ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 20 — Сокеты

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2023

2023.06.07

Оглавление

Сокеты

«Сокет» — это обобщенный канал межпроцессного взаимодействия.
Представлен как файловый дескриптор.

Типы сокетов

Символьные константы, определяющие поддерживаемые типы сокетов, определены в **`sys/socket.h`**.

SOCK_STREAM

Тип **`SOCK_STREAM`** работает через соединение с конкретным удаленным сокетом и надежно передает данные в виде *потока байтов*.

SOCK_DGRAM

Тип **`SOCK_DGRAM`** используется для ненадежной отправки пакетов с и*ндивидуальной адресацией*. Это полная противоположность **`SOCK_STREAM`**. Каждый раз, когда записываются данные в такой сокет, эти данные становятся одним пакетом.Поскольку сокеты **`SOCK_DGRAM`** не имеют соединений, необходимо указывать адрес получателя для каждого пакета.

SOCK_RAW

Этот тип обеспечивает доступ к сетевым протоколам и интерфейсам низкого уровня.

SOCK_SEQPACKET

Обеспечивает работу последовательного двустороннего канала для передачи дейтаграмм на основе соединений. Дейтаграммы имеют постоянный размер.

Некоторые типы сокетов могут быть не реализованы для всех семейств протоколов.

Адреса сокетов

Имя сокета обычно называется «адресом». Когда речь идет о сокетах, можно рассматривать термины «имя» и «адрес» как синонимы.

У нового сокета, созданного с помощью функции **`socket()`**, нет адреса.

Другие процессы могут найти сокет для связи, только если задать адрес.

Это называется «привязкой» адреса к сокету, и это можно сделать с помощью функции **`bind()`**.

Подробная информация об адресах сокетов зависит от того, какое пространство имен используется, например, локальное пространство имен или интернет-пространство.

Независимо от пространства имен для установки и проверки адреса сокета используются одни и те же функции **`bind()`** и **`getsockname()`**.

Эти функции используют фиктивный тип данных **`struct sockaddr *`** для получения адреса в качестве параметров.

На практике адрес находится в структуре некоторого другого типа данных, подходящего для того формата адреса, который в данный момент используется, но при передаче его в **`bind()`** он приводится к **`struct sockaddr *`**.

Для правильной интерпретации адреса или его создания необходимо использовать правильный тип данных для пространства имен сокета.

Обычная практика состоит в том, что создается адрес надлежащего типа, зависящего от пространства имен и используемого семейства протоколов, а затем при вызове **`bind()`** или **`getsockname()`** адрес приводится к указателю на **`struct sockaddr *`**.

Единственная информация, можно получить из типа данных **`struct sockaddr`**, — это код семейства протоколов, который определяет формат адреса.

`struct sockaddr` имеет следующие члены:

`short int sa_family` — код формата адреса. Он определяет формат данных, которые содержатся в объекте типа **`sockaddr`**.

`char sa_data[14]` — фактические данные адреса сокета, которые зависят от формата.

Его длина также зависит от формата и вполне может быть больше 14. Длина **`sa_data`**, равная 14, по существу произвольна.

Каждый формат адреса имеет символическое имя, которое начинается с **`AF_`**.

Каждому из них соответствует символ **`PF_`**, который обозначает соответствующее пространство имен. Ниже приведен список некоторых имен форматов адресов:

AF_LOCAL (AF_UNIX, AF_FILE) – обозначает формат адреса, который соответствует локальному пространству имен **`PF_LOCAL`**.

AF_INET – обозначает формат адреса, который соответствует пространству имен Internet.

AF_INET6 – похоже на `AF_INET`, но относится к протоколу IPv6.

AF_UNSPEC – не означает никакого конкретного формата адреса и используется только в редких случаях, например, для очистки адреса назначения по умолчанию у «подключенного» дейтаграммного сокета.

`sys/socket.h` определяет символы, начинающиеся с «AF_» для многих различных типов сетей, большинство или все из которых фактически не реализованы.

Работа с сокетами

Для работы с ними существует восемь основных системных вызовов, пять из которых предназначены исключительно для сокетов:

```
socket()   -- создать сокет
bind()     -- связать сокет с локальным адресом
listen()   -- отметить сокет принимающим запросы на соединение от других сокетов
accept()   -- ожидать запроса на соединение
connect()  -- запросить соединение
read()     -- читать из сокета
write()    -- писать в сокет
close()    -- закрыть сокет
```

Существует еще около 60 системных вызовов, имеющих отношение к сокетам. Они используются для отправки или получения пакетов, а также для выполнения других операций с сокетами.

socketpair(2) возвращает два подключенных анонимных сокета (реализовано только для нескольких локальных семейств, таких как **AF_UNIX**).

send(2), **sendto(2)** и **sendmsg(2)** отправляют данные через сокет.

recv(2), **recvfrom(2)**, **recvmsg(2)** получают данные из сокета.

poll(2) и **select(2)** ожидают поступления данных или готовности к отправке данных.

writev(2), **sendfile(2)** и **readv(2)**

getsockname(2) возвращает адрес локального сокета

getpeername(2) возвращает адрес удаленного сокета

getsockopt(2) и **setsockopt(2)** используются для установки или получения параметров уровня сокета или протокола.

ioctl(2) можно использовать для установки или чтения некоторых других параметров.

shutdown(2) закрывает части полнодуплексного сокетного соединения.

Можно выполнять неблокирующий ввод-вывод на сокетах, установив флаг **O_NONBLOCK** в дескрипторе файла сокета с помощью **fcntl(2)**.

В этом случае все операции, которые могут заблокировать, будут (обычно) возвращаться с **EAGAIN** (операцию следует повторить позже).

connect(2) вернет ошибку **EINPROGRESS**.

Затем пользователь может ждать различных событий с помощью **poll(2)** или **select(2)**.

Последовательность создания и инициализации сокетов на стороне сервера и клиента.

	Сервер		**Клиент**
1)	socket	1)	socket
2)	bind "superserver"		
3)	listen	2)	connect "superserver"
4)	accept		
5)	read/write	5)	read/write

На стороне сервера:

1) Вызывается **socket** для создания объекта и связанного с ним файлового дескриптора.

2) Вызывается **bind**, для назначения сокету адреса (имени) .

3) Вызывается **listen**, чтобы пометить сокет, как предназначенный для приема запросов на соединение.

