В.В. Новый

Компьютерные сети. Использование библиотеки Windows Sockets 2

Практикум по дисциплине «Компьютерные сети» для студентов 3 курса специальности Прикладная математика (1-31 03 03)

2010

Лабораторная работа № 9
(2 часа)

Разрешение имен

Задание. Напишите программу, выполняющую разрешение имен для введенных пользователем данных. При вводе пользователем символьного имени оно должно разрешаться в IP-адрес, при вводе корректного IP-адреса – в символьное имя.

API Winsock

Наиболее распространенным API для работы с сетью в Windows является сетевой интерфейс **Windows Sockets** или сокращенно **Winsock**. Интерфейс **Winsock** унаследовал многие возможности от реализации BSD Sockets для платформы UNIX. В настоящее время актуальной версией API является **Winsock 2**. Название API происходит от основного понятия – сокета.

Основной заголовочный файл **Winsock** – `Winsock2.h`. Кроме того, для использования API к вашему проекту должна быть подключена библиотека Ws2_32.lib.

Перед вызовом любой функции **Winsock** необходимо загрузить правильную версию библиотеки **Winsock**. Для инициализации **Winsock** используется функция `WSAStartup`:

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

здесь **`wVersionRequested`** задает версию библиотеки Winsock, которую требуется загрузить. На современных платформах используется версия 2.2 (для ее загрузки укажите значение **`0x0202`** или макрос **`MAKEWORD(2,2)`** – старший байт задает ревизию библиотеки, младший байт – версию библиотеки);

`lpWSAData` – указатель на структуру, возвращаемую по завершению вызова и содержащую информацию о загруженной версии **Winsock**:

```
typedef struct WSAData {
WORD wVersion;
WORD wHighVersion;
char szDescription[WSADESCRIPTION_LEN+1];
char szSystemStatus[WSASYS_STATUS_LEN+1];
unsigned short iMaxSockets;
unsigned short iMaxUdpDg;
char FAR* lpVendorInfo;
} WSADATA, *LPWSADATA;
```

Наиболее полезная информация, получаемая из этой структуры – **wVersion** и **wHighVersion**: версия **Winsock**, которую предлагает использовать вызов и высшая версия, поддерживаемая библиотекой.

После завершения работы с библиотекой **Winsock** нужно выгрузить библиотеку и освободить ресурсы с помощью функции **WSACleanup**:

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

Каждому вызову **WSAStartup()** должен соответствовать вызов **WSACleanup()**.

Обе функции возвращают **0** в случае успешного завершения или код ошибки в противном случае.

Для подключения к узлу по протоколу IP **Winsock**-приложение должно знать IP адрес этого узла. При этом может потребоваться преобразование адреса из символьного вида в IP-адрес – разрешение имени. В **Winsock** предусмотрено две функции для этого: **gethostbyname** и **WSAAsyncGetHostByName**. Эти функции отыскивают в базе данных сведения об узле, соответствующие его имени. Обе функции возвращают структуру **HOSTENT**:

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

Поле **h_name** задает официальное имя узла. Если в сети используется DNS в качестве имени будет возвращено полное имя домена (FQDN). Если в сети применяется локальный файл узлов (hosts, lmhosts) – это первая запись после IP-адреса.
Поле **h_aliases** задает массив дополнительных имен узла.
Поле **h_addrtype** задает семейство возвращаемых адресов.
Поле **h_length** задает длину в байтах каждого адреса из поля **h_addr_list**.
Поле **h_addr_list** задает массив, содержащий IP-адреса узла (узел может иметь несколько IP-адресов). Каждый адрес в массиве задан в сетевом порядке.

Функция **gethostbyname** определена так:

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

Параметр **name** задает дружественное имя искомого узла. При успешном выполнении функция возвращает указатель на структуру **HOSTENT**, которая хранится в системной памяти. Поскольку эта память

обслуживается системой приложение не должно освобождать возвращаемую структуру.

