**Лекция «Модели ввода-вывода»**

**Рязанова Н.Ю.**

В Unix/Linux поддерживается 5 моделей ввода-вывода [1]. В [2] отмечается, что в Unix подобных системах доступно 5+1 модели ввода-вывода. Речь идет о моделях ввода-вывода с точки зрения программиста.

Пять моделей ввода-вывода:

* блокирующий ввод-вывод (blocking I/O)
* неблокирующий ввод-вывод (nonblocking I/O)
* ввод-вывод с мультиплексированием (I/O multiplexing: select and poll)
* ввод-вывод, управляемый сигналом (signal driven I/O: SIGIO)
* асинхронный ввод-вывод (asynchronous I/O: the POSIX aio functions)

Обычно есть две отдельные фазы для операции ввода:

1. Ожидание готовности данных
2. Копирование данных из ядра в буфер процесса

Для операции ввода в сокет первый шаг обычно включает ожидание поступления данных в сеть. Когда пакет прибывает, он копируется в буфер в ядре. Второй шаг - копирование этих данных из буфера ядра в буфер текущего приложения.

***Модель блокирующего ввода-вывода***

Самая распространенная модель ввода-вывода - это модель блокирующего ввода / вывода (рис 1).

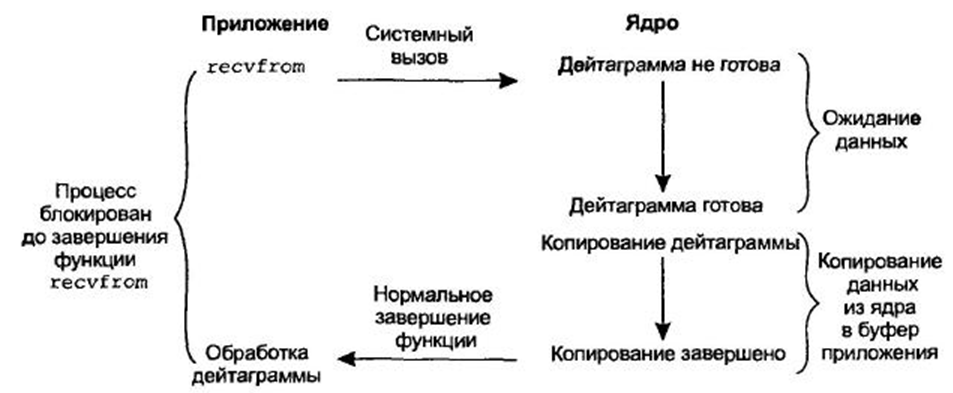


Рис.1

При блокирующем вводе-выводе процесс блокирован все время от момента, когда был выполнен вызов recvfrom, до момента, когда дейтаграмма поступает и копируется в буфер приложения. Системный вызов может передать в буфер приложения данные или ошибку. При успешном завершении вызова приложение обрабатывает полученные данные.

***Модель неблокирующего ввода-вывода***

Неблокирующий ввод-вывод не ожидает наличия данных (или возможности вывода), а получает результат выполнения операции или невозможность её выполнения в данный момент, что определяется по коду возврата. В листинге 1 представлен фрагмент кода с примером неблокирующего ввода [3].

Листинг 1. Неблокирующий ввод-вывод

|  |  |
| --- | --- |
|  | int fo[ 2 ];         // pipe–канал для чтения из дочернего процесса  if ( pipe ( fo ) ) perror ( "pipe" ), exit( EXIT\_FAILURE );  close ( fo[ 1 ] );  int cur\_flg = fcntl( fo[ 0 ], F\_GETFL );  // чтение должно быть в режиме O\_NONBLOCK  if ( -1 == fcntl( fo[ 0 ], F\_SETFL, cur\_flg | O\_NONBLOCK ) )     perror ( "fcntl" ), exit( EXIT\_FAILURE );  ...  while( 1 ) {     int n = read( fdi, buf, buflen );     if( n > 0 ) {         // данные считаны ...         // обработка     }     else if ( -1 == n ) {        if( EAGAIN == errno ) {  // данные не готовы           printf( "not ready!\n" );           usleep( 300 );           continue;        }        else perror( "\nread pipe" ), exit( EXIT\_FAILURE );     }  }  Устанавливается неблокирующий режим ввода (рис.2). |