4) Вызывается **accept**, чтобы дождаться и принять входящее соединение. После этого вызов **accept** создает второй сокет с новым файловым дескриптором, который и используется для обмена данными.

5) В операциях ввода-вывода (системные вызовы **read** и **write**) участвует созданный (второй) файловый дескриптор.

На стороне клиента:

1) Вызывается **socket** для создания объекта и связанного с ним файлового дескриптора.

2) Для установления соединения с сервером вызывается **connect**, которому в качестве аргумента передается имя сервера, (этот шаг, не обязательно должен быть синхронизирован со вторым шагом на стороне сервера).

5) Для выполнения операций ввода-вывода используется файловый дескриптор сокета.

select() — ожидает готовности операций ввода-вывода

```
// В соответствии POSIX.1-2001, POSIX.1-2008
#include <sys/select.h>

// в соответствии с ранними стандартами
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int     nfds,              // количество всех файловых дескрипторов
           fd_set *readfds,           // набор дескрипторов для чтения
           fd_set *writefds,          // набор дескрипторов для записи
           fd_set *exceptfds,         // набор дескрипторов для обнаружения
                                      // исключительных условий.
           struct timeval *timeout); // время ожидания

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);            // очищает набор
```

Вызов **select()** позволяет программам отслеживать изменения нескольких файловых дескрипторов ожидая, когда один или более файловых дескрипторов станут «готовы» для операции ввода-вывода определённого типа (например, ввода).

Файловый дескриптор считается готовым, если к нему возможно применить соответствующую операцию ввода-вывода, например, **read()** или очень «маленький» **write()** без блокировки.

Вызов **select()** может следить только за номерами файловых дескрипторов, которые меньше значения **FD_SETSIZE** (вызов **poll()** не имеет этого ограничения).

timeout — указывается интервал, на который должен заблокироваться **select()** в ожидании готовности файлового дескриптора. Вызов будет блокироваться пока:

- файловый дескриптор не станет готов;
- вызов не прервётся обработчиком сигнала;
- не истечёт время ожидания.

Время ожидания задаётся секундами и микросекундами.

```
struct timeval {
    long tv_sec;   // секунды
    long tv_usec;  // микросекунды
};
```

Интервал **timeout** будет округлён с точностью системных часов, а из-за задержки при планировании в ядре блокирующий интервал будет немного больше.

Если оба поля структуры **timeval** равны нулю, то **select()** завершится немедленно (полезно при опросе (polling)).

Если значение **timeout** равно **NULL** (время ожидания не задано), то **select()** может быть заблокирован в течение неопределённого времени.

```
int select(int     nfds,              // количество всех файловых дескрипторов
           fd_set *readfds,           // набор дескрипторов для чтения
           fd_set *writefds,          // набор дескрипторов для записи
           fd_set *exceptfds,         // набор дескрипторов для обнаружения
                                      // исключительных условий.
           struct timeval *timeout); // время ожидания
```

Отслеживаются 3 независимых набора файловых дескрипторов.

readfds — в тех, что перечислены в **readfds**, будет отслеживаться появление символов, доступных для чтения (проверяется доступность чтения без блокировки). В частности, если файловый дескриптор указывает на конец файла, он считается готовым для чтения (EOF).

writefds — файловые дескрипторы, указанные в **writefds**, будут отслеживаться для возможности записи без блокировки, если доступно пространство для записи. Однако, при большом количестве данных для записи блокировка по-прежнему будет выполнена.

exceptfds — файловые дескрипторы, указанные в **exceptfds**, будут отслеживаться для обнаружения исключительных условий.

Значение каждого из трёх наборов файловых дескрипторов может быть задано как **NULL**, если слежение за определённым классом событий над файловыми дескрипторами не требуется. Иногда **select()** вызывается с пустыми наборами (всеми тремя), **nfds** равным нулю и непустым **timeout** для переносимой реализации перехода в режим ожидания (sleep) на периоды с точностью менее секунды.

Значение **nfds** должно быть на единицу больше самого большого номера файлового дескриптора из всех трёх наборов плюс 1 — указанные файловые дескрипторы в каждом наборе проверяются на этот порог.

Для манипуляций наборами существуют четыре макроса:

FD_ZERO() — очищает набор;
FD_SET() — добавляет заданный файловый дескриптор к набору;
FD_CLR() — удаляет файловый дескриптор из набора;
FD_ISSET() — проверяет, является ли файловый дескриптор частью набора.

Эти макросы полезны после возврата из вызова **`select()`**.

Тип **`fd_set`** представляет собой буфер фиксированного размера.

При возврате из вызова наборы файловых дескрипторов изменяются, показывая какие файловые дескрипторы фактически изменили состояние.

Если используется `select()` в цикле, то наборы дескрипторов должны заново инициализироваться перед каждым вызовом.

Возвращает

При успешном выполнении **`select()`** возвращает количество файловых дескрипторов, находящихся в трёх возвращаемых наборах (то есть, общее количество бит, установленных в **`readfds`**, **`writefds`**, **`exceptfds`**).

Это количество может быть нулевым, если время ожидания истекло, а интересующие события так и не произошли.

При ошибке возвращается значение -1, а переменной **`errno`** присваивается соответствующий номер ошибки. Наборы файловых дескрипторов в этом случае не изменяются, а значение **`timeout`** становится неопределённым.

Ошибки

EBADF — в одном из наборов находится неверный файловый дескриптор (возможно файловый дескриптор уже закрыт, или при работе с ним произошла ошибка).

EINTR — при выполнении поступил сигнал.

EINVAL — значение **`nfds`** отрицательно или превышает ограничение ресурса **RLIMIT_NOFILE** (**`getrlimit(2)`**).

EINVAL — значение, содержащееся внутри timeout, некорректно.

ENOMEM — не удалось выделить память для внутренних таблиц.

