Системное программное обеспечение вычислительных машин (СПОВМ)

Лекция 20 — Сокеты

Преподаватель: Поденок Леонид Петрович, 505a-5

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

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

Кафедра ЭВМ, 2022

2022.05.12

Оглавление

Сокеты

«Сокет» — это обобщенный канал межпроцессного взаимодействия.

Как и канал, сокет представлен как файловый дескриптор.

Однако, в отличие от каналов, сокеты поддерживают связь между несвязанными процессами и даже между процессами, запущенными на разных машинах, которые обмениваются данными по сети.

Сокеты являются основным средством связи с другими машинами.

telnet, ssh, rlogin, ftp, http, talk и другие знакомые сетевые программы используют сокеты.

Не все операционные системы поддерживают сокеты.

В библиотеке GNU C файл заголовка «sys/socket.h» существует независимо от операционной системы, и функции сокетов существуют всегда, но если система на самом деле не поддерживает сокеты, эти функции всегда завершаются неудачно.

Основные концепции

Когда создается сокет, необходимо указать *тип связи*, который предполагается использовать, и *тип протокола*, который должен ее реализовать.

Тип связи

«Тип (стиль) связи» сокета определяет семантику отправки и получения данных на уровне пользователя через сокет.

Выбор типа связи определяет ответы на следующие вопросы:

- **Каковы единицы передачи данных?** Некоторые стили связи рассматривают данные как последовательность байтов без какой-либо структуры, другие группируют байты в записи (известные в этом контексте как «пакеты»).
- **Могут ли данные быть потеряны во время нормальной работы?** Некоторые типы связи гарантируют, что все отправленные данные прибывают в том порядке, в котором они были отправлены (запрет системы или сбоев сети), другие стили иногда теряют данные как нормальная часть работы, а иногда могут доставлять пакеты более одного раза или в неправильном порядке.

Разработка программы с использованием ненадежных стилей связи обычно включает меры предосторожности для обнаружения потерянных или неправильно упорядоченных пакетов и повторной передачи данных по мере необходимости.

Некоторые стили общения похожи на телефонный звонок — устанавливается «соединение» с одним удаленным сокетом, а затем происходит свободный обмен данными. Другие стили подобны обмену почтовыми письмами — указывается адрес назначения для каждого отправляемого сообщения.

У сокета есть имя («адрес»). Имя сокета имеет смысл только в контексте определенного пространства имен. Поэтому для наименования сокета необходимо выбрать «пространство имен». Пространства имен иногда называют «доменами», но обычно следует избегать этого слова, поскольку его можно спутать с другим использованием того же термина.

У каждого пространства имен есть символическое имя, которое начинается с **PF_**.

Соответствующее символическое имя, начинающееся с **AF_**, обозначает формат адреса для этого пространства имен.

Тип протокола

Для осуществления связи необходимо выбрать «протокол».

Протокол определяет, какой низкоуровневый механизм используется для передачи и приема данных. Каждый протокол действителен для определенного пространства имен и стиля связи. Из-за этого пространство имен иногда называют «семейством протоколов» — именно поэтому имена пространств имен начинаются с «**PF_**».

Правила протокола применяются к передаче данных между двумя программами, которые, возможно, выполняются на разных компьютерах. Большинство этих правил обрабатываются операционной системой и всех деталей знать не требуется.

Тем не менее, нужно знать о протоколах следующее:

1) Для связи между двумя сокетами они должны указать один и тот же протокол.

2) Каждый протокол имеет смысл с определенными комбинациями стиля/пространства имен и не может использоваться с несоответствующими комбинациями. Например, протокол TCP соответствует только стилю связи с потоком байтов и пространству имен Internet.

3) Для каждой комбинации стиля и пространства имен существует «протокол по умолчанию», который можно запросить, указав 0 в качестве номера протокола. И это то, что обычно следует делать — использовать в качестве номера протокола значение по умолчанию.

В различных местах требуются переменные/параметры для обозначения размеров.

В первых реализациях тип этих переменных был просто **`int`**. На большинстве машин того времени **`int`** имел ширину 32 бита, что создавало _*de facto*_ стандарт, требующий 32-битных переменных. Это важно, поскольку ссылки на переменные этого типа передаются ядру.

Затем пришли специалисты по POSIX и объединили интерфейс со словами «все значения размера относятся к типу **`size_t`**». На 64-битных машинах **`size_t`** имеет ширину 64 бита, поэтому использовать указатели на переменные больше не было возможности.

Спецификация Unix98 предлагает решение, вводя тип **`socklen_t`**.

Этот тип используется во всех случаях, когда POSIX изменил использование **`size_t`**. Единственное требование к этому типу — это беззнаковый тип длиной не менее 32 бит. Следовательно, реализации, которые требуют передачи ссылок на 32-битные переменные, могут быть такими же счастливыми, как и реализации, использующие 64-битные значения.

Типы сокетов

Библиотека GNU C включает поддержку нескольких различных типов сокетов, каждый из которых имеет различные характеристики. Символьные константы, определяющие поддерживаемые типы сокетов, определены в **sys/socket.h**.

SOCK_STREAM

Тип **SOCK_STREAM** похож на канал (pipes и FIFO). Он 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 индивидуальной 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.

Единственная гарантия, которую дает система в отношении запросов на передачу данных этого типа, — это то, что она будет делать все возможное, чтобы доставить каждый отправленный пакет. Доставка может быть успешной для шестого пакета после неудачи с четвертым и пятым, седьмой пакет может прибыть раньше шестого и может прийти второй раз после шестого.

Типичное использование **SOCK_DGRAM** — это ситуации, когда допустимо просто повторно отправить пакет, если в течение разумного периода времени не наблюдается ответа.

SOCK_RAW

Этот тип обеспечивает доступ к сетевым протоколам и интерфейсам низкого уровня. Обычные пользовательские программы обычно не нуждаются в использовании этого стиля.

SOCK_SEQPACKET

Обеспечивает работу последовательного двустороннего канала для передачи дейтаграмм на основе соединений. Дейтаграммы имеют постоянный размер. От получателя требуется за один раз прочитать целый пакет.

