Организация сетевого взаимодействия. Сокеты. Каналы

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

Дата последней модификации: 03.04.2015 14:47:00

История изменений

15.08.2014 – Версия 0.1. Первичный документ. Ковтун В.Ю.

Содержание

Лекция 9. Особенности сетевого взаимодействия

Вопросы

1. Сокеты.
2. Каналы.

Взаимодействие процессов через сокеты

Понятие сокета

Сокет – это объект ОС, через который можно передавать и принимать данные от процессов независимо от того, выполняется она на одном или разных компьютерных сетях. Сокет, также как и файл, характеризуется номером дескриптора и к нему применены файловые операции. Но в отличие от файла сокет существует лишь до момента, пока на него ссылается хотя бы один из процессов. Сокет передается по наследству и с ним могут быть связаны один или несколько процессов. Сокеты, имеющие одинаковые схемы адресации и семейство протоколов, группируются в домены. В системе LINUX поддерживается следующие домены:

- AF_UNSPEC 0 /*не специфицированный*/
- AF_UNIX 1 /*для взаимодействия процессов внутри одной ЭВМ*/
- AF_INET 2 /*для взаимодействия через локальную сеть по протоколу TCP/IP*/
- AF_AX25 3 /*для взаимодействия аппаратному интерфейсу X.25*/
- AF_IPX 4 /*для взаимодействия через сеть по протоколу IPX*/

В дальнейшем приводится описание параметров и операций с сокетами только для домена AF_INET.

В рамках одного домена сокеты могут иметь различный тип и протокол обмена. Тип сокета определяет способ передачи данных между процессами. В системе определены следующие типы сокетов:

- SOCK_STREAM 1 /*потоковый сокет*/
- SOCK_DGRAM 2 /*дейтаграммный сокет*/
- SOCK_RAM 3 /*исходный сокет*/
- SOCK_RDM 4 /*надежная передача сообщения*/
- SOCK_SEQPACKET 5 /*последовательные пакеты*/

Поточный сокет обеспечивает двухсторонний, последовательный, надежный, и недублированный поток данных без определенных границ. Перед началом обмена данными через такой сокет процессы должны установить канал связи. Тип сокета - SOCK_STREAM, в домене Интернета использует протокол TCP.

Датаграммный сокет не гарантирует надежную передачу и получение данных в той последовательности, в какой они посылались, но этот способ передачи более быстрый, поскольку для него не требуется сложные установочные операции. Этот тип сокетов не требует предварительной установки соединения, адреса отправителя и получателя передаются с каждым сообщением. Тип сокета - SOCK_DGRAM, в домене Интернета использует протокол UDP.

Сокет последовательных пакетов: можно организовать передачу сообщений фиксированный длинны с предварительным установлением соединением. Тип сокета - SOCK_SEQPACKET. Для этого типа сокета не существует специального протокола.

Простой сокет - обеспечивает доступ к основным протоколам связи. Они используется, когда нужно создать новый протокол связи или получить доступ к параметрам существующего протокола.

Одна из сильных сторон программного интерфейса сокетов состоит в том, что в нем заложены средства для работы с различными сетевыми протоколами. Например, для домена AF_INET определены следующие типы протоколов:

- IPPROTO_IP = 0, /* Dummy protocol for TCP*/
- IPPROTO_ICMP = 1, /* Control Message Protocol*/
- IPPROTO_IGMP = 2, /* Internet Gateway Management Protocol */
- IPPROTO_IP IP = 4, /* IPIP tunnels (older KA9Q tunnels use 94)*/
- IPPROTO_TCP = 8, /* Transmission Control Protocol*/
- IPPROTO_PUP = 12, /* Exterior Gateway Protocol */

- IPPROTO_UDP = 17, /* PUP protocol*/
- IPPROTO_IDP = 22, /* User Datagram Protocol */
- IPPROTO_PUP = 12, /* XNS IDP protocol*/
- IPPROTO_RAW = 255, /* Raw IP packets*/

Для каждой допустимой комбинации домен - тип сокета в системе поддерживается умолчание на используемый протокол. Так, например, для домина `AF_INET` услуги виртуального канала выполняет протокол транспортной связи `IPPROTO_TCP`, а функции дейтаграммы – пользовательский дейтаграммный протокол `IPPROTO_UDP`: Для сокетов типа `SOCK_RAW` и `SOCK_PACKET` обязательно должен быть указан номер протокола суперпользователя.

Операции с сокетом

Программный интерфейс сокетов очень похож на программный интерфейс файлов. Операции с сокетом осуществляется в основном по той же схеме, как и с файлом: открытие – чтение (запись) - закрытие. Для сокета также применима функция fcntl, которая устанавливает флаг O_NONBLOCK, запрещающий блокировать процесс при выполнении операций чтения и записи.

Однако имеется и ряд отличий, которые связаны с тем, что сокет после открытия не привязан к конкретному сетевому адресу и не связан с другим сонетом, с которым он должен обмениваться данными. Поэтому программный интерфейс сокетов, прежде всего, расширен операциями, подготавливающими сокет к обмену данными. Одной из таких операций является привязка сокета к сетевому адресу и порту. Другие операции настраивают сокеты на соединение по виртуальному каналу связи.

Полный программный интерфейс сокетов включает в себя более 20 функций. В дальнейшем будем рассматривать лишь несколько функций.

Открытие совета.

Открытие сокета - это операция, выполняющая построение дескриптора сокета. В этом дескрипторе хранится информация о типе сокета, его текущем состоянии и используемом протоколе. Для открытия сокета используйте системный вызов:

```
int socket(int family, int type, int protocol).
```

Параметрами вызова являются тип домена, тип сокета и номер протокола. Если номер протокола не указан (значение параметра 0), то система самостоятельно назначит протокол в зависимости от типа сокета.

Значением функции `socket` является номер дескриптора сокета, через который можно будет обращаться к сокету, или (-1), если функция завершилась неудачно. Одновременно в системе могут быть открыты не более 256 сокетов. Сокет не будет создан, если неправильно задан один из параметров или для указанного домена и типа сокета в системе отсутствует протокол.