Пример

```
include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {

    fd_set         rfds;
    struct timeval tv;
    int            retval;

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);  // Следить, когда что-нибудь появится в stdin (fd 0)
    tv.tv_sec  = 5;    // Ждать не больше пяти секунд.
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Есть данные.\n");  // FD_ISSET(0, &rfds) будет TRUE
    else
        printf("Данные не появились в течение пяти секунд.\n");
    exit(EXIT_SUCCESS);
}
```

Пример: Обслуживание нескольких клиентов

Код сервера

```
static int run_server(struct sockaddr_un *sap) {

    int fd_server, fd_client, fd_hwm = 0, fd; // файловые дескрипторы
    char buf[100];
    fd_set set, read_set; // наборы дескрипторов для отслеживания select()

    fd_server = socket(AF_UNIX, SOCK_STREAM, 0);
    bind(fd_server, (struct sockaddr *)sap, sizeof(*sap));
    listen(fd_server, SOMAXCONN);
    if (fd_server > fd_hwm)
        fd_hwm = fd_server;
    FD_ZERO(&set);           // очистка select():fd_set
    FD_SET(fd_server, &set); // регистрация серверного дескриптора
    for (;;) {
        read_set = set;      // копия набора отслеживаемых дескрипторов
        select(fd_hwm + 1, &read_set, NULL, NULL, NULL);
        for (fd = 0; fd <= fd_hwm; fd++) {
            if (FD_ISSET(fd, &read_set)) {
                if (fd == fd_server) { // запрос на соединение
                    fd_client = accept(fd_server, NULL, 0);
                    FD_SET(fd_client, &set);
                    if (fd_client > fd_hwm) {
                        fd_hwm = fd_client;
                    }
```

```
                } else { // готовность ввода-вывода
                    ssize_t nread = read(fd, buf, sizeof(buf));
                    if (nread == 0) {     // конец файла
                        FD_CLR(fd, &set);
                        if (fd == fd_hwm) {
                            fd_hwm--;
                        }
                        close(fd);
                    } else {
                        printf("Сервер получил сообщение \"%s\"\n", buf);
                        write(fd, "Bye!", sizeof("Bye!"));
    }   }   }   }   }
    return 1;
}
```

Используется два набора дескрипторов — **set** хранит все файловые дескрипторы сокетов — сокет сервера, который принимает запросы на соединение, и сокеты связи с клиентами, а **read_set**, копируется из набора **set** перед каждым обращением к **select()**, поскольку **select()** модифицирует переданный ему набор, возвращая список готовых дескрипторов .

Для корректной работы **select()** необходимо отслеживать наибольший номер файлового дескриптора — **fd_hwm** обновляется всякий раз при создании нового дескриптора вызовом **accept()**, и при закрытии файлового дескриптора с наибольшим порядковым номером.

После закрытия дескриптор удаляется из набора — это совершенно необходимо, в противном случае **select()** будет сообщать, что дескриптор готов для выполнения операции чтения, не в том смысле, что есть данные для чтения, а в том смысле, что вызов **read()** не будет блокироваться.

Код клиента

```
static int run_client(struct sockaddr_un *sap) {

    if (fork() == 0) {
        int fd_client;
        char buf[100];

        fd_client = socket(AF_UNIX, SOCK_STREAM, 0);
        while (connect(fd_client, (struct sockaddr *)sap, sizeof(*sap)) == -1) {
            if (errno == ENOENT) {
                sleep(1);
                continue;
            } else {
                perror("connect");
                exit(errno);
            }
        }
        snprintf(buf, sizeof(buf), "Привет от клиента %ld", (long)getpid());
        write(fd_client, buf, sizeof(buf));
        read(fd_client, buf, sizeof(buf));
        printf("Клиент получил сообщение \"%s\"\n", buf);
        close(fd_client);
        exit(EXIT_SUCCESS);
    }
    return 1;
}
```

Функция main()

```
int main(void) {

    struct sockaddr_un sa;
    int nclient;
    (void)unlink(SOCKETNAME);
    sa.sun_family = AF_UNIX;
    for (nclient = 1; nclient <= 4; nclient++) {
        run_client(&sa);
    }
    run_server(&sa);
    exit(EXIT_SUCCESS);
}
```

В результате запуска было получено:

```
$ ./Debug/usocket
Сервер получил сообщение "Привет от клиента 286001!"
Клиент получил сообщение "OK! Good Bye..."
Сервер получил сообщение "Привет от клиента 285999!"
Клиент получил сообщение "OK! Good Bye..."
Сервер получил сообщение "Привет от клиента 286000!"
Клиент получил сообщение "OK! Good Bye..."
Сервер получил сообщение "Привет от клиента 285998!"
Клиент получил сообщение "OK! Good Bye..."
```

Порядок следования байт

При обмене однобайтными символами между разными платформами не возникает никаких проблем — они везде трактуются одинаково. Проблемы возникают при обмене многобайтовыми числами. Причина — разный порядок следования байт.

Различают прямой (BigEndian) и обратный (LittleEndian) порядок следования байт.

В архитектурах с ***прямым порядком следования байт*** адрес числа определяется адресом его старшего байта. Это значит, что число 12345678h будет располагаться в памяти следующим образом — 12 34 56 78.

В архитектурах с ***обратным порядком следования байт*** адрес числа определяется адресом его младшего байта и число 12345678h будет располагаться в памяти так — 78 56 34 12.

Это же касается и многобайтовых символов Unicode.

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011
Q	w	e	R	t	Y		§	Д (UTF-8)		Д (UTF-16BE)		Д (UTF-16LE)		𝄞			
51	77	65	52	74	59	00	FD	D4	A4	05	24	24	05	F0	9D	84	9E

@0000 — строка ASCII символов «QWERTY» или 'Q','W','E','R','T','Y','\0'
— байт со значением 0x51 (81)
— двубайтное слово со значением 0x7751 (LE) или 0x5177 (BE)
— четырехбайтное слово со значением 0x52657751 (LE) или 0x51776552 (BE)
@0007 — символ «§» (параграф) или байт со значением 0xFD (253 или -3)
@0008 — символ юникода U+0524 в кодировке UTF-8 со значением 0xD4A4
@000A — символ юникода U+0524 в кодировке UTF-16BE со значением 0x0524
@000C — символ юникода U+0524 в кодировке UTF-16LE со значением 0x2405
@000E — символ юникода U+1D11E в кодировке UTF-8 со значением 0xF09D849E

Для преобразования из BigEndian в LittleEndian и обратно существует ряд специальных функций.

