Содержание

1 Сокеты

В данном разделе мы рассмотрим ещё один механизм межпроцессного взаимодействия — *сокеты*. Сокеты — это универсальный интерфейс межпроцессного взаимодействия, допускающий взаимодействие процессов, работающих как на одном компьютере, так и на разных компьютерах, объединённых в сеть. Изначально интерфейс сокетов был реализован для 4.2BSD, затем был перенесён на все другие Unix-подобные операционные системы. Значительно позднее похожий интерфейс был реализован для Windows 3.1. Несмотря на то, что функции работы с сокетами называются в Windows точно также, работа с ними в деталях отличается от работы с сокетами в Unix, поэтому говорить о совместимости не приходится.

Мы рассмотрим работу с сокетами только в случае сети компьютеров и с установлением виртуального соединения. Обмен данными в этом случае ведётся по протоколу TCP.

1.1 Адресация компьютеров

Каждый компьютер в сети IP должен иметь уникальный номер (адрес). В настоящее время (протокол IPv4) адрес имеет размер 32 бита (4 байта)[1]. Традиционно IP-адреса записываются в точечной нотации: 4 десятичных числа, каждое для соответствующего байта адреса, начиная от старшего байта к младшему, разделённые точкой, например `192.168.10.1`.

Чтобы преобразовать адрес в точечной нотации в одно 32-битное слово можно использовать функцию `gethostbyaddr`.

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyaddr(char const *addr, int len, int type);
```

Здесь параметр `addr` — это адрес, `len` — длина строки адреса, `type` — тип адреса, этот параметр должен быть установлен в `AF_INET`. В случае неудачи возвращается `NULL`, а в случае успеха возвращается указатель на заполненную структуру `hostent`.

[1]В новой версии протокола IP — IPv6 — адрес имеет размер 128 бит. Впрочем, переход на новый протокол только начинается.

```
struct hostent {
        char    *h_name;        /* официальное имя */
        char    **h_aliases;    /* список алиасов имени */
        int     h_addrtype;     /* тип адреса (AF_INET) */
        int     h_length;       /* длина адреса (4) */
        char    **h_addr_list;  /* список адресов */
};
```

Указатели на адреса во внутреннем представлении находятся в массиве h_addr_list. Последний элемент массива содержит NULL. Чтобы сделать описание структуры максимально общим, разработчики сделали указатель на внутреннее представление адреса указателем типа char*, что здесь по смыслу то же самое, что void*[2]. Чтобы использовать адреса нужно либо приводить тип указателя к типу unsigned long *, либо использовать функции типа memcpy.

Поскольку запись адресов в десятичной нотации немнемонична, была разработана доменная система имён. Имя компьютера в доменной системе имён записывается как последовательность из нескольких слов, разделённых точкой, например www.cs.msu.su. В локальных сетях могут использоваться имена, состоящие только из одного слова, например elwing. Чтобы преобразовать доменное имя компьютера в его адрес во внутреннем представлении можно использовать функцию gethostbyname.