Привязка сокета к сетевому адресу

Для того чтобы сокеты, открытые на разных машинах, могли обмениваться данными, необходимо их привязать к сетевым адресам. Эту операцию выполняет системный вызов:

```
int bind(int fd, struct sockaddr *umyaddr, int addrlen),
```

где `fd` - номер дескриптора открытого сокета; `umyaddr` - указатель на структуру, в которой задан сетевой адрес; `addrlen` - размер этой структуры. Структура типа `sorkaddr` определена в файле `/Linux/include/socket.h` и имеет следующее описание: struct sockaddr {

```
unsigned short   sa_family;        /* тип домена, AF_xxx     */
char             sa_data[14];     /* 14 байт для адреса сокета  */
};
```

Для различных доменов адрес сокета задается по-разному. Для домена AF_INET адрес сокета определяется через следующую структуру:

```
struct sockaddr_in {
short int         sin_family;       /* тип домена – значение AF_ INET */
```

```
unsigned short int sin_port;        /* 16-разрядный номер порта*/
struct in_addr        sin_addr;     /* 32-разрядный IP-адрес */
/*массив pad пользователем не заполняется */
unsigned char     _pad _pad[_SOCK_SIZE_ - sizeof(short int) –
sizeof(unsigned short int) – sizeof(struct in_addr)];
};
```

При определении адреса сокета для домена AF_INET в поле `sin_port` необходимо указать номер порта, в который будут приниматься данные или откуда они будут посылаться. Можно посылать данные по одному IP-адресу, но в разные порты. Если задать 0 в поле `sin_port`, то ядро самостоятельно выделяет порт. Номер порта является целым положительным числом в диапазоне от 0 до 32767. Первые 1023 номера выделены для суперпользователей. Если порт уже используется другим сокетом, то функция `bind` возвратит (-1).

Особенностью поля `sin_port` является то, что значение в нем должно храниться в перевернутом формате, т. е. по младшему адресу содержится старший байт. Поэтому перед записью значения необходимо выполнить соответствующие преобразование с помощью функции `u_short htons (u_short numport)`, которая осуществляет обмен байт в слове.

В поле `sin_addr` должен быть записан IP-адрес, который является 32-разрядным числом, разбитым на 4 части но 8 бит, например 194.85.168.20. Компьютеру, подключенному в сеть, администратор при ее настройке присваивает символическое имя и один или несколько IP-адресов. Соответствие между IP-адресом и именем машины задается в файле `/etc/hosts`. Если в поле `sin_addr` записать константу `INADDR_ANY`, то считается, что сокет имеет IP-адрес, определенный в файле `/etc/hosts`. Например, заполнение структуры `struct sockaddr_in saddr` для сокета, которому IP-адрес и номер порта присвоит ОС, будет таким:

```
saddr. sin_tamily=AF_INET;
saddr. sin_port=0;
saddr.sin addr.s_addr= INADDR_ANY;
```

Приложение может узнать имя машины и ее IP-адрес. Для этого можно воспользоваться двумя функциями:

```
int uname(struct utsname * buf);
struct hostent * gethostbyname(char * name);
```

С помощью функции `uname` можно получить имя машины. Оно будет помещено в поле `nodename` структуры `utsname`. Адрес этой структуры указывается в качестве параметров функции. Затем по имени машины можно получить параметры ее IP-адреса. Эти параметры функция `gethostbyname` помещает в следующие поля структуры `hostent`: `h_addr` - IP-адрес, `h_length` - длина адреса, `h_addttype` - тип домена, к которому принадлежит адрес.

Процесс может ужать адреса данного сокета и сокета, с которым он связан, обратившись соответственно к системным вызовам:

```
int getsockname(int fd, struct sockaddr * umyaddr, int *usockaddr_len) и
int getpeername(int fd, struct sockaddr * umyaddr, int * usockaddr_len),
```

где параметры имеют тот же смысл, как и в функции `bind`.

Для удобства чтения IP-адреса его числовое значение можно преобразовать в строку символов, разбитую на 4 части, которые будут отделены друг от друга точками. Такое преобразование выполняет функция `char*inet_ntoa(struct in_addr)`.

Подготовка сокета к созданию канала связи

Для создания сокета, ориентированного на установление виртуальных каналов с клиентами (тип `SOCK_STREAM` или `SOCK_SEQPACKET`), необходимо вызвать функцию `listen`, которая имеет следующее описание:

```
int listen(int fd, int acklog),
```

где fd - номер дескриптора сокета, acklog - максимальное количество запросов на соединения, которые могут быть поставлены в очередь к данному сокету.

Пока процесс-сервер устанавливает связь по виртуальному каналу с одним клиентом, другие клиенты могут запрашивать соединение с сервером. Поэтому поступающие запросы должны быть поставлены в очередь на обслуживание. Если очередь полна, сервер игнорирует новые запросы на установку связи. Это заставляет клиента снова попытаться осуществить связь и дает серверу время, чтобы подготовиться к принятию запроса на соединение. Максимальное значение аргумента acklog может быть 128, но, как правило, он имеет значение 5.

При успешном выполнении эта функция информирует ядро о готовности процесса принимать запросы на соединение и возвращает 0, а в случае неудачи - (-1).

Приём запросов на создание канала

Процесс-сервер всегда работает в режиме ожидания поступления запросов от клиентов на создание виртуального канала. Эти запросы система помещает в очередь, из которой они извлекаются СВ:

`int accept(int fd, struct sockaddr * upper_sockaddr, int *upeer_addrlen)`, где fd - номер дескриптора файла сокета сервера; upper_sockaddr - указатель на структуру, в которую будет передано имя сокета клиента; upeer_addrlen -указатель на переменную, в которую будет передана длина имени. При выполнении вызова создается новый сокет для связи с клиентом. Этот сокет используется для обслуживания данного клиента. Если в очереди нет запросов на соединение, процесс-сервер переходит в состояние ожидания. Как только поступит запрос от клиента, процесс-сервер будет активизирован.

Функция accept возвращает номер дескриптора нового сокета, когда приходит запрос от клиента, и (-1), когда она завершается неудачно.

Посыпка запроса на создание канала

Соединение сокета клиента с сокетом сервера осуществляется с помощью системного вызова:

`int connect(int fd, strukt sockaddr * uservaddr, int_addrlen),`

где fd - номер дескриптора сокета клиента; uservaddr - адрес сокета сервера; addrlen - размер адреса.

Оба сокета должны использовать одни и те же домен и протокол связи, только в этом случае ядро подтвердит правильность установки линии связи. Если клиент и сервер слизываются через потоковые сокеты, то между ними устанавливается соединение с использованием виртуального канала. Связь между сокетами устанавливается только в том случае, если сервер уже выполнил функцию accept. В противном случае виртуальный канал не образуется и функция connect возвратит (-1). Потоковый сокет клиента может соединяться с сокетом сервера только один раз, поэтому повторную установку связи должен осуществляться уже другой сокет. При успешном соединении функция connect возвращается 0.