htonl, htons, ntohl, ntohs — преобразование данных BigEndian/LittleEndian

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);  // из порядка узла (host) в сетевой (network)
uint16_t htons(uint16_t hostshort); // из порядка узла в сетевой
uint32_t ntohl(uint32_t netlong);   // из сетевого в порядок узла
uint16_t ntohs(uint16_t netshort);  // из сетевого в порядок узла
```

Функция **htonl()** преобразует значение беззнакового целого **hostlong** из узлового порядка расположения байтов в сетевой порядок расположения байтов.

Функция **htons()** преобразует значение короткого беззнакового целого **hostshort** из узлового порядка расположения байтов в сетевой порядок расположения байтов.

Функция **ntohl()** преобразует значение беззнакового целого **netlong** из сетевого порядка расположения байтов в узловой порядок расположения байтов.

Функция **ntohs()** преобразует значение короткого беззнакового целого **netshort** из сетевого порядка расположения байтов в узловой порядок расположения байтов.

В архитектуре x86-32/64 используется узловой порядок расположения байтов — в начале числа стоит наименее значимый байт (LittleEndian), в то время как сетевым порядком байт, используемым в интернет, считается BigEndian (в начале числа стоит наиболее значимый байт).

poll(), ppoll() — ожидает некоторое событие над файловым дескриптором

```
#include <poll.h>

int poll(struct pollfd *fds,         // Отслеживаемый набор файловых дескрипторов
         nfds_t         nfds,        // Количество элементов в массиве fds
         int            timeout);    //
```

Вызов **poll()** выполняет сходную с **select(2)** задачу – он ждёт пока один дескриптор из набора файловых дескрипторов не станет готов выполнить операцию ввода-вывода.

Отслеживаемый набор файловых дескрипторов задаётся в аргументе **fds**, который представляет собой массив структур:

```
struct pollfd {
    int   fd;         // файловый дескриптор
    short events;     // запрашиваемые события
    short revents;    // возвращённые события
};
```

Количество элементов в массиве **fds** указывается в аргументе **nfds**.

В поле **fd** содержится файловый дескриптор открытого файла.

Если значение поля отрицательно, то соответствующее поле **events** игнорируется, а в поле **revents** возвращает ноль (простой способ игнорирования файлового дескриптора в одиночном вызове **poll()** – просто сделать значение поля **fd** отрицательным.

Заметим, что это нельзя использовать для игнорирования файлового дескриптора 0).

```
struct pollfd {
    int   fd;         // файловый дескриптор
    short events;     // запрашиваемые события
    short revents;    // возвращённые события
};
```

Поле **events** представляет собой входной параметр — битовую маску, указывающую на события, происходящие с файловым дескриптором **fd**, которые важны для приложения.

Если это поле равно нулю, то возвращаемыми событиями в **revents** могут быть флаги ошибочных состояний – **POLLHUP**, **POLLERR** и **POLLNVAL**.

В поле **revents** ядро помещает информацию о произошедших событиях.

В **revents** могут содержаться любые битовые флаги из задаваемых в **events**, или там может быть одно из значений — **POLLERR**, **POLLHUP** или **POLLNVAL**.

Эти три битовых флага не имеют смысла в поле **events**, но будут установлены в поле **revents**, если соответствующее условие истинно.

Если ни одно из запрошенных событий с файловыми дескрипторами не произошло или не возникло ошибок, то **poll()** блокируется до их появления.

В **timeout** указывается количество миллисекунд, на которые будет блокироваться **poll()** в ожидании готовности файлового дескриптора. Вызов будет заблокирован до тех пор, пока:

- файловый дескриптор не станет готов;
- вызов не прервётся обработчиком сигнала;
- не истечёт время ожидания.

Интервал **timeout** будет округлён с точностью системных часов, а из-за задержки при планировании в ядре блокирующий интервал будет немного больше.

Отрицательное значение в **timeout** означает бесконечное ожидание.

Значение **timeout**, равное нулю, приводит к немедленному завершению **poll()**, даже если нет ни одного готового файлового дескриптора.

В **events** / **revents**: могут быть установлены / получены следующие биты:

POLLIN – есть данные для чтения.

POLLOUT – теперь возможна запись, но запись данных больше, чем доступно места в сокете или канале, по-прежнему приводит к блокировке, если не указан **O_NONBLOCK**.

POLLPRI – исключительное состояние файлового дескриптора. Может быть из-за появления внеполосных данных в сокете TCP (tcp(7)) или событий, связанных с терминалом.

POLLRDHUP – удалённая сторона потокового сокета закрыла соединение, или отключила запись в одну сторону. Для использования данного флага должен быть определён макрос тестирования свойств **_GNU_SOURCE** (до включения каких-либо заголовочных файлов).

POLLERR – состояние ошибки (возвращается только в **revents** и игнорируется в **events**).

Также этот бит устанавливается для файлового дескриптора, указывающего в пишущий конец канала при закрытом читающем конце.

POLLHUP – зависание (hang up, возвращается только в **revents** и игнорируется в **events**).

При чтении из канала, такого как канал (pipe) или потоковый сокет, это событие всего-навсего показывает, что партнёр закрыл канал со своего конца. Дальнейшее чтение из канала будет возвращать 0 (конец файла) только после потребления всех неполученных данных в канале.

POLLNVAL – неверный запрос – **fd** не открыт (возвращается только в **revents** и игнорируется в **events**).

При компилировании с установленным **_XOPEN_SOURCE** также определены следующие значения, которые не передают дополнительной информации вне упомянутых выше битов:

POLLRDNORM — эквивалентно **POLLIN**.

POLLRDBAND — доступны для чтения приоритетные внутриполосные данные (в Linux, обычно, не используется).

POLLWRNORM — эквивалентно POLLOUT.

POLLWRBAND — можно писать приоритетные данные.

В Linux также есть **POLLMSG**, но он не используется.

Возвращаемое значение

При успешном выполнении возвращается положительное значение — оно означает количество структур, в которых поля **revents** имеют ненулевое значение, т.е. количество дескрипторов, для которых возникли события или ошибки.

Значение 0 означает, что время ожидания истекло, и нет готовых файловых дескрипторов.

В случае ошибки возвращается -1, а **errno** устанавливается в соответствующее значение.

Ошибки

EFAULT — указанный аргументом массив содержится вне адресного пространства вызывающей программы.

EINTR — получен сигнал раньше какого-либо запрашиваемого события (signal(7)).

EINVAL — значение **nfds** превышает значение **RLIMIT_NOFILE**.

ENOMEM — нет места под таблицы файловых дескрипторов.

Структуры адресов сокетов

Для каждого из доменов адресов предназначена своя структура и свой собственный заголовок:

AF_UNIX – **sockaddr_un**;
AF_INET – **sockaddr_in**;
AF_INET6 – **sockaddr_in**;
AF_X25 – **sockaddr_x25.** (/usr/include/sys/socket.h)
…
В [SUSv.4-2018] стандартизованы только первые три типа.

Есть несколько подходов к решению проблемы специфической структуры адреса для разных семейств имен.

struct sockaddr

Данный подход используется для адресации типа **AF_UNIX**. Он заключается в том, чтобы разместить в памяти специфическую для домена структуру, заполнить необходимые поля и передать указатель на нее системному вызову **bind()** или **connect()**, приведя указатель к типу **struct sockaddr ***.

sockaddr_storage

Подход заключается в том, что для хранения адресов сокетов используются переменные типа **sockaddr_storage**, которые гарантирует достаточный объем пространства в памяти под адрес любого типа.

Есть правило:

- если точно известно, с каким доменом адресов предстоит работать, следует объявлять переменную соответствующего типа (например **sockaddr_un**) или размещать ее в динамической памяти (malloc( ));

- если необходимо предусмотреть возможность создания сокетов с адресацией любого типа, следует использовать тип **sockaddr_storage**, но при обращении к таким переменным необходимо будет выполнять приведение к конкретному типу;

- при обращении к системным вызовам **bind( )** и **connect( )**, в любом случае должно выполняться приведение типов в соответствии с их прототипами.

Пример

