ошибка connect WSAECONNREFUSED: В соединении отказано

(Предполагается, что вы завершили написание функции расшифровки ошибок encodeWSAGetLastError из листинга 2)

Завершите отладку. Всё логично. Мы попытались соединиться к незапущенному серверу.

Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpServer и нажмём F5. Программа остановится на установленной контрольной точке на функции accept. Подведя курсор к различным переменным, посмотрите их значения. В консольном окне появится

Ожидаем соединение на порту 5150.

Нажмём F10. Приложение TcpServer заблокировалось на функции accept, ожидая запроса от клиента на соединение. Заголовок окна Visual Studio 2008 поменялся от Debugging до Running.

Команда netstat –an, выдаст информацию, в которой есть строка

```
TCP    0.0.0.0:5150           0.0.0.0:0              LISTENING
```

Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpClient и нажмём F5. Программа остановится на установленной контрольной точке на функции connect. Нажмём F5. Программа остановится на установленной контрольной точке на функции send. Подведя курсор к переменной ca клиентского адреса сокета s, посмотрите её значения. В консольном окне появятся дополнительные строки

Наше соединение успешно.
Локальный адрес:127.0.0.1 и порт:2215 сокета клиента
Попытка передать сообщение Привет.

К вас естественно будет номер порта отличный от 2215. Команда netstat –an, выдаст информацию, в которой есть строки

```
TCP    127.0.0.1:2215 127.0.0.1:5150       ESTABLISHED
TCP    127.0.0.1:5150 127.0.0.1:2215       ESTABLISHED
```

То есть установлено двустороннее полнодуплексное TCP соединение. Первая строка соответствует клиенту, а вторая серверу.

Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpServer. Программа разблокировалась на вызове accept. Заголовок окна Visual Studio 2008 поменялся от Running до Debugging. Вычисления остановились на операторе myprintf. Нажав F5 мы переёдём к следующей контрольной точке на функции recv. Подведя курсор к переменной NewConnection нового сокета и к переменной адреса

подсоединившегося сокета ClientAddr, посмотрите их значения. В консольном окне появятся дополнительные строки

Успешно соединились с 127.0.0.1:2215.
Ждём данные для получения..

Обратите внимание на порт 2215 – он одинаков и для клиента и для сервера. Этот порт система выделила динамически. У вас будет другой порт. TCP соединение характеризуется четвёркой

IP-адрес источника:порт источника, IP-адрес приёмника:порт приёмника плюс информация о качестве обслуживания QoS (quality of service), которое здесь не рассматривается. В нашем случае четвёрка, определяющая соединение имеет вид

127.0.0.1:2215, 127.0.0.1:5150

Далее можно поступать двумя способами, по двум сценариям.

Сценарий 1. Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpClient. Программа остановлена на установленной контрольной точке на функции send. Нажмём F5. Программа с помощью функции send передаст данные по направлению к серверу и заблокируется на функции getchar, ожидая от пользователя нажатия клавиши Enter. Заметим, что сервер ещё не вызвал функцию приёма recv, то есть данные отправились не в функцию recv сервера непосредственно, а в сеть по пути в эту функцию. Заметим, что объём таких данных, которые некому непосредственно принять в данный конкретный момент теорертически неограничен и определяется лишь ресурсами компьютеров, участвующими в обмене данными. В консольном окне появятся дополнительные строки

Успешно передано 6 байтов сообщения Привет.
Закрываем соединение с сервером.
Нажмите Enter для завершения.

Данные переданы, они в сети. Клиент можно даже остановить. Сделайте это, завершив отладку, нажав Enter

Команда netstat –an, выдаст информацию, в которой есть строки

```
TCP    127.0.0.1:2215         127.0.0.1:5150         FIN_WAIT_2
TCP    127.0.0.1:5150         127.0.0.1:2215         CLOSE_WAIT
```

То есть соединения начинают закрываться.

Через несколько минут netstat –an, выдаст информацию

```
  TCP    127.0.0.1:5150         127.0.0.1:2215         CLOSE_WAIT
```

Видим, что клиентский сокет закрылся. Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpServer. Программа остановлена на установленной контрольной точке на функции recv. Нажмём F5. На этот момент клиента не существует и его сокет закрыт. Программа с помощью функции recv заберёт данные из сети и заблокируется на функции getchar, ожидая от пользователя нажатия клавиши Enter. Заметим, что данные сервер может забрать из сети через весьма длительное время после смерти клиента. В консольном окне появятся дополнительные строки

Успешно получено 6 байтов в сообщении Привет.
Закрываем соединение с клиентом.
Нажмите Enter для завершения.

Команда netstat –an, не выдаст информации о наших сокетах – они закрылись.

Сценарий 2. Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpServer. Программа остановлена на установленной контрольной точке на функции recv. Нажмём F5. Программа блокируется на вызове recv, ожидая данные из сети.

Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpClient. Программа остановлена на установленной контрольной точке на функции send. Нажмём F5. Программа с помощью функции send

передаст данные серверу и заблокируется на функции getchar, ожидая от пользователя нажатия клавиши Enter. В консольном окне появятся дополнительные строки

Успешно передано 6 байтов сообщения Привет.
Закрываем соединение с сервером.
Нажмите Enter для завершения.

Перейдём к экземпляру Visual Studio 2008 с активным проектом TcpServer. Программа прияла данные из сети, разблокировалась и остановилась на следующем операторе. Нажмём F5. Программа заблокируется на функции getchar, ожидая от пользователя нажатия клавиши Enter. В консольном окне появятся дополнительные строки

Успешно получено 6 байтов в сообщении Привет.
Закрываем соединение с клиентом.
Нажмите Enter для завершения.

Закроем все экземляры Visual Studio 2008.

UDP

Рассморим теперь пару проектов для UDP: UDPsender и UDPreceiver. Откроем каждый из них в отдельном экземпляре Visual Studio 2008. Установите в UDPreceiver контрольную точку на вызове recvfrom, а в UDP sender на вызове sendto.

Запустите UDPsender. Он остановится на установленной контрольной точке на функции sendto . Нажмите F5. UDPsender успешно отправит датаграмму с помощью вызова sendto и заблокируется на функции getchar, ожидая ввода с клавиатуры Enter. В консольном окне увидим

Успешно отослано 6 байт сообщения Привет к 127.0.0.1:5151.
Нажмите Enter для завершения.

Запустите UDPreceiver. Он остановится на установленной контрольной точке на функции recfrom. Нажмите F5. UDPreceiver заблокируется на функции recfrom. То есть датаграмма посланная UDPsender потеряется. Как вы помните, что для случая TCP после установки соединения данные не терялись. В консольном окне увидим

Мы готовы получить 1 диаграмму с любого интерфейса на порту 5151...

Netstat –an покажет, что порт UDP 5150 открыт

UDP 0.0.0.0:5150 *:*

Откроем экземпляр Visual Studio 2008 с проектом UDPsender. Уберём в нём контрольные точки и запустим на выполнение.

Откроем экземпляр Visual Studio 2008 с проектом UDPreceiver. Он разблокируется из функции recfrom и заблокируется на функции getchar, ожидая ввода с клавиатуры Enter. В консольном окне увидим дополнительные строки

Мы успешно получили 6 байт сообщения Привет от адреса 127.0.0.1:2605.
Нажмите Enter для завершения.

Использование функций обвёрок

Листинги наших программ изобилуют диагностическими сообщениями. И это правильно. Однако мешает чтению программ. Вынесем диагностический код, окружающий вызовы функций Winsock в отдельные файлы. Тогда, например для проекта UDP передатчика получим новый проект (назовём его udpsenderWrapd – см. решение lab1 из архива src) в котором главная программа (назовём его udpsenderWrapd.c) имеет значительно более читаемый вид

```
#include <winsock2.h>
#include <stdio.h>
#include "../wrap.h"

void main(int argc, char **argv)
{
SOCKET              SendingSocket;
```

```
struct sockaddr_in         ReceiverAddr;
int                Port = 5151;
int                Ret;
char *msg = "Привет";
int lmsg = strlen(msg);
ZeroMemory(ReceiverAddr, sizeof(ReceiverAddr));
if (argc <= 1){
     myprintf("Использование: udpsender <receiver IP address>.\n");
     return;
   }
// Инициализация Winsock версии 2.2
myWSAStartup();
// Создаём новый сокет для отсылки датаграмм
SendingSocket = UDPsocket();
// Заполним в переменной  ReceiverAddr типа  struct sockaddr_in адресную информацию о
//получателе датаграмм. Получатель ждёт датаграммы на порту 5150. IP-адрес получателя задаётся
//в командной строке
ReceiverAddr.sin_family = AF_INET;
ReceiverAddr.sin_port = htons(Port);
ReceiverAddr.sin_addr.s_addr = inet_addr(argv[1]);
// Отсылка датаграммы  получателю
Ret = Sendto(SendingSocket,msg , lmsg, 0, (struct sockaddr *)&ReceiverAddr, sizeof(ReceiverAddr));
myprintf("Успешно отослано %d байт сообщения  %s к %s:%d.\n", Ret, msg,
        inet_ntoa(ReceiverAddr.sin_addr), htons(ReceiverAddr.sin_port));
// По окончании отсылки датаграммы приложение закрывает сокет
closesocket(SendingSocket);
//  Выгружаем Winsock
WSACleanup();
myprintf("Нажмите Enter для завершения.\n");
getchar();
}
```

Листинг 10. Пример простого UDP передатчика с использоваанием фунций обвёрток

Новые функции оболочки myWSAStartup, UDPsocket и Sendto помещены в файл wrap.c

```
#include <winsock2.h>
#include <stdio.h>
#include "wrap.h"
void myWSAStartup(){
WSADATA wsaData;
int Ret;
if ((Ret = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0) {
// Так как Winsock не загрузился мы не можем использовать WSAGetLastError
// для определения ошибки. Код ошибки возвращает сама функция WSAStartup
     myprintf("Ошибка WSAStartup %s\n", encodeWSAGetLastError(Ret));
     return;
   }
 }

SOCKET UDPsocket(){
SOCKET SendingSocket;
if ((SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))
      == INVALID_SOCKET)
   {
     myprintf("Ошибка socket%s\n", encodeWSAGetLastError(WSAGetLastError()));
     WSACleanup();
     return INVALID_SOCKET;
   }
  else return SendingSocket;
}

int Sendto(SOCKET s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen){
 int ret;
```

```
 if ((ret=sendto(s ,buf , len, 0, to, tolen)) == SOCKET_ERROR)
   {
     myprintf("Ошибка sendto %s\n", encodeWSAGetLastError(WSAGetLastError()));
     closesocket(s);
     WSACleanup();
   }
 return ret;
  }
```

Листинг 11. Функции обвёртки.

Заголовочный файл "wrap.h" содержит прототипы наших функций

```
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
SOCKET UDPsocket(void);
int Sendto(SOCKET s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen);
void myWSAStartup(void);
```

Листинг 12. **Заголовочный файл**, **содержащий прототипы функции обвёртки.**

При написании оболочек обязательно используйте прототипы обворачиваемых функций, приведенные выше в данной лабораторнрй работе. Заметьте, что наша функция Sendto имеет тот же прототип, что и обворачиваемая функция Winsock sendto. Хотя функция UDPsocket и myWSAStartup не следуют никаким прототипам. Следует так разрабатывать оболочки, чтобы они были максимально похожи на оригинальные функции и с лёгкостью могли бы быть использованы в следующих проектах следующих лабораторных работ.

Иллюстрация полного дуплекса

TCP-сервер (листинг 13), приняв соединение от TCP-клиента (листинг 14), сразу с помощью функции writen отсылает ему порцию из DATA_BUFSIZE байт. Затем сервер с помощью функции readn, читает данные, переданные клиентом.

TCP-клиент, соединившись с TCP-сервером, с помощью функции writen отсылает ему порцию из DATA_BUFSIZE байт. Затем клиент с помощью функции readn, читает данные, переданные сервером.

Таким образом, сразу после соединения одновременно существуют два потока байт: от клиента к серверу и от сервера к клиенту.

Загрузите в Visual Studio проекты duplexServer и duplexClient из решения lab1 архива. Изменим программу writen, раскомментировав в неё строки

```
myprintf("Передано %d байт. Нажмите Enter.\n",nwritten);
getchar();
```

Изменим программу readn, раскомментировав в неё строки

```
myprintf("Принято %d байт. Нажмите Enter.\n",rc);
getchar();
```

Создайте исполняемые файлы duplexServer.exe и duplexClient.exe. Запустите на выполнение такой командный файл duplex.cmd

```
start duplexServer.exe
start duplexClient.exe <IP-адрес duplexServer>
```

Вы увидите консольные окна клиента и сервера. Нажимая в этих окна клавишу Enter в произвольном порядке, вы увидите нечто подобное представленному на рисунке 1.

```
#include <winsock2.h>
#include "../duplex.h"
void main(void){
WSADATA             wsaData;
SOCKET              ListeningSocket, NewConnection;
struct sockaddr_in        ServerAddr,  ClientAddr;
int ClientAddrLen, Port = 5150,  Ret,i ;
int *msgForR, *msgForW;

WSAStartup(MAKEWORD(2,2), &wsaData);
ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
```

```
ServerAddr.sin_port = htons(Port);
ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
Bind(ListeningSocket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr);
Listen(ListeningSocket, 5);
NewConnection = accept(ListeningSocket, (struct sockaddr *) &ClientAddr, &ClientAddrLen);
msgForW = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
msgForR = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
for(i=0;i<DATA_BUFSIZE_INT;i++)        msgForW[i]=i;//rand();
writen(NewConnection , msgForW , DATA_BUFSIZE);
readn(NewConnection , msgForR , DATA_BUFSIZE);
closesocket(ListeningSocket);
closesocket(NewConnection);
WSACleanup();
myprintf("Нажмите Enter для завершения.\n");
getchar();
}
```

Листинг 13. **Сервер полного дуплекса**

```
#include "../duplex.h"
void main(int argc, char **argv){
WSADATA             wsaData;
SOCKET              s;
struct sockaddr_in          ServerAddr;
int                 Port = 5150,  Ret, i;
int *msgForR, *msgForW;
if (argc <= 1)   {
myprintf("Использование: tcpclient <Server IP address> .\n");
return;
}
WSAStartup(MAKEWORD(2,2), &wsaData);
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(Port);
ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr);
msgForW = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
msgForR = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
for(i=0;i<DATA_BUFSIZE_INT;i++)        msgForW[i]=i;//rand();
writen(s , msgForW , DATA_BUFSIZE);
readn(s , msgForR , DATA_BUFSIZE);
closesocket(s);
WSACleanup();
myprintf("Нажмите Enter (ы) для завершения.\n");
getchar();
}
```

Листинг 14. **Клиент полного дуплекса**

```
#include <winsock2.h>
#include "../../common/DATA_BUFSIZE.h"
int myprintf (const char *lpFormat, ... );
int readn( SOCKET fd, char *bp, int len);
int writen( SOCKET fd, char *bp, int len);
```

Листинг 15. **Заголовочный файл для программ из листингов 13 и 14**

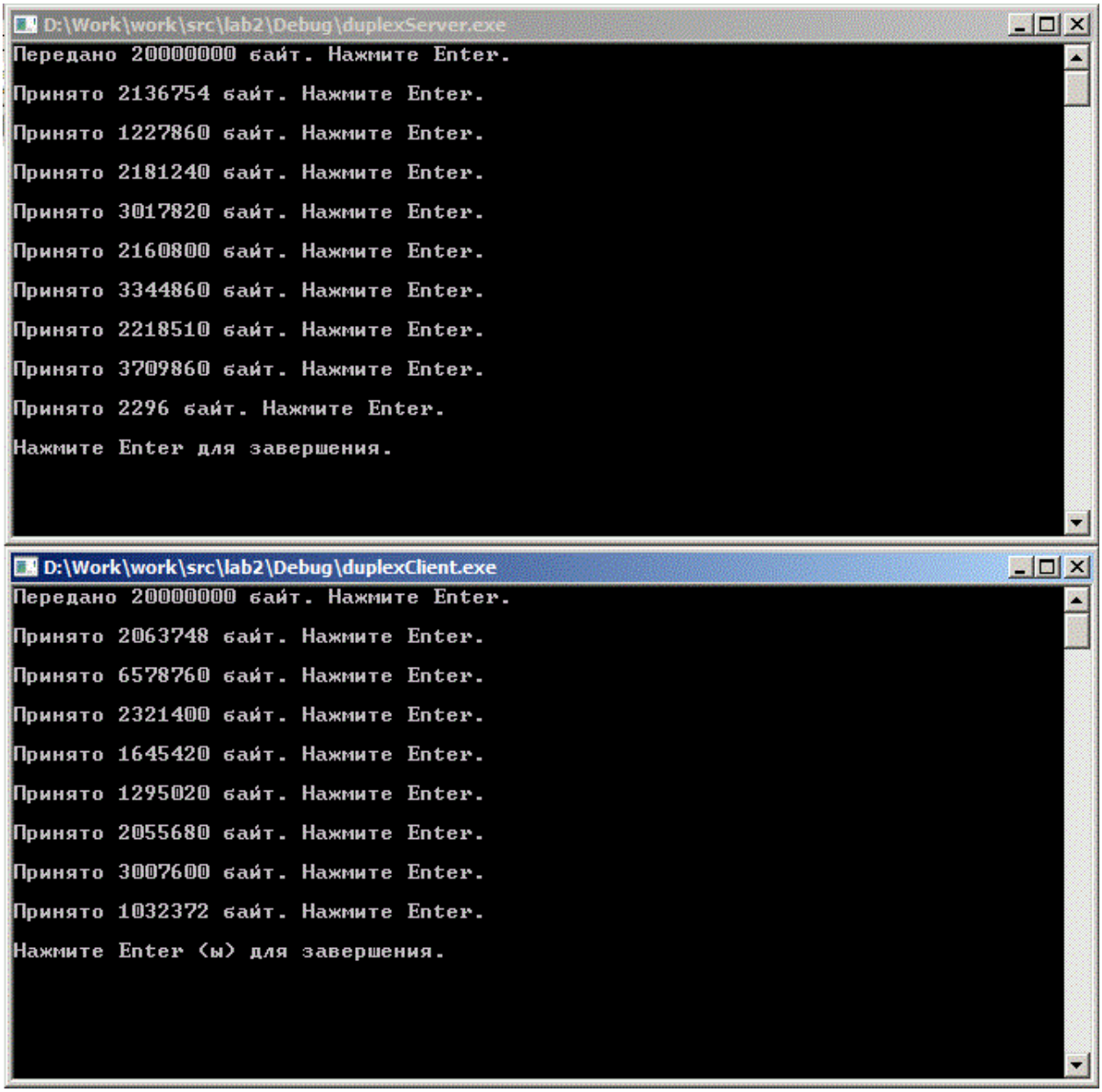

Рисунок 1. Иллюстрация полного дуплекса.

Из рисунка и текста программ мы видим, что в начале работы и клиенту и серверу удалось одновременно вывести в сеть все DATA_BUFSIZE=20000000 байт в сторону друг друга. Затем и клиент и сервер постепенно выбирают из сети предназначенную им информацию.

Контрольные вопросы

1. Какие вы знаете сетевые протоколы?
2. Что такое Winsock и зачем он нужен.
3. Что такое протокол называют ориентированным на передачу сообщений? Приведите пример.
4. Что такое протокол называют ориентированным на поток? Приведите пример.
5. Что такое протокол, требующий установления логического соединения? Приведите пример.
6. Что такое протокол, не требующий установления логического соединения? Приведите пример.
7. Почему здесь вместо стандартной функции printf разработана функция myprintf?
8. Как узнать с какой ошибкой завершился вызов функции Winsock?
9. Может ли быть первой в прогамме любая функция Winsock?
10. Что такое сокет и как его создать?
11. Чем отличается сокет Winsock от сокета UNIX/LINUX и во что выливается это отличие?
12. Какие два тиав сокетов вы можете создать и как?
13. Зачем в Winsock для IP-адресации используется две структуры sockaddr_in и sockaddr?
14. Что такое порт?

15. Какие поля есть в структурах sockaddr_in и sockaddr?
16. Как преобразовать IP-адрес из точечной нотации в 32-битное длинное целое без знака и обратно?
17. Какую роль играет порядок байт в сетефом программтровании?
18. Как преобразовать многобайтные числа из сетевого порядка в системный и обратно?
19. Сколько IP-адресов соответствует имени узла и как их узнать?
20. Что такое связанный и несвязанный сокет?
21. Как явно связать сокет?
22. Какие функции связывают сокет неявно?
23. Как перевести сокет в пассивный режим для приёма запросов на соединение?
24. Что обозначает второй параметр функции listen?
25. Какая функция принимает запросы на соединение?
26. Какая функция отсылает запросы на соединение?
27. В чём отличие функционирования функции connect для TCP и UDP?
28. Что такое трехэтапное рукопожатие?
29. Как узнать адрес и порт локального и удалённого сокета?
30. Что такое соединенный сокет? Когда он таковым становится?
31. Какие функции служат для передачи информации через сокет?
32. Почему на клиентских сокетах предпочтительнее неявное связывание?
33. как узнать адрес и порт сокета после его неявного связывания?
34. Когда на клиентских сокетах предпочтительнее явное связывание?
35. Можно ли сбросить на сервере очередь запросов на соединение к серверу?
36. Что возвращает функция accept?
37. К чему приведёт повторный вызов функции accept?
38. Чем различаются бокирующие и неблокирующие сокеты (на примере функций accept, connect, recv, send)?
39. Что возвращает функция recv?
40. Как ведёт себя функция recv призакрвтии удалённого соединения удалённой стороной?
41. Что возвращает функция send?
42. Что можно сказать о судьбе тех данных, которые отосланы функцией send?
43. В чём различие между функциями send и sendto, recv и recvfrom?
44. Что возвращает функция recvfrom?
45. Как ведёт себя функция recv и recvfrom на сокетах, ориентированных на сообщения, если размер её буфера не совпадает с размером сообщения в очереди?
46. Как ведёт себя функция sendto на соединённом сокете?

Задания

1. Выполните практическую часть работы
2. Завершите написания функции расшифровки кодов ошибок (листинг 2)
3. Измените программу из Листинг 4, так, чтобы имя хоста задавалось в командной строке.
4. Измените прогамму TCP-клиента (листинг 7) *) и UDP-передатчика (листинг 9) *), так, чтобы они понимали не только IP-адреса, но и имена. Перепешите программы TCP-сервера (листинг 6), TCP-клиента (листинг 7) *), UDP-приёмника (листинг 8) *), UDP-передатчика (листинг 8) *) с использованием функий обвёрток. См. Листинги 10, 11 и 12. Запустите программы на разных машинах.
5. Измените прогамму TCP-клиента *) (листинг 7) и UDP-передатчика *) (листинг 9), так, чтобы они обменивались не строками, а бинарными данными, например типа double.
6. Разработайте программы UDP эхо сервера и клиента. Клиент принимает с клавиатуры строку и передаёт её серверу. Тот отсылает её обратно клиенту. Клиент выводит её на консоль. Если сервер получил строку «qs», то он закрывается. Если вы ввели строку «qc», то закрывается только клиент. Адрес или имя сервера задаётся в командной строке клиента. Запустите программы на разных машинах. Шаблон UDP сервера, который принимает датаграмму и отправляет её обратно клиенту:

```
struct sockaddr_in        RA, SA;
int               Port = 5151;
char               Buf[1024];
int               BufLength = 1024;
int               SASize = sizeof(SA);
int               Ret;
SOCKET            s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP));
ZeroMemory(RA , sizeof(RA));
```

```
ZeroMemory(SA, SASize);
RA.sin_family = AF_INET;
RA.sin_port = htons(Port);
RA.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (struct sockaddr *)&RA, sizeof(RA));
while(1){
Ret = recvfrom(s, Buf, BufLength, 0,  (struct sockaddr *)&SA, &SASize);
Buf[Ret] = '\0';
sendto(s, Buf , strlen(Buf), 0, (struct sockaddr *)&SA, SASize);
}
```

Шаблон UDP клиента, который отсылает датаграмму UDP серверу и выводит его ответ в консоль:

```
struct sockaddr_in        RA;
int               Port = 5151;
int               Ret;
char *msg;
char               ReceiveBuf[1024], ansiBuffer[1024];
int               BufLength = 1024;
SOCKET  s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
ZeroMemory(RA , sizeof(RA));
RA.sin_family = AF_INET;
RA.sin_port = htons(Port);
RA.sin_addr.s_addr = inet_addr(argv[1]);
while(1){
msg = gets(ReceiveBuf);
sendto(s,msg , strlen(msg), 0, (struct sockaddr *)&RA, sizeof(RA));
//bind не надо, так как сокет s неявно был связан в функции sendto
recvfrom(s, ReceiveBuf, BufLength, 0, NULL, NULL);
ReceiveBuf[Ret] = '\0';
OemToChar(ReceiveBuf,ansiBuffer);
myprintf("Принимаем отосланное обратно %s\n", ansiBuffer);}
```

Если в UDP клиенте указывается в качестве адреса сервера локальный адрес, то при отсутствии серверов клиент разговаривает сам с собой. Чтобы убедится, что клиент разговаривает с сервером, добавьте на сервере какой-то префикс к возвращаемому сообщению. Заметим, что в отличие от TCP (п. 6) можно запустить несколько UDP эхо клиентов одного сервера. Покажите это.

Самостоятельно разработайте программы TCP эхо сервера и клиента. Клиент принимает с клавиатуры строку и передаёт её серверу. Тот отсылает её обратно клиенту. Клиент выводит её на консоль. Если сервер получил строку «qs», то он закрывается. Если вы ввели строку «qc», то закрывается только клиент. Адрес или имя сервера задаётся в командной строке клиента. Запустите программы на разных машинах. Заметим, что в отличие от UDP (п. 5) можно запустить только один TCP эхо клиент одного сервера. Покажите это.

7. Доработайте UDP сервер, чтобы он понимал команды от клиента и отсылал бы ему результат выполнения. Запустите клиента и сервер на разных машинах. Пример таких команд:

Сложить 6 3 – получаем от сервера 9.
Вычесть 6 3 – получаем от сервера 3.
Умножить 6 3 - получаем от сервера 18
Делить 6 3 - получаем от сервера 2
Qc - закрывается только клиент.

8. Аналогично 7, только для TCP.

Лабораторная работа №2. Параллельные сервера TCP

Целью этой работы служит предоставление материала необходимого для понимания различных способов построения параллельных серверов TCP, то есть таких серверов, которые допускают параллельную, одновременную работу с несколькими TCP-клиентами.

TCP: протокол управления передачей

Клиент TCP устанавливает соединение с сервером, обменивается с ним данными по этому соединению и затем разрывает соединение. TCP также обеспечивает надежность. Когда TCP отправляет данные с порядковым номером s на другой конец связи, он требует, чтобы в случае их получения было выслано подтверждение ACK s, где s порядковый номер данных, которые подтверждаются. (При возможности TCP помещает подтверждения в пакеты с данными). Если подтверждение не приходит через некоторый промежуток времени, TCP автоматически передает данные повторно и увеличивает время ожидания подтверждения . После некоторого количества повторных передач TCP оставляет попытки отправить эти данные. В среднем суммарное время попыток отправки данных занимает от 4 до 10 минут (в зависимости от реализации). TCP содержит алгоритмы, позволяющие динамически прогнозировать время (период) обращения (round-trip time, RTT) пакетов между клиентом и сервером, и таким образом узнавать, сколько времени необходимо для получения подтверждения. Например, RTT в локальной сети может иметь значение порядка миллисекунд, в то время как для глобальной сети эта величина может достигать нескольких секунд. Более того, в определенный момент времени TCP может получить значение RTT между данными клиентом и сервером, равное одной секунде, а затем через 30 секунд измерить RTT на том же соединении и получить значение, равное 2 секундам, что объясняется различиями сетевого трафика.

TCP также упорядочивает данные, связывая некоторый порядковый номер с каждым отправляемым сегментом. Предположим, например, что приложение записывает 2048 байт в сокет TCP, что приводит к отправке двух сегментов TCP. (Сегмент (segment) — это блок данных, передаваемых протоколом TCP протоколу IP.) Если какой-либо сегмент приходит вне очереди (то есть если нарушается последовательность сегментов), принимающий TCP заново упорядочит сегменты, основываясь на их порядковых номерах, прежде чем отправить данные принимающему приложению. Если TCP получает дублированные данные (допустим, компьютер на другом конце ошибочно решил, что сегмент потерян, и передал его заново, когда на самом деле он потерян не был, а просто сеть была перегружена), он может выявить эту ситуацию (на основе порядковых номеров), и дублированные данные будут проигнорированы.

Протокол UDP не обеспечивает надежности. UDP сам по себе не имеет ничего похожего на подтверждения, порядковые номера, определение RTT, тайм-ауты или повторные передачи. Если дейтаграмма UDP дублируется в сети, на принимающий узел могут быть доставлены оба экземпляра. Также если клиент UDP отправляет две дейтаграммы одному и тому же получателю, их порядок может быть изменен сетью, и они будут доставлены с нарушением исходного порядка. Приложения UDP должны обрабатывать все подобные случаи.

TCP обеспечивает управление потоком (flow control). TCP всегда сообщает своему собеседнику, сколько именно байтов он хочет получить от него. Это называется объявлением окна (window). В любой момент времени окно соответствует свободному пространству в приемном буфере. Таким образом гарантируется, что отправитель не переполнит буфер получателя. Окно изменяется динамически с течением времени: по мере того как приходят данные от отправителя, размер окна уменьшается, но по мере считывания данных принимающим приложением окно увеличивается. Окно может стать нулевым: если приемный буфер TCP для данного сокета заполнен, он должен подождать, когда приложение считает данные из буфера, перед тем как сможет получать другие данные.

UDP не обеспечивает управления потоком. Быстрый отправитель UDP может с легкостью передавать дейтаграммы с такой скоростью, с которой не может работать получатель UDP. Наконец, соединение TCP также является двусторонним (full-duplex). Это значит, что два приложения могут одновременно отправлять и принимать данные в обоих направлениях через заданное соединение. Поэтому TCP отслеживает состояние таких характеристик, как порядковые номера и размеры окна, для каждого направления потока данных: отправки и приема. UDP может быть (а может и не быть) двусторонним.

Существует несколько ограничений, влияющих на размер дейтаграмм IP. Максимальный размер дейтаграммы IPv4 — 65 535 байт, включая заголовок IPv4. Это связано с тем, что размер дейтаграммы ограничен 16-разрядным полем общей длины, расположенном в IP- дейтаграмме.

Во многих сетях определён параметр **MTU** (maximum transmission unit — максимальная единица передачи), величина которой диктуется аппаратными средствами. Например, размер MTU для Ethernet равен 1500 байтам. Другие канальные уровни, такие как связь «точка-точка» с использованием протокола PPP (Point to-Point Protocol), имеют конфигурируемую MTU. Более ранние соединения по протоколу SLIP (Serial Line Internet Protocol — межсетевой протокол для последовательного канала) часто использовали MTU, равную 296 байтам. Величина транспортной MTU между любыми двумя узлами не обязательно одинакова в обоих направлениях, поскольку маршрутизация в Интернете часто асимметрична, то есть маршрут от A до B может отличаться от маршрута от B до A.

Если размер дейтаграммы превышает канальную MTU, то IPv4 выполняют фрагментацию (fragmentation). Фрагменты не соединяются заново (reassemble), пока они не достигнут конечного получателя. Узлы IPv4 выполняют фрагментацию генерируемых дейтаграмм, а маршрутизаторы IPv4 выполняют фрагментацию передаваемых дейтаграмм.

TCP-сегменты инкапсулируются в IP-дейтаграммы и могут содержать параметры TCP. Для протокола TCP определен максимальный размер сегмента (maximum segment size, **MSS**). MSS указывает собеседнику максимальный объем данных, которые можно отправлять в каждом сегменте TCP. Цель параметра MSS — сообщить собеседнику действительный размер буфера сборки и попытаться предотвратить фрагментацию. Этот параметр TCP позволяет узлу, отправляющему сегмент синхронизации SYN, объявить собеседнику свой MSS - максимальное количество данных, которое он будет принимать в каждом сегменте TCP по этому соединению. В качестве MSS часто используется значение, равное MTU интерфейса минус фиксированные размеры заголовков IP и TCP. В Ethernet при использовании IPv4 это будет 1460. Получить и установить этот параметр TCP можно с помощью параметра сокета TCP_MAXSEG в вызове функций gersockopt и setsockopt.

Значение MSS в TCP-пакете представлено 16-разрядным полем, ограничивающим значение до 65535. Это допустимо для IPv4, поскольку максимальное количество данных TCP в дейтаграмме IPv4 равно 65 495 (65 535 минус 20-байтовый заголовок IPv4 и минус 20-байтовый заголовок TCP). Значение MSS, равное 65 535, считается особым случаем, обозначающим «бесконечность». Это значение используется только вместе с параметром окна (см. ниже) для увеличения объема полезных данных, когда требуется размер MTU, превышающий 65 535. Если TCP использует увеличение объема полезных данных и получает от собеседника объявление размера MSS, равного 65 535 байтам, то предельный размер дейтаграммы, посылаемой им, будет равен MTU интерфейса. Если оказывается, что этот размер слишком велик (например, на пути существует канал с меньшим размером MTU), то будет установлено меньшее значение.

Максимальный размер окна, который может быть установлен в заголовке TCP, равен 65 535, поскольку соответствующее поле занимает 16 бит. Но высокоскоростные соединения (45 Мбит/с и больше) или линии с большой задержкой (спутниковые сети) требуют большего размера окна для достижения максимально возможной пропускной способности. **Параметр TCP масштабирования окна (Window scale option)**, появившийся не так давно, определяет, что объявленная в заголовке TCP величина окна должна быть масштабирована - сдвинута влево на 0-14 бит1, предоставляя максимально возможное окно размером почти в гигабайт (65 535 x 2^14). Для использования параметра масштабирования окна в соединении необходима его поддержка обоими связывающимися узлами.

Работа по TCP между двумя абонентами всегда проходит в три этапа: установление соединения, обмен данными и завершение соединения. Процедура установление соединения TCP была предварительно рассмотрена в предыдущей работе. Обмен данными был рассмотрена выше в этой работе. Рассмотрим

Установка и завершение соединения TCP

Соединение устанавливается с помощью трехстороннего квитирования (рукопожатия):

1. Сервер должен быть подготовлен для того, чтобы принять входящее соединение. Обычно это достигается путем вызова функций socket, bind и listen и называется пассивным открытием. Сервер переходит в состояние LISTEN.

2. Клиент выполняет активное открытие, вызывая функцию connect. Это заставляет клиент TCP послать сегмент SYN j (от слова synchronize —синхронизировать), чтобы сообщить серверу начальный порядковый номер j данных, которые клиент будет посылать по соединению. Клиент переходит в состояние SYN_SENT.

3. Сервер получает SYN j , переходит в состояние SYN_RCVD и отправляет SYN k (k - начальный порядковый номер для данных, которые сервер будет посылать по соединению) и АСК j+1 — подтверждение приема клиентского SYN j .

4. Клиент получает SYN k и АСК j+1 и переходит в состояние ESTEBLISHED. Он подтверждает получение сегмента SYN k от сервера, посылая ему пакет АСК k+1. Сервер, получив АСК k+1, переходит в состояние ESTEBLISHED.

В то время как для установления соединения необходимо три сегмента, для его разрыва требуется четыре сегмента. Рассмотрим процедуру разрыва соединения между приложением А на узле А и приложением Б на узле Б:

1. Приложение А первым вызывает функцию closesocket, и мы в этом случае говорим, что данный узел А выполняет активное закрытие {active close). TCP этого узла отправляет узлу Б сегмент FIN m, обозначающий прекращение передачи данных. Здесь m – начальный порядковый номер данных. TCP узла А переходит в состояние FIN_WAIT_1.

2. Другой узел Б, получающий сегмент FIN m, выполняет пассивное закрытие (passive close). TCP узла Б переходит в состояние CLOSE_WAIT. Полученный сегмент FIN m подтверждается TCP пакетом АСК m+1. (TCP узел А, получив АСК m+1, переходит в состояние FIN_WAIT_2.) Полученный сегмент FIN m передается приложению Б как признак конца файла (после данных, которые уже стоят в очереди, ожидая приема приложением). Получение сегмента FIN означает для приложения Б, что оно уже не получит никаких данных по этому соединению.

3. Через некоторое время после того как приложение Б получило признак конца файла, оно (по логике своей работы) вызывает функцию closesocket для закрытия своего сокета. При этом его TCP отправляет на узел А сегмент FIN n. TCP узла Б переходит в состояние LAST_ACK.

4. TCP узла А, получая окончательный сегмент FIN n, переходит в состояние TIME_WAIT и подтверждает получение сегмента FIN n пакетом ACK n +1. TCP узла Б, получая сегмент ACK n +1, переходит в состояние CLOSED.

Поскольку сегменты FIN и АСК передаются в обоих направлениях, обычно требуется четыре сегмента. Мы говорим обычно, поскольку в ряде сценариев сегмент FIN на первом шаге отправляется вместе с данными. Кроме того, оба сегмента, отправляемых на шаге 2 и 3, отправляются с узла Б, выполняющего пассивное закрытие, и могут быть объединены в один сегмент.

Возможно, что между шагами 2 и 3 некоторые данные перейдут от узла, выполняющего пассивное закрытие, к узлу, выполняющему активное закрытие. Это явление называется половинным закрытием (half-close).

Отправка каждого сегмента FIN происходит при закрытии сокета. Мы указывали, что приложение вызывает для этого функцию closesocket, но при этом мы понимаем, что когда процесс прерывается либо произвольно (при вызове функции exit или при завершении функции main), либо непроизвольно, все его открытые дескрипторы закрываются, что также вызывает отправку сегмента FIN каждому открытому на данный момент соединению TCP.

Активное закрытие может выполнять любой узел - и клиент, и сервер. Часто активное закрытие выполняет клиент, но с некоторыми протоколами (особенно HTTP) активное закрытие выполняет сервер.

Здесь не рассматривается случай внезапного отключения компьютера, при котором никаких сегментов FIN не вырабатывается. Сам TCP такие случаи не отслеживает и хорошее TCP-приложение должно учитывать эту ситуацию: обнаруживать и закрывать соединения с такими компьютерами.

Диаграмма состояний TCP

Операции TCP при установлении и разрыве соединения можно определить с помощью диаграммы состояний на рис. 1. Для соединения определено 11 различных состояний. Правила TCP предписывают переходы из одного состояния в другое, основываясь на текущем состоянии и сегменте, полученном в этом состоянии. Например, если приложение выполняет активное открытие в состоянии CLOSED (закрыто), TCP отправляет сегмент SYN, и новым состоянием становится Клиент. Если затем TCP получает сегмент SYN с сегментом АСК, он отправляет сегмент АСК, и следующим состоянием становится ESTABLISHED (соединение установлено). В этом последнем состоянии происходит большая часть обмена данными.

Две стрелки, идущие от состояния ESTABLISHED, относятся к разрыву соединения. Если приложение вызывает функцию closesocket перед получением признака конца файла (активное закрытие), происходит переход к состоянию FIN_WAIT_1. Но если приложение получает сегмент FIN в состоянии ESTABLISHED {пассивное закрытие), происходит переход в состояние CLOSE_WAIT.

Необходимо отметить, что существует два перехода, о которых мы ранее не говорили: одновременное открытие (когда оба конца связи отправляют сегменты SYN приблизительно в одно время, и эти сегменты пересекаются в сети) и одновременное закрытие (когда оба конца связи отправляют сегменты FIN).

Одна из причин, по которым мы приводим здесь диаграмму перехода состояний, заключается в том, что мы хотим показать все 11 состояний TCP и их названия. Эти состояния отображаются программой netstat, которая является полезным средством отладки клиент-серверных приложений.

Передача пакетов

На рис. 2 представлен реальный обмен пакетами, происходящий во время соединения TCP: установление соединения, передача данных и разрыв соединения. Показаны также состояния TCP, через которые проходит каждый узел.

В этом примере клиент объявляет максимальный размер сегмента (MSS) 1460 байт (обычное значение для IPv4 в Ethernet), а сервер — 1024 байт (обычное значение для более поздних Беркли-реализаций в Ethernet). MSS в каждом направлении отличаются.

Как только соединение установлено, клиент формирует запрос и посылает его серверу. Мы считаем, что этот запрос соответствует одиночному сегменту TCP и его размер меньше 1024 байт — анонсированного размера MSS сервера. Сервер обрабатывает запрос и отправляет ответ, и мы также считаем, что ответ соответствует одиночному сегменту (в данном примере меньше 1460 байт).

Обратите внимание, что подтверждение запроса отправляется клиенту вместе с ответом сервера. Это называется вложенным подтверждением (piggybacking) и обычно происходит, когда время, требуемое серверу для обработки запроса и генерации ответа, меньше приблизительно 200 миллисекунд. Если серверу требуется больше времени — скажем, 1 секунда — ответ будет приходить после подтверждения. Затем мы показываем четыре сегмента, закрывающих соединение. Обратите внимание, что узел, выполняющий активное закрытие (в данном сценарии клиент), входит в состояние TIMEWAIT.

Важно отметить, что если целью данного соединения было отправить запрос, занимающий один сегмент, и получить ответ, также занимающий один сегмент, то при использовании TCP будет задействовано всего 8 пакетов (при подсчёте учтён piggybacking). Если же используется UDP, произойдет

обмен только двумя сегментами: запрос и ответ. Но при переходе от TCP к UDP теряется надежность, которую TCP предоставляет приложению, и множество задач по обеспечению надежности транспортировки данных переходит с транспортного уровня (TCP) на уровень приложения. Другое важное свойство, предоставляемое протоколом TCP, - это управление в условиях перегрузки, которое в случае протокола UDP также должно принимать на себя приложение. Тем не менее важно понимать, что для многих приложений, обменивающихся небольшими объемами данных, использование UDP позволяет избежать накладных расходов, возникающих при установлении и разрыве соединения TCP.

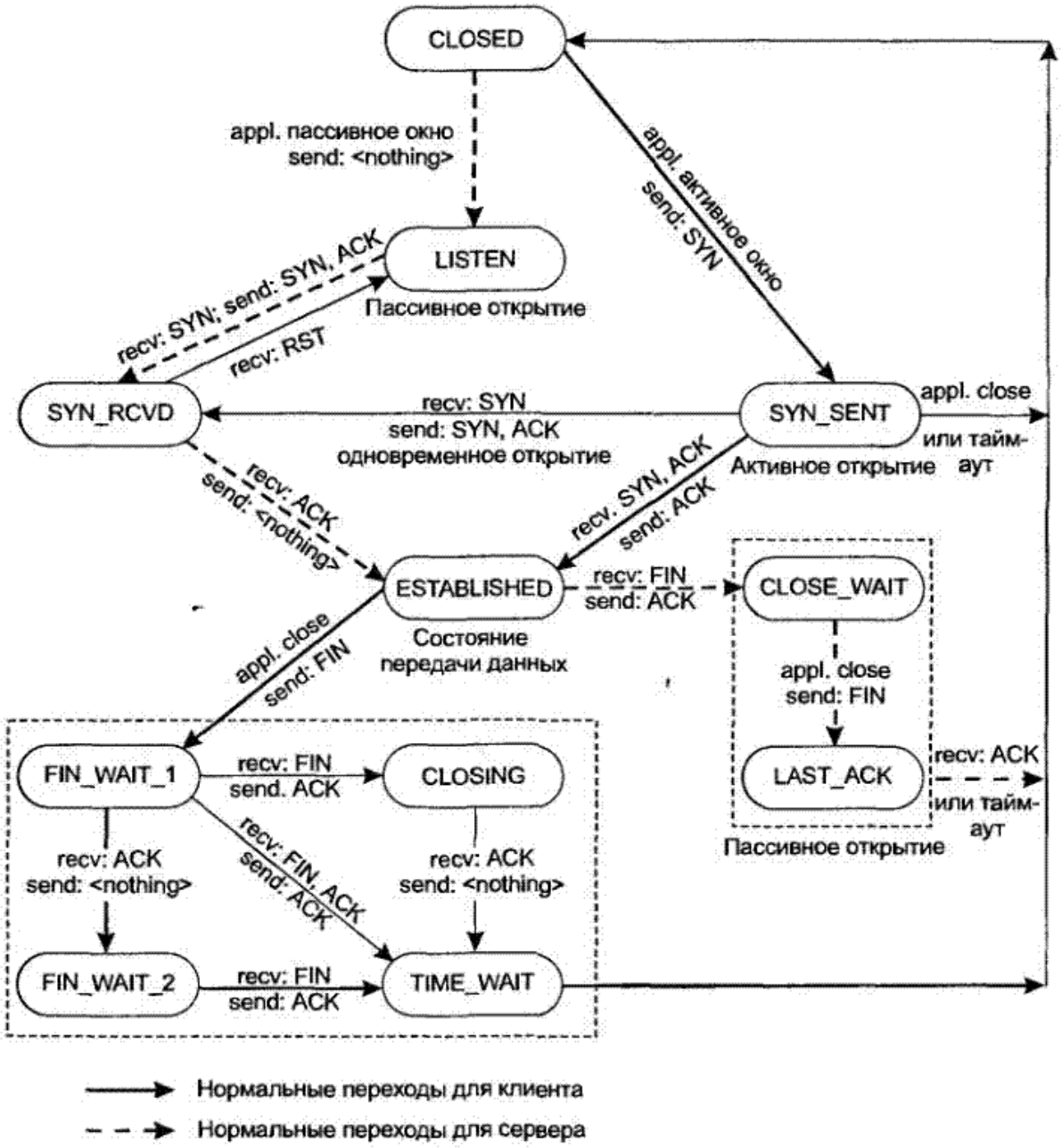

Рис. 1 Диаграмма состояний TCP. appl: переходы между состояниями, возникающие, когда приложение выполняет указанную операцию. recv: переходы между состояниями, возникающие, когда получен указанный сегмент. send: указывает, что отправляется при этом переходе. Переходы клиента отмечены с помощью обычной линии, а переходы сервера - с помощью штриховой линии.

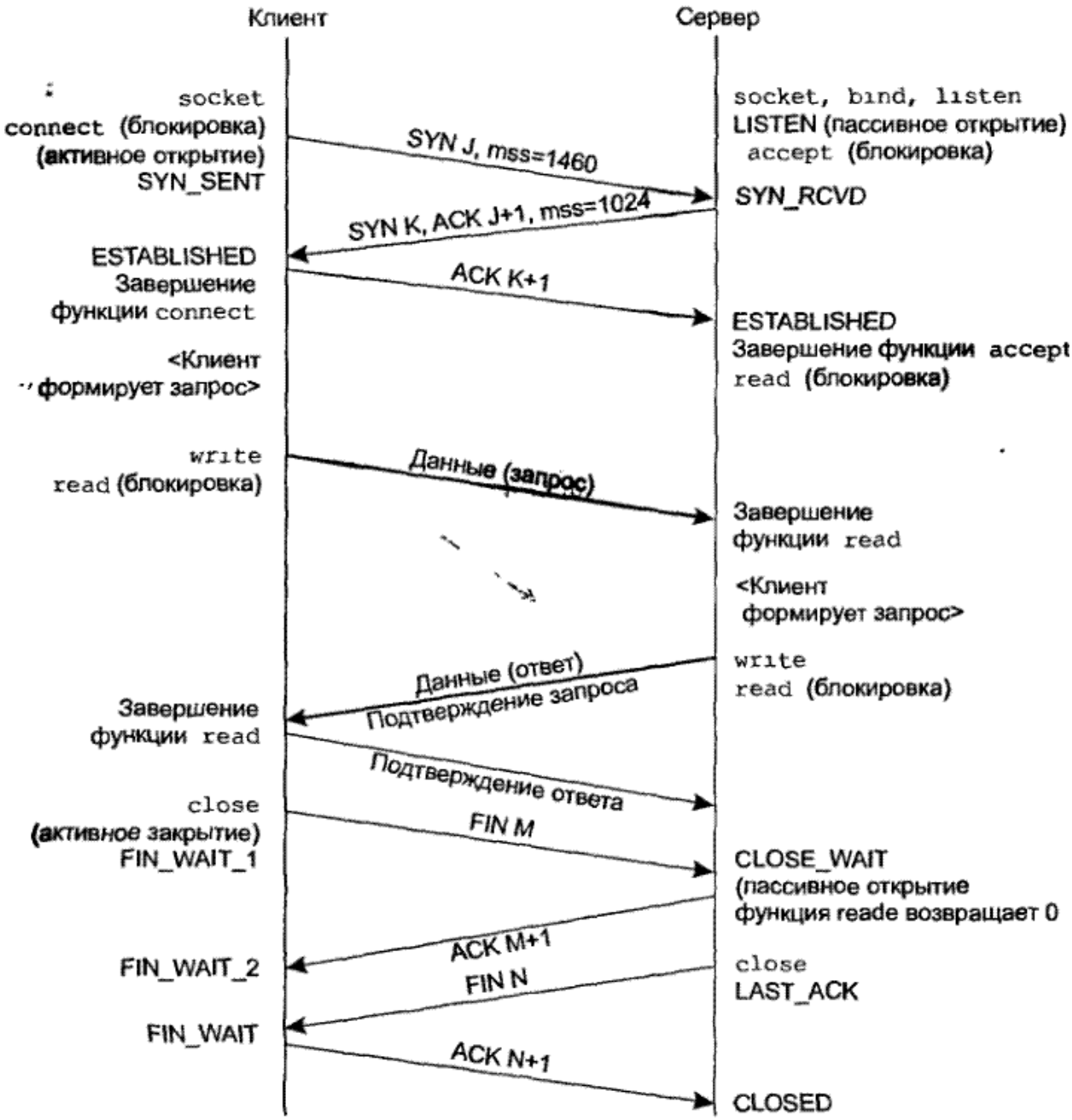

Рис. 2. Обмен пакетами для соединения TCP. Здесь FIN_WAIT это TIME_WAIT.