```
#include <netdb.h>
struct hostent *gethostbyname(char const *name);
```

Если вызов закончился неудачно, функция возвращает NULL, а в случае удачи возвращается указатель на структуру hostent, описанную выше. В Unix gethostbyname корректно обрабатывает адреса компьютеров в точечной нотации, в отличие от Windows.

IP-адрес 127.0.0.1 на любом компьютере обозначает этот же компьютер. Этому адресу соответствует имя localhost.

Иногда требуется по адресу во внутреннем представлении получить адрес в точечной нотации. Для этого используется функция inet_ntoa.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

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

По адресу во внутреннем представлении, передаваемом в параметре in, эта функция возвращает указатель на строку, содержащую запись адреса в точечной нотации. Строка размещается в статической области памяти, поэтому содержимое строки переписывается при каждом вызове функции. Структура struct in_addr определена следующим образом.

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

Поле s_addr содержит адрес компьютера во внутреннем представлении (32-битное целое число в сетевом представлении).

Кроме адреса компьютера для установления связи необходимо знать номер *порта*. Это — небольшое (16 бит) неотрицательное целое число. Именно по номеру порта различаются

[2] Когда разрабатывался интерфейс сокетов, в языке Си ещё не было ключевого слова void.

разные сетевые сервисы, работающие на одном компьютере. Например, порт с номером 80 обычно используется web-сервером, а порт с номером 25 — почтовым сервером. Порты с номерами, меньшими 1025 зарезервированы для использования привилегированными процессами. Все остальные порты доступны для использования всеми процессами. Если процесс не выполнил операцию привязки сокета к определённому номеру порта, номер порта назначается автоматически. К одному и тому же порту могут подключаться несколько клиентов, поэтому соединение полностью идентифицируется 4 числами: двумя адресами компьютеров и двумя номерами портов.

1.2 Сетевое представление чисел

В сети могут работать компьютеры различных аппаратных платформ под управлением различных операционных систем. Даже целые числа хранятся в памяти по-разному у разных типов процессоров. Например, у процессоров i386 и выше целые числа хранятся в памяти от младшего байта к старшему (Little Endian). У других процессоров (например, sparc) целые числа хранятся в памяти от старшего байта к младшему (Big Endian). Поэтому необходимо единое согласованное представление целых чисел при пересылке пакетов по сети[3]. В качестве такого представления выбрано представление от старшего байта к младшему. Для перевода целых чисел из сетевого представления в локальное и наоборот определены следующие функции.

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Функция `htonl` переводит длинное целое (32 бита) из локального представления в сетевое представление. Функция `ntohl` переводит длинное целое (32 бита) из сетевого представления в локальное. Функция `htons` переводит короткое целое (16 бит) из локального представления в сетевое, а функция `ntohs` — наоборот. Когда по сети передаются целые числа в двоичном представлении, они должны быть преобразованы в сетевое представление.

Заметьте, что не существует стандартных способов передачи по сети вещественных чисел в бинарном формате. Предпочтительнее их преобразовывать в символьное представление.

1.3 Связь с установлением соединения

Рассмотрим взаимодействие двух процессов, когда они связываются с установлением виртуального соединения. В этом случае работа двух сторон не симметрична. Один процесс должен выступить в роли «сервера», другой — в роли «клиента». Такое разделение может не иметь ничего общего с архитектурой «клиент-сервер», но чаще всего роли сторон в архитектуре «клиент-сервер» естественно отображаются на роли сторон при установлении соединения.

Сервер должен создать сокет, привязать его к определённому порту и/или адресу, переключить сокет в режим ожидания соединения, затем ожидать подключения от клиента. Клиент должен создать сокет, после этого он может привязать его к определённому порту и/или

[3] И вообще, необходимо согласованное представление любых пересылаемых данных. Для этого предназначены такие языки, как ASN.1, IDL и пр.

адресу, затем клиент должен подключиться к серверу. После установления соединения обе стороны могут обмениваться информацией друг с другом.

1.3.1 Создание сокета

Рассмотрим эти шаги подробнее. Чтобы создать сокет используется системный вызов `socket`.

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

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

Параметр `domain` задаёт коммуникационный домен. Например, `PF_UNIX`, `PF_LOCAL` — это локальное соединение, которое доступно только для процессов, работающих на одном компьютере, но его мы здесь рассматривать не будем. Нас интересует домен `PF_INET` — протоколы, основанные на **IPv4**. Параметр `type` задаёт тип соединения. Нас интересует тип соединения `SOCK_STREAM` — связь с установлением двунаправленного виртуального соединения с надежной доставкой (протокол **TCP**). Возможен тип соединения `SOCK_DGRAM` — связь без установления соединения (протокол **UDP**). Параметр `protocol` позволяет выбрать один из возможно нескольких протоколов, удовлетворяющих двум первым параметрам. Если здесь передаётся 0, протокол будет выбран автоматически, что рекомендуется в данном случае.

Системный вызов `socket` возвращает номер файлового дескриптора, ассоциированного с новым сокетом. С ним можно выполнять все операции, допустимые для файловых дескрипторов: `dup`, `dup2`, `close`. В случае ошибки системный вызов `socket` возвращает `-1`, а переменная `errno` содержит код ошибки.

Системный вызов `bind` позволяет привязать сокет к определённому адресу и/или порту.

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

int  bind(int fd, struct sockaddr *paddr, int addrlen);
```