```
struct sockaddr_storage  sas;
struct sockaddr_un      *sa = (struct sockaddr_un *)&sas;
sa->sun_family = AF_UNIX;
...
bind(fd, (struct sockaddr *)sa, sizeof(*sa));
```

Структура типа **sockaddr_storage** полностью скрыта — нет ни одного поля, к которому можно обратиться и, для того, чтобы ее проинициализировать, ее необходимо привести к конкретному типу.

send(), sendto(), sendmsg() — отправляет сообщения в сокет

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int         sockfd, // отправляющий сокет
             const void *buf,    // буфер с данными
             size_t      len,    // длина данных
             int         flags); //

ssize_t sendto(int         sockfd,              // отправляющий сокет
               const void *buf,                // буфер с данными
               size_t      len,                // длина данных
               int         flags,              //
               const struct sockaddr *dest_addr, // адрес назначения
               socklen_t   addrlen);           // длина адреса

ssize_t sendmsg(int sockfd,               // отправляющий сокет
                const struct msghdr *msg, //
                int flags);               //
```

Системные вызовы **send()**, **sendto()** и **sendmsg()** используются для пересылки сообщений в другой сокет.

Вызов **send()** можно использовать, только если сокет находится в состоянии соединения, то есть если известен получатель.

Вызов **send()** отличается от **write(2)** только наличием аргумента **flags**.

Если значение **flags** равно нулю, то вызов **send()** эквивалентен **write(2)**.

У **send()** и **sendto()** сообщение находится в **buf**, а его длина в **len**.

```
send(sockfd, buf, len, flags);
```

эквивалентен

```
sendto(sockfd, buf, len, flags, NULL, 0);
```

Аргумент **sockfd** представляет файловый дескриптор сокета отправления.

Если **sendto()** используется с сокетом в режиме с установлением соединения (**SOCK_STREAM**, **SOCK_SEQPACKET**), то аргументы **dest_addr** и **addrlen** игнорируются. При этом, если их значения не равны **NULL** и 0, может быть возвращена ошибка **EISCONN**.

Если соединение через сокет не установлено, возвращается ошибка **ENOTCONN**.

А вообще в **dest_addr** задаётся адрес назначения и его размер в **addrlen**.

Определение структуры **msghdr**, используемой **sendmsg()**:

```
struct msghdr {
    void         *msg_name;       // необязательный адрес
    socklen_t     msg_namelen;    // размер адреса
    struct iovec *msg_iov;        // массив приёма/передачи
    size_t        msg_iovlen;     // количество элементов в msg_iov
    void         *msg_control;    // вспомогательные данные
    size_t        msg_controllen; // размер буфера вспомогательных данных
    int           msg_flags;      // флаги (не используется)
};
```

Поле **msg_name** используется на неподключённом сокете для указания адреса назначения дейтаграммы. Оно указывает на буфер с адресом.

В поле **msg_namelen** должен быть указан размер адреса.

Для подключённого сокета значения этих полей должны быть равны **NULL** и 0, соответственно.

В полях **msg_iov** и **msg_iovlen** задаются места приёма/передачи (см. **writev(2)**[1]).

```
struct iovec {
    void   *iov_base; // начальный адрес
    size_t  iov_len;  // количество передаваемых байт
};
```

Вызов **sendmsg()** помимо обычных передач позволяет отправлять вспомогательные данные (так называемую управляющую информацию).

Управляющую информацию можно посылать через поля **msg_control** и **msg_controllen**.

Максимальная длина **msg_controllen** управляющего буфера **msg_control**, которую поддерживает ядро, ограничена значением **/proc/sys/net/core/optmem_max** (socket(7)).

```
$ cat /proc/sys/net/core/optmem_max
81920
```

Поле **msg_flags** игнорируется.

Для **sendmsg()** адрес назначения указывается в **msg.msg_name**, а его размер в **msg.msg_namelen**.

У **sendmsg()** сообщение указывается в элементах массива **msg.msg_iov**.

Если сообщение слишком длинно для передачи за раз через используемый нижележащий протокол, то возвращается ошибка **EMSGSIZE** и сообщение не передаётся.

---

1) разнесенный ввод/вывод

Неудачная отправка с помощью **send()** никак не отмечается.

При обнаружении локальных ошибок возвращается значение -1.

Когда сообщение не помещается в буфер отправки сокета, выполнение блокируется в **send()**, если сокет не находится в неблокирующем режиме.

Если сокет находится в неблокирующем режиме, то возвращается ошибка **EAGAIN** или **EWOULDBLOCK**.

Для выяснения, возможна ли отправка данных, можно использовать вызов **select(2)**.