WSAAsyncGetHostByName – асинхронная версия функции **gethostbyname**, оповещающая приложение о завершении своего выполнения с помощью сообщений Windows:

```
HANDLE WSAAsyncGetHostByName(
  HWND hWnd,
  unsigned int wMsg,
  const char* name,
  char* buf,
  int buflen
);
```

Параметр **hWnd** задает дескриптор окна, которое получит сообщение о выполнении асинхронного запроса, параметр **wMsg** задает сообщение Windows, которое будет возвращено по завершении выполнения запроса. Параметр name – дружественное имя узла. Параметр **buf** – указатель на область памяти, куда помещается **HOSTENT**. Этот буфер должен иметь размер **BMAXGETHOSTSTRUCT** (параметр **buflen**).

Обратное преобразование – поиск по известному IP-адресу понятного пользователю имени выполняется с помощью функции **gethostbyaddr** и **WSAAsyncGetHostByAddr**.

Функция **gethostbyaddr** определена так:

```
struct HOSTENT* FAR gethostbyaddr(
  const char* addr,
  int len,
  int type
);
```

Параметр **addr** указывает на IP-адрес в сетевом порядке, параметр **len** задает длину параметра **addr** в байтах. Параметр **type** должен иметь значение **AF_INET** соответствующее протоколу IP и ассоциированным с ним TCP и UDP).

Функция **WSAAsyncGetHostByAddr** определена как:

```
HANDLE WSAAsyncGetHostByAddr(
  HWND hWnd,
  unsigned int wMsg,
  const char* addr,
```

```
  int len,
  int type,
  char* buf,
  int buflen
);
```

и является асинхронной версией **gethostbyaddr**.

Смысл параметров **hWnd**, **wMsg**, **buf**, **buflen** аналогичен их назначению для функции **WSAAsyncGetHostByName**. Параметры **addr** и **len** аналогичны вызову **gethostbyaddr**.

Указание 1. Разные процессоры в зависимости от конструкции представляют числа в одном из двух порядков байт **big-** или **little-endian**. Например, процессоры Intel представляют многобайтные числа следуя от менее значимого к более значимому байту (little-endian). Когда номер порта и IP-адрес хранятся в памяти компьютера они представляются в ***системном порядке*** (**host-byte-order**). Стандарты Интернета требуют, чтобы многобайтные числа передавались в от старшего к младшему байту (big-endian), что обычно называется ***сетевым порядком*** (**network-byte-order**).
Есть целый ряд функции для преобразования многобайтных чисел из системного порядка в сетевой порядок и обратно. Например, из системного порядка в сетевой **htonl**, **WSAHtonl**, **htons**, **WSAHtons**. Обратную задачу решают функции **ntohl**, **WSANtohl**, **ntohs**, **WSANtohs**.

Указание 2. Как Вам известно, запись IP-адреса протокола IPv4 в виде четырех октетов, разделенных точками (т.н. точечная десятичная нотация) используется для удобства восприятия его человеком. Формально, IP-адрес представляет собой 32-битовое двоичное число (или 128-битовое в IPv6). В таком виде он чаще всего и используется в **Winsock**. Для его преобразования используются функции **inet_addr** (из точечной десятичной нотации в двоичную) и **inet_ntoa** (из двоичной в строку, представляющую собой точечную десятичную нотацию). Прототипы функции:

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

cp – указатель на строку, содержащую IP-адрес;
возвращаемое значение – двоичное представление соответствующего IP-адреса или значение **INADDR_NONE**, если входная строка не содержит корректного IP-адреса. IP-адрес возвращается в сетевом порядке.

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

in – указатель на структуру **in_addr**, содержащую IP-адрес;
возвращаемое значение – указатель на буфер, содержащий строку с адресом в десятичной точечной нотации или **NULL** в случае ошибки. Буфер выделяется **Winsock** и не требует освобождения приложением. Данные гарантировано будут в нем до следующего вызова функции **Winsock**, но не дольше.