Параметр `fd` — это номер файлового дескриптора сокета. Параметр `paddr` — указатель на структуру, которая содержит информацию о привязке сокета. Содержимое структуры существенно зависит от коммуникационного домена, задаваемого в системном вызове `socket`. Так, для коммуникационного домена `PF_INET` должна использоваться структура `struct sockaddr_in`, определённая следующим образом:

```
struct sockaddr_in {
        int             sin_family;
        unsigned short  sin_port;
        struct in_addr  sin_addr;
};
```

Поле `sin_family` задаёт тип адреса, для коммуникационного домена `PF_INET` тип адреса должен быть установлен в `AF_INET`. Поле `sin_port` определяет номер порта. Номер порта должен задаваться в сетевом представлении. Если в качестве номера порта указан 0, ядро назначит номер порта автоматически. Поле `sin_addr` задаёт адрес привязки сокета. Допустима константа `INADDR_ANY`, обозначающая, что адрес привязки будет назначен автоматически.

В случае успешного выполнения системный вызов возвращает 0, а при ошибке — `-1`. Переменная `errno` устанавливается в код ошибки. Возможные ошибки приведены в таблице ниже.

`EBADF`	`fd` не является файловым дескриптором.
`EINVAL`	Сокет уже привязан к некоторому адресу.
`EACCES`	Запрошен адрес или порт, который может использоваться только привилегированным процессом.
`ENOTSOCK`	Файловый дескриптор не ассоциирован с сокетом.

1.3.2 Работа клиента

Дальнейшее использование сокетов отличается для сервера и для клиента. Клиент должен выполнить вызов функции `connect`.

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

int connect(int fd, const struct sockaddr *paddr, int addrlen);
```

Параметр `fd` задаёт файловый дескриптор сокета. `paddr` — адрес структуры, определяющей адрес подсоединения, `addrlen` задаёт размер структуры `paddr`. В случае успешного завершения функция возвращает 0, а при ошибке возвращается `-1`. Переменная `errno` устанавливается в код ошибки. Возможные коды ошибок приведены ниже.

`EBADF`	Неверный файловый дескриптор `fd`.
`EFAULT`	Адрес `paddr` выходит за пределы адресного пространства процесса.
`ENOTSOCK`	Указанный файловый дескриптор не ассоциирован с сокетом.
`EISCONN`	Соединение уже установлено.
`ECONNREFUSED`	Никакой процесс не прослушивает указанный адрес и порт.
`ETIMEDOUT`	Тайм-аут при подключении. Возможно сервер слишком загружен.
`ENETUNREACH`	Сеть недоступна.
`EADDRINUSE`	Локальный адрес уже используется.
`EINPROGRESS`	Сокет работает в неблокирующем режиме, а соединение не может быть установлено немедленно.
`EALREADY`	Сокет работает в неблокирующем режиме, и предыдущая попытка соединения ещё не закончилась.
`EAGAIN`	Недостаточно ресурсов ядра, чтобы выполнить запрос.
`EAFNOSUPPORT`	Передан неверный тип адреса.
`EACCES`, `EPERM`	Недостаточно прав, чтобы выполнить операцию.

Если вызов `connect` завершился успешно, это значит, что соединение установлено, и можно передавать данные. Один и тот же файловый дескриптор сокета можно использовать и для передачи данных, и для приёма данных. Работа с ним аналогична работе с каналом. Данные рассматриваются как поток байтов, причём границы сообщений не сохраняются.

1.3.3 Работа сервера

Сервер сначала должен переключить сокет в режим *прослушивания*. Для этого используется системный вызов `listen`.

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

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

Параметр `fd` задаёт номер файлового дескриптора сокета. `backlog` — максимальный размер очереди запросов на подключение. Когда очередь переполняется, клиент, который попытается выполнить подключение, получит ошибку `ECONNREFUSED`. Обычное значение для параметра `backlog` — 5.

Системный вызов возвращает `0` при успешном завершении, `-1` — при ошибке. В этом случае переменная `errno` содержит код ошибки.

Для ожидания подключения сервер должен выполнить функцию `accept`.

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

int accept(int fd, struct sockaddr *paddr, int *paddrlen);
```