Состояние TIME_WAIT

Без сомнений, самым сложным для понимания аспектом TCP в отношении сетевого программирования является состояние TIME_WAIT (время_ожидания). Узел, выполняющий активное закрытие, проходит это состояние. Продолжительность этого состояния конечной точки равна двум MSL (maximum segment lifetime - максимальное время жизни сегмента), иногда этот период называется 2MSL.

В каждой реализации TCP выбирается определенное значение MSL. Продолжительность состояния TIME_WAIT - от 1 до 4 минут. MSL - это максимальное количество времени, в течение которого дейтаграмма IP может оставаться в объединенной сети. Это время ограничено, поскольку каждая дейтаграмма содержит 8-разрядное поле предельного количества транзитных узлов, или прыжков, максимальное значение которого равно 255. Хотя этот предел ограничивает количество транзитных узлов, а не время пребывания пакета в сети, считается, что пакет с максимальным значением этого предела (которое равно 255) не может существовать в объединенной сети дольше, чем предписывает значение MSL.

В результате различных аномалий, происходящих в объединенных сетях, пакеты обычно теряются. Если возникает сбой на маршрутизаторе или отключается связь между двумя маршрутизаторами, то для стабилизации и поиска альтернативного пути требуются секунды или минуты. В течение этого периода

могут возникать петли маршрутизации (маршрутизатор A отправляет пакеты маршрутизатору B, а маршрутизатор B отправляет их обратно маршрутизатору A), и пакеты теряются в этих петлях. Если потерянный пакет - это сегмент TCP, то по истечении установленного времени ожидания отправляющий узел снова передает пакет, и этот заново переданный пакет доходит до конечного получателя по некоему альтернативному пути. Но если спустя некоторое время (не превосходящее MSL) после начала передачи потерянного пакета петля маршрутизации исправляется, пакет, потерянный в петле, отправляется к конечному получателю. Начальный пакет называется потерянной копией или дубликатом, а также блуждающей копией или дубликатом. TCP должен обрабатывать потерянные пакеты.

Состояние TIME_WAIT позволяет добиться двух целей:

1. Обеспечить надежность разрыва двустороннего соединения TCP.
2. Подождать, когда истечет время жизни в сети старых дублированных сегментов.

Первую причину можно объяснить, обратившись к рис. 2 с предположением, что последний сегмент ACK потерян. Сервер еще раз отправит свой последний сегмент FIN, и клиент должен будет обработать эту информацию. Он ответит сегментом RST (другой тип сегмента TCP), что сервер интерпретирует как ошибку. Если TCP выполняет всю работу, необходимую для корректного разрыва потока данных в обоих направлениях соединения (его двустороннего закрытия), он должен корректно обрабатывать потерю любого из этих четырех сегментов. Этот пример также объясняет, почему узел, выполняющий активное закрытие, остается в состоянии TIME WAIT: это тот узел, который, возможно, должен будет передать повторно последний сегмент ACK.

Чтобы понять вторую причину, по которой необходимо состояние TIME_WAIT, предположим, что у нас имеется соединение между IP-адресом 206.62.226.33, порт 1500, и IP-адресом 198.69.10.2, порт 21. Это соединение закрывается, и спустя некоторое время мы устанавливаем другое соединение между теми же IP-адресами и портами: 206.62.226.33, порт 1500, и 198.69.10.2, порт 21. Последнее соединение называется новым воплощением предыдущего соединения, поскольку использует те же IP-адреса и порты. TCP должен предотвратить появление старых дубликатов, относящихся к данному соединению, в новом воплощении соединения. Чтобы выполнить это, TCP не инициирует новое воплощение соединения, которое в данный момент находится в состоянии TIME_WAIT. Поскольку продолжительность состояния TIME_WAIT равна двум MSL, это позволяет удостовериться, что истечет и время жизни пакетов, посланных в одном направлении, и время жизни пакетов, посланных в ответ. Используя это правило, мы гарантируем, что в момент успешного установления соединения TCP время жизни в сети всех старых дубликатов от предыдущих воплощений этого соединения уже истекло.

Отправка по TCP

У каждого сокета TCP есть буфер отправки, размер которого мы можем изменять с помощью установки опции сокета SO_SNDBUF. Когда приложение вызывает функцию send, ядро копирует данные из буфера приложения в буфер отправки сокета. Если для всех данных приложения недостаточно места в буфере сокета (либо буфер приложения больше буфера отправки сокета, либо в буфере отправки сокета уже имеются данные), этот процесс приостанавливается (переходит в состояние ожидания). Подразумевается, что мы используем обычный блокируемый сокет. Ядро возвращает управление из функции send только после того, как последний байт в буфере приложения будет скопирован в буфер отправки сокета. Следовательно, успешное возвращение управления из функции send в сокет TCP говорит нам лишь о том, что мы можем снова использовать наш буфер приложения. Оно не говорит о том, получил ли собеседник отправленные данные.

TCP помещает данные в буфер отправки сокета и отправляет их TCP собеседнику, основываясь на всех правилах передачи данных TCP. Собеседник TCP должен подтвердить данные, и только когда от него придет сегмент ACK, подтверждающий прием данных, наш TCP сможет удалить подтвержденные данные из буфера отправки сокета. TCP должен хранить копию данных, пока их прием не будет подтвержден получателем. TCP отправляет данные IP порциями размером MSS или меньше, добавляя свой заголовок TCP к каждому сегменту. Здесь MSS — это значение, анонсированное собеседником, или 536, если собеседник не указал значение MSS. IP добавляет свой заголовок, ищет в таблице маршрутизации IP-адрес получателя (соответствующая запись в таблице маршрутизации задает исходящий интерфейс, то есть интерфейс для исходящих пакетов) и передает дейтаграмму на соответствующий канальный уровень. IP может выполнить фрагментацию перед передачей дейтаграммы, но, как мы отмечали выше, одна из целей параметра MSS — не допустить фрагментации. У каждого канального соединения имеется очередь вывода, и если она заполнена, пакет игнорируется и вверх по стеку протоколов возвращается ошибка: от канального уровня к IP и затем от IP к TCP. TCP учтет эту ошибку и попытается отправить сегмент позже. Приложение не информируется об этом временном состоянии.

Отправка по UDP

Буфер отправки сокета на самом деле не существует. У сокета UDP есть размер буфера отправки (который мы можем изменить с помощью параметра сокета SO_SNDBUF), но это просто верхний предел размера дейтаграммы UDP, которая может быть записана в сокет. Если приложение записывает

дейтаграмму размером больше буфера отправки сокета, возвращается сообщение об ошибке EMSGSIZE. Поскольку протокол UDP не является надежным, ему не нужно хранить копию данных приложения. Ему также не нужно иметь настоящий буфер отправки (данные приложения обычно копируются в буфер ядра по мере их движения вниз по стеку протоколов, но эта копия сбрасывается канальным уровнем после передачи данных). UDP просто добавляет свой 8-байтовый заголовок и передает дейтаграмму протоколу IP. IPv4 добавляет свой заголовок, определяет исходящий интерфейс, выполняя функцию маршрутизации, а затем либо добавляет дейтаграмму в очередь вывода канального уровня (если размер дейтаграммы не превосходит MTU), либо фрагментирует дейтаграмму и добавляет каждый фрагмент в очередь вывода канального уровня.

Если приложение UDP отправляет большие дейтаграммы (например, 2000-байтовые), существует гораздо большая вероятность фрагментации, чем в случае TCP, поскольку TCP разбивает данные приложения на порции, равные по размеру MSS, которому нет аналога в UDP.

Успешное завершение функции записи в сокет UDP говорит о том, что либо дейтаграмма, либо фрагменты дейтаграммы были добавлены к очереди вывода канального уровня. Если недостаточно места для дейтаграммы или одного из ее фрагментов, приложению часто возвращается сообщение об ошибке или не возвращают этой ошибки, не предоставляя приложению никаких указаний на то, что дейтаграмма была проигнорирована еще до начала передачи.

Функции readn, writen и readline

Потоковые сокеты (например, сокеты TCP) демонстрируют с функциями recv и send поведение, отличное от обычного ввода-вывода файлов. Функция recv или send на потоковом сокете может ввести или вывести намного меньше байтов, чем запрашивалось, но это не будет ошибкой. Причиной может быть достижение границ буфера для сокета в ядре. Все, что требуется в этой ситуации, — чтобы процесс повторил вызов функции recv или send для ввода или вывода оставшихся байтов. Этот сценарий всегда возможен на потоковом сокете при выполнении функции recv, но с функцией send он обычно наблюдается только если сокет неблокируемый. Тем не менее вместо send мы всегда вызываем функцию writen на тот случай, если в данной реализации возможно возвращение меньшего количества данных, чем мы запрашиваем.

Введем две функции для чтения и записи в потоковый сокет.

```
int readn( SOCKET fd, char *bp, int len);
int writen( SOCKET fd, char *bp, int len);
```

Обе функции возвращают количество считанных или записанных байтов, которые расположены в буфере bp. Если нет объективных препятствий, то обе функции возвращают затребованное число байт. Так функция readn возвратит меньше байт, чем запрошено, если удалённое приложение закроет соединение. В этом случае вызов функция recv внутри функции readn возвратит 0. При ошибке обе функции возвращают SOCKET_ERROR и выводят расшифровку ошибки.

```
int readn( SOCKET fd, char *bp, int len){
        int cnt;
        int rc;
        cnt = len;
        while ( cnt > 0 )
        {
                rc = recv( fd, bp, cnt, 0 );
                if (rc == SOCKET_ERROR){
                        myprintf("Ошибка recv %s\n", encodeWSAGetLastError(WSAGetLastError()));
                        return SOCKET_ERROR;
                        }
                if ( rc == 0 ) //
                return len - cnt;
                bp += rc;
                cnt -= rc;
        }
        return len;
}
```

Листинг 1. Функция readn.

```
int writen(SOCKET fd, const char * bp, int len){
        int     nleft;
        int     nwritten;
        const char      *ptr;
```

```
        ptr = bp;
        nleft = len;
        while (nleft > 0) {
                if ( (nwritten = send(fd, ptr, nleft, 0)) == SOCKET_ERROR) {
                        myprintf("Ошибка send %s\n", encodeWSAGetLastError(WSAGetLastError()));
                        return SOCKET_ERROR;
                        }
                nleft -= nwritten;
                ptr   += nwritten;
        }
        return(len);
}
```

Листинг 2. Функция writen.

TCP передаёт данные в виде непрерывного потока байт. Пусть на уровне приложений происходит обмен записями с определённой в программе структурой. Причём размер записи заранее неизвестен. Передать запись с помощью функции writen можно, но принять её на противоположной стороне невозможно, татк как приёмная сторона не знает истинного размера записи. Поэтому на передающей стороне нпеобходимо определить способ для разделения записей. Либо перед самой записью передавать её длину. Либо записи разделять специальным символом.

Рассмотрим простейший случай, когда запись представляет собой строку языка Си: последовательность ненулевых байт, завершающуюся нулевым байтом. Для чтения такой записи можно использовать функцию

```
int readLine(SOCKET fd, char *buf){
        int rc, n =0;
        while(1){
        rc = recv( fd, buf, 1, 0 );
        if (rc == SOCKET_ERROR){
                myprintf("Ошибка recv %s\n", encodeWSAGetLastError(WSAGetLastError()));
                return SOCKET_ERROR;
                }
        if (rc == 0)
                return 0;
        if (*buf == '\0')
                return n+1;
        n++;
        buf++;
        }
    }
```

Листинг 3. Функция readLine

Функция readLine вынуждена читать данные побайтно, чтобы не пропусить признак конца записи – нулевой байт т не забрать лишние (возможно не свои) байты из потока. Функция возвращает чичло принятых байт, ключая нулевой байт.

Исходные тексты всех трёх функций приведены в листингах 1, 2 и 3.

Параллельный эхо сервер TCP с использованием функции select

В API сокетов присутствует замечательная функция. Функция select определяет статус одного или более сокетов для выполнения синхронного (блокирующего) ввода/вывода. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set* exceptfds, const struct timeval* timeout);
```

Параметр nfds игнорируется и включён для совместимости с сокетами Беркли.

readfds – необязательный указатель на множество сокетов, контролируемое на возможность чтения из них.

writefds – необязательный указатель на множество сокетов, контролируемое на возможность записи в них.

exceptfds – необязательный указатель на множество сокетов, контролируемое на ошибку.

Множество сокетов определено так

```
typedef struct fd_set {
u_int fd_count;
```

SOCKET fd_array[FD_SETSIZE];
} fd_set;

fd_count задаёт число сокетов в множестве, а fd_array – это массив сокетов. Константа FD_SETSIZE определяет максимальное число сокетов и равна 64. Может быть изменена (до включения файла Winsock2.h)

timeout – максимальное время ожидания, пока хотя бы один сокет в множествах readfds, writefds, exceptfds не будет иметь статус. Установка параметра в нулевой указатель равносильно бесконечному ожиданию. Задаётся в структуре TIMEVAL. Структура TIMEVAL определена так

typedef struct timeval {
long tv_sec;
long tv_usec;
} timeval;
Здесь tv_sec – задаёт время в секундах, а tv_usec в микросекундах.

Функция возвращает ноль, если время ожидания истекло либо возвращает число готовых сокетов, которые содержаться в структурах fd_set. Иначе функция возвращает SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок WSANOTINITIALISED, WSAEFAULT, WSAENETDOWN, WSAEINVAL, WSAEINTR, WSAEINPROGRESS, WSAENOTSOCK. Расшифровку ошибок смотри в приложении или в MSDN.

При возврате функция обновляет множества fd_set и отражает в них состояния контролируемых сокетов.

Параметр readfds идентифицирует сокеты, которые контролируются на возможность чтения. Прослушивающий сокета будет отмечен как читаемый, если им получен такой входящий запрос на соединение, что функция accept примет его без блокировки. Для других сокетов возможность чтения означает, что в очереди у сокету имеются такие данные для чтения, что вызов функций чтения recv и recvfrom прочитает их без блокировки. Для сокетов, ориентированных на соединение, читаемость также показывает, что с противоположной стороны пришёл запрос на закрытие сокета.

Параметр writefds идентифицирует сокеты, которые контролируются на возможность записи. Если сокет выполняет неблокирующий вызов функции соединения connect, то он отмечается как доступный для записи, если установка соединения успешно завершена. Если сокет не выполняет вызов функции соединения connect, то доступность для записи означает, что последующий вызов функций записи send, sendto на нём будет успешен. Однако функций записи могут блокироваться при недостаточном выходном системном буфере. Кроме того возможность записи из-за событий в приложении и системе с течением времени может быть утеряна.

Параметр exceptfds идентифицирует сокеты, которые контролируются на ошибку или присутствие внеполосных данных. Если сокет выполняет неблокирующий вызов функции соединения connect, то сбой соединения отражается в множестве exceptfds (приложение должно вызвать функцию getsockopt с параметром SO_ERROR для определения причины ошибки).

По крайней мере один из параметров readfds, writefds или exceptfds должен быть ненулевым и должны указывать на множества с не менее чем одним сокетом.

Определены четыре макроса для работы с множествами сокетов.

FD_CLR(s, *set) – удаляет сокет s из множества set.
FD_ISSET(s, *set) - не ноль, если s содержится в множества set. Иначе ноль.
FD_SET(s, *set) – добавляет сокет s в множество set.
FD_ZERO(*set) - устанавливает множество set в нулевое множество.

Параметр time-out задаёт как долго может длится вызов функции select. Нулевой указатель блокирует вызов пока хотя бы один из указанных в множествах сокетов не станет удовлетворять указанному критерию. Структура TIMEVAL задаёт максимальное время, сколько select будет ждать перед возвратом. При возврате содержимое этой структуры не меняется. Если структуру TIMEVAL инициализировать значениями {0, 0}, то select завершится немедленно; это используется для опроса состояния указанных сокетов.

Практическая часть.

Последовательный TCP эхо сервер и клиент.

Превратим программу tcpclient из первой работы (листинг 7) в TCP эхо клиент tcpSclient. Для этого заменим фрагмент

```
myprintf("Попытка передать сообщение Привет.\n");
if ((Ret = send(s, msg, lmsg, 0)) == SOCKET_ERROR)
        ...
myprintf("Успешно передано %d байтов сообщения %s.\n", Ret, msg);
```

на фрагмент

```
myprintf("Можем обмениваться данными с Эхо сервером .\n");
TCPechoC(s);
```

Листинг 4. Логика работы эхо клиента tcpSclient

Превратим программу tcpserver из первой работы (листинг 6) в последовательный TCP эхо сервер tcpSserver. Для этого заменим фрагмент

```
if ((Ret = recv(NewConnection, DataBuffer, sizeof(DataBuffer), 0))
 .............
myprintf("Успешно получено %d байтов в сообщении  %s.\n", Ret, ataBuffer);
```

на фрагмент

```
while(1){
        Ret = TCPechoS(NewConnection);
        if ( Ret == 0 || Ret== SOCKET_ERROR)
        break;
        myprintf("Отражено %d байт.\n", Ret-1);
    }
```

Листинг 5. Логика работы эхо сервера tcpSserver

Функции TCPechoC и TCPechoS реализуют логику работы эхо клиента и сервера, соответственно, и представлены в листингах 6 и 7.

Функция TCPechoC для эхо клиента (листинг 6) считывает в бесконечном цикле с помощью функции gets строку с консоли. Дополняет её нулевым байтом (наш разделитель записей) и с помощью нашей функции writen (листинг 2) отправляет её серверу. Далее с помощью функции readLine (листинг 3) данные получаем от сервера. Цикл разрывается, если вводится q или ^Z, от сервера нет данных (readLine возвратил 0) или произошла ошибка передачи SOCKET_ERROR.

```
void TCPechoC(SOCKET            fd){
        char    buf[LINELEN+1];                 /* buffer for one line of text          */
        int     outchars, inchars;/* characters counts                    */
        myprintf("q или ^Z- выход.\n");
                while (gets(buf)) {
                if (!strcmp(buf,"q")) break;
                outchars = strlen(buf);
                buf[outchars] = '\0';
                outchars = writen(fd, buf, outchars+1);
                if ( outchars == SOCKET_ERROR){
                        myprintf("Не могу отправить данные серверу SOCKET_ERROR.\n");
                        continue;
                        }
                inchars = readLine(fd, buf);
                if ( inchars == SOCKET_ERROR){
                        myprintf("Не могу получить данные от сервера: SOCKET_ERROR.\n");
                        continue;
                        }
                if ( inchars == 0){
                        myprintf("От сервера нет данных.\n");
                        continue;
                        }
                puts(buf);
                }
        myprintf("Клиент завершил работу.\n");
        closesocket(fd);
}
```

Листинг 6. Функция TCPechoC для эхо клиента

Функция TCPechoS для сервера (листинг7) с помощью функции readLine забирает строку от клиента и далее с помощью функции writen отправляем её обратно. Если readLine возвратил 0, то это признак того, что клиент завершил работу. Функция выводит об этом сообщение на консоль и завершается,возвращает ноль. Если произошла ошибка SOCKET_ERROR функция выводит об этом сообщение на консоль и завершается ,возвращая SOCKET_ERROR. В коде сервера (листинг5) функция TCPechoS вызывается в бесконечном цикле, но если она возвращает 0 или SOCKET_ERROR, то цикл разрывается. Это свидетельствкет обокончании работы клиента.

```
int TCPechoS(SOCKET fd){
        char    buf[BUFSIZE];
        struct sockaddr_in ca;
        int     inchars, outchars, caSize;
        caSize = sizeof(ca);
        inchars = readLine(fd, buf);
        if ( inchars == SOCKET_ERROR){
                getpeername(fd, (struct sockaddr *)&ca, &caSize);
                myprintf("Не могу получить данные от клиента %s:%d. SOCKET_ERROR.\n",
                inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
                return SOCKET_ERROR;
                }
        if ( inchars == 0){
                getpeername(fd, (struct sockaddr *)&ca, &caSize);
                myprintf("Клиент %s:%d завершил работу.\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
                return inchars;
        }
        outchars = writen(fd, buf, inchars);
        if (outchars == SOCKET_ERROR){
                getpeername(fd, (struct sockaddr *)&ca, &caSize);
                myprintf("Не могу передать данные клиенту %s:%d. SOCKET_ERROR.\n",
                inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
                return SOCKET_ERROR;
                }
        return outchars;
}
```

Листинг 7. Функция TCPechoS для эхо сервера

Результат работы последовательного эхо клиента и сервера представлена на рисунке 3. В случае закрытия клиента или потери связи вывод сервера представлен на рисунке 4.

Второй эхо клиент запустить не удастся, так как вычислительный процесс эхо сервера tcpSserver всегда находится внутри цикла (листинг5) и некому осуществить вызов функции accept для прослушивающего сокета ListeningSocket (См. Листинг 6 лабораторной работы №1) для принятия соединения от нового клиента.

Параллельный эхо сервер TCP с использованием нитей

По умолчанию вычислительный процесс имеет одну нить выполнения (в другой терминологии поток) , связанную с функцией main. Можно для процесса создать дополнительные нити выполнения. Каждая нить имеет свой счётчик команд и свой стек выполнения. Все нити в процессе выполняются параллельно. В качестве носителя кода для нити задаётся функция, код которой находится в адресном пространстве процесса. Все внешние и статические имена процесса видны в его нитях. В упрощённой форме в Win32 API нити создаются вызовом

```
HANDLE hThread = CreateThread(NULL, 0, ClientThread,
        (LPVOID)Parametr, 0, &dwThreadId);
```

где ClientThread – функция носитель кода нити с прототипом

DWORD WINAPI ClientThread(LPVOID lpParam);

Parametr – фактический параметр передаваемый функции нити ClientThread;

DWORD dwThreadId – идентификатор созданной нити.

Функция возвращает дескриптор созданной нити или NULL при неудаче создания.

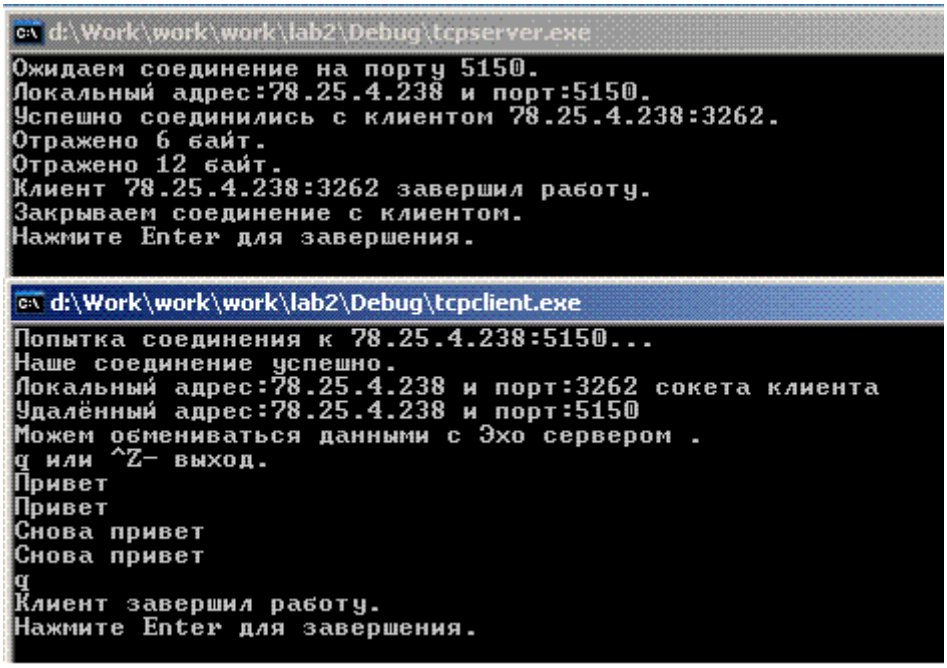

Рис. 3. Скришот работы последовательного эхо клиента и сервера.

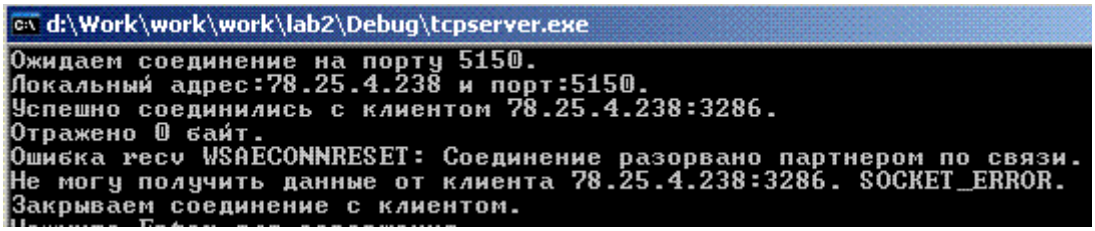

Рис. 4. Скришот вывод эхо сервера в случае закрытия клиента или потери связи.

Идея создания параллельный сервера TCP с использованием нитей проста: после вызова функции accept для прослушивающего сокета для принятия соединения от нового клиента и возврата ею нового сокета для соединения с новым клиентом создать нить для работы с этим новым сокетом. Главная нить продолжает слушать сеть с помощью функции accept.

Скелет кода для параллельного TCP сервера tcpTreadServer имеет вид

```
WSADATA            wsaData;
SOCKET             ListeningSocket, NewConnection;
struct sockaddr_in         ServerAddr, ClientAddr;
int ClientAddrLen, Port = 5150,     Ret;
HANDLE       hThread;
DWORD        dwThreadId;

WSAStartup(MAKEWORD(2,2), &wsaData));
ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(Port);
ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(ListeningSocket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr));
listen(ListeningSocket, 5);
while (1){
            myprintf("Ожидаем соединение на порту %d.\n", Port);
            ClientAddrLen = sizeof(ClientAddr);
            NewConnection = accept(ListeningSocket, (struct sockaddr *) &ClientAddr,
                        &ClientAddrLen);
            myprintf("Принят клиент: %s:%d\n", inet_ntoa(ClientAddr.sin_addr), ntohs(ClientAddr.sin_port));
            hThread = CreateThread(NULL, 0, ClientThread, (LPVOID)NewConnection, 0, &dwThreadId);
    if (hThread == NULL) {
        myprintf("Ошибка CreateThread: %d\n", GetLastError());
        continue;
```

```
        }
    CloseHandle(hThread);
        }
  closesocket(ListeningSocket);
WSACleanup();
myprintf("Нажмите Enter для завершения.\n");
getchar();
}

DWORD WINAPI ClientThread(LPVOID lpParam){
…
}
```

Листинг 8. Скелет кода для параллельного TCP сервера tcpTreadServer с использованием нитей

В листинге 8 сервер вызывает accept в бесконечном цикле. Заметим, что accept блокируется до прихода нового запроса на соединение. Пр приходе запроса на соединение accept возвращает сокет NewConnection для нового соединения. Далее создаётся нить с функцией носителем ClientThread. В качестве параметра для функции ClientThread передаётся новый сокет NewConnection. Далее функция accept ждёт нового запроса на соединение от нового клиента.

Для получения параллельного эхо сервера следует лишь поместить логику его работы в функцию нити ClientThread (см. Листинг 9). В листинге 9 по сравнению с листингом 5 добавлен лишь код, указывающий адрес клиента.

```
DWORD WINAPI ClientThread(LPVOID lpParam){
SOCKET   sock    =(SOCKET)lpParam; //Преобразуем параметр к типу SOCKET
struct sockaddr_in ClientAddr;
int Ret, caSize = sizeof(ClientAddr);
getpeername(sock, (struct sockaddr *)&ClientAddr, &caSize);
myprintf("Успешно соединились с клиентом %s:%d.\n",inet_ntoa(ClientAddr.sin_addr),
        ntohs(ClientAddr.sin_port));
while(1){
        Ret = TCPechoS(sock);
        if ( Ret == 0 || Ret== SOCKET_ERROR)
        break;
        myprintf("Отражено %d байт от клиента %s:%d.\n",
                Ret-1,inet_ntoa(ClientAddr.sin_addr), ntohs(ClientAddr.sin_port));                }
myprintf("Связь с клиентом %s:%d прекращена.\n",inet_ntoa(ClientAddr.sin_addr),
                ntohs(ClientAddr.sin_port));
return 0;
}
```

Листинг 9. Функцию нити параллельного эхо TCP сервера tcpTreadServer

Собрав и отладив параллельный эхо TCP сервера tcpTreadServer, можно убедиться, что к нему можно подсоединяться несколькими старыми эхо клиентами tcpSclient.

Работа с прикладными пакетами фиксированной длины и измерение скорости обмена

Поставим задачу разработать эхо сервер и клиент к нему с использованием фиксированной длины пакетов. Предпишем серверу следующую логику работы: в цикле для каждого клиента выбрать из сокета весь пакет (DATA_BUFSIZE байт), а затем отослать его полностью обратно клиенту. Предпишем клиенту такую логику. Подготовить пакет. Полностью его отправить. Дождаться ответа и выбрать из сокета весь пакет. Сравнить полученный пакет с переданным. Затем подготовить очередной пакет для отправки.

Мы используем обвёртки для некоторых функций. Напомним, что внутри обвёрток находится диагностический код и, как правило имя обвёртки и её семантика совпадает с оригинальной функцией, но начинается с большой буквы или подчёркивания.

Начнём с клиента. Потребуем, чтобы он работал в двух режимах: непрерывном и по пакетном. Его код приведен в листинге 12. Заголовочный файл Client.h имеет вид (листинг 11). Он включает файл DATA_BUFSIZE.h (листинг 10), в котором определены длины буферов в байтах и целых числах

```
#define DATA_BUFSIZE_INT 65536//5000000
#define DATA_BUFSIZE DATA_BUFSIZE_INT*sizeof(int)
```

Листинг 10. Заголовочный файл DATA_BUFSIZE.h

В файле Client.h определены прототипы рассмотренных в первой работе функций myprintf и encodeWSAGetLastError и прототипы функций readn и written из листингов 1 и 2. В него также включены прототипы функций обвёрток _WSAStartup, Socket и Connect. Функция printSpeed будет выводить скорости обмена в заданную позицию консольного окна.

```
#include <winsock2.h>
#include <stdio.h>
#include "../DATA_BUFSIZE.h"
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int Connect( SOCKET s,const struct sockaddr* name, int namelen);
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
int readn( SOCKET fd, char *bp, int len);
int writen( SOCKET fd, char *bp, int len);
void printSpeed(double dCurrenReadSpeed, double dAvgDuplexSpeed, int x, int y);
```

Листинг 11. Заголовочный файл Client.h

В начале программы определяется задан ли в командной строке IP-адрес сервера и режим работы. Далее идёт стандартная последовательность установки соединения с сервером. Затем динамически выделяется память для исходящего пакета msgForW и входящего пакета с проверкой на факт выделения. Текущий тик начала работы программы заносится в переменные startTime и prevTime. Тики система генерирует 1000 раз в секунду. Далее входим в бесконечный цикл обмена информацией с сервером. Если не задан непрерывный рехим работы, то проверяем ввод пользователя. Затем моделируем содержимое исходящего пакета, путём заполнения его случайными числами, полученными с помощью функции rand, и отправляем пакет серверу, используя функцию writen. Далее вычисляем текущее время currentTime, длительность durationOnce обмена информацией с сервером и общее время работы durationTotal этой программы. Вычисляем общее число переданных байт totalBytes, текущую dCurrenSpeed и среднюю dAvgSpeed скорости обмена в мегабитах в секунду. С помощью функции printSpeed (листинг 13) выводим скорости в левый верхний угол консольного окна. Далее с помощью функции memcmp сравниваем переданное с принятым и в случае несовпадения подаём звуковой сигнал об ошибке частотой в 333 герца длительностью 3.333 секунды. Обновляем время начала обмена prevTime=currentTime.

Трудно оценить отдельно скорость записи и чтения, так как реально запись и чтение с помощью функций recv и send происходит не с сетью, а с системными сетевыми буферами локального компьютера. Поэтому мы оцениваем время и обмена информации в двух направлениях.

```
#include "Client.h"
void main(int argc, char **argv){
  WSADATA            wsaData;
  SOCKET             s;
  struct sockaddr_in         ServerAddr;
  int                Port = 5150;
  int                i;
  int *msgForR, *msgForW;
     DWORD startTime, prevTime;
     double totalBytes=0;
   if (argc <= 1)   {
     myprintf("Использование: tcpclient <Server IP address> [c].\n");
     myprintf("c - для непрерывной передачи данных серверу.\n");
     return;
   }
  WSAStartup(MAKEWORD(2,2), &wsaData);
  Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(Port);
  ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
  Connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr));
  If(!(msgForW = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE)) return;
  If(!(msgForR = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE)) return;
prevTime =startTime=GetTickCount();

  while(1){
```

```
	DWORD currentTime;
	DWORD durationOnce, durationTotal;
	double dCurrenSpeed, dAvgSpeed;
	if(!argv[2]){ // Не непрерывный режим
		char c;
		myprintf("c<Enter> - послать порцию данных.\n");
		myprintf("q<Enter> - выход.\n");
		while(1){
			c=getchar();
			if(c=='c')break;
			if(c=='q'){
				closesocket(s);
				WSACleanup();
				ExitProcess(0);
			}
		}
	}

	for(i=0;i<DATA_BUFSIZE_INT;i++)		msgForW[i]= rand();
	if(writen(s , msgForW , DATA_BUFSIZE)==SOCKET_ERROR){
		myprintf("ошибка  writen %s\n", encodeWSAGetLastError(WSAGetLastError()));
		break;
	}
		Ret=readn(s , msgForR , DATA_BUFSIZE);
	if(Ret==SOCKET_ERROR){
		myprintf("ошибка  readn %s\n", encodeWSAGetLastError(WSAGetLastError()));
		break;
	}
	if(Ret==SOCKET_ERROR){
		myprintf("Сервер завершил работу.\n");
		break;
	}
	currentTime=GetTickCount();
	durationOnce=currentTime-prevTime;
	durationTotal=currentTime-startTime;
	totalBytes+=2*DATA_BUFSIZE;
	if(durationOnce)
		dCurrenSpeed=(1000.0* DATA_BUFSIZE*8.0)/(1024.0*1024.0*durationOnce);
	if(durationTotal)
		dAvgSpeed=(1000.0*totalBytes*8.0)/(1024.0*1024.0*durationTotal);
	if(durationOnce&&durationTotal)
		printSpeed(dCurrenSpeed, dAvgSpeed, 0, 0);
	if(memcmp(msgForR,msgForW,DATA_BUFSIZE)){
		puts("ERROR\n");
		Beep(333,3333);
		//exit(77);
		}
	prevTime=currentTime;
}
}
```

Листинг 12. Код клиента Client.c для пакетов фиксированной длины и измерением скорости обмена

```
#include <windows.h>
#include <stdio.h>
void printSpeed(double dCurrenReadSpeed, double dAvgDuplexSpeed, int x, int y){
char B[256];
HANDLE hStdout;
DWORD cWritten;
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
GetConsoleScreenBufferInfo(hStdout, &csbiInfo);
```

```
csbiInfo.dwCursorPosition.X =x;
csbiInfo.dwCursorPosition.Y =y;
SetConsoleCursorPosition(hStdout, csbiInfo.dwCursorPosition);
WriteFile(  hStdout,B, sprintf(B,"%.2f Mbs - current speed ",dCurrenReadSpeed),&cWritten,NULL);
GetConsoleScreenBufferInfo(hStdout, &csbiInfo);
csbiInfo.dwCursorPosition.X =x;
csbiInfo.dwCursorPosition.Y =y +1;
SetConsoleCursorPosition(hStdout, csbiInfo.dwCursorPosition);
WriteFile(  hStdout,B, sprintf(B,"%.2f Mbs - Avg speed ",dAvgDuplexSpeed),&cWritten,NULL);
}
```

Листинг 13. Функция printSpeed.

Приведём в листинге 15 код многопоточного эхо сервера для пакетов фиксированной длины и измерением скорости обмена и измерения скоростей обмена. Заголовочный файл приведен в листинге 14. Здесь N – максимальное число клиентов.

Вначале определён ряд глобальных данных: prevTimes – времена началов обмена для сокетов; startTime – начало работы сервера; nTransfers – общее число обменов пакетами со всеми клиентами начиная со старта сервера. Атрибут volatile предписывает компилятору генерировать код, в котором переменная с этим атрибутом хранится в памяти, а не в регистре. Полезно для организации корректного доступа к переменной, когда она используется несколькими нитями.

```
#include <winsock2.h>
#include <stdio.h>
#include "../DATA_BUFSIZE.h"
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
int readn( SOCKET fd, char *bp, int len);
int writen( SOCKET fd, char *bp, int len);
DWORD WINAPI ClientThreadF(LPVOID lpParam);
void printSpeed(double dCurrenReadSpeed, double dAvgDuplexSpeed, int x, int y);
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Listen(SOCKET s,int backlog);
SOCKET Accept(SOCKET s, struct sockaddr* addr,int* addrlen);
HANDLE WINAPI _CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
     SIZE_T dwStackSize,      LPTHREAD_START_ROUTINE lpStartAddress,
      LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);

#define N 64
```

Листинг 14. Заголовочный файл tcpTreadServerF.h для программы в листинге 15.

В программе в бесконечном цикле для каждого принятого функцией Accept соединения клиента создаётся нить, с кодом носителем в виде функции ClientThreadF, код которой приведен в листинге 16.

```
#include "tcpTreadServerF.h"
DWORD prevTimes[65536];
DWORD startTime;
volatile nTransfers=0;
volatile Start=0;

void main(void){
  WSADATA            wsaData;
  SOCKET             ListeningSocket;
  SOCKET             NewConnection;
  struct sockaddr_in       ServerAddr;
  struct sockaddr_in       ClientAddr;
  int              ClientAddrLen;
  int              Port = 5150;
  int              Ret, i=0;
  HANDLE      hThread;
  DWORD       dwThreadId;
  _WSAStartup(MAKEWORD(2,2), &wsaData);
```

```
  ListeningSocket = Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(Port);
  ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  Bind(ListeningSocket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr)) ;
  Listen(ListeningSocket, 5);
  startTime=GetTickCount();
     for (i=0;i<65536;i++)prevTimes[i]=startTime;
  while (1)  {
     ClientAddrLen = sizeof(ClientAddr);
     NewConnection=Accept(ListeningSocket, (struct sockaddr *)&ClientAddr,
                                                              &ClientAddrLen);
     myprintf("Принят клиент: %s:%d\n", inet_ntoa(ClientAddr.sin_addr), ntohs(ClientAddr.sin_port));
     hThread = CreateThread(NULL, 0, ClientThreadF, (LPVOID)NewConnection, 0, &dwThreadId);
     CloseHandle(hThread);
  }
}
```

Листинг 15. Код многопоточного эхо сервера tcpTreadServerF.c для пакетов фиксированной длины

Число экземпляров в памяти функции ClientThread равно числу активных клиентов. В начале функции в переменной startTime запоминается время старта прогаммы и оно заносится во все элементы масива prevTimes. За счёт глобального флага Start это производится один раз для всего сервера. Далее в функции выделяется память Buffer для обмена с клиентом. Далее в бесконечном цикле сервер функцией readn читает пакет от клиента и отдаёт его функцией writen обратно. В переменную currentTime записываем время конца обменом пакета, а в переменной durationOnce вычисляем длительность обмена именно для этого сокета lpParam. В переменной durationTotal вычисляется длительность работы сервера Переменная nTransfers является общей для всех экземпляров функции ClientThread, поэтому работать с ней надо аккуратно. Выражение InterlockedIncrement(&nTransfers) корректно вычисляет количество пакетов (это суть nTransfers++). В переменной dCurrenSpeed вычисляем текущую скорость обмена в мегабитах в секунду. В переменной dAvgSpeed вычисляем среднюю скорость обмена. Далее переустанавливаем начало обмена пакетом prevTimes[(int)lpParam]=currentTime для этого сокета

При разрыве или окончании соединения цикл разрывается (по оператору break), сокет закрывается, память освобожлается и нить завершается.

```
DWORD WINAPI ClientThreadF(LPVOID lpParam){
    DWORD currentTime, durationTotal, durationOnce;
    double dCurrenSpeed, dAvgSpeed;
    struct sockaddr_in ClientAddr;
    SOCKET   sock    =(SOCKET)lpParam;
    int Ret, caSize = sizeof(ClientAddr);
    char *Buffer = (char*)GlobalAlloc(GPTR,DATA_BUFSIZE);
    getpeername(sock, (struct sockaddr *)&ClientAddr, &caSize);
    if(!Start){
            int i;
            Start=1;
            startTime=GetTickCount();
            for (i=0;i<65536;i++)prevTimes[i]=startTime;
    }     while(1){
            Ret=readn(sock, Buffer, DATA_BUFSIZE);
            if(Ret==0){
                    myprintf("Клиентом %s:%d закрыл соединение.\n",inet_ntoa(ClientAddr.sin_addr),
                            ntohs(ClientAddr.sin_port));
                    break;
            }
            if(Ret==SOCKET_ERROR){
                            myprintf("Ошибка readn. Клиент %s:%d.\n",inet_ntoa(ClientAddr.sin_addr),
                                    ntohs(ClientAddr.sin_port));
                    break;
            }
            Ret = writen(sock, Buffer, DATA_BUFSIZE);
            if ( Ret == SOCKET_ERROR){
                    myprintf("Ошибка writen. Клиент %s:%d.\n",inet_ntoa(ClientAddr.sin_addr),
```

```
                                        ntohs(ClientAddr.sin_port));
                    break;
                }
                currentTime=GetTickCount();
                durationOnce=currentTime-prevTimes[(int)lpParam];
                durationTotal=currentTime-startTime;
                InterlockedIncrement(&nTransfers);
                if(durationOnce)
                        dCurrenSpeed=(1000.0*2*DATA_BUFSIZE*8.0)/(1024.0*1024.0*durationOnce);
                if(durationTotal)
                     dAvgSpeed=(1000.0*nTransfers*8.0*DATA_BUFSIZE*2.0)/
(1024.0*1024.0*durationTotal);
                if(durationOnce&&durationTotal)
                        printSpeed(dCurrenSpeed, dAvgSpeed, 0, 0);
                prevTimes[(int)lpParam]=currentTime;
        }
        closesocket(sock);
        GlobalFree(Buffer);
        ExitThread(0);
    }
```

Листинг 16. Функция ClientThreadF.

Для клиента в непрерывном режиме работы не предусмотрен механизма завершения работы. Для сервера он также отсутствует. Мы вынуждены будем завершать программы либо путём закрытия консольного окна либо нажатием комбинации клавиш Ctrl+C. Это приведёт к тому, что все вычислительные ресурсы, которые взяла себе программа (сокеты и буфера) не будут освобождены и обявлены как свободные для их дальнейшего использования операционной системой. Произойдёт так называемая утечка памяти. Если она значительная, то опреационная система станет медленно работать и единственным средством восстановления её нормальной работы является перегрузка. **Помните об этом.**

Однопоточный параллельный эхо сервер TCP с использованием функции select

Можно предложить алгоритм работы параллельный TCP эхо сервера, реализованный в виде одного потока выполнения:

1. Создать сокет и привязать его к порту. Добавить сокет в множество fd_set сокетов, через которые может осуществляться ввод/вывод.
2. Использовать функцию select для получения информации о готовности существующих сокетов к вводу/выводу.
3. Если готов первоначальный сокет, использовать функцию accept для получения очередного запроса на установление соединения и добавить новый сокет в множество сокетов, через которые может осуществляться ввод/вывод.
4. Если готов сокет, отличный от первоначального, использовать функцию recv для получения очередного запроса, сформировать ответ и передать ответ клиенту с использованием функции send.
5. Продолжить обработку запросов, начиная с приведенного выше этапа 2.

Код для параллельный TCP эхо сервера tcpSelectServer, реализованный в виде одного потока выполнения с использованием функции select приведён в листинге 10.

