Лабораторная работа «Сокеты»

Сокеты в пространстве файловых имен и сетевые сокеты

Сокеты были созданы как универсальное средство взаимодействия параллельных процессов, причем безразлично, где они выполняются: на одной машине или на разных. Абстракция сокетов была введена в BSD Unix (Berkley Software Distribution), поэтому сокеты часто называют сокетами BSD.

Сокет представляет собой абстракцию конечной точки взаимодействия.

```
#include <sys/types.h>
#include <sys/socket.h>
sockfd = socket(int socket_family, int socket_type, int protocol);
```

socket() создаёт, так называемую, конечную точку соединения (взаимодействия, коммуникации) и возвращает файловый дескриптор.

В Линукс определен унифицированный интерфейс между пользовательскими процессами и стеком сетевых протоколов в ядре.

Модули протоколов группируются по *семействам протоколов*, такими, как **AF_INET**, **AF_IPX** и **AF_PACKET**, и *типам сокетов*, такими, как **SOCK_STREAM** или **SOCK_DGRAM**. Более подробная информация о семействах и типах приведена в **socket**(2).

Первый параметр *domain* или семейство (family) задает домен соединения или семейство адресов (address family - AF). Семейства описаны в <*sys/socket.h*>. В настоящее время определены такие форматы:

- AF_UNIX – сокеты в файловом пространстве имен.
- AF_INET – сетевые сокеты по протоколу TCP/IP для сетей IPv4.
- AF_INET6 - сетевые сокеты по протоколу TCP/IP для сетей IPv6.
- AF_UNSPEC – не специфицированное семейство

и некоторые другие.

Второй параметр функции **socket**() - **тип** сокета (*type)*. В настоящее время определены следующие типы:

SOCK_STREAM
Обеспечивает создание двусторонних, надёжных потоков байтов на основе установления соединения. Может также поддерживаться механизм внепоточных 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
Этот тип устарел и не должен использоваться в новых программах; см. packet(7).

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

Начиная с Linux 2.6.27, аргумент *type* предназначается для двух вещей: кроме определения типа сокета, для изменения поведения **socket**() он может содержать побитовую сумму любых следующих значений:
SOCK_NONBLOCK - устанавливает флаг состояния файла **O_NONBLOCK** для нового открытого файлового дескриптора. Использование данного флага заменяет дополнительные вызовы **fcntl**(2) для достижения того же результата.
SOCK_CLOEXEC - устанавливает флаг close-on-exec (**FD_CLOEXEC**) для нового открытого файлового дескриптора. Смотрите описание флага **O_CLOEXEC** в **open**(2) для того, чтобы узнать, как это может пригодиться.

Since Linux 2.6.27, the *type* argument serves a second purpose: in addition to specifying a socket type, it may include the bitwise OR of any of the following values, to modify the behavior of **socket**():
SOCK_NONBLOCK
Set the **O_NONBLOCK** file status flag on the new open file description. Using this flag saves extra calls to **fcntl**(2) to achieve the same result.
SOCK_CLOEXEC
Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in **open**(2) for reasons why this may be useful.

В третьем параметре - ***protocol*** задаётся определённый протокол, используемый с сокетом. Обычно, только единственный протокол существует для поддержи определённого типа сокета с заданным семейством протоколов. В этом случае для параметра *protocol* можно указать 0. Однако, может существовать несколько протоколов.
Номер используемого протокола зависит от "домена соединения", по которому устанавливается соединение; см. **protocols**(5). Смотрите **getprotoent**(3), где описано, как соотносить имена протоколов с их номерами.

Системные вызовы:

```
asmlinkage long sys_socketcall(int call, unsigned long *args)
{
        unsigned long a[6];
        unsigned long a0,a1;
        int err;

        if(call<1||call>SYS_RECVMSG)
                return -EINVAL;
```

```
        /* copy_from_user should be SMP safe. */
        if (copy_from_user(a, args, nargs[call]))
                return -EFAULT;

        a0=a[0];
        a1=a[1];

        switch(call)
        {
                case SYS_SOCKET:
                        err = sys_socket(a0,a1,a[2]);
                        break;
                case SYS_BIND:
                        err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
                        break;
                case SYS_CONNECT:
                        err = sys_connect(a0, (struct sockaddr *)a1, a[2]);

        .....
                default:
                        err = -EINVAL;
                        break;
        }
        return err;
}
```

Функция **sys_socket()** создает сокет:

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
        int retval;
        struct socket *sock;

        retval = sock_create(family, type, protocol, &sock);
        if (retval < 0)
                goto out;

        retval = sock_map_fd(sock);
        if (retval < 0)
                goto out_release;

out:
        /* It may be already another descriptor 8) Not kernel problem. */
        return retval;

out_release:
        sock_release(sock);
        return retval;
}
```

Структура struct socket описывает сокет.

```
/**
 *  struct socket - general BSD socket
 *  @state: socket state (%SS_CONNECTED, etc)
 *  @type: socket type (%SOCK_STREAM, etc)
 *  @flags: socket flags (%SOCK_NOSPACE, etc)
 *  @ops: protocol specific socket operations
 *  @file: File back pointer for gc
 *  @sk: internal networking protocol agnostic socket representation
 *  @wq: wait queue for several uses
 */
struct socket {
     socket_state           state;

     short                  type;
```

```
        unsigned long           flags;

        struct socket_wq __rcu  *wq;

        struct file             *file;
        struct sock             *sk;
        const struct proto_ops  *ops;
};
```

ubuntu manual

