Лабораторная работа 2

МЕХАНИЗМЫ СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ

Цель работы — изучить программный интерфейс сетевых сокетов, получить навыки организации взаимодействия программ при помощи протоколов Internet и разработки прикладных сетевых сервисов.

Теоретические сведения

Сетевые сокеты

Сетевой сокет — это одно из средств коммуникации процессов. Главное отличие сокетов от других средств межпроцессного взаимодействия — обменивающиеся информацией процессы могут быть удаленными, т.е. они не обязательно должны находиться на одном компьютере. В частности, механизм сокетов применяется в таких программах, как telnet, rlogin, ssh, wget, ftp, tftp и многих других.

При создании сокета необходимо определить три параметра: стиль взаимодействия, пространство имен и протокол. Стиль взаимодействия контролирует, как сокет обрабатывает передаваемые данные, и определяет количество партнеров взаимодействия. Через сокеты данные передаются блоками или пакетами. Стиль взаимодействия определяет, как эти данные будут обработаны и как они передаются от отправителя к получателю.

Пространство имен определяет, как записаны адреса сокета, которые идентифицируют один конец подключения сокета. Адреса сокета в локальном пространстве имен являются именами файлов. В пространстве имен Интернет адрес сокета состоит из IP-адреса компьютера, присоединенного к сети и номера порта, который идентифицирует сокет среди множества сокетов на том же компьютере.

Протокол определяет, как передаются данные. Например, в виде потока данных с гарантированной доставкой и очередностью, либо в виде пакетов.

Поддерживаются не все комбинации стилей, пространств имен и протоколов.

С помощью сокетов можно легко организовать взаимодействия программ и разработать прикладные сетевые сервисы.

Локальные сокеты

Сокеты, подключающие процессы на одном компьютере, могут использовать локальное пространство имен. Они называются локальными сокетами или сокетами UNIX-домена. Адреса этих сокетов, определяемые именами файлов, используются только при создании соединения.

Имя сокета указывается в структуре *sockaddr_un*. Если в *AF_LOCAL* установлено поле *sun_family*, это указывает на то, что адрес в локальном пространстве имен. Поле *sun_path* указывает, что используется имя файла; максимальная длина поля — 108 байт. Может использоваться любое имя файла, но для процесса должно быть установлено право на запись в каталог. Для вычисления длины *struct sockaddr_un*, используется макрокоманда *SUN_LEN*.

Чтобы соединиться с сокетом, процесс должен иметь право на чтения файла. Хотя различные компьютеры могут совместно использовать одну файловую

систему, только процессы, запущенные на этом компьютере, могут взаимодействовать, используя сокеты локального пространства имен.

Единственный допустимый протокол для локального пространства имен — 0.

В файловой системе локальный сокет представлен как файл. Пример отображения сокета в системе:

```
% ls -l /tmp/socket
srwxrwx--x 1 user group 0 Nov 13 19:18 /tmp/socket
```

Вызов *unlink* удаляет локальный сокет при завершении работы с ним.

Internet сокеты

Данные сокеты используются для соединения нескольких процессов на различных машинах, подключенных к сети. Интернет адрес сокета состоит из двух частей: адреса компьютера и номера порта. Различают два основных типа сокетов, определяемых константами *SOCK_STREAM* и *SOCK_DGRAM*. Для первого типа данные передаются непрерывным потоком, гарантируется доставка всех байтов и в нужном порядке — протокол Интернета *TCP*. Второй тип сокетов предполагает передачу данных в виде отдельных пакетов (*дейтаграмм*), доставка и порядок отправки/приема пакетов не гарантируется — протокол Интернета *UDP*.

Для создания сокетов Интернета следует использовать функцию:

```
int socket(int domain, int type, int protocol);
```

параметр *domain* определяет семейство протоколов, которое будет использоваться для взаимодействия (*AF_INET* для сокетов интернета семейства протоколов IP версии 4). Второй параметр определяет тип сокета, т.е будет ли передаваться информация дейтаграммами без гарантии доставки (*SOCK_DGRAM*), с установкой логического соединения (*SOCK_STREAM*) и т. д. Третий параметр служит для указания конкретного протокола для данного типа сокета (в случае использования констант SOCK_STREAM и SOCK_DGRAM в предыдущем параметре может быть передан 0). В случае успешного завершения возвращается неотрицательное число — файловый дескриптор созданного сокета, в случае неудачи возвращается -1.

По завершению работы с сокетом его необходимо "закрыть" с помощью функции *close* (*closesocket* для Windows). Единственным параметром функции является файловый дескриптор созданного ранее сокета.

Клиентские TCP-соединения

Для установки нового соединения с удаленным узлом по протоколу TCP открытый дескриптор сокета с типом SOCK_STREAM необходимо передать в функцию:

```
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen)
```

где *sockfd* — дескриптор сокета; *serv_addr* — указатель на структуру, содержащую информацию о сокете; *addrlen* — размер этой структуры. В случае успеха *connect* возвращает 0, в случае неудачи -1.

Структура *sockaddr* определена в общем виде следующим образом:

```
struct sockaddr {
	unsigned short sa_family;
	char sa_data[14];
};
```

Для семейства протоколов IP версии 4 (*AF_INET*) эта структура переопределена как *sockaddr_in*:

```
struct sockaddr_in {
	short sin_family;		//семейство протоколов: AF_INET
	unsigned short sin_port;	//номер порта
	struct in_addr sin_addr;	//IP-адрес
	char sin_zero[8];		//не используется, заполняется нулями
};
```

Номер порта нужно задавать, используя сетевой порядок байтов. Для преобразования между обычным и сетевым порядками для 2-х байтовых переменных служат функции *htons* и *ntohs*. Структура sin_addr содержит поле *s_addr*, содержащее адрес удаленного компьютера в сетевом порядке байтов. Для преобразования IP-адресов или других 4-х байтовых значений в сетевой порядок следует применять функцию *htonl*, обратно — *ntohl*.

Для преобразования адреса из символьного представления в числовое может быть использована функция *inet_ntoa*, обратно — функция *inet_aton*.

Для передачи данных от клиента к удаленному серверу используется функция *send*, для приема данных от сервера — *recv*:

```
int send(int sockfd, const void *buf, int len, int flags);
```

где *sockfd* — дескриптор сокета, *buf* — адрес буфера, содержащего передаваемые данные, *len* — размер передаваемых данных, *flags* — набор флагов: в Windows может быть передан 0, в Linux — рекомендуется применять *MSG_NOSIGNAL* (не прерывать выполнение программы в случае разрыва связи). Функция возвращает количество отправленных данных, либо -1 в случае ошибки.

```
int recv(int sockfd, void *buf, int len, int flags);
```

где *sockfd* — дескриптор сокета, *buf* — адрес буфера, в который будут записаны принятые данные, *len* — размер буфера, *flags* — набор флагов: может быть передан 0. Функция возвращает количество принятых данных, -1 — в случае ошибки, либо 0 — если все данные были приняты и соединение было корректно закрыто удаленной стороной.

Следует отметить, что функция *recv* не возвращает управление до тех пор, пока какой-либо объем данных будет получен от удаленной стороны, т.е.

управление программой *блокируется* на все время работы функции. Такое поведение называется *блокирующим* режимом работы сокетов.

Примечание: в ОС Linux для записи в сокет и чтения из него, можно также использовать функции *write* и *read*, точно так же как и при работе с файлами.

Системный вызов *int close(int sockfd)* закрывает дескриптор сокета, завершая соединение (в Windows функция имеет имя *closesocket*).