```
WSADATA            wsaData;
SOCKET             ListeningSocket;
struct sockaddr_in        ServerAddr, ClientAddr;
int ClientAddrLen, Port = 5150, Ret;
fd_set    rfds, afds;
WSAStartup(MAKEWORD(2,2), &wsaData);
ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(Port);
ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(ListeningSocket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr));
listen(ListeningSocket, 5);
FD_ZERO(&afds);
FD_SET(ListeningSocket, &afds);
myprintf("Ожидаем соединение на порту %d.\n", Port);
```

```
while (1) {
        unsigned fdndx;
        memcpy(&rfds, &afds, sizeof(rfds));// rfds = afds;
        if (select(FD_SETSIZE, &rfds, (fd_set *)0, (fd_set *)0,
                        (struct timeval*)0) == SOCKET_ERROR){
                myprintf("Ошибка select %s\n", encodeWSAGetLastError(WSAGetLastError()));
                continue;
                }
        if (FD_ISSET(ListeningSocket, &rfds)) {
                SOCKET          ssock;
                ClientAddrLen = sizeof(ClientAddr);
                ssock = accept(ListeningSocket, (struct sockaddr *)&ClientAddr,
                         &ClientAddrLen);
                myprintf("Принят клиент: %s:%d\n", inet_ntoa(ClientAddr.sin_addr),
                ntohs(ClientAddr.sin_port));
                        FD_SET(ssock, &afds);
                }

        for (fdndx=0; fdndx<rfds.fd_count; ++fdndx){
                SOCKET fd = rfds.fd_array[fdndx];
                if (fd != ListeningSocket && FD_ISSET(fd, &rfds)){
                        ClientAddrLen = sizeof(ClientAddr);
                        getpeername(fd, (struct sockaddr*)&ClientAddr,
                                &ClientAddrLen);
                        Ret = TCPechoS(fd);
                        if ( Ret == 0 || Ret== SOCKET_ERROR) {
                                closesocket(fd);
                                FD_CLR(fd, &afds);
                                myprintf("Связь с клиентом %s:%d прекращена. \n",
                                        inet_ntoa(ClientAddr.sin_addr),
                                        ntohs(ClientAddr.sin_port));
                                }
                        else
                        myprintf("Отражено %d байт от клиента %s:%d.\n",
                                Ret-1,inet_ntoa(ClientAddr.sin_addr), ntohs(ClientAddr.sin_port));

                        }
                }//for
        } //while
WSACleanup();
myprintf("Нажмите Enter для завершения.\n");
getchar();
}
```

Листинг 17. Код для параллельный TCP эхо сервера tcpSelectServer с использованием функции select

Select проверяет готовность сокетов из множества rfds в бесконечном цикле. Далее с помощью макроса FD_ISSET проверяем имеется ли в числе готовых сокетов прослушивающий сокет ListeningSocket. Если имеется, то это значит, что присутствует запрос на соединение от клиента. Мы принимаем этот запрос функцией accept и помещаем новый сокет ssock в множество активных сокетов afds. Далее в цикле for перебираем все сокеты о готовности которых сообщил select и если найдётся готовый сокет fd, отличный от прослушивающего, то мы вызываем ранее рассмотренную в листинге 7 серверную функцию эха TCPechoS(fd. Проверяя код возврата Ret этой функции можем установить факт отключения клиента или обрыва связи. В это случае закрываем сокет и удаляем его из множества всех активных сокетов afds.

Собрав и отладив параллельный эхо TCP сервера tcpSelectServer, можно убедиться, что к нему можно подсоединяться несколькими старыми эхо клиентами tcpSclient.

Контрольные вопросы.

1. Как в протоколе TCP обеспечивается надёжность передачи?
2. Как в протоколе TCP обеспечивается правильную последовательность передачи пакетов?
3. Какую роль играет в TCP окно?
4. Что такое MTU и какое отношение оно имеет к фрагментации пакетов?

5. Что такое MSS м какую роль оно играет?
6. Зачем нужен параметр TCP масштабирования окна?
7. Опишите процедуру установки соединения в TCP.
8. Опишите процедуру закрвтия соединения в TCP.
9. Что такое активное и пасивное закрытие сокета?
10. Укажите на диаграмме состояний TCP на рис.1 какие состояния сокет проходит при открытии на стороне клиента и сервера.
11. Укажите на диаграмме состояний TCP на рис.1 какие состояния сокет проходит при пасивном и активном закрытии
12. Покажите на рис. 2, что если целью TCP соединения было отправить запрос, занимающий один сегмент, и получить ответ, также занимающий один сегмент, то при использовании TCP будет задействовано всего 8 пакетов.
13. Какие две задачи решает состояние TIME_WAIT?
14. Чем отправка пакетов по TCP отличается от отправки по UDP?
15. Что явилось причиной написания функций readn и writen?
16. Что явилось причиной написания функции readline?
17. С помощью листингов 5 и 7 объясните логику работы последовательного TCP эхо сервера.
18. С помощью листингов 4 и 6 объясните логику работы последовательного TCP эхо клиента.
19. Обясните, что такое в Win32 API нить, как её создать и как ей передать параметры.
20. В чём идея создания параллельный сервера TCP с использованием нитей?
21. С помощью листингов 8 и 9 объясните логику работы параллельного TCP эхо сервера с использованием нитей.
22. С помощью листингов 10 -13 объясните логику работы TCP эхо клиента для пакетов фиксированной длины и измерения скорости.
23. С помощью листингов 14 -16 объясните логику работы TCP эхо сервера для пакетов фиксированной длины и измерения скорости
24. Почему с общими переменными для нескольких нитей следует работать особым образом.
25. Назовите средства работы с общими переменными для нескольких нитей
26. Как работает функция select.
27. Когда сокет отмечается функцией select, как читаемый?
28. Когда сокет отмечается функцией select, как записываемый?
29. Какие существуют макросы для работв с множеством сокетов, используемых в функции select?
30. Опишите алгоритм работы параллельный TCP эхо сервера с использованием функции select.
31. С помощью листинга 17 объясните логику работы параллельного TCP эхо сервера с использованием функции select.

Задания

1. Для просмотра состояний TCP удобно воспользоваться утилитой с графическим интерфейсом tcpview.exe, которую можно взять на сайте sysinternals.com. Запусти эту утилиту.

Загрузим сервер tcpserver из первой лабораторной работы в Visual Studio 2008 и остановим его отладку на контрольной точке сразу после после вызова функции accept. Утилита tcpview покажет для tcpserver состояние TCP равное LISTENING (слушаю).

Загрузим клиент tcpclient из первой лабораторной работы в Visual Studio 2008 и остановим его отладку на контрольной точке сразу после после вызова функции connect. Утилита tcpview покажет и для tcpserver и для tcpclient состояние TCP равное ESTEBLISHED (установлено).

Выполним на tcpclient активное закрытие с помощью функции closetsocket. В экземпляре Visual Studio 2008 с tcpserver функции closetsocket выполнять не будем. Утилита tcpview покажет для tcpclient состояние TCP равное FIN_WAIT_2 и для tcpserver состояние TCP равное CLOSE_WAIT.

Выполним на tcpserver активное закрытие с помощью функции closetsocket. Сокет tcpserver закроется, а сокет tcpserver перейдёт в состояние TIME_WAIT и через несколько минут закроется.

Аналогичные результаты получим при активном закрытии tcpserver.

Если приложение закрыть, то его сокет перейдёт в состояние FIN_WAIT_2, а другая сторона перейдёт в состояние CLOSE_WAIT. Если затем закрыть процесс другой стороны, то его сокет также закроется, а сокет исходной стороны будет некоторое время находиться в состоянии TIME_WAIT.

2. Остановим сервер tcpserver из первой лабораторной работы на контрольной точке в любом месте после accept. Остановим клиент на контрольной на функции send. Остановим сервер, завершив отладку.

Сокет сервера перейдёт в состояние FIN_WAIT_2, а сокет клиента в состояние CLOSE_WAIT Убедитесь в этом.

Нажмём F5 и выполним функцию send на клиенте. Успешно! Но сервера нет! То есть информация ушла в никуда. Пред работой с сокетом надо проверять живой ли он. Попробуйте это сделать с помощью функции select. Или так. Следующий фрагмент кода возвращает в переменной optVal время соединения сокекта Socket в секундах или 0xffffffff (-1), если сокет не соединён.

```
DWORD optVal;
int optLen=sizeof(optVal);
getsockopt(Socket, SOL_SOCKET, SO_CONNECT_TIME, (char*)&optVal, &optLen);
```

3. На основании листингов 4, 5, 6 и 7 напишите и отладьте программы последовательного TCP клиента tcpSclient и сервера tcpSserver. Получите и сохраните скриншоты его работы, типа изображённых на рис. 3 и 4. Запустите клиент и сервер на разных машинах. Воспользуйтесь проектами tcpSserver и tcpSclient решения lab2 из архива src.

4. На основании листингов 8 и 9 напишите и отладьте программы параллельного TCP сервера tcpTreadServer. Воспользуйтесь проектом tcpTreadServer решения lab2 из архива src. Снабдите кода параллельного TCP сервера диагностикой для вызовов функций. Запустите сервер tcpTreadServer и несколько клиентов tcpSclient. Убедитесь, что всё работает и, что при принудительном закрытии любого клиента сервер не падает. Запустите клиенты и сервер на разных машинах.

5. На основании листингов 10 – 16 напишите и отладьте программы эхо клиентов и сервера для пакетов фиксированной длины и измерения скорости (см. проекты ClientF и tcpTreadServerF в *).)

6. На основании листинга 17 напишите и отладьте программы параллельного TCP сервера tcpSelectServer. Запустите сервер tcpSelectServer и несколько клиентов tcpSclient. Убедитесь, что всё работает и, что при принудительном закрытии любого клиента сервер не падает. Запустите клиент и сервер на разных машинах. Воспользуйтесь проектом tcpSelectServer решения lab2 из архива src.

7. Возьмите прогаммы из пункта 5. Запустите сервер на одном компьютере, а несколько клиентов на других. Постройте графики зависимостей скоростей обмена от длины пакета DATA_BUFSIZE_INT (1, 10, 100, 1000, 10000 … 1000000) и числа клиентов 1, 2, 4, 8, 16. Рекомендация – перенесите задание длины пакета в командную строку и следите за загруженностью центрального процессора в диспетчере задач. Не допускайте 100% утилизации процессора. Разнесите клиенты на разные машины.

8. Возьмите прогаммы из пункта 6. Модифицируйте её для работы с пакетами фиксированной длины и добавьте в неё измерение скорости. Запустите сервер на одном компьютере, а несколько клиентов ClientF на другом. Постройте графики зависимостей скоростей обмена от длины пакета DATA_BUFSIZE_INT (1, 10, 100, 1000, 10000 … 1000000) и числа клиентов 1, 2, 4, 8, 16. Сравните с результатами пункта 7. Воспользуйтесь проектом tcpSelectServerF решения lab2 из архива src.

9. Модифицируйте программы из п.5 для работы с пакетами переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом.

10. Модифицируйте программы из п.8 для работы с пакетами переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом.

Лабораторная работа №3. Разработка приложений прикладного уровня для стека протоколов TCP/IP

В прикладной уровень протокола TCP/IP входят такие протоколы, как FTP (file transfer protocol – протокол для передачи файлов, HTTP (hyper text transfer protocol протокол передачи текста), telnet – протокол удалённого доступа с консольным окном, SMTP (simple mail transfer protocol – простой протокол для передачи почты), POP3(post office protocol – протокол почтового офиса) и многие другие.

Информация по протоколам передаётся в виде пакетов. Пакеты передаются либо через TCP соединение либо инкапсулируются в UDP датаграммы.

Пакет состоит из заголовка и данных. Каждый протокол имеет свой формат пакета. Форматы пакетов стандартизированы и их официальное описание приводится в стандартах RFC (request for comment), которые легко можно найти в Интернет в том числе и на русском языке.

Обмен данными между приложениями прикладного уровня осуществляется по технологии клиент-сервер. Сервер ждёт запросов на обслуживание от клиентов в виде пакетов определённого формата, слушая определённый порт. Получив запрос, сервер его анализирует и выполняет, оформляя ответ в виде пакетов определённого формата.

Например HTTP-сервер (веб или интернет сервер типа IIS или Apache) слушает порт 80 и ждёт от HTTP-клиентов (броузеров типа интернет проводника) HTTP-запросы. Получив HTTP-запрос, HTTP-сервер его анализирует, выполняет и оформляет ответ в виде HTTP-ответа.

И HTTP-сервер и HTTP-клиент являются TCP-серверами и клиентами соответственно, построение которых мы начали изучать. То есть зная формат протокола HTTP, мы сможем построить свой интернет сервер и проводник.

Можно выкачать из Интернет спецификацию HTTP, изучить и начать программировать. Но для учебных целей подойдёт другой подход. У нас уже есть TCP-сервер (смотри листинги 14 и 15 работы 2), который использует функцию readn (смотри листинги 1 работы 2). Временно вставьте в код функции readn перед оператором if (rc == 0) фрагмент кода, выводящих на консоль запрос клиента

```
bp[rc]='\0';
puts(bp);
getchar();
```

Запустите наш сервер. Запустите интернет проводник, отключите в нём прокси (!!! Обязательно) и набирите в нём URL http://localhost:5150/a/b/c.def, где 5150 порт нашего сервера. В консольном окне нашего сервера вы увидите, что-то типа изображённого на рисунке 1. Не трудно догадаться, что это HTTP-запрос от интернет проводника. Нажмите мышью в левом верхнем углу консольного окна, выберите в системном меню окна пункт Edit_>Mark, далее клавиатурой выделите весь текст и нажмите Enter. Откройте блокнот и втавьте содержимое кармана.

```
GET /a/b/c.def HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-ms-applic
ation, application/x-ms-xbap, application/vnd.ms-xpsdocument, application/xaml+x
ml, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword,
 application/x-shockwave-flash, */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.2; Trident/4.0; MRSP
UTNIK 2, 0, 1, 56 SW; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2
152; .NET CLR 3.5.30729)
Accept-Encoding: gzip, deflate
Host: localhost:5150
Connection: Keep-Alive
```

Рисунок 1. HTTP-запрос

Оставим в HTTP-запрос только информацию, присутствующую в URL

```
GET /a/b/c.def HTTP/1.1
Host: localhost:5150
```

Рисунок 2. Сокращённый HTTP-запрос

Здесь имеются две существенные детали. Первая: строки запроса разделены двумя символами возврата каретки \r (код 13)и перевода строки \n (код 10). Вторая деталь: в конце запроса идёт 4 символа \r\n\r\n. При желании вы это можете проверить в отдадчике, выделив переменную bp, нажав в контекстном меню QuickWatch и набрав в поле Evaluate выражения bp[23], bp[23] и т.д.

Закоментируйте сделанные в нашем сервере изменения в функции readn.

Теперь напишем одноразовый примитивный HTTP-клиент, который отсылает на библиотечный сайт 192.168.10.11:8001 простейший HTTP-запрос, приведенный на рисунке 2. Его код приведен в листинге 1. (См. проект onceClient в решении lab2 архива src). Так как программа одноразовая, то мы не делаем диагностики ошибок. В программе HTTP-запрос находится в строке httpRequest. Программа подсоединяет сокет s к удалённому сокету сервера 192.168.10.11, который слушает порт 8001. Далее она отсылает HTTP-запрос и принимает HTTP-ответ в масив httpResponce. Делает из httpResponce строку языка C, путём

добавления нулевого байта в конец принятых данных и выводит ответ на консоль. Результат работы приведен на рисунке 3. Видим присутствие в HTTP-ответе HTTP-заголовка. Первая строка говорит клиенту, пославшему запрос, что сервер его понял. Вторая строка показывает длину данных. Третья строка говорит клиенту, как ему интерпретировать данные, которые идут после заголовка. Этих трёх строк достаточно для формирования простейшего HTTP-заголовка в HTTP-ответе

```
#include <winsock2.h>
#include <stdio.h>
void main(){
int             Ret;
WSADATA         wsaData;
struct sockaddr_in      ServerAddr;
SOCKET          s;
struct hostent    *host = NULL;
char httpResponce[8192];
char *httpRequest="GET / HTTP/1.1\r\nHost: 192.168.10.11\r\n\r\n";
WSAStartup(MAKEWORD(2,2), &wsaData);
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(8001);
ServerAddr.sin_addr.s_addr = inet_addr("192.168.10.11");
Ret=connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr));
Ret=send( s, httpRequest, strlen(httpRequest), 0);
Ret=recv(s, httpResponce,8192, 0);
httpResponce[Ret]='\0';
puts(httpResponce);
getchar();
}
```

Листинг 1. HTTP-клиент onceClient, отсылающий на сайт http:// 192.168.10.11:8001 простейший HTTP-запрос

```
HTTP/1.1 200 OK
Content-Length: 877
Content-Type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Fri, 25 Sep 2009 22:09:57 GMT

<html><head><META http-equiv="Content-Type" content="text/html; charset=UTF-8"><
```

Рисунок 3. HTTP-ответ сайта http://192.168.10.11:8001 на наш HTTP-запрос

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 877
```

Рисунок 4. Простейший HTTP-заголовок HTTP-ответа

Простейший HTTP-клиент

Создадим HTTP-клиент webclient, который запускается из командной строки командой

webclient хост порт путь

и сохраняет содержимое URL http://хост:порт/путь в файле file папки запуска этой программы. Например, чтобы скачать домашнюю страницу сайта http://192.168.10.11:8001 следует дать команду

webclient 192.168.10.11 8001 /

Это не проводник Интернет. Проводник должен уметь интерпретировать язык HTML. Это скорее примитивная прогамма для закачки файлов.

Код приведен в листинге 2. (См. проект webclient решения lab3 архива src). До вызова функции snprintf нам всё знакомо. Функция snprintf осуществляет форматированный вывод аргументов командной строки для формирования HTTP-запроса. Так для командной строки webclient 192.168.10.11 8001 / получим в буфере buf="GET / HTTP/1.1\r\nHost: 192.168.10.11\r\n\r\n". Порт мы не указываем. Отсылаем HTTP-серверу этот запрос. Далее в цикле while(1) побайтно забираем от сервера заголовок header HTTP-ответа и выводим его на консоль. Далее извлекаем из последнего аргумента командной строки имя fileName запрашиваемого

файла. Если мы не задаём имя, то fileName полагаем раным “file”. Вызываем свою функцию = readnWriteFile, которая читает из сокета оставшиеся байты данных из HTTP-ответа и запишет их в файл fileName в папке старта программы. Попутно программа вычисляет скорость закачки bps.

```
#define GET_CMD "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n" //Шаблон для формирования HTTP-запроса
#define        BSZ 1000000//Буфер для чтения
#include <winsock2.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int Connect( SOCKET s,const struct sockaddr* name, int namelen);
struct hostent    *Gethostbyname(char*);

int readnWriteFile( SOCKET fd, char *file,int len);
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);

void main(int argc, char **argv){
unsigned          gBytesSent=0,  gStartTime=0,      tick;
double  bps;
WSADATA              wsaData;
SOCKET              s;
struct sockaddr_in ServerAddr, ca;
int Ret, n;
char buf[1024];
char fileName[256];
int lmsg = strlen(buf);
int caSize = sizeof(ca);
int  i=0;
struct hostent    *host = NULL;
unsigned iMode = 1;
unsigned char c,  *ContentLength="Content-Length: ";
char seps[]   = "\r\n",seps1[]=" "; //Разделители
char header[2048];

if (argc != 4)   {
      myprintf("Использование: webclient хост порт путь.\n");
      return;
}
_WSAStartup(MAKEWORD(2,2), &wsaData);
s = Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(atoi(argv[2]));
ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
if (ServerAddr.sin_addr.s_addr == INADDR_NONE){
      host = Gethostbyname(argv[1]);
      CopyMemory(&ServerAddr.sin_addr, host->h_addr_list[0],           host->h_length);
}
Connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr));
n = _snprintf(buf, sizeof(buf), GET_CMD, argv[3], argv[1]);//Формируем HTTP-запрос
Ret = send(s, buf, strlen(buf), 0);
while(1){
      Ret = recv(s,&c,1,0);
      header[i++]=c;
      if (c!='\r')
                  continue;
      Ret = recv(s,&c,1,0);
      header[i++]=c;
      if (c!='\n')
                  continue;
```

```
        Ret = recv(s,&c,1,0);
        header[i++]=c;
        if (c!='\r')
                continue;
        Ret = recv(s,&c,1,0);
        header[i++]=c;
        if (c!='\n')
                continue;
        break;
}
header[i]='\0';
puts(header);
{
int n=strlen(argv[3]);
i=0;
        while(argv[3][n]!='/')
                fileName[i++]=argv[3][n--];
        --i;
        for(n=0;n<i/2;n++){
                char t;
                t=fileName[n];
                fileName[n]=fileName[i-n];
                fileName[i-n]=t;
        }
}
if(!strlen(fileName))
        strcpy(fileName,"file");
gStartTime = GetTickCount();
Ret = readnWriteFile(s, fileName);
closesocket(s);
WSACleanup();
myprintf("Нажмите Enter для завершения.\n");
tick = GetTickCount();
bps = Ret*1000.0 / (1024.0*1024.0*(tick - gStartTime));
printf("Read speed: %.3f Mbps.\n", bps);
getchar();
}
```

Листинг 2. Простейший HTTP-клиент

Рассмотрим функцию readnWriteFile, код которой приведен в листинге 3. Функция CreateFile или создаёт или открывает нам файл file и пишет туда данные из сокета fd. Она возвращает число записанных байт. В тексте программы даны комментарии для использованных параметров этой функции. Далее логика функции следует логике функции readn, дополняя её тем, что очередную считанную из сети порцию данных мы дописываем в конец файла file с помощью функции WriteFile.

```
int readnWriteFile( SOCKET fd, char *file){
int cnt;
int rc, max=0;
DWORD dwResult;
static char *buf;

HANDLE hFile = CreateFile(file,              // имя
                GENERIC_WRITE,          // открыть для записи
                0,                      // не разделять с другими процессами
                NULL,                   // безопастность по умолчанию
                CREATE_ALWAYS,          // переписать существующий
                FILE_ATTRIBUTE_NORMAL,  // обычный файл
                NULL);
if (hFile == INVALID_HANDLE_VALUE){
        printf("Could not open file (error %d)\n", GetLastError());
        return -1;
}
buf= (char*)GlobalAlloc(GPTR,           BSZ);
```

```
if(!buf) printf("GlobalAlloc error %d\n", GetLastError());
cnt = 0;
do{
      rc = recv( fd, buf, BSZ, 0 );
      if (rc == SOCKET_ERROR){
                  myprintf("Ошибка recv %s\n", encodeWSAGetLastError(WSAGetLastError()));
                  cnt= SOCKET_ERROR;
                  break;
      }
      if(!WriteFile (hFile, buf, rc, &dwResult, NULL)) {
       printf("Could not write to file (error %d)\n", GetLastError());
       cnt= -2;
                  break;
   }
      cnt += rc;
      }
while(rc);
if(!CloseHandle(hFile)){
      printf("Could not CloseHandle(hFile) (error %d)\n", GetLastError());
       return -1;
   }
return cnt;
}
```

Листинг 3. Код функции readnWriteFile

Запустите программу webclient 192.168.10.11 8001 /l/. Откройте в проводнике Интернет файл "file". Вы увидите стартовую страницу сайта http://192.168.10.11:8001/l/.

Простейший HTTP-сервер

Задача сервера принять заголовок (рис.2), найти в нём файл, который следует отправить клиенту, сформировать заголовок (рис.4) HTTP-ответа, отослать заголовок клиенту и вслед за заголовком отослать файл. Это неполноценный сервер. Так он не будет выполнять скриптов PHP, ASP.NET, CGI и т.д.

За основу возьмём сервер, который вы разработали в работе 2 на основании листинга 8 . В нём можно ничего не менять. Поменяем лишь старое содержимое функции нити ClientThread (листинг 9 работы 2) на новое, представленное в листинге 4. (См. проект httpServer решения lab3 архива src)

Важное значение имеет макрос #define ROOT "/tmp/". Это мы определяем корневую папку нашего сервера. В функции ClientThread мы принимаем HTTP-запрос клиента в buf. Производит проверку на ошибку и с помощью функции getFilenameFromHTTPrequest (листинг 5) извлекаем в fileName имя запрашиваемого файла. Далее в path формируется полный путь к файлу. Файл открывается и читается всё его содержимое за один приём в буфер inBuffer (большие файлы лучше читать частями, благо параметр вызова nBytesToRead позволяет это делать). Далее в HTTPresp формируется заголовок HTTP-ответа клиенту (см. Рис.4). Ответ отсылается и вслед за ним отсылается содержимое файла. Сокет закрывается. Сессия общения с клиентом завершена.

```
#define     BUFSIZE                 4096
#define     PATHNAMESIZE 4096
#define     MAXFILESIZE 65536*16
#define     ROOT            "/tmp/"
char inBuffer[MAXFILESIZE];
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
int readn( SOCKET fd, char *bp, int len);
int   writen(SOCKET fd, const char * bp, int len);
int TCPechoS(SOCKET fd);
int readLine(SOCKET fd, char *buf);
int getFilenameFromHTTPrequest(char *request, char *fileName);

DWORD WINAPI ClientThread(LPVOID fd){  HANDLE hFile;
      char      buf[BUFSIZE],  fileName[PATHNAMESIZE], path[PATHNAMESIZE]=ROOT;
      char HTTPresp[246];
      struct sockaddr_in ca;
      int       inchars, outchars, caSize, rc, nBytesToRead=MAXFILESIZE, nBytesRead;
      caSize = sizeof(ca);
```

```
        rc = recv( fd, buf, BUFSIZE,0);
        if (rc == SOCKET_ERROR){
                getpeername(fd, (struct sockaddr *)&ca, &caSize);
                myprintf("Не могу получить данные от клиента %s:%d. SOCKET_ERROR.\n",
                inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
                return SOCKET_ERROR;
        }
        if ( rc == 0){
                getpeername(fd, (struct sockaddr *)&ca, &caSize);
                myprintf("Клиент %s:%d завершил работу.\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
                return rc;
        }
        getFilenameFromHTTPrequest(buf, fileName);
        strcat(path,fileName);
        hFile = CreateFile(path,              // имя файла
                  GENERIC_READ,         // открываем для записи
                  FILE_SHARE_READ,//чтобы несколько клиентов одновременно могли читать файл
                  NULL,                 // обычная безопастность
                  OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL,  // обычный файл
                  NULL);                // no attr. template
     if (hFile == INVALID_HANDLE_VALUE){
         printf("Could not open file (error %d)\n", GetLastError());
         return -1;
     }
       //Читаем сразу всё
    if (!ReadFile(hFile,  inBuffer, nBytesToRead, &nBytesRead, NULL)){
         printf("Could not read file (error %d)\n", GetLastError());
    }
    inchars=sprintf(HTTPresp,   "HTTP/1.1 200 OK\r\n"
                                                  "Content-Length: %d\r\n"
                                                  "Content-Type: text/html\r\n\r\n",nBytesRead);
    outchars = writen(fd, HTTPresp, inchars);
    outchars = writen(fd, inBuffer, nBytesRead);
    if (outchars == SOCKET_ERROR){
        getpeername(fd, (struct sockaddr *)&ca, &caSize);
        myprintf("Не могу передать данные клиенту %s:%d. SOCKET_ERROR.\n",
        inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
        return SOCKET_ERROR;
    }
  return outchars;
  }
```

Листинг 4. Функция нити HTTP-сервера

```
int getFilenameFromHTTPrequest(char *request, char *fileName){
        int i=5, j=0;
        while(request[i]!=' ')
                fileName[j++]=request[i++];
        fileName[j]='\0';
        //Согласно стандарту URL ряд символов пробел, #, кирилица и пр. заменяется  на
        //последовательность %код_символа. Например, пробел заменяется на %20 . Надо здесь сделать
        //обратное преобразование, но мы это делать не будем,
        return 1;
}
```

Листинг 5. Функция getFilenameFromHTTPrequest

Проверим, что наш сервер работает. Поместим в его корневую папку файл q.html, содержащий простейший HTML-текст

```
<html>HW</html>
```

Листинг 6 . Файл q.html

Наберём в проводнике Интерне URL http://localhost:5150/q.html. Получим в проводнике изображённое на рисунке 5.

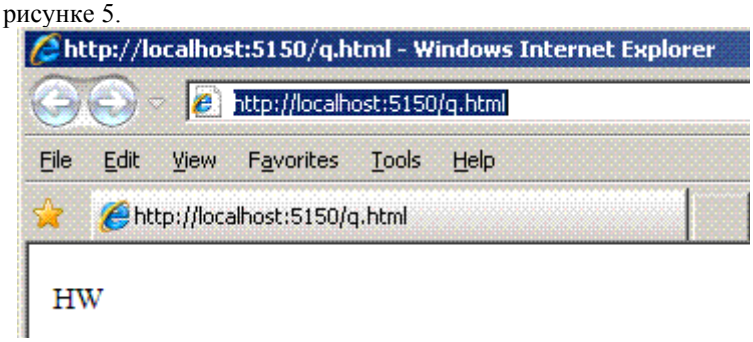

Рисунок 5. Проводник обратился к нашему HTTP-серверу.

Войдём в папку, где расположен webclient и введём команду

Webclient localhost 5150 /q.html

В файле file будет содержимое файла q.html, показанное в листинге 6.

Использование функции TransmitFile для передачи файлов

Микросоыт расширил библиотеку сокетов, добавив в неё новую функцию TransmitFile. Эта функция передаёт данные из файла через соединённый сокет. Она использует менеджер кеша операционной системы и обеспечивает высокопродуктивную передачу содержимого файла в сеть. Мы не будем приводить подробного описания этой функции, а предоставим сразу листинг 6 функции нити HTTP-сервера с использованием функции TransmitFile.

Посмотрим, чем этот листинг отличается от листинга 4. Объявлена пременная ol имеющая тип перекрытия OVERLAPPED. Объявлен массив событий WSAEVENT. В листинге закоментирован код, которым можно воспользоваться, если вы хотите получить системный указатель lpfnTransmitFile на функцию TransmitFile. Отличается предпоследний параметр функции CreateFile. Атрибут FILE_FLAG_OVERLAPPED позволит winsock читать данные из файла без блокировки. Так как winsock будет читать данные из файла строго последовательно, то атрибут FILE_FLAG_SEQUENTIAL_SCAN повысит производительность.

Теперь прейдём к описанию особенностей вызова функции TransmitFile. Вначале надо обнулить структуру перекрытия ol и создать для поля ol.hEvent этой структуры системный объект события с помощью функции WSACreateEvent. Далее осуществляем вызов TransmitFile(s, hFile, 0, 0, &ol, NULL, 0). Система выполнит этот вызов без блокировки и установит событие ol.hEvent, когда вывод реально завершится. Нам нельзя закрывать сокет до завершения вывода. Поэтому мы ждём события ol.hEvent о реальном завершении вывода файла в сеть.

```
DWORD WINAPI ClientThread(LPVOID fd){
    HANDLE hFile;
    char      buf[BUFSIZE],  fileName[PATHNAMESIZE], path[PATHNAMESIZE]=ROOT;
    char HTTPresp[246];
    struct sockaddr_in ca;
    OVERLAPPED ol;
    WSAEVENT ea[1];
    //LPFN_TRANSMITFILE    lpfnTransmitFile;
    //GUID guidTransmitFile = WSAID_TRANSMITFILE;
    //DWORD bytes;
    int         inchars, outchars, caSize, rc;
    SOCKET s=(SOCKET)fd;
    caSize = sizeof(ca);
    rc = recv( s, buf, BUFSIZE, 0 );
    if (rc == SOCKET_ERROR){
                        getpeername(s, (struct sockaddr *)&ca, &caSize);
                        myprintf("Не могу получить данные от клиента %s:%d. SOCKET_ERROR.\n",
                        inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
                        return SOCKET_ERROR;
    }
    if ( rc == 0){
              getpeername(s, (struct sockaddr *)&ca, &caSize);
              myprintf("Клиент %s:%d завершил работу.\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
```

```
                return rc;
        }
        buf[rc]='\0';
        myprintf("%s", buf);
        inchars=sprintf(HTTPresp,           "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n");
        outchars = writen(s, HTTPresp, inchars);
        getFilenameFromHTTPrequest(buf, fileName);
        strcat(path,fileName);

        hFile = CreateFile(path,            // имя
                    GENERIC_READ,
                    FILE_SHARE_READ,      //общий доступ
                    NULL,                 // обычная безопастность
                    OPEN_EXISTING,
                   FILE_FLAG_OVERLAPPED |FILE_FLAG_SEQUENTIAL_SCAN,
                    NULL);                // no attr. template
    if (hFile == INVALID_HANDLE_VALUE){
        printf("Could not open file (error %d)\n", GetLastError());
        return -1;
    }
        /*rc = WSAIoctl(s,SIO_GET_EXTENSION_FUNCTION_POINTER, &guidTransmitFile,
                    sizeof(guidTransmitFile),lpfnTransmitFile,sizeof(lpfnTransmitFile),
                    &bytes,          NULL,          NULL          );
              if (rc == SOCKET_ERROR)             {
                 fprintf(stderr, "WSAIoctl: SIO_GET_EXTENSION_FUNCTION_POINTER faled: %d\n",
                       WSAGetLastError());
                 return -1;
              }
        */
        memset(&ol, 0, sizeof(ol));
       ol.hEvent=WSACreateEvent();

     rc = TransmitFile(s, hFile, 0,  0, &ol,  NULL,  0  );
  //rc = lpfnTransmitFile(s, hFile, 0,  0, &ol,  NULL,  0  );
        if (rc == FALSE)
     {
         if (WSAGetLastError() != WSA_IO_PENDING)        {
                           getpeername(s, (struct sockaddr *)&ca, &caSize);
                           myprintf("Не могу передать данные клиенту %s:%d.TransmitFile failed: %d\n",
                 inet_ntoa(ca.sin_addr),ntohs(ca.sin_port),WSAGetLastError());
            return SOCKET_ERROR;
         }
     }
         ea[0]=ol.hEvent;
         WSAWaitForMultipleEvents(1,ea,TRUE,WSA_INFINITE, FALSE);
       closesocket(s);
       return 0;
}
```

Листинг 7. Функция нити HTTP-сервера с использованием функции TransmitFile.

Контрольные вопросы

1. Какие вы знаете протоколы прикладного уровня TCP/IP?
2. Где можно найти форматы пакетов протоколы прикладного уровня TCP/IP?
3. Изложите суть технологии клиент-сервер.
4. Куда направляются пакты из Интернет проводника при включенном и выключенном прокси сервере?
5. Что содержит HTTP-запрос клиента к серверу?
6. Как мы сумели получить формат HTTP-запроса клиента к серверу?
7. Что содержит HTTP-ответ сервера клиенту?
8. Как мы сумели получить формат HTTP-ответа сервера клиенту?
9. Объясните работу нашего HTTP-клиента.
10. Почему наш клиент нельзя газвать Интернет проводником?
11. Какой цели служит программа из листинга 1?
12. Объясните работу функции readnWriteFile из листинга 3.

13. Объясните работу нашего HTTP-сервера с функцией нити из листинга 4.
14. Объясните работу нашего HTTP-сервера с функцией нити из листинга 6.
15. Объясните назначение вызова функцииWSAWaitForMultipleEvents в листинге 6.
16. Какие недостатки вы видите у клиента и как их устранить?
17. Какие недостатки вы видите у сервера и как их устранить?

Задания

1. Следуя тексту работы, создайте проекта и постройте исполняемый файла HTTP-клиента onceClient (см. проект onceClient решения lab3 архива src).
2. Следуя тексту работы, создайте проекта и постройте исполняемый файла HTTP-клиента webclient (см. проект webclient решения lab3 архива src).
3. Модифицируйте клиент для работы через прокси сервер. Если у Интернет проводника указан прокси сервер, то HTTP-запрос проводника отсылает прокси серверу, а не сайту к которому идёт обоащение. Проанализируйте, что проводник шлёт в сторону прокси, установив в проводнике в качестве прокси свой TCP сервер, для выяснения особенностей HTTP-запроса, направляемого проводником в сторону прокси.
4. Выполнив п.2, обратитесь своим HTTP-клиентом к нескольким сайтам. Посмотрите в Интернет проводнике содержимое полученного файла. Сравните с ориниалом.
5. Следуя тексту работы, постройте два исполняемых файла HTTP-серверов с функциями нити, приведённых в листингах 4 и 6. (см. проекты httpServer и httpServerTransmitFile решения lab3 архива src).
6. Поместите в корень каждого HTTP-сервера произвольный html-файл. Обратитесь к вашим HTTP-серверам из проводника Интернет. Вы должны получить в нём содержимое, подобное изображённому на рисунке 5.
7. Обратитесь к вашим HTTP-серверам из вашего HTTP-клиента webclient для получения файла. Содержимое полученого файла должно совпадать с содержимым запрашиваемого у HTTP-сервера файла.
8. Повторите пункты 6 и 7, разместив клиент и сервер на разных машинах

Лабораторная работа №4. Асинхронные сокеты. Модель, основанная на применении функции WSAAsyncSelect

Теоретическая часть

В Winsock управлению вводом-выводом для Windows-приложениях реализовано с помощью режимов работы и моделей ввода-вывода. Режим сокета определяет поведение функций, работающих с сокетом. Модель сокета описывает, как приложение производит ввод-вывод при работе с сокетом. Модели не зависят от режима работы и позволяют обходить их ограничения.

Режимы работы сокетов

Winsock поддерживает два режима: блокирующий и неблокирующий. В блокирующем режиме функции ввода-вывода, такие как send и recv, перед завершением ожидают окончания операции. В неблокирующем — работа функций завершается немедленно.

При блокировке сокета необходима осторожность, так в как этом режиме любой вызов функции Winsock именно блокирует сокет на некоторое время. Большинство приложений Winsock следуют модели поставщик — потребитель, в которой программа считывает или записывает определенное количество байт и затем выполняет с ними какие-либо операции. Проблема в том, что функция ввода-вывода в сокете может не завершиться никогда. В такой ситуации некоторые программисты могут соблазниться подглядыванием данных (чтение без удаления из буфера), используя флаг MSG_PEEK в функции recv или вызывая ioctlsocket с параметром FIONREAD. Подобный стиль программирования заслуживает резкой критики. Издержки, связанные с подглядыванием, велики, так как необходимо сделать один или более системных вызовов для определения числа доступных байт, после чего все равно приходится вызывать recv для чтения и удаления данных из буфера.

Чтобы этого избежать, следует предотвратить замораживание приложения из-за недостатка данных (из-за сетевых проблем или проблем клиента) без постоянного подглядывания в системные сетевые буферы. Один из методов — разделить приложения на считывающий и вычисляющий потоки (нити), совместно использующие общий буфер данных. Доступ к буферу регулируется синхронизирующим объектом , таким как событие или мьютекс. Целесообразно, чтобы считывающий и вычисляющий потоки создавались отдельно для каждого сокета. Данный вариант не предусматривает масштабирования для большого количества сокетов.

Альтернатива описанному режиму — режим без блокировки. Он несколько сложнее в использовании, но обеспечивает те же возможности, что и режим блокировки, плюс некоторые преимущества. Покажем, как создать сокет и перевести его в неблокирующий режим:

```
SOCKET       s;
unsigned long ul = 1;
int          nRet;


s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIOBIO, (unsigned long *) &ul);
```

Если сокет находится в неблокирующем режиме, функции Winsock завершаются немедленно. В большинстве случаев они будут возвращать ошибку WSAEWOULDBLOCK, означающую, что требуемая операция не успела завершиться за время вызова. Например, вызов recv вернет WSAEWOULDBLOCK, если в системном буфере нет данных Часто функцию требуется вызывать несколько раз подряд, пока не будет возвращен код успешного завершения. Поясним, что означает эта ошибка для некоторых функций.

accept — приложение не получило запрос на соединение;

connect — соединение инициировано;

recv и recvfrom — данные не были приняты;

send, и sendto — в буфере нет места для записи исходящих данных.

Большинство вызовов в неблокирующем режиме будут возвращать ошибку WSAEWOULDBLOC. Непрерывный вызов в цикле функции recv до успешного завершения чтения 200 байт ничуть не лучше, чем вызов recv в блокирующем режиме с флагом MSG_PEEK.

У каждого режима работы сокета свои достоинства и недостатки. Режим блокировки проще концептуально, но в нем сложнее обрабатывать несколько сокетов одновременно или нерегулярные потоки данных С другой стороны, режим без блокировки требует больше кода для обработки ошибки WSAEWOULDBLOCK в каждом вызове. Модели ввода-вывода сокетов помогают приложению асинхронно обрабатывать соединения на одном и более сокетах.

Модели сокетов

Модели помогают приложению управлять несколькими сокетами одновременно в асинхронном режиме. Существуют модели:

- модель, основанная на применении функции select. Эта модель была нами рассмотрена в работе №2;
- модель перекрытого ввод-вывод;
- модель, основанная на применении функции завершения;
- модель, основанная на применении функции WSAAsyncSelect;
- модель, основанная на применении функции WSAEventS'elect;
- модель порта завершения.

Модели ввода-вывода Winsock могут помочь приложению определить, когда сокет доступен для чтения и записи.

Модель WSAAsyncSelect

Эта модель позволяет приложению получать информацию о событиях, связанных с сокетом, при помощи сообщений Windows. Это достигается вызовом функции WSAAsyncSelect после создания сокета.

Прежде чем использовать модель WSAAsyncSelect, приложение должно создать окно, используя функцию CreateWmdow, и процедуру обработки сообщений для этого окна. Можно использовать диалоговое окно с диалоговой процедурой (так как это частный случай окна). Создав инфраструктуру окна, вы вправе создавать сокеты и активизировать уведомления окна о сетевых событиях вызовом функции WSAAsyncSelect. Сетевые события связаны с готовностью сокета к обмену информацией.

При вызове функции WSAAsyncSelect сокет автоматически переходит в неблокирующий режим. В результате ряд функций Winsock при вызове возвратят ошибку WSAEWOULDBLOCK о неготовности сокета к обмену информацией. Чтобы избежать ошибки, приложение должно обработать в функции окна пользовательское оконное сообщение, заданное при вызове WSAAsyncSelect, и показывающее, когда на сокете происходят сетевые события того или иного типа.

Функция WSAAsyncSelect организовывает извещение о сетевых событиях в сокете в виде сообщений Windows. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int WSAAsyncSelect(SOCKET s, HWND hWnd,  unsigned int wMsg,  long lEvent);
```

Параметр s определяет сокет для которого требуется извещение о событиях. Дескриптор окна, которое будет получать сообщения при возникновении сетевых событий, определяется параметром hWnd.

wMsg – сообщение, получаемое окном при сетевых событиях. lEvent – битовая маска, определяющая комбинацию сетевых событий в которых заинтересовано приложение.

Если не произошла ошибка, то функция возвращает ноль, что говорит о том, что интерес приложения к сетевым событиям будет удовлетворен. Иначе функция возвращает SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, WSAENOTSOCK. Расшифровку ошибок смотри в приложении к или в MSDN.

После вызова функуции WSAAsyncSelect библиотека сокетов WS2_32.DLL будет посылать сообщение wMsg окну hWnd при любых сетевых событиях на сокете s, определяемых параметром lEvent.

Функуция WSAAsyncSelect автоматически устанавливает сокет s в неблокирующий режим в независимости от значения параметра lEvent. Для перевода сокета в блокирующий режим нужно вначале вызвать функцию WSAEventSelect с нулевым параметром lEvent и затем вызвать функцию ioctlsocket с соответствующим параметром.

Параметр lEvent конструируется с помощью операции побитового ИЛИ из следующих величин – кодов сетевых событий:

FD_READ – желание получать сообщения о готовности сокета к чтению.

FD_WRITE - желание получать сообщения о готовности сокета к записи.

FD_ACCEPT - желание получать сообщения о готовности сокета к приёму входных соединений.

FD_CONNECT - желание получать сообщения о завершении установки соединения.

FD_CLOSE - желание получать сообщения о закрытии сокета.

FD_QOS и FD_GROUP_QOS связано с качеством обслуживания QOS.

FD_ROUTING_INTERFACE_CHANGE - желание получать сообщения об изменениях маршрутизирующих интерфейсов.

FD_ADDRESS_LIST_CHANGE - желание получать сообщения об изменении списка локальных адресов.

Мы видим, насколько эта функция даёт больше возможностей использовать сетевые события по сравнению с функцией select.

При получении окном сообщения могут возникнуть дополнительные коды ошибки, которые можно извлечь в процедуре окна из фактического параметра lParam с помощью макроса WSAGETSELECTERROR.

Событие FD_CONNECT может привести к кодам ошибок: WSAEAFNOSUPPORT, WSAECONNREFUSED, WSAENETUNREACH, WSAEFAULT, WSAEINVAL, WSAEISCONN, WSAEMFILE, WSAENOBUFS, WSAENOTCONN, WSAETIMEDOUT.

Событие FD_CLOSE может привести к кодам ошибок: WSAENETDOWN, WSAECONNRESET, WSAECONNABORTED.

Вызов функции WSAAsyncSelect для сокета сбрасывает все предыдущие установки осуществлённые этой функцией или функцией WSAEventSelect.

Пример вызова функции для получения извещений в процедуру окна hWnd о событиях чтения и завписи на сокете s:

```
#define WM_SOCKET (WM_USER + 1)
rc = WSAAsyncSelect(s, hWnd, WM_SOCKET, FD_READ|FD_WRITE);
```

Нельзя определить разные сообщения WM_ для разных сетевых событий.

Для отключения окна от получения сетевых событий следует вызвать

```
rc = WSAAsyncSelect(s, hWnd, 0, 0);
```

Такой вызов WSAAsyncSelect равно как и закрытие сокета s функцией closesocket немедленно прекратит передачу сообщений окну о событиях на сокете, но возможно, что в очереди сообщений приложения остались необработанные сообщения о событиях на сокете.

Сокет, созданный функцией accept, имеет те же свойства, что и слушающий сокет, указанный в аргументах функции accept. Следовательно, множество событий для слушающего сокета также применится к сокету, созданному функцией accept. Например, если слушающий сокет имеет события WSAAsyncSelect в виде FD_ACCEPT, FD_READ и FD_WRITE, то и новый сокет, принятый на этом прослушивающем сокете также будет реагировать на события FD_ACCEPT, FD_READ и FD_WRITE в виде того же WM_ сообщения wMsg. Если это не устраивает, то для нового сокета, принятого функцией accept, следует отдельно вызвать функцию WSAAsyncSelect с желаемым множеством событий и сообщением WM_.

Когда на сокете s случится заявленное в lEvent событие, окно hWnd приложения получит сообщение wMsg. Параметр wParam сообщения будет равен дескриптору сокета, на котором возникло событие. Младшее слово параметра lParam сообщения содержит код возникшего сетевого события, а старшее слово содержит код ошибки, определённый в Winsock2.h и нет смысла вызывать функцию WSAGetLastError, которая к тому же в этой ситуации может дать неверный код ошибки. Код события и код ошибки можно выделить из lParam макросами

```
#define WSAGETSELECTEVENT(lParam)      LOWORD(lParam),
#define WSAGETSELECTERROR(lParam)      HIWORD(lParam)
```

соответственно.

Хотя WSAAsyncSelect может быть вызвана с интересом ко многим событиям, окно приложения получит отдельное сообщение для каждого сетевого события.

Как и в случае функции select функция WSAAsyncSelect используется для определения, когда функции приёма/передачи типа recv или send гарантировано выполнятся с успехом и без блокировки. Тем не менее, надёжное приложение должно быть готовым к тому, что оно, получив сообщение о сетевом событии, и вызвав функцию Winsock, получит код ошибки WSAEWOULDBLOCK о неготовности данных. Например, возможна следующая последовательность событий:

Данные возникли на сокете s и Winsock направил приложению соответствующее сообщение wMsg. Приложение тем временем обрабатывает другое сообщение и вызывает функцию ioctlsocket(s, FIONREAD...), которая показывает наличие данных для чтения. Приложение читает данные. Со временем очередь доходит до сообщения wMsg. Приложение пытается читать данные и получает ошибку WSAEWOULDBLOCK о том, что данных нет.

Winsock не будет наводнять приложение сообщениями об определённом сетевом событии. При возникновении сетевого события генерируется одно сообщение, если только приложение не сделает повторного вызова некоей возобновляющая функции Winsock, а именно:

Вызов функций recv, recvfrom, WSARecv или WSARecvFrom приведёт к повторной генерации сообщения о сетевом событии FD_READ.

Вызов функций send, sendto, WSASend или WSASendTo приведёт к повторной генерации сообщения о сетевом событии FD_WRITE.

Вызов функций accept или WSAAccept приведёт к повторной генерации сообщения о сетевом событии FD_ACCEPT, если толко не возникнет ошибка WSATRY_AGAIN.

Нет функций, приводящих к повторной генерации сообщений для событий FD_CONNECT и FD_CLOSE.

Повторная генерация проводится, даже если вызов соответствующей возобновляющей функции был неудачен.

Для событий FD_READ, FD_OOB и FD_ACCEPT вызов возобновляющей функции приводит к повторной генерации сообщений, если всё ещё имеются условия для существования этих событий. Это позволяет приложениям быть событийно управляемыми и не иметь дело с количеством данных, которое возникает в каждый момент времени. Рассмотрим следующую последовательность.

Сетевой транспортный стек получил 100 байтов данных на сокете s и Winsock отправил приложение сообщение FD_READ. Приложение прочитало 50 байт с помощью функции recv(s, , 50, 0). Поскольку остались данные для чтения, Winsock отправит приложению ещё одно сообщение FD_READ

В ответ на сообщение FD_READ приложение может не читать все данные доступные для чтения. Пока останутся непрочитанные данные, приложение будет получать сообщение FD_READ. Можно отключить генерацию сообщений для события FD_READ, повторно вызвав функцию WSAAsyncSelect, не указав в ней события FD_READ.

Рассмотрим следующую последовательность.

Приложение вызывает функцию listen. Поступил, но ещё не принят запрос на соединение. Приложение вызывает функцию WSAAsyncSelect, указывая, что оно требует получения сообщений о событии FD_ACCEPT. Так как соответствующее условие имеет место, то Winsock направляет приложению сообщение о событии FD_ACCEPT.

Сообщение о событии FD_WRITE отправляется, когда сокет соединяется (после сообщения о FD_CONNECT, если это зарегистрировано) с помощью функции connect или WSAConnect либо принимается с помощью функции accept или WSAAccept и далее, когда операция отправки закончится с ошибкой WSAEWOULDBLOCK и буфер для отправки станет доступным. Следовательно, приложение может полагать, что отправка данных доступна начиная от первого сообщения о событии FD_WRITE и до тех пор, пока отправляющая функция не вернёт код ошибки WSAEWOULDBLOCK. После такой ошибки приложение будет извещено о возможности новой отправки с помощью нового сообщения о событии FD_WRITE.

Код ошибки в сообщении о событии FD_CLOSE показывает, был ли сокет закрыт корректно или некорректно. Если код ошибки равен нулю, то закрытие было корректным. Если код ошибки равен WSAECONNRESET, то соединение было закрыто некорректно. Это касается только сокетов, ориентированных на соединение, таких как SOCK_STREAM.

Сообщение о событии FD_CLOSE отправляется, когда сокет переходит в состояние TIME_WAIT или CLOSE_WAIT. Это является результатом того, что на противоположном конце соединения была вызвана функция shutdown с опцией о прекращении передачи или функция closesocket.

Сообщения о событии FD_CLOSE должно быть послано Winsock после прочтения всех данных из сокета, но приложение после получения сообщения о событии FD_CLOSE должно проверить сокет на наличие там оставшихся данных, чтобы избежать их потери.

При корректном закрытии сокета приложение получит сообщение о событии FD_CLOSE после того, как все данные будут получены. Свидетельством конца корректного закрытия является отсутствие сообщении о событии FD_READ.

Приведём итоговые сведения о том, когда возникают сообщения о сетевых событиях.

FD_READ:

1.При вызове WSAAsyncSelect, если есть данные для получения.
2.При возникновении данных и если FD_READ ещё не был отправлен.
3.После вызова recv или recvfrom, если ещё есть данные для получения.

FD_WRITE:

1.При вызове WSAAsyncSelect, если возможны вызовы send или sendto.
2.После вызова connect или accept и установки соединения.
3.После неудачного вызова send или sendto с ошибкой WSAEWOULDBLOCK, если последующий вызов send или sendto будет успешен.

FD_ACCEPT:

1.При вызове WSAAsyncSelect, если имеется запрос на соединение к функции accept.
2.Если имеется запрос на соединение к функции accept и, если FD_ACCEPT ещё не был отправлен.
3.После вызова accept, если есть другой запрос на соединение к функции accept.

FD_CONNECT:

1.При вызове WSAAsyncSelect, если соединение уже установлено.
2.После вызова connect и установки соединения, даже если соединение установится немедленно, что типично для датаграмных протоколов.
3.После вызова connect или WSAConnect на неблокирующем сокете, ориентированном на соединение. Вначале эти вызовы вернут ошибку WSAEWOULDBLOCK, но сетевые операции

будут продолжаться. В не зависимости от успеха или неудачи этих вызовов сообщение FD_CONNECT будет отправлено. Следует далее проверять код ошибки в сообщении FD_CONNECT на предмет успешности соединения.

FD_CLOSE: Имеет место только для сокетов, ориентированных на соединение, например SOCK_STREAM.

1.Когда вызвана функция WSAAsyncSelect, если соединение было уже закрыто.

2.Когда удалённая система инициировала корректное закрытие и нет данных для чтения. Если данные получены, то FD_CLOSE не возникает, пока данные не будут прочитаны.

3.После того, как локальная система инициировала корректное закрытие с помощью функции shutdown и удалённая система подтвердила запрос на закрытие, например пакетом TCP FIN и когда на локальной системе нет данных для чтения.

4.Когда удалённая система разрывает соединение, посылая, например пакет TCP RST и lParam будет содержать код ошибки WSAECONNRESET.

FD_CLOSE не передаётся после вызова функции closesocket.

Практическая часть

TCP сокеты SOCK_STREAM, ориентированы на соединение и передают данные в виде непрерывного потока байт. Для разделения потока на пакеты в приложении должен быть определён формат пакета. Во второй работе в качечтве разделителя пактов использовался нулевой байт. В этой работе и в ряде последующих, мы определим пакеты фиксированной длины в байтах DATA_BUFSIZE.

Поставим задачу разработать эхо сервер и клиент к нему с помощью модели WSAAsyncSelect с использованием фиксированной длины пакетов. Предпишем серверу следующую логику работы: в цикле для каждого клиента выбрать из сокета весь пакет (DATA_BUFSIZE байт), а затем отослать его полностью обратно клиенту. Предпишем клиенту такую логику. Подготовить пакет. Полностью его отправить. Дождаться ответа и выбрать из сокета весь пакет. Сравнить полученный пакет с переданным. Затем подготовить очередной пакет для отправки.

Начнём с сервера. Его листинг приведен в листинге 1. Мы используем обвёртки для некоторых функций. Напомним, что внутри обвёрток находится диагностический код и, как правило, имя обвёртки и её семантика совпадает с оригинальной функцией, но начинается с большой буквы или подчёркивания.

