Семинары 14-15. Работа с сокетами.

Понятие сокета. *Сокет* (socket) - это конечная точка сетевых коммуникаций. Он является чем-то вроде "портала", через которое можно отправлять байты во внешний мир. Приложение просто пишет данные в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.

В программе сокет идентифицируется *дескриптором* - это просто переменная типа **int**. Программа получает дескриптор от операционной системы при создании сокета, а затем передаёт его сервисам socket API для указания сокета, над которым необходимо выполнить то или иное действие.

Атрибуты сокета. С каждым сокет связываются три атрибута: *домен*, *тип* и *протокол*. Эти атрибуты задаются при создании сокета и остаются неизменными на протяжении всего времени его существования. Для создания сокета используется функция **socket**, имеющая следующий прототип.

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

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

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Чаще других используются домены Unix и Internet, задаваемые константами **AF_UNIX** и **AF_INET** соответственно (префикс AF означает "address family" - "семейство адресов"). При задании **AF_UNIX** для передачи данных используется файловая система ввода/вывода Unix. В этом случае сокеты используются для межпроцессного взаимодействия на одном компьютере и не годятся для работы по сети. Константа **AF_INET** соответствует Internet-домену. Сокеты, размещённые в этом домене, могут использоваться для работы в любой IP-сети. Существуют и другие домены (**AF_IPX** для протоколов Novell, **AF_INET6** для новой модификации протокола IP - IPv6 и т. д.), но в этой статье мы не будем их рассматривать.

Тип сокета определяет способ передачи данных по сети. Чаще других

применяются:

- **SOCK_STREAM**. Передача потока данных с предварительной установкой соединения. Обеспечивается надёжный канал передачи данных, при котором фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются. Поскольку этот тип сокетов является самым распространённым, до конца раздела мы будем говорить только о нём. Остальным типам будут посвящены отдельные разделы.
- **SOCK_DGRAM**. Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Обмен данными происходит быстрее, но является ненадёжным: сообщения могут теряться в пути, дублироваться и переупорядочиваться. Допускается передача сообщения нескольким получателям (multicasting) и широковещательная передача (broadcasting).
- **SOCK_RAW**. Этот тип присваивается низкоуровневым (т. н. "сырым") сокетам. Их отличие от обычных сокетов состоит в том, что с их помощью программа может взять на себя формирование некоторых заголовков, добавляемых к сообщению.

Обратите внимание, что не все домены допускают задание произвольного типа сокета. Например, совместно с доменом Unix используется только тип **SOCK_STREAM**. С другой стороны, для Internet-домена можно задавать любой из перечисленных типов. В этом случае для реализации **SOCK_STREAM** используется протокол TCP, для реализации **SOCK_DGRAM** - протокол UDP, а тип **SOCK_RAW** используется для низкоуровневой работы с протоколами IP, ICMP и т. д.

Наконец, последний атрибут определяет протокол, используемый для передачи данных. Как мы только что видели, часто протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции **socket** можно передать 0, что соответствует протоколу по умолчанию. Тем не менее, иногда (например, при работе с низкоуровневыми сокетами) требуется задать протокол явно. Числовые идентификаторы протоколов зависят от выбранного домена; их можно найти в документации.

Адреса. 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 мы хотим привязать к заданному адресу. Второй параметр, **addr**, содержит указатель на структуру с адресом, а третий - длину этой структуры. Посмотрим, что она собой представляет.

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

Поле **sa_family** содержит идентификатор домена, тот же, что и первый параметр функции **socket**. В зависимости от значения этого поля по-разному интерпретируется содержимое массива **sa_data**. Разумеется, работать с этим массивом напрямую не очень удобно, поэтому вы можете использовать вместо **sockaddr** одну из альтернативных структур вида **sockaddr_XX** (XX - суффикс, обозначающий домен: "un" - Unix, "in" - Internet и т. д.). При передаче в функцию **bind** указатель на эту структуру приводится к указателю на **sockaddr**. Рассмотрим для примера структуру **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 и двойном слове. Один из них называется *порядком хоста* (host byte order), другой - *сетевым порядком* (network byte order) хранения байтов. 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** и **ntohl**.