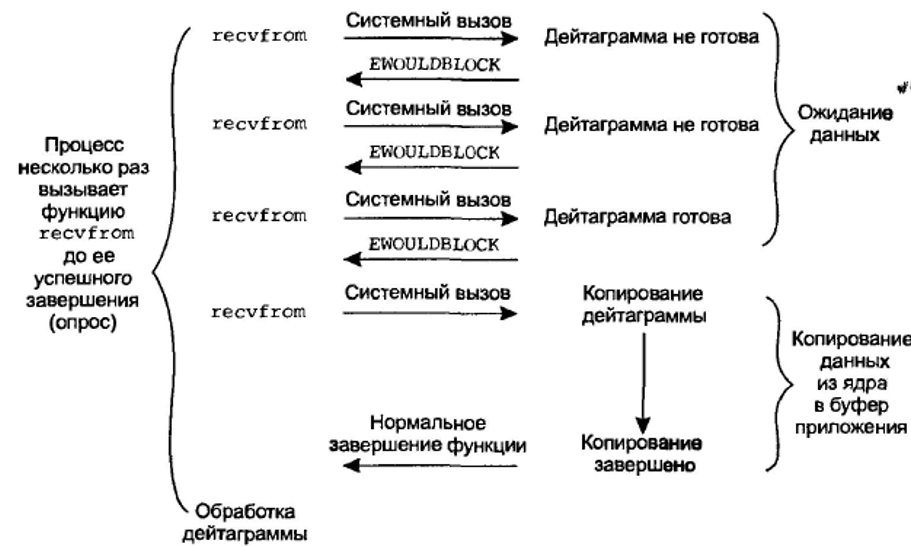


Рис.2

Первые три раза системный вызов на чтение данные не возвращает, а возвращает ошибку EWOULDBLOCK. Следующий системный вызов выполняется успешно, так как данные готовы для чтения. Ядро скопирует данные в буфер приложения и будут обработаны.

Такой режим работы называется опросом (polling), так как выполняется постоянный вызов, в данном случае, recvfrom. Для организации такого опроса может быть создан цикл, вызывающий соответствующий системный вызов. Очевидно, что такой подход приводит к большим накладным расходам (overhead) – пустой трате процессорного времени. Эта модель иногда встречается, как правило, при работе с внешними устройствами.

***Модель с мультиплексирование ввода-вывода***

При мультиплексировании ввода-вывода вызываются, так называемые мультиплексоры[[1]](#footnote-1), select или poll и вместо блокировки в непосредственном системном вызове ввода-вывода процесс блокируется на одном из этих системных вызовов.

Для подобной функциональности в POSIX API предлагаются две реализации [3]: старая на основе вызова **select()** и новая на основе **pselect()**:

|  |  |
| --- | --- |
|  | int select( int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout );  int  pselect( int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, const struct timespec \*timeout, sigset\_t \*sigmask ); |

Вызов **select()** использует тайм-аут в виде **struct timeval** (с секундами и микросекундами), а в **pselect()** для этой же цели используется **struct timespec** (с секундами и наносекундами).

Вызов **select()** может изменить значение параметра **timeout**, чтобы сообщить, сколько времени осталось, а вызов **pselect()** не поддерживает такую функциональность.

Вызов **select()** не содержит параметра **sigmask**, и ведет себя как **pselect()** с параметром **sigmask** равным **NULL**. Если этот параметр для **pselect()** не равен **NULL**, то **pselect()** сначала замещает текущую маску сигналов на переданную в **sigmask**, а затем выполняет **select()** и восстанавливает исходную маску сигналов.

Параметр тайм-аута может задаваться несколькими способами:

1. **NULL**, что означает ожидать вечно;
2. ожидать конкретный указанный период времени;
3. не ожидать вообще (программный опрос, pooling), когда структура тайм-аута инициализируется значением **{0, 0}**.

Обе эти функции возвращают значение больше нуля, которое соответствует числу готовых к операции дескрипторов, ноль — в случае истечения тайм-аута, и отрицательное значение при ошибке.