Следующий пример демонстрирует использование описанных функций и содержит пример простого TCP-клиента, отправляющего на удаленный сервер http-запрос, а затем принимающий от сервера ответ с сохранением в файл page.html.

```
#ifdef _WIN32
	#define WIN32_LEAN_AND_MEAN
	#include <windows.h>
	#include <winsock2.h>
	#include <ws2tcpip.h>
	// Директива линковщику: использовать библиотеку сокетов
	#pragma comment(lib, "ws2_32.lib")
#else // LINUX
	#include <sys/types.h>
	#include <sys/socket.h>
	#include <netdb.h>
	#include <errno.h>
#endif

#include <stdio.h>
#include <string.h>

#define WEBHOST "google.com"

int init()
{
#ifdef _WIN32
	// Для Windows следует вызвать WSAStartup перед началом использования сокетов
	WSADATA wsa_data;
	return (0 == WSAStartup(MAKEWORD(2, 2), &wsa_data));
#else
	return 1; // Для других ОС действий не требуется
#endif
}

void deinit()
{
#ifdef _WIN32
	// Для Windows следует вызвать WSACleanup в конце работы
	WSACleanup();
#else
	// Для других ОС действий не требуется
#endif
}

int sock_err(const char* function, int s)
{
	int err;
#ifdef _WIN32
	err = WSAGetLastError();
#else
```

```
	err = errno;
#endif

	fprintf(stderr, "%s: socket error: %d\n", function, err);
	return -1;
}

void s_close(int s)
{
#ifdef _WIN32
	closesocket(s);
#else
	close(s);
#endif
}

// Функция определяет IP-адрес узла по его имени.
// Адрес возвращается в сетевом порядке байтов.
unsigned int get_host_ipn(const char* name)
{
	struct addrinfo* addr = 0;
	unsigned int ip4addr = 0;

	// Функция возвращает все адреса указанного хоста
	// в виде динамического однонаправленного списка
	if (0 == getaddrinfo(name, 0, 0, &addr))
	{
		struct addrinfo* cur = addr;
		while (cur)
		{
			// Интересует только IPv4 адрес, если их несколько - то первый
			if (cur->ai_family == AF_INET)
			{
				ip4addr = ((struct sockaddr_in*) cur->ai_addr)->sin_addr.s_addr;
				break;
			}
			cur = cur->ai_next;
		}
		freeaddrinfo(addr);
	}

	return ip4addr;
}

// Отправляет http-запрос на удаленный сервер
int send_request(int s)
{
	const char* request = "GET / HTTP/1.0\r\nServer: " WEBHOST "\r\n\r\n";
	int size = strlen(request);
	int sent = 0;

#ifdef _WIN32
	int flags = 0;
#else
	int flags = MSG_NOSIGNAL;
#endif

	while (sent < size)
	{
		// Отправка очередного блока данных
		int res = send(s, request + sent, size - sent, flags);

		if (res < 0)
			return sock_err("send", s);
```

```
                sent += res;
                printf("  %d bytes sent.\n", sent);
        }

        return 0;
}

int recv_response(int s, FILE* f)
{
        char buffer[256];
        int res;

        // Принятие очередного блока данных.
        // Если соединение будет разорвано удаленным узлом recv вернет 0
        while ((res = recv(s, buffer, sizeof(buffer), 0)) > 0)
        {
                fwrite(buffer, 1, res, f);
                printf("  %d bytes received\n", res);
        }

        if (res < 0)
                return sock_err("recv", s);

        return 0;
}


int main()
{
        int s;
        struct sockaddr_in addr;
        FILE* f;

        // Инициалиазация сетевой библиотеки
        init();

        // Создание TCP-сокета
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0)
                return sock_err("socket", s);

        // Заполнение структуры с адресом удаленного узла
        memset(&addr, 0, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_port = htons(80);
        addr.sin_addr.s_addr = get_host_ipn(WEBHOST);

        // Установка соединения с удаленным хостом
        if (connect(s, (struct sockaddr*) &addr, sizeof(addr)) != 0)
        {
                s_close(s);
                return sock_err("connect", s);
        }

        // Отправка запроса на удаленный сервер
        send_request(s);

        // Прием результата
        f = fopen("page.html", "wb");
        recv_response(s, f);
        fclose(f);

        // Закрытие соединения
        s_close(s);
```

```
	deinit();

	return 0;
}
```

UDP-клиенты

В отличие от TCP протокол UDP не поддерживает установку соединения, не гарантирует порядок доставки пакетов и доставку вообще. Протоколы передачи данных и файлов, основанные на UDP, должны предусматривать отправку подтверждений и повторных пакетов, чтобы обеспечить доставку данных.

Данные протокола UDP отправляются "дейтаграммами" — небольшими блоками данных. Отправленный блок будет доставлен на сервер полностью или не доставлен совсем. Возможна доставка повторных или ранее отправленных пакетов.

Отправка и прием данных осуществляются с помощью функций *sendto* и *recvfrom*:

```
int sendto(int sockfd, const void *buf, int len, int flags,
           const struct sockaddr *dest_addr, int addrlen);
```

где *sockfd* — дескриптор сокета, *buf* — адрес буфера, содержащего передаваемые данные, *len* — размер передаваемых данных, *flags* — набор флагов: в Windows может быть передан 0, в Linux — рекомендуется применять *MSG_NOSIGNAL* (не прерывать выполнение программы в случае разрыва связи), *dest_addr* — адрес удаленной стороны, *addrlen* — размер данных, занимаемых *dest_addr*. Функция возвращает количество отправленных данных (совпадает с len), либо -1 в случае ошибки.

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

где *sockfd* — дескриптор сокета, *buf* — адрес буфера, в который будут записаны принятые данные, *len* — размер буфера, *flags* — набор флагов: может быть передан 0. Функция возвращает количество принятых данных (размер принятой дейтаграммы), либо -1 — в случае ошибки. В переменную *from* сохраняется адрес удаленной стороны, приславшей дейтаграмму, в переменную *fromlen* — размер сохраненных в *from* данных. При вызове функции *fromlen* должна содержать максимальный размер, который допустимо записывать по адресу *from*.

Следует отметить, что функция *recvfrom* не возвращает управление до тех пор, пока какая-либо дейтаграмма не будет получена от удаленной стороны, т.е. управление программой *блокируется* на все время работы функции. Такое поведение называется *блокирующим* режимом работы сокетов.

Если удаленная сторона не отправляет данные или они по каким-либо причинам не доставляются, то программа может навсегда "зависнуть" в функции ожидания очередной дейтаграммы. Чтобы исключить такое поведение следует проверить наличие данных в буфере приема UDP-сокета и, убедившись, что дейтаграмма присутствует, вызывать *recvfrom*.

Следующий пример демонстрирует осуществление DNS-запроса к серверу 8.8.8.8 с целью получения IP-адреса хостов с доменным именем www.yandex.ru. *Примечание:* Протокол DNS используется для трасляции доменных имен в IP-адреса: клиент сообщает серверу интересующий его домен, а сервер возвращает IP-адреса этого домена. Данный пример лишь демонстрирует способ взаимодействия клиента с сервером с помощью дейтаграмм, использовать этот способ для получения адресов при выполнении лабораторной работы не требуется: современные операционный системы используют собственные механизмы запросов DNS, IP адреса могут быть получены автоматически операционной системой. Для этого программе следует вызывать функцию *gethostbyname*.

```
#ifdef _WIN32
	#define WIN32_LEAN_AND_MEAN
	#include <windows.h>
	#include <winsock2.h>
	// Директива линковщику: использовать библиотеку сокетов
	#pragma comment(lib, "ws2_32.lib")
#else // LINUX
	#include <sys/types.h>
	#include <sys/socket.h>
	#include <sys/time.h>
	#include <sys/select.h>
	#include <netdb.h>
	#include <errno.h>
#endif

#include <stdio.h>
#include <string.h>

// Функция извлекает IPv4-адрес из DNS-дейтаграммы.
// Задание л/р не требует детального изучения кода этой функции
unsigned int get_addr_from_dns_datagram(const char* datagram, int size);

void send_request(int s, struct sockaddr_in* addr)
{
	// Данные дейтаграммы DNS-запроса. Детальное изучение для л/р не требуется.
	char dns_datagram[] = {0x00, 0x00, 0x00, 0x00,
				0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
				3, 'w', 'w','w', 6, 'y','a','n','d','e','x', 2, 'r', 'u', 0,
				0x00, 0x01, 0x00, 0x01};

#ifdef _WIN32
	int flags = 0;
#else
	int flags = MSG_NOSIGNAL;
#endif

	int res = sendto(s, dns_datagram, sizeof(dns_datagram), flags, (struct sockaddr*) addr,
sizeof(struct sockaddr_in));
	if (res <= 0)
		sock_err("sendto", s);
}


// Функция принимает дейтаграмму от удаленной стороны.
// Возвращает 0, если в течение 100 миллисекунд не было получено ни одной дейтаграммы
unsigned int recv_response(int s)
{
	char datagram[1024];
	struct timeval tv = {0, 100*1000}; // 100 msec
```

```
	int res;

	fd_set fds;
	FD_ZERO(&fds); FD_SET(s, &fds);

	// Проверка - если в сокете входящие дейтаграммы
	// (ожидание в течение tv)
	res = select(s + 1, &fds, 0, 0, &tv);
	if (res > 0)
	{
		// Данные есть, считывание их
		struct sockaddr_in addr;
		int addrlen = sizeof(addr);

		int received = recvfrom(s, datagram, sizeof(datagram), 0, (struct sockaddr*) &addr,
&addrlen);

		if (received <= 0)
		{
			// Ошибка считывания полученной дейтаграммы
			sock_err("recvfrom", s);
			return 0;
		}

		return get_addr_from_dns_datagram(datagram, sizeof(datagram));
	}
	else if (res == 0)
	{
		// Данных в сокете нет, возврат ошибки
		return 0;
	}
	else
	{
		sock_err("select", s);
		return 0;
	}
}

int main()
{
	int s;
	struct sockaddr_in addr;
	int i;

	// Инициалиазация сетевой библиотеки
	init();

	// Создание UDP-сокета
	s = socket(AF_INET, SOCK_DGRAM, 0);
	if (s < 0)
		return sock_err("socket", s);

	// Заполнение структуры с адресом удаленного узла
	memset(&addr, 0, sizeof(addr));
	addr.sin_family = AF_INET;
	addr.sin_port = htons(53); // Порт DNS - 53
	addr.sin_addr.s_addr = inet_addr("8.8.8.8");

	// Выполняется 5 попыток отправки и затем получения дейтаграммы.
	// Если запрос или ответ будет потерян - данные будут запрошены повторно
	for (i = 0; i < 5; i++)
	{
		printf(" sending request: attempt %d\n", i + 1);

		// Отправка запроса на удаленный сервер
```

```
			send_request(s, &addr);

			// Попытка получить ответ. Если ответ получен - завершение цикла попыток
			if (recv_response(s))
			{
				break;
			}
		}

		// Закрытие сокета
		s_close(s);

		deinit();

		return 0;
}


unsigned int get_addr_from_dns_datagram(const char* datagram, int size)
{
		unsigned short req_cnt, ans_cnt, i;
		const char* ptr;

		req_cnt = ntohs(*(unsigned short*)(datagram + 4));
		ans_cnt = ntohs(*(unsigned short*)(datagram + 6));

		ptr = datagram + 12;
		for (i = 0; i < req_cnt; i++)
		{
			unsigned char psz;
			do
			{
				psz = *ptr;
				ptr += psz + 1;
			} while (psz > 0);
			ptr += 4;
		}

		for (i = 0; i < ans_cnt; i++)
		{
			unsigned char psz;
			unsigned short asz;
			do
			{
				psz = *ptr;
				if (psz & 0xC0)
				{
					ptr += 2;
					break;
				}

				ptr += psz + 1;
			} while (psz > 0);

			ptr += 8;
			asz = ntohs(*(unsigned short*)ptr);

			if (asz == 4)
			{
				printf(" Found IP: %u.%u.%u.%u\n",
					(unsigned char)ptr[1], (unsigned char)ptr[2], (unsigned char)ptr[3],
(unsigned char)ptr[4]);
			}

			ptr += 2 + asz;
```

```
	}

	return 1;
}
```