Установка соединения (сервер). Установка соединения на стороне сервера состоит из четырёх этапов, ни один из которых не может быть опущен. Сначала сокет создаётся и привязывается к локальному адресу. Если компьютер имеет несколько сетевых интерфейсов с различными IP-адресами, вы можете принимать соединения только с одного из них, передав его адрес функции **bind**. Если же вы готовы соединяться с клиентами через любой интерфейс, задайте в качестве адреса константу **INADDR_ANY**. Что касается номера порта, вы можете задать конкретный номер или 0 (в этом случае система сама выберет произвольный неиспользуемый в данный момент номер порта).

На следующем шаге создаётся очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов. Всё это выполняет функция **listen**.

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

Первый параметр - дескриптор сокета, а второй задаёт размер очереди запросов. Каждый раз, когда очередной клиент пытается соединиться с сервером, его запрос ставится в очередь, так как сервер может быть занят обработкой других запросов. Если очередь заполнена, все последующие запросы будут игнорироваться. Когда сервер готов обслужить очередной запрос, он использует функцию **accept**.

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

```
int accept(int sockfd, void *addr, int *addrlen);
```

Функция **accept** создаёт для общения с клиентом *новый* сокет и возвращает его дескриптор. Параметр **sockfd** задаёт слушающий сокет. После вызова он остаётся в слушающем состоянии и может принимать другие соединения. В структуру, на которую ссылается **addr**, записывается адрес сокета клиента, который установил соединение с сервером. В переменную, адресуемую указателем **addrlen**, изначально записывается размер структуры; функция **accept** записывает туда длину, которая реально была использована. Если вас не интересует адрес клиента, вы можете просто передать NULL в качестве второго и третьего параметров.

Обратите внимание, что полученный от **accept** новый сокет связан с тем же самым адресом, что и слушающий сокет. Сначала это может показаться странным. Но дело в том, что адрес TCP-сокета не обязан быть уникальным в Internet-домене. Уникальными должны быть только *соединения*, для идентификации которых используются *два* адреса сокетов, между которыми происходит обмен данными.

Установка соединения (клиент). На стороне клиента для установления соединения используется функция **connect**, которая имеет следующий прототип.

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

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

Здесь **sockfd** - сокет, который будет использоваться для обмена данными с сервером, **serv_addr** содержит указатель на структуру с адресом сервера, а **addrlen** - длину этой структуры. Обычно сокет не требуется предварительно привязывать к локальному адресу, так как функция **connect** сделает это за вас, подобрав подходящий свободный порт. Вы можете принудительно назначить клиентскому сокету некоторый номер порта, используя **bind** перед вызовом **connect**. Делать это следует в случае, когда сервер соединяется с только с клиентами, использующими определённый порт (примерами таких серверов являются rlogind и rshd). В остальных случаях проще и надёжнее предоставить системе выбрать порт за вас.

Обмен данными. После того как соединение установлено, можно начинать обмен данными. Для этого используются функции **send** и **recv**. В Unix для работы с сокетами можно использовать также файловые функции **read** и **write**, но они обладают меньшими возможностями, а кроме того не будут работать на других платформах (например, под

Windows), поэтому я не рекомендую ими пользоваться.

Функция **send** используется для отправки данных и имеет следующий прототип.

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

Здесь **sockfd** - это, как всегда, дескриптор сокета, через который мы отправляем данные, **msg** - указатель на буфер с данными, **len** - длина буфера в байтах, а **flags** - набор битовых флагов, управляющих работой функции (если флаги не используются, передайте функции 0). Вот некоторые из них (полный список можно найти в документации):

- **MSG_OOB**. Предписывает отправить данные как *срочные* (out of band data, OOB). Концепция срочных данных позволяет иметь два параллельных канала данных в одном соединении. Иногда это бывает удобно. Например, Telnet использует срочные данные для передачи команд типа Ctrl+C. В настоящее время использовать их не рекомендуется из-за проблем с совместимостью (существует два разных стандарта их использования, описанные в RFC793 и RFC1122). Безопаснее просто создать для срочных данных отдельное соединение.
- **MSG_DONTROUTE**. Запрещает маршрутизацию пакетов. Нижележащие транспортные слои могут проигнорировать этот флаг.