Здесь `fd` — это файловый дескриптор сокета, который должен быть уже переведён в режим прослушивания. Функция `accept` приостанавливает выполнение процесса до тех пор, пока от какого-либо клиента не поступит запрос на подключение. Тогда в структуру, на которую указывает `paddr`, будет записан адрес компьютера и номер порта, с которого производится подключение, а в переменную по адресу `paddrlen` будет записан размер структуры, содержащей информацию о подключении. При вызове функции эта переменная уже должна содержать размер структуры. При работе с коммуникационным доменом `PF_INET` параметр `paddr` должен указывать на структуру типа `struct sockaddr_in`.

В случае успешного выполнения функция возвращает новый файловый дескриптор сокета. Этот файловый дескриптор уже не находится в состоянии прослушивания и должен использоваться для обмена данными с клиентами. Исходный файловый дескриптор `fd` остаётся в состоянии прослушивания и может использоваться для ожидания подключения других клиентов. При ошибке возвращается `-1`, переменная `errno` устанавливается в код ошибки.

1.3.4 Обмен данными

Работа с виртуальным каналом полностью аналогична работе с обычным каналом. Для чтения из канала используется системный вызов `read`, для записи в канал используется системный вызов `write`. Вызов `read` приостановит процесс до появления данных от другой стороны соединения, а вызов `write` может приостановить выполнение процесса, если у ядра полностью заполнилась очередь на передачу. В простейшей ситуации (соединение в локальной сети, небольшой объём данных) можно предполагать, что `write` не приостанавливает процесс. Когда соединение закрывается, системный вызов `read` возвращает 0. Для закрытия сокета, когда он становится ненужным, можно использовать системный вызов `close`.

Как обычно можно создать дескриптор потока по файловому дескриптору сокета с помощью функции `fdopen`, а затем использовать все средства высокоуровневого ввода-вывода. Если высокоуровневые средства используются и для записи в сокет, и для чтения из сокета, необходимо открыть два разных дескриптора потока, один для записи, другой для чтения, которым при создании передать два разных файловых дескриптора, получив второй дескриптор из файлового дескриптора сокета с помощью `dup`.

1.3.5 Установка режимов работы сокета

С помощью функции `setsockopt` можно установить режимы работы сокета.