TCP-серверы

Чтобы созданный TCP-сокет имел возможность принимать входящие подключения необходимо "привязать" его к определенным адресам локального компьютера, а также указать номер прослушивающего порта. Эти действия выполняет функция *bind:*

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Функция возвращает 0 в случае успешного завершения, отрицательное число в противном случае.

Для TCP-клиентов вызов функции *bind* не требуется т.к. привязка сокета к адресу производится функцией *connect*.

После успешной привязки сокет должен начать "прослушивание", для этого должна быть вызвана функция *listen*:

```
int listen(int sockfd, int backlog);
```

переводит сокет в слушающее состояние, *backlog* — размер очереди соединений, например если указано 1 и то сервер одновременно может принять только одно соединение, все другие попытки подключения клиентов к серверу будут отменены. Это относится только к клиентам, желающим подключиться к серверу в настоящий момент, но не относится к уже подключившимся клиентам.

Ожидание подключения следующего клиента выполняется с помощью функции *accept*. До момента подключения функция ожидает подключений и блокирует выполнение программы. Новое подключение возвращается функцией в виде нового TCP-сокета, описывающего подключенного клиента. Прослушивающий сокет при этом не закрывается и может продолжать принимать подключения.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t
*addrlen);
```

выбирает первый запрос на соединение из очереди. Информация о подключившемся клиенте записывается в структуру **addr*, ее размер записывается в **addrlen*. При вызове функции значение в *addrlen* должно содержать максимальный размер памяти, выделенной под *addr*.

Следующий пример демонстрирует использование перечисленных функций и реализует простой сервер. От подключившегося клиента ожидается строка, завершаемая символом перевода строки ('\n'). Приняв такую строку, сервер отправляет клиенту сообщение с размером полученной строки и закрывает соединение.

```
#ifdef _WIN32
	#define WIN32_LEAN_AND_MEAN
```

```
	#include <windows.h>
	#include <winsock2.h>
	// Директива линковщику: использовать библиотеку сокетов
	#pragma comment(lib, "ws2_32.lib")
#else // LINUX
	#include <sys/types.h>
	#include <sys/socket.h>
	#include <netdb.h>
	#include <errno.h>
#endif

#include <stdio.h>
#include <string.h>

int recv_string(int cs)
{
	char buffer[512];
	int curlen = 0;
	int rcv;

	do
	{
		int i;
		rcv = recv(cs, buffer, sizeof(buffer), 0);
		for (i = 0; i < rcv; i++)
		{
			if (buffer[i] == '\n')
				return curlen;
			curlen++;
		}

		if (curlen > 5000)
		{
			printf("input string too large\n");
			return 5000;
		}
	} while (rcv > 0);

	return curlen;
}

int send_notice(int cs, int len)
{
	char buffer[1024];
	int sent = 0;
	int ret;

#ifdef _WIN32
	int flags = 0;
#else
	int flags = MSG_NOSIGNAL;
#endif

	sprintf(buffer, "Length of your string: %d chars.", len);

	while (sent < (int) strlen(buffer))
	{
		ret = send(cs, buffer + sent, strlen(buffer) - sent, flags);
		if (ret <= 0)
			return sock_err("send", cs);
		sent += ret;
	}

	return 0;
}
```

```c
int main()
{
	int s;
	struct sockaddr_in addr;

	// Инициализация сетевой библиотеки
	init();

	// Создание TCP-сокета
	s = socket(AF_INET, SOCK_STREAM, 0);
	if (s < 0)
		return sock_err("socket", s);

	// Заполнение адреса прослушивания
	memset(&addr, 0, sizeof(addr));
	addr.sin_family = AF_INET;
	addr.sin_port = htons(9000); // Сервер прослушивает порт 9000
	addr.sin_addr.s_addr = htonl(INADDR_ANY); // Все адреса

	// Связывание сокета и адреса прослушивания
	if (bind(s, (struct sockaddr*) &addr, sizeof(addr)) < 0)
		return sock_err("bind", s);

	// Начало прослушивания
	if (listen(s, 1) < 0)
		return sock_err("listen", s);

	do
	{
		// Принятие очередного подключившегося клиента
		int addrlen = sizeof(addr);
		int cs = accept(s, (struct sockaddr*) &addr, &addrlen);
		unsigned int ip;
		int len;

		if (cs < 0)
		{
			sock_err("accept", s);
			break;
		}

		// Вывод адреса клиента
		ip = ntohl(addr.sin_addr.s_addr);
		printf(" Client connected: %u.%u.%u.%u ",
			(ip >> 24) & 0xFF, (ip >> 16) & 0xFF, (ip >> 8) & 0xFF, (ip) & 0xFF);

		// Прием от клиента строки
		len = recv_string(cs);

		// Отправка клиенту сообщения о длине полученной строки
		send_notice(cs, len);

		printf(" string len is: %d\n", len);

		// Отключение клиента
		s_close(cs);

	} while (1); // Повторение этого алгоритма в беск. цикле

	s_close(s);
	deinit();

	return 0;
}
```

Для проверки работоспособности сервера можно использовать консольную утилиту *telnet* в качестве клиента: *telnet 127.0.0.1 9000*. После подключения можно ввести строку, нажать Enter и сервер ответит сообщением о длине присланной строки.

Недостатком такой реализации сервера является невозможность параллельного обслуживания нескольких клиентов. Пока сервер взаимодействует с текущим клиентом, другие подключающиеся клиенты не смогут передавать и принимать данные. В этом можно убедиться, открыв второе подключение с помощью telnet: соединение либо не будет установлено, либо сервер не будет реагировать на присылаемую строку, пока не будет закрыто первое соединение.

UDP-серверы

Реализация UDP-сервера практически не отличается от реализации UDP-клиента. Исключением является необходимость "привязки" сокета к определенному адресу и порту компьютера функцией *bind*. Функцию *listen* и *accept* вызывать для серверных UDP-сокетов не требуется.

Следующий пример реализует простой сервер. От подключившегося клиента ожидается дейтаграмма со строкой, завершаемой символом перевода строки ('\n'). Приняв такую дейтаграмму, сервер отправляет клиенту дейтаграмму, содержащую сообщение с размером полученной строки.