Организация обмена данными по протоколам TCP/UDP с использованием Winsock 2

Изученные Вами ранее компоненты **TClientSocket**, **TTcpClient**, **TServerSocket**, **TTcpServer** являются объектно-ориентированной оболочкой над основным средством коммуникации библиотеки **Windows Sockets** – **сокетом** (**socket**).

В **Winsock** сокет представляет собой дескриптор, используя который можно получать и отправлять данные. При этом общий алгоритм работы выглядит так: после инициализации библиотеки **Winsock** мы создаем сокет с определенными свойствами и используем его для подключения, приема и передачи данных, и т.д. Создавая сокет, следует указать его параметры: будет использоваться **TCP/IP** или **IPX/SPX** (если **TCP/IP**, то какой тип и т.д.). Для **TCP/IP** поддерживаются два основных типа сокетов:

— **SOCK_STREAM** (потоковый или **connection-based socket**);

— **SOCK_DGRAM** (дейтаграммный или **connectionless socket**).

Первый тип сокетов использует в качестве транспортного протокола **TCP**, второй тип сокетов – **UDP**. Для объявления сокетов используется тип **SOCKET**:

```
SOCKET s;
```

Для создания сокета следует использовать функцию **socket**:

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

af – спецификация семейства протоколов (для **TCP/IP** должен иметь значение **AF_INET**);

type – спецификация типа для создаваемого сокета, – какой транспортный протокол следует использовать (**TCP** – **SOCK_STREAM** или **UDP** – **SOCK_DGRAM**). Нулевое значение соответствует выбору по умолчанию - **TCP**;

protocol – протокол, используемый для передачи данных через сокет.

Если ошибок при создании сокета не возникло, функция возвращает дескриптор сокета, иначе значение **INVALID_SOCKET** (более подробные сведения можно получить, используя **WSAGetLastError()**, коды ошибок приведены в справке к **Winsock**).

Пример использования:

```
SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
if (s == INVALID_SOCKET)
{
     …
     error = WSAGetLastError();
     …
}
```

Клиентская часть приложения.

После создания сокета его можно использовать для обмена данными с другими узлами. Для этого нужно установить соединение с другим узлом, задав его IP-адрес и порт, который прослушивает приложение-сервер:

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

s – неподключенный сокет;

name – задает сокет, к которому выполняется подключение. Структура **sockaddr** зависит от используемого семейства протоколов. Для IPv4 используется **sockaddr_in**;

namelen – длина имени сокета в байтах;

функция возвращает нуль, если соединение создано или, в противном случае, **SOCKET_ERROR**.

Используемая в качестве параметра **name** структура описывается как:

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

где **sin_family** – задает семейство адресов. Должно быть равно **AF_INET**;

sin_port – порт, к которому выполняется подключение;

sin_addr – IP-адрес, которому выполняется подключение в сетевом порядке байт. Для его хранения используется структура **in_addr**, знакомая Вам по предыдущей лабораторной работе;

sin_zero – заполнитель, выравнивающий размер структуры с размером **sockaddr**.

Напомним формат структуры **in_addr**:

```
struct in_addr {
union {
struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
struct { u_short s_w1,s_w2; } S_un_w;
u_long S_addr;
} S_un;
};
```

здесь **S_un_b** – адрес в виде четырех **u_char**;

S_un_w – адрес в виде двух **u_short**;

S_addr – адрес в виде **u_long**.

Для получения упакованного адреса используются функции из Указания 2. предыдущей лабораторной работы.