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

int setsockopt(int fd, int level, int optname,
               const void *optval, int optlen);
```

fd задаёт файловый дескриптор сокета. Параметр level определяет, режимы какого уровня необходимо изменить. Возможно изменить режимы работы как самого сокета (константа SOL_SOCKET), так и протокола, используемого для обмена данными, в этом случае нужно указать номер протокола (например, IPPROTO_TCP). Параметр optname задаёт имя изменяемого параметра, а optval — новое значение. Параметр optlen — размер области памяти, на которую указывает optval.

Из всех возможных режимов работы мы здесь упомянем только один. По умолчанию, когда заканчивает работу процесс, прослушивавший некоторый порт, в течение нескольких минут система не позволяет другим процессам привязываться к этому порту, возвращая ошибку EINVAL. Это необходимо для того, чтобы проигнорировать пакеты, адресованные уже завершившемуся процессу, которые, возможно, «задержались» в сети. Однако это приводит к тому, что серверный процесс нельзя перезапустить сразу после того, как он завершился, что очень неудобно при разработке и отладке. Если параметр level установить в SOL_SOCKET, optname установить в SO_REUSEADDR, и значение этого параметра установить в 1, порт будет становиться доступным сразу же после завершения процесса-сервера.

1.4 Пример клиента

Рассмотрим простейшую программу-клиент, которая открывает соединение на заданный адрес и порт и посылает текущее время. После этого программа ожидает получения текущего времени сервера, печатает его и завершает работу.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>

int main(int argc, char const *argv[])
{
  int                port, n, sfd;
  struct hostent    *phe;
  struct sockaddr_in  sin;
  time_t             t;

  if (argc != 3) {
    fprintf(stderr, "Too␣few␣parameters\n");
    exit(1);
  }
  /* получим адрес компьютера */
  if (!(phe = gethostbyname(argv[1]))) {
    fprintf(stderr, "Bad␣host␣name:␣%s\n", argv[1]);
```

```
    exit(1);
  }
  /* прочитаем номер порта */
  if (sscanf(argv[2], "%d%n", &port, &n) != 1
      || argv[2][n] || port <= 0 || port > 65535) {
    fprintf(stderr, "Bad␣port␣number:␣%s\n", argv[2]);
    exit(1);
  }
  /* создаём сокет */
  if ((sfd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
  }
  /* для нашего клиента привязка сокета не важна,
     поэтому bind пропускаем */
  /* устанавливаем адрес подключения */
  sin.sin_family = AF_INET;
  sin.sin_port = htons(port);
  memcpy(&sin.sin_addr, phe->h_addr_list[0], sizeof(sin.sin_addr));
  /* подсоединяемся к серверу */
  if (connect(sfd, (struct sockaddr*) &sin, sizeof(sin)) < 0) {
    perror("connect");
    exit(1);
  }
  /* отсылаем текущее время */
  t = time(0);
  t = htonl(t);
  /* желательно проверять результат write */
  write(sfd, &t, sizeof(t));
  /* считываем время сервера */
  if (read(sfd, &t, sizeof(t)) != sizeof(t)) {
    fprintf(stderr, "read␣failed\n");
  } else {
    printf("Server␣time:␣%s", ctime(&t));
  }
  /* закрываем сокет */
  close(sfd);

  return 0;
}
```

1.5 Пример сервера

В качестве программы-сервера рассмотрим программу, которая ожидает подключения от клиента и считывает время клиента. Текущее время сервера пересылается клиенту. После подключения клиента работа с клиентом ведётся в отдельном процессе. Это позволяет параллельно обслуживать несколько клиентов и принимать новые запросы.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>
#include <sys/wait.h>
#include <arpa/inet.h>

int main(int argc, char const *argv[])
{
  int                lfd, n, port, afd, alen, sopt, pid;
  struct sockaddr_in baddr, aaddr;
  time_t             ct, st;

  if (argc != 2) {
    fprintf(stderr, "Too␣few␣parameters\n");
    exit(1);
  }
  /* прочитаем номер порта */
  if (sscanf(argv[1], "%d%n", &port, &n) != 1
      || argv[1][n] || port <= 0 || port > 65535) {
    fprintf(stderr, "Bad␣port␣number:␣%s\n", argv[2]);
    exit(1);
  }
  /* создаём сокет */
  if ((lfd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(1);
  }
  /* задаём режим сокета */
  sopt = 1;
  if (setsockopt(lfd,SOL_SOCKET,SO_REUSEADDR,&sopt,sizeof(sopt))< 0){
    perror("setsockopt");
    exit(1);
  }
  /* задаём адрес сокета */
  baddr.sin_family = AF_INET;
  baddr.sin_port = htons(port);
  baddr.sin_addr.s_addr = INADDR_ANY;
  /* привязываем сокет */
  if (bind(lfd, (struct sockaddr *) &baddr, sizeof(baddr)) < 0) {
    perror("bind");
    exit(1);
  }
  /* включаем режим прослушивания */
  if (listen(lfd, 5) < 0) {
    perror("listen");
    exit(1);
  }
  while (1) {
```