```
#ifdef _WIN32
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")
#else // LINUX
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/select.h>
#include <netdb.h>
#include <errno.h>
#endif

#include <stdio.h>
#include <string.h>

int main()
{
	int s;
	struct sockaddr_in addr;
	int i;

#ifdef _WIN32
	int flags = 0;
#else
	int flags = MSG_NOSIGNAL;
#endif

	// Инициалиазация сетевой библиотеки
	init();

	// Создание UDP-сокета
```

```
	s = socket(AF_INET, SOCK_DGRAM, 0);
	if (s < 0)
		return sock_err("socket", s);

	// Заполнение структуры с адресом прослушивания узла
	memset(&addr, 0, sizeof(addr));
	addr.sin_family = AF_INET;
	addr.sin_port = htons(8000); // Будет прослушиваться порт 8000
	addr.sin_addr.s_addr = htonl(INADDR_ANY);

	// Связь адреса и сокета, чтобы он мог принимать входящие дейтаграммы
	if (bind(s, (struct sockaddr*) &addr, sizeof(addr)) < 0)
		return sock_err("bind", s);

	do
	{
		char buffer[1024] = { 0 };
		int len = 0;
		int addrlen = sizeof(addr);

		// Принятие очередной дейтаграммы
		int rcv = recvfrom(s, buffer, sizeof(buffer), 0, (struct sockaddr*) &addr,
&addrlen);

		if (rcv > 0)
		{
			unsigned int ip = ntohl(addr.sin_addr.s_addr);

			printf("Datagram received from address: %u.%u.%u.%u ",
				(ip >> 24) & 0xFF, (ip >> 16) & 0xFF, (ip >> 8) & 0xFF, (ip)& 0xFF);

			for (i = 0; i < rcv; i++)
			{
				if (buffer[i] == '\n')
					break;
				len++;
			}

			printf(" string len is: %d\n", len);
		}

		sprintf(buffer, "Length of your string: %d chars.", len);

		// Отправка результата-дейтаграммы клиенту
		sendto(s, buffer, strlen(buffer), flags, (struct sockaddr*) &addr, addrlen);

	} while (1);


	// Закрытие сокета
	s_close(s);

	deinit();

	return 0;
}
```

Параллельное обслуживание клиентов

В современных операционных системах предусмотрен ряд механизмов для обспечения серверами параллельного обслуживания нескольких клиентов без использования механизмов многопоточности — *неблокирующий режим* работы сокетов. В этом режиме функции *connect*, *accept*, *send*, *recv*, *sendto*, *recvfrom* и некоторые другие не блокируют управление до наступления соответствующего события, а возвращают значение немедленно. При этом, если данные от клиента еще не были получены, функции возвращают соответствующую ошибку. Блокирующие сокеты блокировали бы выполнение программы.

Перевод сокета в неблокирующий режим зависит от ОС и может быть выполнен для Windows и Linux следующей функцией:

```
#ifdef _WIN32
	#include <winsock2.h>
	#pragma comment(lib, "ws2_32.lib")
#else // LINUX
	#include <fcntl.h>
#endif

int set_non_block_mode(int s)
{
#ifdef _WIN32
	unsigned long mode = 1;
	return ioctlsocket(s, FIONBIO, &mode);
#else
	int fl = fcntl(s, F_GETFL, 0);
	return fcntl(s, F_SETFL, fl | O_NONBLOCK);
#endif
}
```

Рекомендуется переводить **сокет в неблокирующий режим** сразу **после создания** функцией *socket* или принятия подключения функцией *accept*.

Чтобы программа не перебирала все открытые сокеты, пытаясь из каждого получить данные или отправить их, предусмотрены функции и механизмы поиска сокетов, для которых вызов функций чтения или отправки данных имеет смысл в данный момент. Например, с помощью этих механизмов можно отобрать только те сокеты, которые содержат готовые для чтения данные, пришедшие от клиентов. С помощью этих же механизмов можно отобрать те прослушивающие TCP-сокеты, к которым подключились клиенты.

Универсальным механизмов отбора таких сокетов является *select* и *poll* (*WSAPool* для Windows). В ОС Windows дополнительно предлагается не менее эффективный механизм — модель событий *(WSAEvents)*.

Более совершенными механизмами являются *epoll* (для Linux) и *IO Completion ports* (для Windows).

Другие операционные системы могут предлагать свои методы отбора готовых к использованию сокетов, например ОС NetBSD и FreeBSD предлагают механизм *kevent*, во многом схожий с механизмом *epoll*.

Выбор сокетов функцией select

Наиболее универсальным механизмом, имеющим при этом несколько ограничений, является использование функции *select*, определенной стандартом POSIX.

Для использования этого механизма вызывающая программа должна сохранить дескрипторы сокетов, на которых могли произойти события, в некоторую структуру *fd_set* и передать ее в функцию *select*. При этом функция отметит среди всех переданных сокетов только те, на которых произошли события. В функцию передается 3 структуры *fd_set*: первая содержит дескрипторы сокетов, которые могли стать доступны для чтения данных, вторая — для записи данных, третья — дескрипторы тех сокетов, при работе с которыми могли возникнуть ошибки. Первый параметр функции *select* содержит значение максимального дескриптора сокета + 1. Последний параметр функции — максимальное время ожидания события: передается в виде секунд и микросекунд:

```
struct timeval {
  long    tv_sec;         /* seconds */
  long    tv_usec;        /* and microseconds */
};
```

Прототип функции *select* выглядит следующим образом:

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

В случае ошибки функция возвращает -1, в случае таймаута — 0, иначе — количество сокетов во всех наборах fd_set, имеющих какие-либо сработавшие события.

Структура *fd_set* может быть описана в разных ОС по-разному. Для обеспечения переносимости кода предусмотрены следующие макросы для работы с fd_set:

```
FD_ZERO(fd_set *set);
```

Очищает и инициализирует объявленную структуру fd_set.

```
FD_SET(int fd, fd_set *set);
```

Добавляет сокет в fd_set.

```
FD_CLR(int fd, fd_set *set);
```

Удаляет сокет из fd_set.

```
FD_ISSET(int fd, fd_set *set);
```

Проверяет наличие сокета в fd_set.

Структуру *fd_set* необходимо формировать перед каждым вызовом *select*.

Следующий код демонстрирует способ заполнения *fd_set*, вызова *select* и выбора сокетов с событиями. В примере использован сокет ls — прослушивающий (ожидающий новые подключения), а также и несколько сокетов (N) уже подключенных клиентов.

```
	int ls; // Сокет, прослушивающий соединения
	int cs[N]; // Сокеты с подключенными клиентами
	fd_set rfd;
	fd_set wfd;
	int nfds = ls;
	int i;
	struct timeval tv = { 1, 0 };

	while (1)
	{
		FD_ZERO(&rfd);
		FD_ZERO(&wfd);

		FD_SET(ls, &rfd);

		for (i = 0; i < N; i++)
		{
			FD_SET(cs[i], &rfd);
			FD_SET(cs[i], &wfd);
			if (nfds < cs[i])
				nfds = cs[i];
		}

		if (select(nfds + 1, &rfd, &wfd, 0, &tv) > 0)
		{
			// Есть события
			if (FD_ISSET(ls, &rfd))
			{
				// Есть события на прослушивающем сокете, можно вызвать accept,
принять
				// подключение и добавить сокет подключившегося клиента в массив cs
			}

			for (i = 0; i < N; i++)
			{
				if (FD_ISSET(cs[i], &rfd))
				{
					// Сокет cs[i] доступен для чтения. Функция recv вернет данные,
recvfrom - дейтаграмму
				}

				if (FD_ISSET(cs[i], &wfd))
				{
					// Сокет cs[i] доступен для записи. Функция send и sendto будет
успешно завершена
				}
			}
		}
		else
		{
			// Произошел таймаут или ошибка
		}

	}
```

Выбор сокетов функцией poll / WSAPoll

Механизм *poll* (*WSAPoll* для Windows) с точки зрения вызывающей программы во многом напоминает *select*. Отличие состоит в способе упаковки дескрипторов сокетов в соответствующую структуру данных. Дескриптор каждого

сокета, события которого интересуют программу, сохраняется в структуре *struct pollfd*, определенной следующим образом:

```
struct pollfd {
        int fd;           /* описатель сокета */
        short events;     /* запрошенные события */
        short revents;    /* возвращенные события */
};
```

Поле *events* представляет собой битовую маску: каждое интересующее событие заносится в виде бита в это поле. Поле *revents* заполняется функцией *poll*, в случае, если какое-либо из запрашиваемых событий возникло. Поддерживаемые события определены следующими константами:

```
#define POLLIN      ...    /* Можно считывать данные */
#define POLLOUT     ...    /* Можно отравлять (запись не будет блокирована) */
#define POLLERR     ...    /* Произошла ошибка */
#define POLLHUP     ...    /* Соединение разорвано удаленной стороной
                              ("Положили трубку") */
```

Функция *poll* определена следующим образом:

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

параметр *ufds* — массив структур (по одному экземпляру на сокет), *nfds* — количество элементов в массиве *ufds*, *timeout* — время ожидания событий на всех переданных сокетах (в миллисекундах).

При успешном завершении вызова возвращается положительное значение, равное количеству структур с ненулевыми полями *revents* (другими словами, описатели с обнаруженными событиями или ошибками). Нулевое значение указывает на то, что время ожидания истекло, и ни один из дескрипторов не был выбран. При ошибке возвращается -1.

В отличие от структуры *fd_set*, используемой в механизме *select*, массив структур *poolfd* не обязательно готовить каждый раз перед вызовом *poll*. Массив может быть заполнен один раз и передан в *poll* многократно.

Следующий код демонстрирует способ заполнения *struct poolfd*, вызова *poll* и выбора сокетов с событиями. В примере использован сокет ls — прослушивающий (ожидающий новые подключения), а также и несколько сокетов (N) уже подключенных клиентов.