Кроме того, вводится понятие набора дескрипторов и макросы для работы с ними

|  |  |
| --- | --- |
| 1  2  3  4 | FD\_CLR( int fd, fd\_set \*set );  FD\_ISSET( int fd, fd\_set \*set );  FD\_SET( int fd, fd\_set \*set );  FD\_ZERO( fd\_set \*set ); |

Ещё один вариант мультиплексирования ввода-вывода вывода предлагается в функции **poll()**, где представление битового набора дескрипторов заменено на массив структур вида:

|  |  |
| --- | --- |
|  | struct pollfd {     int fd;           /\* файловый дескриптор \*/     short events;     /\* запрошенные события \*/     short revents;    /\* возвращенные события \*/  }; |

В структуре: **fd** — открытый файловый дескриптор; **events** — набор битовых флагов запрошенных событий для этого дескриптора; **revents** — набор битовых флагов, возвращенных событием для этого дескриптора (из числа запрошенных, или **POLLERR**, **POLLHUP**, **POLLNVAL**).

Возможные комбинации битов описаны в файлах <**sys/poll.h**> и <**asm/poll.h**>:

|  |  |
| --- | --- |
|  | #define POLLIN      0x0001    /\* Можно читать данные \*/  #define POLLPRI     0x0002    /\* Есть срочные данные \*/  #define POLLOUT     0x0004    /\* Запись не будет блокирована \*/  #define POLLERR     0x0008    /\* Произошла ошибка \*/  #define POLLHUP     0x0010    /\* Разрыв соединения \*/  #define POLLNVAL    0x0020    /\* Неверный запрос: fd не открыт \*/ |

Сам вызов **poll()**, объявленный в файле <**sys/poll.h**>, оперирует с массивом таких структур, где каждому дескриптору соответствует один элемент:

|  |  |
| --- | --- |
| 1 | int poll( struct pollfd \*ufds, unsigned int nfds, int timeout ); |

* **ufds** - сам массив структур;
* **nfds** - его размерность;
* **timeout** – величина тайм-аута в миллисекундах (ожидание при положительном значении, немедленный возврат при нулевом, бесконечное ожидание при значении, заданном константой **INFTIM**).

В листинге 2 представлены фрагменты приложений, использующий вызов **select()**. Пример демонстрирует фрагмент TCP сервера, относящийся к вызову **select()**.

##### Листинг 1. Пример ретранслирующего TCP-сервера на основе вызова select()

|  |  |
| --- | --- |
|  | ...     int                             nready, client[ FD\_SETSIZE ];     fd\_set                          rset, allset;     socklen\_t                       clilen;     struct sockaddr\_in      cliaddr, servaddr;     ...     listenfd = socket( AF\_INET, SOCK\_STREAM, 0 );     bzero( &servaddr, sizeof(servaddr) );     servaddr.sin\_family      = AF\_INET;     servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);     servaddr.sin\_port        = htons(SERV\_PORT);     bind( listenfd, (SA\*)&servaddr, sizeof(servaddr) );     listen( listenfd, LISTENQ );     maxfd = listenfd;                           /\* инициализация \*/     maxi = -1;     for( i = 0; i < FD\_SETSIZE; i++ )        client[i] = -1;        FD\_ZERO( &allset );        FD\_SET( listenfd, &allset );        for ( ; ; ) {           rset = allset;                        /\* присвоение структуры \*/           nready = select( maxfd + 1, &rset, NULL, NULL, NULL );           if( FD\_ISSET( listenfd, &rset ) ) {   /\* подключение нового клиента \*/           connfd = accept(listenfd, (SA \*) &cliaddr, &clilen);  ... |

Запуск сервера выполняется через консоль, а для остановки используется сочетание клавиш **Ctrl+C**. Сервер прослушивает фиксированный порт 9877 и ретранслируют клиенту данные, которые только что получил от него:

|  |  |
| --- | --- |
| 1  2  3 | $ ./tcpservselect01  ...  ^C |

Проверить то, что сервер прослушивает порт и готов к работе, можно следующим образом:

|  |  |
| --- | --- |
| 1  2 | $ netstat -a | grep :9877  tcp        0      0 \*:9877                      \*:\*                         LISTEN |

После запуска приложения в него можно вводить строки, которые будут передаваться на сервер и ретранслироваться обратно:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ ./tcpcli01 127.0.0.1  1 строка  1 строка  2 строка  2 строка  последняя  последняя  ^C |