```
socketcall - socket system calls
#include <linux/net.h>
       int socketcall(int call, unsigned long *args);
socketcall()  is a common kernel entry point for the socket system calls.
```

Название	Назначение	Справочная страница
AF_UNIX, **AF_LOCAL**	Локальное соединение	**unix**(7)
AF_INET	Протоколы Интернет IPv4	**ip**(7)
AF_INET6	Протоколы Интернет IPv6	**ipv6**(7)
AF_IPX	Протоколы Novell IPX	
AF_NETLINK	Устройство для взаимодействия с ядром	**netlink**(7)
AF_X25	Протокол ITU-T X.25/ISO-8208	**x25**(7)
AF_AX25	Протокол любительского радио AX.25	
AF_ATMPVC	Доступ к низкоуровневым PVC в ATM	
AF_APPLETALK	AppleTalk	**ddp**(7)
AF_PACKET	Низкоуровневый пакетный интерфейс	**packet**(7)
AF_ALG	Интерфейс к ядерному крипто-API	

Типы сокетов

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). 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). Когда сеанс закончен, выполняется команда **close**(2). Внепоточные данные могут передаваться, как описано в **send**(2), и приниматься, как описано в **recv**(2).

Протоколы связи, которые реализуют **SOCK_STREAM**, выполняются такБ 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_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_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.

Тип **SOCK_PACKET** считается устаревшим типом сокета; он позволяет получать необработанные пакеты прямо от драйвера устройства. Используйте вместо него **packet**(7).

Системный вызов **fcntl**(2) с аргументом **F_SETOWN** может использоваться для задания группы процессов, которая будет получать сигнал **SIGURG**, когда прибывают внепоточные данные, или сигнал **SIGPIPE**, когда соединение типа **SOCK_STREAM** неожиданно обрывается. 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*>. Вызовы **setsockopt**(2) и **getsockopt**(2) используются, чтобы установить и получить необходимые параметры соответственно.

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 *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.

Другие ошибки могут быть созданы модулями протоколов более низкого уровня.

СООТВЕТСТВИЕ СТАНДАРТАМ

POSIX.1-2001, POSIX.1-2008, 4.4BSD.
Флаги **SOCK_NONBLOCK** и **SOCK_CLOEXEC** есть только в Linux.
Вызов **socket**() появился в 4.2BSD. Обычно он переносим в/из не-BSD систем на уровне сокетов BSD (включая варианты System V).

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, вероятно, 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_*.
Тип протокола **AF_ALG** был добавлен в Linux 2.6.38.

Структуры адреса сокета

Каждый сокетный домен имеет свой формат сокетных адресов, выраженный в отдельной адресной структуре. Каждая из этих структур начинается с целочисленного поля «семейства» (с типом *sa_family_t*), в котором указывается тип адресной структуры. Это позволяет различным системным вызовам (например, **connect**(2), **bind**(2), **accept**(2),**getsockname**(2), **getpeername**(2)), которые являются общими для всех сокетов, определить домен конкретного сокетного адреса.
Для передачи сокетного адреса любого типа через программный интерфейс сокетов служит тип ***struct sockaddr***. Целью данного типа является приведение типов сокетных адресов определённого домена к «общему» типу, что позволяет избежать предупреждений компилятора о несовпадении типов в вызовах API сокетов.

Функции, определенные на сокетах

Эти функции используются пользовательскими процессами для отправки или приёма пакетов и выполнения других операций над сокетами. Более подробная информация приведена в соответствующих справочных страницах.

Вызовы:

- **socket**(2) создаёт сокет,
- **connect**(2) соединяет сокет с удалённым сокетным адресом,
- **bind**(2) привязывает сокет к локальному адресу,
- **listen**(2) сообщает сокету, что должны приниматься новые соединения,
- **accept**(2) используется для получения нового сокета для нового входящего соединения.
- **send**(2), **sendto**(2) и **sendmsg**(2) отправляют данные в сокет,
- **recv**(2),**recvfrom**(2) и **recvmsg**(2) принимают данные из сокета.

- для чтения и записи данных могут использоваться стандартные операции ввода-вывода: **write**(2),**writev**(2), **sendfile**(2), **read**(2) и **readv**(2).
- **poll**(2) и **select**(2) ожидают поступления данных или готовятся к передаче данных. **Пользователь может подождать наступления различных событий через poll() или select().**

Вызов **socketpair**(2) возвращает два соединённых анонимных сокета (реализовано только для некоторых локальных семейств, например **AF_UNIX**).

Вызов **getsockname**(2) возвращает адрес локального сокета, а **getpeername**(2) возвращает адрес удалённого сокета. Вызовы **getsockopt**(2) и **setsockopt**(2) используются для установки или считывания параметров протокола или уровня сокетов. Вызов **ioctl**(2) может быть использован для установки или чтения некоторых других параметров.

Вызов **close**(2) используется для закрытия сокета. Вызов **shutdown**(2) закрывает части полнодуплексного сокетного соединения.

Перемещение (seeking), или вызовы **pread**(2) и **pwrite**(2) с ненулевой позицией, для сокетов не поддерживается.

Для **сокетов** возможно создание **неблокирующего ввода/вывода** путём установки в файловый дескриптор сокета флага **O_NONBLOCK** с помощью вызова **fcntl**(2). При этом все блокировавшие раньше операции, будут возвращать **EAGAIN** (операция должна быть повторена позднее); **connect**(2) возвратит ошибку **EINPROGRESS**.

События ввода-вывода		
Событие	Флаг poll	Когда происходит
Чтение	POLLIN	Поступили новые данные
Чтение	POLLIN	Установка соединения выполнена (для сокетов, ориентированных на соединение)
Чтение	POLLHUP	Другая сторона инициировала запрос на разъединение
Чтение	POLLHUP	Соединение разорвано (только для протоколов, ориентированных на соединение). Если производится запись в сокет, то также посылается сигнал **SIGPIPE**
Запись	POLLOUT	Сокет имеет достаточно места в буфере отправки для записи в него новых данных
Чтение/Запись	POLLIN|POLLOUT	Исходящий вызов **connect**(2) завершён
Чтение/Запись	POLLERR	Произошла асинхронная ошибка

Чтение/Запись	POLLHUP	Другая сторона закрыла (shut down) одно направление
Исключение	POLLPRI	Пришли неотложные данные. При этом посылается сигнал **SIGURG**

В ядре существует возможность информировать приложение о событиях с помощью сигнала **SIGIO, что является алтернативой** вызовам **poll**(2) и **select**(2). Для этого необходимо установить с помощью **fcntl**(2) в файловом дескрипторе сокета флаг **O_ASYNC**, а также назначить с помощью **sigaction**(2) корректный обработчик сигнала **SIGIO**.

Адреса сокетов

Функция socket() создает "безымянный" сокет, т.е. не связанный ни с локальным адресом, ни с номером порта. Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (эту процедуру называют именованием сокета). Иногда связывание осуществляется неявно (внутри функций **connect** и **accept**), но выполнять его необходимо во всех случаях. Вид адреса зависит от выбранного вами домена. В Unix-домене это текстовая строка - имя файла, через который происходит обмен данными. В Internet-домене адрес задаётся комбинацией IP-адреса и 16-битного номера порта. IP-адрес определяет хост в сети, а порт - конкретный сокет на этом хосте. Протоколы TCP и UDP используют различные пространства портов.

Для явного связывания сокета с некоторым адресом используется функция **bind**. Её прототип имеет вид:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

В качестве первого параметра передаётся дескриптор сокета, который привязывается к заданному адресу. Второй параметр – **addr** - содержит указатель на структуру struct sockaddr, а третий - длину этой структуры.

```
struct sockaddr
{
    unsigned short sa_family; // Семейство адресов, AF_xxx
    char sa_data[14]; // 14 байтов для хранения адреса
 };
```

Структура определяет адрес в самом общем виде. Для сетевого взаимодействия определена другая структура **sockaddr_in**:

```
struct sockaddr_in
 {
   short int sin_family; // Семейство адресов
   unsigned short int sin_port; // Номер порта
   struct in_addr sin_addr; // IP-адрес
   unsigned char sin_zero[8]; // Дополнение до размера структуры sockaddr
 };
```

Здесь поле **sin_family** соответствует полю **sa_family** в **sockaddr**, в **sin_port** записывается номер порта, а в **sin_addr** - IP-адрес хоста. Поле **sin_addr** само является структурой, которая имеет вид:

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

Зачем понадобилось заключать всего одно поле в структуру? Дело в том, что раньше **in_addr** представляла собой объединение (union), содержащее гораздо большее число полей. Сейчас, когда в ней осталось всего одно поле, она продолжает использоваться для обратной совместимости.

И ещё одно важное замечание. Существует два порядка хранения байтов в слове (2 байта) и двойном слове (4 байта). Один из них называется *порядком хоста* (host byte order), другой -*сетевым порядком* (network byte order) хранения байтов (big-endian).

Чтобы было понятнее, рассмотрим пример. 4-байтное целое число 0x01020304 будет сохранено в памяти системы big endian следующим образом:

Байт0	Байт1	Байт2	Байт3
0x01	0x02	0x03	0x04

Та же самая величина, которая будет храниться в памяти системы little endian, разместится в противоположном порядке:

Байт0	Байт1	Байт2	Байт3
0x04	0x03	0x02	0x01

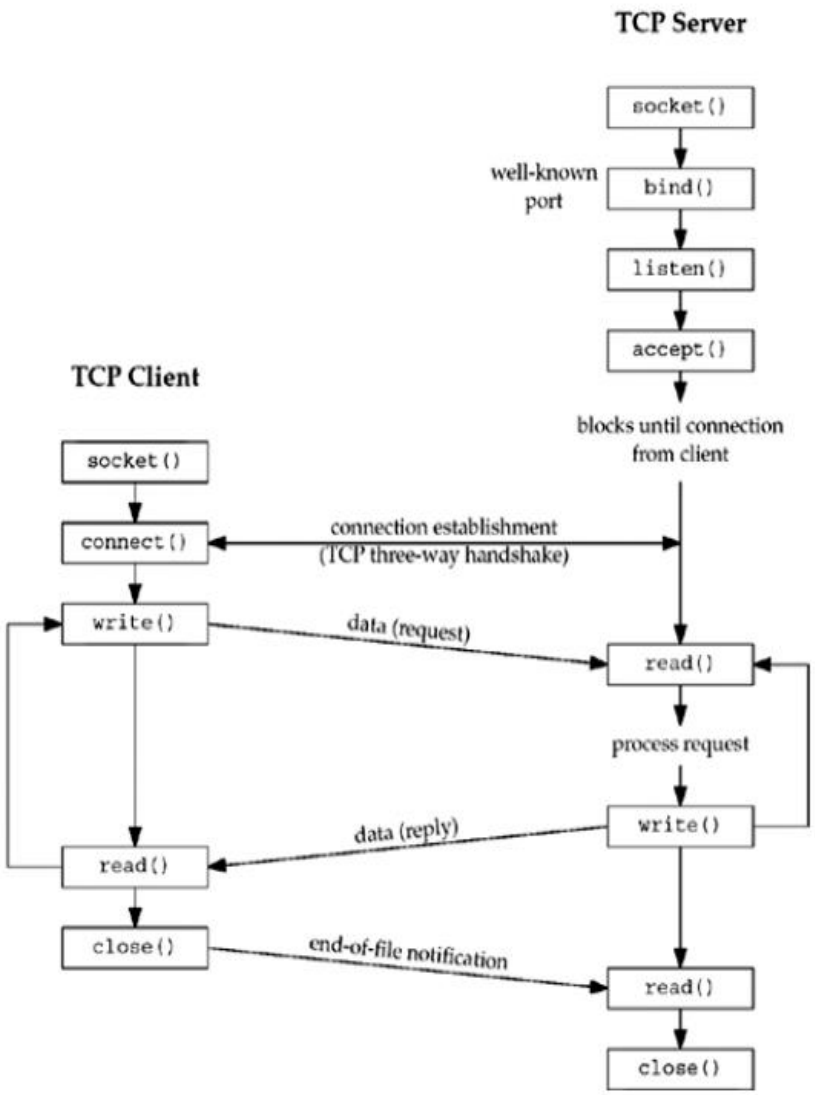
Обычно при программировании можно не обращать внимания на endianness, то есть не важно, как процессор сохранит байты чисел в системе - big endian или little endian; ядро CPU просто загружает данные из памяти и сохраняет данные в память, и представляет данные в Вашей программе уже в правильном виде. Однако, когда нужно обмениваться данными с другой системой, обе системы должны учитывать формат хранения данных в памяти (endianness).

Linux kernel может быть либо big endian, либо little endian, в зависимости от архитектуры, в расчете на которую kernel скомпилировано. Ниже в таблице показан endianness для различных типов архитектур процессоров и протоколов.

Big Endian	Little Endian	Оба варианта
Архитектуры процессоров		
AVR32 FR-V H8300 PA-RISC S390 Motorola 680x0 PowerPC SPARC	Alpha CRIS Blackfin Intel 64 IA-32 (x86) MN10300 AT91SAM7	ARM SuperH (sh) M32R MIPS Xtensa
Протоколы		
TCP/IP	USB	

При указании IP-адреса и номера порта необходимо преобразовать число из порядка хоста в сетевой. Для этого используются функции **htons** (Host To Network Short) и **htonl** (Host To Network Long). Обратное преобразование выполняют функции **ntohs (**Network To Host Short) и **ntohl**.

Взаимодействие параллельных процессов через сокеты выполняется по модели клиент-сервер.



manpages-dev 4.16-1 all