```
#ifdef _WIN32
	#define WIN32_LEAN_AND_MEAN
	#include <windows.h>
	#include <winsock2.h>
	#pragma comment(lib, "ws2_32.lib")
#else // LINUX
	#include <sys/poll.h>
#endif


	int ls; // Сокет, прослушивающий соединения
	int cs[N]; // Сокеты с подключенными клиентами
	struct pollfd pfd[N+1];
	int i;

	// В отличие от select, массив pfd не обязательно заполнять перед каждым вызовом poll
	for (i = 0; i < N; i++)
	{
```

```
            pfd[i].fd = cs[i];
            pfd[i].events = POLLIN | POLLOUT;
        }

        pfd[N].fd = ls;
        pfd[N].events = POLLIN;

        while (1)
        {
            // Ожидание событий в течение 1 сек
#ifdef _WIN32
            int ev_cnt = WSAPoll(pfd, sizeof(pfd) / sizeof(pfd[0]), 1000);
#else
            int ev_cnt = poll(pfd, sizeof(pfd) / sizeof(pfd[0]), 1000)
#endif
            if (ev_cnt > 0)
            {
                for (i = 0; i < N; i++)
                {
                    if (pfd[i].revents & POLLHUP)
                    {
                        // Сокет cs[i] - клиент отключился. Можно закрывать сокет
                    }

                    if (pfd[i].revents & POLLERR)
                    {
                        // Сокет cs[i] - возникла ошибка. Можно закрывать сокет
                    }

                    if (pfd[i].revents & POLLIN)
                    {
                        // Сокет cs[i] доступен на чтение, можно вызывать recv/recvfrom
                    }

                    if (pfd[i].revents & POLLOUT)
                    {
                        // Сокет cs[i] доступен на запись, можно вызывать send/sendto
                    }
                }

                if (pfd[N].revents & POLLIN)
                {
                    // Сокет ls доступен на чтение - можно вызывать accept, принимать
                    // новое подключение. Новый сокет следует добавить в cs и создать для
                    // него структуру в pfd.
                }
            }
            else
            {
                // Тайтмаут или ошибка
            }
        }
```

Механизм событий сокетов в ОС Windows

Механизм событий сокетов (WSAEvents) позволяет вызывающей программе производить синхронизацию (ожидание) нескольких объектов, при этом и сокеты и другие объекты ядра Windows могут быть использованы однообразно. Например, вызывающая программа может ожидать наступления одного из двух событий: поступления данных от клиента (сокет стал доступен для чтения), либо завершения работы какого-либо внутреннего потока. Поведение программы в дальнейшем

может зависеть от наступившего события. Такое гибкое поведение не было предложено ни одним из универсальных механизмов (*select, poll*).

Для использования данного механизма необходимо создать одно или несколько Событий (*WSAEvent*) функцией *WSACreateEvent*, а затем каждому созданному Событию сопоставить сокет и вид сетевых взаимодействий, при завершении которых Событие перейдет в сигнальное состояние. Одному Событию может быть сопоставлено несколько сокетов, одному сокету может быть сопоставлено несколько Событий. Для сопоставления используется функция *WSAEventSelect*:

```
int WSAEventSelect(SOCKET  s, WSAEVENT hEventObject, long
lNetworkEvents);
```

параметр *s* содержит дескриптор сокета, *hEventObject* — событие WSAEVENT, созданное функцией *WSACreateEvent*, lNetworkEvent — битовая маска, содержащая сетевые взаимодействия, которые необходимо отслеживать и переводить событие *hEventObject* в сигнальное состояние. Поддерживаются следующие биты:

FD_ACCEPT — клиент подключился, соединение можно принять функцией *accept*;

FD_READ — есть данные, отправленные удаленным клиентом, их можно принять с помощью *recv/recvfrom*;

FD_WRITE — сокет готов для передачи данных, можно вызывать *send/sendto*;

FD_CLOSE — соединение разорвано удаленной стороной, сокет можно закрыть функцией *closesocket*;

FD_CONNECT — соединение было установлено (после вызова функции *connect*).

Выбор событий, находящихся в сигнальном состоянии, осуществляется функцией *WSAWaitForMultipleEvents*:

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

cEvents — количество событий, переданных в *lphEvents* (не более чем *WSA_MAXIMUM_WAIT_EVENTS*), *lphEvents* — массив событий, *fWaitAll* — флаг, показывающий что необходимо дождаться срабатывания всех (*TRUE*) или любого (*FALSE*) из перечисленных событий. *dwTimeout* — максимальное время ожидания в миллисекундах, *fAlertable* — обычно не используется и может быть установлено в *FALSE*. По завершении работы функция возвращает *WSA_WAIT_TIMEOUT* в случае возникновения таймаута, либо положительное число.

Вызывающая программа должна проанализировать каждое из событий, находящихся в сигнальном состоянии, считать перечень завершенных сетевых взаимодейтсвий (битовую маску), и выполнить в соответствие с наступившими событиями какие-либо действия. Считывание перечня сетевых взаимодействий сработавшего события осуществляется функцией *WSAEnumNetworkEvents*.

```
int WSAEnumNetworkEvents(SOCKET  s, WSAEVENT hEventObject,
LPWSANETWORKEVENTS lpNetworkEvents);
```

s — дескриптор сокета, с которым связано событие, *hEventObject* — описатель события, *lpNetworkEvents* — указатель на структуру, которая будет заполнена в результате вызова функции. Функция возвращает 0 в случае успешного завершения. Структура *WSANETWORKEVENTS* определена следующим образом:

```
typedef struct _WSANETWORKEVENTS {
  long lNetworkEvents;
  int  iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, *LPWSANETWORKEVENTS;
```

lNetworkEvents содержит битовую маску завершившихся сетевых взаимодейтсвий (*FD_zzz*), *iErrorCode* — коды ошибок, возникших при выполнении каких-либо взаимодействий.

Вторым параметром этой функции является описатель события WSAEVENT, параметр является опциональным. Если параметр указан, то функция сбросит указанное событие, причем сделает это только в том случае, если для указанного сокета есть какие-либо уведомления о завершенных или готовых взаимодействиях.

Безусловный сброс события можно выполнить функцией *WSAResetEvent*. В документации к функции указывается, что правильный путь — поручить сброс события функции WSAEnumNetworkEvents, а не выполнять его с помощью WSAResetEvent. Это объясняется тем, что между обработкой поступивших сигналов WSANETWORKEVENTS и вызовом WSAResetEvent могут поступить новые сигналы, но программа не получит о них никаких сведений и не обработает их, событие будет сброшено, а сигналы — не обработаны.

Корректным является следующий алгоритм: дождаться возникновения события (WSAWaitForMultipleEvents), выполнить сброс всех ожидаемых событий (WSAResetEvent), а затем — выполнить опрос ВСЕХ сокетов, привязанных к указанным событиям, на предмет поступивших сигналов о завершенных взаимодействиях. Тогда, если какие-либо события поступят между вызовами WSAWaitForMultipleEvents и WSAResetEvent — они все равно будут учтены.

Вызов функции *closesocket* автоматически удаляет сокет из всех сопоставленных с ним событий. Освобождение события выполняется функцией *WSACloseEvent*.

Следующий пример кода показывает — каким образом может быть создан массив событий WSAEVENT, выполнено их сопоставление прослушивающему сокету и сокетам подключенных клиентов, а также выполнена выборка сработавших событий, завершенных сетевых взаимодействий и сброс сработавших событий.