```
#include <winsock2.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "../DATA_BUFSIZE.h"
#define PORT 5150
#define WM_SOCKET (WM_USER + 1)
typedef struct _SOCKET_INFORMATION {
    WSABUF DataBuf;
    CHAR Buffer[DATA_BUFSIZE];
    char *bp;
    SOCKET Socket;
    DWORD leftToRead;
    DWORD leftToWrite;
    struct _SOCKET_INFORMATION *Next;
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;

int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int _WSAAsyncSelect( SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Listen(SOCKET s,int backlog);
SOCKET Accept(SOCKET s, struct sockaddr* addr,int* addrlen);

void CreateSocketInformation(SOCKET s);
LPSOCKET_INFORMATION GetSocketInformation(SOCKET s);
void FreeSocketInformation(SOCKET s);

HWND MakeWorkerWindow(void);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

```
void forFD_READ(WPARAM wParam, HWND hwnd);
void forFD_WRITE(WPARAM wParam, HWND hwnd);
int myprintf (const char *lpFormat, ... )

LPSOCKET_INFORMATION SocketInfoList;

void main(void)
{
  MSG msg;
  SOCKET Listen;
  struct sockaddr_in InternetAddr;
  HWND W;
  WSADATA wsaData;

W = MakeWorkerWindow();
_WSAStartup(0x0202, &wsaData);
ListenS = Socket (PF_INET, SOCK_STREAM, 0);
_WSAAsyncSelect(ListenS, Window, WM_SOCKET, FD_ACCEPT);
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(PORT);

Bind(ListenS, (struct sockaddr*) &InternetAddr, sizeof(InternetAddr));
Listen(ListenЫ, SOMAXCONN);

while(GetMessage(&msg, NULL, 0, 0)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

Листинг 1. Асинхронный эхо сервер модели WSAASyncselect.

Подключаемый файл DATA_BUFSIZE.h содержит размеры буферов.

```
#define DATA_BUFSIZE_INT 65536
#define DATA_BUFSIZE DATA_BUFSIZE_INT*sizeof(int)
```

Определяем порт для портPORT для прослушивания равным 5150 и пользовательское сообщение Windows WM_SOCKET.

Далее определён типы данных SOCKET_INFORMATION и LPSOCKET_INFORMATION в виде списка и указателя к нему, в котором будет храниться информация о каждом сокете, к которому подсоединён клиент. Поле DataBuf типа WSABUF структуры SOCKET_INFORMATION необходимо для организации ввода и вывода через сокет с использованием функций WSARecv и WSASend. Эти функции являются расширением WinSock стандартных функций recv и send и более подробно рассмотрены в следующей работе. Данные от клиента принимаются и возвращаются через символьный буфер Buffer длинной DATA_BUFSIZE. В поле bp находится текущий указатель записи и чтения для буфера Buffer. Socket определяет дескриптор сокета. Поле leftToRead содержит информацию о том, сколько байт осталось передать клиенту, а поле leftToWrite – сколько осталось принять. Next содержит указатель на следующий элемент списка.

Далее в листинге приведены прототипы наших функций. Прототипы WSAStartup, Socket, _WSAAsyncSelect, Bind, Accept и Listen соответствуют функциям WinSock WSAStartup, socket, WSAAsyncSelect, bind, accept и listen. Далее идут прототипы трёх функций для работы со списком SOCKET_INFORMATION. Функция CreateSocketInformation помещает список информацию для сокета. Получить информацию о сокете можно функцией GetSocketInformation(SOCKET s); Удалить информацию о сокете позволит функция FreeSocketInformation. Функция MakeWorkerWindow служит для создания рабочего окна с прцедурой окна WindowProc. Функции forFD_READ и forFD_WRITE являются обработчиками сообщений Windows о сетевых событиях FD_READ и FD_WRITE о готовности сокета к приёму и передаче. Myprintf это приведенная в первой работе функция форматированного вывода кирилицы в консольное окно.

Далее следует определение глобального списка SocketInfoList, в котором хранится информация о соединённых сокетах.

Логика работы функции main проста. Вначале с помощью функции MakeWorkerWindow создаётся рабочее окно W. Функция _WSAStartup запускает WinSock. Создаётся поточный TCP сокет ListenS ориентированный на соединение. Вызов _WSAAsyncSelect(Listen, Window, WM_SOCKET, FD_ACCEPT)

предписывает WinSock отсылать сообщения окну Window в случае, когда клиент выдаст запрос на подключение.

Далее связываем сокет с портом 5150 и переводим в режим прослушивания. После чего входим в главный цикл по обработки сообщений. Функция GetMessage извлекает сообщение msg из очереди сообщений к этому приложению, а функция DispatchMessage направляет его в сторону окна W, что приведёт к вызову функции окна WindowProc.

Функция MakeWorkerWindow содержит стандартную последовательность по созданию окна Window (см. Листинг 2)

```
HWND MakeWorkerWindow(void){
  WNDCLASS wndclass;
  CHAR *ProviderClass = "AsyncSelect";
  HWND Window;

  wndclass.style = CS_HREDRAW | CS_VREDRAW;
  wndclass.lpfnWndProc = (WNDPROC)WindowProc;
  wndclass.cbClsExtra = 0;
  wndclass.cbWndExtra = 0;
  wndclass.hInstance = NULL;
  wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
  wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
  wndclass.lpszMenuName = NULL;
  wndclass.lpszClassName = ProviderClass;

  if (RegisterClass(&wndclass) == 0){
    printf("RegisterClass() failed with error %d\n", GetLastError());
    return NULL;
  }
  if ((Window = CreateWindow(ProviderClass,"", WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,CW_USEDEFAULT, CW_USEDEFAULT,
    NULL, NULL,     NULL,     NULL)) == NULL) {
    printf("CreateWindow() failed with error %d\n", GetLastError());
    return NULL;
  }
  return Window;
}
```

Листинг 2. Функция MakeWorkerWindow

Из листинга 2 следует, что созданное окно в качестве процедуры окна будет иметь функцию WindowProc. Её текст приведён в листинге 3. Мы видим, что функция будет обрабатывать только сообщения WM_SOCKET и только те, которые относятся к сетевым событиям FD_ACCEPT, FD_READ и FD_WRITE. В начале с помощью макроса WSAGETSELECTERROR проверяется наличие ошибок. В случае таковых печатается код ошибки с адресом и портом клиента и информация о сбойном клиенте удаляется из списка с помощью функции FreeSocketInformation. Напомним, что дескриптор сокета приходит в функцию окна через параметр wParam. Если ошибок нет, то макрос WSAGETSELECTEVENT выделяет сетевое событие и далее с помощью оператора switch-case происходит ветвление логики работы программы. Если сокет готов, чтобы в него писали, то будет вызвана процедура forFD_READ, в если он готов к записи, то процедура forFD_WRITE. Если имеется запрос на соединение, то программа пойдёт по ветке case FD_ACCEPT. Здесь она функцией Accept примет запрос на соединение, с помощью функции CreateSocketInformation создаст и поместит в список SocketInfoList необходимую информацию для нового сокета AcceptS и напечатает адрес и порт нового клиента. Вызов _WSAAsyncSelect приведёт к тому, что все новые сокеты AcceptS будут иметь набор сетевых событий FD_READ|FD_WRITE отличный от набора для прослушивающего сокета.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam){
SOCKET AcceptS;
struct sockaddr_in  ca;
int caSize;
if (uMsg == WM_SOCKET){
```

```
    if (WSAGETSELECTERROR(lParam)){
                caSize = sizeof(ca);
                getpeername((SOCKET) wParam, (struct sockaddr *)&ca, &caSize);
                myprintf("WindowProc: Клиент %s:%d вызвал ошибку %d и будет отключен.\n",
                        net_ntoa(ca.sin_addr), ntohs(ca.sin_port),WSAGETSELECTERROR(lParam));
                FreeSocketInformation(wParam);
    }
    else{
          switch(WSAGETSELECTEVENT(lParam)){
                case FD_ACCEPT:
                        int caSize = sizeof(ca);
                         AcceptS = Accept((SOCKET) wParam, (struct sockaddr *)&ca, &caSize);
                         CreateSocketInformation(AcceptS);
                        myprintf("Число клиентов %d.\n", ++s);
                        myprintf("Новый клиент %s:%d \n", inet_ntoa(ca.sin_addr), ntohs(ca.sin_port));
                        _WSAAsyncSelect(AcceptS, hwnd,WM_SOCKET,FD_READ|FD_WRITE);
                        break;
                case FD_READ:
                        forFD_READ(wParam, hwnd);
                         break;
                case FD_WRITE:
                        forFD_WRITE(wParam, hwnd);
                        break;
            }
          }
        }
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
      }
```

Листинг 3. Функция WindowProc

Перейдём к функциям по работе со списком SocketInfoList. Функция CreateSocketInformation создаёт и помещает в список информацию о сокете. Её текст приведен в листинге 4. В начале с помощью функции GlobalAlloc резервируем память для нового элемента списка. Проверка на ошибку очень важна, так как элементы списка содержат буфер Buffer размером DATA_BUFSIZE, который может быть очень большим. После выделения памяти заполняются поля структуры для нового элемента. Изначально устанавливается, что серверу из сокета s осталось читать весь буфер (SI->leftToRead = DATA_BUFSIZE), а писать ему в s не надо SI->leftToWrite = 0. Указатель записи-чтения bp устанавливается в начало буфера Buffer. Текущая голова списка назначается полю Next, а новый элемент становится головой списка.

```
void CreateSocketInformation(SOCKET s){
  LPSOCKET_INFORMATION SI;
  if ((SI = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
    sizeof(SOCKET_INFORMATION))) == NULL)   {
    printf("GlobalAlloc() failed with error %d\n", GetLastError());
    return;
  }
  SI->Socket = s;
  SI->leftToRead = DATA_BUFSIZE;
  SI->leftToWrite = 0;
  SI->bp=SI->Buffer;
  SI->Next = SocketInfoList;
  SocketInfoList = SI;
}
```

Листинг 4. Функция CreateSocketInformation

Функция GetSocketInformation возвращает информацию о сокете. Её код прост и приведен в листинге 5.

```
LPSOCKET_INFORMATION GetSocketInformation(SOCKET s){
  SOCKET_INFORMATION *SI = SocketInfoList;
   while(SI){
        if (SI->Socket == s)
       return SI;
```

```
        SI = SI->Next;
      }
      return NULL;
    }
```

Листинг 5. Функция GetSocketInformation

Функция FreeSocketInformation закрывает сокет и уничтожает о нём информацию в списке, освобождая память. Её код приведен в листинге 6. Элемент PrevSI хранит предыдущий элемент списка. Оператор if-else призван корректно удалить элемент из списка для случая, когда это будет первый элемент.

```
void FreeSocketInformation(SOCKET s){
SOCKET_INFORMATION *SI = SocketInfoList;
SOCKET_INFORMATION *PrevSI = NULL;

while(SI)   {
    if (SI->Socket == s) {
        if (PrevSI)
            PrevSI->Next = SI->Next;
        else
            SocketInfoList = SI->Next;
        closesocket(SI->Socket);
        GlobalFree(SI);
        return;
    }
     PrevSI = SI;
     SI = SI->Next;
}
}
```

Листинг 6. Функция FreeSocketInformation

Как только из сокета можно читать, то процедура окна вызывает функцию forFD_READ, листинг которой приведен в листинге 7. Дескриптор сокета, из которого можно читать функция получает в параметре wParam. Для этого сокета с помощью функции GetSocketInformation запрашивается информация и помещается в переменную SocketInfo. Если сокета нет в списке выводится сообщение об ошибке и функция завершает работу. Далее функция выполняется, если только не весь пакет выбран из сокета (SocketInfo->leftToRead>0). Мы для чтения используем функцию WSARecv, которая будет работать без блокировки. Для её вызова надо подготовить буфер SocketInfo->DataBuf, куда она будет читать. Хоть из сокета можно читать, мы всё же проверим результат чтения на ошибку. Напомним, что ошибка WSAEWOULDBLOCK для неблокирующих сокетов таковой не считается и свидетельствует лишь о неготовности данных. Число прочитанных байт функция WSARecv возвратит в переменной RecvBytes. Уменьшим число байт, которые осталось прочитать в текущем принимаемом пакете и передвинем указатель записи-чтения. Если выбран весь пакет (SocketInfo->leftToRead==0), то мы готовимся к его отправке, указав, что нам надо передать весь пакет (SocketInfo->leftToWrite=DATA_BUFSIZE;) и установив указатель записи-чтения на начало буфера. В заключении следует повторно переустановить реакцию текущего сокета на сетевые события FD_READ|FD_WRITE с помощью функции WSAAsyncSelect.

```
    void forFD_READ(WPARAM wParam, HWND hwnd){
    LPSOCKET_INFORMATION SocketInfo;
    DWORD RecvBytes;
    DWORD Flags;
    SOCKET s = (SOCKET)wParam;
    if(!(SocketInfo = GetSocketInformation(s))){
        struct sockaddr_in ca;
        int caSize = sizeof(ca);
        getpeername(s, (struct sockaddr *)&ca, &caSize);
        myprintf("forFD_READ: Клиент (%s:%d) отпал с ошибкой %s.\n", inet_ntoa(ca.sin_addr),
                        ntohs(ca.sin_port), encodeWSAGetLastError(WSAGetLastError()));
        return;
    }
     if(SocketInfo->leftToRead>0){
        SocketInfo->DataBuf.buf = SocketInfo->bp;
        SocketInfo->DataBuf.len = SocketInfo->leftToRead;
```

```
        Flags = 0;
        if (WSARecv(s, &(SocketInfo->DataBuf), 1, &RecvBytes,
                        &Flags, NULL, NULL) == SOCKET_ERROR){
                if (WSAGetLastError() != WSAEWOULDBLOCK){
                        printf("forFD_READ: WSARecv () failed with error %d\n", WSAGetLastError());
                        FreeSocketInformation(wParam);
                        return;
                }
        }
        else {
                SocketInfo->bp += RecvBytes;
                SocketInfo->leftToRead -= RecvBytes;
                if(SocketInfo->leftToRead==0){
                        //Можно обрабатывать полученную информацию
                        //Мы делаем просто Эхо
                        SocketInfo->leftToWrite=DATA_BUFSIZE;
                        SocketInfo->bp = SocketInfo->Buffer;
                        _WSAAsyncSelect(s, hwnd, WM_SOCKET, FD_READ|FD_WRITE);
                }

    }
  }
 }
```

Листинг 7. Функция forFD_READ

Как только в сокет можно писать, то процедура окна вызывает функцию forFD_ WRITE, листинг которой приведен в листинге 8. Дескриптор сокета, из которого можно читать функция получает в параметре wParam. Для этого сокета с помощью функции GetSocketInformation запрашивается информация и помещается в переменную SocketInfo. Если сокета нет в списке, выводится сообщение об ошибке и функция завершает работу. Далее функция выполняется, если только не весь пакет передан клиенту (SocketInfo->leftToWrite>0). Мы для записи используем функцию WSASend, которая будет работать без блокировки. Для её вызова надо подготовить буфер SocketInfo->DataBuf, откуда она будет писать в сокет. Хоть в сокет можно писать, мы всё же проверим результат записи на ошибку. Напомним, что ошибка WSAEWOULDBLOCK для неблокирующих сокетов таковой не считается и свидетельствует лишь о неготовности данных. Число записанных байт функция WSASend возвратит в переменной SendBytes. Уменьшим число байт, которые осталось записать из текущего передавемого пакета и передвинем указатель записи-чтения. Если передан весь пакет (SocketInfo->leftToWrite==0), то мы готовимся к его отправке, указав, что нам надо прочитать весь пакет (SocketInfo->leftToRead=DATA_BUFSIZE;) и установив указатель записи-чтения на начало буфера. В заключении следует с помощью функции WSAAsyncSelect повторно переустановить реакцию текущего сокета на сетевые события FD_READ|FD_WRITE.

```
void forFD_WRITE(WPARAM wParam, HWND hwnd){
        LPSOCKET_INFORMATION SocketInfo;
        DWORD SendBytes;
        DWORD Flags;
        SOCKET s = (SOCKET)wParam;
        if(!(SocketInfo = GetSocketInformation(s))){
                struct sockaddr_in ca;
                int caSize = sizeof(ca);
                getpeername(s, (struct sockaddr *)&ca, &caSize);
                myprintf("forFD_ WRITE Клиент (%s:%d) отпал с ошибкой %s.\n", inet_ntoa(ca.sin_addr),
                        ntohs(ca.sin_port), encodeWSAGetLastError(WSAGetLastError()));
                return;
        }
        if(SocketInfo->leftToWrite>0){
                SocketInfo->DataBuf.buf = SocketInfo->bp;
                SocketInfo->DataBuf.len = SocketInfo->leftToWrite;
                Flags = 0;
                if (WSASend(s, &(SocketInfo->DataBuf), 1, &SendBytes, 0,
                                                NULL, NULL) == SOCKET_ERROR){
                        if (WSAGetLastError() != WSAEWOULDBLOCK){
```

```
                        printf("forFD_WRITE: WSASend() failed with error %d\n",
                        WSAGetLastError());
                        FreeSocketInformation(wParam);
                        return;
                }
        }
        else{
                SocketInfo->bp += SendBytes;
                SocketInfo->leftToWrite-= SendBytes;
                if(SocketInfo->leftToWrite==0){
                        SocketInfo->leftToRead=DATA_BUFSIZE;
                        SocketInfo->bp = SocketInfo->Buffer;
                        _WSAAsyncSelect(s, hwnd, WM_SOCKET,FD_READ|FD_WRITE);
                }
        }
    }
}
```

Листинг 8. Функция forFD_ WRITE

Клиент

Перейдём к клиенту. Его главная функция, глобальные переменные и прототипы функций представлены в листинге 9. Флаг needNewRnd определяет потребность в генерации новых данных для пакета, отправляемого серверу. Адреса msgForR и msgForW указывают на буфер приёма и передачи. Адрес bp являетсяч текущим указателем записи чтения. leftToRead содержит число байт, которое надо прочитать от сервера в msgForR, a leftToWrite содержит число байт, которое надо передать из msgForW серверу. Поскольку клиент вначале передаёт, а затем принимает, то устанавливае leftToRead=0 и leftToWrite=DATA_BUFSIZE.

В функции main осуществляется соединение к серверу по адресу указанному в командной строке. Далее выделяется память для передачи и приёма и запускается нить с функцией ClientThread, листинг которой приведен в листинге 10. Чтобы программа не закончила работать раньше времени, присутствует вызов макроса getchar, который ожидает ввода с клавиатуры.

```
#include <winsock2.h>
#include <stdio.h>
#include "../DATA_BUFSIZE.h"
#define WM_SOCKET (WM_USER + 1)
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int _WSAAsyncSelect( SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Connect(SOCKET s, struct sockaddr* ServerAddr, int );
HANDLE WINAPI _CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,     LPTHREAD_START_ROUTINE lpStartAddress,
         LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
DWORD WINAPI ClientThread(LPVOID lpParam);
HWND MakeWorkerWindow(void);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
void forFD_WRITE(WPARAM wParam, HWND hwnd);
void forFD_READ(WPARAM wParam, HWND hwnd);
int needNewRnd=1;
int *msgForR, *msgForW;
char *bp;
volatile DWORD leftToRead=0;
volatile DWORD leftToWrite =DATA_BUFSIZE;

void main(int argc, char **argv){
WSADATA             wsaData;
SOCKET              s;
struct sockaddr_in          ServerAddr, ca;
int                 Port = 5150;
```

```
int                Ret;
int caSize = sizeof(ca);
HANDLE        hThread;
DWORD         dwThreadId;

if (argc <= 1)   {
     myprintf("Использование: tcpclient <Server IP address>.\n");
     return;
}
WSAStartup(MAKEWORD(2,2), &wsaData);
 s = Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(Port);
ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
Connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr));
getsockname(s, (struct sockaddr *)&ca, &caSize);
myprintf("Локальный адрес:%s и порт:%d сокета клиента\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
msgForW = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
msgForR = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
hThread = _CreateThread(NULL, 0, ClientThread,(LPVOID)s, 0, &dwThreadId);
CloseHandle(hThread);
myprintf("Обмениваемся данными с Эхо сервером .\n");
myprintf("Нажмите Enter  для выхода.\n");
getchar();
closesocket(Socket);
WSACleanup();
}
```

Листинг 9. Главная функция клиента WSAAsyncSelec.

В функции ClientThread мы с помощью вызова функции MakeWorkerWindow (листинг 2) создаём рабочее окно Window и с помощью функции WSAAsyncSelect предписываем Winsock направлять в функцию WindowProc главного окна сообщения о сетевых событиях о возможностях записи FD_WRITE и чтения FD_READ на сокете s. Код WindowProc приведен в листинге 11.

```
DWORD WINAPI ClientThread(LPVOID lpParam){
  MSG msg;
  DWORD Ret;
  HWND Window;
  SOCKET   s     =(SOCKET)lpParam;
  if ((Window = MakeWorkerWindow()) == NULL)      return 0;
  _WSAAsyncSelect(s, Window, WM_SOCKET,  FD_READ|FD_WRITE);
  while(Ret = GetMessage(&msg, NULL, 0, 0)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

  return 0;
}
```

Листинг 10. Код функции ClientThread.

Функция WindowProc проста и предназгачена лишь для пеагирования на сетевые события FD_WRITE и FD_READ путём вызова функций forFD_WRITE и forFD_READ. Коды этих функций приведены в листингах 12 и 13.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam){
 if (uMsg == WM_SOCKET)   {
     if (WSAGETSELECTERROR(lParam))      {
       myprintf("Socket failed with error %d\n", WSAGETSELECTERROR(lParam));
       closesocket(wParam);
     }
     else      {
       switch(WSAGETSELECTEVENT(lParam)){
         case FD_WRITE:
```

```
                                forFD_WRITE(wParam,  hwnd);
                                break;
        case FD_READ:
                                forFD_READ(wParam,  hwnd);
                                break;
      }
    }
    return 0;
  }
  return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

Листинг 11. Код функции WindowProc клиента.

В функции forFD_WRITE в листинге 12 переменные DataBuf, SendBytes и Flags используются при вызове функции записи в сеть WSASend. Число записанных байт функция WSASend возвращает в переменной SendBytes. Вначале, если установлен глобальный флаг needNewRnd, производится генерация данных для заполнения нового пакета на отправку и текущий указатель записи чтения bp устанавливается на начало сгенерированных данных. Остальной код выполняется, если только остались байты на передачу серверу (leftToWrite>0). Если это так, то вызывается функция передачи данных серверу WSASend. Производится диагностика ошибок. Передвигается вперёд на число переданных байт SendBytes указатель bp и на это же число уменьшается число байт leftToWrite, осташихся на передачу. Если это число равно нулю, то пакет передан и надо готовиться к приёму ответа от сервера. Для это указываем, что от сервера надо принять leftToRead=DATA_BUFSIZE байт, устанавливаем глобальный указатель bp на начало буфера приёма msgForR и с помощью функции WSAAsyncSelect предписываем Winsock направлять в функцию WindowProc главного окна hwnd сообщения о сетевых событиях о возможностях записи FD_WRITE и чтения FD_READ на сокете wParam.

```
void forFD_WRITE(WPARAM wParam, HWND hwnd){
        DWORD Flags, SendBytes;
        WSABUF DataBuf;
        if(needNewRnd){
                int i;
                for(i=0;i<DATA_BUFSIZE_INT;i++) msgForW[i]=rand();
                bp=(char*)msgForW;
                needNewRnd=0;
        }
        if(leftToWrite>0){
                DataBuf.buf = bp;
                DataBuf.len = leftToWrite;
                Flags = 0;
                if (WSASend((SOCKET)wParam, &DataBuf, 1, &SendBytes, 0,        NULL,
                                                                NULL) == SOCKET_ERROR){
                        if (WSAGetLastError() != WSAEWOULDBLOCK){
                                printf("forFD_ WRITE: WSASend failed with error %d\n",
                                        WSAGetLastError());
                                closesocket(wParam);
                                return;
                        }
                }
                else{
                        bp += SendBytes;
                        leftToWrite -= SendBytes;
                        if(leftToWrite==0){
                                leftToRead=DATA_BUFSIZE;
                                bp = (char*)msgForR;
                                //WSAAsyncSelect((SOCKET)wParam, hwnd, WM_SOCKET,
                                                        //FD_READ|FD_WRITE);
                        }
                }
        }
}
```

Листинг 12. Код функции forFD_WRITE клиента

В функции forFD_ READ в листинге 13 переменные DataBuf, RecvBytes и Flags используются при вызове функции чтения WSARecv. Число прочитанных байт функция WSARecv возвращает в переменной RecvBytes. Код функции выполняется, если только остались байты для приёма пакета от сервера (leftToRead>0). Если это так, то вызывается функция приёма данных от сервера WSARecv. Производится диагностика ошибок. Передвигается вперёд на число принятых байт RecvBytes указатель bp и на это же число уменьшается число байт leftToRead, осташихся для приёма. Если это число равно нулю, то пакет принят и мы с помощью функции memcmp сравниваем переданный пакет с принятым и в случае несовпадения подаём звуковой сигнал об ошибке частотой в 333 герца длительностью 3.333 секунды. Далее надо готовиться к отправке серверу нового пакета. Для это указываем, что к серверу надо передать leftToWrite=DATA_BUFSIZE байт, устанавливаем глобальный указатель bp на начало буфера передачи msgForW, устанавлтваем флаг needNewRnd о необходимости генерации новых данных и с помощью функции WSAAsyncSelect предписываем Winsock направлять в функцию WindowProc главного окна hwnd сообщения о сетевых событиях о возможностях записи FD_WRITE и чтения FD_READ на сокете wParam.

```
void forFD_READ(WPARAM wParam, HWND hwnd){
        DWORD RecvBytes, Flags = 0;
        WSABUF DataBuf;
        if(leftToRead>0){
                DWORD Flags = 0;
                DataBuf.buf = bp;
                DataBuf.len = leftToRead;
                if (WSARecv((SOCKET)wParam, &DataBuf, 1, &RecvBytes,
                                        &Flags, NULL, NULL) == SOCKET_ERROR){
                        if (WSAGetLastError() != WSAEWOULDBLOCK){
                                printf("forFD_READ: WSARecv() failed with error %d\n",
                                        WSAGetLastError());
                                closesocket(wParam);
                                return;
                        }
                }
                else {
                        bp += RecvBytes;
                        leftToRead -= RecvBytes;
                        if(leftToRead==0){
                                if(memcmp(msgForR,msgForW,DATA_BUFSIZE)){
                                        puts("ERROR\n");
                                        Beep(333,3333);
                                        //exit(77);
                                }
                                leftToWrite=DATA_BUFSIZE;
                                bp=(char*)msgForW;
                                needNewRnd=1;
                                WSAAsyncSelect((SOCKET)wParam, hwnd, WM_SOCKET,
                                        FD_READ|FD_WRITE);
                }
        }
}
```

Листинг 13. Код функции forFD_READ клиента

И для клиента и для сервера не предусмотрен механизма завершения работы. Мы вынуждены будем завершать программы либо путём закрытия консольного окна, либо нажатием комбинации клавиш Ctrl+C. Это приведёт к тому, что все вычислительные ресурсы, которые взяла себе программа (сокеты и буфера) не будут освобождены и обявлены как свободные для их дальнейшего использования операционной системой. Произойдёт так называемая утечка памяти. Если она значительная, то операционная система станет медленно работать и единственным средством восстановления её нормальной работы является перегрузка. Помните об этом.

Контрольные вопросы.

1. Назовите и охарактеризуйте режимы работы сокетов.
2. Назовите модели ввода-вывода сокетов.
3. Какие недостатки и преимущества работы с сокетми с блокировкой?
4. Какие недостатки и преимущества работы с сокетми без блокировки?

5. Каково главное назначение моделей сокетов и что модели позволяют определить?
6. Объясните одной фразой суть модели WSAAsyncSelect.
7. Что нужно предварительно сделать в программе, чтобы запустить в ней модель ввода-вывода WSAAsyncSelect?
8. Объясните назначение параметров функции WSAAsyncSelect.
9. Перечислите и охарактеризуйте сетевые события.
10. Когда возникает сетевое событие FD_READ и к чему это приведёт в модели WSAAsyncSelect?
11. Когда возникает сетевое событие FD_WRITE и к чему это приведёт в модели WSAAsyncSelect?
12. Какая задача на разработку ставится в практической части?
13. Зачем использовать функции обвёртки?
14. Объясните логику работы сервера из практической части.
15. Объясните используемые в сервере глобальные переменные и типы данных.
16. Каково назначение функций MakeWorkerWindow и WindowProc?
17. Что содержится в элементах списка SocketInfoList?
18. Объясните работу трёх функций по работе со списком SocketInfoList.
19. Объясните работу функции forFD_READ сервера.
20. Объясните работу функции forFD_WRITE сервера.
21. Обясните логику работы клиента из практической части.
22. Объясните используемые в сервере глобальные переменные
23. Объясните работу функции forFD_READ клиента.
24. Объясните работу функции forFD_WRITEклиента.
25. Какой главный недостаток предложенного клиента и сервера.

Задания

1. Загрузите в Visual Studio 2008 проект сервера AsyncSelect (листинги 1 – 8) и клиента AsyncSelectclient (листинги 9 – 13) из решения lab4 архива src. Произведите отладку проектов. Рекомендация: и клиент и сервер должен иметь одинаковый размер буферов DATA_BUFSIZE, поэтому сделайте для их обоих общий заголовочный файл DATA_BUFSIZE.h
2. Сравните текущую и среднюю скорость обмена информацией и объясните результат для соединений:
   a. Клиента AsyncSelectclient из этой работы к серверу AsyncSelect из этой работы.
   b. Клиента AsyncSelectclient из этой работы к серверу tcpTreadServerF из работы 2.
   c. Клиента AsyncSelectclient из этой работы к серверу tcpSelectServerF из работы 2.
   d. Клиента ClientF из работы 2 к серверу AsyncSelect из этой работы.
3. Модифицируйте программы из этой работы для пакетов переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом.

Лабораторная работа №5. Корректное завершение работы сетевых приложений

Теоретическая часть

Как стало уже понятно, при общении по сети через соединённые TCP-сокеты данные могут быть записаны в сокет , но не приняты на сокете с противоположной стороны. То есть данные могут находиться в сети. Если сетевое приложение не будет учитывать этот факт, то при его завершении данные могут быть потеряны. Если приложение намерено корректно и правильно прекратить свою работу, то оно должно отослать все необходимые данные в сеть, известить удалённую сторону о своём закрытии, принять все данные из сети, дождаться от удалённой стороны сигнала о том, что она поняла извещение о закрытии и только тогда закрываться.

Важно различать отключение сокета от его закрытия. Отключение сокета приводит к обмену сообщениями между двумя оконечными узлами, который называется последовательностью отключения. Определены два класса последовательностей: корректная (graceful) и обрывающая или тяжелая. В первой все данные в очереди для отправки могут быть переданы до закрытия соединения. Во второй – неотосланные данные потеряются. О возникновение и корректной и обрывающей последовательности отключения удалённая сторона может узнать, если она отслеживает сетевое событие FD_CLOSE.

С другой стороны закрытие сокета приводит к закрытию дескриптора и освобождению соответствкющих системных ресурсов. После чего к сокету уже нельзя будет никак обратиться.

Для инициализации корректной последовательности отключения можно использовать функцию shutdown. Функция closesocket приводит закрытию сокета. Closesocket неявно инициализирует последовательность отключения, если она еще не началась.

В ряде случаев достаточно вызвать только функцию closesocket для корректного отключения и закрытия сокета. Здесь могут быть полезны опции сокета, которые разрешают программисту указывать будет ли неявная последовательность отключения корректной или обрывающей, а также будет ли функция closesocket ждать некоторое время, чтобы позволить последовательности отключения завершиться корректно.

При соответствующих опциях SO_LINGER и SO_DONTLINGER можно добиться таких поведений функции closesocket:

-Немедленный возврат из функции и обрывающая последовательностью отключения.

-Задержка возврата из функции либо до окончания последовательности отключения либо по истечению заданного интервала времени. Если по истечению этого интервала времени корректное отключение не произойдёт, то будет некорректное отключение.

-По умолчанию возврат из closesocket происходит немедленно, а последовательность отключения выполняется в фоновом режиме. К сожалению нет способа узнать как завершилась эта последовательность.

Рекомендуется не полагаться на неявную последовательность отключения функции closesocket, а инициализировать эту последовательность явным образом функцией shutdown.

Функция shutdown запрещает на сокете передачу или приём.

int shutdown(SOCKET s, int how);

Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

Сокет задаётся параметром s. Параметр определяет какие операции над сокетом будут в дальнейшем невозможны. В случае успеха функция возвращает ноль, иначе - SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRESS, WSAENOTCONN, WSAENOTSOCK. Расшифровку ошибок смотри в приложении или в MSDN.

Если параметр how равен SD_RECEIVE, то дальнейший приём данных на сокете запрещается. Для TCP сокетов даные принять будет невозможно, даже если они уже отправлены, имеются в очереди к сокету или там появятся.

Если параметр how равен SD_SEND, то дальнейшая передача данных на сокете запрещается. Для TCP сокетов после отсылки всех оставшихся внутри сокета данных и подтверждения их приёма получателем в сторону получателя отправляется пакет FIN.

Если параметр how равен SD_BOTH, то дальнейший приём и передача данных на сокете запрещается. Сокет не закрывается и ресурсы не освобождаются.

Для уверенности, что все данные на соединённом TCP сокете отправлены и получены до его закрытия, следует перед вызовом closesocket использовать shutdown. Здесь полезен вызов функции WSAAsyncSelect или WSAEventSelect для установки реакции приложения на сетевое событие FD_CLOSE, возникающее при начале на противоположной стороне последовательности отключения сокета.

В таблице 1 приведена рекомендуемая процедура корректного отключения сокетов и со стороны клиента и со стороны сервера.

Таблица 1. Процедура корректного отключения сокетов

Клиент	**Сервер**
(1) Закрыть сокет s по записи:	

shutdown(s,SD_SEND), сигнализируя серверу, что нет больше данных на отправку	
	(2) Отреагировать на событие FD_CLOSE, о том, что клиент вызвал функцию shutdown(s,SD_SEND), началась корректная последовательность отключения и что все данные приняты.
	(3) Отослать клиенту необходимые данные от сервера, если таковые имеются
	(4) Закрыть сокет s по записи: shutdown(s,SD_SEND), сигнализируя клиенту, что нет больше данных на отправку.
(4) Отреагировать на события FD_READ и принять все данные, отосланные сервером, если таковые имеются	
(5) Отреагировать на событие FD_CLOSE, о том, что сервер вызвал функцию shutdown(s,SD_SEND)	
(6)Выполняем closesocket	(6)Выполняем closesocket

.

Воплотим процедуру корректного отключения сокетов из таблицы 1 в виде программы. Используем код процедуры MakeWorkerWindow, приведенный в листинге 2 работы 4. По-прежнему используем обвёртки функций Winsock. Код клиента приведен в листинге 1. Код сервера приведен в листинге 2.

```
#include <winsock2.h>
#include <stdio.h>
#define WM_SOCKET (WM_USER + 1)
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int _WSAAsyncSelect( SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
int Connect(SOCKET s, struct sockaddr* ServerAddr, int );

int myprintf (const char *lpFormat, ... );
HWND MakeWorkerWindow(void);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
void main(int argc, char **argv){
WSADATA             wsaData;
SOCKET              s;
struct sockaddr_in          ServerAddr, ca;
int                 Port = 5150;
int                 Ret;
int caSize = sizeof(ca);
MSG msg;
HWND hwnd;
if (argc <= 1)   {
     myprintf("Использование: tcpclient <Server IP address>.\n");
     return;
}
hwnd=MakeWorkerWindow();
_ WSAStartup(MAKEWORD(2,2), &wsaData);
s=Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(Port);
ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
Connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr))
_WSAAsyncSelect(s, hwnd, WM_SOCKET,  FD_READ|FD_CLOSE);
shutdown(s,SD_SEND);
//getchar();// Чтобы посмотреть состояние TCP
myprintf("(1) Закрыли сокет s по записи: shutdown(s,SD_SEND), сигнализируя серверу, "
                "что нет больше данных на отправку.\n");
while(GetMessage(&msg, NULL, 0, 0)){
      TranslateMessage(&msg);
      DispatchMessage(&msg);
```

```
}
WSACleanup();
myprintf("Нажмите Enter для выхода.\n");
getchar();
}
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam){
int RecvBytes;

char Buf[256];
if (uMsg == WM_SOCKET)   {
     if (WSAGETSELECTERROR(lParam))      {
               myprintf("Socket failed with error %d\n", WSAGETSELECTERROR(lParam));
               closesocket(wParam);
     }
     else     {
               switch(WSAGETSELECTEVENT(lParam)){
                         case FD_READ:
                                   RecvBytes=recv(wParam, Buf, sizeof(Buf),0);
                                   Buf[RecvBytes]='\0';
                                   myprintf("(4)Реакция на событие FD_READ. Приняты все данные,"
                                                       "отосланные сервером:%s\n", Buf);
                                   break;
                         case FD_CLOSE:
                                   myprintf("(5)Реакция на событие FD_CLOSE, о том, что сервер вызвал"
                                             "функцию  shutdown(s,SD_SEND).\n");
                                   closesocket(wParam);
                                   myprintf("(6)Выполняем closesocket.\n");
                                   PostQuitMessage(0);
                                   break;
               }
     }
}
return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

Листинг 1. Клиент с коректной последовательностью отключения.

```
#include <winsock2.h>
#include <stdio.h>
#define PORT 5150
#define WM_SOCKET (WM_USER + 1)
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int _WSAAsyncSelect( SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Listen(SOCKET s,int backlog);
SOCKET Accept(SOCKET s, struct sockaddr* addr,int* addrlen);

HWND MakeWorkerWindow(void);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
int myprintf (const char *lpFormat, ... );

void main(void){
MSG msg;
DWORD Ret;
SOCKET AcceptS, Listen;
HWND hwnd;
struct sockaddr_in InternetAddr, ca;
WSADATA wsaData;
int caSize=sizeof(ca);

hwnd = MakeWorkerWindow();
_ WSAStartup(0x0202, &wsaData)'
listen = Socket (PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(PORT);
Bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr));
Listen(Listen, SOMAXCONN);
AcceptS = Accept(Listen,  (struct sockaddr *)&ca, &caSize);
_WSAAsyncSelect(AcceptS, hwnd, WM_SOCKET, FD_CLOSE);
while(Ret = GetMessage(&msg, NULL, 0, 0))   {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
closesocket(Listen);
WSACleanup();
myprintf("Нажмите Enter для выхода.\n");
getchar();
}


LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam){
if (uMsg == WM_SOCKET){
    if (WSAGETSELECTERROR(lParam)){
            myprintf("WindowProc Клиент отпал с ошибкой %d.\n", WSAGETSELECTERROR(lParam));
            closesocket(wParam);
    }
    else if (WSAGETSELECTEVENT(lParam)==FD_CLOSE){
            char Buf[256]="Необходимые данные от сервера.";
            myprintf("(2) Реакция на событие FD_CLOSE о том, что клиент вызвал функцию”
                    “shutdown(s,SD_SEND), началась корректная последовательностью отключения”
                    “ и что все данные приняты.\n");
            Send(wParam, Buf, strlen(Buf), 0);
            myprintf("(3)Отослали клиенту %s\n", Buf);
            //getchar();// Чтобы посмотреть состояние TCP
            shutdown(wParam, SD_SEND);
            myprintf("(4) Закрыли сокет s по записи: shutdown(s,SD_SEND), сигнализируя клиенту, "
                    "что нет больше данных на отправку.\n");
            //getchar();// Чтобы посмотреть состояние TCP
            closesocket(wParam);
            myprintf("(6)Выполняем closesocket.\n");
            PostQuitMessage(0);
    }
}
return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

Лисинг 2. Сервер с корректной последовательностью отключения.

Скриншоты результатов работы клиента и сервера приведены на рисунках 1 и 2.

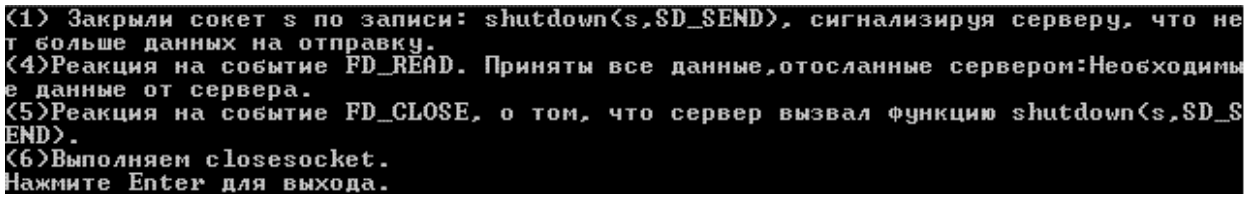

Рисунок 1. Результат работы клиента с корректной последовательностью отключения.

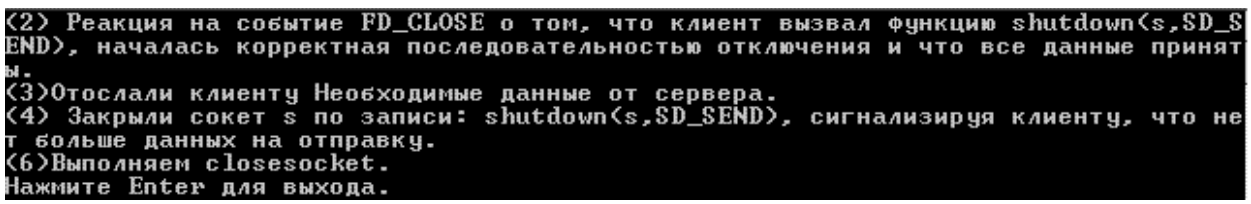

Рисунок 2. Результат работы сервера с корректной последовательностью отключения.

Можно посмотреть состояния сокетов с помощью утилиты netstat –an или tcpview после выполнения , функции shutdown у клиента, а затем на сервере. Для этого либо запустите два экземпляра Visual Studio в режиме отладки, либо раскомментируйте в программах getchar. Если пойти по второму пути, то клиент остановится сразу после shutdown, а сервер перед shutdown. Состояние клиента будет FIN_WAIT_2, а состояние сервера будет CLOSE_WAIT. Нажмём Enter в окне сервера. Он выполнит shutdown и его сокет мы не увидим с помощью утилит –an или tcpview. Состояние сокета клиента станет TIME_WAIT и будет таковым в течении нескольких минут после закрытия клиента. Более подробно процедура изменений состояний сокетов приведена на рисунке 1 в работе 2.

Синхронизация вычислительных нитей

После запуска приложения стартует вычислительный процесс с одной вычислительной нитью, соответствующей функции main. Как указывалось в работе 2, с помощью функции CreateThread можно создать внутри процесса дополнительные нити. Кодом-носителем новой нити может быть любая особым образом организованная функция в приложении. Каждая нить имеет свой стек для хранения локальных переменных.

Термин нить (thread) часто переводят как поток. Но в контексте сетевых приложений термин поток уже используется: данные между сокетами, как известно, передаются в виде потока или сообщений (см. Работу 1).

Каждая нить внутри вычислительного процесса , выполняется независимо от других нитей этого приложения и параллельно с ними. Каждая нить выполняется на отдельном виртуальном (мнимом) центральном процессоре. Нити начинают взаимодействовать друг с другом, когда они начинают читать или писать в общие переменные. Глобальные переменные приложения являются общими для всех нитей этого приложения. Последовательность и порядок общения нитей с общими переменными предугадать невозможно. Особенно опасна и непредсказуема неконтролируемая запись несколькими нитями разных значений в общие переменные. Это может привести к краху вашего приложения. Если мы хотим задать свой желаемый порядок общения нитей с общими переменными, следует использовать системные примитивы синхронизации. Среди богатого арсенала таких примитивов, который имеется в Windows, воспользуемся лишь критическими секциями и событиями Winsock. Критические секции рассмотрим в работе 7.

Объект события имеет тип WSAEVENT. Объект события создаётся функцией WSACreateEvent(WSAEVENT hEvent). Устанавливают события в сигнальное состояние функцией WSASetEvent(WSAEVENT hEvent), сбрасывают функцией WSAResetEvent(WSAEVENT hEvent) и закрывают - WSARCloseEvent(WSAEVENT hEvent). Нить может ждать наступления событий из массива событий lphEvents с помощью функции

DWORD WSAWaitForMultipleEvents(DWORD cEvents,const WSAEVENT* lphEvents,
BOOL fWaitAll,DWORD dwTimeout,BOOL fAlertable);

Здесь cEvents – число элементов в масиве. Если fWaitAll равно TRUE, то ожидаются все события, иначе (FALSE) – ожидается любое. dwTimeout – таймаут в миллисекундах, WSA_INFINITE для бесконечного ожидания. Если fAlertable равно TRUE, то произойдёт возврат из функции только после выполнения системной процедуры завершения ввода вывода.