SOCK_RDM

Обеспечивает надежную доставку дейтаграмм без гарантии, что они будут расположены по порядку.

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

Адреса сокетов

Имя сокета обычно называется «адресом».

Функции и символы для работы с адресами сокетов были названы непоследовательно, иногда с использованием термина «имя», а иногда с использованием термин «адрес». Когда речь идет о сокетах, можно рассматривать эти термины 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()`**.

Нужно беспокоиться об адресе сокета только в том случае, если другие процессы должны его найти и начать с ним общаться. Можно указывать адреса для других сокетов, но обычно это бессмысленно — если адрес не был указан, система автоматически присваивает сокету адрес, как только из сокета отправляются данные или он используется для установления соединения.

Иногда клиенту необходимо указать адрес, потому что сервер различает его по адресу, например, протоколы **`rsh`** и **`rlogin`** просматривают адрес клиентского сокета и если он меньше **`IPPORT_RESERVED`**, обходят проверку парольной фразы.

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 *`**.

Форматы имен/адресов

Некоторые функции, в частности **bind()** и **getsockname()**, для представления указателя на адрес сокета используется общий тип данных **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

Обозначает формат адреса, который соответствует локальному пространству имен. (**PF_LOCAL** — это имя этого пространства имен.)

AF_UNIX

Это синоним **AF_LOCAL**.
AF_UNIX было традиционным названием, происходящим от BSD, поэтому большинство систем POSIX поддерживают его.

AF_FILE

Это еще один синоним **AF_LOCAL** для совместимости.

AF_INET

Обозначает формат адреса, который соответствует пространству имен Internet. Имя этого пространства имен — **PF_INET**.

AF_INET6

Похоже на **AF_INET**, но относится к протоколу IPv6. Имя соответствующего пространства имен — **PF_INET6**.

AF_UNSPEC

Не означает никакого конкретного формата адреса. Он используется только в редких случаях, например, для очистки адреса назначения по умолчанию «подключенного» сокета дейтаграммы. Соответствующий символ обозначения пространства имен **PF_UNSPEC** существует исключительно для полноты и нет причин использовать его в программе.

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

Работа с сокетами

Сокеты — основной способ организации обмена информацией между компьютерами, объединенными в сеть.

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()     --
write()    --
close()    -- закрыть сокет
```

Поскольку сетевое взаимодействие является сложным и требует большое количество разнообразных протоколов связи, существует еще около 60 системных вызовов, имеющих отношение к сокетам. Эти функции используются пользовательским процессом для отправки или получения пакетов и 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.

close(2) используется для закрытия сокета

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)**.

Принцип действия

Как работают сокеты можно понять, отталкиваясь от аналогий с именованными каналами. Процесс может открыть именованный канал на чтение примерно таким образом:

```
fd = open("the_fifo", 0_RD0NLY);
```

Этот системный вызов делает следующее:

1. Создает объект для операций ввода-вывода и назначает ему файловый дескриптор.
2. Связывает файловый дескриптор с внешним именем «the_fifo».
3. Ожидает, пока кто-нибудь, не откроет этот канал на запись.
4. Возвращает файловый дескриптор, который затем может быть использован в системном вызове **read()**.

Аналогичным образом работают и сокеты, только каждый шаг выделен в отдельный системный вызов:

socket — создает объект (сокет) и назначает ему файловый дескриптор.
bind — устанавливает адрес (имя) сокета, чтобы другой процесс мог на него сослаться.
listen — помечает сокет для приема запросов на соединение от других сокетов.
accept — блокируется в ожидании запроса на соединение.
connect — устанавливает соединение с сокетом, который был заблокирован на вызове **accept()**.

Именованные каналы не предусматривают различий между сервером и клиентом, в том смысле, что оба они обращаются к одному и тому же системному вызову **open()**, тем не менее при открытии канала они используют различные флаги — **0_RDONLY** и **0_WRONLY**.

Последовательность создания и инициализации сокетов на стороне сервера отличается от последовательности действий на стороне клиента.

	Сервер		**Клиент**
1)	socket	1)	socket
2)	bind "superserver"	2)	connect "superserver"
3)	listen		
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) Файловый дескриптор сокета используется для выполнения операций ввода-вывода.

```c
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#define SOCKETNAME "my_socket"

int main(void) {

    struct sockaddr_un sa; // AF_UNIX address

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;

    if (fork() == 0) { // потомок-клиент
        int  fd_client;
        char buf[100];

        fd_client = socket(AF_UNIX, SOCK_STREAM, 0); // 0/-1
        while (connect(fd_client, (struct sockaddr *)&sa, sizeof(sa)) == -1) {
            if (errno == ENOENT) {
                sleep(1);
                continue;
            } else {
                perror("connect");
                exit(errno);
        }   }
```

```
        write(fd_client, "Привет!", strlen("Привет!"));
        read(fd_client, buf, sizeof(buf));
        printf("Клиент получил сообщение \"%s\"\n", buf);
        close(fd_client);
        exit(EXIT_SUCCESS);

    } else { // родитель-сервер
        int fd_server, fd_client;
        char buf[100];

        fd_server = socket(AF_UNIX, SOCK_STREAM, 0); // 0/-1
        bind(fd_server, (struct sockaddr *)&sa, sizeof(sa));
        listen(fd_server, SOMAXCONN);
        fd_client = accept(fd_server, NULL, 0);
        read(fd_client, buf, sizeof(buf));
        printf("Сервер получил сообщение \"%s\"\n", buf);
        write(fd_client, "OK! Good Bye...", strlen("OK! Good Bye..."));
        close(fd_server);
        close(fd_client);
        exit(EXIT_SUCCESS);
    }
}
```

Соединение может существовать сколь угодно долго, пока какая-нибудь из сторон его не разорвет. По своей сути соединение напоминает двунаправленный неименованный канал.

Файл сокета действительно является файлом