Функция **send** возвращает число байтов, которое на самом деле было отправлено (или -1 в случае ошибки). Это число может быть меньше указанного размера буфера. Если вы хотите отправить весь буфер целиком, вам придётся написать свою функцию и вызывать в ней **send**, пока все данные не будут отправлены. Она может выглядеть примерно так.

```
int sendall(int s, char *buf, int len, int flags)
{
   int total = 0;
   int n;

   while(total < len)
   {
      n = send(s, buf+total, len-total, flags);
      if(n == -1) { break; }
      total += n;
   }
```

```
    return (n==-1 ? -1 : total);
}
```

Использование **sendall** ничем не отличается от использования **send**, но она отправляет весь буфер с данными целиком.

Для чтения данных из сокета используется функция **recv**.

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

В целом её использование аналогично **send**. Она точно так же принимает дескриптор сокета, указатель на буфер и набор флагов. Флаг **MSG_OOB** используется для приёма срочных данных, а **MSG_PEEK** позволяет "подсмотреть" данные, полученные от удалённого хоста, не удаляя их из системного буфера (это означает, что при следующем обращении к **recv** вы получите те же самые данные). Полный список флагов можно найти в документации. По аналогии с **send** функция **recv** возвращает количество прочитанных байтов, которое может быть меньше размера буфера. Вы без труда сможете написать собственную функцию **recvall**, заполняющую буфер целиком. Существует ещё один особый случай, при котором **recv** возвращает 0. Это означает, что соединение было разорвано.

Закрытие сокета. Закончив обмен данными, закройте сокет с помощью функции **close**. Это приведёт к разрыву соединения.

```
#include <unistd.h>

int close(int fd);
```

Вы также можете запретить передачу данных в каком-то одном направлении, используя **shutdown**.

```
int shutdown(int sockfd, int how);
```

Параметр **how** может принимать одно из следующих значений:

- 0 - запретить чтение из сокета
- 1 - запретить запись в сокет
- 2 - запретить и то и другое

Хотя после вызова **shutdown** с параметром **how**, равным 2, вы больше не сможете использовать сокет для обмена данными, вам всё равно потребуется вызвать **close**, чтобы

освободить связанные с ним системные ресурсы.

Обработка ошибок. До сих пор я ни слова не сказал об ошибках, которые могут происходить (и часто происходят) в процессе работы с сокетами. Так вот: если что-то пошло не так, все рассмотренные нами функции возвращают -1, записывая в глобальную переменную **errno** код ошибки. Соответственно, вы можете проанализировать значение этой переменной и предпринять действия по восстановлению нормальной работы программы, не прерывая её выполнения. А можете просто выдать диагностическое сообщение (для этого удобно использовать функцию **perror**), а затем завершить программу с помощью **exit**. Именно так я буду поступать в демонстрационных примерах.

Отладка программ. Начинающие программисты часто спрашивают, как можно отлаживать сетевую программу, если под рукой нет сети. Оказывается, можно обойтись и без неё. Достаточно запустить клиента и сервера на одной машине, а затем использовать для соединения адрес *интерфейса внутренней петли* (loopback interface). В программе ему соответствует константа INADDR_LOOPBACK (не забудьте применять к ней функцию **htonl**!). Пакеты, направляемые по этому адресу, в сеть не попадают. Вместо этого они передаются стеку протоколов TCP/IP как только что принятые. Таким образом моделируется наличие виртуальной сети, в которой вы можете отлаживать ваши сетевые приложения. Для простоты я буду использовать в демонстрационных примерах интерфейс внутренней петли.

Обмен датаграммами. Как уже говорилось, датаграммы используются в программах довольно редко. В большинстве случаев надёжность передачи критична для приложения, и вместо изобретения собственного надёжного протокола поверх UDP программисты предпочитают использовать TCP. Тем не менее, иногда датаграммы оказываются полезны. Например, их удобно использовать при транслировании звука или видео по сети в реальном времени, особенно при широковещательном транслировании.