```
    /* хороним <<зомби>> */
    while (waitpid(-1, NULL, WNOHANG) > 0);
    /* ожидаем подключения */
    alen = sizeof(aaddr);
    if ((afd = accept(lfd, (struct sockaddr*) &aaddr, &alen)) < 0) {
      perror("accept");
      exit(1);
    }
    /* запросы клиентов будем обслуживать в другом процессе */
    if ((pid = fork()) < 0) {
      perror("fork");
    } else if (!pid) {
      close(lfd);
      /* печатаем информацию о подключении */
      printf("%d:␣Connection␣from␣%s:%d␣accepted.\n", getpid(),
             inet_ntoa(aaddr.sin_addr), ntohs(aaddr.sin_port));
      if (read(afd, &ct, sizeof(ct)) != sizeof(st)) {
        fprintf(stderr, "read␣error\n");
      } else {
        /* печатаем время клиента и сервера */
        ct = ntohl(ct);
        st = time(0);
        printf("%d:␣Client␣time␣%lu,␣server␣time␣%lu\n",
               getpid(), ct, st);
        st = htonl(st);
        /* желательно проверять результат write */
        write(afd, &st, sizeof(st));
      }
      close(afd);
      _exit(0);
    }
    close(afd);
  }

  return 0;
}
```

2 Мультиплексирование ввода/вывода

В программе может возникнуть необходимость выполнять операции одновременно с несколькими файловыми дескрипторами. Например, процессу может потребоваться считывать данные со стандартного потока ввода и в то же время считывать данные из сокета. Каждая из операций чтения, выполняемая по отдельности, может приостановить процесс на неопределённое время.

Одно из возможных решений этой проблемы заключается в том, чтобы открывать файловые дескрипторы в неблокирующем режиме и постоянно опрашивать их. Недостаток этого метода состоит в неоправданно большой загрузке процессора из-за постоянных системных вызовов. Другой вариант состоит в том, чтобы для каждой операции ввода-вывода, которая может заблокировать процесс, создать отдельный процесс. В этом случае возникает проблема обмена информацией и синхронизации между этими процессами. Если система поддерживает множественность потоков управления (multithreading), можно создать отдельную нить для каждой операции ввода-вывода.

Традиционный способ в системе Unix заключается в использовании системного вызова select[4].

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

int select(int n, fd_set *rfds, fd_set *wfds,
           fd_set *efds, struct timeval *timeout);

FD_CLR(int fd, fd_set *pset);
FD_ISSET(int fd, fd_set *pset);
FD_SET(int fd, fd_set *pset);
FD_ZERO(fd_set *pset);
```

Системный вызов select позволяет приостановить выполнение процесса до тех пор, пока в одном из файловых дескрипторов, заданных в rfds, не появятся данные, немедленно доступные на чтение, либо один из файловых дескрипторов, заданных в wfds, не станет доступным для немедленной записи в него, либо в одном из файловых дескрипторов, заданных в efds, не возникнет исключительная ситуация, либо не закончится интервал времени, заданный в timeout[5]. Любой из этих аргументов может быть равен NULL, что означает пустое множество или бесконечный интервал времени ожидания. Параметр n должен быть на 1 больше максимального номера файлового дескриптора, присутствующего во множествах rfds, wfds, efds. Можно передавать значение FD_SETSIZE, равное максимальному количеству файловых дескрипторов во множестве.

Если файловый дескриптор ассоциирован с сокетом, переключённым в режим прослушивания, его можно проверять на готовность на чтение. Готовность на чтение такого дескриптора означает, что готово подключение клиента, и системный вызов accept не приведёт к приостановке выполнения процесса.

Тип fd_set — это тип для хранения множества файловых дескрипторов. Как и в случае множества типов сигналов sigset_t, определены специальные функции работы со множествами. Функция (макрос) FD_ZERO очищает множество файловых дескрипторов, на которое указывает аргумент pset. Функция FD_SET добавляет файловый дескриптор fd ко множеству файловых дескрипторов, на которое указывает аргумент pset. Функция FD_CLR удаляет файловый дескриптор fd из множества файловых дескрипторов, на которое указывает аргумент pset. Функция FD_ISSET проверяет, присутствует ли файловый дескриптор fd во множестве, на которое указывает аргумент pset.