Пример использования:

```
// объявляем переменную для хранения адреса
sockaddr_in s_addr;
// очищаем ее
```

```
ZeroMemory(&s_addr, sizeof(s_addr));
// задаем семейство адресов Internet
s_addr.sin_family = AF_INET;
// задаем адрес узла, к которому выполняем подключение. Не забываем преобразовать адрес из строкового десятичного формата в бинарный.
s_addr.sin_addr.S_un.S_addr = inet_addr("192.168.1.1");
// задаем порт приложения-сервера. Не забываем преобразовывать номер порта в сетевой порядок байт.
s_addr.sin_port = htons(2009);
…
// открываем соединение
int con_status;
con_status = connect(s, (sockaddr *)&s_addr, sizeof(s_addr));
if (con_status == SOCKET_ERROR)
{
…
error = WSAGetLastError();
…
}
```

После установки соединения можно начинать обмен данными.

Замечание: **UDP**-сокеты не требуют установки соединения, поэтому для таких сокетов обычно не используется **connect**. Однако при его использовании **UDP**-сокет может обмениваться данными не только с помощью **sendto** и **recevfrom**, но и с помощью более простых **send** и **recv**.

Серверная часть приложения.

На стороне сервера, прежде чем использовать, сокет нужно связать его с локальным адресом и портом, который будет прослушиваться в ожидании входящих подключений. Если у узла сервера есть несколько IP-адресов, то можно указать один из них или привязать сокет сразу ко всем IP-адресам (для этого нужно указать в качестве IP-адреса константу **INADDR_ANY**, равную нулю). Для связывания сокета с локальным адресом и портом используется:

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

s – сокет, к которому выполняется привязка;
name – задает адрес, к которому привязывается сокет;
namelen – задает длину параметра **name** в байтах;
возвращает **0**, если привязка завершилась успешно, или **SOCKET_ERROR** в

противном случае.

Замечание. Клиент тоже должен связывать свой сокет с локальным адресом, но за него это выполняет connect, связывая клиента со случайно выбранным портом из диапазона `1024-5000`. Сервер же должен использовать строго определенный протоколом порт, поэтому выполняет связывание «вручную».

Выполнив связывание, TCP-сервер переходит в режим ожидания подключений клиентов с помощью:

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

s – сокет сервера;
backlog – максимальный размер очереди ожидающих подключений. Ограничивает количество одновременных соединений. При его превышении очередному клиенту будет отказано в подключении к серверу;
возвращает нуль в случае успешного выполнения или `SOCKET_ERROR` в противном случае.

Замечание. **UDP**-сервера не используют функцию **`listen`** так как не устанавливают соединение и после связывания сразу же могут читать входящие данные.

После начала прослушивания серверного сокета приложение может начать извлекать запросы на соединение из очереди ожидающих подключение. Это выполняется вызовом **`accept`**, который автоматически создает новый сокет для входящего подключения, выполняет связывание и возвращает его дескриптор, а в структуру **`sockaddr`** заносит сведения о подключившемся клиенте (IP-адрес и порт):

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

здесь **s** – серверный «прослушиваемый» сокет;
addr – адрес извлеченного из очереди ожидающих подключений сокета;
addrlen – длина структуры **sockaddr**, извлеченной из очереди;
функция возвращает дескриптор сокета для обмена данными с подключившимся клиентом или `INVALID_SOCKET`, если произошла ошибка. Если очередь входящих подключений пуста, то функция не возвращает управление, пока с сервером не будет установлено соединение.

Замечание. Для параллельной работы с несколькими клиентами после получения сокета следует создать новый поток или процесс, передать ему полученный от `accept` сокет и вернуться к обработке входящих соединении. Иначе, пока не будет завершена обработка одного клиента, сервер не сможет обслужить другого.

Передача и прием данных.

Для передачи данных по установленному соединению используется **send**:

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

где **s** – подключенный сокет;
buf – указывает на буфер с данными для передачи;
len – длина передаваемых данных;
flags – флаги, задающие способ, которым выполняется вызов;
функция возвращает количество отправленных байтов или **SOCKET_ERROR** в случае ошибки.