```
User programs should call the appropriate functions by their usual names.  Only
standard
       library implementors and kernel hackers need to know about socketcall().

       call              Man page
       SYS_SOCKET        socket(2)
       SYS_BIND          bind(2)
       SYS_CONNECT       connect(2)
       SYS_LISTEN        listen(2)
       SYS_ACCEPT        accept(2)
       SYS_GETSOCKNAME   getsockname(2)
```

```
SYS_GETPEERNAME   getpeername(2)
SYS_SOCKETPAIR    socketpair(2)
SYS_SEND          send(2)
SYS_RECV          recv(2)
SYS_SENDTO        sendto(2)
SYS_RECVFROM      recvfrom(2)
SYS_SHUTDOWN      shutdown(2)
SYS_SETSOCKOPT    setsockopt(2)
SYS_GETSOCKOPT    getsockopt(2)
SYS_SENDMSG       sendmsg(2)
SYS_RECVMSG       recvmsg(2)
SYS_ACCEPT4       accept4(2)
SYS_RECVMMSG      recvmmsg(2)
SYS_SENDMMSG      sendmmsg(2)
```

Сокеты в файловом пространстве имен

Сокеты в файловом пространстве имен (*file namespace*, их еще называют «сокеты Unix») используют в качестве адресов имена файлов специального типа. Важной особенностью этих сокетов является то, что соединение с их помощью локального и удаленного приложений невозможно, даже если файловая система, в которой создан сокет, доступна удаленной операционной системе.
Сокеты для локального межпроцессного взаимодействия

#include <sys/socket.h>
#include <sys/un.h>
unix_socket **= socket(AF_UNIX, type, 0);**
error **= socketpair(AF_UNIX, type, 0, int ***sv*);

Семейство сокетов **AF_UNIX** (также известное, как **AF_LOCAL**) используется для эффективного взаимодействия между процессами на одной машине. Доменные сокеты UNIX могут быть как безымянными, так и иметь имя файла в файловой системе (типизированный сокет). В Linux также поддерживается абстрактное пространство имён, которое не зависит от файловой системы.

Допустимые типы сокета для домена UNIX: потоковый сокет **SOCK_STREAM**, датаграмный сокет **SOCK_DGRAM**, сохраняющий границы сообщений (в большинстве реализаций UNIX, доменные датаграмные сокеты UNIX всегда надёжны и не меняют порядок датаграмм); и (начиная с Linux 2.6.4) ориентированный на соединение задающий последовательность пакетам сокет **SOCK_SEQPACKET**, сохраняющий границы сообщений и доставляющий сообщения в том же порядке, в каком они были отправлены.

Доменные сокеты UNIX поддерживают передачу файловых дескрипторов или информацию (credentials) о процессе другим процессам, используя вспомогательные (ancillary) данные.

В следующем фрагменте кода создается сокет и связываем его с файлом socket.soc:

```
sock = socket(AF_UNIX, SOCK_DGRAM, 0);
if (sock < 0) {
perror("socket failed");
return EXIT_FAILURE;
}
srvr_name.sa_family = AF_UNIX;
strcpy(srvr_name.sa_data, "socket.soc");
if (bind(sock, &srvr_name, strlen(srvr_name.sa_data) +
sizeof(srvr_name.sa_family)) < 0) {
perror("bind failed");
return EXIT_FAILURE;
}
```

Константы и функции, необходимые для работы с сокетами в файловом пространстве имен, объявлены в файлах <sys/types.h> и <sys/socket.h>. Как и файлы, сокеты в программах представлены дескрипторами. Дескриптор сокета можно получить с помощью функции socket(2). Первый параметр этой функции – домен, к которому принадлежит сокет. Домен сокета обозначает тип соединения (а не доменное имя Интернета). Домен, обозначенный константой AF_UNIX, соответствует сокетам в файловом пространстве имен. Второй параметр socket() определяет тип сокета. значение SOCK_DGRAM указывает датаграммный сокет. В пространстве файловых имен датаграммные сокеты также надежны, как и потоковые сокеты. Третий параметр функции socket() позволяет указать протокол, используемый для передачи данных. Значение этого параметра равно нулю. В случае ошибки функция socket() возвращает -1.

После получения дескриптора сокета вызывается функция bind(2), которая связывает сокет с заданным адресом (связывать сокет с адресом необходимо в программе-сервере, но не в клиенте). Первым параметром функции является дескриптор, а вторым – указатель на структуру sockaddr (переменная srvr_name), содержащую адрес, на котором регистрируется сервер (третий параметр функции – длина структуры, содержащей адрес). Вместо общей структуры sockaddr для сокетов Unix (сокетов в файловом пространстве имен) можно использовать специализированную структуру sockaddr_un. Поле sockaddr.sa_family позволяет указать семейство адресов, которым мы будем пользоваться. В данном случае это семейство адресов файловых сокетов Unix AF_UNIX. Сам адрес семейства AF_UNIX (поле sa_data) представляет собой обычное имя файла сокета. После вызова bind() программа-сервер становится доступна для соединения по заданному адресу (имени файла).

При обмене данными с датаграммными сокетами используются специальные функции recvfrom(2) и sendto(2). Эти же функции могут применяться и при работе с потоковыми сокетами. Для чтения данных из датаграммного сокета используется функция recvfrom(2), которая по умолчанию блокирует программу до тех пор, пока на входе не появятся новые данные.

```
bytes = recvfrom(sock, buf, sizeof(buf), 0, &rcvr_name, &namelen);
```

При вызове функции recvfrom() ей передается указатель на еще одну структуру типа sockaddr, в которой функция возвращает данные об адресе клиента, запросившего соединение (в случае файловых сокетов этот параметр не несет полезной информации). Последний параметр функции recvfrom() – указатель на переменную, в которой будет возвращена длина

структуры с адресом. Если информация об адресе клиента не нужна, то можно передать значения NULL в предпоследнем и последнем параметрах. По завершении работы с сокетом он «закрывается» с помощью «файловой» функции close(). Перед выходом из программы-сервера следует удалить файл сокета, созданный в результате вызова socket(), что делается с помощью функции unlink().

В программе-клиента (fsclient.c) открывается сокет с помощью функции socket() и передаются данные (текстовую строка) серверу с помощью функции sendto(2):

```
srvr_name.sa_family = AF_UNIX;
strcpy(srvr_name.sa_data, SOCK_NAME);
strcpy(buf, "Hello, Unix sockets!");
sendto(sock, buf, strlen(buf), 0, &srvr_name,
strlen(srvr_name.sa_data) + sizeof(srvr_name.sa_family));
```

Первый параметр функции sendto() – дескриптор сокета, второй и третий параметры позволяют указать адрес буфера для передачи данных и его длину. Четвертый параметр предназначен для передачи дополнительных флагов. Предпоследний и последний параметры несут информацию об адресе сервера и его длине, соответственно. Если при работе с датаграммными сокетами вызвать функцию connect(2) (см. ниже), то можно не указывать адрес назначения каждый раз (достаточно указать его один раз, как параметр функции connect()). Перед вызовом функции sendto() нам надо заполнить структуру sockaddr (переменную srvr_name) данными об адресе сервера. После окончания передачи данных сокет закрывается с помощью close(). Если запустить программу-сервер, а затем программу-клиент, то сервер распечатает тестовою строку, переданную клиентом.

Сетевые сокеты

Сетевые сокеты являются универсальным типом сокетов. Система приложений, предназначенных для работы на одном компьютере, может использовать сетевые сокеты для обмена данными. Использование сетевых сокетов позволяет сделаеть процесс масштабирования проекта безпроблемным. Обнако даже если сокеты используются для обмена данными на одной и той же машине, передаваемые данные должны пройти все уровни сетевого стека, что отрицательно сказывается на быстродействии и нагрузке на систему.

В качестве примера рассмотривается комплекс из двух приложений, клиента и сервера, использующих сетевые сокеты для обмена данными. Рассмотрим некоторые фрагменты программы. Прежде всего, надо получить дескриптор сокета:

```
sock_fd = socket(AF_INET, SOCK_STREAM, 0);
if (sock_fd < 0) {
printf("socket() failed: %d\n", errno);
return EXIT_FAILURE;
}
```

В первом параметре функции socket() передаеся константа AF_INET, которая указывает, что открываемый сокет должен быть сетевым. Значение второго параметра требует, чтобы сокет был потоковым. Третий параметр равен 0, т.е. протокол выбирается по умолчанию. Затем вызывается функция bind():

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

```
serv_addr.sin_port = htons(port);
if (bind(sock_fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
printf("bind() failed: %d\n", errno);
return EXIT_FAILURE;
}
```

Переменная serv_addr, - это структура типа sockaddr_in. Тип sockaddr_in специально предназначен для хранения адресов в формате Интернета. Самое главное отличие sockaddr_in от sockaddr_un – наличие параметра sin_port, предназначенного для хранения значения порта. Функция htons() переписывает двухбайтовое значение порта так, чтобы порядок байтов соответствовал сетевому. В качестве семейства адресов указывается AF_INET, а в качестве самого адреса – специальную константу INADDR_ANY. Благодаря этой константе наша программа сервер зарегистрируется на всех адресах той машины, на которой она выполняется.

Сетевой сервер должен обрабатывать запросы множества клиентов одновременно. При этом в соединениях «точка-точка», например, при использовании потоковых сокетов, для каждого клиента у сервера должен быть открыт отдельный сокет. Для обеспечения такой возможности вызывается функция listen(2), которая переводит сервер в режим ожидания (пассивного) запроса на соединение:

```
listen(sock, 1);
```

Второй параметр listen() – максимальное число соединений, которые сервер может обрабатывать параллельно.

Запрос на пассивное открытие соединения означает, что процесс ждет получения запросов на соединение вместо того, чтобы пытаться самому установить его. Часто процесс, сделавший запрос на пассивное открытие, будет принимать запросы на соединение от любого другого процесса.

Процессы могут осуществлять пассивные открытия соединений и ждать, пока от других процессов придут соответствующие запросы на активное открытие, а протокол TCP проинформирует их об установлении соединения. Два процесса, сделавшие друг другу одновременно запросы на активное открытие, получат корректное соединение. Гибкость такого подхода становится критичной при поддержке распределенных вычислений, когда компоненты системы взаимодействуют друг с другом асинхронным образом.

Когда осуществляется подбор сокетов для локального запроса пассивного открытия и чужого запроса на активное открытие, то принципиальное значение имеют два случая. В первом случае местное пассивное открытие полностью определяет чужой сокет. При этом подбор должен осуществляться очень аккуратно. Во втором случае во время местного пассивного открытия чужой сокет не указывается. Тогда в принципе может быть установлено соединение с любых чужих сокетов. Во всех остальных случаях подбор сокетов имеет частичные ограничения.

Если на один и тот же местный сокет осуществлено несколько ждущих пассивных запросов на открытие (записанных в блоки TCB), и осуществляется извне активный запрос на открытие, то чужой активный сокет будет связываться с тем блоком TCB, где было указание именно на этот запросивший соединения сокет. И только если такого блока TCB не существует, выбор партнера осуществляется среди блоков TCB с неопределенным чужим сокетом.

Затем вызывается функция accept(2), которая устанавливает соединение в ответ на запрос клиента и создает копию сокета для того, чтобы исходный сокет мог продолжать прослушивание:

```
newsock_fd = accept(sock_fd, (struct sockaddr *) &cli_addr, &clen);
if (newsock_fd < 0)
{
  printf("accept() failed: %d\n", errno);
  return EXIT_FAILURE;
}
```

Получив запрос на соединение, функция accept() возвращает файловый дескриптор нового сокета, который открыт для обмена данными с клиентом, запросившего соединение. Сервер как бы перенаправляет запрошенное соединение на другой сокет, оставляя сокет sock свободным для прослушивания запросов на установку соединения. Второй параметр функции accept() содержит сведения об адресе клиента, запросившего соединение, а третий параметр указывает размер второго.

Процедура установки соединения по протоколу TCP использует флаг управления синхронизацией (SYN) и трижды обменивается сообщениями. Такой обмен называется трехвариантным подтверждением.
Соединение инициируется при встрече пришедшего сегмента, несущего флаг синхронизации (SYN), и ждущей его записи в блоке TCB. И сегмент, и запись создаются пришедшими от пользователей запросами на открытие.
Соответствие местного и чужого сокетов устанавливается при инициализации соединения. Соединение признается установленным, когда номера очередей синхронизированы в обоих направлениях между сокетами.
Отмена соединения также включает обмен сегментами, несущими на этот раз управляющий флаг FIN.

Так же как и при вызове функции recvfom(), может быть передано значение NULL в последнем и предпоследнем параметрах. Для чтения и записи данных сервер использует функции read() и write(), а для закрытия сокетов, естественно, close().

В программе-клиенте, прежде всего, нужно решить задачу, с которой не было при написании сервера, а именно выполнить преобразование доменного имени сервера в его сетевой адрес. Разрешение доменных имен выполняет функция gethostbyname():

```
server = gethostbyname(argv[1]);
if (server == NULL)
 {
   printf("Host not found\n");
   return EXIT_FAILURE;
 }
```

Функция получает указатель на строку с Интернет-именем сервера (например, www.unix.com или 192.168.1.16) и возвращает указатель на структуру hostent (переменная server), которая содержит имя сервера в приемлемом для дальнейшего использования виде. При этом, если необходимо, выполняется разрешение доменного имени в сетевой адрес.

Далее заполняются поля переменной serv_addr (структуры sockaddr_in) значениями адреса и порта:

```
serv_addr.sin_family = AF_INET;
strncpy((char *)&serv_addr.sin_addr.s_addr,
(char *)server->h_addr, server->h_length);
serv_addr.sin_port = htons(port);
```

Программа-клиент открывает новый сокет с помощью вызова функции socket() и вызывается функция connect(2) для установки соединения:

```
if (connect(sock_fd, &serv_addr, sizeof(serv_addr)) < 0)
 {
   printf("connect() failed: %d", errno);
   return EXIT_FAILURE;
 }
```

Теперь сокет готов к передаче и приему данных. Программа-клиент, например, может считывать символы, вводимые пользователем в окне терминала. Когда пользователь нажимает <Ввод>, программа передает данные серверу, ждет ответного сообщения сервера и распечатывает его.

По умолчанию функция socket() создает блокирующий сокет. Чтобы сделать его не-блокирующим, надо использовать функцию fcntl(2) с флагом O_NONBLOCK:

```
sock_fd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sock_fd, F_SETFL, O_NONBLOCK);
```

Теперь любой вызов функции read() для сокета sock будет возвращать управление сразу же. Если на входе сокета нет данных для чтения, функция read() вернет значение EAGAIN.

Для поверки состояния неблокирующих сокетов можно использовать мультиплексирование и воспользоваться функциями **select**(2) или poll() и их современные версии pselect и epoll.

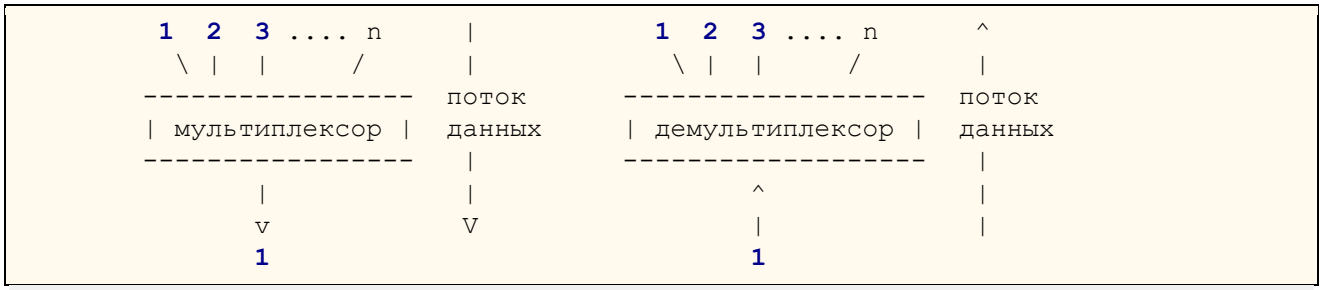


Рис.2. Мультиплексор n x 1 и демультиплексор 1 x n

Функция **select**() (см. Приложение или man pages) способна проверять состояние нескольких дескрипторов сокетов (или файлов) сразу. Первый параметр функции – количество проверяемых дескрипторов. Второй, третий и четвертый параметры функции представляют собой наборы дескрипторов, которые следует проверять, соответственно, на готовность к чтению, записи и на наличие исключительных ситуаций. Сама функция **select**() – блокирующая, она возвращает управление, если хотя бы один из проверяемых сокетов готов к выполнению соответствующей операции. В качестве последнего параметра функции **select**() можно указать интервал времени, по прошествии которого она вернет управление в любом

случае. Вызов **select**() для проверки наличия входящих данных на сокете sock может выглядеть так:

```
fd_set set;
struct timeval interval;
FD_SET(sock, &set);
tv.tv_sec = 1;
tv.tv_usec = 500000;
...
select(1, &set, NULL, NULL, &tv);
if (FD_ISSET(sock, &set) {
// Есть данные для чтения
}
```

Все, что касается функции **select**() теперь объявляется в заголовочном файле <sys/select.h> (раньше объявления элементов функции **select**() были разбросаны по файлам <sys/types.h>, <sys/time.h> и <stdlib.h>). В приведенном фрагменте кода FD_SET и FD_ISSET – макросы, предназначенные для работы с набором дескрипторов fd_set.

Лабораторная работа состоит из двух частей:

1. Организовать взаимодействие параллельных процессов на отдельном компьютере.
2. Организовать взаимодействие параллельных процессов в сети (ситуацию моделируем на одной машине).

Задание 1

- Написать приложение по модели клиент-сервер, демонстрирующее взаимодействие параллельных процессов на отдельном компьютере с использованием сокетов в файловом пространстве имен: семейство - AF_UNIX, тип - SOCK_DGRAM. При демонстрации работы программного комплекса необходимо запустить несколько клиентов (не меньше 3) на разных терминалах и продемонстрировать:

    1. Клиент посылает серверу сообщение со своим pid, сервер выводит сообщение на экран.
    2. Клиент посылает серверу сообщение с pid, сервер обрабатывает обращения каждого запущенного клиента и посылает клиенту ответное сообщение.

Задание 2

- Написать приложение по модели клиент-сервер, осуществляющее взаимодействие параллельных процессов, которые выполняются на разных компьютерах. Для взаимодействия с клиентами сервер должен использовать мультиплексирование (предпочтительно epoll). Сервер должен обслуживать запросы параллельно выполняемых клиентов. При демонстрации работы программного комплекса необходимо запустить несколько клиентов (не меньше 3) на разных терминалах и продемонстрировать, что сервер обрабатывает обращения каждого клиента, запросившего соединение и посылает клиентам ответные сообщения. Для защиты лабораторной работы необходимо представить алгоритм работы мультиплексора epoll.

Алгоритм строится по примеру, приведенному в manual page (https://man7.org/linux/man-pages/man7/epoll.7.html).

NAME top

epoll - I/O event notification facility

SYNOPSIS top

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

Пример предлагаемого использования (см. manual)
В то время как использование epoll имеет ту же семантику, что и poll(2), запускаемого с помощью edge- triggered, требует дополнительных разъяснений, чтобы избежать зависаний в цикле событий приложения. В этом примере прослушивающий сокет сервера - это неблокирующий сокет, для которого был вызван системный вызов listen(2).
Функция
do_use_fd(events[n].data.fd);
Получает в качестве параметра новый готовый файловый дескриптор до тех пор, пока EAGAIN не будет возвращен либо read(2), либо write(2).
Управляемое событиями приложение конечного автомата должно после получения EAGAIN записать свое текущее состояние, чтобы при следующем вызове do_use_fd() оно продолжило read(2) или write(2) с того места, где оно остановилось ранее.
В примере объявляются переменные:

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
```

Структура epoll_event имеет следующие поля:

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

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

union epoll_data {
    void     *ptr;
    int       fd;//file descriptor
    uint32_t  u32;
    uint64_t  u64;
};
```

Структура epoll_event определяет данные, которые ядро должно сохранить и вернуть, когда соответствующий файловый дескриптор станет готовым.

Дополнительно:

Потоки данных

Рассмотрим потоки данных, проходящие через стек протоколов, изображенный на рис.1.

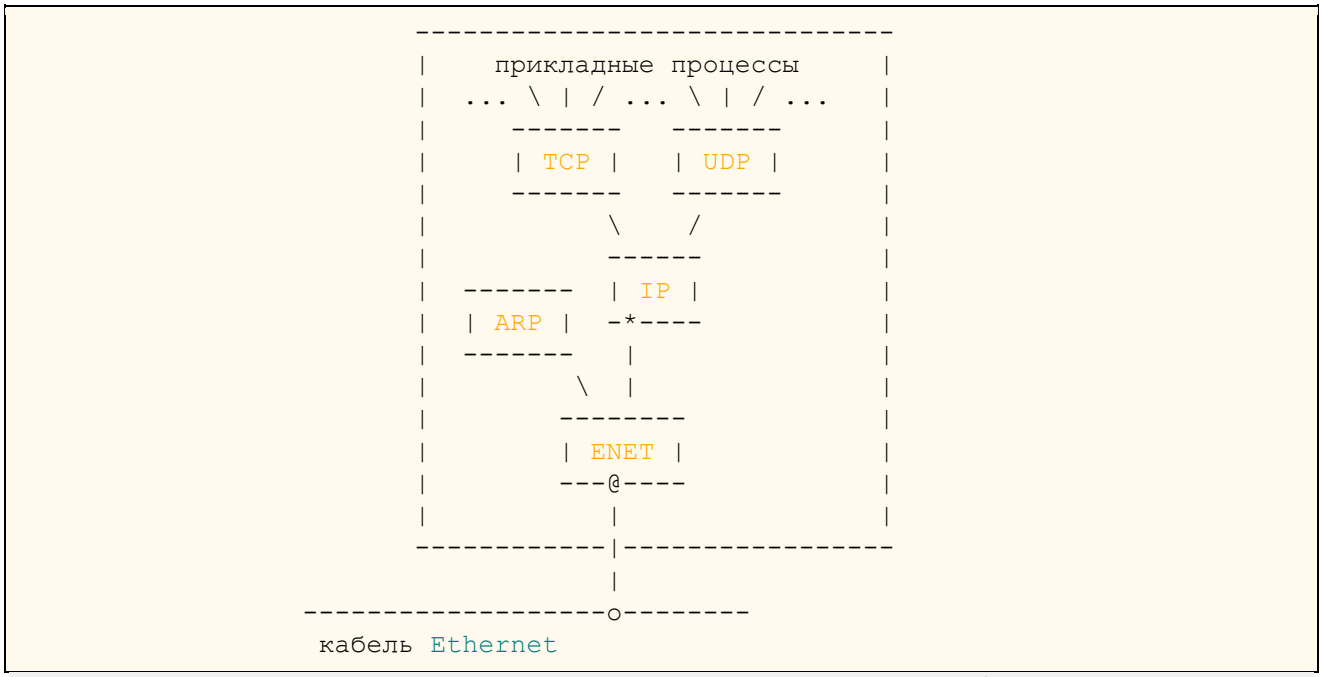


Рис.1. Структура протокольных модулей в узле сети TCP/IP

В случае использования протокола TCP (Transmission Control Protocol - протокол управления передачей), данные передаются между прикладным процессом и модулем TCP. Типичным прикладным процессом, использующим протокол TCP, является модуль FTP (File Transfer Protocol -протокол передачи файлов). Стек протоколов в этом случае будет FTP/TCP/IP/ENET. При использовании протокола UDP (User Datagram Protocol - протокол пользовательских датаграмм), данные передаются между прикладным процессом и модулем UDP. Например, SNMP (Simple Network Management Protocol - простой протокол управления сетью) пользуется транспортными услугами UDP. Его стек протоколов выглядит так: SNMP/UDP/IP/ENET.

Модули TCP, UDP и драйвер Ethernet являются мультиплексорами n x 1. Действуя как мультиплексоры, они переключают несколько входов на один выход. Они также являются демультиплексорами 1 x n. Как демультиплексоры, они переключают один вход на один из многих выходов в соответствии с полем типа в заголовке протокольного блока данных (рис.2).

Когда Ethernet-кадр попадает в драйвер сетевого интерфейса Ethernet, он может быть направлен либо в модуль ARP (Address Resolution Protocol - адресный протокол), либо в модуль IP (Internet Protocol - межсетевой протокол). На то, куда должен быть направлен Ethernet-кадр, указывает значение поля типа в заголовке кадра.

Если IP-пакет попадает в модуль IP, то содержащиеся в нем данные могут быть переданы либо модулю TCP, либо UDP, что определяется полем "протокол" в заголовке IP-пакета.

Если UDP-датаграмма попадает в модуль UDP, то на основании значения поля "порт" в заголовке датаграммы определяется прикладная программа, которой должно быть передано прикладное сообщение. Если TCP-сообщение попадает в модуль TCP, то выбор прикладной программы, которой должно быть передано сообщение, осуществляется на основе значения поля "порт" в заголовке TCP-сообщения.

Протокол UDP

Протокол UDP (User Datagram Protocol - протокол пользовательских датаграмм) является одним из двух основных протоколов, расположенных непосредственно над IP. Он предоставляет прикладным процессам транспортные услуги, которые не многим отличаются от услуг, предоставляемых протоколом IP. Протокол UDP обеспечивает ненадежную доставку датаграмм и не поддерживает соединений из конца в конец. К заголовку IP-пакета он добавляет два поля, одно из которых, поле "порт", обеспечивает мультиплексирование

информации между разными прикладными процессами, а другое поле - "контрольная сумма" - позволяет поддерживать целостность данных.

Примерами сетевых приложений, использующих UDP, являются NFS (Network File System - сетевая файловая система) и SNMP (Simple Network Management Protocol - простой протокол управления сетью).

Порты

Взаимодействие между прикладными процессами и модулем UDP осуществляется через UDP-порты. Порты нумеруются начиная с нуля. Прикладной процесс, предоставляющий некоторые услуги другим прикладным процессам (сервер), ожидает поступления сообщений в порт, специально выделенный для этих услуг. Сообщения должны содержать запросы на предоставление услуг. Они отправляются процессами-клиентами.

Например, сервер SNMP всегда ожидает поступлений сообщений в порт 161. Если клиент SNMP желает получить услугу, он посылает запрос в UDP-порт 161 на машину, где работает сервер. В каждом узле может быть только один сервер SNMP, так как существует только один UDP-порт 161. Данный номер порта является общеизвестным, то есть фиксированным номером, официально выделенным для услуг SNMP. Общеизвестные номера определяются стандартами Internet.

Данные, отправляемые прикладным процессом через модуль UDP, достигают места назначения как единое целое. Например, если процесс-отправитель производит 5 записей в UDP-порт, то процесс-получатель должен будет сделать 5 чтений. Размер каждого записанного сообщения будет совпадать с размером каждого прочитанного. Протокол UDP сохраняет границы сообщений, определяемые прикладным процессом. Он никогда не объединяет несколько сообщений в одно и не делит одно сообщение на части.

Литература:

1. Стивенс У., UNIX: Разработка сетевых приложений. - СПб.: Питер, 2004
2. W. R. Stevens, S. A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, Addison Wesley Professional, 2005
3. Сокеты. Статья из серии "Программирование для Linux", журнал Linux Format Андрей Боровский, symmetrica.net http://citforum.ru/programming/unix/sockets/
4. https://www.rsdn.org/ Программирование сокетов в Linux Автор: *Александр Шаргин*
5. RFC1180 Семейство протоколов TCP/IP https://www.lissyara.su/doc/rfc/rfc1180/

Приложение 1

Для того, чтобы посмотреть какие порты открыты и кто их открыл достаточно выполнить команду:

netstat -tulpn

Если запускать с правами не привилегированного пользователя, то узнаем только какие порты открыты, а кто их открыл узнать можно только с правами суперпользователя.