Системный вызов select возвращает количество файловых дескрипторов, готовых к затребованной операции. Если select завершился из-за истечения заданного лимита вре-

[4] select присутствует во всех UNIX-подобных ОС, но по каким-то причинам эта функция не была стандартизирована POSIX.

[5] К сожалению отсутствует функция, которая позволяла бы приостанавливать процесс до наступления события в файловых дескрипторах, в объектах SysV IPC и у сыновних процессов. То, что с точки зрения ожидания события процессы, файловые дескрипторы и SysV IPC неравноценны, одно из самых неудачных особенностей UNIX. В Windows эта ошибка «исправлена»: работа с почти всеми объектами ведётся через дескриптор, имеющий тип HANDLE, функция WaitForMultipleObjects принимает массив значений этого типа. Почти всеми, кроме сокетов, что практически сводит полезность такого подхода на нет.

мени, возвращается 0. Если выполнение функции завершилось из-за ошибки, возвращается -1, а переменная errno устанавливается в код ошибки. Возможные ошибки перечислены ниже.

EBADF	В одном из множеств был указан неверный файловый дескриптор.
EINTR	Процесс получил и обработал неблокируемый и неигнорируемый сигнал.
EINVAL	Параметр n отрицательный.
ENOMEM	Недостаточно памяти ядра для выполнения операции.

Кроме того системный вызов select модифицирует каждое из переданных в него множеств файловых дескрипторов, так, что установленными остаются только готовые файловые дескрипторы. Последний аргумент — timeout — ведёт себя по-разному в разных системах. На Linux аргумент timeout модифицируется, чтобы отражать время, проведённое в режиме ожидания. В других системах аргумент timeout не модифицируется. Поэтому максимальное время ожидания должно каждый раз переустанавливаться перед вызовом select, и значение timeout после завершения работы функции не должно использоваться.

В качестве примера рассмотрим программу, которая открывает соединение с заданным компьютером и заданным портом, а затем копирует всё, что поступает со стандартного потока ввода, в сокет, и всё, что поступает с сокета, на стандартный поток вывода.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>

int main(int argc, char const *argv[])
{
  int                 port, n, sfd, sz;
  struct hostent     *phe;
  struct sockaddr_in  sin;
  fd_set              fds;
  char                buf[256];

  if (argc != 3) {
    fprintf(stderr, "Too few parameters\n");
    exit(1);
  }
  if (!(phe = gethostbyname(argv[1]))) {
    fprintf(stderr, "Bad host name: %s\n", argv[1]);
    exit(1);
  }
  if (sscanf(argv[2], "%d%n", &port, &n) != 1
      || argv[2][n] || port <= 0 || port > 65535) {
    fprintf(stderr, "Bad port number: %s\n", argv[2]);
    exit(1);
  }
  if ((sfd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
```

```
    perror("socket");
    exit(1);
  }
  sin.sin_family = AF_INET;
  sin.sin_port = htons(port);
  memcpy(&sin.sin_addr, phe->h_addr_list[0], sizeof(sin.sin_addr));
  if (connect(sfd, (struct sockaddr*) &sin, sizeof(sin)) < 0) {
    perror("connect");
    exit(1);
  }

  while (1) {
    FD_ZERO(&fds);
    FD_SET(sfd, &fds);
    FD_SET(0, &fds);
    if (select(FD_SETSIZE, &fds, NULL, NULL, NULL) <= 0) {
      perror("select");
      exit(1);
    }
    if (FD_ISSET(sfd, &fds)) {
      /* надо бы проверять на ошибки */
      sz = read(sfd, buf, sizeof(buf));
      if (!sz) break;
      write(1, buf, sz);
    }
    if (FD_ISSET(0, &fds)) {
      /* надо бы проверять на ошибки */
      sz = read(0, buf, sizeof(buf));
      if (!sz) break;
      write(sfd, buf, sz);
    }
  }

  close(sfd);
  return 0;
}
```