Например:

```
if (send(s, (char *)&buff, 512,0) == SOCKET_ERROR)
{
…
error = WSAGetLastError();
…
}
```

Длина пакета данных ограничена протоколом. Функция не возвращает значение до тех пор, пока данные не будут отправлены.

Замечание. Функция не гарантирует, что данные были доставлены клиенту. Она возвращает число посланных клиенту байт, а не число им полученных.

Для приема данных от узла, с которым установлено соединение, используется функция **recv**:

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

здесь **s** – дескриптор подключенного сокета;
buf – буфер для входящих данных;
len – длина буфера в байтах;
flags – флаги, задающие способ, которым выполняется вызов;
функция возвращает число полученных байт, если соединение было закрыто – нуль, **SOCKET_ERROR** в случае ошибки. Если данных нет, то функция не возвращает управление, пока не придет пакет (или пока не истечет тайм-аут).

Замечание. **recv** получает данные порциями, которыми они передавались функцией send (если был предоставлен буфер достаточного размера).

Пример использования:

```
int recv_len = 0;
recv_len = recv(s, (char *)&buff, MAX_PACKET_SIZE, 0);
if (recv_len == SOCKET_ERROR)
{
…
error = WSAGetLastError();
```

```
…
}
```

Дейтаграммный сокет (**UDP**) может использовать функции **send** и **recv**, если перед ними был выполнен вызов **connect** или использовать собственные вызовы: **sendto** и **recvfrom**.

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

где **s** – дескриптор сокета;
buf – буфер с данными;
len – длина данных в буфере отправки в байтах;
flags – флаги, задающие способ, которым выполняется вызов;
to – указывает на структуру **sockaddr**, содержащую адрес сокета получателя;
tolen – размер адреса в **to**;
функция возвращает число отправленных байт или **SOCKET_ERROR**, если при отправке возникла ошибка.

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

где **s** – дескриптор сокета;
buf – буфер для входящих данных;
len – размер буфера в байтах;
flags – флаги, задающие способ, которым выполняется вызов;
from – указатель на адрес сокета с которого пришли данные;
fromlen – длина адреса;
возвращает число полученных байт или **SOCKET_ERROR** при возникновении ошибки.
В отличие от рассмотренных выше **send** и **recv** функциям **sendto** и **recvfrom** требуется указывать адрес узла передающего или принимающего данные.

Замечание. Рассмотренные выше функции могут управляться с помощью флагов **flags**. Они могут принимать значение `MSG_PEEK` или `MSG_OOB`. Флаг `MSG_PEEK` заставляет функции получения данных просматривать данные вместо их получения. Флаг `MSG_OOB` предназначен для передачи/приема срочных данных.

Закрытие соединения.

После завершения передачи данных соединение должно быть закрыто. Для этого используются функции **shutdown** и **closesocket**:

```
int shutdown(SOCKET s, int how);
```

s – закрываемый сокет;
how – способ закрытия.

```
int closesocket(SOCKET s);
```

s – закрываемый сокет.

Оба вызова возвращают нуль в случае успеха или **SOCKET_ERROR,** если произошла ошибка.

Различают два типа закрытия соединения: **abortive** (экстренное закрытие сокета, выполняется **closesocket()**) и **graceful**.

Замечание. Существуют более сложные способы закрытия соединения (выборочное закрытие соединения на одной из сторон, оставляя другую сторону в рабочем состоянии). Для этого используется функция `shutdown` и флаг `how` (`SD_RECEIVE` закрывает канал «сервер-клиент», `SD_SEND` закрывает канал «клиент-сервер», `SD_BOTH` – оба). Это «мягкое» закрытие соединения – удаленному узлу посылается уведомление о желании разорвать соединение, но соединение не разрывается, пока клиент не возвратит свое подтверждение.

Замечание. Вызов **shutdown()** не освобождает от последующего вызова **closesocket()**.

После завершения работы с сокетом не забудьте вызвать **WSACleanup()**.

Лабораторная работа № 10
(4 часа)

Задание. Используя библиотеку `Windows Sockets 2`, разработайте приложение соответственно Вашему варианту.