Если fWaitAll равно FALSE, то возвращаемое значение минус WSA_WAIT_EVENT_0 равно наименьшему индекс наступившего события из массива lphEvents. При ошибке функция возвращает WSA_WAIT_FAILED и код ошибки можно определить с помощью функции WSAGetLastError.

Одно событие могут ждать сразу несколько нитей. При наступлении события нити продолжат своё выполнение. Важно отметить, что при ожидании нить не занимает времени центрального процессора. Выводит нить из состояния ожидания ни она сама, а система.

Приведём пример использования событий Winsock. Рассмотрим классическую задачу производитель-потребитель в упрощённой форме. Потреблять можно только после производства. Пусть производителем является нить producerThread, а потребителем – consumerThread. Факт производства обозначим установкой общей переменной share из нуля в единицу в коде consumerThread. Факт потребления обозначим выводом на экран значения переменной share. То есть нить consumerThread должна подождать, пока нить consumerThread проделает работу по установке переменной share и только тогда вывести значение share на консоль. Нить producerThread должна известить нить consumerThread о том, что она проделала ожидаемую от неё работу. Такое ожидание и извещение можно реализовать с помощью событий Winsock.

Рассмотрим код в листинге 3. Факту производства сопоставим событие hEvent . Нить consumerThread ожидает события hEvent с помощью функции WSAWaitForMultipleEvents. Нить consumerThread устанавливает событие hEvent после факта производства (назначения share=1);

Нельзя предугадать какая нить начнёт первой выполняться. Нить consumerThread создаётся первой, но нет гарантии, что её код начнёт выполняться раньше, чем код нити producerThread. На однопроцессорной машине иллюзия параллельного выполнения процессов и нитей в них создаётся за счёт выделения им квантов, отрезков процессорного времени. Время переключения процессорного времени на другую нить с точки зрения нитей непредсказуемо. Непредсказуем также оператор функции нити, на котором истечёт квант процессорного времени и начнут выполняться операторы функции другой нити. Например, на

центральном процессоре может выполниться первый оператор функции producerThread, затем первый и второй оператор функции consumerThread, затем второй и третий оператор функции producerThread и т.д. Но при любом сценарии развития функция consumerThread будет ожидать события hEvent в функции WSAWaitForMultipleEvents до тех пор, пока функуия producerThread его не установит. Можно гарантировать, что оператор printf функции consumerThread всегда выполнится после оператора printf функции producerThread:

```
producerThread share=1
consumerThread share=1
```

```
#include <winsock2.h>
DWORD WINAPI consumerThread(LPVOID lpParam);
DWORD WINAPI producerThread(LPVOID lpParam);

int share=0;
WSAEVENT hEvent;
void main(){
    WSADATA            wsaData;
    DWORD       dwThreadId;
    WSAStartup(MAKEWORD(2,2), &wsaData);
    hEvent=WSACreateEvent();
    CreateThread(NULL, 0, consumerThread,(LPVOID)0, 0, &dwThreadId);
    CreateThread(NULL, 0, producerThread,(LPVOID)0, 0, &dwThreadId);
    getchar();
    WSACloseEvent(hEvent);
}
DWORD WINAPI consumerThread(LPVOID lpParam){
    WSAEVENT hEventArray[1];
    hEventArray[0]=hEvent;
    WSAWaitForMultipleEvents(1,hEventArray,TRUE, WSA_INFINITE, FALSE);
    printf("consumerThread share=%d\n", share);
}
DWORD WINAPI producerThread(LPVOID lpParam){
    share=1;
    printf("producerThread share=%d\n", share);
    WSASetEvent(hEvent);
}
```

Листинг 3. Синхронизация нити производителя и нити потребителя с помощью событий Winsock.

Практическая часть

Модифицируем код работы №4, снабдив разработанные в ней клиент и сервер свойствами корректного выхода и закрытия соединений. Мы будем давать комментарии только к новому коду.

Клиент работает в прерывном и непрерывном режиме. В прерывном режиме по команде “c” клиентом отсылается очередной пакет. По команде q клиент закрывает сокет на запись командой shutdown(, SD_SEND) , ждёт подтверждение от сервера и закрывается. В непрерывном режиме клиент непрерывно посылает пакеты серверу. Командой на закрытие клиента будет нажатие клавиши Enter. Клиенту надо выбрать нужное время для закрытия после получения команды на закрытие. Таким моментом будет окончание приёма ответного пакета от сервера. В этот момент он закрывает сокет на запись, ждёт подтверждения от сервера и закрывается.

Мы можем решить прекратить работу сервера, например путём нажатия клавиши Enter. Сервер должен уведомить клиентов о своём закрытии. Это он сделает, помещая в свой ответ клиенту строку “Shutdown” в начало пакета. Клиент должен каждый принятый пакет на предмет наличия в нём строки “Shutdown”. Обнаружив такую строку в начале пакета, он закрывает сокет на запись, ждёт подтверждения от сервера и закрывается.

В задачу сервера входит (помимо подтверждения клиентам его запросов на закрытие) обнаружение клиентов, которые не проявляют активности в течение определённого интервала времени и закрывать с ними соединение.

Клиент также должен обнаруживать, что сервер закрывает соединение с ним. Заметим, что физический обрыв связи не может быть обнаружен средствами протокола TCP/IP. Разве, что по таймауту.

Программирование изложенной логики работы не составило бы труда, если бы не наличие нескольких нитей у клиента и сервера. Каждая нить имеет своё логическое время и нити надо синхронизировать для выполнения поставленной задачи.

Сервер

Рассмотрим вначале модифицированный сервер. Его заголовочный файл приведен в листинге 4. Новое по сравнению с листингом 1 из работы 4 следующее. Константа TIMEOUT определяет, через сколько секунд отключается неактивный клиент. В структуре SOCKET_INFORMATION добавлены 2 поля Shutdown и tick. Первое из них устанавливается для сокета из нуля в единицу, когда сервер передаёт клиентам команду «Shutdown». В поле tick содержит тики операционной системе и служит для поиска неактивных клиентов. Это поле обновляется при каждом обращении к сокету. Функция stopThread является носителем кода для нити, которая инициализирует плавное завершение работы сервера. Функция stopDeadClientsThread является носителем кода для нити, которая отключает клиентов не активных более TIMEOUT секунд. В функции CtrlHandler устанавливается нестандартная реакция приложения на попытку его закрытия. Глобальный флаг Shutdown устанавливается из нуля в единицу, когда сервер начинает процедуру плавного закрытия. И поскольку почти все функции проекта сервера используют дескриптор рабочего окна, то он вынесен в глобальную переменную hwnd. Добавлен обработчик forFD_CLOSE сетевого события FD_CLOSE.

```
#include <winsock2.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "../DATA_BUFSIZE.h"
#define PORT 5150
#define WM_SOCKET (WM_USER + 1)
#define TIMEOUT 5// через сколько секунд неактивный клиент отключается//New!!!

typedef struct _SOCKET_INFORMATION {
        int Shutdown; //New!!!
        WSABUF DataBuf;
        CHAR Buffer[DATA_BUFSIZE];
        char *bp;
        SOCKET Socket;
        DWORD leftToRead;
        DWORD leftToWrite;
        DWORD tick; //New!!!
        struct _SOCKET_INFORMATION *Next;
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;

void CreateSocketInformation(SOCKET s);
LPSOCKET_INFORMATION GetSocketInformation(SOCKET s);
void FreeSocketInformation(SOCKET s);
void forFD_READ(SOCKET socket);
void forFD_WRITE(SOCKET socket);
void forFD_CLOSE(SOCKET socket); //New!!!
HWND MakeWorkerWindow(void);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
DWORD WINAPI stopThread(LPVOID lpParam); //New!!!
DWORD WINAPI stopDeadClientsThread(LPVOID lpParam); //New!!!
BOOL WINAPI CtrlHandler (DWORD dwEvent); //New!!!
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int _WSAAsyncSelect( SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Listen(SOCKET s,int backlog);
SOCKET Accept(SOCKET s, struct sockaddr* addr,int* addrlen);
HANDLE WINAPI _CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,      LPTHREAD_START_ROUTINE lpStartAddress,
         LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);

extern LPSOCKET_INFORMATION SocketInfoList;
extern volatile   int Shutdown;
extern HWND hwnd;
```

Листинг 4. Заголовочный файл GracefulShutdown.h сервера.

Рассмотрим функцию main сервера, приведенную в лисинге 4. Она отличается от кода в листинге 1 работы 4 следующим. Дескриптор рабочего окна помещается в глобальную переменную hwnd. Определён глобальный дескриптор hEvent объекта события WSAEVENT. С помощью функции SetConsoleCtrlHandler для приложения устанавливается реакция на попытку закрытия в виде вызова функции CtrlHandler. С помощью функции WSACreateEvent создаётся объект событие hEvent. В функцию _WSAAsyncSelect добавлено новое сетевое событие о закрытии сокета FD_CLOSE. Создаются две нити с кодами носителями stopThread и stopDeadClientsThread.

```
#include "GracefulShutdown.h"
LPSOCKET_INFORMATION SocketInfoList;
volatile   int Shutdown=0;
HWND hwnd;
WSAEVENT hEvent;
void main(void){
  MSG msg;
  DWORD Ret;
  SOCKET Listen;
  SOCKADDR_IN InternetAddr;
  WSADATA wsaData;
  HANDLE       hThreadS, hThreadD;
  DWORD        dwThreadId;
  if ((hwnd = MakeWorkerWindow()) == NULL)      return;
  _ WSAStartup(0x0202, &wsaData);
  Listen = Socket (PF_INET, SOCK_STREAM, 0);
  _WSAAsyncSelect(Listen, hwnd, WM_SOCKET, FD_ACCEPT|FD_CLOSE); //New!!!
  InternetAddr.sin_family = AF_INET;
  InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  InternetAddr.sin_port = htons(PORT);
  Bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr));
  Listen(Listen, SOMAXCONN);
if( !SetConsoleCtrlHandler(CtrlHandler, TRUE) ) {
            myprintf("SetConsoleCtrlHandler() failed to install console handler: %d\n",
                        GetLastError());
            return;
    }
hEvent= WSACreateEvent();
  hThreadS = _CreateThread(NULL, 0, stopThread,(LPVOID)0, 0, &dwThreadId); //New!!!
  CloseHandle(hThread);
  hThreadD = CreateThread(NULL, 0, stopDeadClientsThread,(LPVOID)0, 0, &dwThreadId); //New!!!
  CloseHandle(hThread);
  while(Ret = GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}

BOOL WINAPI CtrlHandler (DWORD dwEvent) {
switch( dwEvent ) {
      case CTRL_BREAK_EVENT:
      case CTRL_C_EVENT:
      case CTRL_LOGOFF_EVENT:
      case CTRL_SHUTDOWN_EVENT:
      case CTRL_CLOSE_EVENT:
      Shutdown=1;
      WSASetEvent(hEvent);
      return(TRUE);
      default:
      return(FALSE);
      }
}
```

Листинг 5. Код функции main сервера и функции CtrlHandler.

Функция CtrlHandler занимает центральное место в работе, поэтому её текст мы привели вместе с текстом функции main в листинге 5. Консольное приложение можно закрыть разными способами. При этом система генерирует различные сигналы. Если мы закрываем приложение из диспетчера задач или закрываем консольное окно приложения, то генерируется сигнал CTRL_CLOSE_EVENT. Если мы завершаем сессию работы с Windows, то генерируется сигнал CTRL_ LOGOFF _EVENT. Если мы завершаем работу Windows, то то генерируется сигнал CTRL_ SHUTDOWN _EVENT. При нажатии комбинации клавиш Ctrl+C или Ctrl+Break генерируются сигналы CTRL_C_EVENT и CTRL_BREAK_EVENT, соответственно. Благодаря функции SetConsoleCtrlHandler, любая попытка каким-либо способом закрыть приложение приведёт не к закрытию приложения, а к вызову функции CtrlHandler, в которой мы устанавливаем объект события hEvent, который ждёт нить stopThread (листинг 10) и устанавливаем глобальный флаг Shutdown, подхватив который, функция forFD_WRITE (листинг 8) пошлёт клиенту строку команду «Shutdown». Новый клиент мы запрограммировали так, что получив эту команду, он закроет сокет на запись. В ответ на это сработает функция forFD_CLOSE сервера (листинг 9), которая закроет сокет и уберёт информацию о нём из списка. Далее сработает аналогичная функция на стороне клиента и он полностью закроет сокет со своей стороны.

Код функции MakeWorkerWindow приведен в листинге 2 работы 4. Код процедуры WindowProc окна приведен в листинге 6. Она отличается от кода в листинге 3 работы 4 лишь установкой флага события сети FD_CLOSE в функции _WSAAsyncSelect и реакцией на это событие в виде вызова функции forFD_CLOSE. Кроме того вызовы функций forFD_READ и forFD_WRITE проводится без параметра hwnd, так как он теперь глобальный.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam){
    static s=0;
    SOCKET AcceptS;
    struct sockaddr_in ca;
    int caSize = sizeof(ca);
    if (uMsg == WM_SOCKET){
    if (WSAGETSELECTERROR(lParam)){
    getpeername(wParam, (struct sockaddr *)&ca, &caSize);
        myprintf("WindowProc: Клиент %s:%d вызвал ошибку %d и будет отключен.\n ",
                inet_ntoa(ca.sin_addr),ntohs(ca.sin_port), WSAGETSELECTERROR(lParam));
         FreeSocketInformation(wParam);
       }
       else{
        switch(WSAGETSELECTEVENT(lParam)) {
          case FD_ACCEPT:
                int caSize = sizeof(ca);
                AcceptS = Accept(wParam, (struct sockaddr *)&ca, &caSize);
                CreateSocketInformation(Accept);
                myprintf("Число клиентов %d.\n", ++s);
                myprintf("Новый клиент %s:%d \n", inet_ntoa(ca.sin_addr),          ntohs(ca.sin_port));
                _WSAAsyncSelect(AcceptS, hwnd, WM_SOCKET, FD_READ|FD_WRITE|FD_CLOSE);//!!
                break;
        case FD_READ:
                forFD_READ(wParam);
                break;
        case FD_WRITE:
                forFD_WRITE(wParam);
                break;
        case FD_CLOSE: //New!!!
                forFD_CLOSE(wParam); //New!!!
                 break; //New!!!
          }
       }//else
     }
     return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
```

Листинг 6. Код функции WindowProc сервера.

Функции CreateSocketInformation, GetSocketInformation и FreeSocketInformation по работе со списком SocketInfoList аналогичны представленным в листингах 4, 5 и 6 работы 4, за исключением того, что в новой функции CreateSocketInformation (листинг 7) устанавливается тик времени создания сокета и устанавливается в ноль поле Shutdown

```
void CreateSocketInformation(SOCKET s){
  LPSOCKET_INFORMATION SI;
  if ((SI = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
    sizeof(SOCKET_INFORMATION))) == NULL)   {
    printf("GlobalAlloc() failed with error %d\n", GetLastError());
    return;
  }
  SI->Socket = s;
  SI->leftToRead = DATA_BUFSIZE;
  SI->leftToWrite = 0;
  SI->bp=SI->Buffer;
  SI->Next = SocketInfoList;
  SocketInfoList = SI;
  SI->Shutdown=0; //New!!!
  SI->tick=GetTickCount();//New!!!
}
```

Листинг 6. Код функции CreateSocketInformation сервера.

Для удобства поместите функции CreateSocketInformation, GetSocketInformation и FreeSocketInformation по работе со списком SocketInfoList в отдельный файл SocketInformation.c, предварив его оператором препроцессора #include "GracefulShutdown.h"

Новая функция FD_READ (листинг 7) отличается от одноимённой функции, приведенной в листинге 7 работы 4 наличием оператора обновления тика доступа к сокету SI->tick и наличием флага события о закрытии сокета FD_CLOSE в функции WSAAsyncSelect. Вынесите код листинга 6 в отдельный файл forFD_READ.c, иначе будьте осторожны с опреатором #include "GracefulShutdown.h" во избежание повторных определений.

```
#include "GracefulShutdown.h" //New!!!
void forFD_READ(SOCKET s){ //New!!! Один параметр
LPSOCKET_INFORMATION SocketInfo;
DWORD RecvBytes;
DWORD Flags;
if(!(SocketInfo = GetSocketInformation(s))){
        struct sockaddr_in ca;
        int caSize = sizeof(ca);
        getpeername(s, (struct sockaddr *)&ca, &caSize);
        myprintf("forFD_READ Клиент (%s:%d) отпал с ошибкой %s.\n",
                inet_ntoa(ca.sin_addr),ntohs(ca.sin_port),
                encodeWSAGetLastError(WSAGetLastError()));
        return;
}
SocketInfo->tick=GetTickCount();//New!!!
if(SocketInfo->leftToRead>0){
        SocketInfo->DataBuf.buf = SocketInfo->bp;
        SocketInfo->DataBuf.len = SocketInfo->leftToRead;
        Flags = 0;
        if (WSARecv(s, &(SocketInfo->DataBuf), 1, &RecvBytes,
                        &Flags, NULL, NULL) == SOCKET_ERROR){
            if (WSAGetLastError() != WSAEWOULDBLOCK){
                printf("forFD_READ; WSARecv () failed with error %d\n",WSAGetLastError());
                FreeSocketInformation(s);
                 return;
            }
        }
        else {
                SocketInfo->bp += RecvBytes;
                SocketInfo->leftToRead -= RecvBytes;
                if(SocketInfo->leftToRead==0){
                        //Можно обрабатывать полученную информацию
                        //Мы делаем просто Эхо
                        SocketInfo->leftToWrite=DATA_BUFSIZE;
                        SocketInfo->bp = SocketInfo->Buffer;
```

```
                        WSAAsyncSelect(s, hwnd, WM_SOCKET,
                                FD_READ|FD_WRITE|FD_CLOSE); //New!!!
                }
            }
}
}
```

Листинг 7. Код функции forFD_READ сервера

Новая функция FD_ WRITE (листинг 8) отличается от одноимённой функции, приведенной в листинге 8 работы 4 наличием нового фрагмента, который отмечен в комментариях. В новом фрагменте в момент времени, когда сервер только начинает отвечать клиенту (SocketInfo->leftToWrite==DATA_BUFSIZE) и установлен глобальный флаг Shutdown на закрытие (он устанавливается в функции CtrlHandler) вместо эхо ответа сервер отсылает клиенту команду на закрытие в виде строки "Shutdown".

Вынесите код листинга 8 в отдельный файл forFD_ WRITE.c, иначе будьте осторожны с оператором #include "GracefulShutdown.h" во избежание повторных определений.

```
#include "GracefulShutdown.h"
void forFD_WRITE(SOCKET s){
LPSOCKET_INFORMATION SocketInfo;
DWORD SendBytes;
DWORD Flags;
if(!(SocketInfo = GetSocketInformation(s))){
        struct sockaddr_in ca;
        int caSize = sizeof(ca);
        getpeername(s, (struct sockaddr *)&ca, &caSize);
        myprintf("forFD_WRITE Клиент (%s:%d) отпал с ошибкой %s.\n",
                        inet_ntoa(ca.sin_addr),ntohs(ca.sin_port),
                        encodeWSAGetLastError(WSAGetLastError()));
        }
//Начало нового
SocketInfo->tick=GetTickCount();
if(SocketInfo->leftToWrite==DATA_BUFSIZE && Shutdown && !SocketInfo->Shutdown){
        SocketInfo->Shutdown=1;
        SocketInfo->DataBuf.buf = "Shutdown";
        SocketInfo->DataBuf.len = strlen("Shutdown")+1;
        Flags = 0;
        if (WSASend(s, &(SocketInfo->DataBuf), 1, &SendBytes, 0,
                        NULL, NULL) == SOCKET_ERROR){
                if (WSAGetLastError() != WSAEWOULDBLOCK){
                        printf("forFD_WRITE: WSASend() failed with error %d\n", WSAGetLastError());
                        FreeSocketInformation(s);
                        return;
                }
        }
        else{
                SocketInfo->leftToWrite=0;
                SocketInfo->leftToRead=0;
                return;
        }
}
//Конец нового
if(SocketInfo->leftToWrite>0){
        SocketInfo->DataBuf.buf = SocketInfo->bp;
        SocketInfo->DataBuf.len = SocketInfo->leftToWrite;
        Flags = 0;
        if (WSASend(s, &(SocketInfo->DataBuf), 1, &SendBytes, 0,
                        NULL, NULL) == SOCKET_ERROR){
                if (WSAGetLastError() != WSAEWOULDBLOCK){
                        printf("forFD_WRITE WSASend() failed with error %s\n",
                                WSAGetLastError(WSAGetLastError());
                        FreeSocketInformation(s);
                        return;
                }
```

```
		}
		else{
			SocketInfo->bp += SendBytes;
			SocketInfo->leftToWrite-= SendBytes;
			if(SocketInfo->leftToWrite==0){
				SocketInfo->leftToRead=DATA_BUFSIZE;
				SocketInfo->bp = SocketInfo->Buffer;
				_WSAAsyncSelect(s, hwnd, WM_SOCKET,
				 FD_READ|FD_WRITE|FD_CLOSE); //New!!!
			}
		 }
}
}
```

Листинг 8. Код функции forFD_ WRITE сервера

Функции forFD_CLOSE (Листинг 9) нет в работе 4. Она запускается при сетевом событии FD_CLOSE о закрытии сокета. Если не установлен глобальный флаг Shutdown, то эта функция вызывается, когда клиент инициировал закрытие соединения. В этом случае мы сбрасываем в _WSAAsyncSelect для этого сокета s реакцию на все сетевые события. Иначе, клиент закрывается в ответ на нашу команду о закрытии. В обоих случаях мы закрываем сокет на запись и освобождаем информацию о сокете из списка.

Вынесите код листинга 9 в отдельный файл forFD_ CLOSE.c иначе будьте осторожны с оператором #include "GracefulShutdown.h" во избежание повторных определений.

```
#include "GracefulShutdown.h"
void forFD_CLOSE(SOCKET s){
static n=1;
LPSOCKET_INFORMATION SocketInfo;
struct sockaddr_in ca;
int caSize;
if(!(SocketInfo = GetSocketInformation(s))){
		caSize = sizeof(ca);
		getpeername(s, (struct sockaddr *)&ca, &caSize);
		myprintf("forFD_CLOSE: Клиент (%s:%d) отпал с ошибкой %d.\n",
			inet_ntoa(ca.sin_addr),ntohs(ca.sin_port),WSAGetLastError());
		return;
}
caSize = sizeof(ca);
getpeername(s, (struct sockaddr *)&ca, &caSize);
if(!Shutdown){
		myprintf("FD_CLOSE:Подтверждаем клиенту №%d (%s:%d) его запрос на закрытие \n",
			n++, inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
		_WSAAsyncSelect(s, hwnd, WM_SOCKET, 0);
		}
else
		myprintf("FD_CLOSE:Клиент №%d (%s:%d) понял наш Shutdown и мы подтверждаем   его
			 запрос на закрытие\n",n++, inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
shutdown(s, SD_SEND);
FreeSocketInformation(s);
}
```

Листинг 9. Код функции forFD_CLOSE сервера.

Функции stopThread (Листинг 10) нет в работе 4. Она ждёт события hEvent из функции CtrlHandler. Как только будет осуществлена попытка любым способом закрыть приложение, функция CtrlHandler установит объект события hEvent. С помощью функции SetConsoleCtrlHandler восстанавливается стпндартная реакция на закрытие приложения. Далее функция stopThread в бесконечном цикле ждёт закрытия всех сокетов и освобождения выделенных ресурсов (что соответствует пустоте списка SocketInfoList) и закрывает приложение.

```
DWORD WINAPI stopThread(LPVOID lpParam){
```

```
	WSAEVENT EA[1];
	EA[0]=hEvent;
	WSAWaitForMultipleEvents(1,EA, TRUE, WSA_INFINITE, FALSE);
	SetConsoleCtrlHandler(CtrlHandler, FALSE);

	while(1){
		if(!SocketInfoList){
			WSACleanup();
			myprintf("Возможно Ждём stopDeadClientsThread %d c.\n", TIMEOUT);
			WaitForSingleObject(stopDeadClientsThread, TIMEOUT);
			myprintf("Сервер завершил свою работу. Нажмите Enter.\n");
			getchar();
			ExitProcess(0);
		}
		else{
			myprintf("Еще подключены клиенты.\n");
		}
	}
	return 0;
}
```

Листинг 10. Код функции stopThread сервера

Функции stopDeadClientsThread (Листинг 11) нет в работе 4. Функция отключает клиентов, которые не активны более TIMEOUT секунд. В бесконечном цикле с интервалом в TIMEOUT секунд фиксируются неактивные более TIMEOUT секунд сокеты. Эти сокеты закрываются, а информация о них удаляется из списка SocketInfoList.

```
DWORD WINAPI stopDeadClientsThread(LPVOID lpParam){
SOCKET_INFORMATION *SI = SocketInfoList;
SOCKET_INFORMATION *PrevSI = NULL;
DWORD tick, timeout = TIMEOUT*1000;
struct sockaddr_in ca;
int caSize;
while(1){
Sleep(timeout);
tick=GetTickCount();
PrevSI = NULL;
if(NULL==(SI = SocketInfoList)) continue;
while(SI) {
	if ((tick - SI->tick)>timeout){
		if (PrevSI)
			PrevSI->Next = SI->Next;
		else
			SocketInfoList = SI->Next;
		caSize = sizeof(ca);
		getpeername(SI->Socket, (struct sockaddr *)&ca, &caSize);
			myprintf("Таймаут для клиента %s:%d\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
			closesocket(SI->Socket);
			GlobalFree(SI);
			break;
		}
		PrevSI = SI;
		SI = SI->Next;
	}
 }
}
```

Листинг 11. Код функции stopDeadClientsThread сервера

Клиент

Рассмотрим модифицированный клиент. Его заголовочный файл приведен в листинге 12. Новое по сравнению с листингом 9 из работы 4 следующее. Изменены названия функций forFD_WRITEClient и void

forFD_READClient. Добавлена функция forFD_CLOSEClient. Введено несколько глобальных флагов Shutdown, ShutdownS, q, C.

```
#include <winsock2.h>
#include <stdio.h>
#include "../DATA_BUFSIZE.h"
#define WM_SOCKET (WM_USER + 1)
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET WSAAPI Socket(int af,int type,int protocol);
int _WSAAsyncSelect( SOCKET s,HWND hWnd,unsigned int wMsg,long lEvent);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Connect(SOCKET s, struct sockaddr* ServerAddr, int );
HANDLE WINAPI _CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,      LPTHREAD_START_ROUTINE lpStartAddress,
        LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
DWORD WINAPI ClientThread(LPVOID lpParam);
HWND MakeWorkerWindow(void);
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
void forFD_WRITEClient(SOCKET s); //New!!!
void forFD_READClient(SOCKET s); //New!!!
void forFD_CLOSEClient(SOCKET s); //New!!!
extern volatile int   Shutdown, ShutdownS; //New!!!
extern int needNewRnd;
extern int *msgForR, *msgForW;
extern WSABUF DataBuf;
extern char *bp;
extern volatile DWORD leftToRead;
extern volatile DWORD leftToWrite;
extern volatile int C;//- для непрерывной передачи данных серверу//New!!!
int q; //New!!!
```

Листинг12. Заголовочный файл GracefulShutdownClient.h клиента.

Рассмотрим функцию main клиента, приведенную в лисинге 13. Она значительно отличается от кода в листинге 9.

Рассмотрим вначале глобальные переменные. Глобальный флаг Shutdown устанавливается при инициализации корректного закрытия сокета со стороны клиента при непрерывном режиме работы. . Глобальный флаг ShutdownS устанавливается при инициализации корректного закрытия сокета со стороны сервера командой строкой «Shutdown». Флаг C устанавливается, если задан непрерывный режим работы. Флаг q устанавливается в прерывном режиме работы, если дана команда на выход из программы.

При старте функции main проверяется корректность аргументов командной строки. Если задан непрерывный режим работы (if(argv[2])), то устанавливаем флаг C. Осуществляется соединение с сервером, адрес которого задан в командной строке. Создаётся рабочее окно hwnd с помощью функции MakeWorkerWindow. Выделяется память для буферов вывода msgForW и ввода msgForR. Устанавливается реакция приложения в виде вызова функции CtrlHandler на попытку его закрытия. Далее приложение входит в главный цикл по приёму и обработке оконных сообщений.

```
#include "GracefulShutdownClient.h"
volatile int   Shutdown=0, ShutdownS=0; //New!!!
int needNewRnd=1;
int *msgForR, *msgForW;
WSABUF DataBuf;
char *bp;
volatile DWORD leftToRead=0;
volatile DWORD leftToWrite =DATA_BUFSIZE;
volatile int C=0;//- для непрерывной передачи данных серверу//New!!!
int q=0; //New!!!

void main(int argc, char **argv){
  WSADATA            wsaData;
  SOCKET             s;
```

```
    struct sockaddr_in         ServerAddr, ca;
    int                Port = 5150;
    int                Ret;
    MSG msg;
    HWND hwnd;
    int caSize = sizeof(ca);

    if (argc <= 1)   {
      myprintf("Использование: tcpclient <Server IP address> [c].\n");
      myprintf("c - для непрерывной передачи данных серверу.\n");//New!!!
      return;
    }
  if(argv[2])    C=1;
   _ WSAStartup(MAKEWORD(2,2), &wsaData);
  s = Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);
    ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
    Connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr));
       getsockname(s, (struct sockaddr *)&ca, &caSize);
       myprintf("Локальный адрес:%s и порт:%d сокета клиента\n",inet_ntoa(ca.sin_addr),ntohs(ca.sin_port));
       if ((hwnd = MakeWorkerWindow()) == NULL)      return 0;
       msgForW = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
       msgForR = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
       WSAAsyncSelect(s, hwnd, WM_SOCKET,  FD_READ|FD_WRITE|FD_CLOSE);
       if( !SetConsoleCtrlHandler(CtrlHandler, TRUE) ) {
                myprintf("SetConsoleCtrlHandler() failed to install console handler: %d\n",
                                  GetLastError());
                return;
       }
       if(C){
                myprintf("Обмениваемся данными с Эхо сервером .\n");
                myprintf("Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то клиетн плавно
                 завершит  свою работу.\n");
       }
  while(GetMessage(&msg, NULL, 0, 0)){
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
       CloseWindow(hwnd);
       SetConsoleCtrlHandler(CtrlHandler, FALSE);
       Myprintf("Для завершения работы нажмите Enter.");
       getchar();
       getchar();
  }
```

Листинг 13. Главная функция клиента WSAAsyncSelec.

Рассмотрим функцию CtrlHandler, приведенную в лисинге 13. Она отличается от кода функции CtrlHandler сервера листинге 5 отсутствием вызова функции WSASetEvent. Мы лишь устанавливаем глобальный флаг Shutdown о закрытии приложения.

```
BOOL WINAPI CtrlHandler (DWORD dwEvent) {

switch( dwEvent ) {
        case CTRL_BREAK_EVENT:
        case CTRL_C_EVENT:
        case CTRL_LOGOFF_EVENT:
        case CTRL_SHUTDOWN_EVENT:
        case CTRL_CLOSE_EVENT:
        Shutdown=1;
        return(TRUE);
        default:
        return(FALSE);
```

```
        }
}
```

Листинг 13. Код функции CtrlHandler.

Рассмотрим функцию WindowProc клиента, приведенную в листинге 14. Она отличается от кода функции WindowProc в листинге 11 из работы 4 наличием вызова обработчика сетевого события FD_CLOSE о закрытии сокета.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam){
  if (uMsg == WM_SOCKET)   {
    if (WSAGETSELECTERROR(lParam))      {
      myprintf("Socket failed with error %d\n", WSAGETSELECTERROR(lParam));
      closesocket(wParam);
    }
    else     {
      switch(WSAGETSELECTEVENT(lParam)){
        case FD_WRITE:
                                        forFD_WRITEClient(wParam);
                                        break;
        case FD_READ:
                                        forFD_READClient(wParam);
                                        break;
        case FD_CLOSE: //New!!!
                                        forFD_CLOSEClient(wParam); //New!!!
                                        break; //New!!!
      }
    }
    return 0;
  }
```

Листинг 14. Код функции WindowProc.

Рассмотрим функцию forFD_WRITEClient клиента, приведенную в листинге 14. Она отличается от кода функции forFD_WRITE в листинге 12 из работы 4 наличием флага FD_CLOSE в вызове функции WSAAsyncSelect, а также наличием кода для организации прерывного обмена. Если пользователь ввёл c<Enter>, то серверу отсылается очередной пакет. Если введено q<Enter>, то сокет закрывается на запись и устанавливается флаг q.

```
void forFD_WRITEClient(SOCKET s){
DWORD Flags, SendBytes;
//Начало нового!!!
if(!C && leftToWrite==DATA_BUFSIZE){ //прерывный обмен
          char c;
          myprintf("c<Enter> - послать порцию данных.\n");
          myprintf("q<Enter> - выход.\n");
          while(1){
                    c=getchar();
                    if(c=='c')break;
                    if(c=='q'){
                              q=1;
                              shutdown(s, SD_SEND);
                              myprintf(" Функция forFD_WRITEClient приняла команду q, выполнила
                                        shutdown(SD_SEND).\n");
                              return;
                    }
          }
}
//Конец нового!!!

if(needNewRnd){
          int i;
          //Моделирования данных передаваемых на сервер для обработки
          for(i=0;i<DATA_BUFSIZE_INT;i++)msgForW[i]=rand();
          bp=(char*)msgForW;
```

```
        needNewRnd=0;
}
if(leftToWrite>0){
        DataBuf.buf = bp;
        DataBuf.len = leftToWrite;
        Flags = 0;
        if (WSASend(s, &DataBuf, 1, &SendBytes, 0,          NULL, NULL) == SOCKET_ERROR){
                if (WSAGetLastError() != WSAEWOULDBLOCK){
                        printf("forFD_ WRITE: WSASend failed with error %d\n",
                                        WSAGetLastError());
                        closesocket(s);
                        return;
                }
        }
        else{
                bp += SendBytes;
                leftToWrite-= SendBytes;
                if(leftToWrite==0){
                        leftToRead=DATA_BUFSIZE;
                        bp = (char*)msgForR;
                }
        }
}
}
```

Листинг 15. Код функции forFD_WRITEClient.

Рассмотрим функцию forFD_ READClient клиента, приведенную в листинге 16. Она отличается от кода функции forFD_ READ в листинге 13 из работы 4 наличием двух новых фрагменты кода. В первом фрагменте функция в самом начале приёма ответного пакета от сервера ищет в начале пакета строку-команду "Shutdown". Если она есть, то начинаем плавно закрываться. Устанавливаем переменные leftToWrite и leftToRead в ноль, говоря тем самым, что нам ничего больше читать и писать не надо. Закрываем сокет по записи. Устанавливаем флаг ShutdownS о начале процедуры закрытия по инициативе от сервера.

Рассмотрим второй новый фрагмент кода. При плавном закрытии сокета главное выбрать момент закрытия. В нашем случае таким моментом является случай, когда полностью принят ответ от сервера (leftToRead==0). Если в этот момент установлен в функции CtrlHandler флаг закрытия Shutdown, то закроем сокет на запись.

```
#include "GracefulShutdownClient.h"
void forFD_READClient(SOCKET s){
DWORD RecvBytes, Flags = 0;
if(leftToRead>0){
    DWORD Flags = 0;
    DataBuf.buf = bp;
    DataBuf.len = leftToRead;
    if (WSARecv(s, &DataBuf, 1, &RecvBytes, &Flags, NULL, NULL) == SOCKET_ERROR){
        if (WSAGetLastError() != WSAEWOULDBLOCK){
            myprintf("forFD_READ: WSARecv failed with error %s\n", WSAGetLastError());
            closesocket(s);
            return;
        }
    }
    else {
        //Начало нового 1 !!!
        if(leftToRead==DATA_BUFSIZE && !memcmp(msgForR,"Shutdown", strlen("Shutdown")+1)){
            leftToWrite=0;
            leftToRead=0;
            shutdown(s, SD_SEND);
            myprintf(" Функция forFD_READClient приняла от сервера фразу ShutDown и выполнила
                                                    shutdown(SD_SEND).\n");
            ShutdownS=1;
            return;
        }
```

```
        //Конец нового 1 !!!
        bp += RecvBytes;
        leftToRead -= RecvBytes;
        if(leftToRead==0){
                //Начало нового 2 !!!
                if(Shutdown){
                        shutdown(s, SD_SEND);
                        myprintf(" Функция forFD_READClient приняла все данные от сервера и  выполнила
                                        shutdown(, SD_SEND).\n");
                        return;
                }
                //Конец нового 2 !!!
                leftToWrite=DATA_BUFSIZE;
                bp=(char*)msgForW;
                needNewRnd=1;
                if(memcmp(msgForR,msgForW,DATA_BUFSIZE)){
                        puts("ERROR\n");
                        Beep(333,3333);
                        //exit(77);
                }
                forFD_WRITEClient(s); //New!!!
        }
    }//else
 }// if(leftToRead>0)
}
```

Листинг 16. Код функции forFD_ READClient.

Рассмотрим функцию forFD_CLOSEClient клиента, приведенную в листинге 17. Логика её работы видна из текста программы. Функция PostQuitMessage разорвёт цикл GetMessage в функции main и приложение закроется.

```
#include "GracefulShutdownClient.h"
void forFD_CLOSEClient(SOCKET s){
        if(!ShutdownS&&!Shutdown&!q)
                myprintf("Таймаут.\n");
        myprintf("Функция forFD_CLOSEClient получила подтверждающий shutdown(, SD_SEND) от
сервера\n");
        closesocket(s);
        WSACleanup();
        GlobalFree(msgForR);
        GlobalFree(msgForW);
        PostQuitMessage(0);
}
```

Листинг 17. Код функции forFD_CLOSEClient

Результаты работы программы.

1. Запустим сервер в отдельном консольном окне с помощью команды
start GracefulShutdownAsyncSelect.exe.
В другом окне запустим по крайней мере один клиент в прерывном режиме
start GracefulShutdownAsyncSelectClient.exe 172.17.17.2
Для завершения работы клиента нажмём в его окне q. Результат работы приведен на рисунке 3. Проследите по листингам программ ход выполнения клиента и сервера.

2. Перезапустим сервер и запустим клиент в прерывном режиме. Остановим сервер. Введём в окне клиента команду продолжения работы «c». Результат работы приведен на рисунке 4. Проследите по листингам программ ход выполнения клиента и сервера.

3. Перезапустим сервер и запустим клиент в непрерывном режиме работы
start GracefulShutdownAsyncSelect.exe 172.17.17.2 c
Остановим клиент. Результат работы приведен на рисунке 5. Проследите по листингам программ ход выполнения клиента и сервера.

4. Перезапустим сервер и запустим клиент в непрерывном режиме работы. Остановим сервер. Результат работы приведен на рисунке 6. Проследите по листингам программ ход выполнения клиента и сервера.

5. Перестроим сервер, задав в нём таймаут TIMEOUT 5 секунд. Запустим сервер и затем запустим клиент в прерывном режиме работы. Подождём 5 секунд и в ведём в окне клиента команду продолжения работы «c». Результат работы приведен на рисунке 7. Проследите листингам программ ход выполнения клиента и сервера.

```
Локальный адрес:172.17.17.2 и порт:1031 сокета клиента
c<Enter> - послать порцию данных.
q<Enter> - выход.
q
 Функция forFD_WRITEClient приняла команду q и выполнила shutdown(SD_SEND).
Функция forFD_CLOSEClient получила подтверждающий shutdown(, SD_SEND) от сервера

Для завершения работы нажмите Enter.
```

Клиент.

```
Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то сервер плавно завер
шит  свою работу.
Новый клиент 172.17.17.2:1031
Число клиентов 1.
FD_CLOSE:Подтверждаем клиенту 1 (172.17.17.2:1031) его запрос на закрытие  shutd
own
```

Сервер.

Рис. 3. Прерывный режим. Клиент завершает работу.

```
Локальный адрес:172.17.17.2 и порт:3032 сокета клиента
c<Enter> - послать порцию данных.
q<Enter> - выход.
c
 Функция forFD_READClient приняла от сервера фразу ShutDown, выполнила shutdown(
SD_SEND).
Функция forFD_CLOSEClient получила подтверждающий shutdown(, SD_SEND) от сервера

Для завершения работы нажмите Enter._
```

Клиент.

```
Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то сервер плавно завер
шит  свою работу.
Новый клиент 172.17.17.2:3032
Число клиентов 1.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
FD_CLOSE:Клиент 1 (172.17.17.2:3032) понял наш Shutdown и мы подтверждаем  его
запрос на закрытие
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Возможно ждём stopDeadClientsThread 8885 c.
Сервер завершил свою работу. Нажмите Enter.
```

Сервер.

Рис.4. Прерывный режим. Сервер завершает работу

```
Локальный адрес:172.17.17.2 и порт:3495 сокета клиента
Обмениваемся данными с Эхо сервером .
Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то клиетн плавно завер
шит  свою работу.
 Функция forFD_READClient приняла все данные от сервера и  выполнила shutdown(,
SD_SEND).
Функция forFD_CLOSEClient получила подтверждающий shutdown(, SD_SEND) от сервера

Для завершения работы нажмите Enter.
```

Клиент.

```
Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то сервер плавно завер
шит  свою работу.
Новый клиент 172.17.17.2:3495
Число клиентов 1.
FD_CLOSE:Подтверждаем клиенту 1 (172.17.17.2:3495) его запрос на закрытие  shutd
own
```

Сервер.

Рис.5. Непрерывный режим. Клиент завершает работу.

```
Локальный адрес:172.17.17.2 и порт:3970 сокета клиента
Обмениваемся данными с Эхо сервером .
Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то клиетн плавно завер
шит  свою работу.
 Функция forFD_READClient приняла от сервера фразу ShutDown, выполнила shutdown(
SD_SEND).
Функция forFD_CLOSEClient получила подтверждающий shutdown(, SD_SEND) от сервера

Для завершения работы нажмите Enter.
```

Клиент.

```
Новый клиент 172.17.17.2:3970
Число клиентов 1.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
FD_CLOSE:Клиент 1 (172.17.17.2:3970) понял наш Shutdown и мы подтверждаем   его
запрос на закрытие
Еще подключены клиенты.
Еще подключены клиенты.
Еще подключены клиенты.
Возможно ждём stopDeadClientsThread 8885 с.
Сервер завершил свою работу. Нажмите Enter.
```

Сервер.

Рис.6. Непрерывный режим. Сервер завершает работу

```
Локальный адрес:172.17.17.2 и порт:4462 сокета клиента
c<Enter> - послать порцию данных.
q<Enter> - выход.
c
Таймаут.
Функция forFD_CLOSEClient получила подтверждающий shutdown(, SD_SEND) от сервера

Для завершения работы нажмите Enter.
```

Клиент.

```
Если вы закроете это окно, нажмёте Ctrl+C или Ctrl+Break, то сервер плавно завер
шит  свою работу.
Новый клиент 172.17.17.2:4462
Число клиентов 1.
Таймаут для клиента с адресом:172.17.17.2 на порту:4462
```

Сервер.

Рис. 7. Прерывный режим. Сервер завершает работу клиента по таймауту.

Контрольные вопросы.

1. Почему сетевые приложения нельзя сразу закрывать?
2. Изложите процедуру корректного закрытия сетевых приложений.
3. Чем отличается закрытие сокета от его отключения.
4. Что такое последовательность отключения?
5. Чем отличается корректная последовательность отключения от обрывающей или тяжёлой?
6. Кто инициирует неявную последовательность отключения?
7. Как управлять неявной последовательность отключения?
8. Каких поведений функции closesocket и как можно добиться?

9. Объясните назначение второго параметра функции shutdown.
10. Когда функция shutdown отправляет получателю пакет FIN?
11. Почему для корректного закрытия сокетов рекомендуют использовать функцию shutdown?
12. Почему для корректного закрытия сокетов рекомендуют использовать функцию WSAAsyncSelect или WSAEventSelect?
13. Прокомментируйте содержание и смысл таблицы 1.
14. Что такое вычислительная нить (поток)?
15. Когда нити взаимодействуют друг с другом?
16. Какие примитивы синхронизации Windows изучаются в данной работе?
17. Зачем нужны критические секции и как ими пользоваться в программе?
18. Что делает функция WSAWaitForMultipleEvents?
19. Как создать, установить,сбросить и закрыть событие Winsock?
20. Что произойдёт с двумя ожидающими события (на функции WSAWaitForMultipleEvents) нитями, если ожидаемое событие произойдёт?
21. Может ли нить сама себе установить событие и когда?
22. Как много ресурсов процессорного времени потребляет нить, ожидающая события?
23. Почему в прогамме на листинге 3 оператор printf функции consumerThread всегда выполнится после оператора printf функции producerThread?
24. Сопоставьте содержимое скриншотов на рисунках 1 и 2 с листингами 1 и 2. Укажите, когда и какая функция вывода на консоль в листингах выводит каждую строку на скриншоте.
25. Сопоставьте содержимое скриншотов на рисунках 3 -7 с листингами 3-15. Укажите, когда и какая функция вывода на консоль в листингах выводит каждую строку на скриншоте.

Задания

4. Загрузите в Visual Studio 2008 проект сервера ShutdownSequence (листинг 1) и клиента ShutdownSequenceClient (листинг 2) из решения lab5 архива src. Произведите отладку проектов. Получите скриншоты, приведенные на рисунках 1 и 2.
5. Загрузите и отладьте в Visual Studio 2008 проект WSAWaitForMultipleEvents из решения lab5 архива src (листинг 3).
6. Загрузите в Visual Studio 2008 проект сервера GracefulShutdownAsyncSelect (листинги 4-11) и клиента GracefulShutdownAsyncSelectClient (листинги 12-17) из решения lab5 архива src. Произведите отладку проектов. Получите скриншоты, приведенные на рисунках 3-7. Используя подход, изложенный в работе №2, дополните проекты клиента и сервера кодом, позволяющим выводить в окно консоли текущую и среднюю скорость обмена информацией.
7. Сделайте так, чтобы программы клиента ClientF и серверов tcpTreadServerF, tcpSelectServerF из работы 2 корректно завершали свою работу.

Лабораторная работа №6. Асинхронные сокеты. Модель, основанная на применении функции WSAEventSelect

Теоретическая часть

В Winsock поддерживается еще одна полезная модель асинхронного ввода-вывода, позволяющая получать уведомления о сетевых событиях на сокетах. Эта модель похожа на WSAAsyncSelect — приложение получает и обрабатывает те же сетевые события. Но есть и отличие — сообщения отправляются дескриптору объекта события, а не процедуре окна. Модель WSAEventSelect требует, чтобы приложение создало объект события для каждого сокета, вызвав функцию WSACreateEvent.

Функция WSAEventSelect ассоциирует объект события с определённым множеством FD_XXX сетевых событий. Прототип функции определён в файле Winsock2.h. Библиотека импорта - Ws2_32.lib. Код расположен в Ws2_32.dll.

```
int WSAEventSelect(SOCKET s,WSAEVENT hEventObject,long lNetworkEvents);
```

Параметр s определяет сокет. Параметр hEventObject задаёт существующий объект события, который будет установлен, если случится одно из сетевых событий, заданных множеством lNetworkEvents.

Если не произошла ошибка, то функция возвращает ноль, что говорит о том, что интерес приложения к сетевым событиям будет удовлетворен. Иначе функция возвращает SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEINVAL, WSAEINPROGRES, WSAENOTSOCK . Расшифровку ошибок смотри в приложении или в MSDN.

Как и в случае функций select и WSAAsyncSelect функция WSAEventSelect используется для определения того момента времени, когда последующие операции записи или чтения в сокет выполнятся немедленно без ожидания и успешно.

При возникновении сетевого события объект события будет установлен только один раз и не будет переустановлен заново, если только приложение не сделает повторного вызова некоей возобновляющая функции Winsock, а именно:

Для сетевого события FD_READ возобновляющими функциями будут recv, recvfrom, WSARecv или WSARecvFrom. Для FD_WRITE - send, sendto, WSASend или WSASendTo. Для FD_ACCEPT - accept или WSAAccept, если только не возникнет ошибка WSATRY_AGAIN.