```
ekaterina@ekaterina-HP-470-G7-Notebook-PC:~/OS/OperatingSystems/lab_26/task_2$ sudo ss -tulpn
Netid    State      Recv-Q     Send-Q         Local Address:Port          Peer Address:Port
udp      UNCONN     0          0          127.0.0.53%lo:53                0.0.0.0:*         users:(("systemd-resolve",pid=813,fd=12))
udp      UNCONN     0          0                0.0.0.0:68                0.0.0.0:*         users:(("dhclient",pid=2607,fd=6))
udp      UNCONN     0          0                0.0.0.0:631               0.0.0.0:*         users:(("cups-browsed",pid=1059,fd=7))
udp      UNCONN     0          0                0.0.0.0:51555             0.0.0.0:*         users:(("firefox",pid=3041,fd=293))
udp      UNCONN     0          0                0.0.0.0:43739             0.0.0.0:*         users:(("avahi-daemon",pid=1039,fd=14))
udp      UNCONN     0          0                0.0.0.0:44673             0.0.0.0:*         users:(("firefox",pid=3041,fd=321))
udp      UNCONN     0          0                0.0.0.0:5353              0.0.0.0:*         users:(("avahi-daemon",pid=1039,fd=12))
udp      UNCONN     0          0                0.0.0.0:47254             0.0.0.0:*         users:(("firefox",pid=3041,fd=256))
udp      UNCONN     0          0                   [::]:34007                [::]:*         users:(("avahi-daemon",pid=1039,fd=15))
udp      UNCONN     0          0                   [::]:5353                 [::]:*         users:(("avahi-daemon",pid=1039,fd=13))
tcp      LISTEN     0          128         127.0.0.53%lo:53                0.0.0.0:*         users:(("systemd-resolve",pid=813,fd=13))
tcp      LISTEN     0          5              127.0.0.1:631                0.0.0.0:*         users:(("cupsd",pid=962,fd=7))
tcp      LISTEN     0          5                0.0.0.0:2000              0.0.0.0:*         users:(("server.o",pid=4667,fd=3))
tcp      LISTEN     0          5                  [::1]:631                  [::]:*         users:(("cupsd",pid=962,fd=6))
```

sudo ss -tulpn

```
parallels@parallels-Parallels-Virtual-Platform:~/OSBMSTU/sock$ netstat -at
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:domain        0.0.0.0:*               LISTEN
tcp        0      0 localhost:ipp           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:9999            0.0.0.0:*               LISTEN
tcp        1      0 10.211.55.6:44924       192.168.0.1:netbios-ssn CLOSE_WAIT
tcp        0      0 10.211.55.6:37960       ec2-52-35-57-239.:https ESTABLISHED
tcp        0      0 10.211.55.6:33790       ec2-34-211-62-63.:https ESTABLISHED
tcp6       0      0 ip6-localhost:ipp       [::]:*                  LISTEN
```

netstat -at

/* According to POSIX.1-2001, POSIX.1-2008 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

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

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*);

#include <sys/select.h>

int pselect(int *nfds*, **fd_set ***readfds*, **fd_set ***writefds*,
**fd_set ***exceptfds*, **const struct timespec ***timeout*,
**const sigset_t ***sigmask*);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

pselect(): _POSIX_C_SOURCE >= 200112L

Описание

select() and **pselect**()позволяет программе контролировать несколько файловых дескрипторов, ожидающих, пока один или несколько файловых дескрипторов не станут доступны (готовы) для некоторого класса операций ввода / вывода (например, возможен ввод). Дескриптор файла считается готовым, если возможно выполнить соответствующую операцию ввода / вывода (e.g., read(2), or a sufficiently small write(2)) without blocking.

select() может опрашивать колочество файловых дескрипторов, которое должно быть меньше чем FD_SETSIZE; poll(2) такого ограничения не имеет

Функции select() and pselect() почти идентичны, но имеют три различия:

- select() использует timeout представленный a struct timeval (with seconds and microseconds), в то время как pselect(использует a struct timespec (with seconds and nanoseconds).
- **select**() может обновить аргумент *timeout* , чтобы указать сколько времени осталось pselect() не изменяет этот аргумент
- select() не имеет аргумент sigmask, и ведет себя как pselect() вызов с NULL sigmask.

Причина того, что pselect() необходим, заключается в том, что если какой-то процесс ждет либо сигнал или файловый дескриптор, чтобы стать готовым, то атомарная проверка необходима для предотвращения гонки (race).
Предположим, что обработчик signal устанавливает глобальный флаг и возвращает. Тогда проверка этого глобального флага с последующим вызовом select() может зависать бесконечно, если сигнал поступил сразу после проверки, но перед вызовом. Напротив, pselect () позволяет сначала блокировать сигналы, обрабатывать поступившие сигналы, затем вызвать pselect () с нужным значением маски, избегая гонки (race).

The **timeout**