```
	int ls; // Прослушивающий сокет
	int cs[N]; // Клиентские соединения
	int cs_cnt = ...; // Количество текущих открытых соединений
	WSAEVENT events[2]; // Первое событие - прослушивающего сокета, второе - клиентских
соединений

	int i;

	events[0] = WSACreateEvent();
```

```
	events[1] = WSACreateEvent();

	WSAEventSelect(ls, events[0], FD_ACCEPT);
	for (i = 0; i < cs_cnt; i++)
		WSAEventSelect(cs[i], events[1], FD_READ | FD_WRITE | FD_CLOSE);

	while (1)
	{
		WSANETWORKEVENTS ne;

		// Ожидание событий в течение секунды
		DWORD dw = WSAWaitForMultipleEvents(2, events, FALSE, 1000, FALSE);

		WSAResetEvent(events[0]);
		WSAResetEvent(events[1]);


		if (0 == WSAEnumNetworkEvents(ls, events[0], &ne) &&
			(ne.lNetworkEvents & FD_ACCEPT) )
		{
			// Поступило событие на прослушивающий сокет, можно принимать подключение
функцией accept
			// Принятый сокет следует добавить в массив cs и подключить его к событию
events[1]

		}

		for (i = 0; i < cs_cnt; i++)
		{
			if (0 == WSAEnumNetworkEvents(cs[i], events[1], &ne))
			{
				// По сокету cs[i] есть события
				if (ne.lNetworkEvents & FD_READ)
				{
					// Есть данные для чтения, можно вызывать recv/recvfrom на
cs[i]
				}

				if (ne.lNetworkEvents & FD_WRITE)
				{
					// Сокет cs[i] готов для записи, можно отправлять данные
				}

				if (ne.lNetworkEvents & FD_CLOSE)
				{
					// Удаленная сторона закрыла соединение, можно закрыть сокет и
удалить его из cs
				}
			}
		}

	}
```

Механизм epoll в ОС Linux

Механизм *epoll* в ОС Linux был разработан как более эффективная замена определенным в POSIX механизмам *select* и *poll*. При использовании данного механизма весь набор дескрипторов, события которых требуется отслеживать, хранится не в памяти программы, а в памяти ядра ОС, поэтому нет необходимости передавать весь массив дескрипторов, как это делалось при вызове *select* или *poll*.

Для реализации механизма ядро ОС поддерживает специальный объект "очередь событий", отвечающий за хранение сокетов, отбор и хранение возникших

событий. Для создания очереди событий программа вызывает функцию *epoll_create*. Единственный аргумент функции не используется, но должен быть больше 0.

Для подключения созданного сокета к очереди используется функция *epoll_ctl*:

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

где *epfd* — очередь, созданная с помощью epoll_create, *op* — производимая операция (*EPOLL_CTL_ADD* — добавление дескриптора в очередь, *EPOLL_CTL_DEL* — удаление), *fd* — сокет, *event* — структура, описывающая перечень отслеживаемых событий:

```
typedef union epoll_data {
        void *ptr;
        int fd;
} epoll_data_t;

struct epoll_event {
    uint32_t events;   /* Epoll events */
    epoll_data_t data;        /* User data variable */
};
```

в поле *events* перечислены события (константы, подобные poll: *EPOLLIN*, *EPOLLOUT*, *EPOLLRDHUP*, *EPOLLHUP* и специальные флаги, такие как *EPOLLET*), в поле data может хранится либо указатель на какие-либо необходимые программе данные, либо дескриптор сокета.

Для чтения событий из очереди используется функция *epoll_wait*:

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents,
int timeout);
```

где *epfd* — очередь, созданная с помощью epoll_create, *events* — указатель на массив структур, который будет заполнен функцией, *maxevents* — максимальное количество событий, которое будет считано за этот вызов, *timeout* — время ожидания событий в миллисекундах. Функция возвращает 0, если произошел таймаут, либо количество сокетов, для которых есть необработанные события.

Следующий пример демонстрирует использование перечисленных функций и реализует простой сервер. От подключившегося клиента ожидается строка, завершаемая символом перевода строки ('\n'). Приняв такую строку, сервер отправляет клиенту сообщение с размером полученной строки и закрывает соединение.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/epoll.h>
#include <string.h>
#include <stdio.h>

#define MAXEVENTS_PER_CALL       (16)
```

```
#define MAX_CLIENTS               (256)

struct client_ctx
{
	int socket;                              // Дескриптор сокета
	unsigned char in_buf[512]; // Буфер принятых данных
	int received;                       // Принято данных
	unsigned char out_buf[512]; // Буфер отправляемых данных
	int out_total;                           // Размер отправляемых данных
	int out_sent;                       // Данных отправлено
};

// Массив структур, хранящий информацию о подлкюченных клиентах
struct client_ctx g_ctxs[1 + MAX_CLIENTS];

int set_non_block_mode(int s)
{
#ifdef _WIN32
	unsigned long mode = 1;
	return ioctlsocket(s, FIONBIO, &mode);
#else
	int fl = fcntl(s, F_GETFL, 0);
	return fcntl(s, F_SETFL, fl | O_NONBLOCK);
#endif
}


int create_listening_socket()
{
	struct sockaddr_in addr;

	int s = socket(AF_INET, SOCK_STREAM, 0);
	if (s <= 0)
		return s;

	set_non_block_mode(s);

	memset(&addr, 0, sizeof(addr));
	addr.sin_family = AF_INET;
	addr.sin_port = htons(9000);

	if (bind(s, (struct sockaddr*) &addr, sizeof(addr)) < 0 ||
		listen(s, 1) < 0)
	{
		printf("error bind() or listen()\n");
		return -1;
	}

	printf("Listening: %hu\n", ntohs(addr.sin_port));

	return s;
}

void epoll_serv()
{
	int s;
	int epfd;
	struct epoll_event ev;
	struct epoll_event events[MAXEVENTS_PER_CALL];

	memset(&g_ctxs, 0, sizeof(g_ctxs));

	// Создание прослушивающего сокета
	s = create_listening_socket();
	if (s <= 0)
```

```
	{
		printf("socket create error: %d\n", errno);
		return;
	}

	// Создание очереди событий
	epfd = epoll_create(1);
	if (epfd <= 0)
		return;

	// Добавление прослушивающего сокета в очередь событий
	ev.events = EPOLLIN;
	ev.data.fd = 0; // В data будет храниться 0 для прослушивающего сокета.
				// Для других сокетов - индекс в массиве ctxs
	if ( 0 != epoll_ctl(epfd, EPOLL_CTL_ADD, s, &ev))
	{
		printf("epoll_ctl(s) error: %d\n", errno);
		return;
	}

	// Бесконечный цикл обработки событий из очереди epfd
	while(1)
	{
		int i, events_cnt;

		// Получение поступивших событий из очереди в теч. 1 секунды
		events_cnt = epoll_wait(epfd, events, MAXEVENTS_PER_CALL, 1000);

		if (events == 0)
		{
			// Никаких событий нет, программа может выполнить другие операции
			// ...
		}

		for(i = 0; i < events_cnt; i++)
		{
			struct epoll_event* e = &events[i];

			if (e->data.fd == 0 && (ev.events & EPOLLIN))
			{
				// Поступило подключение на прослушивающий сокет, принять его
				struct sockaddr_in addr;
				int socklen = sizeof(addr);
				int as = accept(s, (struct sockaddr*) &addr, &socklen);

				if (as > 0)
				{
					int j;
					for(j = 1; j < MAX_CLIENTS; j++)
					{
						if (g_ctxs[j].socket == 0) // Слот свободен, можно занять
сокетом
						{
							memset(&g_ctxs[j], 0, sizeof(g_ctxs[j]));
							g_ctxs[j].socket = as;
							break;
						}
					}

					if (j != MAX_CLIENTS)
					{
						// Регистрация сокета клиента в общей очереди событий
						unsigned int ip = ntohl(addr.sin_addr.s_addr);
						set_non_block_mode(as);
						ev.events = EPOLLIN | EPOLLOUT | EPOLLHUP | EPOLLRDHUP;
```

```
                                        ev.data.fd = j;
                                        epoll_ctl(epfd, EPOLL_CTL_ADD, as, &ev);