```
$ ls -l
drwxr-xr-x. 2 leo leo  4096 мая  11 10:04 Debug
-rw-rw-r--. 1 leo leo  2653 мая  11 09:29 makefile
srwxrwxr-x. 1 leo leo     0 мая  11 09:58 my_socket
-rw-rw-r--. 1 leo leo  1570 мая  11 10:04 usocket.c
```

В результате работы программы выходит следующее:

```
$ ./Debug/usocket
Сервер получил сообщение "Привет!"
Клиент получил сообщение "OK! Good Bye..."
```

Работа с сокетами очень проста, за исключением следующих моментов.

- обслуживание нескольких клиентов одним сервером.
- наличие нескольких доменов (областей) адресов, включая **AF_INET**, который представляет наибольший интерес, при организации взаимодействий через сеть.
- ориентированность на обмен дейтаграммами, а не на потоки данных.
- возможность обмена дейтаграммами без создания логического соединения.
- большое количество вариантов тонкой настройки поведения сокетов, особенно для доменов **AF_INET** и **AF_INET6**.
- передача двоичных данных между компьютерами, аппаратные платформы которых используют различные способы представления чисел.
- обращение к узлам сети по именам (например **lsi.bas-net.by/pub**).

Порядок работы

1) Создание объекта для организации сетевого взаимодействия

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

int socket(int socket_family, // семейство протоколов[1] (AF_UNIX, AF_INET, ...)
           int socket_type,   // SOCK_STREAM, SOCK_DGRAM, ...
           int protocol);     // детализация type; обычно ноль

Возвращает файловый дескриптор или -1/errno в случае ошибки
```

Файловый дескриптор может использоваться:

- **сервером** — для приема входящих соединений системным вызовом accept(). Фактические операции ввода-ввода выполняются над другим файловым дескриптором, который возвращается системным вызовом **`accept()`**.

- **клиентом** — для выполнения операций ввода-вывода после того, как будет установлено соединение вызовом **`connect()`**.

---

1) или адресный домен, что эквивалентно

2) После создания сокета сервер должен присвоить ему адрес, используя системный вызов **bind()**.

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

int bind(int                    sockfd,   // файловый дескриптор сокета
         const struct sockaddr *addr,     // адрес сокета
         socklen_t              addrlen); // длина адреса

Возвращает 0 в случае успеха или -1/errno в случае ошибки
```

В большинстве систем для адресов из домена **AF_UNIX** системный вызов **bind()** создает новый файл сокета — он не может повторно использовать существующий файл. Поэтому, чтобы не напороться на то, что файл уже существует, следует перед вызовом удалять возможно существующий файл сокета с помощью системного вызова **unlink()**.

3) Чтобы подготовить сокет к приему входящих соединений, сервер должен вызвать **listen()**.

Системный вызов **listen()** подготавливает сокет к приему запросов на соединение и устанавливает ограничение на размер очереди запросов.

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

int listen(int sockfd,   // файловый дескриптор сокета
           int backlog); // максимальный размер очереди ожидающих соединений

Возвращает 0 в случае успеха или -1/errno в случае ошибки
```

Сервер может обслуживать большое число клиентов. Но какой бы мощный сервер ни был, он не может принимать соединения быстрее, чем это позволит системный вызов **accept()**, поэтому запросы на соединение помещаются в очередь, где дожидаются обработки. Размер этой очереди ограничивается вторым аргументом **backlog**.

Как правило, когда очередь будет заполнена до отказа, системный вызов **connect()** на стороне клиента возвратит признак ошибки с кодом **ECONNREFUSED**.

Максимально возможное число соединений в системе определяется константой **SOMAXCONN**.

4) Прием соединений на стороне сервера выполняется системным вызовом **accept()**.

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

int accept(int                sockfd,   // файловый дескриптор слушающего сокета
           struct sockaddr *addr,     // адресс сокета или NULL
           socklen_t       *addrlen); // длина адреса

Возвращает файловый дескриптор или -1/errno в случае ошибки
```

Обычно системный вызов **accept()** блокируется до поступления запроса на соединение от другого процесса, обратившегося к вызову **connect()**. После поступления запроса создает новый сокет для принятого соединения и возвращает его файловый дескриптор.

Этот дескриптор может использоваться как обычными системными вызовами **read()** и **write()**, так и другими вызовами, специфичными для сокетов, которые дают больший контроль над операциями ввода-вывода:

send(), **sendto()**, **sendmsg()** -- отправляют сообщения в сокет
recv(), **recvfrom()**, **recvmsg()** -- принимают сообщение из сокета

Если для файлового дескриптора установлен флаг **O_NONBLOCK** (с помощью вызова **fcntl()**) и очередь запросов пуста, вместо ожидания поступления запроса вызов **accept()** будет возвращать признак ошибки с кодом **EAGAIN** или **EWOULDBLOCK**.

Файловый дескриптор нового сокета может использоваться в системных вызовах **select()** или **poll()**.

Вызов **select()** позволяет 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 **poll()** выполняет сходную с **select()** задачу — он ждёт пока какой-нибудь дескриптор из набора зарегистрированных файловых дескрипторов не будет готов выполнить операцию ввода-вывода.

Обычно сервер в набор **fd_set** системного вызова **select()** помещает дескриптор своего сокета, который ожидает приема запросов на соединение, а также файловые дескрипторы всех сокетов, которые возвращает системный вызов **accept()**.

Если **select()** или **poll()** обнаружат готовность файлового дескриптора серверного сокета, это означает, что сервер должен вызвать **accept()**, который не будет заблокирован.

Готовность других сокетов означает, что от клиента (или от клиентов) поступили новые данные, которые следует прочитать.

Если аргумент **struct sockaddr *addr** не **NULL**, по указанному адресу записываются сведения о сокете, соединение с которым было принято.

При этом в аргументе **addrlen** необходимо передать указатель на переменную, которая хранит размер области памяти, отведенной под сведения о сокете.

По возвращении из системного вызова эта переменная будет содержать фактический объем данных, записанных в **addr**.

2) Клиент после создания сокета вызывает **connect()**, которому передает адрес сервера.

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

int connect(int                    sockfd,   // сокет инициатора соединения
            const struct sockaddr *addr,     // адрес соединения
            socklen_t              addrlen); // размер addr
```

Вызов **connect()** так же, как и системный вызов **accept()**, блокируется до того момента, как запрос на соединение будет принят сервером. Однако, в отличие от **accept()**, он не возвращает новый файловый дескриптор — клиент выполняет операции ввода-вывода, используя прежний дескриптор, который возвратил вызов **socket()**.