```
Временные структуры определены в <sys/time.h>
struct timeval
{
   long    tv_sec;        /* seconds */
   long    tv_usec;       /* microseconds */
};
и  struct timespec
{
   long    tv_sec;        /* seconds */
   long    tv_nsec;       /* nanoseconds */
};
```

Возвращаемое значение

В случае успеха select () и pselect () возвращают номер файлового дескриптора, содержащиеся в трех возвращенных наборах дескрипторов (то есть общее количество битов, которые установлены в readfds, writefds, исключением из fds), которое может быть нулевым, если тайм-аут истекает раньше, то возможны проблемы.

При ошибке возвращается -1, a errno определяет ошибку; наборы дескрипторов файлов не изменены, и тайм-аут становится неопределенным.

Ошибки

EBADF - в одном из наборов указан неверный дескриптор файла (возможно, дескриптор файла, который уже был закрыт, или дескриптор, в котором произошла ошибка). Однако см. BUGS.

EINTR сигнал был перехвачен; смотри signal(7).

EINVAL nfds is negative or exceeds the RLIMIT_NOFILE resource limit
(see getrlimit(2)).

EINVAL Значение, содержащееся в тайм-ауте, недействительно.

ENOMEM невозможно выделить память для внутренних таблиц.

Версии

pselect() был добавлен в Linux в ядро 2.6.16. (Prior to this, pselect() was emulated in glibc (but see BUGS)).

- select () соответствует POSIX.1-2001, POSIX.1-2008 и 4.4BSD (select () впервые появился в 4.2BSD). Обычно переносимы в / из не-BSD систем, поддерживающих клоны уровня сокетов BSD (включая варианты System V). Однако обратите внимание, что вариант System V обычно устанавливает переменную тайм-аута перед выходом, а вариант BSD - нет.

- pselect() определен в POSIX.1g, и в POSIX.1-2001 и POSIX.1-2008.

Замечания

fd_set - это буфер фиксированного размера. Выполнение FD_CLR () или FD_SET () со значением fd, которое является отрицательным или равно или больше, чем FD_SETSIZE приведет к неопределенному поведению. Кроме того, POSIX требует, чтобы fd был допустимым дескриптором файла.

Операции select () и pselect () не зависят от флага O_NONBLOCK.

В некоторых других системах UNIX select () может завершиться с ошибкой EAGAIN. Если система не может выделить внутренние ресурсы ядра скорее, чем ENOMEM, как в Linux, POSIX указывает эту ошибку для poll (2), но не для select(). Переносимые программы могут проверяться на EAGAIN и цикл так же, как с EINTR.

Что касается задействованных типов, классическая ситуация такова, что два поля структуры timeval определяются как длинные (как показано выше), и структура определена в <sys / time.h>. В POSIX.1 имеет вид:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

Структура определена в <sys/select.h>, а типы данных time_t и suseconds_t определены в <sys/types.h>.

Что касается прототипов, классическая ситуация такова, что нужно включить <time.h> для select (). Ситуация с POSIX.1 заключается в том, что нужно включить <sys / select.h> для select () и pselect ().

В glibc 2.0 <sys / select.h> дает неправильный прототип для pselect (). В glibc 2.1 до 2.2.1 он выдает pselect (), когда определено _GNU_SOURCE. Начиная с glibc 2.2.2, требования соответствуют описанию, приведенному в ОПИСАНИИ.

Мультипоточные приложения

Если дескриптор файла, отслеживаемый select (), закрывается в другом потоке, результат не определен. В некоторых системах UNIX select () разблокирует приложение и возвращает с указанием того, что дескриптор файла готов (последующая операция ввода-вывода, скорее всего, завершится с ошибкой, если другой процесс повторно открывает дескриптор файла между временем возврата из select () и выполнением операции ввода / вывода). В Linux (и некоторых других системах), закрытие дескриптора файла в другом потоке не влияет на select (). Таким образом, любое приложение, которое полагается на определенное поведение в этом сценарии следует считать ошибочным.

C library/kernel differences the Linux kernel allows file descriptor sets of arbitrary size, determining the length of the sets to be checked from the value of nfds. However, in the glibc implementation, the fd_set type is fixed in size. See also BUGS.
The pselect() interface described in this page is implemented by glibc. The underlying Linux system call is named pselect6(). This system call has somewhat different behavior from the glibc wrapper function.
The Linux pselect6() system call modifies its timeout argument. However, the glibc wrapper function hides this behavior by using a local variable for the timeout argument that is passed to the system call.Thus, the glibc pselect() function does not modify its timeout argument; this is the behavior required by POSIX.1-2001.

The final argument of the pselect6() system call is not a sigset_t * pointer, but is instead a structure of the form:

```
struct {
    const kernel_sigset_t *ss;   /* Pointer to signal set */
    size_t ss_len;               /* Size (in bytes) of object
                                    pointed to by 'ss' */
};
```

Это позволяет системному вызову получить указатель на набор сигналов и его размер, в то же время учитывая тот факт, что большинство архитектур поддерживают максимум 6 аргументов для системного вызова. См. Sigprocmask (2) для обсуждения различий между ядром и представлением libc набора сигналов.

Ошибки (BUGS)

POSIX позволяет реализации определять верхний предел, объявленный через константу FD_SETSIZE, для диапазона файловых дескрипторов, которые могут быть указаны в наборе файловых дескрипторов. Ядро Linux не накладывает фиксированного ограничения, но реализация glibc делает fd_set типом фиксированного размера с FD_SETSIZE, определяемым как 1024, и макросами FD _ * (), работающими в соответствии с этим пределом. Чтобы отслеживать файловые дескрипторы больше 1023, используйте poll (2).

Реализация аргументов fd_set в качестве аргументов значение-результат означает, что они должны быть повторно инициализированы при каждом вызове select (). Эта ошибка проектирования предотвращается poll(2), который использует отдельные поля структуры для ввода и вывода вызова.

Согласно POSIX, select () должен проверять все указанные файловые дескрипторы в трех наборах файловых дескрипторов, вплоть до предела nfds-1. Однако текущая реализация игнорирует любой файловый дескриптор в этих наборах, который больше, чем максимальный номер файлового дескриптора, который процесс в настоящее время имеет открытым. Согласно POSIX, любой такой файловый дескриптор, указанный в одном из наборов, должен привести к ошибке EBADF.

Glibc 2.0 предоставил версию pselect (), которая не принимала аргумент sigmask.

Начиная с версии 2.1, glibc предоставил эмуляцию pselect (), которая была реализована с использованием sigprocmask (2) и select (). Эта реализация оставалась уязвимой для состояния «гонок», для предотвращения которого была разработана функция pselect (). Современные версии glibc используют (без гонки) системный вызов pselect () в тех ядрах, где он предоставляется.

В Linux select () может сообщить, что дескриптор файла сокета готов для чтения, но, тем не менее, последующее чтение блокируется. Это может, например, произойти, когда данные поступили, но при проверке они имеют неверную контрольную сумму и отбрасываются. Могут быть другие обстоятельства, при которых файловый дескриптор ложно сообщается как готовый. Таким образом, может быть безопаснее использовать O_NONBLOCK на сокетах, которые не должны блокироваться.

В Linux select () также изменяет время ожидания, если вызов прерывается обработчиком сигнала (то есть возвращением ошибки EINTR). Это не разрешено POSIX.1. Системный вызов Linux pselect () ведет себя также, но оболочка glibc скрывает это поведение, копируя тайм-аут в локальную переменную и передавая эту переменную системному вызову.

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;

   /* Watch stdin (fd 0) to see when it has input. */

   FD_ZERO(&rfds);
   FD_SET(0, &rfds);

   /* Wait up to five seconds. */

   tv.tv_sec = 5;
   tv.tv_usec = 0;

   retval = select(1, &rfds, NULL, NULL, &tv);
   /* Don't rely on the value of tv now! */

   if (retval == -1)
      perror("select()");
   else if (retval)
      printf("Data is available now.\n");
      /* FD_ISSET(0, &rfds) will be true. */
   else
      printf("No data within five seconds.\n");

   exit(EXIT_SUCCESS);
}
```

Приложение 2

НАЗВАНИЕ

socket - создать оконечную точку коммуникации

КРАТКАЯ СВОДКА

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

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

ОПИСАНИЕ

socket создает оконечную точку для коммуникации и возвращает её дескриптор.

Параметр *domain* задает "домен" коммуникации; выбирает набор протоколов, которые будут использоваться для коммуникации. Такие наборы описаны в **<sys/socket.h>**.

В настоящее время существуют такие форматы:

Название	Назначение	Страница
PF_UNIX,PF_LOCAL	Локальная коммуникация	unix(7)
PF_INET	IPv4, протоколы Интернет	ip(7)
PF_INET6	IPv6, протоколы Интернет	
PF_IPX	IPX - протоколы Novell	
PF_NETLINK	Устройство для общения пользователя с ядром	netlink(7)
PF_X25	Протокол ITU-T X.25 / ISO-8208	x25(7)
PF_AX25	Протокол AX.25, любительское радио	
PF_ATMPVC	ATM -- доступ к низкоуровневым PVC	
PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Низкоуровневый пакетный интерфейс	packet(7)

Сокет имеет указанный тип, *type*, задающий семантику коммуникации. В настоящее время определены следующие типы:

SOCK_STREAM
Обеспечивает надежные, двунаправленные последовательные потоки байтов, с поддержкой соединений. Может также поддерживаться механизм вне-поточных данных.