При запуске клиента обязательно нужно указать IP адрес (а не имя) сервера. Оба сервера поддерживают работу с несколькими клиентами одновременно, и во время работы клиента можно увидеть состояние сокетов — клиентского и серверных, прослушивающего и присоединённого (клиент не закрывает соединение после обслуживания каждого запроса):

|  |  |
| --- | --- |
| 1  2  3  4 | $ netstat -a | grep :9877  tcp   0      0 \*:9877                  \*:\*                         LISTEN  tcp   0      0 localhost:46783         localhost:9877              ESTABLISHED  tcp   0      0 localhost:9877          localhost:46783             ESTABL  **API epoll** выполняет задачу, аналогичную poll (2): отслеживание нескольких файловых дескрипторов для определения возможности ввода-вывода для любого из них. API **epoll** хорошо масштабируется для большого количества дескрипторов просматриваемых файлов.   Центральная концепция API **epoll** это - экземпляр epoll, структура данных в ядре, которую с точки зрения пространства пользователя можно рассматривать как контейнер для двух списков:   1. список интересов (иногда также называемый набором epoll): набор файловых дескрипторов, в которых процесс зарегистрировал интерес к мониторингу; 2. список готовности: набор файловых дескрипторов, которые «готовы» для ввода / вывода. Список готовности представляет собой подмножество (или, точнее, набор ссылок) файловых дескрипторов в списке интересов, который динамически заполняется ядром в результате операций ввода-вывода с этими файловыми дескрипторами.    Для создания и управления экземпляром **epoll** предусмотрены следующие системные вызовы:   * int epoll\_create(int *size* ); или int epoll\_create1(int *flags*); - создает новый экземпляр epoll и возвращает дескриптор файла, ссылающийся на этот экземпляр. (Более поздняя версия epoll\_create1 (2) расширяет функциональность epoll\_create (2).) Начиная С Linux 2.6.8, аргумент *size* игнорируется, но должен быть больше нуля;   epoll\_create () возвращает файловый дескриптор, ссылающийся на новый экземпляр epoll. Этот файловый дескриптор используется для всех последующих обращений к epoll. Когда больше не требуется, дескриптор файла, возвращаемый epoll\_create (), должен быть закрыт с помощью close (2). Когда все файловые дескрипторы, ссылающиеся на экземпляр epoll, были закрыты, ядро уничтожает экземпляр и освобождает связанные ресурсы для повторного использования. epoll\_create1 () - если **flags** равен 0, то, кроме факта, что аргумент устаревшего размера отброшен, epoll\_create1 () совпадает с epoll\_create ().   * **int epoll\_ctl(int** *epfd***, int** *op***, int** *fd***, struct epoll\_event \****event***);** Этот системный вызов используется для добавления, изменения или удаления записей в списке интересов экземпляра epoll (7), на который ссылается дескриптор файла epfd. Он запрашивает выполнение операции op для дескриптора целевого файла fd. * **int epoll\_wait(int** *epfd***, struct epoll\_event \****events***,** **int** *maxevents***, int** *timeout***); или**  **int epoll\_pwait(int** *epfd***, struct epoll\_event \****events***,** **int** *maxevents***, int** *timeout***,** **const sigset\_t \****sigmask***);** Системный вызов epoll\_wait () ожидает события в экземпляре epoll (7), на которые ссылается дескриптор файла epfd. Область памяти, на которую указывают события, будет содержать события, которые будут доступны для звонящего. Функция epoll\_wait () возвращает до maxevents событий.   Аргумент maxevents должен быть больше нуля. Аргумент timeout указывает количество миллисекунд, которые epoll\_wait () будет блокировать. Время измеряется по часам CLOCK\_MONOTONIC.  Вызов будет заблокирован до тех пор, пока:  \* дескриптор файла доставляет событие;  \* вызов прерывается обработчиком сигнала;  или же  \* таймаут истекает.  *Листинг 2. Фрагмент кода сервера с системным вызовом epoll():*  int main() {  int listenSocket;  struct sockaddr\_in serverAddr;  char buffer[1024];  listenSocket = socket(AF\_INET, SOCK\_STREAM, 0);  serverAddr.sin\_family = AF\_INET;  serverAddr.sin\_port = htons(2345);  serverAddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");  memset(serverAddr.sin\_zero, 0, sizeof(serverAddr.sin\_zero));  bind(listenSocket, (struct sockaddr \*)&serverAddr, sizeof(serverAddr));  . . .  int efd = epoll\_create1(0);  struct epoll\_event event;  event.data.fd = listenSocket;  event.events = EPOLLIN | EPOLLET;  epoll\_ctl(efd, EPOLL\_CTL\_ADD, listenSocket, &event);  struct epoll\_event \*events = calloc(MAX\_EVENTS, sizeof(event));  while (1) {  int n = epoll\_wait (efd, events, MAX\_EVENTS, -1);  for (int i = 0; i < n; i++) {  if (events[i].data.fd == listenSocket) {  struct sockaddr in\_addr;  socklen\_t in\_len = sizeof(in\_addr);  int infd = accept (listenSocket, &in\_addr, &in\_len);  make\_socket\_non\_blocking (infd);  event.data.fd = infd;  event.events = EPOLLIN | EPOLLET;  epoll\_ctl(efd, EPOLL\_CTL\_ADD, infd, &event);  }  else { . . .}  . . .  return 0;  }  На рис.3 схематически изображена последовательность действий на стороне сервера при мультиплексировании. |