Если установлен флаг **O_NONBLOCK**, вызов **connect()** не будет ожидать установления соединения, а сразу же вернет признак ошибки с кодом **EINPROGRESS**. Запрос на соединение при этом не теряется, а помещается в очередь.

Все последующие обращения к **connect()** в этот период будут заканчиваться ошибкой с кодом **EALREADY** — предыдущая попытка установить соединение ещё не завершилась.

После того как соединение будет установлено, файловый дескриптор можно использовать различным образом. Например, можно использовать вызовы **select()** или **poll()**, чтобы дождаться готовности дескриптора на запись (не на чтение). Типичная последовательность действий в этом случае будет следующей:

- вызвать **connect()** в неблокирующем режиме (**fcntl(2)**);
- выполнить дополнительные действия по инициализации;
- вызвать **select()** или **poll()**, чтобы дождаться установления соединения.

socket() — создаёт конечную точку соединения

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

int socket(int socket_family, // семейство протоколов, которое будет использоваться
           int socket_type,   // семантика соединения (stream, datagram, raw, ...)
           int protocol);     // протокол, используемый с сокетом
```

Системный вызов **socket()** создаёт конечную точку соединения и возвращает файловый дескриптор, указывающий на эту точку. Возвращаемый при успешном выполнении файловый дескриптор будет иметь самый маленький номер, который не используется процессом.

socket_family

Параметр **socket_family** задает коммуникационный домен — выбирает семейство протоколов, которое будет использоваться для соединения. Семейства описаны в **<sys/socket.h>**. В настоящее время ядром Linux распознаются следующие форматы:

AF_UNIX	Локальное соединение
AF_LOCAL	Синоним AF_UNIX
AF_INET	Протоколы Интернет IPv4
AF_AX25	Протокол любительского радио AX.25
AF_IPX	Протоколы Novell IPX
AF_APPLETALK	AppleTalk
AF_X25	Протокол ITU-T X.25/ISO-8208

AF_INET6	Протоколы Интернет IPv6
AF_DECnet	Протокольные сокеты DECnet
AF_KEY	Протокол управления ключами IPsec
AF_NETLINK	Устройство для взаимодействия с ядром
AF_PACKET	Низкоуровневый пакетный интерфейс
AF_RDS	Протокол надёжных датаграмных сокетов (RDS)
AF_PPPOX	Транспортный слой PPP общего назначения для настройки туннелей L2 (L2TP и PPPoE)
AF_LLC	Протокол управления логической связью (IEEE 802.2 LLC)
AF_IB	Собственная адресация InfiniBand
AF_MPLS	Многопротокольная коммутация по меткам
AF_CAN	Протокол шины сети транспортных контроллеров
AF_TIPC	Протокол «кластерных доменных сокетов» TIPC
AF_BLUETOOTH	Протокол сокетов Bluetooth низкого уровня
AF_ALG	Интерфейс к ядерному крипто-API
AF_VSOCK	Протокол VSOCK («VMWare Vsockets») для связей гипервизор-гость
AF_KCM	Интерфейс KCM (мультиплексор соединений ядра)
AF_XDP	Интерфейс XDP (express data path)

Подробнее об адресных семействах, приведённых выше, а также информацию о других адресных семействах можно найти в **address_families(7)**.

socket_type

Сокет имеет тип **socket_type**, задающий семантику соединения. В настоящее время определены следующие типы:

SOCK_STREAM — Обеспечивает создание двусторонних, надёжных потоков байтов на основе установления соединения. Может также поддерживаться механизм внепоточных (urgent) данных.

SOCK_DGRAM — Поддерживает дейтаграммы (ненадежные сообщения с ограниченной длиной без установки соединения).

SOCK_SEQPACKET — Обеспечивает работу последовательного двустороннего канала для передачи дейтаграмм на основе соединений; дейтаграммы имеют постоянный размер; от получателя требуется за один раз прочитать целый пакет.

SOCK_RAW — Обеспечивает прямой доступ к сетевому протоколу.

SOCK_RDM — Обеспечивает надежную доставку дейтаграмм без гарантии, что они будут расположены по порядку.

SOCK_PACKET — Устарел и не должен использоваться в новых программах (**packet(7)**.

Некоторые типы сокетов могут быть не реализованы во всех семействах протоколов.

В Linux, начиная с 2.6.27, **socket_type** кроме определения типа сокета может содержать побитно сложенные любые следующие значения для изменения поведения сокета:

SOCK_NONBLOCK — Устанавливает флаг состояния файла **O_NONBLOCK** для нового открытого файлового описания (**open(2)**), на которое ссылается новый файловый дескриптор.

Использование данного флага делает ненужными дополнительные вызовы **fcntl(2)** для достижения того же результата.

SOCK_CLOEXEC — Устанавливает флаг close-on-exec (**FD_CLOEXEC**) для нового открытого файлового дескриптора (**open(2)**).

protocol

В **protocol** задаётся определённый протокол, используемый с сокетом.

Обычно, для поддержи определённого типа сокета с заданным семейством протоколов существует только единственный протокол, в этом случае в **protocol** можно указать 0. Однако, может существовать несколько протоколов, тогда нужно конкретно указать один из них.

Номер используемого протокола зависит от "семейства протоколов[2]", по которому устанавливается соединение (**protocols(5)**).

Как соотносить имена протоколов с их номерами, описано в **getprotoent(3)**.

Сокеты типа SOCK_STREAM

Сокеты типа **SOCK_STREAM** являются соединениями полнодуплексных байтовых потоков. Они не сохраняют границы записей. Потоковый сокет должен быть в состоянии соединения перед тем, как из него можно будет отсылать данные или принимать их. Соединение с другим сокетом создается с помощью системного вызова **connect(2)**. Когда сеанс закончен, выполняется команда **close(2)**.