Если тип сокетов - дейтаграмма, то в момент вызова функции connect никаких соединений не производится, а сообщаемый адрес узла будет использован по умолчанию при передаче сообщения. В этом случае вместе с сообщением можно не передавать адрес узла назначения. Один и тот же дейтаграммный сокет может соединяться с различными сокетами, путем установки нового адреса функцией connect.

Передача данных

Послать данные через сокет можно с помощью следующих системных вызовов:

```
int write(int fd, char*ubuf, int len);

int send(int fd, void * buff, int len, unsigned flags);

int sendto(int fd, void * buff, int len, unsigned flags, struct sockaddr*addr,
int addr_len);
```

Параметры вызовов: fd - номер дескриптора сокета, через который передаётся данные; ubuf - указатель на буфер, где содержатся передаваемые данные; len - длинна передаваемых данных в байтах; flags - флаги; addr - адрес сокета получателя; addr_len

- размер адреса. Функции возвращают количество переданных байт, а в случае неудачи – (-1).

Если данные посылаются через потоковый сокет, то можно использовать функции `write`, `send` и `sendto`. Для дейтаграммного сокета обычно используют функцию `sendto`. Но если сокет с помощью функции `connect` установил соединение с другим сокетом, то для передачи сообщения можно также воспользоваться функциями `write` и `send`.

Аргумент `flags` может иметь значение 0 или `MSG_OOB`. Флаг `MSG_OOB` означает, что должно быть передано высокоприоритетное сообщение, которое будет обслуживаться получателем в первую очередь.

При выполнении операции передачи данных могут быть три результата:

- процесс перейдет в состояние ожидания, если нет свободного места в буфере передачи и не установлен флаг `O_NONBLOCK`. Он будет находится в этом состоянии до тех пор, пока не поступят данные;
- операция завершится с сокетом (-1), если неправильно указаны параметры или нет свободного места в буфере передачи и установлен флаг `O_NONBLOCK`;
- операция возвратит число переданных байт.

Прием данных

Принять данные из сокета можно с помощью следующих системных вызовов:

```
int read(int fd, char *ubuf, int size);

int recv(int fd, void * buff, int len, unsigned flags);

int recvfrom(int fd, void * buff, int len, unsigned flags, struct sockaddr * addr, int * addr_len).
```

Параметры вызовов: `fd` – номер дескриптора сокета, через который принимаются данные; `ubuf` - указатель на буфер, в который читаются данные; `len` - длина передаваемых данных; `flags` – флаги; `addr` - адрес сокета, передающего данные; `addr_len` - размер адреса.

Если данные принимаются через потоковый сокет, то можно использовать функции `read`, `recv` и `recvfrom`. Перед выполнением функций `read` и `recv` необходимо, чтобы в сокете находился адрес отправителя. Для дейтаграммного сокета можно воспользоваться только функциями `recv` и `recvfrom`.

Аргумент `flags` может иметь значения 0, `MSG_OOB` или `MSG_PEEK`. Значение 0 указывает, что будут приниматься обычные сообщения; MSG_OOB -что приему подлежат высокоприоритетные сообщения; MSO_PEEK - что процесс будет только просматривать сообщение, не извлекая его из сокета. Такой процесс может повторно прочесть просмотренное сообщение.

При выполнении операции приема данных могут быть три результата:

- процесс перейдет в состояние ожидания, если данных нет и не установлен флаг `O_NONBLOCK`. Он будет находиться в этом состоянии до тех нор, пока не поступят данные;
- операция завершится с кодом (-1), если неправильно укачаны параметры или нет данных и установлен флаг `O_NONBLOCK`;
- операции возвратит число прочитанных байт. Если сообщение содержит больше байт, чем заказано для чтения, то будет прочитано только закачанное число, если меньше - будит прочитано все сообщение.

Прерывание связи с сокетом

Полностью или частично прервать связь между двумя сокетами можно с помощью системного вызова

```
int shutdown(int fd, int how).
```

Параметры: `fd` - номер дескриптора одного из сокетов, между которыми происходит разрыв соединения, `how` - задает режим разрыва связи. При значении how равным нулю сокет закрывается для приема данных, при 1 - сокет закрывается для передачи данных, при 2 - сокет закрывается для передачи и приема данных.

Закрытие пикета

Закрытие сокета выполняет вызов `int close(int fd)`, параметром которого является номер дескриптора сокета. После закрытия сокета, все операции с ним запрещены, сообщения, находящиеся в сокете теряются, а процессы, ожидающие соединения или данных от сокета, активизируются. Если приложение завершается аварийно, то ОС закрывает все открытые сокеты.

Получение и установка параметров сокета

Получить параметры сокета можно с помощью системного вызова

`getsockopt(int fd, int level, int optname, char*optval, int*optlen),`

а установить параметры - с помощью выгона:

`setsockopt(int fd, int level, int optname, char*optval, int optlen).`

Аргументы вызовов: `fd` - номер дескриптора сокета; `level` - указывает, требуется ли информация о самом сокете или об использующем его протоколе (при значении `SOL_SOCKET` запрашивается информация о сокете, при указании номера протокола передается информация о протоколе); `optname` - указывает тип запрашиваемой информации; `optval` - указатель на область памяти, в которую заносятся результаты запроса или из которой берутся значения для записи; `optlen` - размер области памяти.

Чаще всего к функциям `getsockopt` и `setsockopt` обращаются для чтения и установки размеров буферов приёма и передачи данных. По умолчанию для потоковых сокетов эти буфера имеют размеры 16384 байт, а для дейтаграммных сокетов – 9216 байт. Чтобы получить информацию о размерах буферов приёма и передачи сообщений, параметр `optname` должен соответственно иметь значение `SO_RCVBUF` и `SO_SNDBUF`.