                                        printf(" New client connected: %u.%u.%u.%u: %d\n", (ip >>
24) & 0xff, (ip >> 16) & 0xff, (ip >> 8) & 0xff, (ip) & 0xff, j);
                                }
                                else
                                {
                                        // Нет свободных слотов, отключить клиента
                                        close(as);
                                }
                        }
                }

                if (e->data.fd > 0)
                {
                        int idx = e->data.fd;
                        if ( (e->events & EPOLLHUP) || (e->events & EPOLLRDHUP) || (e->events
& EPOLLERR) )
                        {
                                // Клиент отключился (или иная ошибка) => закрыть сокет,
освободить запись о клиенте
                                close(g_ctxs[idx].socket);
                                memset(&g_ctxs[idx], 0, sizeof(g_ctxs[idx]));
                                printf(" Client disconnect: %d (err or hup)\n", idx);
                        }
                        else if ((e->events & EPOLLIN) && (g_ctxs[idx].out_total == 0) &&
(g_ctxs[idx].received < sizeof(g_ctxs[idx].in_buf)) )
                        {
                                // Пришли новые данные от клиента => если еще нет результата -
то принять данные, найти результат и отправить его
                                int r = recv(g_ctxs[idx].socket, g_ctxs[idx].in_buf +
g_ctxs[idx].received, sizeof(g_ctxs[idx].in_buf) - g_ctxs[idx].received, 0 );
                                if (r > 0)
                                {
                                        // Данные приняты, будет проведен анализ
                                        int k;
                                        int len = -1;

                                        g_ctxs[idx].received += r;
                                        for(k = 0; k < g_ctxs[idx].received; k++)
                                        {
                                                if (g_ctxs[idx].in_buf[k] == '\n')
                                                {
                                                        len = k;
                                                        break;
                                                }
                                        }

                                        if (len == -1 && k == sizeof(g_ctxs[idx].in_buf))
                                                len = k;

                                        if (len != -1)
                                        {
                                                // Строка получена, форматирование ответа и
отправка
                                                sprintf(g_ctxs[idx].out_buf, "Your string length:
%d\n", len);
                                                g_ctxs[idx].out_total =
strlen(g_ctxs[idx].out_buf);
                                                g_ctxs[idx].out_sent = 0;
                                                r = send(g_ctxs[idx].socket, g_ctxs[idx].out_buf,
g_ctxs[idx].out_total, MSG_NOSIGNAL);
                                                if (r > 0)
                                                        g_ctxs[idx].out_sent += r;
```

```
                                        }

                                        // Иначе - продолжаем ждать данные от клиента
                                }
                                else
                                {
                                        // Клиент отключился, либо возникла иная ошибка, закрыть
соединение
                                        close(g_ctxs[idx].socket);
                                        memset(&g_ctxs[idx], 0, sizeof(g_ctxs[idx]));
                                        printf(" Client disconnect: %d (read error)\n", idx);
                                }

                        }
                        else if ((e->events & EPOLLOUT) && (g_ctxs[idx].out_total > 0))
                        {
                                // Сокет стал готов к отправке данных => если не все данные
переданы - передать.  Если все данные были переданы, сокет можно закрыть, клиента отключить
                                if (g_ctxs[idx].out_sent < g_ctxs[idx].out_total)
                                {
                                        int r = send(g_ctxs[idx].socket, g_ctxs[idx].out_buf +
g_ctxs[idx].out_sent, g_ctxs[idx].out_total - g_ctxs[idx].out_sent, MSG_NOSIGNAL);
                                        if (r > 0)
                                                g_ctxs[idx].out_sent += r;
                                }

                                if (g_ctxs[idx].out_sent >= g_ctxs[idx].out_total)
                                {

                                        printf(" Response has been sent: %d\n", idx);
                                        close(g_ctxs[idx].socket);
                                        memset(&g_ctxs[idx], 0, sizeof(g_ctxs[idx]));
                                        printf(" Client disconnect: %d (all data sent)\n", idx);
                                }

                        }
                }
        }
}


int main()
{
        epoll_serv();
        return 0;
}
```

Механизм IO Completion ports в ОС Windows

Механизм "портов завершения ввода-вывода" (IO Completion ports) предназначен для создания масштабируемых сетевых приложений, которые могут обрабатывать десятки тысяч одновременных подключений и расходовать при этом минимум системных ресурсов.

Суть механизма сводится к тому, что вызывающая программа инициирует какую-либо сетевую операцию, а уведомление о ее завершении получает позже. Во время выполнения операции вызывающая программа свободна и может выполнять любую другую полезную работу. Во время выполнения такой отложенной операции

в памяти должны оставаться доступными буферы для приема-передачи на все время выполнения асинхронной операции.

Порт завершения создается с помощью функции *CreateIoCompletionPort*:

```
HANDLE io_port = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
```

Для связи сокета с портом необходимо использовать ту же функцию, но передавать в нее описатель сокета и описатель существующего порта:

```
SOCKET s;