После соединения данные можно передавать с помощью системных вызовов **read(2)** и **write(2)** или одного из вариантов системных вызовов **send(2)** и **recv(2)**.

Внепоточные данные могут передаваться, как описано в **send(2)**, и приниматься, как описано в **recv(2)**.

---

2) домена

Протоколы связи, которые реализуют **SOCK_STREAM**, следят, чтобы данные не были потеряны или дублированы. Если часть данных, для которых имеется место в буфере протокола, не может быть передана за определённое время, соединение считается разорванным.

Когда в сокете включен флаг **SO_KEEPALIVE**, протокол каким-либо способом проверяет, не отключена ли ещё другая сторона.

Если процесс посылает или принимает данные, пользуясь «разорванным» потоком, ему выдаётся сигнал **SIGPIPE**. Это приводит к тому, что процессы, не обрабатывающие этот сигнал, завершаются.

Сокеты SOCK_DGRAM и SOCK_RAW

Сокеты **SOCK_DGRAM** и **SOCK_RAW** позволяют посылать дейтаграммы принимающей стороне, заданной при вызове **sendto(2)**. Дейтаграммы обычно принимаются с помощью вызова **recvfrom(2)**, который возвращает следующую дейтаграмму с соответствующим обратным адресом.

Сокеты SOCK_SEQPACKET

Сокеты **SOCK_SEQPACKET** используют те же самые системные вызовы, что и сокеты **SOCK_STREAM**. Единственное отличие в том, что вызовы **read(2)** возвращают только запрошенное количество данных, а остальные данные пришедшего пакета будут отброшены. Границы сообщений во входящих дейтаграммах сохраняются.

Сокеты SOCK_PACKET

Тип **SOCK_PACKET** считается устаревшим типом сокета — он позволяет получать необработанные пакеты прямо от драйвера устройства. Вместо него следует использовать **packet(7)**.

Для задания группы процессов, которая будет получать сигнал **SIGURG**, когда прибывают внепоточные данные, или сигнал **SIGPIPE**, когда соединение типа **SOCK_STREAM** неожиданно обрывается, может использоваться системный вызов **fcntl(2)** с аргументом **F_SETOWN**.

Этот вызов также можно использовать, чтобы задать процесс или группу процессов, которые получают асинхронные уведомления о событиях ввода-вывода с помощью **SIGIO**.

Использование **F_SETOWN** эквивалентно использованию вызова **ioctl(2)** с аргументом **FIOSETOWN** или **SIOCSPGRP**.

Когда сеть сообщает модулю протокола об ошибке (например, в случае IP, используя ICMP-сообщение), то для сокета устанавливается флаг ожидающей ошибки. Следующая операция этого сокета вернёт код ожидающей ошибки. Некоторые протоколы позволяют организовывать очередь ошибок в сокете для получения подробной информацию об ошибке (см. IP_RECVERR в ip(7)).

Операции сокетов контролируются их параметрами **options**. Эти параметры описаны в **<sys/socket.h>**.

Чтобы установить и получить необходимые параметры, используются вызовы **setsockopt(2)** и **getsockopt(2)**.

Возвращаемое значение

В случае успешного выполнения возвращается дескриптор, ссылающийся на сокет. В случае ошибки возвращается -1, а значение errno устанавливается соответствующим образом.

Ошибки

EACCES — Нет прав на создание сокета указанного типа и/или протокола.

EAFNOSUPPORT — Реализация не поддерживает указанное семейства адресов.

EINVAL — Неизвестный протокол или недоступное семейство протоколов.

EINVAL — Неверные флаги в **socket_type**.

EMFILE Было достигнуто ограничение по количеству открытых файловых дескрипторов на процесс.

ENFILE — Достигнуто максимальное количество открытых файлов в системе.

ENOBUFS или **ENOMEM** — Недостаточно памяти для создания сокета. Сокет не может быть создан, пока не будет освобождено достаточное количество ресурсов.

EPROTONOSUPPORT — Тип протокола или указанный протокол не поддерживаются в этом домене.

Модулями протоколов более низкого уровня могут быть созданы другие ошибки.

Замечания

В POSIX.1 не требуется включение **<sys/types.h>**, и этот заголовочный файл не требуется в Linux. Однако, для некоторых старых реализаций (BSD) требует данный файл, и в переносимых приложениях для предосторожности лучше его указать.

Для семейств протоколов в 4.x BSD используются константы **PF_UNIX**, **PF_INET**, **PF_INET** и т. д., тогда как **AF_UNIX**, **AF_INET** и т. п. используется для указания семейства адресов. Однако, в справочной странице BSD сказано: «Обычно, семейство протоколов совпадает с семейством адресов» и во всех последующих стандартах используется **AF_***.

Пример

Пример использования **socket()** показан в **getaddrinfo(3)**.

bind() — связывает сокет с локальным адресом

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

int bind(int                    sockfd,   // файловый дескриптор сокета
         const struct sockaddr *addr,     // адрес сокета
         socklen_t              addrlen); // длина адреса
```

После создания с помощью **socket(2)**, сокет появляется в адресном пространстве (семействе адресов), но без назначенного адреса.

Функция **bind()** назначает сокету, указанному дескриптором файла **sockfd**, адрес, заданный в **addr**.

В аргументе **addrlen** задаётся размер структуры адреса (в байтах), на которую указывает **addr**.

В силу традиции, эта операция называется «*присваивание сокету имени*».

Обычно, сокету типа **SOCK_STREAM** до того, как он сможет принимать соединения (**accept(2)**), нужно назначить локальный адрес с помощью **bind()**.

Правила, используемые при привязке имён, отличаются в разных семействах адресов.