Опрос сокетов

Приложение может открыть несколько сокетов и взаимодействовать через них с другими приложениями. Для обслуживания каждого сокета можно породить отдельный процесс, но в этом случае в системе будет много процессов и производительность компьютера резко снизится. Постоянный анализ состояния каждого сокета также приведет к замедлению работы приложения. Поэтому в таких приложениях целесообразно использовать функцию `select`, которая позволяет задержать выполнение приложения до тех пор, пока не произойдёт некоторое событие в сокете. Эта функция имеет следующее описание:

`int select(int numfds, fd_set * readfds, fd_set*writefds, fd_set*exeptfds, struct timeval*timeout).`

Параметр `numfds` определяет диапазон дескрипторов сокетов, по которым ожидаются запросы. Обычно в качестве значения этого параметра указывают константу `FD_SETSIZE`, означающую, что нужно анализировать все открытые сокеты. Следующие три параметра являются указателями на битовые маски, в которых записываются номера дескрипторов сокетов, подлежащих анализу. В параметре `readfds` указывается, какие сокеты надо проверять на готовность приема данных, в параметре `writefds` - какие сокеты надо проверять на готовность посылки данных, и в параметре `exeptfds` - какие сокеты надо проверять на возникновение ошибочных ситуаций. Если проверка не требуется, то соответствующий параметр должен иметь значение 0.

Номера опрашиваемых сокетов устанавливаются в переменных типа `fd_set`, которые представляют собой массив из 256 бит и имеют следующее описание:

```
typedef struct fd_set }
            unsigned long fds_bits [8];
} fd_set;
```

Единица, записанная в соответствующий разряд переменной типа `fd_set`, указывает, какие номера сокетов используются для анализа. Для работы с переменными типа `fd_set` имеется следующий набор макросов, определенных в библиотечном файле `types.h`:

- FD_SET(int fd, fd_set *set) - запись номера дескриптора сокета;
- FD_CLR(int fd, fd_set *set) - сброс номера дескриптора сокета;
- FD_ZERO(fd_set *set) - погнан очитка;

- FD_ISSET(int fd, fd_ set*set) - проверка значения.

В аргументе `fd` задается номер дескриптора сокета, а в аргументе `set` должен содержаться адрес переменной типа `fd_set`, где хранятся номера анализируемых сокетов.

Параметр `timeout` (тайм-аут) указывает, какое максимальное время должен ожидать процесс до поступлении информации от сокетов. Он относится к типу `struct timeval`, определенному в `sys/times.h` и имеющему следующее описание:

```
struct  timeval {
long tv_sec;      /*время в секундах */
            long tv_usec;     /*время в микросекундах */
```

Если информация поступает до истечения тайм-аута, то `select` сразу возвратит управление процессу, указывая в двоичных масках, от каких сокетов поступила информация. Например, если пользователь пожелал приостановиться до момента получения данных от сокетов с номерами 0, 1 и 2, то в параметре `readfds` он должен задать двоичную маску 7. Когда `select` возвратит управление, двоичная маска, будет заменена маской, указывающей номера кокетов, в которые пришли данные.

Если параметр `timeout` имеет значение 0, то процесс будет приостановлен до тех пор, пока от одного ни сокетов не придёт сигнал об изменении его состояния.

Сокет готов выполнить операцию приема данных:

- если, пришел запрос на установку виртуального канала;
- если в буфере приема появились данные;
- если несвязанный сокет послал сообщение.

Сокет готов выполнить операцию передачи данных:

- если установлен виртуальный канал;
- если имеется свободное пространство в буфере передачи.

Значением функция `select` является число сокетов, от которых поступила информация, а в переменных `readfds`, `writefds`, `exeptfds` будут содержаться номера дескрипторов этих сокетов. Если было задано время тайм-аута, то в переменную `timeout` будет записан остаток времени. Если вызов функции завершается по истечении времени и ни от одного из сокетов не поступила информация, тогда возвращается значение (-1).

Пример использования функции select для проверки поступления запросов на соединение:

```
int id_socket, res_select;
fd_set select_set;
struct timeval time_out;
time_out.tv_sec=2; /* время ожидания 2 с */
time_out.tv_uset=0;
FD_ZERO(&select_set);   /* очистка */
FD_SET(id_socket, &select_set);   /*запись номера дескриптора сокета */
res_select=select(FD_SETSIZEr &select_set, 0, 0, &time_out);
```

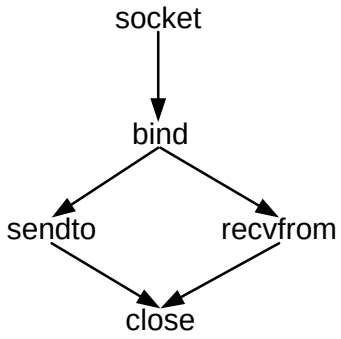
Схемы установки связи и передачи данных через сокеты

Сокеты, которые используются для организации взаимодействия между процессами, должны быть одного типа и принадлежать одному домену. Тип сокета выбираемся в зависимости от требований надежности, скорости передачи данных и способа установки связи. Механизм сокетов позволяет организовать взаимодействие между процессами двумя способами:

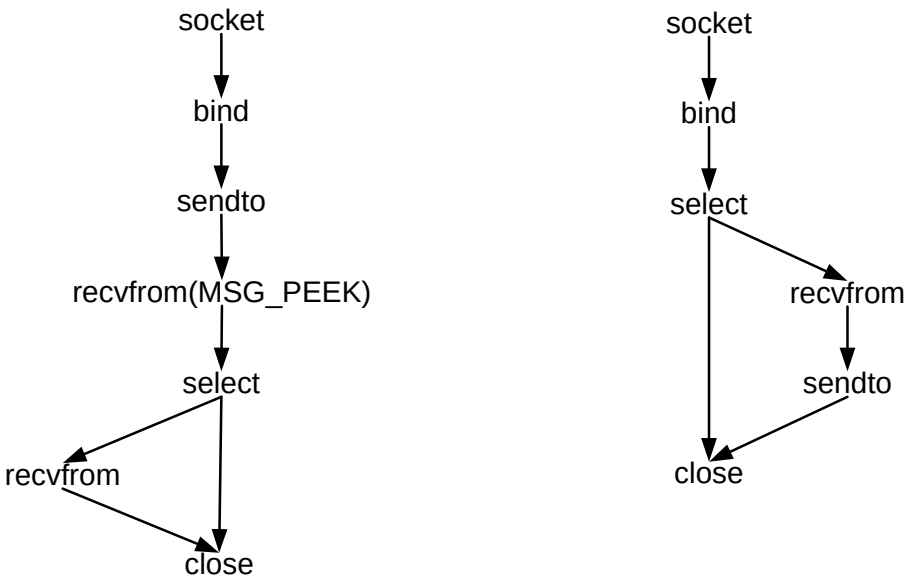
- без установления соединений - в этом случае адреса отравителя и получателя передаются с каждым сообщением,
- с установлением соединения, когда адреса получателя и отправителя определяются до передачи сообщения.