Флаги

Аргумент **flags** является битовой маской и может содержать следующие флаги:

MSG_DONTWAIT

Включить неблокирующий режим. Если операция могла бы привести к блокировке, возвращается **EAGAIN** или **EWOULDBLOCK**.

Такое поведение подобно заданию флага **O_NONBLOCK** в **fcntl(2)** операцией **F_SETFL**, но отличие состоит в том, что **MSG_DONTWAIT** указывается в вызове, а **O_NONBLOCK** задаётся в описании открытого файла (open(2)), что влияет на все потоки вызывающего процесса, а также на другие процессы, у которых есть файловые дескрипторы, ссылающиеся на это же описание открытого файла.

MSG_NOSIGNAL

Не генерировать сигнал **SIGPIPE**, если сторона потокоориентированного сокета закрыла соединение. Однако, по прежнему возвращается ошибка **EPIPE**. Это создаёт поведение как при использовании **sigaction(2)** для игнорирования **SIGPIPE**, но **MSG_NOSIGNAL** является свойством вызова, а установка **SIGPIPE** в атрибутах процесса влияет на все нити процесса.

MSG_EOR

Завершить запись (record) (если поддерживается, например в сокетах типа **SOCK_SEQPACKET**).

MSG_OOB

Послать внепоточные данные, если сокет это поддерживает (как, например, сокеты типа **SOCK_STREAM**); протокол более низкого уровня также должен поддерживать внепоточные данные.

MSG_CONFIRM (Linux)

Сообщить канальному уровню (link layer), что процесс пересылки произошел, т.е. получен успешный ответ с другой стороны.

Если канальный уровень не получит уведомление, то он будет регулярно перепроверять наличие ответной стороны (например посредством однонаправленной передачи ARP). Это работает только с сокетами **SOCK_DGRAM** и **SOCK_RAW** и в настоящее время реализовано только для IPv4 и IPv6. В arp(7) представлена более подробная информация[2].

---

2) arp — Address Resolution Protocol (протокол разрешения адресов), определённый в RFC 826. Протокол предназначен для преобразования аппаратных адресов второго уровня (Layer2) в адреса протокола IPv4 в соединённых напрямую сетях. Как правило, пользователю не приходится работать с этим модулем непосредственно, исключая случаи его настройки — модуль используется другими протоколами ядра.

MSG_DONTROUTE

Не использовать маршрутизацию для отправки пакета, а посылать его только на узлы локальной сети.

Обычно это используется в диагностических программах и программах маршрутизации. Этот флаг определён только для маршрутизируемых семейств протоколов — пакетные сокеты не используют маршрутизацию.

MSG_MORE

Вызывающий имеет дополнительные данные для отправки. Этот флаг используется с сокетами TCP для получения такого же эффекта как с параметром сокета **TCP_CORK** (см. tcp(7)), с той разницей, что этот флаг можно устанавливать при каждом вызове.

Начиная с Linux 2.6 этот флаг также поддерживается для сокетов UDP и информирует ядро, о том что нужно упаковать все отправляемые данные вызовов с этим флагом в одну дейтаграмму, которая передаётся только когда выполняется вызов без указания этого флага (udp(7)).

В POSIX.1-2001 описаны только флаги MSG_OOB и MSG_EOR.
В POSIX.1-2008 добавлено описание MSG_NOSIGNAL.
Флаг MSG_CONFIRM является нестандартным расширением Linux.

Возвращаемое значение

При успешном выполнении эти вызовы возвращают количество отправленных байт.

В случае ошибки возвращается -1, а **`errno`** устанавливается в соответствующее значение.

Ошибки

Представлено несколько стандартных ошибок, возвращаемых с уровня сокетов.

Могут также появляться и другие ошибки, возвращаемые из соответствующих модулей протоколов — их описание следует искать в соответствующих справочных страницах.

EACCES (для доменных сокетов UNIX, которые идентифицируются по имени пути) — нет прав на запись в файл сокета назначения или в одном из каталогов пути запрещён поиск.

EACCES — (для сокетов UDP) Попытка отправки по сетевому/широковещательному адресу, как будто это был однозначный (unicast) адрес.

EAGAIN или **EWOULDBLOCK** — Сокет помечен как неблокирующий, но запрошенная операция привела бы к блокировке. POSIX.1-2001 допускает в этих случаях возврат ошибки и не требует, чтобы эти константы имели одинаковое значение, поэтому переносимое приложение должно проверять обе возможности.

EAGAIN (доменные датаграммные сокеты Интернета) — Сокет, указанный **`sockfd`**, ранее не был привязан к адресу и при попытке привязать его к динамическому порту[3], было определено, что все номера в диапазоне динамических портов уже используются.

EALREADY — в данный момент выполняется другая операция Fast Open.

---

3) Динамический порт, или «эфемерный порт», — временный порт, открываемый соединением межсетевого протокола транспортного уровня (IP) из определённого диапазона программного стека TCP/IP.

EBADF — значение **sockfd** не является правильным открытым файл. дескриптором.

ECONNRESET — соединение сброшено другой стороной.

EDESTADDRREQ — сокет в режиме без установления соединения и адрес второй стороны не задан.

EFAULT — в аргументе указано неверное значение адреса пользовательского пространства.

EINTR — получен сигнал до начала передачи данных; смотрите signal(7).

EINVAL — передан неверный аргумент.

EISCONN — сокет в режиме с установлением соединения уже выполнил подключение, но указан получатель (теперь или возвращается эта ошибка, или игнорируется указание получателя).

EMSGSIZE — для типа сокета требуется, чтобы сообщение было отослано за время одной операции (атомарно), а размер сообщения не позволяет этого.

ENOBUFS — исходящая очередь сетевого интерфейса заполнена. Обычно это означает, что интерфейс прекратил отправку, но это может быть также вызвано временной перегрузкой сети. Обычно, в Linux этого не происходит. Пакеты просто отбрасываются, когда очередь устройства переполняется.

ENOMEM — больше нет доступной памяти.

ENOTCONN — сокет не подключён и назначение не задано.

ENOTSOCK — файловый дескриптор **sockfd** указывает не на каталог.

EOPNOTSUPP — один из битов в аргументе **flags** не может устанавливаться для этого типа сокета.

EPIPE — локальный сокет, ориентированный на соединение, был закрыт. В этом случае процесс также получит сигнал SIGPIPE, если не установлен флаг MSG_NOSIGNAL.