Поскольку для обмена датаграммами не нужно устанавливать соединение, использовать их гораздо проще. Создав сокет с помощью **socket** и **bind**, вы можете тут же использовать его для отправки или получения данных. Для этого вам понадобятся функции **sendto** и **recvfrom**.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
        const struct sockaddr *to, int tolen);
```

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

Функция **sendto** очень похожа на **send**. Два дополнительных параметра **to** и **tolen** используются для указания адреса получателя. Для задания адреса используется структура **sockaddr**, как и в случае с функцией **connect**. Функция **recvfrom** работает аналогично **recv**. Получив очередное сообщение, она записывает его адрес в структуру, на которую ссылается **from**, а записанное количество байт - в переменную, адресуемую указателем **fromlen**. Как мы знаем, аналогичным образом работает функция **accept**.

Некоторую путаницу вносят *присоединённые датаграммные сокеты* (connected datagram sockets). Дело в том, что для сокета с типом SOCK_DGRAM тоже можно вызвать функцию **connect**, а затем использовать **send** и **recv** для обмена данными. Нужно понимать, что никакого соединения при этом не устанавливается. Операционная система просто запоминает адрес, который вы передали функции **connect**, а затем использует его при отправке данных. Обратите внимание, что присоединённый сокет может получать данные *только* от сокета, с которым он соединён.

Использование низкоуровневых сокетов. Низкоуровневые сокеты открывают перед вами новые горизонты. Они предоставляют программисту полный контроль над содержимым пакетов, которые отправляются в путешествие по сети. С другой стороны, они сложнее в использовании и обладают плохой переносимостью. Вот почему использовать их следует только в случае необходимости. Например, без них не обойтись при разработке системных утилит типа ping и traceroute.

Первым делом выясним, чем низкоуровневые сокеты отличаются от обычных. Работая с обычными сокетами, вы передаёте системе "чистые" данные, а она сама заботится о добавлении к ним необходимых заголовков (а иногда ещё и концевиков). Например, когда вы посылаете сообщение через UDP-сокет, к нему добавляется сначала UDP-заголовок, потом IP-заголовок, а в самом конце - заголовок аппаратного протокола, который используется в вашей локальной сети (например, Ethernet). В результате получается кадр, показанный на рисунке 1.

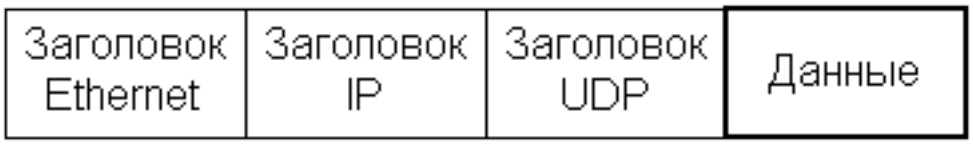

Рисунок 1

Низкоуровневые сокеты позволяют вам включать в буфер с данными заголовки некоторых протоколов. Например, вы можете включить в ваше сообщение TCP- или UDP-заголовок, предоставив системе сформировать для вас IP-заголовок, а можете вообще сформировать все заголовки самостоятельно. Разумеется, при этом вам придётся изучить работу соответствующих протоколов и строго соблюсти формат их заголовков, иначе программа работать не будет.

При работе с низкоуровневыми сокетами вам придётся указывать в третьем параметре функции **socket** тот протокол, к заголовкам которого вы хотите получить доступ. Константы для основных протоколов Internet объявлены в файле **netinet/in.h**. Они имеют вид **IPPROTO_XXX**, где XXX-название протокола: **IPPROTO_TCP**, **IPPROTO_UDP**, **IPPROTO_RAW** (в последнем случае вы получите возможность поработать с "сырым" IP и формировать IP-заголовки вручную).

Чтобы проиллюстрировать всё это примером, я переписал программу sender из предыдущего раздела с использованием низкоуровневых UDP-сокетов. При этом мне пришлось вручную формировать UDP-заголовок отправляемого сообщения. Я выбрал для примера UDP, потому что у этого протокола заголовок выглядит совсем просто (рисунок 2).

Порт отправителя (16 бит)	Порт получателя (16 бит)
Длина (заголовок+данные) (16 бит)	Контрольная сумма (16 бит)

Рисунок 2

Функции для работы с адресами и DNS. В этом разделе мы обсудим несколько функций, без которых можно написать учебный пример, но без которых вряд ли обойдётся реальная программа. Поскольку для идентификации хостов в Internet широко используются доменные имена, мы должны изучить механизм преобразования их в IP-адреса. Кроме того мы изучим несколько удобных вспомогательных функций.

IP-адреса принято записывать в виде четырёх чисел, разделённых точками. Для преобразования адреса, записанного в таком формате, в число и наоборот используется семейство функций **inet_addr**, **inet_aton** и **inet_ntoa**.

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

int inet_aton(const char *cp, struct in_addr *in_p);
unsigned long int inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

Функция **inet_addr** часто используется в программах. Она принимает строку и возвращает адрес (уже с сетевым порядком следования байтов). Проблема с этой функцией состоит в том, что значение -1, возвращаемое ею в случае ошибки, является в то же время корректным адресом 255.255.255.255 (широковещательный адрес). Вот почему сейчас рекомендуется использовать более новую функцию **inet_aton** (Ascii TO Network). Для обратного преобразования используется функция **inet_ntoa** (Network TO Ascii). Обе эти функции работают с адресами в сетевом формате. Обратите внимание, что в случае ошибки они возвращают 0, а не -1.

Для преобразования доменного имени в IP-адрес используется функция **gethostbyname**.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Эта функция получает имя хоста и возвращает указатель на структуру с его описанием. Рассмотрим эту структуру более подробно.