Нет функций, приводящих к повторной установки объетов событий для сетевых событий FD_CONNECT и FD_CLOSE

Для событий FD_READ, FD_OOB и FD_ACCEPT вызов возобновляющей функции приводит к повторной установке объекта события, если всё ещё имеются условия для существования этих сетевых событий. Это позволяет приложениям быть событийно управляемыми. Рассмотрим следующую последовательность.

Сетевой транспортный стек получил 100 байтов данных на сокете s и Winsock установил объект события для FD_READ. Приложение прочитало 50 байт с помощью функции recv(s, , 50, 0). Поскольку остались данные для чтения, Winsock установит объект события для FD_READ ещё раз. В ответ на сообщение FD_READ приложение может не читать все данные доступные для чтения. Пока останутся непрочитанные данные, приложение будет устанавливать объект события для FD_READ.

Рассмотрим следующую последовательность.

Приложение вызывает функцию listen. Поступил, но ещё не принят запрос на соединение. Приложение вызывает функцию WSAEventSelect, указывая, что оно требует получения сообщений о событии FD_ACCEPT. Так как соответствующее условие имеет место, то Winsock устанавливает объект события о сетевом событии FD_ACCEPT.

Для FD_WRITE объект события устанавливается, когда сокет соединяется (после сообщения о FD_CONNECT, если это зарегистрировано) с помощью функции connect или WSAConnect, либо принимается с помощью функции accept или WSAAccept и далее когда операция отправки закончится с ошибкой WSAEWOULDBLOCK и буфер для отправки станет доступным. Следовательно, приложение может полагать, что отправка данных доступна начиная от первого сообщения о событии FD_WRITE и до тех пор, пока отправляющая функция не вернёт код ошибки WSAEWOULDBLOCK. После такой ошибки приложение будет извещено о возможности новой отправки с помощью повторной установки объекта события о сетевом событии FD_WRITE.

Код ошибки в объекте событии о FD_CLOSE показывает был ли сокет закрыт корректно или некорректно. Если код ошибки равен нулю то закрытие было корректным. Если код ошибки равен WSAECONNRESET, то соединение было закрыто некорректно. Это касается только сокетов, ориентированных на соединение, таких как SOCK_STREAM.

Объект события для FD_CLOSE устанавливается, когда сокет переходит в состояние TIME_WAIT или CLOSE_WAIT. Это является результатом того, что на противоположном конце соединения была вызвана функция shutdown с опцией о прекращении передачи или функция closesocket.

При корректном закрытии сокета для приложения будет установлен объект события о FD_CLOSE после того, как все данные будут получены. Свидетельством конца корректного закрытия является сброс объекта события FD_READ.

Функция WSAEventSelect выполняется почти также как и функция WSAAsyncSelect. Разница видна в том, какие действия выполняются при возникновении события в сети. Функция WSAAsyncSelect организует отправку Windows сообщений. Функция WSAEventSelect устанавливает заданный в ней объект события и записывает информацию о сетевом событии во внутренние сетевые структуры. Приложение может использовать WSAWaitForMultipleEvents для ожидания или опроса объекта события и использовать WSAEnumNetworkEvents для извлечения содержимого внутренних сетевых структур и определения какое сетевое событие имело место.

Функуция WSAEventSelect автоматически устанавливает сокет s в неблокирующий режим в независимости от значения параметра lEvent. Для перевода сокета в блокирующий режим нужно вначале вызвать функцию WSAEventSelect с нулевым параметром lEvent и затем вызвать функцию ioctlsocket с соответствующим параметром.

Параметр lEvent конструируется с помощью операции побитового ИЛИ из следующих величин – кодов сетевых событий:

FD_READ – желание получать сообщения о готовности сокета к чтению.

FD_WRITE - желание получать сообщения о готовности сокета к записи.

FD_ACCEPT - желание получать сообщения о готовности сокета к приёму входных соединений.

FD_CONNECT - желание получать сообщения о завершении установки соединения.

FD_CLOSE - желание получать сообщения о закрытии сокета.

FD_QOS и FD_GROUP_QOS связано с качеством обслуживания QOS.

FD_ROUTING_INTERFACE_CHANGE - желание получать сообщения об изменениях маршрутизирующих интерфейсов.

FD_ADDRESS_LIST_CHANGE - желание получать сообщения об изменении списка локальных адресов.

Вызов функции WSAAsyncSelect для сокета сбрасывает все предыдущие установки, осуществлённые этой функцией или функцией WSAEventSelect. Например, для сопоставления объекта события с сетевыми событиями о возможности о записи и чтения следует сделать вызов

```
rc = WSAEventSelect(s, hEventObject, FD_READ|FD_WRITE);
```

Нельзя определить разные объекты события для разных сетевых событий. Следующий код работать не будет

```
rc = WSAEventSelect(s, hEventObject1, FD_READ);
rc = WSAEventSelect(s, hEventObject2, FD_WRITE); //Плохо
```

Для отключения сопоставления объекта события с сетевыми событиями на сокете и для отключения реакции сокета на сетевые события, следует осуществить вызов

```
rc = WSAEventSelect(s, hEventObject, 0);
```

Параметр hEventObject при этом игнорируется.

Закрытие сокета с посощью closesocket также отключает сопоставление объекта события с сетевыми событиями и отключает реакции сокета на сетевые события. Приложение, однако, должно в явном виде закрыть объект события с помощью функции WSACloseEvent и освободить тем самым системные ресурсы.

Сокет, созданный функцией accept, имеет те же свойства, что и слушающий сокет, указанный в аргументах функции accept. Следовательно, множество сетевых событий для слушающего сокета также применится к созданному сокету. Например, если слушающий сокет имеет события WSAEventSelect в виде FD_ACCEPT, FD_READ и FD_WRITE, то и новый сокет, принятый на этом прослушивающем сокете также будет реагировать на события FD_ACCEPT, FD_READ и FD_WRITE путём установки объекта события. Если это не устраивает, то для нового сокета, принятого функцией accept, следует отдельно вызвать функцию WSAEventSelect с желаемым множеством событий и новым объектом события.

Модель WSAEventSelect требует, чтобы приложение создало объект события для каждого сокета, вызвав функцию WSACreateEvent, которая возвращает дескриптор объекта события. Получив дескриптор объекта нужно связать его с сокетом и зарегистрировать те типы сетевых событий, которые надо отслеживать. Это достигается вызовом функции WSAEventSelect.

У объекта события имеются два рабочих состояния — свободное (signaled) и занятое (nonsignaled), а также два оперативных режима —ручного (manual) и автоматического сброса (auto reset). Первоначально событие создается в занятом состоянии и режиме ручного сброса. Когда на сокете происходит сетевое событие, связанный с эти событием объект становится свободным. Так как объект события создается в режиме ручного сброса, приложение ответственно за его возврат в занятое состояние после обработки ввода-вывода. Это можно сделать, вызвав функцию WSAResetEvent. Она принимает описатель события в

качестве единственного параметра и возвращает TRUE или FALSE, в зависимости от успешности вызова. Закончив работу с объектом события, приложение должно вызвать функцию WSACloseEvent для освобождения системных ресурсов, используемых объектом.

Сопоставив сокет дескриптору события, приложение может начать обработку ввода-вывода, ожидая изменения состояния объекта события. Функция

```
DWORD WSAWaitForMultipleEvents(DWORD cEvents,const WSAEVENT FAR * lphEvents,BOOL fWaitAll,
        DWORD dwTimeout,BOOL fAlertable);
```

будет ожидать сетевое событие и вернет управление, когда один из заданных дескрипторов объектов событий перейдет в свободное состояние или когда истечет заданный таймаут.

Если параметр fWaitAll равен TRUE, функция завершается после освобождения всех событий, перечисленных в массиве lphEvents. Если же FALSE — функция завершится, как только будет свободен любой объект события. В последнем случае возвращаемое значение показывает, какой именно объект стал свободен. Как правило, приложения присваивают этому параметру FALSE и обрабатывают одно событие сокета за раз.

Параметр dwTimeout указывает, сколько миллисекунд функция должна ожидать сетевого события. Когда истекает таймаут, функция завершается, даже если не выполнены условия, определенные параметром fWaitAll. Если таймаут равен 0, функция проверяет состояние заданных объектов и выходит немедленно, что позволяет приложению эффективно проверить все события. Задавать нулевой таймаут не рекомендуется из соображений быстродействия. Если нет событий для обработки, функция WSAWaitForMultipleEvents возвращает значение WSA_WAIT_TIMEOUT. Если параметр divsTimeout равен WSAINF1NITE, функция закончит работу только после освобождения какого-либо события.

Последним параметром fAlertable, можно пренебречь в модели WSAEventSelect, присвоив ему FALSE: он применяется в процедурах завершения процессов в модели перекрытого ввода-вывода, которая описана далее в работе 7.

Функция WSAWaitForMultipleEvents, получив уведомление о сетевом событии, возвращает значение, определяющее, какое событие произошло. Найдя событие в массиве событий и связанный с ним сокет, приложение может определить, событие какого типа произошло на конкретном сокете. Для определения индекса события в массиве lphEvents нужно вычесть из возвращаемого значения константу WSA_WAIT_EVENT_0:

```
Index = WSAWaitForMultipleEvents(...);
MyEvent =lphEvents [Index - WSA_WAIT_EVENT_0];
```

Выяснив сокет, на котором произошло событие, определяют доступные сетевые события, вызвав функцию

```
int WSAEnumNetworkEvents(SOCKET s, WSAEVENT hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents).
```

Параметр s — сокет, на котором произошло сетевое событие. Необязательный параметр hEventObject — дескриптор связанного события, которое вы хотите сбросить. Этот параметр использовать не рекомендуется, а для перевода объекта события в занятое состояние следует использовать функцию WSAResetEvent. Последний параметр — lpNetworkEvents, принимает указатель на структуру WSANETWORKEVENTS, в которой системой передается тип произошедшего события и код ошибки. Структура WSANETWORKEVENTS определена так

```
typedef struct _WSANETWORKEVENTS{
  long lNetworkEvents;
  int  iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

Параметр lNetworkEvents определяет тип произошедшего события. Параметр iErrorCode — массив кодов ошибок, связанных с событиями из массива lNetworkEvents. Для каждого типа сетевого события существует индекс события, обозначаемый тем же именем с суффиксом BIT. Например, для типа события FD_READ идентификатор индекса в массиве iErrorCode обозначается FD_READ_BIT. Приведём анализ кода ошибки для события FD READ:

```
if (NetworkEvents.lNetworkEvents & FD_READ){
  if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)    {
```

```
    printf("FD_READ failed with error %d\n",
        NetworkEvents.iErrorCode[FD_READ_BIT]);
  }
}
```

Отметим, что интенсивно используемый сервер может одновременно получить сообщения FD_READ и FD_WRITE.

После обработки событий, описанных в структуре WSANETWORKEVENTS, приложение может продолжить ожидание сетевых событий на доступных сокетах.

Практическая часть

Рассмотрим вначале клиентскую программу. Код функции main и внешние переменные приведены в листинге 1. Логика работы аналогична главной функции клиента WSAAsyncSelec из листинга 9 работы 4. Мы вводим новую глобальную перекменную – объект события hEvent.

```
#include <winsock2.h>
#include <stdio.h>
#include "../../common/DATA_BUFSIZE.h"
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
DWORD WINAPI ClientThread(LPVOID lpParam);
void forFD_WRITE(WPARAM wParam);
void forFD_READ(WPARAM wParam);
WSAEVENT hEvent;
int needNewRnd=1;
int *msgForR, *msgForW;
WSABUF DataBuf;
char *bp;
SOCKET Socket;
DWORD leftToRead=0;
DWORD leftToWrite =DATA_BUFSIZE;
int main(int argc, char **argv){
  WSADATA          wsaData;
  SOCKET           s;
  int              Port = 5150;
  struct sockaddr_in ServerAddr;
  int              Ret;
  HANDLE       hThread;
  DWORD        dwThreadId;
  if (argc <= 1)   {
    //myprintf("Использование: AsyncEventClient <Server IP address>.\n");
    return;
  }
WSAStartup(MAKEWORD(2,2), &wsaData);
s = Socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  ServerAddr.sin_family = AF_INET;
  ServerAddr.sin_port = htons(Port);
  ServerAddr.sin_addr.s_addr = inet_addr(argv[1]);
  myprintf("Попытка соединения к %s:%d...\n",inet_ntoa(ServerAddr.sin_addr), htons(ServerAddr.sin_port));
Connect(s, (struct sockaddr*) &ServerAddr, sizeof(ServerAddr);
msgForW = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
msgForR = (int*)GlobalAlloc(GPTR,DATA_BUFSIZE);
hThread = CreateThread(NULL, 0, ClientThread,(LPVOID)s, 0, &dwThreadId);
myprintf("Обмениваемся данными с Эхо сервером .\n");
myprintf("Нажмите Enter для завершения.\n");
getchar();
closesocket(s);
}
```

Листинг 1. Код функции main клиента WSAEventSelect

Главные действия разворачиваются в функции ClientThread, являющейся носителем новой нити. Код приведен в листинге 2. С помощью функции WSAEventSelect сетевые события FD_READ, FD_WRITE и FD_CLOSE на сокете s связываются с созданным объектом события hEvent. Далее все действия происходят в бесконечном цикле. Вначале нить с помощью функции WSAWaitForMultipleEvents ожидает сетевые события на объекте события hEvent. Далее функция WSAEnumNetworkEvents помещает инфомацию о случившемся событии в переменную NetworkEvents. Анализируя эту переменную, определяем, готов ли сокет для записи или чтения. Если это так, то вызываем нашу функцию forFD_WRITE (см. листинг 3 ниже) для записи в сокет.

```
DWORD WINAPI ClientThread(LPVOID lpParam){
  WSAEVENT EventArray[1];
  SOCKET   s    =(SOCKET)lpParam;
  WSANETWORKEVENTS NetworkEvents;
  if ((hEvent = EventArray[0]=WSACreateEvent()) == WSA_INVALID_EVENT) {
    printf("WSACreateEvent() failed with error %d\n", WSAGetLastError());
    return FALSE;
  }
if (WSAEventSelect(s, hEvent, FD_READ|FD_WRITE) == SOCKET_ERROR)   {
                printf("WSAEventSelect() failed with error %d\n", WSAGetLastError());
               return 0;
}
while(TRUE)   {
    // Ждём сетевых событий на сокете
if ((WSAWaitForMultipleEvents(1, EventArray, FALSE,
      WSA_INFINITE, FALSE)) == WSA_WAIT_FAILED) {
      printf("WSAWaitForMultipleEvents failed with error %d\n", WSAGetLastError());
      break;
}
if (WSAEnumNetworkEvents(s, hEvent, &NetworkEvents) == SOCKET_ERROR)      {
      printf("WSAEnumNetworkEvents failed with error %d\n", WSAGetLastError());
      break;
}
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]);
               continue;
    }
    if (NetworkEvents.lNetworkEvents&FD_WRITE &&NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0){
               printf("FD_WRITE failed with error %d\n", NetworkEvents.iErrorCode[FD_WRITE_BIT]);
               continue;
    }
    forFD_WRITE(s);
}
}//while
return -1;
}
```

Листинг 2. Код функции ClientThread клиента WSAEventSelect.

Логика работы функции forFD_WRITE для записи в сокет в целом аналогична логике работы одноимённой функции из листинга 12 работы 4. Главное отличие состоит в том, что в теле функции в листинге 3 присутствуют явные вызовы функции forFD_READ для чтения из сокета, а в теле функции из листинга 12 работы 4 такие вызовы отсутствуют. Это объясняется различиями в механизмах регенерации сетевых событий FD_READ и |FD_WRITE для моделей асинхронного ввода-вывода WSAAsyncSelect (работа 4) и WSAEventSelect (эта работа).

```
void forFD_WRITE(WPARAM wParam){
    DWORD Flags, SendBytes;
    int i;
    if(needNewRnd){
               //Моделирования данных передаваемых на сервер для обработки
               for(i=0;i<DATA_BUFSIZE_INT;i++) msgForW[i]=rand();
               bpW=(char*)msgForW;
```

```
                needNewRnd=0;
        }
        if(leftToWrite>0){
                DataBuf.buf = bp;
                DataBuf.len = leftToWrite;
                Flags = 0;
                if (WSASend((SOCKET)wParam, &DataBuf, 1, &SendBytes, 0,        NULL, NULL) ==
                        SOCKET_ERROR){
                        if (WSAGetLastError() != WSAEWOULDBLOCK){
                                myprintf("WSASend() failed with error %d\n", WSAGetLastError());
                                closesocket(wParam);
                                return;
                        }
                }
                else{
                        bp += SendBytes;
                        leftToWrite-= SendBytes;
                        if(leftToWrite==0){
                                leftToRead=DATA_BUFSIZE;
                                bp = (char*)msgForR;
                                forFD_READ((SOCKET)wParam);
                        }
                }
        }
        else
                forFD_READ((SOCKET)wParam);

}
```

Листинг 3. Функция клтента forFD_WRITE.

Код функции forFD_READ для чтения из сокета приведен в листинге 4. Логика работы функции в целом аналогична логике работы одноимённой функции из листинга 13 работы 4. Небольшие отличия объясняются особенностями регенерации сетевых событий FD_READ и FD_WRITE для моделей асинхронного ввода-вывода WSAAsyncSelect (работа 4) и WSAEventSelect (эта работа).

```
void forFD_READ(WPARAM wParam){
        DWORD RecvBytes, Flags = 0;
        if(leftToRead>0){
                DataBuf.buf = bpR;
                DataBuf.len = leftToRead;
                if (WSARecv((SOCKET)wParam, &DataBuf, 1, &RecvBytes, &Flags, NULL, NULL) ==
                                SOCKET_ERROR){
                        if (WSAGetLastError() != WSAEWOULDBLOCK){
                                myprintf("Recv() failed with error %d\n", WSAGetLastError());
                                return;
                        }
                }
                else {
                        bpR += RecvBytes;
                        leftToRead -= RecvBytes;
                        if(leftToRead==0){
                                //Исходящее должно совпадать с входящим
                                if(memcmp(msgForR,msgForW,DATA_BUFSIZE))
                                        puts("ERROR\n");
                                leftToWrite=DATA_BUFSIZE;
                                bp=(char*)msgForW;
                                needNewRnd=1;
                                forFD_WRITE(wParam);
                        }
                }
        }
}
```

Листинг 4. Код функции клиента forFD_READ для чтения из сокета.

Перейдём к эхо-серверу модели AsyncEvent. Код функции main и внешние переменные приведены в листинге 5. По сравнению с асинхронный эхо-сервер модели WSAASyncselect (листинг 1 работы 4) мы видим изменения в способе хранения информации о сокетах. В работе 4 используется список, а в этой работе используется масив SocketArray элементов SOCKET_INFORMATION. Дополнительно используется массив объектов события EventArray. Текущее число соединённых сокетов хранится в переменной EventTotal. Вначале создаём слушающий сокет Listen и заполняем с помощью нашей функции CreateSocketInformation информацию для него. Используя функции WSAEventSelect, готовимся принимать от этого сокета сетевые события о подключениях FD_ACCEPT.

Далее все действия происходят в бесконечном цикле. Вначале с помощью функции WSAWaitForMultipleEvents ожидаются объекты события из массива EventArray. Функция возвращает номер Event сокета, на котором случилось событие. Далее функция WSAEnumNetworkEvents помещает информацию о случившемся событии в переменную NetworkEvents. Анализируя эту переменную определяем, была ли попытка подключения (событие FD_ACCEPT). Если так, то с помощью функции accept принимаем новое соединение в новый сокет Accept, заполняем с помощью нашей функции CreateSocketInformation информацию для него и, используя функцию WSAEventSelect, готовимся принимать от этого сокета сетевые события FD_READ и FD_WRITE. Далее анализируем переменную NetworkEvents и определяем, готов ли сокет для записи или чтения (события FD_ READ и FD_ WRITE). Если это так, то вызываем нашу функцию чтения из сокета forFD_ READ.

```
#include <winsock2.h>
int myprintf (const char *lpFormat, ... );
#include "../../common/DATA_BUFSIZE.h"
#define PORT 5150
typedef struct _SOCKET_INFORMATION {
      WSABUF DataBuf;
      CHAR Buffer[DATA_BUFSIZE];
      char *bp;
      SOCKET Socket;
      DWORD leftToRead;
      DWORD leftToWrite;
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;
BOOL CreateSocketInformation(SOCKET s);
void FreeSocketInformation(DWORD Event);
void forFD_READ(DWORD Event);
void forFD_WRITE(DWORD Event);

DWORD EventTotal = 0;
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];

void main(void){
  SOCKET Listen;
  SOCKET Accept;
    struct sockaddr_in InternetAddr;
  DWORD Event;
  WSANETWORKEVENTS NetworkEvents;
  WSADATA wsaData;
  DWORD Ret;
WSAStartup(0x0202, &wsaData);
Listen = Socket (PF_INET, SOCK_STREAM, 0);
CreateSocketInformation(Listen);
 if (WSAEventSelect(Listen, EventArray[EventTotal - 1], FD_ACCEPT) == SOCKET_ERROR)   {
     myprintf("WSAEventSelect() failed with error %d\n", WSAGetLastError());
     return;
  }
  InternetAddr.sin_family = AF_INET;
  InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  InternetAddr.sin_port = htons(PORT);
  Bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr));
```

```
  Listen(Listen, 5);
 while(TRUE) {
    // Ждём сетевых событий на сокетах
    if ((Event = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE,
      WSA_INFINITE, FALSE)) == WSA_WAIT_FAILED) {
      myprintf("WSAWaitForMultipleEvents failed with error %d\n", WSAGetLastError());
      continue;
    }
    if (WSAEnumNetworkEvents(SocketArray[Event - WSA_WAIT_EVENT_0]->Socket, EventArray[Event -
      WSA_WAIT_EVENT_0], &NetworkEvents) == SOCKET_ERROR)     {
      myprintf("WSAEnumNetworkEvents failed with error %d\n", WSAGetLastError());
      continue;
    }
    if (NetworkEvents.lNetworkEvents & FD_ACCEPT) {
            if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0) {
                    myprintf("FD_ACCEPT failed with error %d\n",
                            NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
                    continue;
            }
            if ((Accept = accept(SocketArray[Event - WSA_WAIT_EVENT_0]->Socket, NULL, NULL)) ==
                            INVALID_SOCKET){
                    myprintf("accept() failed with error %d\n", WSAGetLastError());
                    continue;
            }
            if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS){
                     myprintf("Слишком много соединений.\n");
                    closesocket(Accept);
                    continue;
            }
            CreateSocketInformation(Accept);
            if (WSAEventSelect(Accept, EventArray[EventTotal - 1], FD_READ|FD_WRITE) ==
                            SOCKET_ERROR){
            myprintf("WSAEventSelect() failed with error %d\n", WSAGetLastError());
            continue;
            }
             myprintf("Socket %d connected\n", Accept);
    }
    if (NetworkEvents.lNetworkEvents & FD_READ ||NetworkEvents.lNetworkEvents & FD_WRITE){
            if (NetworkEvents.lNetworkEvents & FD_READ
                            && lNetworkEvents.iErrorCode[FD_READ_BIT] != 0){
                    myprintf("FD_READ failed with error %d\n",
                            NetworkEvents.iErrorCode[FD_READ_BIT]);
                    continue;
            }
            if (NetworkEvents.lNetworkEvents & FD_WRITE
                            &&NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0){
                    myprintf("FD_WRITE failed with error %d\n",
                            NetworkEvents.iErrorCode[FD_WRITE_BIT]);
                    continue;
            }
            forFD_READ(Event);
      }
  }//while
}
```

Листинг 5. Код функции main эхо-сервера модели AsyncEvent.

Функция CreateSocketInformation создаёт информацию для сокета. Её код приведен в листинге 6.

```
BOOL CreateSocketInformation(SOCKET s){
LPSOCKET_INFORMATION SI;
if ((EventArray[EventTotal] = WSACreateEvent()) == WSA_INVALID_EVENT) {
    myprintf("WSACreateEvent() failed with error %d\n", WSAGetLastError());
    return FALSE;
```

```
  }
if ((SI = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR,
                                        sizeof(SOCKET_INFORMATION))) == NULL){
    myprintf("GlobalAlloc() failed with error %d\n", GetLastError());
    return FALSE;
  }
SI->Socket = s;
SI->leftToRead = DATA_BUFSIZE;
SI->leftToWrite = 0;
SI->bp=SI->Buffer;
SocketArray[EventTotal] = SI;
EventTotal++;
return(TRUE);
}
```

Листинг 6. Код функции CreateSocketInformation

Код функции forFD_READ для чтения из сокета приведен в листинге 7. Логика работы функции в целом аналогична логике работы одноимённой функции из листинга 7 работы 4. Небольшие отличия объясняются особенностями регенерации сетевых событий FD_READ и FD_WRITE для моделей асинхронного ввода-вывода WSAAsyncSelect (работа 4) и WSAEventSelect (эта работа), а также разными способами хранения информации о сокетах.

```
void forFD_READ(DWORD Event){
    DWORD RecvBytes;
    DWORD Flags;
    LPSOCKET_INFORMATION SocketInfo = SocketArray[Event - WSA_WAIT_EVENT_0];
    if(SocketInfo->leftToRead>0){
            SocketInfo->DataBuf.buf = SocketInfo->bp;
            SocketInfo->DataBuf.len = SocketInfo->leftToRead;
            Flags = 0;
            if (WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &RecvBytes,
                                        &Flags, NULL, NULL) == SOCKET_ERROR){
                    if (WSAGetLastError() != WSAEWOULDBLOCK){
                            myprintf("Recv() failed with error %d\n", WSAGetLastError());
                            FreeSocketInformation(Event);
                             return;
                    }
            }
            else {
                    SocketInfo->bp += RecvBytes;
                    SocketInfo->leftToRead -= RecvBytes;
                    if(SocketInfo->leftToRead==0){
                            //Можно обрабатывать полученную информацию
                            //Мы делаем просто Эхо
                            SocketInfo->leftToWrite=DATA_BUFSIZE;
                            SocketInfo->bp = SocketInfo->Buffer;
                            forFD_WRITE(Event);
                    }
            }
    }
}
```

Листинг 7. Код функции forFD_READ сервера для чтения из сокета.

Код функции forFD_WRITE для записи в сокет приведен в листинге 8. Логика работы функции в целом аналогична логике работы одноимённой функции из листинга 8 работы 4. Небольшие отличия объясняются особенностями регенерации сетевых событий FD_READ и FD_WRITE для моделей асинхронного ввода-вывода WSAAsyncSelect (работа 4) и WSAEventSelect (эта работа), а также разными способами хранения информации о сокетах.

```
void forFD_WRITE(DWORD Event){
    DWORD SendBytes;
    DWORD Flags;
```

```
    LPSOCKET_INFORMATION SocketInfo = SocketArray[Event - WSA_WAIT_EVENT_0];
    if(SocketInfo->leftToWrite>0){
            SocketInfo->DataBuf.buf = SocketInfo->bp;
            SocketInfo->DataBuf.len = SocketInfo->leftToWrite;
            Flags = 0;
            if (WSASend(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, &SendBytes, 0,
                                            NULL, NULL) == SOCKET_ERROR){
                    if (WSAGetLastError() != WSAEWOULDBLOCK){
                            myprintf("WSASend() failed with error %d\n", WSAGetLastError());
                            FreeSocketInformation(Event);
                            return;
                    }
            }
            else{
                    SocketInfo->bp += SendBytes;
                    SocketInfo->leftToWrite-= SendBytes;
                    if(SocketInfo->leftToWrite==0){
                            SocketInfo->leftToRead=DATA_BUFSIZE;
                            SocketInfo->bp = SocketInfo->Buffer;
                            forFD_READ(Event);
                    }
            }
    }
}
```

Листинг 8. Код функции forFD_ WRITE сервера для чтения из сокета.

В функциях forFD_READ и forFD_WRITE при ошибках вызывается функция FreeSocketInformation, освобождающая информацию для сокета. Её код приведен в листинге 9.

```
void FreeSocketInformation(DWORD Event){
    LPSOCKET_INFORMATION SI = SocketArray[Event];
    DWORD i;
    closesocket(SI->Socket);
    GlobalFree(SI);
    WSACloseEvent(EventArray[Event]);
    for (i = Event; i < EventTotal; i++)   {
            EventArray[i] = EventArray[i + 1];
            SocketArray[i] = SocketArray[i + 1];
    }
    EventTotal--;
}
```

Листинг 9. Код функции FreeSocketInformation

Конторольные вопросы

1. Чем отличается эта модель ввода-вывода от модели WSAAsyncSelect?
2. Объясните назначение параметров функции WSAEventSelect
3. Что такое возобновляющая функция
4. Что нужно предварительно сделать в программе, чтобы запустить в ней модель ввода-вывода WSAEventSelect?
5. Перечислите и охарактеризуйте сетевые события.
6. Когда возникает сетевое событие FD_READ и к чему это приведёт в модели WSAEventSelect?
7. Когда возникает сетевое событие FD_WRITE и к чему это приведёт в модели WSAEventSelect?
8. Что такое объект события и как с ним работать?
9. Как работает функция WSAWaitForMultipleEvents?
10. Объясните назначение параметров функции WSAWaitForMultipleEvents
11. С помощью какой функции и как можно выяснить какие сетевые события произошли на сокете?
12. Объясните назначение полей стуктуры WSANETWORKEVENTS
13. Объясните логику работы функции ClientThread клиента WSAEventSelect (Листинг 2)
14. Найдите и объясните различия функции forFD_WRITE клиента, приведенной в листинге 3 и функции forFD_WRITE приведенной в листинге 12 работы 4.

15. Найдите и объясните различия функции forFD_ READ клиента, приведенной в листинге 3 и функции forFD_WRITE приведенной в листинге 13 работы 4.
16. Объясните логику работы функции main эхо-сервера модели AsyncEvent (Листинг 5).
17. Найдите и объясните различия функции forFD_WRITE сервера, приведенной в листинге 8 и функции forFD_WRITE приведенной в листинге 8 работы 4.
18. Найдите и объясните различия функции forFD_ READ клиента, приведенной в листинге 7 и функции forFD_WRITE приведенной в листинге 7 работы 4.

Задания

8. Загрузите в Visual Studio 2008 проект сервера AsyncEvent (листинги 5 – 9) и клиента AsyncEventClient (листинги 1 – 4) из решения lab6 архива src. Произведите отладку проектов. Рекомендация: и клиент и сервер должен иметь одинаковый размер буферов DATA_BUFSIZE, поэтому сделайте для их обоих общий заголовочный файл DATA_BUFSIZE.h.
9. Используя подход, изложенный в работе №2, дополните проекты клиента и сервера кодом, позволяющим выводить в окно консоли текущую и среднюю скорость обмена информацией.
10. Используя подход, изложенный в работе №5, дополните проекты клиента и сервера кодом, обеспечивающим корректное завершение работы. Указание: не используйте функцию WSAAsyncSelect для установки реакции на сетевое событие FD_CLOSE.
11. Сравните текущую и среднюю скорость обмена информацией и объясните результат для соединений:
    a. Клиента AsyncEventClient из этой работы к серверу AsyncEvent из этой работы
    b. Клиента ClientF из работы 2 к серверу AsyncEvent из этой работы
    c. Клиента GracefulShutdownAsyncSelectClient из работы 5 к серверу AsyncEvent из этой работы
    d. Клиента AsyncEventClient из этой работы к серверу GracefulShutdownAsyncSelect из работы 5
    e. Клиента AsyncEventClient из этой работы к серверу tcpTreadServerF из работы 2.
    f. Клиента AsyncEventClient из этой работы к серверу tcpSelectServerF из работы 2.
12. Модифицируйте программы сервера и клиента WSAEventSelect для работы с пакетами переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом.

Лабораторная работа №7. Асинхронные сокеты. Перекрытия и функции завершения

Теоретическая часть

С помощью модели перекрытого ввода-вывода приложение может асинхронно, без блокировки выдать несколько запросов на запись или чтение, а затем обслужить эти запросы после их завершения. Модель перекрытого ввода-вывода встроена в новые функции Winsock, такие как WSASend и WSARecv. Winsock позволяет использовать перекрытый сетевой ввод-вывод при помощи стандартных функций Win32 API ReadFile и WriteFile для чтения и записи файлов.

Для использования модели перекрытого ввода-вывода следует создать сокет с флагом WSA_FLAG_OVERLAPPED с помощью функция WSASocket . Функция socket задается флаг WSA_FLAG_VERLAPPED неявно.

Функция WSASocket создаёт сокет, котрый привязывается к провайдеру службы сетевого транспорта. Прототип определён в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

```
SOCKET WSASocket(int af, int type, int protocol,
                 LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g,  DWORD dwFlags);
```

Параметр af задаёт семейство адресов, параметр type задаёт тип сокета, а параметр protocol задаёт протокол. Значения этих параметры приведены в работе №1 при описании функции socket. Указатель lpProtocolInfo на структуру WSAPROTOCOL_INFO определяет характеристики нового сокета. Если он не равен NULL, то сокет будет привязан к провайдеру службы сетевого транспорта, определённому в структуре WSAPROTOCOL_INFO. Аргумент g зарезервирован. Флаг dwFlags определбяет атрибуты сокета путём использования побитового ИЛИ флагов. Нас будет интересовать только один флаг WSA_FLAG_OVERLAPPED. Этот флаг приводит к созданию сокета с перекрытием. Такие сокеты могут использовать вызовы функций WSASend, WSASendTo, WSARecv, WSARecvFrom и WSAIoctl для операций перекрывающегося ввода-вывода, которые могут начинаться и продолжаться одновременно и параллельно на одном и том же сокете.

В случае успеха функция возвратит дескриптор нового сокета. Иначе функция возвращает SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSANOTINITIALISED, WSAENETDOWN, WSAEAFNOSUPPORT, WSAEINPROGRESS, WSAEMFILE, WSAENOBUFS, WSAEPROTONOSUPPORT, WSAEPROTOTYPE, WSAESOCKTNOSUPPORT, WSAEINVAL, WSAEFAULT, WSAINVALIDPROVIDER, WSAINVALIDPROCTABLE. Расшифровку ошибок смотри в приложении или в MSDN.

Функция WSASocket выдаёт дескриптор сокета. При этом выделяются ресурсы, которые ассоциируются с провайдером службы сетевого транспорта. По умолчанию сокет не имеет атрибута перекрытия.

Все функции, разрешающие операции с перекрытием (WSASend, WSARecv и др.) также поддерживают их использование без перекрытия на сокете с перекрытием, если значение параметров, относящихся к операциям с перекрытием равно NULL.

Сокеты, ориентированные на соединение, такие как SOCK_STREAM обеспечивают полнодуплексную связь и должны быть в состоянии соединения до того, как данные могут быть переданы или приняты через них. Соединение к другому сокету осуществляется с помощью функций connect/WSAConnect. Для приёма запросов на соединение используется accept/WSAAccept/AcceptEX. После соединения данные могут быть переданы и приняты функциями send/WSASend и recv/WSARecv calls. После завершения сессии следует вызвать closesocket.

Коммуникационные протоколы, ориентированные на соединение гарантируют, что данные будут доставлены без потерь, ошибок и дублирования. Если локальная и удалённая сторона не может этого обеспечить после определённого числа попыток в течение разумного интервала времени, то соединение рассматривается как разорванное и все последующие вызовы потерпят неудачу с кодом ошибки WSAETIMEDOUT.

Перекрытый ввод-вывод можно использовать, вызвав функции WSASend, WSARecv и другие, используя в них структуру WSAOVERLAPPED. Тогда эти функции завершаются немедленно, даже если сокет работает в блокирующем режиме. Структура WSAOVERLAPPED используется для завершения обслуживания ввода-вывода. Есть два способа завершения запросов перекрытого ввода-вывода: приложение может ожидать уведомления от объекта событие или обрабатывать завершившиеся запросы процедурами завершения. У функций WSASend и WSARecv есть параметр - указатель на процедуру завершения, которая вызывается при завершении запроса перекрытого ввода-вывода.

Функция WSASend передаёт данные на соединенном сокете.

```
int WSASend(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent, DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

Функция WSARecv принимает данные на соединенном сокете.

```
int WSARecv (SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecv, LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

Прототипы функций определёны в файле Winsock2.h. Библиотека импорта Ws2_32.lib. Код расположен в Ws2_32.dll.

Сокет задаётся параметрои s. lpBuffers указывает на массив структур WSABUF. Каждая такая структура содержит указатель на буфер buf и его длину len в байтах. Параметр dwBufferCount содержит число структур в массиве lpBuffers. Если операция вывода (ввода) осуществится немедленно, то параметр lpNumberOfBytesSent (lpNumberOfBytesRecv) указывает на число, в котором функция вернёт число переданных (принятых) байт. Если указатель lpOverlapped на структуру WSAOVERLAPPED не равен NULL, то этот параметр необязателен и может быть установлен в NULL. Когда операция завершится, то производится вызов функции завершения, указатель на которую помещён в lpCompletionRoutine. Для неперекрывающихся сокетов последние два параметра игнорируется .

Флаг dwFlags (lpFlags) позволяет модифицировать поведение функции. Они здесь подробно не рассматриваются. Для функции WSARecv он является и входным и выходным.

Если не произошла ошибка, то функция возвращает ноль, что говорит в частности о том, что функция lpCompletionRoutine запланирована на выполнение. Иначе функция возвращает SOCKET_ERROR и код ошибки может быть получен с помощью функции WSAGetLastError. Возможные коды ошибок: WSAECONNABORTED, WSAECONNRESET, WSAEFAULT, WSAEINTR, WSAEINPROGRESS, WSAEINVAL, WSAEMSGSIZE, WSAENETDOWN, WSAENETRESET, WSAENOBUFS, WSAENOTCONN, WSAENOTSOCK, WSAEOPNOTSUPP, WSAESHUTDOWN, WSAEWOULDBLOCK, WSANOTINITIALISED, WSA_IO_PENDING, WSA_OPERATION_ABORTED. Расшифровку ошибок смотри в приложении к или в MSDN.

Код ошибки WSA_IO_PENDING говорит о том, что операция успешно инициализирована, но ещё не завершена.

Функции имеют большую функциональность по сравнению со стандартными функциями send и recv: использоваться для выполнения перекрывающихся операций и одновременно передавать (принимать) информацию из нескольких буферов.

Функции могут быть использованы и на сокетах не ориентированных на соединение. Для WSASend сокет должен иметь оговоренный адрес удалённого узла, установленный в функции connect или WSAConnect. Для WSARecv локальный сокет должен быть связан.

Для перекрывающихся сокетов (созданных с помощью WSASocket с флагом SA_FLAG_OVERLAPPED и ненулевыми параметрами lpOverlapped и/или lpCompletionRoutine) обмен информации происходит в режиме перекрытия. Когда данные для передачи (приёма) будут восприняты транспортным провайдером (и в случае приёма, помещены в пользовательский буфер), то начнёт выполняться функция завершения или будет установлен объект событие. Если операция не закончится немедленно, то её результаты можно получить или в функции завершения или путём вызова функции WSAGetOverlappedResult, дождавшись установки объекта события.

Если оба параметра lpOverlapped и lpCompletionRoutine равны NULL, то сокет рассматривается как не имеющий перекрытия и вызов функций ведёт себя также как и блокирующий вызов функции send (recv).

Для функции WSASend данные копируются из буферов функции в буфер транспортного провайдера. Если сокет неблокированный, ориентированный на поток и нет достаточно места в транспортном буфере, то функция WSASend завершится, передав транспортному провайдеру часть буфера приложения. Для блокирующего сокета функция WSASend будет ждать пока все данные из приложения не будут переданы транспортному провайдеру. Успешное выполнение функции WSASend не означает, что данные будут доставлены.

Для блокирующих сокетов с помощью функции setsockopt можно установить таймауты ожидания SO_SNDTIMEO и SO_RCVTIMEO для передачи и приёма.

Массив lpBuffers структур WSABUF может быть временным. При выполнении операции с перекрытием транспортный провайдер захватывает эти структуры до того, как завершится вызов. Это позволяет приложению размещать массив lpBuffers локальных переменных (в стеке).

Для сокетов, ориентированных на сообщение, нельзя превышать максимальную длину датаграммы транспортного провайдера. Эта длина может быть получена с помощью вызова функции getsockopt с параметром SO_MAX_MSG_SIZE. Если данных слишком много, то возникнет ошибка WSAEMSGSIZE и данные переданы не будут. Если сообщение превысит длину входного буфера функции WSARecv, то его оставшаяся часть будет отброшена. Будет она потеряна или нет, зависит от транспортного провайдера.

Если операция с перекрытием завершится немедленно, то функции возвратят 0 и параметр lpNumberOfBytesSent (lpNumberOfBytesRecv) укажут на число переданных байт. Если операция с

перекрытием успешно стартовала, но завершится позже, то WSASend returns возвратит SOCKET_ERROR с кодом ошибки WSA_IO_PENDING. В этом случае число переданных байт не установится. Когда операция с перекрытием завершится, то число переданных байт можно получить или через параметр cbTransferred функции завершения или через параметр lpcbTransfer функции WSAGetOverlappedResult. При успешном приёме информации с помощью функции WSARecv нулевое число принятых бай означает, что удалённая сторона закрыла соединение.

Все операции ввода-вывода, начатые в нити, будут прекращены при завершении нити. Для сокетов с перекрытием отложенные асинхронные операции могут потерпеть крах, если нить, которая их инициировала завершится до завершения операций.

Параметр lpOverlapped, указывающий на структуру перекрытия должен быть действителен на всё время операции с перекрытием. Если одновременно выполняются операции с перекрытием от двух и более вызовов функции, то каждая операция должна иметь свою структуру перекрытия.

Если параметр lpCompletionRoutine равен NULL, то при завершении операции устанавливается объект события hEvent структуры lpOverlapped (объект события должен быть создан и помещён в структуру lpOverlapped до первого вызова функции). Приложение может использовать функции WSAWaitForMultipleEvents и WSAGetOverlappedResult для ожидания объекта события и получения результата.

Если указатель на функцию завершения lpCompletionRoutine не равен NULL, то параметр hEvent игнорируется, но может быть использован приложением для передачи контекстной информации для функции завершения.

Как уже отмечалось, есть два способа завершения запросов перекрытого ввода-вывода: приложение может ожидать уведомления от объекта событие или обрабатывать завершившиеся запросы процедурами завершения.

Сначала рассмотрим способ уведомления о событиях, а затем — использование процедур завершения.

События

Для использования событий в модели перекрытого ввода-вывода необходимо сопоставить объекты события со структурами WSAOVERLAPPED. Вызовы функции WSASend и WSARecv в режиме перекрытия завершаются немедленно и, как правило возвращают ошибку SOCKET_ERROR. Функция WSAGetLastError возвращает значение WSA_IO_PENDING. Это означает, что операция ввода-вывода продолжается. О завершении перекрытого ввода-вывода сообщает событие, связанное со структурой WSAOVERLAPPED. Структура WSAOVERLAPPED осуществляет связь между началом запроса ввода-вывода и его завершением. Поле hEvent структуры WSAOVERLAPPED позволяет приложению связать описатель объекта событие с сокетом. Этот описатель создают с помощью функцию WSACreateEvent. После создания описателя события достаточно присвоить его значение полю hEvent структуры WSAOVERLAPPED, после чего можно вызывать функции Winsock, использующие структуры перекрытой модели, такие как WSASend или WSARecv.

При завершении перекрытого ввода-вывода Winsock устанавливает объект событие, связанный со структурой WSAOVERLAPPED. Завершение запроса перекрытого ввода-вывода легко определить, вызвав функцию WSAWaitForMultipleEvents для ожидания объекта события, указанного в структуре WSAOVERLAPPED. Функция WSAWaitForMultipleEvents возвратит номер наступившего события, что позволит определить, какая операция ввода-вывода завершилась. Дождавшись события, следует извлечь результаты ввода вывода функцией

```
BOOL WSAGetOverlappedResult(SOCKET s,LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer, BOOL fWait, LPDWORD lpdwFlags);
```

Параметры s — сокет, и lpOverlapped — указатель на структуру WSAOVERLAPPED, переданный в запросе перекрытого ввода-вывода. Параметр lpcbTransfer - указатель на переменную типа DWORD, куда записывается количество байт, фактически перемещенных операцией перекрытого ввода-вывода. Параметр fWait определяет, должна ли функция ждать завершения операции. Так как в нашем случае мы уже дождались завершения операции ввода-вывода, этот параметр не важен. Последний параметр — lpdwFlags, указатель на DWORD, куда будут записаны результирующие флаги, если исходный перекрытый вызов осуществлялся функцией WSARecv или WSARecvFrom. Если функция WSAGetOverlappedResult завершилась успешно, она возвращает TRUE. Это означает, что запрос перекрытого ввода-вывода успешен и значение, на которое ссылается lpcbTransfer обновлено. Иначе функция возвращает FALSE. В случае неудачи значение по указателю lpcbTransfer не обновляется и приложение должно вызвать функцию WSAGetLastError, чтобы определить причины неудачи.

Функции завершения

Имена этих функции передаются функциям для перекрытого ввода-вывода WSARecv и WSASend как параметры и эти функции вызываются системой по завершении ввода-вывода. Их основная роль —

обслуживать завершенный запрос в нити, откуда они были вызваны. Приложение может продолжать обработку перекрытого ввода-вывода в процедуре завершения.

Функции WSARecv или WSASend могут быть вызваны внутри функций завершения ранее вызванных функций WSARecv или WSASend. Транспортный провайдер позволяет приложениям вызывать операции передачи и приёма внутри функции завершения и гарантирует, что они не будут вложенными. Функции завершения могут быть вызваны в любом порядке, а не в том порядке как завершаются операции с перекрытием. Однако гарантируется, что буфер будет отправлен (принят) в заданном порядке.

Возвращение из этой функции разрешает продолжаться другой незаконченной функции завершения для этого сокета.

Функции завершения имеет прототип:

void CALLBACK CompletionROUTINE(DWORD dwError, DWORD cbTransferred, LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags);

Когда завершается запрос перекрытого ввода-вывода, параметры функции завершения устанавливаются системой следующим образом

dwError — статус завершения для перекрытой операции, на которую указывает lpOverlapped;
cbTransferred — количество байт, перемещенных при этой операции;
lpOverlapped — структура WSAOVERLAPPED, переданная в исходный вызов функции ввода-вывода;
dwFlags — не используется и равен 0.

Основное отличие перекрытого запроса с функцией завершения от запроса с объектом события — поле hEvent структуры WSAOVERLAPPED не используется, то есть нельзя связать объект события с запросом ввода-вывода. Сделав перекрытый запрос с процедурой завершения, вызывающая нить должна обязательно выполнить процедуру завершения по окончании запроса. Для этого нужно перевести нить в состояние ожидания (alertable wait state) и дать выполниться процедуре завершения по окончании операции ввода-вывода. Если вызывающая нить всегда занята и не находится в состоянии ожидания, процедура завершения никогда не будет вызвана.

Для перевода нити в состояние ожидания используйте функцию WSAWaitForMultipleEvents, создав фиктивный, ни с чем не связанный объект события. Функция WSAWaitForMultipleEvents будет переводить нить в состояние ожидания и позволит сисиеме вызывать процедуру завершения ввода-вывода, если её последний параметр fAlertable равен TRUE. Когда запрос ввода-вывода заканчивается через процедуру завершения, функция WSAWaitForMultipleEvents возвратит WSA_IO_COMPLETION, а не индекс в массиве событий, как при обычном ожидании

Для перевода нити в состояние ожидания можно задействовать функцию SleepEx. Она ведет себя так же, как и функция WSAWaitForMultipleEvents и ей не нужны объекты события:

DWORD SleepEx(DWORD dwMilliseconds, BOOL bAlertable)

Параметр dwMilliseconds определяет, сколько миллисекунд функция SleepEx будет ждать. Если он равен INFINITE, SleepEx будет ждать бесконечно. Параметр bAlertable задает, как будет выполняться процедура завершения. Если он равен FALSE и происходит обратный вызов по окончании ввода-вывода, процедура завершения не выполняется и операция не завершится, пока не истечет период ожидания, заданный в dwMilliseconds. Если bAlertable равен TRUE, выполняется процедура завершения и функция SleepEx возвращает WAIT_IO_COMPLETION.

Практическая часть

События

Напишем программу для эхо-сервера пакетов постоянного размера с использованием перекрытий и событий. Заголовочный файл приведен в листинге 2.

Используются функции обвёртки. Все обвёртки лишь выводят диагностическое сообщение о результате вызова соответствующей функции. Обвертки _WSASend и _WSARecv для для функций _WSASend и _WSARecv приведены в листинге 1. Функция GlobalAlloc при ошибке возвращает NULL и код ошибки выдаёт функция GetLastError. Для функций WSARecv и WSASend учтён тот факт, что ошибка ERROR_IO_PENDING ошибкой для асинхронного обмена не является.Функция WSAWaitForMultipleEvents при ошибке возвращает WSA_WAIT_FAILED. Используется функция myprintf для вывода кириллицы на консоль и функция encodeWSAGetLastError для декодирования кода ошибки в текст ошибки, которые рассмотрены в работе 1.