Замечания

В соответствие с POSIX.1-2001 поле **msg_controllen** структуры **msghdr** должно иметь тип **socklen_t**, но в настоящее время в glibc оно имеет тип **size_t**.

В **sendmmsg(2)** можно найти информацию о специальном системном вызове Linux, который можно использовать для передачи нескольких дейтаграмм за один вызов.

recv(), recvfrom(), recvmsg() — принимает сообщение из сокета

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int    sockfd,                 // send()
             void  *buf,
             size_t len,
             int    flags);

ssize_t recvfrom(int              sockfd,   // sendto
                 void            *buf,
                 size_t           len,
                 int              flags,
                 struct sockaddr *src_addr,
                 socklen_t       *addrlen);

ssize_t recvmsg(int            sockfd,      // sendmsg()
                struct msghdr *msg,
                int            flags);
```

Системные вызовы **recv()**, **recvfrom()** и **recvmsg()** используются для получения сообщений из сокета. Они могут использоваться для получения данных, независимо от того, является ли сокет ориентированным на соединения или нет.

Общие свойства

Вызов **recv()** отличается от **read(2)** только наличием аргумента **flags**. Если значение **flags** равно нулю, то вызов **recv()** эквивалентен **read(2)** с некоторыми отличиями.

```
recv(sockfd, buf, len, flags);
```

эквивалентен

```
recvfrom(sockfd, buf, len, flags, NULL, NULL);
```

При успешном выполнении все три вызова возвращают длину сообщения.

Если сообщение слишком длинное и не поместилось в предоставленный буфер, лишние байты могут быть отброшены, в зависимости от типа сокета, на котором принимаются сообщения.

Если на сокете не доступно ни одного сообщения и если сокет не помечен как неблокирующий (**fcntl(2)**), то эти вызовы блокируются в ожидании их прибытия.

Если сокет помечен как неблокирующий, возвращается значение -1, а внешняя переменная **errno** устанавливается в значение **EAGAIN** или **EWOULDBLOCK**.

Все эти вызовы обычно сразу возвращают все доступные данные, не ожидая, пока появятся данные полной запрошенной длины.

Для определения появления новых данных в сокете приложение может использовать **select(2)**, **poll(2)** или **epoll(7)**.

Флаги

Аргумент **flags** формируется с помощью объединения логической операцией ИЛИ одного или более следующих значений:

MSG_CMSG_CLOEXEC (только для recvmsg( ))

Установить флаг **close-on-exec** для файлового дескриптора, полученного через доменный файловый дескриптор UNIX, с помощью операции **SCM_RIGHTS** (AF_UNIX). Этот флаг полезен по тем же причинам что и флаг **O_CLOEXEC** у **open(2)**.

MSG_DONTWAIT

Включить неблокирующий режим. Если операция могла бы привести к блокировке, возвращается **EAGAIN** или **EWOULDBLOCK**. Такое поведение подобно заданию флага **O_NONBLOCK** (**fcntl(2)** операция **F_SETFL**), но отличие в том, что **MSG_DONTWAIT** указывается в вызове, а **O_NONBLOCK** задаётся в описании открытого файла (**open(2)**), что влияет на все нити вызывающего процесса, а также на другие процессы, у которых есть файловые дескрипторы, ссылающиеся на это описание открытого файла.

MSG_OOB

Этот флаг запрашивает приём внеполосных данных, которые в противном случае не были бы получены в обычном потоке данных. Некоторые протоколы помещают данные повышенной срочности в начало очереди с обычными данными, и поэтому этот флаг не может использоваться с такими протоколами.

MSG_PEEK

Этот флаг заставляет выбрать данные из начала очереди приёма, но не удалять их оттуда. Таким образом, последующий вызов вернёт те же самые данные.

MSG_TRUNC

Для «сырых» данных (**AF_PACKET**), дейтаграмм Интернета, **netlink** и дейтаграмм UNIX возвращает реальную длину пакета или дейтаграммы, даже если она была больше, чем предоставленный буфер. Описание использования с потоковым сокетами Интернета приведено в tcp(7).

MSG_WAITALL

Этим флагом включается блокирование операции до полной обработки запроса. Однако, этот вызов всё равно может вернуть меньше данных, чем было запрошено, если был пойман сигнал, произошла ошибка или разрыв соединения, или если начали поступать данные другого типа, не того, который был сначала. Этот флаг не влияет на датаграммные сокеты.

MSG_ERRQUEUE

Указание этого флага позволяет получить из очереди ошибок сокета накопившиеся ошибки.

Ошибка передаётся в вспомогательном сообщении тип которого зависит от протокола (для IPv4 это **IP_RECVERR**). Вызывающий должен предоставить буфер достаточного размера. Дополнительная информация приведена в **cmsg(3)** и **ip(7)**.

Содержимое исходного пакета, который привёл к ошибке, передаётся в виде обычных данных через **msg_iovec**.

Исходный адрес назначения датаграммы, которая привела к ошибке, передаётся через **msg_name**.

Ошибка передаётся в виде структуры **sock_extended_err**:

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err {
    uint32_t ee_errno;   // номер ошибки
    uint8_t  ee_origin;  // источник её происхождения
    uint8_t  ee_type;    // тип
    uint8_t  ee_code;    // код
    uint8_t  ee_pad;     // заполнение для выравнивания
    uint32_t ee_info;    // дополнительная информация
    uint32_t ee_data;    // прочие данные
    // далее могут содержаться ещё данные
};

struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);
```

В **ee_errno** содержится значение **errno** для ожидающей ошибки.

В **ee_origin** содержится источник происхождения ошибки.

Смысл остальных полей зависит от протокола.

Макрос **SOCK_EE_OFFENDER** возвращает указатель на адрес сетевого объекта, породившего ошибку.

Если этот адрес неизвестен, то поле **sa_family** структуры **sockaddr** будет содержать значение **AF_UNSPEC**, а прочие поля структуры **sockaddr** не определены.

Содержимое пакета, вызвавшего ошибку, передаётся в виде обычных данных.

Для локальных ошибок адрес не передаётся (это можно выяснить, проверив поле **cmsg_len** структуры **cmsghdr**).

Для получения ошибок при приёме следует в **msghdr** установить флаг **MSG_ERRQUEUE**.