// Присоединение существующего сокета s к порту io_port
CreateIoCompletionPort((HANDLE) s, io_port, 0, 0);
```

При установлении связи сокета и порта в третьем параметре указывается уникальный ключ, который может быть использован программой в каких-либо целях. Например, это может быть индекс элемента в массиве, хранящем текущие состояния клиентов. Ключ сообщается системой при завершении каждой отложенной операции.

Подобно тому, как реализуется цикл ожидания событий в других рассмотренных механизмах, в данном случае необходимо реализовать бесконечный цикл ожидания завершающихся событий. Такой цикл может исполняться в одном или нескольких потоках, что в полной мере позволяет использовать преимущества многопроцессорных систем. Чтение уведомления о завершенном событии из порта завершения осуществляется функцией *GetQueuedCompletionStatus*. При вызове функции сообщается уникальный ключ, присвоенный при связи сокета и порта и указатель на структуру *OVERLAPPED*, указанную в начале операции.

При разработке сетевых приложений с использованием механизма портов завершения необходимо использовать расширенные функции для работы с сокетами:

WSASocket — для создания сокета;

AcceptEx — для принятия подключения;

WSASend / WSASendTo — для отправки данных;

WSARecv / WSARecvFrom — для приема данных.

Функции отличаются от стандартных наличием дополнительного параметра — указателя на структуру *OVERLAPPED*, хранящей некоторые данные, необходимые для подсистемы ввода-вывода ОС. При вызове функции данные структуры должны быть заполнены нулями. При завершении операции ввода-вывода система сообщает указатель на переданную в начале операции структуру *OVERLAPPED*. Обычно, это позволяет определить — какая именно операция была завершена.

В случае, если какие-либо операции были начаты, но сокет был закрыт функцией *closesocket*, уведомление об этих операциях все равно поступит в порт завершения ввода-вывода. При этом буферы с данными должны оставаться доступными до момента завершения операций. Чтобы отменить все начатые действия с сокетом, необходимо вызвать *CancelIo*. Вызов функции отменяет не

завершенные операции, но не удаляет уведомления об уже завершенных операциях из порта. Чтобы однозначно определить момент, когда буферы могут быть освобождены, программа может самостоятельно добавить событие в порт завершения. Эта операция выполняется функцией *PostQueuedCompletionStatus*.

Следующий пример демонстрирует использование перечисленных функций и реализует простой сервер. От подключившегося клиента ожидается строка, завершаемая символом перевода строки ('\n'). Приняв такую строку, сервер отправляет клиенту сообщение с размером полученной строки и закрывает соединение.

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#include <mswsock.h>
#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "mswsock.lib")

#include <stdio.h>
struct client_ctx
{
	int socket;

	unsigned char buf_recv[512];	// Буфер приема
	unsigned int sz_recv;			// Принято данных

	unsigned char buf_send[512];	// Буфер отправки
	unsigned int sz_send_total;	// Данных в буфере отправки
	unsigned int sz_send;			// Данных отправлено

	// Структуры OVERLAPPED для уведомлений о завершении
	OVERLAPPED overlap_recv;
	OVERLAPPED overlap_send;
	OVERLAPPED overlap_cancel;

	DWORD flags_recv; // Флаги для WSARecv
};

#define MAX_CLIENTS (100)

// Прослушивающий сокет и все сокеты подключения хранятся
// в массиве структур (вместе с overlapped и буферами)
struct client_ctx g_ctxs[1 + MAX_CLIENTS];
int g_accepted_socket;
HANDLE g_io_port;


// Функция стартует операцию чтения из сокета
void schedule_read(DWORD idx)
{
	WSABUF buf;
	buf.buf = g_ctxs[idx].buf_recv + g_ctxs[idx].sz_recv;
	buf.len = sizeof(g_ctxs[idx].buf_recv) - g_ctxs[idx].sz_recv;

	memset(&g_ctxs[idx].overlap_recv, 0, sizeof(OVERLAPPED));
	g_ctxs[idx].flags_recv = 0;
	WSARecv(g_ctxs[idx].socket, &buf, 1, NULL, &g_ctxs[idx].flags_recv,
&g_ctxs[idx].overlap_recv, NULL);
}

// Функция стартует операцию отправки подготовленных данных в сокет
```

```
void schedule_write(DWORD idx)
{
	WSABUF buf;
	buf.buf = g_ctxs[idx].buf_send + g_ctxs[idx].sz_send;
	buf.len = g_ctxs[idx].sz_send_total - g_ctxs[idx].sz_send;

	memset(&g_ctxs[idx].overlap_send, 0, sizeof(OVERLAPPED));
	WSASend(g_ctxs[idx].socket, &buf, 1, NULL, 0, &g_ctxs[idx].overlap_send, NULL);
}

// Функция добавляет новое принятое подключение клиента
void add_accepted_connection()
{
	DWORD i;
	// Поиск места в массиве g_ctxs для вставки нового подключения
	for (i = 0; i < sizeof(g_ctxs) / sizeof(g_ctxs[0]); i++)
	{
		if (g_ctxs[i].socket == 0)
		{
			unsigned int ip = 0;
			struct sockaddr_in* local_addr = 0, *remote_addr = 0;
			int local_addr_sz, remote_addr_sz;

			GetAcceptExSockaddrs(g_ctxs[0].buf_recv, g_ctxs[0].sz_recv,
				sizeof(struct sockaddr_in) + 16, sizeof(struct sockaddr_in) + 16,
				(struct sockaddr **) &local_addr, &local_addr_sz, (struct sockaddr **)
&remote_addr, &remote_addr_sz);

			if (remote_addr)
				ip = ntohl(remote_addr->sin_addr.s_addr);

			printf(" connection %u created, remote IP: %u.%u.%u.%u\n",
				i, (ip >> 24) & 0xff, (ip >> 16) & 0xff, (ip >> 8) & 0xff, (ip) & 0xff
);

			g_ctxs[i].socket = g_accepted_socket;

			// Связь сокета с портом IOCP, в качестве key используется индекс массива
			if (NULL == CreateIoCompletionPort((HANDLE)g_ctxs[i].socket, g_io_port, i,
0))
			{
				printf("CreateIoCompletionPort error: %x\n", GetLastError());
				return;
			}

			// Ожидание данных от сокета
			schedule_read(i);
			return;
		}
	}

	// Место не найдено => нет ресурсов для принятия соединения
	closesocket(g_accepted_socket);
	g_accepted_socket = 0;
}

// Функция стартует операцию приема соединения
void schedule_accept()
{
	// Создание сокета для принятия подключения (AcceptEx не создает сокетов)
	g_accepted_socket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);

	memset(&g_ctxs[0].overlap_recv, 0, sizeof(OVERLAPPED));

	// Принятие подключения.
```

```
	// Как только операция будет завершена - порт завершения пришлет уведомление.
	// Размеры буферов должны быть на 16 байт больше размера адреса согласно документации
разработчика ОС
	AcceptEx(g_ctxs[0].socket, g_accepted_socket, g_ctxs[0].buf_recv, 0,
		sizeof(struct sockaddr_in) + 16, sizeof(struct sockaddr_in) + 16, NULL,
&g_ctxs[0].overlap_recv);
}

int is_string_received(DWORD idx, int* len)
{
	DWORD i;
	for (i = 0; i < g_ctxs[idx].sz_recv; i++)
	{
		if (g_ctxs[idx].buf_recv[i] == '\n')
		{
			*len = (int) (i + 1);
			return 1;
		}
	}

	if (g_ctxs[idx].sz_recv == sizeof(g_ctxs[idx].buf_recv))
	{
		*len = sizeof(g_ctxs[idx].buf_recv);
		return 1;
	}

	return 0;
}

void io_serv()
{
	struct sockaddr_in addr;

	// Создание сокета прослушивания
	SOCKET s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);

	// Создание порта завершения
	g_io_port = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

	if (NULL == g_io_port)
	{
		printf("CreateIoCompletionPort error: %x\n", GetLastError());
		return;
	}

	// Обнуление структуры данных для хранения входящих соединений
	memset(g_ctxs, 0, sizeof(g_ctxs));

	memset(&addr, 0, sizeof(addr));
	addr.sin_family = AF_INET;
	addr.sin_port = htons(9000);

	if (bind(s, (struct sockaddr*) &addr, sizeof(addr)) < 0 ||
		listen(s, 1) < 0)
	{
		printf("error bind() or listen()\n");
		return;
	}

	printf("Listening: %hu\n", ntohs(addr.sin_port));

	// Присоединение существующего сокета s к порту io_port.
	// В качестве ключа для прослушивающего сокета используется 0
	if (NULL == CreateIoCompletionPort((HANDLE)s, g_io_port, 0, 0))
	{
```

```
		printf("CreateIoCompletionPort error: %x\n", GetLastError());
		return;
	}

	g_ctxs[0].socket = s;

	// Старт операции принятия подключения.
	schedule_accept();

	// Бесконечный цикл принятия событий о завершенных операциях
	while (1)
	{
		DWORD transferred;
		ULONG_PTR key;
		OVERLAPPED* lp_overlap;

		// Ожидание событий в течение 1 секунды
		BOOL b = GetQueuedCompletionStatus(g_io_port, &transferred, &key, &lp_overlap,
1000);
		if (b)
		{
			// Поступило уведомление о завершении операции
			if (key == 0) // ключ 0 - для прослушивающего сокета
			{
				g_ctxs[0].sz_recv += transferred;

				// Принятие подключения и начало принятия следующего
				add_accepted_connection();
				schedule_accept();
			}
			else
			{
				// Иначе поступило событие по завершению операции от клиента.
				// Ключ key - индекс в массиве g_ctxs

				if (&g_ctxs[key].overlap_recv == lp_overlap)
				{
					int len;

					// Данные приняты:
					if (transferred == 0)
					{
						// Соединение разорвано
						CancelIo((HANDLE)g_ctxs[key].socket);
						PostQueuedCompletionStatus(g_io_port, 0, key,
&g_ctxs[key].overlap_cancel);
						continue;
					}

					g_ctxs[key].sz_recv += transferred;

					if (is_string_received(key, &len))
					{
						// Если строка полностью пришла, то сформировать ответ и
начать его отправлять
						sprintf(g_ctxs[key].buf_send, "You string length: %d\n",
len);
						g_ctxs[key].sz_send_total = strlen(g_ctxs[key].buf_send);
						g_ctxs[key].sz_send = 0;
						schedule_write(key);
					}
					else
					{
						// Иначе - ждем данные дальше
						schedule_read(key);
```

```
                                }
                        }
                        else if (&g_ctxs[key].overlap_send == lp_overlap)
                        {
                                // Данные отправлены
                                g_ctxs[key].sz_send += transferred;
                                if (g_ctxs[key].sz_send < g_ctxs[key].sz_send_total &&
transferred > 0)
                                {
                                        // Если данные отправлены не полностью - продолжить
отправлять
                                        schedule_write(key);
                                }
                                else
                                {
                                        // Данные отправлены полностью, прервать все
коммуникации,
                                        // добавить в порт событие на завершение работы
                                        CancelIo((HANDLE)g_ctxs[key].socket);
                                        PostQueuedCompletionStatus(g_io_port, 0, key,
&g_ctxs[key].overlap_cancel);
                                }
                        }
                        else if (&g_ctxs[key].overlap_cancel == lp_overlap)
                        {
                                // Все коммуникации завершены, сокет может быть закрыт
                                closesocket(g_ctxs[key].socket);
                                memset(&g_ctxs[key], 0, sizeof(g_ctxs[key]));
                                printf(" connection %u closed\n", key);
                        }
                }
        }
        else
        {
                // Ни одной операции не было завершено в течение заданного времени, программа
может
                // выполнить какие-либо другие действия

                // ...
        }
    }
}

int main()
{
    init();
    io_serv();

    return 0;
}
```

Для проверки и отладки сервера в качестве клиентской программы можно использовать telnet, запустив утилиту с подключением к локальному адресу: *telnet 127.0.0.1 9000*

Порядок выполнения работы

1. Получить у преподавателя номер задания (*Приложение*).
2. Выполнить задание.
3. Ответить на контрольные вопросы.

Содержание отчета

1. Цели работы.
2. Задачи работы.
3. Описание алгоритма параллельного обслуживания клиентов для программы-сервера. Блок-схема алгоритма.
4. Описание применяемых в сервере структур данных и функций: назначение структур данных и их полей, таблица с именами и назначением реализованных функций.
5. Описание структуры кода и алгоритма программы-клиента.
6. Временна́я диаграмма взаимодействия сервера и клиента.
7. Описание тестовых ситуаций: набор входных данных и ожидаемый результат. Набор тестов должен быть достаточный для проверки корректной работы программ как для "нормальных", так и "граничных" случаев. Тесты отличаются длиной сообщений, количеством сообщений, количеством одновременно подключенных клиентов, режимом работы клиентов.
8. Тексты программ клиента и сервера с подробными комментариями.

Примечание: пп.3-8 описываются в отчете для каждого из подзаданий: I. TCP, II. UDP

9. Выводы по работе.