```
HGLOBAL WINAPI _GlobalAlloc(  UINT uFlags,  SIZE_T dwBytes){
    HGLOBAL h=GlobalAlloc(uFlags,  dwBytes);
    If(!h){
            myprintf("Ошибка GlobalAlloc %s\n", encodeWSAGetLastError(GetLastError()));
            return NULL;
```

```
        }
        Return h;
}

int _WSARecv (SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
        PDWORD lpNumberOfBytesRecv, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine){
                if(WSARecv (s, lpBuffers, dwBufferCount,lpNumberOfBytesRecv,  lpFlags, lpOverlapped,
                                lpCompletionRoutine)== SOCKET_ERROR){
                        if (WSAGetLastError() != ERROR_IO_PENDING){
                                myprintf("Ошибка WSARecv %s\n",
                                        encodeWSAGetLastError(WSAGetLastError()));
                                return;
                        }
                }
}

int _WSASend (SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
        PDWORD lpNumberOfBytesSend, DWORD Flags, LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine){
                if(WSASend (s, lpBuffers, dwBufferCount,lpNumberOfBytesSEnd,  Flags, lpOverlapped,
                                lpCompletionRoutine)== SOCKET_ERROR){
                        if (WSAGetLastError() != ERROR_IO_PENDING){
                                myprintf("Ошибка WSASend %s\n",
                                        encodeWSAGetLastError(WSAGetLastError()));
                                return;
                        }
                }
}

DWORD _WSAWaitForMultipleEvents(DWORD cEvents,const WSAEVENT* lphEvents,
                BOOL fWaitAll,DWORD dwTimeout,BOOL fAlertable){
        DWORD I;
        I= WSAWaitForMultipleEvents(cEvents, lphEvents, fWaitAll, dwTimeout, fAlertable);
        If(I== WSA_WAIT_FAILED)      {
                myprintf("ошибка WSAWaitForMultipleEvents %s\n",
                        encodeWSAGetLastError (WSAGetLastError()));
                return 0;
        }
}
```

Листинг 1. Обвёртки некоторых функций Winsock.

В листинге 2 константы WRITE и READ задают направление (direction) передачи информации. Информация о каждом открытом соединении хранится в массиве SocketArray элементами в виде структуры SOCKET_INFORMATION. Эта структура была описана в предыдущих работах. В неё добавлено поле Overlapped для хранения структуры WSAOVERLAPPED и поле direction для указания направление передачи информации. Общее число соединений равно EventTotal. С каждым сокетом ассоциируется событие, которое хранится в массиве EventArray.

```
#include <winsock2.h>
#include <windows.h>
#include <stdio.h>
#include "../DATA_BUFSIZE.h"
#define PORT 5150
#define WRITE 7
#define READ -7
typedef struct _SOCKET_INFORMATION {
        WSABUF DataBuf; // Для функций WSARecv и WSASend
        CHAR Buffer[DATA_BUFSIZE];//Буфер для обмена с сетью
        WSAOVERLAPPED Overlapped;
        int direction;// WRITE, READ – направление передачи
        char *bp; // Указатель записи-чтения в буфере Buffer
        SOCKET Socket;
        DWORD leftToRead;// Осталось прочитать
```

```
    DWORD leftToWrite;// Осталось записать
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;

DWORD WINAPI ServerWorkerThread(LPVOID param);
int myprintf (const char *lpFormat, ... );
char* encodeWSAGetLastError(int n);
int _WSAStartup(WORD wVersionRequested,LPWSADATA lpWSAData);
SOCKET _WSASocket(int af, int type, int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g,  DWORD dwFlags);
int Bind(SOCKET s,const struct sockaddr* name,int namelen);
int Listen(SOCKET s,int backlog);
SOCKET Accept(SOCKET s, struct sockaddr* addr,int* addrlen);
HANDLE WINAPI _CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,    LPVOID lpParameter,DWORD dwCreationFlags,
    LPDWORD lpThreadId);
WSAEVENT _WSACreateEvent(void);
HGLOBAL WINAPI _GlobalAlloc(  UINT uFlags,  SIZE_T dwBytes);
int _WSASend(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent, DWORD dwFlags, LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
int _WSARecv (SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecv, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
DWORD _WSAWaitForMultipleEvents(DWORD cEvents,const WSAEVENT* lphEvents,
     BOOL fWaitAll,DWORD dwTimeout,BOOL fAlertable)

extern DWORD EventTotal;
extern WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
extern LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
extern CRITICAL_SECTION CriticalSection;
```

Листинг 2. Заголовочный файл overlap.h сервера с перекрытием и использованием событий

Код главной функции сервера можно увидеть в листинге 3. Создаётся критическая секция CriticalSection, используемая далее.

Остановимся подробнее на критических секциях или, если точнее с критических секций кода. Каждая нить процесса может в ходе своего выполнения зайти в любую функцию приложения и выполнить там любой фрагмент кода. Может так случиться, что две и более нитей одновременно выполняют какой-то фрагмент кода. Если этот фрагмент кода содержит модификацию общих переменных, то одновременное нахождение в нём нескольких процессов нежелательна. Если программист определился с такими фрагментами, то он объявляет их как критические секции. Только одна нить может находиться внутри критической секции. Другие нити становятся в очередь. Все изменения общих переменных в критических секциях будут корректными.

С точки зрения программиста критическая секция это просто переменная типа CRITICAL_SECTION. Критическую секцию пред использованием следует инициализировать функцией InitializeCriticalSectionAndSpinCount. Начало критической секции объявляется путём вызова функции EnterCriticalSection, а конец – LeaveCriticalSection. Например

```
CRITICAL_SECTION cs;
if (!InitializeCriticalSectionAndSpinCount(&cs, 0x80000400) )  return;
…
EnterCriticalSection(&cs);
Критическая секция кода
LeaveCriticalSection(&cs);
```

Здесь число 0x80000400 определяет количество попыток, которое мы даём нити (SpinCount), чтобы войти в критическую секцию, прежде чем стать в очередь. Для однопроцессорных систем игнорируется.

Вернёмся к листингу 3. С помощью функции WSASocket создаются сокет ListenSocket с перекрытием (параметр WSA_FLAG_OVERLAPPED), который слушает порт 5150. Для слушающего сокета ListenSocket создаётся событие и помещается в нулевой элемент массива EventArray. Запускается рабочая нить ServerWorkerThread и программа входит в бесконечный цикл. В цикле программа ожидает на функции accept очередного соединения. После принятия соединения резервируется память под элемент массива SocketArray и в него заносится информация для этого соединения. Далее на этом соединении с помощью функции WSARecv инициируется ввод информации от клиента в режиме перекрытия. Направление передачи direction установлено в READ. Это всё происходит в режиме критической секции CriticalSection.

```
#include "overlap.h"
DWORD EventTotal = 0;
WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
LPSOCKET_INFORMATION SocketArray[WSA_MAXIMUM_WAIT_EVENTS];
CRITICAL_SECTION CriticalSection;

void main(void){
WSADATA wsaData;
SOCKET ListenSocket, AcceptSocket;
SOCKADDR_IN InternetAddr;
DWORD Flags;
INT Ret;
DWORD ThreadId;
InitializeCriticalSection(&CriticalSection);
_WSAStartup(0x0202,&wsaData);
ListenSocket = _WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(PORT);
Bind(ListenSocket, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr));
Listen(ListenSocket, 5);
EventArray[0] = _WSACreateEvent());
EventTotal = 1;
_CreateThread(NULL, 0, ServerWorkerThread, NULL ,0, &ThreadId);

while(TRUE){
    AcceptSocket = Accept(ListenSocket, NULL, NULL);
    EnterCriticalSection(&CriticalSection);
    SocketArray[EventTotal] = (LPSOCKET_INFORMATION) _GlobalAlloc(GPTR,
            sizeof(SOCKET_INFORMATION));
    SocketArray[EventTotal]->Socket = AcceptSocket;
    ZeroMemory(&(SocketArray[EventTotal]->Overlapped), sizeof(OVERLAPPED));
    SocketArray[EventTotal]->direction=READ;
    SocketArray[EventTotal]->leftToRead = DATA_BUFSIZE;
    SocketArray[EventTotal]->leftToWrite = 0;
    SocketArray[EventTotal]->DataBuf.len = DATA_BUFSIZE;
    SocketArray[EventTotal]->bp=SocketArray[EventTotal]->DataBuf.buf
    = SocketArray[EventTotal]->Buffer;
    SocketArray[EventTotal]->Overlapped.hEvent = EventArray[EventTotal]
    = _WSACreateEvent();
   Flags = 0;
    _WSARecv(SocketArray[EventTotal]->Socket,  &(SocketArray[EventTotal]->DataBuf), 1, NULL,
            &Flags, &(SocketArray[EventTotal]->Overlapped), NULL);
   EventTotal++;
   LeaveCriticalSection(&CriticalSection);
    WSASetEvent(EventArray[0]);
}
}
```

Листинг 3. Код главной функции сервера с перекрытием и использованием событий

Основную работу сервер проделывает в функции нити ServerWorkerThread, код которой приведен в лимтинге 4. Функция работает в бесконечном цикле, ожидая с помощью функции WSAWaitForMultipleEvents одного из событий в массиве EventArray. Если произошло событие связанное с прослушивающим сокетом, то оно сбрасывается и происжодит переход на конец тела цикла. Иначе событие произошло из-за завершения операции на одном из соединённых сокетов. Информация для этого сокета помещается в переменную SI. Событие для этого сокета сбрасывается. Результат завершившейся операции ввода-вывода забираем с помощью функции WSAGetOverlappedResult. Если произошла ошибка или перемещено 0 байт, то закрываем сокет, освобождаем память, закрываем событие и в цикле в режиме критической секции убираем информацию о сокете из массивов EventArray и SocketArray. Дальнейшие действия зависят от типа завершившейся операции ввода-вывода, который можно определить по полю direction структуры SI. Если это было чтение (READ), то мы передвигаем указатель записи-чтения bp в буфере SI->Buffer на число принятых байт BytesTransferred. В поле leftToRead структуры SI находится число байт, которое осталось прочитать из сокета. Мы уменьшаем это число на число принятых байт. Если

это число станет равным нулю (SI->leftToRead==0), то это означает, что пакет мы приняли и его надо отправить клиенту обратно. Готовим поля структуры SI для отправки пакета и инициируем асинхронную передачу пакета клиенту с помощью функции WSASend. Если приняты не все данные, то мы готовим поля структуры SI и инициируем асинхронный приём оставшейся части пакета от клиента с помощью функции WSARecv.

В случае, когда завершилась операция записи (WRITE), мы передвигаем указатель записи-чтения bp в буфере SI->Buffer на число переданных байт BytesTransferred. В поле leftToWrite структуры SI находится число байт, которое осталось записать в сокет. Если это число станет равным нулю (SI-> leftToWrite ==0), то это означает, что пакет мы полностью передали и надо принимать от клиента новый пакет. Готовим поля структуры SI для приёма нового пакета и инициируем асинхронный приём пакета от клиента с помощью функции WSARecv. Если приняты не все данные, то мы готовим поля структуры SI и инициируем асинхронную передачу клиенту оставшейся части пакета с помощью функции WSARecv.

```
#include "overlap.h"
DWORD WINAPI ServerWorkerThread(LPVOID param){
DWORD Index;
DWORD Flags;
LPSOCKET_INFORMATION SI;
DWORD BytesTransferred;
DWORD i;
while(TRUE)   {
    Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE,  WSA_INFINITE, FALSE);
    if ((Index - WSA_WAIT_EVENT_0) == 0)      {
            WSAResetEvent(EventArray[0]);
            continue;
  }
  SI = SocketArray[Index - WSA_WAIT_EVENT_0];
  WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
  if (WSAGetOverlappedResult(SI->Socket, &(SI->Overlapped), &BytesTransferred,
    FALSE, &Flags) == FALSE || BytesTransferred == 0)      {
    myprintf("ServerWorkerThread: Закрытие сокета %d.\n", SI->Socket);
    _closesocket(SI->Socket);
    GlobalFree(SI);
    WSACloseEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
    EnterCriticalSection(&CriticalSection);
    if ((Index - WSA_WAIT_EVENT_0) + 1 != EventTotal)
      for (i = Index - WSA_WAIT_EVENT_0; i < EventTotal; i++)          {
        EventArray[i] = EventArray[i + 1];
        SocketArray[i] = SocketArray[i + 1];
      }
    EventTotal--;
    LeaveCriticalSection(&CriticalSection);
    continue;
  }

    switch(SI->direction){
    case READ:{
            SI->bp += BytesTransferred;
            SI->leftToRead-= BytesTransferred;
            if(SI->leftToRead==0){
                    ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                    SI->leftToWrite=DATA_BUFSIZE;
                    SI->bp = SI->Buffer;
                    SI->DataBuf.buf=SI->bp;
                    SI->DataBuf.len=SI->leftToWrite;
                    SI->Overlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];
                    SI->direction=WRITE;
                    Flags = 0;
                    _WSASend(SI->Socket, &(SI->DataBuf), 1, NULL, Flags, &(SI->Overlapped), NULL);
            }
            else{
                    ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                    SI->DataBuf.buf=SI->bp;
```

```
                        SI->DataBuf.len=SI->leftToRead;
                        SI->Overlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];
                        SI->direction=READ;
                        Flags = 0;
                        _WSARecv(SI->Socket,&(SI->DataBuf),1, NULL, &Flags, &(SI->Overlapped), NULL);
                }
        }
        break;
        case WRITE:{
                SI->bp += BytesTransferred;
                SI->leftToWrite-= BytesTransferred;
                if(SI->leftToWrite==0){
                        ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                        SI->leftToRead=DATA_BUFSIZE;
                        SI->bp = SI->Buffer;
                        SI->DataBuf.buf=SI->bp;
                        SI->DataBuf.len=SI->leftToRead;
                        SI->Overlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];
                        SI->direction=READ;
                        Flags = 0;
                        _WSARecv(SI->Socket,&(SI->DataBuf),1, NULL, &Flags, &(SI->Overlapped), NULL);
                }
                else{
                        ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                        SI->DataBuf.buf=SI->bp;
                        SI->DataBuf.len=SI->leftToWrite;
                        SI->Overlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];
                        SI->direction=WRITE;
                        Flags = 0;
                        _WSASend(SI->Socket,&(SI->DataBuf), 1, NULL, Flags, &(SI->Overlapped), NULL);
                }
        }
        }//switch
    }//while
    return 0;
    }
```

Листинг 4. Код функции нити ServerWorkerThread сервера с перекрытием и использованием событий

Функции завершения

Напишем программу для эхо-сервера пакетов постоянного размера с использованием перекрытий и функций завершения. Заголовочный файл приведен в листинге 5. В нём не приведены прототипы функций-обвёрток. Они такие же, как и в листинге 2. Приведен прототип для функции завершения.

```
#include <winsock2.h>
#include <windows.h>
#include <stdio.h>
#include "../DATA_BUFSIZE.h"
#define PORT 5150
#define WRITE 7
#define READ -7
typedef struct _SOCKET_INFORMATION {
  OVERLAPPED Overlapped;
  int direction;// WRITE, READ
  SOCKET Socket;
  CHAR Buffer[DATA_BUFSIZE];
  WSABUF DataBuf;
  char *bp;
  DWORD leftToRead;
  DWORD leftToWrite;
} SOCKET_INFORMATION, * LPSOCKET_INFORMATION;
void CALLBACK WorkerRoutine(DWORD Error, DWORD BytesTransferred,
  LPWSAOVERLAPPED Overlapped, DWORD InFlags);
```

```
DWORD WINAPI WorkerThread(LPVOID lpParameter);
```

Листинг 5. Заголовочный файл callback.h сервера с перекрытием и функцией завершения

Код главной функции сервера можно увидеть в листинге 6. С помощью функции WSASocket создаются сокет ListenSocket с перекрытием (параметр WSA_FLAG_OVERLAPPED), который слушает порт 5150. Для создаваемых в последствии сокетов AcceptSocket создаётся событие AcceptEvent. Запускается рабочая нить WorkerThread, которой , в качестве параметра передаётся событие AcceptEvent. Программа входит в бесконечный цикл. В цикле программа ожидает на функции accept очередного соединения. После принятия соединения устанавливается событие AcceptEvent.

```
SOCKET AcceptSocket;
void main(void){
  WSADATA wsaData;
  SOCKET ListenSocket;
  SOCKADDR_IN InternetAddr;
  INT Ret;
  DWORD ThreadId;
  WSAEVENT AcceptEvent;

_ WSAStartup(0x0202,&wsaData);
ListenSocket = _WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(PORT);
Bind(ListenSocket, (PSOCKADDR) &InternetAddr,     sizeof(InternetAddr));
Listen(ListenSocket, 5);
AcceptEvent = _WSACreateEvent();
CreateThread(NULL, 0, WorkerThread, (LPVOID) AcceptEvent, 0, &ThreadId)) == NULL);
while(TRUE){
    AcceptSocket = accept(ListenSocket, NULL, NULL);
     _WSASetEvent(AcceptEvent);
}
}
```

Листинг 6. Код главной функции сервера с перекрытием и функцией завершения

Код главной нити WorkerThread сервера с перекрытием и функцией завершения можно увидеть в листинге 7. Нить работает в бесконечном цикле. В начале цикла присутствует ещё один бесконечном цикле. В начале второго цикла происходит вызов функции WSAWaitForMultipleEvents с установленным в TRUE последним параметром, что позволит выполниться процедуре завершения. Нить блокируется на функции WSAWaitForMultipleEvents, ожидая события AcceptEvent о соединении очередного клиента или окончания процедуры завершения. Если завершится какая-либо функция завершения, то цикл ожидания на функции WSAWaitForMultipleEvents продолжится. При соединении нового клиента произойдёт выход из цикла по оператору break и сброс события AcceptEvent. После принятия соединения резервируется память под переменную SocketInfo и в неё заносится информация для этого соединения. Направление передачи direction установлено в READ. Далее на этом соединении с помощью функции WSARecv инициируется ввод информации от клиента . В качестве процедуры завершения указывается функция WorkerRoutine.

```
#include "callback.h"
DWORD WINAPI WorkerThread(LPVOID lpParameter){
DWORD Flags;
LPSOCKET_INFORMATION SocketInfo;
WSAEVENT EventArray[1];
DWORD Index;
EventArray[0] = (WSAEVENT) lpParameter;
while(TRUE)   {
    while(TRUE){
            Index = _WSAWaitForMultipleEvents(1, EventArray, FALSE, WSA_INFINITE, TRUE);
            if (Index != WAIT_IO_COMPLETION)
                    break;
    }
    WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
    SocketInfo = (LPSOCKET_INFORMATION) GlobalAlloc(GPTR, sizeof(SOCKET_INFORMATION));
    SocketInfo->Socket = AcceptSocket;
   ZeroMemory(&(SocketInfo->Overlapped), sizeof(WSAOVERLAPPED));
```

```
      SocketInfo->direction=READ;
     SocketInfo->leftToRead = DATA_BUFSIZE;
     SocketInfo->leftToWrite = 0;
     SocketInfo->DataBuf.len = DATA_BUFSIZE;
     SocketInfo->DataBuf.buf = SocketInfo->Buffer;
      SocketInfo->bp=SocketInfo->DataBuf.buf = SocketInfo->Buffer;
     Flags = 0;
     _WSARecv(SocketInfo->Socket, &(SocketInfo->DataBuf), 1, NULL, &Flags,
                                  &(SocketInfo->Overlapped), WorkerRoutine);
   }

   return TRUE;
}
```

Листинг 7. Код главной нити WorkerThread сервера с перекрытием и функцией завершения.

Код функциии завершения WorkerRoutine можно увидеть в листинге 8. В начале формальный параметр Overlapped переводится к нашему типу LPSOCKET_INFORMATION (информация о сокете). Проверяется параметр Error га предмет ошибки. Если произошла ошибка или предано ноль байт, то память освобождается и сокет закрывается. Далее логика работы полностью совпадает с логикой работы функции нити ServerWorkerThread сервера с перекрытием, код которой приведен в листинге 4.

```
#include "callback.h"
void CALLBACK WorkerRoutine(DWORD Error, DWORD BytesTransferred,
                        LPWSAOVERLAPPED Overlapped, DWORD InFlags){
DWORD Flags;
LPSOCKET_INFORMATION SI = (LPSOCKET_INFORMATION) Overlapped;
if (Error != 0)
    myprintf("Сбой опереации ввода вывода с ошибкой %d\n", Error);
if (BytesTransferred == 0)
      myprintf("Closing socket %d\n", SI->Socket);
if (Error != 0 || BytesTransferred == 0)   {
     closesocket(SI->Socket);
     GlobalFree(SI);
     return;
}
      switch(SI->direction){
      case READ:{
                SI->bp += BytesTransferred;
                SI->leftToRead-= BytesTransferred;
                if(SI->leftToRead==0){
                        ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                        SI->direction=WRITE;
                        SI->leftToWrite=DATA_BUFSIZE;
                        SI->bp = SI->Buffer;
                        SI->DataBuf.buf=SI->bp;
                        SI->DataBuf.len=SI->leftToWrite;
                        Flags = 0;
                        _WSASend(SI->Socket, &(SI->DataBuf), 1, NULL, Flags, &(SI->Overlapped)),
                                WorkerRoutine);
                }
                else{
                        ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                        SI->direction=READ;
                        SI->DataBuf.buf=SI->bp;
                        SI->DataBuf.len=SI->leftToRead;
                        Flags = 0;
                        _WSARecv(SI->Socket,&(SI->DataBuf), 1, NULL, &Flags, &(SI->Overlapped),
                                WorkerRoutine);
                }
      }
      break;
      case WRITE:{
```

```
                SI->bp += BytesTransferred;
                SI->leftToWrite-= BytesTransferred;
                if(SI->leftToWrite==0){
                        ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                        SI->direction=READ;
                        SI->leftToRead=DATA_BUFSIZE;
                        SI->bp = SI->Buffer;
                        SI->DataBuf.buf=SI->bp;
                        SI->DataBuf.len=SI->leftToRead;
                        Flags = 0;
                        _WSARecv(SI->Socket,&(SI->DataBuf), 1, NULL, &Flags, &(SI->Overlapped),
                                WorkerRoutine);
                }
                else{
                        ZeroMemory(&(SI->Overlapped), sizeof(OVERLAPPED));
                        SI->direction=READ;
                        SI->DataBuf.buf=SI->bp;
                        SI->DataBuf.len=SI->leftToWrite;
                        Flags = 0;
                        _WSASend(SI->Socket,&(SI->DataBuf), 1, NULL, Flags, &(SI->Overlapped),
                                WorkerRoutine);
                }
        }
}//switch
return;
}
```

Листинг 7. Код функции завершения WorkerRoutine.

Контрольные вопросы

1. Можно ли писать сетевые программы, используя для обмена данных по сети обычные функции для записи и чтения файлов?
2. Как подготовить программу для использования модели перекрытого ввода-вывода.
3. Перечислите различия классической функции для создания сокетов socket с функцией для WSASocket.
4. С помощью каких функций можно осуществлять перекрытый ввод-вывод?
5. Что необходимо указать в функциях для перекрытого ввод-вывода, чтобы они действительно осуществляли ввод-вывод в режиме перекрытия?
6. Как осуществить на сокете с перекрытием операции без перекрытия?
7. Блокируется ли вычислительный процесс при вызове функций начинающих перекрытый ввод-вывод?
8. Назовите два способа получения результатов перекрытого ввод-вывода?
9. Объясните значения параметров и логику работы функции WSASend.
10. Объясните значения параметров и логику работы функции WSARecv.
11. Если функция, осуществляющая перекрытый ввод-вывод, возвратила SOCKET_ERROR, то всегда ли это будет ошибкой?
12. Как организовать перекрывающий ввод вывод с использованием событий?
13. Как организовать перекрывающий ввод вывод с использованием функции завершения?
14. Как получить результат перекрытого ввод-вывода при использовании событий?
15. Как получить результат перекрытого ввод-вывода при использовании функции завершения?
16. Можно ли начат новый перекрытый ввод-вывод внутри функции завершения?
17. Используются ли события при перекрытом ввод-выводе с использованием функции завершения?
18. Зачем переводить нить, начавшую ввод-вывод с использованием функции завершения, в состояние ожидания?

Задания

Все программы, которые вы разработаете, должны выводить в окно консоли текущую и среднюю скорость обмена информацией (работа 2) и корректно завершать свою работу (работа 5).

1. Загрузите в Visual Studio 2008 проект для сервера overlap с перекрытиями и использованием событий (листинги 1 - 4)) из решения lab7 архива src. Для отладки в качестве клиента возьмите

программу клиента ClientF из из решения lab2 архива src (листинг 12 из работы 2). Рекомендация: и клиент и сервер должен иметь одинаковый размер буферов DATA_BUFSIZE, поэтому сделайте для их обоих общий заголовочный файл DATA_BUFSIZE.h. Используя подход, изложенный в работе №2, дополните проекты сервера кодом, позволяющим выводить в окно консоли текущую и среднюю скорость обмена информацией.

2. Загрузите в Visual Studio 2008 проект для сервера callback с перекрытиями и функциями завершения (листинги 5 - 7) из решения lab7 архива src. Для отладки в качестве клиента возьмите программу клиента ClientF из из решения lab2 архива src (листинг 12 из работы 2). Рекомендация: и клиент и сервер должен иметь одинаковый размер буферов DATA_BUFSIZE, поэтому сделайте для их обоих общий заголовочный файл DATA_BUFSIZE.h. Используя подход, изложенный в работе №2, дополните проекты сервера кодом, позволяющим выводить в окно консоли текущую и среднюю скорость обмена информацией.
3. Используя подход, изложенный в работе №5, дополните проекты серверов overlap и callback кодом, обеспечивающим корректное завершение работы. Указание: используйте по своему усмотрению или функцию WSAAsyncSelect или функцию WSAEventSelect для установки реакции на сетевое событие FD_CLOSE.
4. Сравните текущую и среднюю скорость обмена информацией и объясните результат для соединений:
   a. Клиента ClientF из работы 2 к серверу overlap из этой работы
   b. Клиента GracefulShutdownAsyncSelectclient из работы 5 к серверу overlap из этой работы
   c. Клиента AsyncEventClient из работы 6 к серверу overlap из этой работы
   d. Клиента ClientF из работы 2 к серверу callback из этой работы
   e. Клиента GracefulShutdownAsyncSelectclient из работы 5 к серверу callback из этой работы
   f. Клиента AsyncEventClient из работы 6 к серверу callback из этой работы
5. Модифицируйте программы сервера overlap для работы с пакетами переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом. Для отладки используйте клиенты для работы с пакетами переменной длины, которые вы разработали ранее в работах 2, 5 и 6.
6. Модифицируйте программы сервера callback для работы с пакетами переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом. Для отладки используйте клиенты для работы с пакетами переменной длины, которые вы разработали ранее в работах 2, 5 и 6.

Лабораторная работа №8. Асинхронные сокеты. Модель портов завершения

Теоретическая часть

Если приложение должно управлять сотнями и тысячами сокетов одновременно, то эта модель ввода-вывода позволяет достичь наивысшего быстродействия. Её следует использовать, если необходимо добиться хорошей масштабируемости при добавлении новых процессоров.

Порт завершения - это конструкция ввода-вывода из Win32, способная работать не только с сокетами. Модель портов завершения требует создать Win32-обЪект порта завершения, который будет управлять запросами перекрытого ввода-вывода, используя указанное количество нитей для обработки завершения этих запросов. Функция

```
HANDLE CreateIoCompletionPort(HANDLE FileHandle, HANDLE ExistingCompletionPort,
    DWORD CompletionKey, DWORD NumberOfConcurrentThreads)
```

создаёт или привязывает объект порта завершения. Параметр FileHandle - дескриптор сокета, который нужно связать с портом завершения, ExistingCompletionPort определяет существующий порт завершения. Используя параметр CompletionKey, приложение может связать с сокетом любые данные. Функция используется в двух разных целях: создания порта завершения и привязки к нему описателя сокета. При первоначальном создании порта используется один параметр - NumberOfConcurrentThreads, первые три параметра игнорируются. NumberOfConcurrentThreads определяет количество нитей, которые могут одновременно выполняться на порте завершения. В идеале порт должен обслуживаться только одной нитью на каждом процессоре, чтобы избежать переключений контекста. Значение 0 разрешает выделить число нитей, равное числу процессоров в системе. Создать порт завершения можно так:

```
CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
```

При этом возвращается описатель порта завершения, используемый при привязке сокета.

Прежде чем начинать привязку, необходимо создать одну или несколько рабочих нитей для обслуживания порта, когда на него отправляются запросы ввода-вывода сокетов. Требуемое число нитей зависит от структуры приложения. Количество конкурентных нитей, задаваемых при вызове CreateloCompletionPort, и количеством создаваемых рабочих нитей - это не одно и то же. Если параметр NumberOfConcurrentThreads равен n, то разрешается только n нитям одновременно работать с портом завершения. Даже если будет создано более n рабочих нитей для порта завершения, только n смогут работать одновременно. Это значение может быть превышено на короткий промежуток времени, но система быстро сократит количество нитей до величины n.

Если одна из рабочих нитей приостанавливается (путем вызова функции Sleep или WaitForSingleObjecf), тогда должна существовать другая нить, которая сможет работать с портом вместо неё. Поэтому следует создавать рабочих нитей больше, чем указано в вызове CreateloCompletionPort

В листинге 1 приведен скелет приложения эхо-сервера с использованием модель портов завершения. Выделим следующие этапы:

1. Создается порт завершения. Четвертый параметр равен 0, что разрешает только одной рабочей нити на процессор одновременно выполняться на порте завершения.

2 Выясняется, сколько процессоров в системе

3. Создаются рабочие нити для обслуживания завершившихся запросов ввода-вывода порта завершения с использованием информации о процессорах, полученной на шаге 2. В нашем простом примере мы создаем одну рабочую нить на процессор, так как не ожидаем приостановки работы нитей. При вызове функции CreateThread нужно указать рабочую процедуру ServerWorkerThread, которую нить выполняет после создания.

4. Готовится сокет для прослушивания порта 5150

5. Принимается входящее соединение функцией accept.

6. Создается наша структура данных и в ней сохраняется дескриптор принятого сокета.

7. Возвращенный функцией accept новый дескриптор сокета связывается с портом завершения вызовом функции CreateloCompletionPort. Наша структура данных передается в параметре CompletionKey.

8. Начинается обработка ввода-вывода на принятом соединении. Один или несколько асинхронных запросов WSARecv или WSASend отправляются на сокет с использованием механизма перекрытого ввода-вывода. Когда эти запросы завершаются, рабочая нить обслуживает их и продолжает обрабатывать новые (в рабочей процедуре на шаге 3).

9. Шаги 5-8 повторяются до окончания работы сервера.

```
// Шаг 1:
CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
// Шаг 2:
GetSystemlnfo(&SystemInfo);
```

```
// Шаг 3:
for(i = 0; i < SystemInfo.dwNumberOfProcessors;    i++) {
HANDLE ThreadHandle;
ThreadHandle = CreateThread(NULL, 0, ServerWorkerThread, CompletionPort,
        0, &ThreadID);
CloseHandle(ThreadHandle);
}
// Шаг 4:
Listen = WSASocket(AF_INET, SOCK.STREAM, 0, NULL, 0,WSA_FLAG_OVERLAPPED);
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);
bind(Listen, (PSOCKADDR) AlnternetAddr,sizeof(InternetAddr));
listen(Listen, 5);
while(TRUE){
// Шаг 5:
Accept = WSAAccept(Listen, NULL, NULL, NULL, 0);
// Шаг 6:
PerHandleData = (LPPER_HANDLE_DATA)GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));
PerHandleData->Socket = Accept;
// Шаг 7
CreateIoCompletionPort((HANDLE) Accept,CompletionPort, (DWORD) PerHandleData, 0);
// Шаг 8
WSASend или  WSARecv()
}
```

Листинг 1. Настройка порта завершения

Модель портов завершения использует преимущества механизма перекрытого ввода-вывода Win32, в котором вызовы функций Winsock API (таких, как WSASend и WSARecv) завершаются немедленно после вызова. Затем приложение должно правильно извлечь результаты из структуры OVERLAPPED. В модели портов завершения это достигается постановкой одной или нескольких рабочих нитей в очередь ожидания на порте завершения с помощью функции

```
BOOL GetQueuedCompletionStatus(HANDLE CompletionPort, LPDWORD lpNumberOfBytesTransferred,
    LPDWORD lpCompletionKey, LPOVERLAPPED * lpOverlapped, DWORD dwMilliseconds),
```

Параметр CompletionPort - порт завершения, на котором будет ждать нить. Параметр lpNumberOfBytesTransferred принимает количество байт, перемещенных после операции ввода-вывода, такой как WSASend или WSARecv. Как мы уже упоминали, через параметр lpCompletionKey лучше передавать дескриптора сокета, который был первоначально передан в функцию CreateIoCompletionPort. В параметр lpOverlapped записывается перекрытый результат завершенной операции ввода-вывода. Это действительно важный параметр - он позволяет получить данные операции ввода-вывода. Последний параметр - dwMilliseconds, задает, сколько миллисекунд вызывающая нить будет ждать появления пакета завершения на порте (если он равен INFINITE, ожидание длится бесконечно) .

Когда рабочая нить получает уведомление о завершении ввода-вывода от функции GetQueuedCompletionStatus, параметры lpCompletionKey и lpOverlapped содержат информацию о сокете, которая может быть использована для продолжения обработки ввода-вывода через порт завершения.

Параметр lpComplettonKey содержит данные, которые передаются в параметре CompletionKey при вызове функции CreateIOCompletionPort. Приложение может передать через этот параметр любой тип информации, связанной с сокетом. Как правило, здесь хранится дескриптор сокета, связанного с запросом ввода-вывода.

Параметр lpOverlapped содержит структуру OVERLAPPED, за которой следуют дополнительные данные операции. Проще всего определить свою структуру с первым элементом типа OVERLAPPED. Например, мы объявляем следующую структуру для управления данными операции

```
typedef struct{
OVERLAPPED Overlapped,
WSABUF DataBuf,
CHAR Buffer[DATA_BUFSIZE],
BOOL direction;} PER_IO_OPERATION_DATA;
```

В эту структуру входят важные дополнительные элементы данных, например тип завершившейся операции direction (запрос на отправку или прием). В этой структуре полезен и буфер данных (DataBuf , Buffer) для завершившейся операции. При вызове функции Winbock, принимающей в качестве параметра структуру OVERLAPPED, можно привести вашу структуру к указателю на OVERLAPPED или просто передать ссылку на элемент OVERLAPPED вашей структуры

```
    PER_IO_OPERATION_DATA PerIoData;

// Функцию нужно вызывать так
WSARecv(socket, , (OVERLAPPED *)&PerIoData);
// или так
WSARecv(socket, , &(PerIoData.Overlapped)).
```

Затем, когда в рабочей нити функция GetQueuedCompletionStatus вернет структуру OVERLAPPED, можно определить тип запроса, который был отправлен на сокет, приведя структуру OVERLAPPED к вашему типу PER_IO_OPERATION_DATA. Данные об операции весьма полезны, так как позволяют управлять несколькими операциями ввода-вывода (чтение-запись, множественное чтение, множественная запись и т. п.) на одном дескрипторе. Отправлять запросы на несколько операций одновременно на один сокет следует для обеспечения масштабируемости. Например, на многопроцессорной машине, где рабочие нити используют все процессоры, несколько процессоров смогут отправлять и принимать данные через один сокет одновременно.

В листинге 2 показано, как разработать процедуру рабочей нити для обслуживания запросов ввода-вывода.

```
DWORD WINAPI ServerWorkerThread(LPVOID CompletionPortID){
HANDLE CompletionPort = (HANDLE) CompletionPortID;
DWORD BytesTransferred;
LPOVERLAPPED Overlapped;
LPPER_HANDLE_DATA PerHandleData;
LPPER_IO_OPERATION_DATA PerloData;
DWORD SendBytes, RecvBytes;
DWORD Flags;
while(TRUE){
// Ожидание завершения ввода-вывода на любом  из сокетов, связанных с портом завершения
    GetQueuedCompletionStatus(CompletionPort, &BytesTransferred,(LPDWORD)&PerHandleData,
                                                (LPOVERLAPPED *) &PerIoData, INFINITE);
// Сначала проверим, не было ли ошибки  на сокете; если была, закрываем сокет  и очищаем данные
    if (BytesTransferred == 0){
    // Отсутствие перемещенных байт (BytesTransferred) означает, что сокет закрыт партнером по
    //соединению и нам тоже нужно закрыть сокет

        closesocket(PerHandleData->Socket);

        GlobalFree(PerHandleData);

        GlobalFree(PerIoData);

        continue;

    }

 // Обслуживание завершившегося запроса ввода-вывода. Чтобы определить, какой запрос завершился,
// нужно просмотреть в данных операции поле OperationType.
if (PerIoData-> direction == RECV_POSTED){
// Обработка принятых данных  в буфере PerIoData->Buffer
}
// Отправка нового запроса ввода-вывода WSASend или WSARecv.
// В качестве примера отправим еще один асинхронный запрос WSARecv
// Flags = 0;
// Настройка данных операции  для следующего запроса перекрытого ввода-вывода

 ZeroMemory(&(PerIoData->Overlapped),sizeof(OVERLAPPED));

 PerIoData->DataBuf.len = DATA_BUFSIZE;
```

```
      PerIoData->DataBuf.buf = PerIoData->Buffer;
      PerIoData-> direction = RECV_POSTED;
      WSARecv(PerHandleData->Socket,  &(PerIoData->DataBuf), 1, &RecvBytes,
                              &Flags, &(PerIoData->Overlapped), NULL);
   }
}
```

Листинг 2. Рабочая нить порта завершения

Последний момент - корректное закрытие порта завершения. Это особенно важно, если один или несколько выполняющихся нитей ведут ввод-вывод на нескольких сокетах. Главное - не освобождать структуру OVERLAPPED, пока выполняется перекрытый запрос ввода-вывода. Лучше всего вызывать функцию closesocket для каждого описателя сокета, тогда все операции перекрытого ввода-вывода будут завершены. После закрытия всех сокетов нужно завершить все рабочие нити порта завершения. Отправьте каждой нити специальный завершающий пакет функцией PostQueuedCompletionStatus, это заставит нить немедленно прекратить работу:

```
BOOL PostQueuedCompletionStatus(HANDLE CompletlonPort,DWORD dwNumberOfBytesTransferred,
    DWORD dwCompletionKey,LPOVERLAPPED lpOverlapped);
```

Параметр CompletlonPort - объект порта завершения, которому нужно отправить завершающий пакет. Параметры dwNumberOfBytesTransferred, dwCompletionKey и lpOverlapped позволяют задать значение, которое будет записано прямо в соответствующий параметр функции GetQueuedCompletionStatus. Когда рабочая нить получит эти три параметра, он сможет определить, когда следует прекращать работу, на основе специального значения, заданного в одном из трех параметров. Например, можно передать 0 в параметре dwCompletionKey - рабочая нить интерпретирует это как инструкцию об окончании работы. После закрытия всех рабочих нитей закройте порт завершения, вызвав функцию CloseHandle, и безопасно выйдите из программы

Существует несколько приемов, позволяющих увеличить эффективность ввода-вывода через сокеты с использованием портов завершения. Один из них - опытным путем подобрать размер буфера сокета, чтобы увеличить производительность и масштабируемость приложения. Например, приложение, которое использует один большой буфер и только один запрос WSARecv вместо трех маленьких буферов для трех запросов WSARecv, будет плохо масштабироваться на многопроцессорных машинах, поскольку с одним буфером одновременно может работать только одна нить. Пострадает и производительность, если одновременно выполняется только одна операция приема, драйвер сетевого протокола будет недостаточно загружен. То есть если вам приходится ждать завершения WSARecv перед получением новых данных, протокол будет простаивать между завершением WSARecv и следующим приемом.

Существует еще один способ увеличить производительность - проанализируйте результаты использования параметров сокета SO_SNDBUF и SO_RCVBUF для управления размером внутренних буферов сокета. Они позволяют приложению изменять размер внутренних буферов Если приравнять их к 0, Winsock будет напрямую использовать буфер приложения во время перекрытого вызова для передачи данных в стек протокола и обратно, уменьшая межбуферное копирование. Следующий фрагмент показывает, как вызывать функцию setsockopt для настройки параметра SO_SNDBUF

```
int nZero = 0;
setsockopt(socket, SOL.SOCKET, SO_SNDBUF,(char *)&nZero, sizeof(nZero)),
```

Заметьте нулевой размер буферов даст положительный эффект, только если несколько запросов ввода-вывода отправляются одновременно.

В заключение заметим, что Winsock-приложения не должны применять Win32-функции ReadFile и WnteFile для обработки ввода-вывода через порт завершения. Хотя эти функции используют структуру OVERLAPPED и ошибки не произойдет, функции WSARecv и WSASend лучше оптимизированы для обработки ввода-вывода в Winsock 2. Обращение к ReadFile и WnteFile ведет к множеству ненужных вызовов процедур системного ядра, переключений контекста и передачи параметров, что в итоге значительно снижает производительность

Если вы используете порты завершения ввода-вывода, то примите к сведению, что порядок вызовов функций является также порядком по которому наполняются буфера. Не следует вызывать функции на том же сокете, но в отдельных нитях, так как это может привести к непредсказуемому порядку в буфере.

Практическая часть

Рассмотрим листинг 3, в котором предоставлен заголовочный файл IOCPAsyncEvent.h эхо сервера с использованием портов завершения. Сервер также использует модель WSAEventSelect для обрабртки сетевых событий не связанных непосредственно с вводом и выводом (FD_CLOSE). Это вам понадобится для

дополнения сервера свойством корректно завершать свою работу. В файле IOCPAsyncEvent.h мы определили свой тип данных PER_IO_OPERATION_DATA для обмена информацией. Поле direction определяет направление передачи. Поля leftToRead и leftToWrite содержат количество байт, которое осталось прочитать или передать.

```
#include <winsock2.h>
#include "../../common/DATA_BUFSIZE.h"
#define PORT 5150
#define WRITE 7
#define READ -7
typedef struct{
  OVERLAPPED Overlapped;
  int direction;// WRITE, READ
     WSABUF DataBuf;
  CHAR Buffer[DATA_BUFSIZE];
  DWORD leftToRead;
  DWORD leftToWrite;
     char *bp;
     int Shutdown;
} PER_IO_OPERATION_DATA, * LPPER_IO_OPERATION_DATA;

typedef struct {
  SOCKET Socket;
} PER_HANDLE_DATA, * LPPER_HANDLE_DATA;

void FreeSocketInformation(DWORD Event);
DWORD WINAPI ServerWorkerThread(LPVOID CompletionPortID);
int myprintf (const char *lpFormat, ... );
```

Листинг 3. Заголовочный файл эхо сервера

Код главной функции эхо сервера приведен в листинге 4. Вначале мы создаём пустой порт завершения CompletionPort и создаём рабочие нити, числом в два раза превышающим число процессоров в системе. Создаём слушающий сокет Listen с перекрытием и входим в бесконечный цикл. В этом цикле мы принимаем новое соединение в сокет Accept и с помощью функции CreateIoCompletionPort связывыаем новый сокет с портом завершения CompletionPort. Дескриптор сокета Accept мы передаём функции CreateIoCompletionPort с помощью поля Socket структуры PerHandleDatas. Далее готовим данные для приёма первой порции данных от клиента. Для этого заполняем поля структуры LPPER_IO_OPERATION_DATA PerIoData: перекрытие Overlapped, буфера DataBuf и Buffer, указатель записи чтения bp, направление передачи direction и количества байт leftToRead и leftToWrite, которое осталось прочитать или передать. Далее с помощью функции WSARecv инициируем этот приём.

```
#include "IOCP.h"
LPPER_HANDLE_DATA PerHandleData;
LPPER_IO_OPERATION_DATA PerIoData;

void main(void){
  HANDLE CompletionPort, hTh;
  SYSTEM_INFO SystemInfo;
  DWORD Ret;
  SOCKET Listen, Accept;
  SOCKADDR_IN InternetAddr;
  WSADATA wsaData;
  DWORD        dwThreadId;
  int i;
 HANDLE hTh;

  WSAStartup(0x0202, &wsaData);
    if ((CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0)) == NULL) {
     myprintf( "CreateIoCompletionPort failed with error: %d\n", GetLastError());
     return;
  }
  GetSystemInfo(&SystemInfo);
  for(i = 0; i < SystemInfo.dwNumberOfProcessors * 2; i++) {
```

```
    if ((hTh=CreateThread(NULL, 0, ServerWorkerThread, CompletionPort,0, &dwThreadId)) == NULL)     {
      myprintf("CreateThread() failed with error %d\n", GetLastError());
      return;
    }
CloseHandle(hTh);
  }
  if ((Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,  WSA_FLAG_OVERLAPPED))
                                                                    ==INVALID_SOCKET) {
        myprintf("WSASocket() failed with error %d\n", WSAGetLastError());
        return;
  }

  InternetAddr.sin_family = AF_INET;
  InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  InternetAddr.sin_port = htons(PORT);
  Bind(Listen, (PSOCKADDR) &InternetAddr, sizeof(InternetAddr)) ;
  Listen(Listen, SOMAXCONN);
 while(TRUE)   {
        if ((Accept = accept(Listen, NULL, NULL)) == SOCKET_ERROR)     {
            myprintf("WSAAccept() failed with error %d\n", WSAGetLastError());
             continue;
        }

        if ((PerHandleData= (LPPER_HANDLE_DATA) GlobalAlloc(
                    GPTR,sizeof(PER_HANDLE_DATA))) == NULL)     {
            myprintf("GlobalAlloc() failed with error %d\n", GetLastError());
            continue;
        }
        PerHandleData->Socket = Accept;
        if (CreateIoCompletionPort((HANDLE) Accept, CompletionPort,
                                (DWORD) PerHandleData, 0) == NULL) {
            myprintf("CreateIoCompletionPort failed with error %d\n", GetLastError());
            continue;
        }
        //Готовим данные для опрерации WSARecv
        if ((PerIoData= (LPPER_IO_OPERATION_DATA)
                    GlobalAlloc(GPTR,sizeof(PER_IO_OPERATION_DATA))) == NULL) {
            myprintf("GlobalAlloc() failed with error %d\n", GetLastError());
            continue;
        }

        ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
        PerIoData->DataBuf.len = DATA_BUFSIZE;
        PerIoData->DataBuf.buf = PerIoData->Buffer;
        PerIoData->leftToRead = DATA_BUFSIZE;
        PerIoData->leftToWrite = 0;
        PerIoData->bp=PerIoData->Buffer;
        PerIoData->direction=READ;
        Flags = 0;
        if (WSARecv(Accept, &(PerIoData->DataBuf), 1, &RecvBytes, &Flags,
                            &(PerIoData->Overlapped),NULL) ==SOCKET_ERROR) {
             if (WSAGetLastError() != ERROR_IO_PENDING)  {
                myprintf("WSARecv() failed with error %d\n", WSAGetLastError());
                continue;
            }
        }
    EventTotal++;
  }   //   if (… FD_ACCEPT)
 }//while
}
```

Листинг 4. Код главной функции эхо сервера

Код функции ServerWorkerThread рабочей нити порта завершения приведен в листинге 5. В начале бесконечного цикла функция ServerWorkerThread с помощью функции GetQueuedCompletionStatus получает результаты ввода-вывода на порту завершения CompletionPort. Параметр BytesTransferred возвращает число перемещённых байт. Если он равен нулю, то это значит, что клиент завершил работу и мы закрываем сокет, освобождая выделенную память. Дескриптор сокета s, на котором осуществляется операция получаем из параметра PerHandleData. Для работы с остальными результатами завершившейся операции используем параметр PerIoData.

Анализируем, какая операция завершилась (switch(PerIoData->direction)). Если это было чтение (READ) данных от клиента , то изменяем значения полей bp и leftToRead структуры PerIoData на число перемещённых байт BytesTransferred . Если приняты все данные (PerIoData->leftToRead==0), то готовим и инициируем передачу ответных данных клиенту с помощью функции WSASend. Если приняты не все данные, то готовим и инициируем приём оставшихся данных от клиента с помощью функции WSARecv.

Если это был ответ (WRITE) клиенту, то изменяем значения полей bp и leftToWrite структуры PerIoData на число перемещённых байт BytesTransferred . Если переданы все данные (PerIoData-> leftToWrite ==0), то готовим и инициируем приём новых данных от клиента с помощью функции WSARecv. Если в ответе переданы не все данные, то готовим и инициируем передачу оставшихся данных клиенту с помощью функции WSASend.