После того, как ошибка передана программе, следующая ошибка в очереди ошибок становится ожидающей ошибкой и передается программе при следующей операции на сокете.

recvfrom()

Вызов **recvfrom()** помещает принятое сообщение в буфер **buf**.

Вызывающий должен указать размер буфера в **len**.

Если значение **src_addr** не равно **NULL**, и в нижележащем протоколе используется адрес источника сообщения, то адрес источника помещается в буфер, указанный в **src_addr**. В этом случае **addrlen** является аргументом-результатом. Перед вызовом ему должно быть присвоено значение длины буфера, связанного с **src_addr**.

При возврате **addrlen** обновляется и содержит действительный размер адреса источника.

Если предоставленный буфер слишком мал, возвращаемый адрес обрезается и в этом случае **addrlen** будет содержать значение большее, чем указывалось в вызове.

Если вызывающему адрес источника не нужен, то значение **src_addr** и **addrlen** должны быть равно **NULL**.

recv()

Вызов **recv()**, обычно, используется только на соединённом сокете (**connect(2)**).
Он идентичен вызову:

```
recvfrom(fd, buf, len, flags, NULL, 0);
```

recvmsg()

Для минимизации количества передаваемых аргументов в вызов **recvmsg()** используется структура **msghdr**. Она определена в <sys/socket.h> следующим образом:

```
struct iovec {                    // массив элементов приёма/передачи
    void  *iov_base;              // начальный адрес
    size_t iov_len;               // количество передаваемых байт
};

struct msghdr {
    void         *msg_name;       // необязательный адрес
    socklen_t     msg_namelen;    // размер буфера для возврата адреса
    struct iovec *msg_iov;        // массив приёма/передачи
    size_t        msg_iovlen;     // количество элементов в msg_iov
    void         *msg_control;    // вспомогательные данные
    size_t        msg_controllen; // размер буфера вспомогательных данных
    int           msg_flags;      // флаги принятого сообщения
};
```

Поле **msg_name** указывает на выделенный вызывающим буфер, который используется для возврата адреса источника, если сокет не соединён.

Вызывающий должен указать в **msg_namelen** размер этого буфера перед вызовом; при успешном выполнении вызова в **msg_namelen** будет содержаться длина возвращаемого адреса. Если приложению не надо знать адрес источника, то в **msg_name** указывается **NULL**.

В полях **msg_iov** и **msg_iovlen** описываются место приёма/передачи (см. **readv(2)**)[4].

Вспомогательные данные

Поле **msg_control** длиной **msg_controllen** указывает на буфер для других сообщений, связанных с управлением протоколом или на буфер для разнообразных вспомогательных данных. При вызове **recvmsg()** в поле **msg_controllen** должен указываться размер доступного буфера, чей адрес передан в **msg_control**.

При успешном выполнении вызова в этом параметре будет находиться длина последовательности контрольных сообщений в виде:

```
struct cmsghdr {
    size_t cmsg_len;    // счетчик байтов данных с заголовком (socklen_t в POSIX)
    int    cmsg_level;  // начальный протокол
    int    cmsg_type;   // тип, зависящий от протокола
    unsigned char cmsg_data[0];
};
```

К вспомогательным данным нужно обращаться только с помощью макросов, определённых в **cmsg(3)**. Например, этот механизм вспомогательных данных используется в Linux для передачи расширенных ошибок, флагов IP и файловых дескрипторов через доменные сокеты Unix.

---

4) Системный вызов readv() работает как и read(), но считывает несколько буферов из файла, связанного с файловым дескриптором fd, в буферы, описываемые iov («разнесённый ввод»). Буферы заполняются в порядке массива.

При возврате из **`recvmsg()`** устанавливается значение поля **`msg_flags`** в **`msghdr`**.

Оно может содержать несколько флагов:

MSG_EOR — означает конец записи: возвращённые данные заканчивают запись (обычно используется вместе с сокетами типа **SOCK_SEQPACKET**).

MSG_TRUNC — означает, что хвостовая часть датаграммы была отброшена, потому что датаграмма была больше, чем предоставленный буфер.

MSG_CTRUNC — означает, что часть управляющих данных была отброшена из-за недостатка места в буфере вспомогательных данных.

MSG_OOB — возвращается для индикации, что получены курируемые или внеполосные данные.

MSG_ERRQUEUE — означает, что были получены не данные, а расширенное сообщение об ошибке из очереди ошибок сокета.

В POSIX.1 описаны только флаги **MSG_OOB**, **MSG_PEEK** и **MSG_WAITALL**.

Возвращаемое значение

Эти вызовы возвращают количество принятых байт или -1, если произошла ошибка. В случае ошибки в **`errno`** записывается код ошибки.

Когда ответная сторона потока выполняет корректное отключение (shutdown), то возвращается 0 (обычный возврат «конец файла»).

В датаграмных сокетах некоторых доменов (например, доменах UNIX и Internet) разрешены датаграммы нулевой длины. При получении такой датаграммы возвращается значение 0. Также значение 0 может возвращаться, если запрошенное количество принимаемых байт из потокового сокета равно 0.

Ошибки

Здесь представлено несколько стандартных ошибок, возвращаемых с уровня сокетов. Могут также появиться другие ошибки, возвращаемые из соответствующих протокольных модулей, их описание находится в соответствующих справочных страницах.

EAGAIN или **EWOULDBLOCK** — Сокет помечен как неблокируемый, а операция приёма привела бы к блокировке, или установлено время ожидания данных и это время истекло до получения данных. Согласно POSIX.1 в этом случае может возвращаться любая ошибка и не требуется, чтобы эти константы имели одинаковое значение, поэтому переносимое приложение должно проверить оба случая.

EBADF — аргумент **`sockfd`** содержит неверный файловый дескриптор.

ECONNREFUSED — удалённый узел отказался устанавливать сетевое соединение (обычно потому, что там не работает запрошенная служба).

EFAULT — указатель на приёмный буфер указывает вне адресного пространства процесса.

EINTR — приём данных был прерван сигналом, а данные ещё не были доступны.

EINVAL — передан неверный аргумент.

ENOMEM — не удалось выделить память для **`recvmsg()`**.

ENOTCONN — сокет, связанный с протоколом, ориентированным на соединение, не был соединён (см. connect(2) и accept(2)).

ENOTSOCK — файловый дескриптор **`sockfd`** указывает не на каталог.