Первый способ взаимодействия результата через дейтаграммные сокеты, которым обязательно должны быть присвоены сетевые адреса. При этом необходимо, чтобы

процесс, посылающий первое сообщение, знал IP-адрес и номер порта сокета, которому оно адресовано. Обмен данными между ними осуществляется с помощью функций `sendto` и `recvfrom`. При посылке данных в аргументе функции `sendto` указывается адрес получателя, а после приема данных в аргументе функции `recvfrom` будет содержаться адрес отправителя. Процесс, получивший обобщение, может воспользоваться адресом отправителя при подготовке ответа. Последовательность вызовов функций для организации обмена данными без установления соединения будет такой:

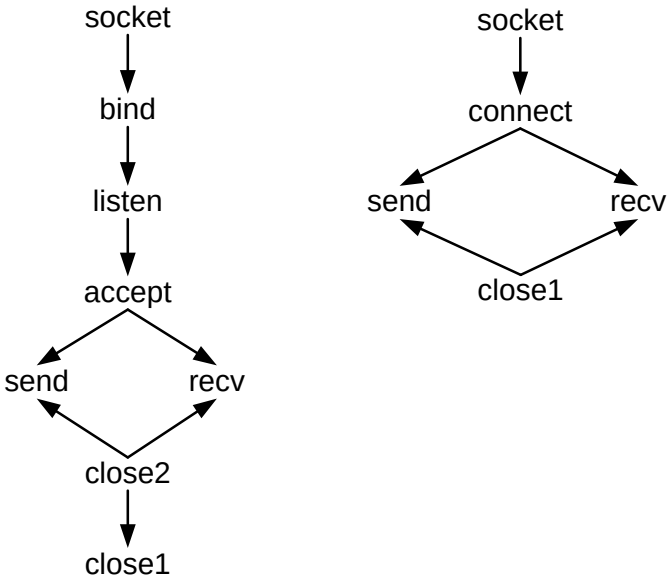


Процессы, обменивавшиеся данными по этой схеме, при выполнении операции `recvfrom` будут находиться в постоянном ожидании, пока не поступят данные. Для ограничения времени ожидания можно воспользоваться функцией `select`, которая проверяет готовность сокета к приему данных. Она приостанавливает процесс либо до прихода данных, либо до истечении заданного времени. Последовательность вызовов функций для организации обмена данными с ограниченным временем ожидания будет такой:

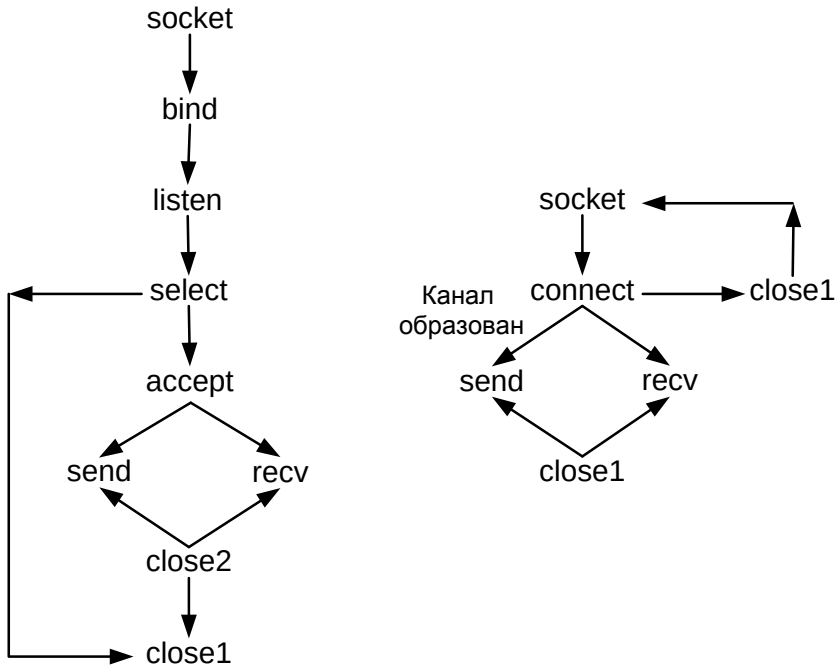


У функции `select` имеется одна особенность: она фиксирует готовность сокета принять данные сразу после выполнения операции передачи (`sendto`), хотя данные в буфер приема еще не поступили. Поэтому перед вызовом функции `select` необходимо предварительно выполнить просмотр буфера приема данных с помощью функций `recvfrom` при установленном флаге `MSG_PEEK`.

Взаимодействие с установкой соединения реализуется через потоковые сокеты. Одному из них обязательно должен быть назначен сетевой адрес. До начала обмена процессы с помощью функций `accept` и `connect` устанавливают виртуальный канал, причем для его образования в аргументах функции `connect` указывается IP-адрес и номер порта сокета, который использует, функцию `accept`. Последовательности вызовов функций для двух процессов, взаимодействующих через виртуальный канал, будут такими:



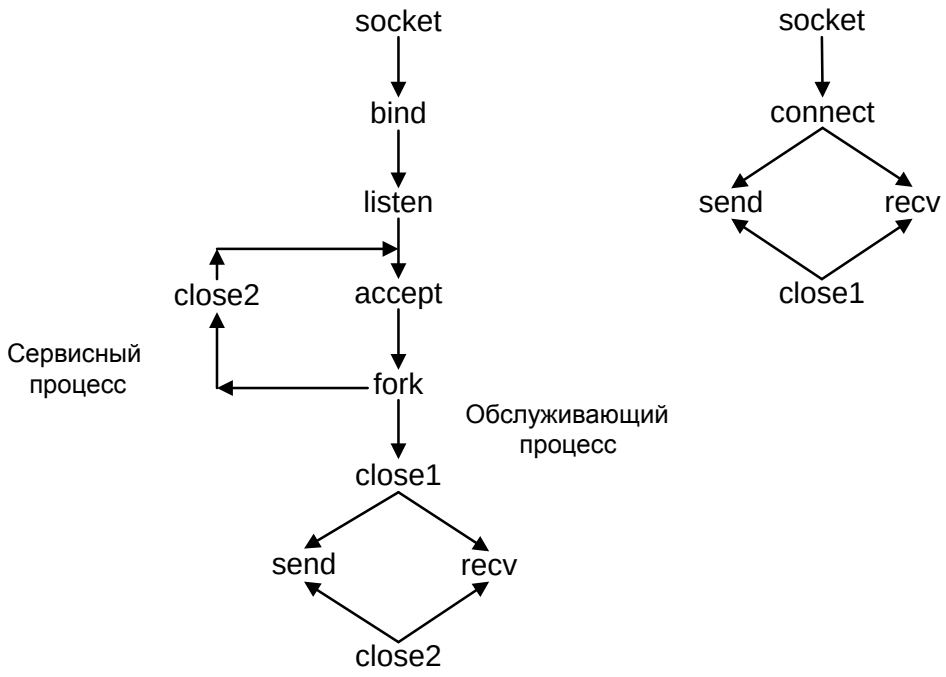
После получения подтверждения соединения оба сокета будут знать, с кем они связаны, и могут начать обмениваться данными посредством функций `sent(write)` и `recv(read)`. Когда обмен данными будет закончен, необходимо закрыть два сокета: первый - открытый функцией `socket`, а второй - функцией `accept`. Приведенная схема взаимодействия предполагает, что процесс, использующий функцию `accept`, запущен первым и постоянно находится в состояния ожидания соединения. Если первым будет запушен процесс, вызывающий функцию `connect`, то канал между сокетами образован не будет. Для повторного соединения необходимо закрыть старый сокет и открыть новый. Обычно эту операцию выполняют несколько раз, пока либо не будет образован канал, либо не будут исчерпаны все попытки соединения. Ограничить время ожидания соединения можно с помощью функции `select`. Последовательность вызовов функций для организации такого соединения будет иметь следующий вид:



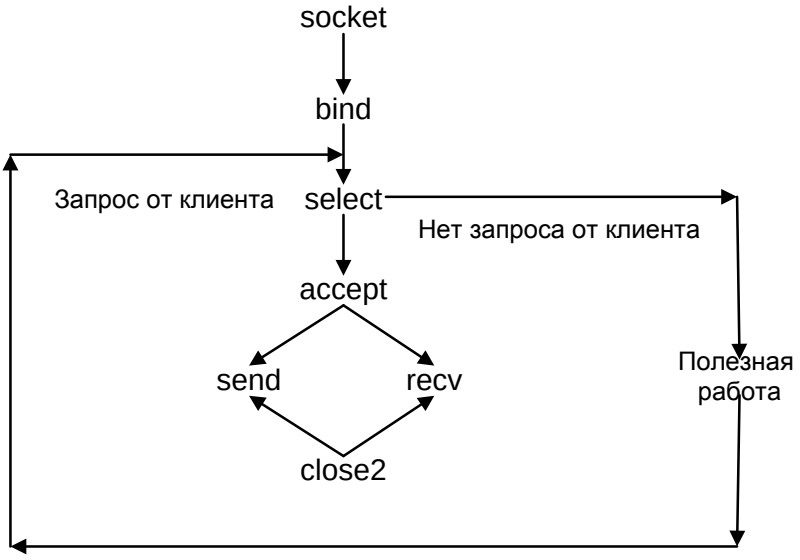
На основе потоковых сокетов строится взаимодействие процессов типа клиент-сервер. Серверный процесс должен непрерывно ожидать запросы от клиентов и по мере возможности одновременно их обслуживать. Такая организация серверного процесса может быть реализована в виде цикла, в котором вызываются функции `accept` и `fork`. Функция `fork` порождает обслуживающий процесс для обмена данными с клиентом. Наличие двух процессов позволит быстро принимать заявки от клиентов и параллельно их выполнять.

Обслуживающий процесс является потомком серверного процесса. Он наследует два сокета. Первый сокет, открытый для установки соединений с клиентами, не нужен в обслуживающем процессе и поэтому он сразу закрывается (close1). Второй сокет, созданный функцией accept, используется для обмена данными с клиентом. Он закрывается (close2) после выполнения операций обмена. В серверном процессе, наоборот, закрывается сокет, порожденный функцией accept, поскольку он используется только в обслуживающем процессе.

Клиентский процесс устанавливает связь с сервером посредством функции `connect`, в которой указывает его сетевой адрес. После установки соединения клиентский процесс выполняет передачу данных серверу и ждет от него ответа. Последовательность вызовов функций для организации серверного и клиентского процессов будет такой:



В предыдущей схеме серверный процесс постоянно находится в состоянии ожидания соединения с клиентом. Но пока нет запросов на соединение, он может выполнять какую-нибудь полезную работу. Для построения такой схемы сервера используют функцию `select`. Она позволяет проверять наличие запроса клиента на соединение и в зависимости от результата этой проверки серверный процесс может либо принять запрос, либо перейди к выполнению полезной работы.



Взаимодействие процессов через каналы

Понятие канала

Программный канал - средство коммуникации процессов, которое можно рассматривать как программно реализованную линию связи для передачи данных между двумя или более процессами. Каналы реализованы в виде снабженных специальным механизмом управления буферов ввода-вывода, которые трактуются как псевдо-файлы. Формальное сходство с файлами позволяет использовать одинаковый аппарат для доступа к данным в канале и файле. Такой подход соответствует базовой концепции унификации ввода-вывода в OS UNIX. При этом каналы имеют специфические особенности, которые позволяют исключить недостатки организации межпроцессного обмена через файлы. Основные особенности каналов перечислены ниже.

Каналы принято использовать следующим образом. Один процесс направляет (записывает) данные в канал, другой процесс получает (читает) данные из канала (Рис. 1).

Рис. 1. Одноканальная схема обмена 2-х процессов

Следует отметить, что в OS UNIX нет принципиальных препятствий для организации обмена нескольких процессов через один или более каналов. Из одноканальных схем наиболее целесообразной в практическом плане является схема с одним читающим процессом-сервером и несколькими пишущими процессами-клиентами, которая показана на Рис. 2.

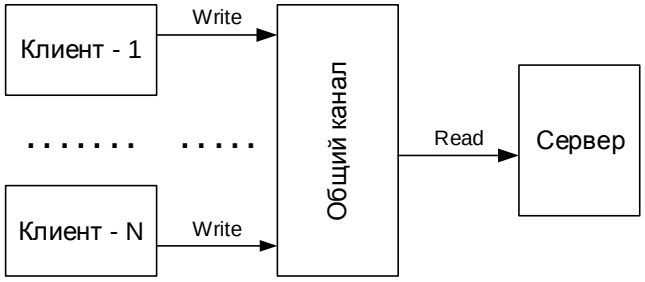


Рис. 2. Одноканальная схема обмена клиент-сервер

Обратная одноканальная схема с одним пишущим и несколькими читающими процессами применима, когда безразличен выбор читающего процесса для обработки текущих данных, получаемых из канала. Применение канального обмена в рамках одного процесса обычно является бессмысленным.