```
struct hostent {
   char    *h_name;
   char    **h_aliases;
   int     h_addrtype;
   int     h_length;
   char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

- **h_name**. Имя хоста.
- **h_aliases**. Массив строк, содержащих псевдонимы хоста. Завершается значением NULL.
- **h_addrtype**. Тип адреса. Для Internet-домена - **AF_INET**.

- **h_length**. Длина адреса в байтах.
- **h_addr_list**. Массив, содержащий адреса всех сетевых интерфейсов хоста. Завершается нулём. Обратите внимание, что байты каждого адреса хранятся с сетевым порядке, поэтому **htonl** вызывать не нужно.

Как видим, **gethostbyname** возвращает достаточно полную информацию. Если нас интересует адрес хоста, мы можем выбрать его из массива **h_addr_list**. Часто берут самый первый адрес (как мы видели выше, для ссылки на него определён специальный макрос **h_addr**). Для определения имени хоста по адресу используется функция **gethostbyaddr**. Вместо строки она получает адрес (в виде **sockaddr**) и возвращает указатель на ту же самую структуру **hostent**. Используя эти две функции, нужно помнить, что они сообщают об ошибке не так, как остальные: вместо указателя возвращается **NULL**, а расширенный код ошибки записывается в глобальную переменную **h_errno** (а не **errno**). Соответственно, для вывода диагностического сообщения следует использовать **herror** вместо **perror**.

В заключение рассмотрим ещё одно семейство полезных функций - **gethostname**, **getsockname** и **getpeername**.

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

Функция **gethostname** используется для получения имени локального хоста. Далее его можно преобразовать в адрес при помощи **gethostbyname**. Это даёт нам способ в любой момент программно получить адрес машины, на которой выполняется наша программа, что может быть полезным во многих случаях.

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

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

Функция **getpeername** позволяет в любой момент узнать адрес сокета на "другом конце" соединения. Она получает дескриптор сокета, соединённого с удалённым хостом, и записывает адрес этого хоста в структуру, на которую указывает **addr**. Фактическое количество записанных байт помещается по адресу **addrlen** (не забудьте записать туда размер структуры **addr** до вызова **getpeername**). Полученный адрес при необходимости можно преобразовать в строку, используя **inet_ntoa** или **gethostbyaddr**. Функция **getsockname** по назначению обратна **getpeername** и позволяет определить адрес сокета на

"нашем конце" соединения.

Параллельное обслуживание клиентов. Следующий важный вопрос, который нам предстоит обсудить, - это параллельное обслуживание клиентов. Эта проблема становится актуальной, когда сервер должен обслуживать большое количество запросов. Конечно, на машине с одним процессором настоящей параллельности достичь не удастся. Но даже на одной машине можно добиться существенного выигрыша в производительности. Допустим, сервер отправил какие-то данные клиенту и ждёт подтверждения. Пока оно путешествует по сети, сервер вполне мог бы заняться другими клиентами. Для реализации такого алгоритма обслуживания существует множество способов, но чаще всего применяются два из них.

Способ 1.Этот способ подразумевает создание дочернего процесса для обслуживания каждого нового клиента. При этом родительский процесс занимается только прослушиванием порта и приёмом соединений. Чтобы добиться такого поведения, сразу после **accept** сервер вызывает функцию **fork** для создания дочернего процесса (я предполагаю, что вам знакома функция **fork**; если нет, обратитесь к документации). Далее анализируется значение, которое вернула эта функция. В родительском процессе оно содержит идентификатор дочернего, а в дочернем процессе равно нулю. Используя этот признак, мы переходим к очередному вызову **accept** в родительском процессе, а дочерний процесс обслуживает клиента и завершается (**_exit**).