Описание **AF_INET** находится в **ip(7)**, **AF_INET6** в **ipv6(7)**, **AF_UNIX** в **unix(7)**, **AF_APPLETALK** в **ddp(7)**, **AF_PACKET** в **packet(7)**, **AF_X25** в **x25(7)**, а **AF_NETLINK** в **netlink(7)**.

Реальная структура, передаваемая через **addr**, зависит от семейства адресов.

Структура **sockaddr** определяется так:

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

Единственным смыслом этой структуры является преобразование указателя структуры, передаваемого в **addr**, чтобы избежать предупреждений компилятора.

Возвращаемое значение

При успешном выполнении возвращается 0.

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

Ошибки

EACCES — Адрес защищён, или пользователь не является суперпользователем.

EADDRINUSE — Указанный адрес уже используется.

EADDRINUSE — (доменные сокеты Интернета) В структуре адреса сокета указан номер порта равный нулю, но при попытке привязаться к короткоживущему порту, было определено, что все номера в диапазоне короткоживущих портов уже используются.

EBADF — Значение **sockfd** не является правильным файловым дескриптором.

EINVAL — Сокет уже привязан к адресу.

EINVAL — Некорректное значение **addrlen**, или в **addr** указан некорректный адрес для этого доменного сокета.

ENOTSOCK — Файловый дескриптор **sockfd** указывает не на сокет.

Следующие ошибки имеют место только для сокетов домена UNIX (**AF_UNIX**):

EACCES — Поиск запрещён из-за одного из частей префикса пути (**path_resolution(7)**).

EADDRNOTAVAIL — Запрошен несуществующий интерфейс или запрашиваемый адрес не является локальным.

EFAULT — **addr** указывает вне адресного пространства, доступного пользователю.

ELOOP — При определении **addr** превышено количество переходов по символьной ссылке.

ENAMETOOLONG — Аргумент **addr** слишком большой.

ENOENT — Компонент из каталожного префикса пути сокета не существует.

ENOMEM — Недостаточное количество памяти ядра.

ENOTDIR — Компонент в префиксе пути не является каталогом.

EROFS — Попытка создания inode сокета на файловой системе, доступной только для чтения.

Пример

Пример использования **bind()** с сокетами домена Internet можно найти в **getaddrinfo(3)**.

Следующий пример показывает как привязать потоковый сокет к домену UNIX (**AF_UNIX**) и принимать соединения:

```
#include <sys/socket.h>
#include <sys/un.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MY_SOCK_PATH "/somepath"
#define LISTEN_BACKLOG 50

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

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

    struct sockaddr_un my_addr;
    struct sockaddr_un peer_addr;
    socklen_t          peer_addr_size;

    int sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        handle_error("socket");
```

```
    memset(&my_addr, 0, sizeof(struct sockaddr_un)); // Очистка структуры
    my_addr.sun_family = AF_UNIX;
    strncpy(my_addr.sun_path, MY_SOCK_PATH, sizeof(my_addr.sun_path) - 1);
    if (bind(sfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr_un)) == -1)
        handle_error("bind");

    if (listen(sfd, LISTEN_BACKLOG) == -1)
        handle_error("listen");

    // Теперь мы можем принимать входящие соединения по одному с помощью accept(2)

    peer_addr_size = sizeof(struct sockaddr_un);
    int cfd        = accept(sfd, (struct sockaddr *)&peer_addr, &peer_addr_size);
    if (cfd == -1)
        handle_error("accept");

    // Здесь код обработки входящего соединения(й)

    // Если имя пути сокета, MY_SOCK_PATH, больше не требуется,
    // его нужно удалить с помощью unlink(2) или remove(3)
}
```

listen() — сообщает сокету, что должны быть приняты новые соединения

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

int listen(int sockfd,   // файловый дескриптор сокета
           int backlog); // максимальный размер очереди ожидающих соединений
```

Вызов **listen()** помечает сокет, указанный в **sockfd** как пассивный, то есть как сокет, который будет использоваться для приёма запросов входящих соединений с помощью **accept(2)**.

sockfd

Аргумент **sockfd** является файловым дескриптором, который ссылается на сокет типа **SOCK_STREAM** или **SOCK_SEQPACKET**.

backlog

Аргумент **backlog** задает максимальный размер, до которого может расти очередь ожидающих соединений у **sockfd**. Если приходит запрос на соединение, а очередь полна, то клиент может получить ошибку **ECONNREFUSED** или, если нижележащий протокол поддерживает повторную передачу, запрос может быть проигнорирован, чтобы попытаться соединиться позднее.

Возвращаемое значение

При успешном выполнении возвращается 0. В случае ошибки возвращается -1, а errno устанавливается в соответствующее значение.

Ошибки

EADDRINUSE — Другой сокет уже слушает на этом же порту.

EADDRINUSE — (доменные сокеты Интернета) Сокет, указанный **sockfd**, ранее не был привязан к адресу и при попытке привязать его к короткоживущему порту, было определено, что все номера в диапазоне короткоживущих портов уже используются.

EBADF — Аргумент **sockfd** не является допустимым файловым дескриптором.

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

EOPNOTSUPP — Тип сокета не поддерживает операцию **listen()**.

Замечания

В Linux 2.2 изменилось поведение аргумента **backlog** на TCP-сокетах. Теперь вместо количества неоконченных запросов на соединение он задает размер очереди для полностью (completely) установленных соединений, ожидающих, пока процесс примет их. Максимальный размер очереди для неполных сокетов может быть задан через **/proc/sys/net/ipv4/tcp_max_syn_backlog**.

Когда разрешено использование *syncookies*, логический максимальный размер отсутствует и эта настройка игнорируется. Подробности см. в **tcp(7)**.

Если значение аргумента **backlog** больше, чем значение **/proc/sys/net/core/somaxconn**, то оно без предупреждения обрезается до этой величины. Значение по умолчанию в этом файле может быть разным для разных реализаций:

```
[podenok@logovo]$ cat /proc/sys/net/core/somaxconn
4096
```

accept(), accept4() — принять соединение на сокете

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

int accept(int               sockfd,   // слушающий сокет
           struct sockaddr *addr,
           socklen_t       *addrlen);

#define _GNU_SOURCE
#include <sys/socket.h>

int accept4(int               sockfd,  // слушающий сокет
            struct sockaddr *addr,
            socklen_t       *addrlen,
            int              flags);
```

Системный вызов **accept( )** используется с сокетами, ориентированными на установление соединения (**SOCK_STREAM**, **SOCK_SEQPACKET**).