Варианты заданий.

1) Разработайте приложение для передачи файла между различными узлами сети на основе протокола **UDP** с использованием блокирующих сокетов библиотеки **Winsock**.
2) Разработайте приложение для передачи файла между двумя узлами сети на основе протокола **TCP** с использованием блокирующих сокетов библиотеки **Winsock**.
3) Разработайте приложение для обмена мгновенными сообщениями между одним сервером и несколькими клиентами на основе протокола **UDP** и блокирующих сокетов библиотеки **Winsock**.
4) Разработайте приложение для обмена сообщениями между одним сервером и клиентом на основе протокола **TCP** и блокирующих сокетов библиотеки **Winsock**.
5) Разработайте сетевой клиент-серверный вариант игры «Крестики-нолики» для поля 3x3 на базе протокола **TCP** и блокирующих сокетов библиотеки **Winsock**.
6) Разработайте сетевой клиент-серверный вариант игры «Крестики-нолики» для поля 3x3 на базе протокола **UDP** и блокирующих сокетов библиотеки **Winsock**.
7) Разработайте приложение для проверки доступности удаленного хоста методом пульсации (“heartbeat”). Метод заключается в том, что клиент периодически передает серверу сообщение о своем присутствии, при получении которого сервер сбрасывает таймер, ассоциированный с клиентом. В случае пропуска нескольких пульсации таймер срабатывает и выдается предупреждение о недоступности удаленного хоста.
8) Разработайте простейшее приложение-редиректор, которое будет переадресовать полученный запрос на указанный порт другого указанного компьютера.

Указание. Начните разработку с проектирования протокола.

Сокеты без блокировки

При выполнении программы, написанной в прошлой лабораторной работе, Вы могли заметить, что некоторые функции Winsock ожидают завершения выполняемых ими операций. При этом приложение практически «зависает», пока не завершится текущая операция. Особенно хорошо это заметно во время отладки по шагам. Эта особенность не очень хорошо подходит для программ, которые кроме получения и отправки данных должны выполнять множество других действий, например, ввод-вывод информации, запись данных на диск. Избежать подобного поведения программы можно различными способами. Например, можно воспользоваться многопоточными приложениями и вынести код, обращающийся к блокирующим вызовам в отдельный поток. Другим вариантом решения этой проблемы является использование средств Winsock.

Первый способ решения описанной выше проблемы – использование функции **ioctlsocket**, которая позволяет изменять режим ввода/вывода любого указанного сокета:

```
int ioctlsocket(SOCKET s, long cmd, u_long* argp);
```

где **s** – сокет;

cmd – команда для выполнения сокетом;

argp – указатель на параметры команды.

Если вызов завершается успешно, функция возвращает нуль, в противном случае, **SOCKET_ERROR**.

Для переключения сокета в неблокирующий режим (**nonblocking mode**) используется команда **FIONBIO**. При этом аргумент **argp** используется для включения или выключения этого режима: если **argp** указывает на ненулевое значение режим включается, в случае нулевого значения – отключается. По умолчанию режим отключен.

Пример использования:

```
…
u_long param = 12345;
if (ioctlsocket(s, FIONBIO, &param) == SOCKET_ERROR)
{
    error = WSAGetLastError();
    // … обрабатываем ошибку
    return -1;
}
```

После перевода сокета в неблокирующий режим **Winsock**-функции вызываемые для этого сокета не дожидаются окончания операции ввода/вывода и не вызывают нежелательных «зависании» в работе программы. Однако возврат из функции может произойти раньше, чем будут получены данные и возникает вопрос: как определить, что текущая операция ввода/вывода завершена? Для этого используют код ошибки **WSAEWOULDBLOCK**, который возвращают **Winsock**-функции для **nonblocked**-

сокета, если текущая операция не завершена. Например, если функция **recv** вернула этот код ошибки, значит, данные еще не готовы для чтения и обращение к ней придется повторить позже. В таком случае, программа может выполнять другие действия, попутно проверяя, не завершено ли текущее действие ввода/вывода.