Очевидное преимущество такого подхода состоит в том, что он позволяет писать весьма компактные, понятные программы, в которых код установки соединения отделён от кода обслуживания клиента. К сожалению, у него есть и недостатки. Во-первых, если клиентов очень много, создание нового процесса для обслуживания каждого из них может оказаться слишком дорогостоящей операцией. Во-вторых, такой способ неявно подразумевает, что все клиенты обслуживаются независимо друг от друга. Однако это может быть не так. Если, к примеру, вы пишете чат-сервер, то ваша основная задача - поддерживать взаимодействие всех клиентов, присоединившихся к нему. В этих условиях границы между процессами станут для вас серьёзной помехой. В подобном случае вам следует серьёзно рассмотреть другой способ обслуживания клиентов.

Способ 2. Второй способ основан на использовании *неблокирующих сокетов* (nonblocking sockets) и функции **select**. Сначала разберёмся, что такое неблокирующие сокеты. Сокеты, которые мы до сих пор использовали, являлись *блокирующими* (blocking).

Это название означает, что на время выполнения операции с таким сокетом ваша программа блокируется. Например, если вы вызвали **recv**, а данных на вашем конце соединения нет, то в ожидании их прихода ваша программа "засыпает". Аналогичная ситуация наблюдается, когда вы вызываете **accept**, а очередь запросов на соединение пуста. Это поведение можно изменить, используя функцию **fcntl**.

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
```

Эта несложная операция превращает сокет в неблокирующий. Вызов любой функции с таким сокетом будет возвращать управление немедленно. Причём если затребованная операция не была выполнена до конца, функция вернёт -1 и запишет в **errno** значение **EWOULDBLOCK**. Чтобы дождаться завершения операции, мы можем опрашивать все наши сокеты в цикле, пока какая-то функция не вернёт значение, отличное от **EWOULDBLOCK**. Как только это произойдёт, мы можем запустить на выполнение следующую операцию с этим сокетом и вернуться к нашему опрашивающему циклу. Такая тактика (называемая в англоязычной литературе polling) работоспособна, но очень неэффективна, поскольку процессорное время тратится впустую на многократные (и безрезультатные) опросы.

Чтобы исправить ситуацию, используют функцию **select**. Эта функция позволяет отслеживать состояние нескольких файловых дескрипторов (а в Unix к ним относятся и сокеты) одновременно.

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

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

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

Функция **select** работает с тремя множествами дескрипторов, каждое из которых имеет тип **fd_set**. В множество **readfds** записываются дескрипторы сокетов, из которых нам требуется читать данные (слушающие сокеты добавляются в это же множество). Множество **writefds** должно содержать дескрипторы сокетов, в которые мы собираемся писать, а **exceptfds** - дескрипторы сокетов, которые нужно контролировать на возникновение ошибки. Если какое-то множество вас не интересуют, вы можете передать вместо указателя на него **NULL**. Что касается других параметров, в **n** нужно записать максимальное значение дескриптора по всем множествам плюс единица, а в **timeout** - величину таймаута. Структура **timeval** имеет следующий формат.

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

Поле "микросекунды" смотрится впечатляюще. Но на практике вам не добиться такой точности измерения времени при использовании **select**. Реальная точность окажется в районе 100 миллисекунд.

Теперь займёмся множествами дескрипторов. Для работы с ними предусмотрены функции **FD_XXX**, показанные выше; их использование полностью скрывает от нас детали внутреннего устройства **fd_set**. Рассмотрим их назначение.

- **FD_ZERO(fd_set *set)** - очищает множество **set**
- **FD_SET(int fd, fd_set *set)** - добавляет дескриптор **fd** в множество **set**
- **FD_CLR(int fd, fd_set *set)** - удаляет дескриптор **fd** из множества **set**
- **FD_ISSET(int fd, fd_set *set)** - проверяет, содержится ли дескриптор **fd** в множестве **set**

Если хотя бы один сокет готов к выполнению заданной операции, **select** возвращает ненулевое значение, а все дескрипторы, которые привели к "срабатыванию" функции, записываются в соответствующие множества. Это позволяет нам проанализировать содержащиеся в множествах дескрипторы и выполнить над ними необходимые действия. Если сработал таймаут, **select** возвращает ноль, а в случае ошибки -1. Расширенный код записывается в **errno**.