Он извлекает первый запрос на подключение из очереди ожидающих подключений прослушивающего сокета **sockfd**, создаёт новый подключенный сокет и возвращает новый файловый дескриптор, указывающий на этот сокет.

Вновь созданный сокет не находится в состоянии прослушивания.

Исходный сокет **sockfd** не изменяется при этом вызове.

sockfd — это сокет, созданный с помощью **socket(2)**, привязанный к локальному адресу с помощью **bind(2)**, и прослушивающий соединения после вызова **listen(2)**.

addr — это указатель на структуру **sockaddr**. В эту структуру помещается адрес ответной стороны в том виде, в каком он известен на коммуникационном уровне.

Точный формат адреса, возвращаемого в параметре **addr**, определяется семейством адресов сокета[3].

Если **addr** равен **NULL**, то ничего не помещается. В этом случае **addrlen** не используется и также должен быть **NULL**.

Через аргумент **addrlen** осуществляется возврат результата — вызывающая сторона должна указать в нём размер (в байтах) структуры, на которую указывает **addr**. При возврате он будет содержать реальный размер адреса ответной стороны.

Возвращаемый адрес обрезается, если предоставленный буфер окажется слишком маленьким. В этом случае в **addrlen** будет возвращено значение большее чем было в вызове.

Если в очереди нет ожидающих запросов на соединение, и сокет не помечен как неблокирующий, то **accept()** заблокирует вызвавшую программу до появления соединения.

Если сокет помечен как неблокирующий, а в очереди нет запросов на соединение, то **accept()** завершится с ошибкой **EAGAIN** или **EWOULDBLOCK**.

Для того, чтобы получать уведомления о входящих соединениях на сокете, можно использовать **select(2)**, **poll(2)** или **epoll(7)**.

Когда поступает запрос на новое соединение, доставляется доступное для чтения событие и после этого можно вызывать **accept()**, чтобы получить сокет для этого соединения.

Можно также настроить сокет так, чтобы он посылал сигнал **SIGIO**, когда на нём происходит какая-либо активность (**socket(7)**).

---

3) см. socket(2) и справочную страницу по соответствующему протоколу

flags

accept4() является нестандартным расширением Linux.

Если **flags** равно 0, то вызов **accept4()** равнозначен **accept()**. Следующие значения могут быть побитово сложены в **flags** для получения различного поведения:

SOCK_NONBLOCK — Устанавливает флаг состояния файла **O_NONBLOCK** для нового открытого файлового описания (**open(2)**), на которое ссылается новый файловый дескриптор.

Использование данного флага делает ненужными дополнительные вызовы **fcntl(2)** для достижения того же результата.

SOCK_CLOEXEC — Устанавливает флаг close-on-exec (**FD_CLOEXEC**) для нового открытого файлового дескриптора (**open(2)**).

Возвращаемое значение

При успешном выполнении данные системные вызовы возвращают неотрицательное целое, являющееся файловым дескриптором принятого сокета. В случае ошибки возвращается -1, **errno** устанавливается в соответствующее значение, а **addrlen** не изменяется.

Ошибки

EAGAIN или **EWOULDBLOCK** — сокет помечен как неблокирующий и нет ни одного соединения, которое можно было бы принять.

В POSIX.1-2001 и POSIX.1-2008 допускается в этих случаях возврат ошибки и не требуется, чтобы эти константы имели одинаковое значение, поэтому переносимое приложение должно проверять обе возможности.

EBADF — значение **sockfd** не является открытым файловым дескриптором.

ECONNABORTED — соединение было прервано.

EFAULT — аргумент **addr** не находится в пользовательском пространстве адресов с возможностью записи.

EINTR — системный вызов прервал сигналом, который поступил до момента прихода допустимого соединения.

EINVAL — сокет не слушает соединения или недопустимое значение addrlen (например, отрицательное).

EINVAL (**accept4()**) — недопустимое значение в **flags**.

EMFILE — было достигнуто ограничение по количеству открытых файловых дескрипторов на процесс.

ENFILE — достигнуто максимальное количество открытых файлов в системе.

ENOBUFS, **ENOMEM** — не хватает свободной памяти. Это зачастую означает, что выделение памяти ограничено размерами буфера сокетов, а не системной памятью.

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

EOPNOTSUPP — тип сокета, на который ссылается дескриптор, отличается от **SOCK_STREAM**.

EPROTO — ошибка протокола.

Linux **accept()** может завершиться с ошибкой если:

EPERM — правила межсетевого экрана запрещают соединение.

Вдобавок, могут также возвращаться сетевые ошибки на новом сокете и ошибки, могущие возникнуть в протоколе.

Различные ядра Linux могут возвращать другие ошибки, например, **ENOSR**, **ESOCKTNOSUPPORT**, **EPROTONOSUPPORT**, **ETIMEDOUT**.

Значение ошибки **ERESTARTSYS** можно увидеть при трассировке.

Обработка ошибок

В Linux вызовы **`accept()`** и **`accept4()`** передают уже ожидающие сетевые ошибки на новый сокет как код ошибки из вызова **`accept()`**. Это поведение отличается от других реализаций BSD-сокетов.

Для надёжной работы приложения должны отслеживать сетевые ошибки, которые могут появиться при работе с протоколом **`accept()`** и обрабатывать их как **`EAGAIN`**, повторно выполняя вызов.

В случае TCP/IP такими ошибками являются **`ENETDOWN`**, **`EPROTO`**, **`ENOPROTOOPT`**, **`EHOSTDOWN`**, **`ENONET`**, **`EHOSTUNREACH`**, **`EOPNOTSUPP`** и **`ENETUNREACH`**.