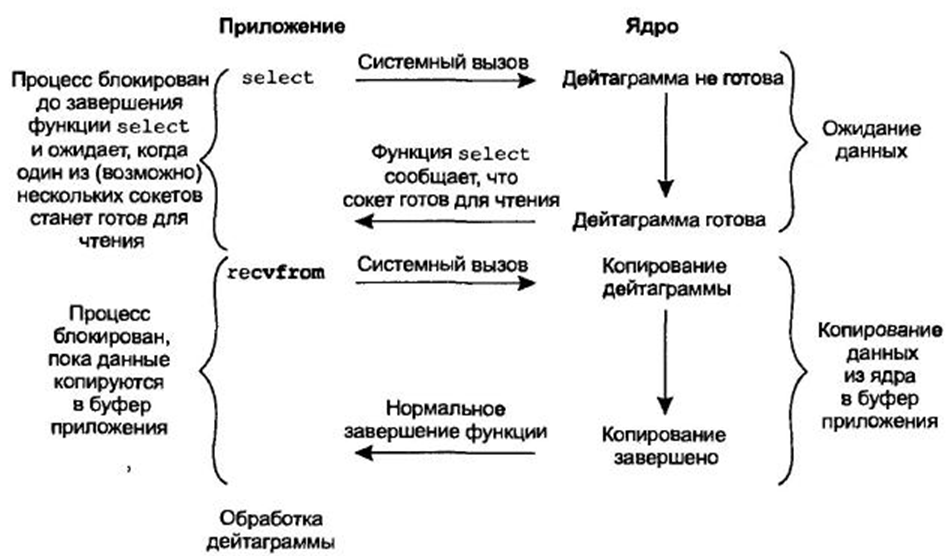


Рис.3

Приложение блокируется при вызове select, ожидая когда сокет станет доступным для чтения. Затем ядро возвращает приложению статус *readable,* сообщая, что можно получать данные помощью *recvfrom*. Снова блокировка, да еще вместо одного два системных вызова (select и recvfrom), что увеличивает накладные расходы. Но в отличии от рассмотренного ранее блокирующего метода, select или любой другой мультиплексор обеспечивает возможность ожидать данные не от одного, а от нескольких файловых дескрипторов, что, очевидно, снижает время блокировки (сна), как видно из поясняющего рисунка 4.

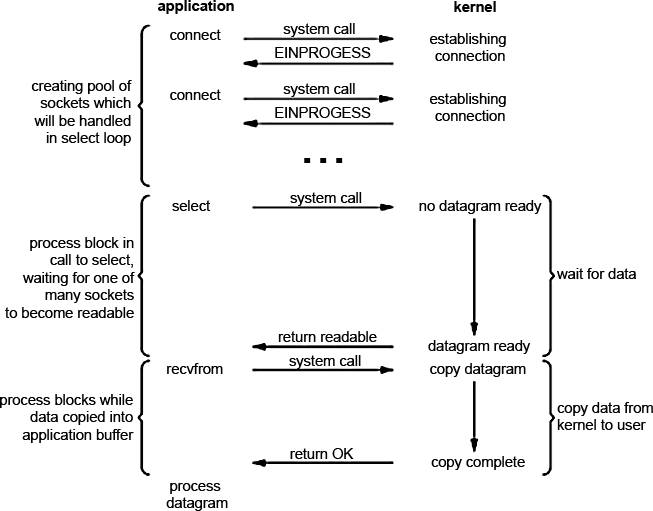


Рис.4

Клиенты пытаются создать соединение с сервером, вызывая системный вызов connect. В результате создается пул (pool) дескрипторов сокетов. Получение процессом-клиентом EINPROGRESS означает, что соединение устанавливается. На работу сервера получение клиентом EINPROGRESS никак не влияет, так как мультиплексор обработает первое соединение, которое произойдет: в цикле проверяются все сокеты и берется первый, который готов. Пока соединение принимается (accept), поступают другие соединения. Таким образом снижается время простоя, так как первый раз ожидание может затянуться, но последующие соединения требуют значительно меньших задержек.  
Ответ на следующий вопрос: «У какого события вероятность больше? Готовность к обмену данными определенного сокета или готовность данных у одного сокета из набора сокетов?» является очевидным.