SOCK_DGRAM
Обеспечивает датаграммы (ненадежные сообщения с ограниченной максимальной длиной, без поддержки соединения).

SOCK_SEQPACKET
Обеспечивает последовательный двунаправленный канал передачи датаграмм с поддержкой соединений; датаграммы имеют ограниченную

максимальную длину; от получателя требуется за один раз прочитать целый пакет.

SOCK_RAW
Обеспечивает доступ к низкоуровневому сетевому протоколу.

SOCK_RDM
Обеспечивает надежную доставку датаграмм без гарантии их последовательности.

SOCK_PACKET
Устарело и не должно использоваться в новых программах; см. **packet**(7).

Некоторые типы сокетов могут не быть реализованными в некоторых наборах протоколов; например, **SOCK_SEQPACKET** не реализовано в наборе **AF_INET**.

Параметр *protocol* задает конкретный протокол, который используется на сокете. Обычно существует только один протокол, обеспечивающий конкретный тип сокета в заданном наборе протоколов. Однако, возможно существование нескольких таких протоколов -- тогда и используется этот параметр. Номер протокола зависит от используемого ``домена коммуникации'', см.~ **protocols**(5). См. **getprotoent**(3), где описано, как сопоставлять имена протоколов их номерам.

Сокеты типа **SOCK_STREAM** являются дуплексными потоками байт, похожими на трубы. Они не сохраняют границы между записями. Потоковый сокет должен быть в *соединённом* состоянии перед тем, как по нему можно отсылать и принимать данные. Соединение с другим сокетом создается с помощью системного вызова **connect**(2). После соединения данные можно передавать, используя системные вызовы **read**(2) и **write**(2), или какой-то из вариантов системных вызовов **send**(2) и **recv**(2). Когда сессия закончена, выполняется **close**(2). Вне-поточные данные могут передаваться, как описано в **send**(2), а приниматься, как описано в **recv**(2).

Коммуникационные протоколы, которые реализуют **SOCK_STREAM**, следят, чтобы данные не были потеряны или продублированы. Если у корреспондента имеется место в буфере, но очередная порция данных не может быть передана за разумное время, то соединение считается мертвым. Когда на сокете включен флаг **SO_KEEPALIVE**, протокол каким-либо способом проверяет, что другая сторона ещё жива. Сигнал **SIGPIPE** появляется, если процесс посылает или принимает данные, пользуясь разорванным потоком; это приводит к тому, что неопытные процессы, не обрабатывающие сигнал, завершаются.
Сокеты **SOCK_SEQPACKET** используют те же самые системные вызовы, что и сокеты **SOCK_STREAM**. Единственное отличие в том, что вызовы **read**(2) вернут только запрошенное количество данных, а остаток прибывшего пакета будет отброшен. Границы между сообщениями во входящих датаграммах сохраняются.

Сокеты **SOCK_DGRAM** и **SOCK_RAW** позволяют посылать датаграммы корреспондентам, заданным при вызове **send**(2). Датаграммы обычно принимаются с помощью вызова **recvfrom**(2), который возвращает следующую датаграмму с соответствующим обратным адресом.

SOCK_PACKET --- это устаревший тип сокета, позволявший получать необработанные пакеты прямо от драйвера устройства. Используйте вместо него **packet**(7).

Системный вызов fcntl(2) с аргументом **F_SETOWN** может использоваться, чтобы задать группу процессов, которая будет получать сигнал **SIGURG**, когда прибывают вне-поточные данные или сигнал **SIGPIPE**, когда соединение типа **SOCK_STREAM** неожиданно обрывается. Этот вызов также можно использовать, чтобы задать процесс или группу процессов, которые получают асинхронные уведомления о вводе-выводе с помощью **SIGIO.** Использование **F_SETOWN** эквивалентно использованию ioctl(2) с аргументом SIOSETOWN.

Когда сеть сообщает протоколу об ошибке (в случае IP, например, используя ICMP-сообщение), то для сокета устанавливается флаг ожидающей ошибки. Следующая операция с этим сокетом вернет код ожидающей ошибки. Для некоторых протоколов можно разрешить для конкретного сокета очередь ошибок, чтобы получить детальную информацию об ошибке; см. **IP_RECVERR** в ip(7)**.**

Операции сокетов контролируются их *параметрами*. Эти параметры описаны в **<sys/socket.h>**. setsockopt(2) и getsockopt(2) используются, чтобы установить и получить параметры, соответственно.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае ошибки возвращается -1; в противном случае возвращается дескриптор, ссылающийся на сокет.

ОШИБКИ

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

ENFILE
: Ядру не хватило памяти, чтобы создать новый сокет.

EMFILE
: Переполнение таблицы файлов процесса.

EACCES
: Не разрешено создание сокета указанного типа и/или протокола.

ENOBUFS или **ENOMEM**
: Недостаточно памяти. Сокет не может быть создан, пока не освободится память.

EINVAL
: Неизвестный протокол, или недоступный набор протоколов.

Другие ошибки могут быть сгенерированы нижележащими модулями протоколов.

СООТВЕТСТВИЕ СТАНДАРТАМ

4.4BSD (системный вызов **socket** появился в 4.2BSD). Обычно переносимо с/на не-BSD системы, имеющие реализацию сокетов BSD (включая варианты System V).

ЗАМЕЧАНИЕ

Для наборов протоколов под BSD 4.* используются константы PF_UNIX, PF_INET и т.~д., тогда как AF_UNIX и т.~п. используются для указания семьи адресов. Однако

же, страница руководства из BSD обещает: "Вообще, набор протоколов совпадает с семьей адресов", и в последующих стандартах везде используется AF_*.

ИЛИ

Семейства описаны в *<sys/socket.h>*. В настоящее время распознаются такие форматы:

Название	Назначение	Справочная страница
AF_UNIX, **AF_LOCAL**	Локальное соединение	**unix**(7)
AF_INET	Протоколы Интернет IPv4	**ip**(7)
AF_INET6	Протоколы Интернет IPv6	**ipv6**(7)
AF_IPX	Протоколы Novell IPX	
AF_NETLINK	Устройство для взаимодействия с ядром	**netlink**(7)
AF_X25	Протокол ITU-T X.25/ISO-8208	**x25**(7)
AF_AX25	Протокол любительского радио AX.25	
AF_ATMPVC	Доступ к низкоуровневым PVC в ATM	
AF_APPLETALK	AppleTalk	**ddp**(7)
AF_PACKET	Низкоуровневый пакетный интерфейс	**packet**(7)
AF_ALG	Интерфейс к ядерному крипто-API	

Сокет имеет тип *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 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; см. **packet**(7).

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

Начиная с Linux 2.6.27, аргумент *type* предназначается для двух вещей: кроме определения типа сокета, для изменения поведения **socket**() он может содержать по битово сложенные любые следующие значения:

SOCK_NONBLOCK
Устанавливает флаг состояния файла **O_NONBLOCK** для нового открытого файлового дескриптора. Использование данного флага заменяет дополнительные вызовы **fcntl**(2) для достижения того же результата.

SOCK_CLOEXEC
Устанавливает флаг close-on-exec (**FD_CLOEXEC**) для нового открытого файлового дескриптора. Смотрите описание флага **O_CLOEXEC** в **open**(2) для того, чтобы узнать больше.

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

Это важно, поэтому еще раз:

It is possible to do nonblocking I/O on sockets by setting the **O_NONBLOCK** flag on a socket file descriptor using fcntl(2). Then all operations that would block will (usually) return with **EAGAIN** (operation should be retried later); connect(2) will return **EINPROGRESS** error. The user can then wait for various events via poll(2) or select(2).

I/O events		
Event	Poll flag	Occurrence
Read	POLLIN	New data arrived.
Read	POLLIN	A connection setup has been completed (for connection-oriented sockets)
Read	POLLHUP	A disconnection request has been initiated by the other end.
Read	POLLHUP	A connection is broken (only for connection-oriented protocols). When the socket is written **SIGPIPE** is also sent.
Write	POLLOUT	Socket has enough send buffer space for writing new data.

Read/Write	POLLIN | POLLOUT	An outgoing `connect(2)` finished.
Read/Write	POLLERR	An asynchronous error occurred.
Read/Write	POLLHUP	The other end has shut down one direction.
Exception	POLLPRI	Urgent data arrived. **`SIGURG`** is sent then.

An alternative to poll(2) and select(2) is to let the kernel inform the application about events via a **SIGIO** signal. For that the **O_ASYNC** flag must be set on a socket file descriptor via fcntl(2) and a valid signal handler for **SIGIO** must be installed via sigaction(2). See the ***Signals*** discussion below.