```
#include "IOCPAsyncEvent.h"
DWORD WINAPI ServerWorkerThread(LPVOID CompletionPortID){
HANDLE CompletionPort = (HANDLE) CompletionPortID;
DWORD BytesTransferred;
LPPER_HANDLE_DATA PerHandleData;
LPPER_IO_OPERATION_DATA PerIoData;
DWORD SendBytes, RecvBytes;
DWORD Flags;
SOCKET s;
while(TRUE) {
    if (GetQueuedCompletionStatus(CompletionPort, &BytesTransferred,
                    (LPDWORD)&PerHandleData, (LPOVERLAPPED *)&PerIoData, INFINITE)== 0) {
            myprintf("GetQueuedCompletionStatus failed with error %d\n", GetLastError());
            continue;
     }
     s=PerHandleData->Socket;
      if (BytesTransferred == 0 &&(PerIoData->direction == READ ||PerIoData->direction == WRITE)){
             if(closesocket(s)== INVALID_SOCKET) {
                    myprintf("closesocket() failed with error %d\n", WSAGetLastError());
                    continue;
             }
             if(GlobalFree(PerHandleData)||GlobalFree(PerIoData)){
                    myprintf("GlobalFree() failed with error %d\n", GetLastError());
                    continue;
             }
             myprintf("Закрыт сокет %d.\n",s);
             continue;
      }
     switch(PerIoData->direction){
     case READ:{
            PerIoData->bp += BytesTransferred;
            PerIoData->leftToRead-= BytesTransferred;
            if(PerIoData->leftToRead==0){
                    ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
                    PerIoData->leftToWrite=DATA_BUFSIZE;
                    PerIoData->bp = PerIoData->Buffer;
                    PerIoData->direction=WRITE;
                    PerIoData->DataBuf.buf=PerIoData->bp;
                    PerIoData->DataBuf.len=PerIoData->leftToWrite;
                    Flags = 0;
                    if (WSASend(s, &(PerIoData->DataBuf), 1, &SendBytes, Flags,
                                    &(PerIoData->Overlapped),NULL)==SOCKET_ERROR){
                            if (WSAGetLastError() != ERROR_IO_PENDING){
                                    myprintf("WSASend failed with error %d\n", WSAGetLastError());
                                    continue;
```

```
                        }
                    }
                }
                else{
                    ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
                    PerIoData->direction=READ;
                    PerIoData->DataBuf.buf=PerIoData->bp;
                    PerIoData->DataBuf.len=PerIoData->leftToRead;
                    Flags = 0;
                    if (WSARecv(s,&(PerIoData->DataBuf), 1, &RecvBytes, &Flags,
                                    &(PerIoData->Overlapped), NULL) == SOCKET_ERROR){
                        if (WSAGetLastError() != ERROR_IO_PENDING){
                            myprintf("WSARecv() failed with error %d\n", WSAGetLastError());
                            continue;
                            }
                        }
                    }
                }
                break;
    case WRITE:{
                PerIoData->bp += BytesTransferred;
                PerIoData->leftToWrite-= BytesTransferred;
                if(PerIoData->leftToWrite==0){
                    ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
                    PerIoData->leftToRead=DATA_BUFSIZE;
                    PerIoData->bp = PerIoData->Buffer;
                    PerIoData->direction=READ;
                    PerIoData->DataBuf.buf=PerIoData->bp;
                    PerIoData->DataBuf.len=PerIoData->leftToRead;
                    Flags = 0;
                    if (WSARecv(s,&(PerIoData->DataBuf), 1, &RecvBytes, &Flags,
                                    &(PerIoData->Overlapped), NULL) == SOCKET_ERROR){
                        if (WSAGetLastError() != ERROR_IO_PENDING){
                            myprintf("WSARecv() failed with error %d\n", WSAGetLastError());
                            continue;
                            }
                        }
                }
                else{
                    ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
                    PerIoData->direction=WRITE;
                    PerIoData->DataBuf.buf=PerIoData->bp;
                    PerIoData->DataBuf.len=PerIoData->leftToWrite;
                    Flags = 0;
                    if (WSASend(s,&(PerIoData->DataBuf), 1, &SendBytes, Flags,
                                    &(PerIoData->Overlapped), NULL) == SOCKET_ERROR){
                        if (WSAGetLastError() != ERROR_IO_PENDING){
                            myprintf("WSARecv() failed with error %d\n", WSAGetLastError());
                            continue;
                            }
                        }
                }
    }// case WRITE
                break;
    }// switch
}//while
return 0;
}
```

Листинг 5. Код рабочей нити порта завершения эхо сервера

Контрольные вопросы

1. Когда применяется модель портов завершения?
2. Можно ли эту модель использовать для работы с дисковыми файлами?

3. Какие функции используются для создания порта завершения и для привязки сокета к порту завершения?
4. Объясните назначение параметров функции CreateIoCompletionPort.
5. Как соотносится число созданных рабочих нитей с числом нитей, указанным при создании порта завершения?
6. Объясните алгоритм работы программы, представленной в листинге 1.
7. Как рабочей нити прта завершения извлечь результаты ввода-вывода из сокета?
8. Объясните назначение параметров функции GetQueuedCompletionStatus
9. Как соотносятся параметры функции GetQueuedCompletionStatus с параметрами функции CreateIOCompletionPort?
10. Как соотносятся параметры функции GetQueuedCompletionStatus с параметрами функций CreateIOCompletionPort WSASend и WSARecv?
11. Как передать через функцию WSASend или WSARecv и получит из функции GetQueuedCompletionStatus свои дополнительные данные?
12. Объясните алгоритм работы программы, представленной в листинге 2.
13. Как корректно закрыть порт завершения?
14. Как увеличить эффективность ввода-вывода через сокеты с использованием портов завершения?
15. Объясните алгоритм работы программы, представленной в листинге 4.
16. Объясните алгоритм работы программы, представленной в листинге 5.

Задания

1. Постройте сервер iocp. Используйте готовое решение lab8 из архива src. Следуя работе 2, снабдите сервер выводом на консоль текущей и средней скоростей обмена.
2. Сравните текущую и среднюю скорость обмена информацией и объясните результат для соединений к этому серверу:
   a. Клиента ClientF из работы 2.
   b. Клиента GracefulShutdownAsyncSelectclient из работы 5.
   c. Клиента AsyncEventClient из работы 6.
3. Исследуйте зависимость быстродействия сервера от размера буфера DATA_BUFSIZE_INT при числе клиентов: 1,2,3 … для клиентов ClientF из работы 2, GracefulShutdownAsyncSelectclient из работы 5 и AsyncEventClient из работы 6.
4. Установите с помощью функции setsockopt размер внутренних буферов приёма и передачи в ноль (см. выше) и повторите пункт 3.
5. Следуя подходу работы 5, обеспечьте корректное завершение работы сервера iocp: сервер должен уведомлять клиентов о своём закрытии. Указание: сохраняйте дескрипторы открытых сокетов в массиве и используйте функцию PostQueuedCompletionStatus (см. выше). В качестве клиента используйте ClientF из работы 2, GracefulShutdownAsyncSelectclient из работы 5 и AsyncEventClient из работы 6.
6. Перепишите сервер для использования пакетов переменной длины. Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных. Длина пакета определяется клиентом случайным образом. Для отладки используйте клиенты для работы с пакетами переменной длины, которые вы разработали ранее в работах 2, 4(5) и 6.

Перечень кодов ошибок Winsock

10004-WSAEINTR
Прерванный вызов функции.

Блокирующий вызов прерывается вызовом WSACancelBlockingCall.

10013-WSAEACCES
Доступ запрещен.

Была сделана попытка обращения к запрещенному сокету. Обычно эта ошибка возникает при попытке использовать широковещательный адрес в функциях sendto или WSASendTo, когда широковещание не разрешено параметрами setsockopt и SO_BROADCAST

10014-WSAEFAULT
Недопустимый адрес.

Функции Winsock передан недопустимый указатель адреса. Ошибка также возникает, когда указан слишком маленький буфер

10022-WSAEINVAL
Недопустимый параметр.

Указано недопустимое значение параметра, например, контрольного кода в функции WSAIoctl. Эта же ошибка может возникнуть из-за текущего состояния сокета, например, при вызове функций accept или WSAAccept на сокете, не находящемся в состоянии прослушивания

10024-WSAEMFILE
Открыто слишком много файлов. Открыто слишком много сокетов. Обычно системы доступа Microsoft ограничены только количеством доступных ресурсов ОС

10035 - WSAEWOULDBLOCK
Ресурс временно не доступен.

Ошибка наиболее часто возвращается неблокирующими сокетами, на которых запрошенную операцию (например, вызов функции connect) нельзя выполнить немедленно

10036-WSAEINPROGRESS
Операция выполняется.

Блокирующая операция находится в процессе выполнения. Ошибка появляется редко, только при разработке 16-битных приложений Winsock

10037-WSAEALREADY
Действие уже выполняется.

На неблокирующем сокете происходит попытка выполнить операцию, которая уже вызвана и выполняется например, при повторном вызове на неблокирующем сокете функции connect или WSAConnect во время установления соединения. Ошибка также возникает, когда поставщик службы находится в процессе выполнения функции обратного вызова (для функций, поддерживающих обратный вызов)

10038-WSAENOTSOCK
Операция на недопустимом сокете.

Ошибка может быть возвращена любой функцией Winsock параметром которой является дескриптор типа SOCKET. Означает, что задан неверный дескриптор сокета

10039-WSAEDESTADDRREQ
Требуется адрес назначения.

Ошибка появляется, если пропущен адрес например, при вызове функции sendto с адресом назначения INADDR_ANY

10040-WSAEMSGSIZE
Слишком длинное сообщение. Сообщение отправляется на дейтаграммный сокет размером больше внутреннего буфера. Ошибка также возникает, если сообщение больше, чем позволяют ограничения сети. Еще одна причина появления данной ошибки — буфер слишком мал, чтобы вместить полученную дейтаграмму

10041-WSAEPROTOTYPE
Неверный тип протокола для сокета. При вызове socket или WSASocket указан протокол, не поддерживающий семантику данного типа сокета. Например, попытка создать IP-сокет типа SOCKSTREAM с протоколом IPPROTOUDP

10042-WSAENOPROTOOPT
Неверный параметр протокола. Неизвестный, не поддерживаемый или неверный параметр сокета. Ошибка также возникает, если указан неверный уровень или опция при вызове getsockopt или setsockopt

10043-WSAEPROTONOSUPPORT
Не поддерживаемый протокол. Запрошенный протокол не установлен в системе Например, попытка создать TCP- или UDP-сокет, при том, что в системе не установлен протокол TCP/IP

10044-WSAESOCKTNOSUPPORT
Не поддерживаемый тип сокета. Данное семейство адресов не поддерживает указанный тип сокета. Например, запрос сокета типа SOCK_RAW по протоколу, не поддерживающему простые сокеты

10045-WSAEOPNOTSUPP
Не поддерживаемая операция. Объект обращения не поддерживает запрошенную операцию. Обычно возникает при попытке вызвать функции Winsock на сокете, не поддерживающем данную операцию. Например, accept или WSAAccept на дейтаграммном сокете

10046-WSAEPFNOSUPPORT
Не поддерживаемое семейство протоколов. Запрошенное семейство протоколов не существует или не установлено в системе. В большинстве случаев ошибка означает то же, что и более частая ошибка WSAEAFNOSUPPORT

10047 - WSAEAFNOSUPPORT
Семейство адресов не поддерживает запрошенную операцию. Попытка выполнить операцию, не поддерживаемую данным типом сокета: например, при вызове sendto или WSASendTo на сокете типа SOCK_STREAM. Ошибка также возникает при вызове socket или WSASocket с указанием неверной комбинации семейства адресов, типа сокета и протокола.

10048-WSAEADDRINUSE
Адрес уже используется. При обычных обстоятельствах только один
сокет может использовать каждый адрес сокета. Например, адрес 1P-сокета
состоит из локального IP-адреса и номера порта. Ошибка обычно связана с
функциями bind, connect и WSAConnect. Для функции setsockopt может быть
задан параметр SO_REUSEADDR, разрешающий использование несколькими
сокетами одного и того же локального IP-адреса и порта (см. главу 9)
10049-WSAEADDRNOTAVAIL
Невозможно назначить запрошенный адрес. При вызове API указан недопустимый для данной функции адрес: например, в функции bind задан IP-адрес, не относящийся к локальному IP-интерфейсу. Также ошибка может возникнуть при указании порта 0 удаленной машины, с которой производится соединение, в функциях connect, WSAConnect, sendto, WSASendTo, и WSAJoinLeaf.

10050-WSAENETDOWN
Сеть не работает. Операция не может быть выполнена из-за неполадок в сети, стеке, сетевом интерфейсе или с локальной сети.

10051-WSAENETUNREACH
Сеть недоступна. Попытка произвести операцию в недоступной сети: локальный узел не знает, как достичь удаленный. Другими словами, не существует известного маршрута к месту назначения.

10052-WSAENETRESET

Сеть разорвала соединение при сбросе. Соединение было разорвано из-за невозможности доставки сообщений о его продолжении. Ошибка также возникает при попытке задать параметр SO_KEEPALIVE в функции setsockopt, когда соединение уже разорвано.

10053-WSAECONNABORTED
Преждевременный разрыв соединения из-за ошибки ПО. Произошла ошибка протокола или истек таймаут.

10054- WSAECONNRESET
Соединение разорвано партнером по связи. Установленное соединения было принудительно закрыто удаленным узлом. Ошибка возникает, если удаленный процесс не работоспособен (например, при ошибке обращения к памяти или сбое аппаратуры) или если на сокете было произведено принудительно закрытие. Сокет можно настроить для резкого закрытия с помощью параметра SO_LINGER и функции setsockopt

10055- WSAENOBUFS
Свободное место в буфере закончилось. Запрошенная операция не может быть произведена, так как системе не хватает свободного места в буфере.

10056-WSAEISCONN
С сокетом уже установлено соединение. Попытка установить соединение с сокетом, который уже используется. Ошибка может произойти как на дейтаграммных, так и на потоковых сокетах. При использовании дейтаграммного сокета ошибка возникает при вызове sendto или WSASendTo,если для установления дейтаграммного соединения вызывались функции connect или WSAConnect.

10057-WSAENOTCONN
Нет соединения с сокетом. Производится запрос на отправление или получение данных на сокете, соединение с которым в данный момент неустановлено.

10058-WSAESHUTDOWN
Отправление невозможно после отключения сокета. Сокет уже частично закрыт вызовом функции shutdown, но к нему делается запрос на отправление или получение данных. Ошибка возникает только на отключенных направлениях потока данных, например, при попытке отправить данные после вызова shutdown.

10060-WSAETIMEDOUT
Время ожидания операции истекло. Сделан запрос на соединение, но удаленный компьютер не отвечает должным образом (или вообще не отвечает) по истечении определенного промежутка времени. Ошибка обычно возникает, когда заданы параметры сокета SO_SNDTIMEO и SO_RCVTIMEO

10061-WSAECONNREFUSED
В соединении отказано. Компьютер-адресат отказывает в установлении соединения, обычно из-за того, что на удаленной машине нет приложения, обслуживающего соединение с указанным адресом.

10064-WSAEHOSTDOWN
Узел не работает. Попытка выполнить операцию не удалась, так как узел назначения выключен. Тем не менее, приложение, скорее всего, получит ошибку WSAETIMEDOUT, обычную при попытке установить соединение.

10065-WSAEHOSTUNREACH
Не известен путь к узлу. Был произведен запрос на выполнение операции на недоступном узле. Эта ошибка похожа на WSAENETUNREACH.

10067-WSAEPROCLIM
Слишком много процессов. Некоторые поставщики служб Winsoc задают предел количества процессов, которые могут обращаться к ним одновременно.

10091 - WSASYSNOTREADY
Подсистема сети не доступна. Ошибка возвращается при обращении к WSAStartup, когда поставщик службы не может отработать из-за того, что базовая система, предоставляющая услуги, не доступна.

10092-WSAVERNOTSUPPORTED
Некорректная версия Winsock.dll. Запрошенная версия поставщика службы Winsock не поддерживается.

10093-WSANOTINITIALISED
Winsock не инициализирован. Вызов WSAStartup не удался.

10101 - WSAEDISCON
Идет корректное завершение работы. Ошибка возвращается функциями WSARecv и WSARecvFrom, указывая, что удаленный партнер находитсяв процессе корректного завершения работы. Возникает при использовании протоколов, ориентированных на передачу сообщений, таких, как ATM.

10109-WSATYPE_NOT_FOUND
Не найден тип класса. Ошибка также относится к функциям регистрации разрешения имен. Когда регистрируется экземпляр службы, то он должен ссылаться на класс службы, заданный ранее функцией WSAInsta/lServiceClass.

11001-WSAHOST_NOT_FOUND
Узел не найден. Ошибку возвращают gethostbyname и gethostbyaddr, указывая, что полномочный узел не найден

11002-WSATRY_AGAlN
Неполномочный узел не найден. Не был найден неполномочный узел, либо произошла ошибка в работе сервера. Ошибка связана с gethostbyname и gethostbyaddr.

11003-WSANO_RECOVERY
Произошла неустранимая ошибка. Ошибка связана с gethostbyname и gethostbyaddr. Следует попытаться выполнить операцию еще раз.

11004-WSANO_DATA
Не найдено записей данных запрошенного типа. Не найдено записей данных запрошенного типа, хотя заданное имя было верным. Ошибка связана с gethostbyname и gethostbyaddr.

11005 - WSA_QOS RECEIVERS
Получено минимум одно сообщение резервирования. Минимум один процесс в сети хочет получать трафик Quality of Service (QoS). Значение связано с реализацией QoS в IP и, по сути, ошибкой не является

11006-WSA_QOS_SENDERS
Получено минимум одно сообщение пути. Минимум один процессов сети хочет отправлять трафик QoS. Значение является сообщением о состоянии.

11007 -WSA_QOS_NO_SENDERS
Нет отправителей QoS. Более не существует процессов, которые хотели бы отправлять данные QoS

11008-WSA_QOS_NO_RECEIVERS
Нет получателей QoS. Более не существует процессов, которые хотели бы получать данные QoS

11009-WSA_QOS_REQUEST_CONFIRMED
Запрос на резервирование подтвержден. Утвердительный ответ на запрос о резервировании пропускной способности, который могут выдавать приложения QoS

11010-WSA_QOS_ADMISSION_FAILURE
Ошибка из-за недостаточности ресурсов. Не хватило ресурсов для удовлетворения запроса на пропускную способность QoS.

11011 -WSA_QOS POLICYFAILURE
Неверные реквизиты. Пользователь не владеет необходимыми полномочиями, или заданные реквизиты не позволяют выполнить запрос резервирования QoS.

11012-WSA_QOS_BAD_STYLE
Неизвестный или противоречивый стиль. Приложения QoS могут задавать разные стили фильтрования для данного сеанса

11013 - WSA_QOS_BAD_OBJECT
Неверная структура FILTERSPEC или специфичный для поставщика объект. Ошибка в структуре FILTERSPEC или специфичных для поставщика буферах объекта QoS

11014-WSA_QOS_TRAFFIC_CTRL_ERROR
Проблемы с FLOWSPEC. У компонента управления трафиком появиласьпроблема с параметрами FLOWSPEC, переданными в составе объекта QoS.

11015-WSA_QOS_GENERIC_ERROR
Общая ошибка QoS. Универсальная ошибка, возвращаемая, когда неприменимы другие ошибки QoS.

OS dependent -WSA_INVALID_HANDLE
Указан неверный объект события. Функции WSAWaitForMultipleEvents
передан неверный описатель. Ошибка Win32, появляется при использованиифункций Winsock, соответствующих функциям Win32.

OS dependent -WSA_NOT_ENOUGH_MEMORY
Не достаточно памяти. Для выполнения операции не хватает свободной памяти, ошибка Win32.

OS dependent - WSA_INVALID_PARAMETER
Один или несколько неверных параметров. Функции передан неверный параметр, ошибка Win32. Также возникает при работе WSAWaitForMultipleEvents, когда задан неверный параметр счетчика событий.

OS dependent - WSA_WAIT_TIMEOUT
Таймаут операции истек. Перекрытая операция не завершилась в положенное время, ошибка Win32.

OS dependent -WSA_OPERATION_ABORTED
Перекрытая операция прервана. Перекрытая операция ввода-вывода отменена из-за закрытия сокета, ошибка Win32. Также возникает при выполнении команды SIOFLUSH.

OS dependent - WSA_IO_INCOMPLETE
Объект события для перекрытого ввода-вывода не свободен. При вызове WSAGetOverlappedResults перекрытая операция ввода-вывода не завершена, ошибка Win32.

OS dependent -WSA_IO_PENDING
Перекрытая операция завершится позже. При перекрытом вызове ввода-вывода операция отложена и завершится позже, ошибка Win32

Курсовая работа

Все программы должны быть выполнены как консольные приложения. Параметры работы программ задаются через командную строку. Необходимо проверять коды возврата всех функций Winsock и Win32 API. При ошибке следует выводить сообщение об ошибке и код ошибки.

Программы должны выводить на консоль текущую и среднюю скорость обмена информацией (см. работу 2).

Программы должны корректно завершать свою работу при их завершении любым способом: закрытии консольного окна или нажатии комбинации клавиш Ctrl+C или Ctrl+Break. (см. работу 5). Корректное завершение программы предполагает, что программа завершает чтение всех непринятых данных из сети, закрывает все сокеты и освобождает зарезервированную память и другие ресурсы. Для корректного закрытия сокетов используйте сетевое событие FD_CLOSE.

Серверная программа перед закрытием должна уведомить о своём закрытии клиентов и дождаться пока закроются сокеты клиентов. Только тогда она может начать корректно завершать свою работу. Клиенты должны быть готовы принять от сервера извещение о его закрытии.

Необходимое условие корректности завершения прогамм состоит в отсутствии ошибок при вызове функций Winsock и Win32 API

Студент выполняет по одному варианту из следующих трёх наборов вариантов. Номера вариантов выдаются преподавателем.

Синхронная модель ввода-вывода в вариантах предполагает использовние функций readn и readn из работы 2.

В моделях ввода-вывода портов завершения в вариантах предполагает использовние функций PostQueuedCompletionStatus, которая представлена в работе 6.

Демонстрация работоспособности программ производится путём запуска сервера на одной машине и нескольких клиентов на другом компьютере

1. Эхо-сервер пакетов переменной длины

Создайте консольные приложения эхо-сервера и эхо-клиента для пакетов переменной длины**.** Формат пакета: заголовок затем данные. Заголовок содержит целое число (4 байта), в котором содержится количество байт в данных в пакете. Длина и содержимое пакета определяется клиентом случайным образом. Клиент должен отсылать серверу пакеты либо непрерывно, либо по одному. Клиент проверяет переданные и принятые данные на совпадение.

	Клиент		Сервер	
	Модель	Реакция на FD_CLOSE	Модель	Реакция на FD_CLOSE
1	Синхронная	WSAAsyncSelect	Синхронная	WSAAsyncSelect
2	Синхронная	WSAAsyncSelect	Select	WSAAsyncSelect
3	Синхронная	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
4	Синхронная	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
5	Синхронная	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
6	Синхронная	WSAAsyncSelect	функции завершения	WSAAsyncSelect
7	Синхронная	WSAAsyncSelect	порты завершения	WSAAsyncSelect
8	WSAAsyncSelect	WSAAsyncSelect	Синхронная	WSAAsyncSelect
9	WSAAsyncSelect	WSAAsyncSelect	Select	WSAAsyncSelect
10	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
11	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
12	WSAAsyncSelect	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
13	WSAAsyncSelect	WSAAsyncSelect	функции завершения	WSAAsyncSelect
14	WSAAsyncSelect	WSAAsyncSelect	порты завершения	WSAAsyncSelect
15	WSAEventSelect	WSAAsyncSelect	Синхронная	WSAAsyncSelect
16	WSAEventSelect	WSAAsyncSelect	Select	WSAAsyncSelect
17	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
18	WSAEventSelect	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
19	WSAEventSelect	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
20	WSAEventSelect	WSAAsyncSelect	функции завершения	WSAAsyncSelect
21	WSAEventSelect	WSAAsyncSelect	порты завершения	WSAAsyncSelect
22	Синхронная	WSAAsyncSelect	Синхронная	WSAEventSelect
23	Синхронная	WSAAsyncSelect	Select	WSAEventSelect
24	Синхронная	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect

25	Синхронная	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
26	Синхронная	WSAAsyncSelect	перекрытия и события	WSAEventSelect
27	Синхронная	WSAAsyncSelect	функции завершения	WSAEventSelect
28	Синхронная	WSAAsyncSelect	порты завершения	WSAEventSelect
29	WSAAsyncSelect	WSAAsyncSelect	Синхронная	WSAEventSelect
30	WSAAsyncSelect	WSAAsyncSelect	Select	WSAEventSelect
31	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
32	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
33	WSAAsyncSelect	WSAAsyncSelect	перекрытия и события	WSAEventSelect
34	WSAAsyncSelect	WSAAsyncSelect	функции завершения	WSAEventSelect
35	WSAAsyncSelect	WSAAsyncSelect	порты завершения	WSAEventSelect
36	WSAEventSelect	WSAAsyncSelect	Синхронная	WSAEventSelect
37	WSAEventSelect	WSAAsyncSelect	Select	WSAEventSelect
38	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
39	WSAEventSelect	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
40	WSAEventSelect	WSAAsyncSelect	перекрытия и события	WSAEventSelect
41	WSAEventSelect	WSAAsyncSelect	функции завершения	WSAEventSelect
42	WSAEventSelect	WSAAsyncSelect	порты завершения	WSAEventSelect
43	Синхронная	WSAEventSelect	Синхронная	WSAAsyncSelect
44	Синхронная	WSAEventSelect	Select	WSAAsyncSelect
45	Синхронная	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
46	Синхронная	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
47	Синхронная	WSAEventSelect	перекрытия и события	WSAAsyncSelect
48	Синхронная	WSAEventSelect	функции завершения	WSAAsyncSelect
49	Синхронная	WSAEventSelect	порты завершения	WSAAsyncSelect
50	WSAAsyncSelect	WSAEventSelect	Синхронная	WSAAsyncSelect
51	WSAAsyncSelect	WSAEventSelect	Select	WSAAsyncSelect
52	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
53	WSAAsyncSelect	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
54	WSAAsyncSelect	WSAEventSelect	перекрытия и события	WSAAsyncSelect
55	WSAAsyncSelect	WSAEventSelect	функции завершения	WSAAsyncSelect
56	WSAAsyncSelect	WSAEventSelect	порты завершения	WSAAsyncSelect
57	WSAEventSelect	WSAEventSelect	Синхронная	WSAAsyncSelect
58	WSAEventSelect	WSAEventSelect	Select	WSAAsyncSelect
59	WSAEventSelect	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
60	WSAEventSelect	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
61	WSAEventSelect	WSAEventSelect	перекрытия и события	WSAAsyncSelect
62	WSAEventSelect	WSAEventSelect	функции завершения	WSAAsyncSelect
63	WSAEventSelect	WSAEventSelect	порты завершения	WSAAsyncSelect
64	Синхронная	WSAEventSelect	Синхронная	WSAEventSelect
65	Синхронная	WSAEventSelect	Select	WSAEventSelect
66	Синхронная	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
67	Синхронная	WSAEventSelect	WSAEventSelect	WSAEventSelect
68	Синхронная	WSAEventSelect	перекрытия и события	WSAEventSelect
69	Синхронная	WSAEventSelect	функции завершения	WSAEventSelect
70	Синхронная	WSAEventSelect	порты завершения	WSAEventSelect
71	WSAAsyncSelect	WSAEventSelect	Синхронная	WSAEventSelect
72	WSAAsyncSelect	WSAEventSelect	Select	WSAEventSelect
73	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
74	WSAAsyncSelect	WSAEventSelect	WSAEventSelect	WSAEventSelect
75	WSAAsyncSelect	WSAEventSelect	перекрытия и события	WSAEventSelect
76	WSAAsyncSelect	WSAEventSelect	функции завершения	WSAEventSelect
77	WSAAsyncSelect	WSAEventSelect	порты завершения	WSAEventSelect
78	WSAEventSelect	WSAEventSelect	Синхронная	WSAEventSelect
79	WSAEventSelect	WSAEventSelect	Select	WSAEventSelect
80	WSAEventSelect	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
81	WSAEventSelect	WSAEventSelect	WSAEventSelect	WSAEventSelect
82	WSAEventSelect	WSAEventSelect	перекрытия и события	WSAEventSelect
83	WSAEventSelect	WSAEventSelect	функции завершения	WSAEventSelect
84	WSAEventSelect	WSAEventSelect	порты завершения	WSAEventSelect

2. Клиент и сервер HTTP

Следуя работе 3, напишите клиент и сервер для передачи файлов с помощью протокола HTTP. Файл должен быть такой длины, чтобы можно было продемонстрировать корректность закрытия вашего клиента и сервера. Запросите у HTTP-сервера файл с помощью HTTP-клиента. Сравните полученный файл с исходным файлом на сервере.

Сервер HTTP должен читать файл с помощью функции ReadFile не сразу, а частями и отправлял их клиенту.

	Клиент		Сервер	
	Модель	Реакция на FD_CLOSE	Модель	Реакция на FD_CLOSE
1	Синхронная	WSAAsyncSelect	Синхронная	WSAAsyncSelect
2	Синхронная	WSAAsyncSelect	Select	WSAAsyncSelect
3	Синхронная	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
4	Синхронная	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
5	Синхронная	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
6	Синхронная	WSAAsyncSelect	функции завершения	WSAAsyncSelect
7	Синхронная	WSAAsyncSelect	порты завершения	WSAAsyncSelect
8	Синхронная	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
9	WSAAsyncSelect	WSAAsyncSelect	Синхронная	WSAAsyncSelect
10	WSAAsyncSelect	WSAAsyncSelect	Select	WSAAsyncSelect
11	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
12	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
13	WSAAsyncSelect	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
14	WSAAsyncSelect	WSAAsyncSelect	функции завершения	WSAAsyncSelect
15	WSAAsyncSelect	WSAAsyncSelect	порты завершения	WSAAsyncSelect
16	WSAAsyncSelect	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
17	WSAEventSelect	WSAAsyncSelect	Синхронная	WSAAsyncSelect
18	WSAEventSelect	WSAAsyncSelect	Select	WSAAsyncSelect
19	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
20	WSAEventSelect	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
21	WSAEventSelect	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
22	WSAEventSelect	WSAAsyncSelect	функции завершения	WSAAsyncSelect
23	WSAEventSelect	WSAAsyncSelect	порты завершения	WSAAsyncSelect
24	WSAEventSelect	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
25	Синхронная	WSAAsyncSelect	Синхронная	WSAEventSelect
26	Синхронная	WSAAsyncSelect	Select	WSAEventSelect
27	Синхронная	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
28	Синхронная	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
2	Синхронная	WSAAsyncSelect	перекрытия и события	WSAEventSelect

9				
30	Синхронная	WSAAsyncSelect	функции завершения	WSAEventSelect
31	Синхронная	WSAAsyncSelect	порты завершения	WSAEventSelect
32	Синхронная	WSAAsyncSelect	TransmitFile	WSAEventSelect
33	WSAAsyncSelect	WSAAsyncSelect	Синхронная	WSAEventSelect
34	WSAAsyncSelect	WSAAsyncSelect	Select	WSAEventSelect
35	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
36	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
37	WSAAsyncSelect	WSAAsyncSelect	перекрытия и события	WSAEventSelect
38	WSAAsyncSelect	WSAAsyncSelect	функции завершения	WSAEventSelect
39	WSAAsyncSelect	WSAAsyncSelect	порты завершения	WSAEventSelect
40	WSAAsyncSelect	WSAAsyncSelect	TransmitFile	WSAEventSelect
41	WSAEventSelect	WSAAsyncSelect	Синхронная	WSAEventSelect
42	WSAEventSelect	WSAAsyncSelect	Select	WSAEventSelect
43	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
44	WSAEventSelect	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
45	WSAEventSelect	WSAAsyncSelect	перекрытия и события	WSAEventSelect
46	WSAEventSelect	WSAAsyncSelect	функции завершения	WSAEventSelect
47	WSAEventSelect	WSAAsyncSelect	порты завершения	WSAEventSelect
48	WSAEventSelect	WSAAsyncSelect	TransmitFile	WSAEventSelect
49	Синхронная	WSAEventSelect	Синхронная	WSAAsyncSelect
50	Синхронная	WSAEventSelect	Select	WSAAsyncSelect
51	Синхронная	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
52	Синхронная	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
53	Синхронная	WSAEventSelect	перекрытия и события	WSAAsyncSelect
54	Синхронная	WSAEventSelect	функции завершения	WSAAsyncSelect
55	Синхронная	WSAEventSelect	порты завершения	WSAAsyncSelect
56	Синхронная	WSAEventSelect	TransmitFile	WSAAsyncSelect
57	WSAAsyncSelect	WSAEventSelect	Синхронная	WSAAsyncSelect
58	WSAAsyncSelect	WSAEventSelect	Select	WSAAsyncSelect
59	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect

60	WSAAsyncSelect	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
61	WSAAsyncSelect	WSAEventSelect	перекрытия и события	WSAAsyncSelect
62	WSAAsyncSelect	WSAEventSelect	функции завершения	WSAAsyncSelect
63	WSAAsyncSelect	WSAEventSelect	порты завершения	WSAAsyncSelect
64	WSAAsyncSelect	WSAEventSelect	TransmitFile	WSAAsyncSelect
65	WSAEventSelect	WSAEventSelect	Синхронная	WSAAsyncSelect
66	WSAEventSelect	WSAEventSelect	Select	WSAAsyncSelect
67	WSAEventSelect	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
68	WSAEventSelect	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
69	WSAEventSelect	WSAEventSelect	перекрытия и события	WSAAsyncSelect
70	WSAEventSelect	WSAEventSelect	функции завершения	WSAAsyncSelect
71	WSAEventSelect	WSAEventSelect	порты завершения	WSAAsyncSelect
72	WSAEventSelect	WSAEventSelect	TransmitFile	WSAAsyncSelect
73	Синхронная	WSAEventSelect	Синхронная	WSAEventSelect
74	Синхронная	WSAEventSelect	Select	WSAEventSelect
75	Синхронная	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
76	Синхронная	WSAEventSelect	WSAEventSelect	WSAEventSelect
77	Синхронная	WSAEventSelect	перекрытия и события	WSAEventSelect
78	Синхронная	WSAEventSelect	функции завершения	WSAEventSelect
79	Синхронная	WSAEventSelect	порты завершения	WSAEventSelect
80	Синхронная	WSAEventSelect	TransmitFile	WSAEventSelect
81	WSAAsyncSelect	WSAEventSelect	Синхронная	WSAEventSelect
82	WSAAsyncSelect	WSAEventSelect	Select	WSAEventSelect
83	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
84	WSAAsyncSelect	WSAEventSelect	WSAEventSelect	WSAEventSelect
85	WSAAsyncSelect	WSAEventSelect	перекрытия и события	WSAEventSelect
86	WSAAsyncSelect	WSAEventSelect	функции завершения	WSAEventSelect
87	WSAAsyncSelect	WSAEventSelect	порты завершения	WSAEventSelect
88	WSAAsyncSelect	WSAEventSelect	TransmitFile	WSAEventSelect
89	WSAEventSelect	WSAEventSelect	Синхронная	WSAEventSelect
9	WSAEventSelect	WSAEventSelect	Select	WSAEventSelect

0				
91	WSAEventSelect	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
92	WSAEventSelect	WSAEventSelect	WSAEventSelect	WSAEventSelect
93	WSAEventSelect	WSAEventSelect	перекрытия и события	WSAEventSelect
94	WSAEventSelect	WSAEventSelect	функции завершения	WSAEventSelect
95	WSAEventSelect	WSAEventSelect	порты завершения	WSAEventSelect
96	WSAEventSelect	WSAEventSelect	TransmitFile	WSAEventSelect

3. **Примитивный протокол для передачи файлов**

Разработаем примитивный протокол PFTP (Primitive File Trnsfer Protocol) для обмена файлами. Пусть PFTP-сервер расположен по адресу host и слушает порт 5150.

PFTP-клиент может забрать у PFTP-сервера файл file и поместить его под тем же именем в свою рабочую папку. Запрашиваемые файлы должны находиться в рабочей папке сервера

PFTPclient get host 5150 file

PFTP-клиент PFTPclien может отправить PFTP-серверу файл file из своей рабочей папки с помощью команды

PFTPclient put host 5150 file

PFTP-сервер принимает файл и записывает его под тем же именем в свою рабочую папку.

Рекомендуемый формат пакетов PFTP-протокола:

GET:
Заголовок get-запроса от клиента к серверу содержит имя запрашиваемого файла, например
“Get:file”

Заголовок get-ответа сервера на get-запрос:
“getresponce:file\r\n\r\n”
За заголовком следует содержимое файла file.
Если файла нет или случилась ошибка, то сервер отвечает
“geterror:file”

PUT:
Заголовок put-запроса от клиента к серверу содержит имя отправляемого файла, например
“put:file\r\n\r\n”
За заголовком следует содержимое файла file.

Заголовок put-ответа сервера на put-запрос можно предложить следующего вида
“Putresponce:file”
Если сервер не смог принять файла, то сервер отвечает
“Puterror:file”

Файл должен быть такой длины, чтобы можно было продемонстрировать корректность закрытия вашего клиента и сервера.

Поместите несколько файлов в рабочую папку PFTP-сервера. Запустите на разных компьютерах несколько PFTP-клиентов на приём и передачу. Сравните переданные и полученные файлы.

Для клиента и сервера модель ввода-вывода TransmitFile для приёма файлов совпадает с синхронной моделью

	Клиент		Сервер	
	Модель	Реакция на FD_CLOSE	Модель	Реакция на

				FD_CLOSE
1	Синхронная	WSAAsyncSelect	Синхронная	WSAAsyncSelect
2	Синхронная	WSAAsyncSelect	Select	WSAAsyncSelect
3	Синхронная	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
4	Синхронная	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
5	Синхронная	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
6	Синхронная	WSAAsyncSelect	функции завершения	WSAAsyncSelect
7	Синхронная	WSAAsyncSelect	порты завершения	WSAAsyncSelect
8	Синхронная	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
9	WSAAsyncSelect	WSAAsyncSelect	Синхронная	WSAAsyncSelect
10	WSAAsyncSelect	WSAAsyncSelect	Select	WSAAsyncSelect
11	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
12	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
13	WSAAsyncSelect	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
14	WSAAsyncSelect	WSAAsyncSelect	функции завершения	WSAAsyncSelect
15	WSAAsyncSelect	WSAAsyncSelect	порты завершения	WSAAsyncSelect
16	WSAAsyncSelect	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
17	WSAEventSelect	WSAAsyncSelect	Синхронная	WSAAsyncSelect
18	WSAEventSelect	WSAAsyncSelect	Select	WSAAsyncSelect
19	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
20	WSAEventSelect	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
21	WSAEventSelect	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
22	WSAEventSelect	WSAAsyncSelect	функции завершения	WSAAsyncSelect
23	WSAEventSelect	WSAAsyncSelect	порты завершения	WSAAsyncSelect
24	WSAEventSelect	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
25	TransmitFile	WSAAsyncSelect	Синхронная	WSAAsyncSelect
26	TransmitFile	WSAAsyncSelect	Select	WSAAsyncSelect
27	TransmitFile	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect
28	TransmitFile	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect
29	TransmitFile	WSAAsyncSelect	перекрытия и события	WSAAsyncSelect
30	TransmitFile	WSAAsyncSelect	функции завершения	WSAAsyncSelect
31	TransmitFile	WSAAsyncSelect	порты завершения	WSAAsyncSelect
32	TransmitFile	WSAAsyncSelect	TransmitFile	WSAAsyncSelect
33	Синхронная	WSAAsyncSelect	Синхронная	WSAEventSelect
34	Синхронная	WSAAsyncSelect	Select	WSAEventSelect
35	Синхронная	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
36	Синхронная	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
37	Синхронная	WSAAsyncSelect	перекрытия и события	WSAEventSelect
38	Синхронная	WSAAsyncSelect	функции завершения	WSAEventSelect
39	Синхронная	WSAAsyncSelect	порты завершения	WSAEventSelect
40	Синхронная	WSAAsyncSelect	TransmitFile	WSAEventSelect
41	WSAAsyncSelect	WSAAsyncSelect	Синхронная	WSAEventSelect
42	WSAAsyncSelect	WSAAsyncSelect	Select	WSAEventSelect
43	WSAAsyncSelect	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
44	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
45	WSAAsyncSelect	WSAAsyncSelect	перекрытия и события	WSAEventSelect
46	WSAAsyncSelect	WSAAsyncSelect	функции завершения	WSAEventSelect
47	WSAAsyncSelect	WSAAsyncSelect	порты завершения	WSAEventSelect
48	WSAAsyncSelect	WSAAsyncSelect	TransmitFile	WSAEventSelect
49	WSAEventSelect	WSAAsyncSelect	Синхронная	WSAEventSelect
50	WSAEventSelect	WSAAsyncSelect	Select	WSAEventSelect
51	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
52	WSAEventSelect	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
53	WSAEventSelect	WSAAsyncSelect	перекрытия и события	WSAEventSelect
54	WSAEventSelect	WSAAsyncSelect	функции завершения	WSAEventSelect
55	WSAEventSelect	WSAAsyncSelect	порты завершения	WSAEventSelect
56	WSAEventSelect	WSAAsyncSelect	TransmitFile	WSAEventSelect
57	TransmitFile	WSAAsyncSelect	Синхронная	WSAEventSelect
58	TransmitFile	WSAAsyncSelect	Select	WSAEventSelect

59	TransmitFile	WSAAsyncSelect	WSAAsyncSelect	WSAEventSelect
60	TransmitFile	WSAAsyncSelect	WSAEventSelect	WSAEventSelect
61	TransmitFile	WSAAsyncSelect	перекрытия и события	WSAEventSelect
62	TransmitFile	WSAAsyncSelect	функции завершения	WSAEventSelect
63	TransmitFile	WSAAsyncSelect	порты завершения	WSAEventSelect
64	TransmitFile	WSAAsyncSelect	TransmitFile	WSAEventSelect
65	Синхронная	WSAEventSelect	Синхронная	WSAAsyncSelect
66	Синхронная	WSAEventSelect	Select	WSAAsyncSelect
67	Синхронная	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
68	Синхронная	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
69	Синхронная	WSAEventSelect	перекрытия и события	WSAAsyncSelect
70	Синхронная	WSAEventSelect	функции завершения	WSAAsyncSelect
71	Синхронная	WSAEventSelect	порты завершения	WSAAsyncSelect
72	Синхронная	WSAEventSelect	TransmitFile	WSAAsyncSelect
73	WSAAsyncSelect	WSAEventSelect	Синхронная	WSAAsyncSelect
74	WSAAsyncSelect	WSAEventSelect	Select	WSAAsyncSelect
75	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
76	WSAAsyncSelect	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
77	WSAAsyncSelect	WSAEventSelect	перекрытия и события	WSAAsyncSelect
78	WSAAsyncSelect	WSAEventSelect	функции завершения	WSAAsyncSelect
79	WSAAsyncSelect	WSAEventSelect	порты завершения	WSAAsyncSelect
80	WSAAsyncSelect	WSAEventSelect	TransmitFile	WSAAsyncSelect
81	WSAEventSelect	WSAEventSelect	Синхронная	WSAAsyncSelect
82	WSAEventSelect	WSAEventSelect	Select	WSAAsyncSelect
83	WSAEventSelect	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
84	WSAEventSelect	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
85	WSAEventSelect	WSAEventSelect	перекрытия и события	WSAAsyncSelect
86	WSAEventSelect	WSAEventSelect	функции завершения	WSAAsyncSelect
87	WSAEventSelect	WSAEventSelect	порты завершения	WSAAsyncSelect
88	WSAEventSelect	WSAEventSelect	TransmitFile	WSAAsyncSelect
89	TransmitFile	WSAEventSelect	Синхронная	WSAAsyncSelect
90	TransmitFile	WSAEventSelect	Select	WSAAsyncSelect
91	TransmitFile	WSAEventSelect	WSAAsyncSelect	WSAAsyncSelect
92	TransmitFile	WSAEventSelect	WSAEventSelect	WSAAsyncSelect
93	TransmitFile	WSAEventSelect	перекрытия и события	WSAAsyncSelect
94	TransmitFile	WSAEventSelect	функции завершения	WSAAsyncSelect
95	TransmitFile	WSAEventSelect	порты завершения	WSAAsyncSelect
96	TransmitFile	WSAEventSelect	TransmitFile	WSAAsyncSelect
97	Синхронная	WSAEventSelect	Синхронная	WSAEventSelect
98	Синхронная	WSAEventSelect	Select	WSAEventSelect
99	Синхронная	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
100	Синхронная	WSAEventSelect	WSAEventSelect	WSAEventSelect
101	Синхронная	WSAEventSelect	перекрытия и события	WSAEventSelect
102	Синхронная	WSAEventSelect	функции завершения	WSAEventSelect
103	Синхронная	WSAEventSelect	порты завершения	WSAEventSelect
104	Синхронная	WSAEventSelect	TransmitFile	WSAEventSelect
105	WSAAsyncSelect	WSAEventSelect	Синхронная	WSAEventSelect
106	WSAAsyncSelect	WSAEventSelect	Select	WSAEventSelect
107	WSAAsyncSelect	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
108	WSAAsyncSelect	WSAEventSelect	WSAEventSelect	WSAEventSelect
109	WSAAsyncSelect	WSAEventSelect	перекрытия и события	WSAEventSelect
110	WSAAsyncSelect	WSAEventSelect	функции завершения	WSAEventSelect
111	WSAAsyncSelect	WSAEventSelect	порты завершения	WSAEventSelect
112	WSAAsyncSelect	WSAEventSelect	TransmitFile	WSAEventSelect
113	WSAEventSelect	WSAEventSelect	Синхронная	WSAEventSelect
114	WSAEventSelect	WSAEventSelect	Select	WSAEventSelect
115	WSAEventSelect	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
116	WSAEventSelect	WSAEventSelect	WSAEventSelect	WSAEventSelect
117	WSAEventSelect	WSAEventSelect	перекрытия и события	WSAEventSelect
118	WSAEventSelect	WSAEventSelect	функции завершения	WSAEventSelect

119	WSAEventSelect	WSAEventSelect	порты завершения	WSAEventSelect
120	WSAEventSelect	WSAEventSelect	TransmitFile	WSAEventSelect
121	TransmitFile	WSAEventSelect	Синхронная	WSAEventSelect
122	TransmitFile	WSAEventSelect	Select	WSAEventSelect
123	TransmitFile	WSAEventSelect	WSAAsyncSelect	WSAEventSelect
124	TransmitFile	WSAEventSelect	WSAEventSelect	WSAEventSelect
125	TransmitFile	WSAEventSelect	перекрытия и события	WSAEventSelect
126	TransmitFile	WSAEventSelect	функции завершения	WSAEventSelect
127	TransmitFile	WSAEventSelect	порты завершения	WSAEventSelect
128	TransmitFile	WSAEventSelect	TransmitFile	WSAEventSelect