В большинстве версий OS UNIX канал является однонаправленным механизмом, т.е. поддерживает передачу данных только в одном направлении. Чтобы реализовать обмен данными в двух направлениях нужно предусмотреть 2 разнонаправленных канала. Вариант 2-х канального обмена 2-х процессов показан на Рис. 3.

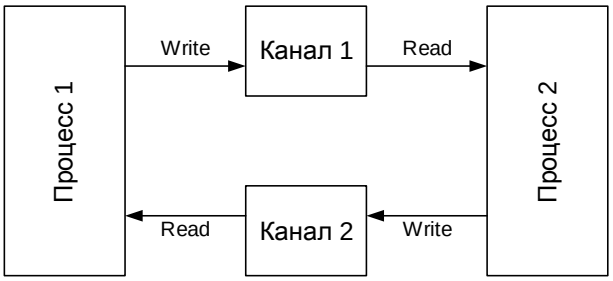


Рис. 3. Двухканальный двунаправленный обмен 2-х процессов

Доступ к данным при любом варианте канального обмена осуществляется через канальные дескрипторы чтения и записи. **Канальные дескрипторы** концептуально эквивалентны пользовательским дескрипторам открытых файлов в контексте процесса и связаны со входами канала по чтению и записи. Канал открыт для обмена, пока существует связанная с ним пара канальных дескрипторов. Через них канальный механизм осуществляет контроль за смещение указателей чтения-записи, положение которых определяет в какой блок канала можно записать данные и из какого блока они должны быть прочитаны.

Через канал может быть передан неограниченный объем информации, хотя мгновенная емкость канала ограничена 10-ю блоками. Ситуации переполнения канала при записи и чтение пустого канала автоматически блокируются скрытым механизмом синхронизации обмена. Он обеспечивает приостановку процесса записи, когда в канале нет места для размещения новых данных, или процесса чтения, при попытке ввода из пустого канала, который пока открыт по записи.

Формальной моделью канала является кольцевая очередь с дисциплиной обслуживания FIFO. Состояние канальной очереди определяется указателями чтения и записи, которые доступны через дескрипторы канала. Смещение этих указателей определяет, куда следует записать поступившие данные и откуда они могут быть прочитаны. На Рис. 4 показано одно из возможных состояний канальной очереди.

Рис. 4. Формальная модель канала

OS UNIX поддерживает 2 типа каналов: **неименованные** (или обычные) и **именованные** (или FIFO-файлы). Неименованные каналы поддерживают обмен данными только между процессами-родственниками, например, между предком и потомком или между потомками одного предка. Именованные каналы могут быть использованы для коммуникации любых, необязательно родственных, процессов. Каналы указанных типов создаются и открываются по-разному, но используются одинаково.

Неименованные каналы

Неименованный канал создается и открывается СВ `pipe`, которому в качестве аргументов передается адрес массива 2-х целых чисел. При успешном завершении СВ `pipe` заполняет адресуемый массив канальными дескрипторами чтения и записи. Эту инициализацию обычно выполняет процесс-предок, когда предполагается использовать канальный механизм для обмена с потомками или между потомками.

Полученные канальные дескрипторы будут наследоваться всеми потомками, которые порождены данным предком после реализации СВ `pipe` вместе с таблицей открытых файлов контекста предка. Потомки и предок могут освободить (закрыть) в своих контекстах тот канальный дескриптор, который не будет использован для обмена. Если обычный канал создается только для коммуникации потомков и не предполагается реализация их обмена с предком, последний может освободить оба канальных дескриптора в своем контексте. Во всех перечисленных случаях нужно использовать СВ `close`, чтобы освободить лишние канальные дескрипторы.

Именованный канал

В отличие от обычного канала, каждому именованному каналу должен соответствовать оригинальный по маршрутному имени канальный файл, который может быть расположен в произвольном каталоге файловой системы OS UNIX. Для создания канального файла в прикладной программе можно использовать системные вызовы `mkfifo` или `mknod`. СВ `mkfifo` ориентирован исключительно на создание FIFO-файлов. СВ `mknod` может быть использован для создания новых файлов любых типов. Аналогично, в командном режиме именованный канал может быть создан специальной командой `/etc/mkfifo` или универсальной командой `/etc/mknod`.

Для начала работы с именованным каналом он должен быть открыт во всех процессах, которые будут использовать его как средство обмена. Чтобы открыть именованный канал, каждый заинтересованный в обмене процесс должен использовать СВ `open`, которому передается имя канального файла и желаемый режим обмена - чтение или запись. СВ `open` является универсальным средством открытия любых файлов, но для FIFO-файлов его реализация имеет свои особенности. В частности, когда именованный канал открывается по чтению (записи), СВ `open` переводит реализующий его процесс в состояние ожидания, пока канал не будет открыт по записи (чтению) каким-либо другим процессом. При успешном завершении СВ `open` возвращает канальный дескриптор чтения или записи, который может быть использован для работы с каналом в контексте данного процесса и его потенциальных потомков.

Реализация обмена данными через каналы

Обмен данными через обычный и именованный каналы реализован одинаково через системные вызовы `read` и `write`. Для работы с каналом им передаются соответствующий канальный дескриптор чтения или записи, адрес и размер объекта обмена. Также как СВ `open`, системные вызовы `read` и `write` являются универсальным средством ввода-вывода данных через заданный дескриптор в контексте процесса, которое с равным успехом применимо для работы с каналом, каталогом, обычным и специальным файлом. Однако, канальная реализация этих универсальных системных вызовов ввода-вывода имеет специфические особенности.

При канальном обмене СВ read применяется для ввода данных из канала. Он возвращает реальное число прочитанных данных или 0, когда канал пуст и закрыт по записи всеми пишущими процессами. Попытка чтения из пустого канала, не закрытого по записи вызывает блокировку читающего процесса. Эта ситуация возникает, когда процесс чтения данных из канала опережает процесс записи данных в него. Чтение данных из канала является деструктивным, т.е. прочитанные данные не могут быть перепрочитаны вновь.