Пример использования:

```
int len = 0;
do
{
len = recv (s, (char*)&buff, 1024, 0);
if (len == SOCKET_ERROR)
{
     int res = WSAGetLastError();
     if (res != WSAEWOULDBLOCK)
     {
       // обработать ошибку
       return -1;
     } else
     len = 1;
else
     for (int i = 0; i < len; i++)
      printf("%c", buff[i]);
// если recv вернул WSAEWOULDBLOCK, то можем продолжать работу
…
} while (len !=0);
```

Другим вариантом решения проблемы блокирующих сокетов является использование функции **select**. Функция **select** позволяет определить текущее состояние одного или более сокетов. То есть из какого-то входящего множества сокетов она формирует на выходе множество сокетов, готовых к операциям чтения/записи:

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

здесь **nfds** – не используется, оставлен для совместимости;

readfds – указатель на множество сокетов, проверяемых на готовность к чтению;

writefds – указатель на множество сокетов, проверяемых на готовность к отправке данных;

exceptfds – указатель н а множество сокетов, проверяемых на ошибки;

timeout – тайм-аут проверки, заданный с помощью структуры **timeval.**

Каждый из параметров может быть установлен в **NULL**, если он не нужен. Если **timeout==NULL**, то функция вызовет блокировку до тех пор, пока не появится первый готовый к вводу/выводу сокет.

Функция возвращает количество сокетов во всех множествах готовых к вводу/выводу, нуль – если в течение тайм-аута ни одного такого сокета не появилось и **SOCKET_ERROR** в случае ошибки.

Параметры **readfds**, **writefds** и **exceptfds** – указатели на структуру **fd_set** определяющую множество сокетов следующего вида:

```
typedef struct fd_set{
  u_int fd_count;
  SOCKET fd_array[FD_SETSIZE];
} fd_set;
```

где **fd_count** – задает количество сокетов во множестве;
fd_array – массив сокетов.

Для работы с этим типом определены следующие макросы:

```
FD_CLR (s, *set);      // удаляет дескриптор сокета s из set
FD_ISSET (s, *set);  //  возвращает ненулевое значение, если s присутствует в set, иначе 0
FD_SET (s, *set);      // добавляет s к set
FD_ZERO (*set);        // очищает множество set
```

Структура **timeval**, используемая вызовом **select** задается следующим образом:

```
struct timeval {
long tv_sec;
long tv_usec;
};
```

Здесь **tv_sec** – секунды;
tv_usec – микросекунды.

Пример использования:

```
int len, res;

fd_set read_s;
timeval time_out;

time_out.tv_sec  =  0;  time_out.tv_usec  =  500000;  // задаем тайм-аут в 0,5 секунды

do
{
FD_ZERO (&read_s);
FD_SET (s, &read_s);
res = select (0, &read_s, NULL, NULL, &time_out);
if (res == SOCKET_ERROR)
{
     // обрабатываем ошибку
     error = WSAGetLastError();
     …
```

```
}

if ((res != 0) && (FD_ISSET(s, &read_s)))
{
     // сокет готов к чтению – данные присутствуют
     len = recv (s, (char*) &buff, 1024, 0);
     if (len == SOCKET_ERROR)
     {
          // обрабатываем ошибку сокета
          error = WSAGetLastError();
          …
     }
     for (int i = 0; i<len; i++)
          printf("%c", buff[i]);
}
// …
} while (len != 0);
…
```

Лабораторная работа № 11
(4 часа)

Задание. Перепишите приложение HTTP-клиент из лабораторной работы №5 на основе библиотеки Winsock 2 и сокетов без блокировки.

Задание *. Напишите приложение FTP-клиент на основе библиотеки Winsock 2 и сокетов без блокировки.