В Linux новый сокет, возвращаемый `accept()`, не наследует файловые флаги состояния такие как `O_NONBLOCK` и `O_ASYNC` от прослушивающего сокета.

Это поведение отличается от каноническое реализации сокетов BSD.

Переносимые программы не должны полагаться на наследуемость файловых флагов состояния или её отсутствия и всегда должны устанавливать на сокете, полученном от `accept()`, все требуемые флаги явно.

Замечания

Возможно, после доставки **SIGIO** или если вызовы **select(2)**, **poll(2)** или **epoll(7)** вернут событие доступности чтения, не всегда будет иметь место ожидающий запрос на подключение. Если это случается, то вызов **accept()** блокируется в ожидании прибытия следующего подключения.

Чтобы гарантировать, что **accept()** никогда не заблокируется, сокету **sockfd** необходимо устанавливать флаг **O_NONBLOCK**.

connect() — подключает сокет к удаленному адресу сокета

Вызов **connect()** — инициирует соединение на сокете

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

int connect(int                    sockfd,   // сокет инициатора соединения
            const struct sockaddr *addr,     // адрес соединения
            socklen_t              addrlen); // размер addr
```

Системный вызов **connect()** устанавливает соединение сокета, который задан файловым дескриптором **sockfd**, с адресом, указанным в **addr**.

Аргумент **addrlen** определяет размер **addr**. Формат адреса в **addr** определяется адресным пространством сокета **sockfd**, как это описано выше.

addr

Если сокет **sockfd** имеет тип **SOCK_DGRAM**, то адрес **addr** является адресом по умолчанию, куда посылаются датаграммы, и единственным адресом, откуда они принимаются.

Если сокет имеет тип **SOCK_STREAM** или **SOCK_SEQPACKET**, то данный системный вызов попытается установить соединение с другим сокетом, заданным параметром **addr**.

Обычно сокеты с протоколами, основанными на соединении, могут устанавливать соединение **connect()** только один раз.

Сокеты с протоколами без установления соединения могут использовать **connect()** многократно. Это делается для изменения адреса назначения.

Сокеты без установления соединения могут прекратить связь с другим сокетом, установив член **sa_family** структуры **sockaddr** в **AF_UNSPEC**.

Возвращаемое значение

Если соединение или привязка прошла успешно, возвращается ноль.

При ошибке возвращается -1, а **errno** устанавливается должным образом.

Ошибки

Ниже приведены только общие ошибки сокетов.

Могут также появляться коды ошибок, существующие в конкретном домене.

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

EACCES, **EPERM** — пользователь попытался соединиться с широковещательным адресом, не установив широковещательный флаг на сокете или же запрос на соединение завершился неудачно из-за правила локального межсетевого экрана.

EADDRINUSE — локальный адрес уже используется.

EADDRNOTAVAIL — (доменные сокеты Интернета) сокет, указанный **sockfd**, ранее не был привязан к адресу и при попытке привязать его к короткоживущему порту, было определено, что все номера в диапазоне короткоживущих портов уже используются[4].

EAFNOSUPPORT — адрес имеет некорректное семейство адресов в поле **sa_family**.

EAGAIN — для неблокирующих доменных сокетов UNIX сокет не блокируется и соединение не может быть выполнено немедленно. Для других семейств сокетов в кэше

---

4) См. /proc/sys/net/ipv4/ip_local_port_range в ip(7).

маршрутизации недостаточно элементов.

EALREADY — сокет является неблокирующим, а предыдущая попытка установить соединение ещё не завершилась.

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

ECONNREFUSED — Вызов **connect()** не нашёл слушающий удалённый адрес для потокового сокета.

EFAULT — адрес структуры сокета находится за пределами пользовательского адресного пространства.

EINPROGRESS — сокет является неблокирующим, а соединение не может быть установлено немедленно (доменные сокеты UNIX вместо этого возвращают ошибку **EAGAIN**).

Можно использовать **select(2)** или **poll(2)**, чтобы закончить соединение, установив ожидание возможности записи в сокет. После того, как **select(2)** сообщит о такой возможности, можно использовать **getsockopt(2)**, чтобы прочитать флаг **SO_ERROR** на уровне **SOL_SOCKET** и определить, успешно ли завершился **connect()** (в этом случае **SO_ERROR** равен нулю) или неудачно (тогда **SO_ERROR** равен одному из обычных кодов ошибок, перечисленных здесь, и объясняет причину неудачи).

EINTR — системный вызов был прерван пойманным сигналом.

EISCONN — соединение на сокете уже произошло.

ENETUNREACH — сеть недоступна.

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

EPROTOTYPE — тип сокета не поддерживается запрошенным протоколом связи. Это ошибка может возникать при попытке подключить доменный датаграммный сокет UNIX к потоковому сокету.

ETIMEDOUT — Произошел тайм-аут во время ожидания соединения. Сервер, возможно, очень занят и не может принимать новые соединения. Для IP-сокетов тайм-аут может быть очень длинным, если на сервере разрешено использование *syncookies*.

Замечания

Если вызов **connect()** завершается с ошибкой, то состояние сокета считается неопределённым. Переносимые приложения должны закрывать сокет и для переподключения создавать новый.

Пример использования **connect()** показан в **getaddrinfo(3)**.

select() — ожидает готовности операций ввода-вывода

```
// в соответствии с 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 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
};
```

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, файловый дескриптор готов для чтения, если он указывает на конец файла).

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

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

При возврате из вызова наборы файловых дескрипторов изменяются, показывая какие файловые дескрипторы фактически изменили состояние. Это значит, если используется **`select()`** в цикле, то наборы дескрипторов должны заново инициализироваться перед каждым вызовом.

Значение каждого из трёх наборов файловых дескрипторов может быть задано как **`NULL`**, если слежение за определённым классом событий над файловыми дескрипторами не требуется.

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.

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.

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.

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		§	П (UTF-8)		П (UTF-16BE)		П (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).