При канальном обмене СВ `write` применяется для вывода данных в канал. Он возвращает реальное число записанных данных. Попытка записи в полный канал вызывает блокировку пишущего процесса. Такая ситуация возникает, когда процесс записи данных в канал опережает процесс чтения данных из него. Если канал закрыт всеми читающими процессами, то пишущий процесс получает сигнал `SIGPIPE` при попытке записи данных в канал.

Блокировка чтения пустого и записи полного канала может быть исключена переводом канала в режим неблокированного ввода-вывода путем установки режимных флагов `O_NDELAY` или `O_NONBLOCK` с помощью системного вызова `fcntl`. Установка режимных флагов обеспечивает немедленный возврат кода 0 или (-1) системными вызовами ввода-вывода `read` и `write` при попытке чтения пустого или записи в полный канал. Для именованного канала режим неблокированного ввода-вывода может быть изначально заказан при открытии канала СВ `open`.

Закрытие каналов

Для корректного завершения работы с каналом любого типа нужно закрыть канал, освободив все канальные дескрипторы во всех процессах, которые использовали канал для обмена. Как указано выше, следует применять СВ `close`, для освобождения канальных дескрипторов в контексте всех процессов, использовавших канал для обмена. Время жизни обычного канала определяет период существования его канальных дескрипторов в контекстах взаимодействующих процессов. Очевидно, что обычный канал не может существовать после завершения процессов, которые использовали его для обмена. В отличие от обычных каналов, именованный канальный файл сохраняется

независимо от существования использующих его процессов, также как файл любого другого типа. Однако, в отличие от обычных файлов данные в именованном канале не будут сохранены после завершения всех процессов, которые использовали его для обмена. Длина канального файла будет равна нулю. Канальный FIFO-файл нулевой длины будет присутствовать в файловой системе, пока он не удален командой `rm` или системным вызовом `unlink`.

Примеры использования каналов

Следующий фрагмент С-кода демонстрирует передачу содержимого текстовой строки между двумя процессами через существующий именованный канал `chanel`. Первый процесс выполняет программу `writer`, обеспечивая запись строки в канал.

```
main() {

static char *str = "Hello"; /* передаваемая строка */

int fd;                     /* дескриптор канала */

fd = open("chanel",1);      /* открыть канал по записи */

write(fd,str,strlen(str));  /* записать данные в канал */

close(fd);                  /* освободить дескриптор канала */

exit(0);                    /* завершить процесс writer */

}
```

Второй процесс, выполняя программу `reader`, обеспечивает чтение данных из каналального файла `chanel` и распечатывает их на стандартный вывод.

```
main() {

char c;                 /* принимаемый символ */

int fd;                 /* канальный дескриптор */

fd = open("chanel",0);  /* открыть канал по чтению */

while(read(fd,&c,1) > 0) /* чтение данных из канала */

write(1,&c,1);          /* запись данных в канал */

close(fd);              /* освободить дескриптор канала */

exit(0);                /* завершить процесс reader */

}
```

Обмен данными через именованный канал `chanel` может быть реализован путем запуска процессов `writer` и `reader` с отдельных экранов, причем последовательность запуска значения не имеет.

Конвейер команд

В практике работы с OS UNIX обычные каналы часто используются для организации конвейера команд. Конвейер образует цепочка параллельных процессов, реализующая последовательную обработку данных. Процессы выполнения соседних команд конвейера взаимодействуют через обычный канал. Для обозначения каналов в конвейере используется символ '|', который понимает командный процессор при разборе командной строки. Стандартный вывод каждого конвейерного процесса, кроме последнего в цепочке

команд, перенаправляется на вывод в канал. Стандартный ввод каждого конвейерного процесса, кроме первого в цепочке команд, перенаправляется на ввод из канала. Например, следующий конвейер обеспечивает постраничный просмотр содержания текущего каталога в длинном формате:

```
$ ls -l | more
```

Для программной реализации перенаправления ввода-вывода при конвейерной обработке могут быть использованы системные вызовы `dup`, `dup2` и `fcntl` (режим `F_DUPFD`) в сочетании с СВ `close`. Эти средства позволяют ассоциировать заданный дескриптор канала с минимальным по номеру свободным элементом таблицы открытых файлов контекста процесса. **Например**, следующий фрагмент С-кода освобождает файловый дескриптор стандартного вывода (по умолчанию равный 1) и делает его синонимом дескриптор `fd`, который соответствует ранее открытому файлу или каналу.

```
close(1);  /* освобождает дескриптор стандартного вывода */

dup(fd);   /* дублирует дескриптор fd */

close(fd); /* освобождает дубликат дескриптора fd */
```

Аналогичная последовательность действий выполняется для перенаправления стандартного ввода в канал или файл. Следующий, более представительный, фрагмент С-кода моделирует интерпретацию приведенного выше конвейера командным процессором `Shell` с помощью обычного канала и рассмотренных средств перенаправления ввода-вывода.

```
/* Модель конвейера команд: ls-l |  more */

main() {

int fd[2];  /* массив канальных дескрипторов */



/* инициализация канальных дескрипторов */

if     (pipe(fd)                 <   0)

       exit(1);



/* Создание 1-го процесса потомка */

if (fork() == 0) {

   close(1);                     /* Стандартный вывод    */

   dup(fd[1]);                   /* перенаправляется     */

   close(fd[0]);                 /* на вывод в канал     */

   close(fd[1]);                 /* по дескриптору fd[1] */

   execl("/bin/ls","ls","-l",0); /* Замена программы     */

   exit(1);                      /* 1-го потомка         */

}  /* if */
```

```
/* Создание 2-го процесса потомка */

if (fork() == 0) {

   close(0);                    /* Стандартный ввод     */

   dup(fd[0]);                  /* перенаправляется     */

   close(fd[0]);                /* на ввод из канала    */

   close(fd[1]);                /* по дескриптору fd[0] */

   execl("/bin/more","more",0); /* Замена программы     */

   exit(1);                     /* 2-го потомка         */

} /* if */


/* Закрытие канала в процессе-предке */

close(fd[0]);

close(fd[1]);


/* Ожидание завершения потомков */

while(wait(0)       !=  (-1));

exit(0);

}
```

Литература

1. Олифер В., Олифер Н. Сетевые операционные системы: Учебник для вузов. 2-е изд. — СПб.: Питер, 2009. — 669 с.: ил.

2. Робачевский А. М. Операционная система UNIX®. % СПб.: 2002. % 528 ил.

3. Microsoft Development Network. URL: http://msdn.com