**Несколько потоков или процессов для обработки запросов к серверу**

Этот способ называют способом +1. Создается несколько потоков или процессов, в каждом из которых выполняется блокирующий ввод-вывод. Он похож на мультиплексирование ввода/вывода, но при этом имеет ряд недостатков.

Процессы Unix/Linux создаются системным вызовом fork(). Любой процесс может создать любое число процессов. В этих ОС существуют развитые средства взаимоисключения. Создание для обслуживания каждого поступающего к серверу запроса отдельного процесса не устраняет проблемы блокировок процессов.

Модель потоков в Linux отличается от модели потоков в Windows. В Windows процесс является контейнером потоков и содержит как минимум один поток. В Linux каждый поток является процессом [4], и для того, чтобы создать новый поток, нужно создать новый процесс. В многопоточных приложениях Linux для создания дополнительных потоков используются специальные процессы – легковесные процессы (lightweight processes). В качестве преимущества потоков по сравнению с процессами можно отметить то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой вереницы зомби или «осиротевших» независимых процессов. Поэтому, говорить, что потоки Linux являются «дорогими» не корректно. При этом надо понимать, что если параллельные потоки выполняют одну и те же функцию, то она должна обладать свойством реентерабельности. Аналогично процессам потоки блокируются в ожидании завершения ввода-вывода или на событиях.

Если рассмотреть ситуацию, когда TCP-клиент обрабатывает два входных потока одновременно: стандартный поток ввода и сокет TCP. Клиент может быть блокирован в вызове функции fgets (чтение из стандартного потока ввода), а в это время процесс сервера мог быть уничтожен. TCP сервера корректно отправляет сегмент FIN протоколу TCP клиента, но поскольку процесс клиента блокирован при чтении из стандартного потока ввода, он не получит признак конца файла, пока не считает данные из сокета (возможно, значительно позже). Для решения этой проблемы нужно иметь возможность сообщить ядру, что приложение ждет получения уведомления о том, что выполняется одно или несколько условий для ввода-вывода (например, присутствуют данные для считывания или дескриптор готов к записи новых данных). Эта возможность называется *мультиплексированием* (multiplexing) ввода-вывода и обеспечивается функциями select и poll.

Мультиплексирование ввода-вывода обычно используется сетевыми приложениями в следующих случаях [1]:

* Когда клиент обрабатывает множество дескрипторов (обычно интерактивный ввод и сетевой сокет).
* Возможно, хотя это и редкий случай, когда клиент одновременно обрабатывает множество сокетов.
* Если сервер TCP обрабатывает и прослушиваемый сокет, и присоединенные сокеты.
* Если сервер работает и с TCP, и с UDP.
* Если сервер обрабатывает несколько служб и, возможно, несколько протоколов (например, демон inetd).

При выборе между многопоточностью и мультиплексированием следует предпочесть мультиплексирование в потоке, когда количество потоков зависит от количества обслуживаемых вами клиентов. Особенно, если имеется фиксированное количество потоков, запускаемых на одном компьютере.

## Модель ввода-вывода, управляемого сигналом

В этой модели используются сигналы. Когда данные готовы к считыванию ядро должно уведомить приложение и послать сигнал SIGIO.

При использовании данного способа на сетевом сокете необходимо установить режим ввода-вывода, управляемый сигналом и обработчик сигнала, используя системный вызов **sigaction()**.



Рис.5

Рис. 5, как и предыдущие схемы, демонстрирует особенности выполнения схемы взаимодействия по модели клиент-сервер. Особенностью модели ввода-вывода, управляемого сигналом, является то, что приложение не блокируется, а продолжает работать несмотря на то, что данные не готовы.

Результат выполнения системного вывода sigaction() возвращается сразу и приложение не блокируется. Всю работу берет на себя ядро:

* отслеживает готовность данных
* затем посылает сигнал SIGIO, который вызывает установленный на него обработчик (функция обратного вызова – callback).

Сам вызов recvfrom() может быть выполнен либо в обработчике сигнала, который отправляет основному циклу информацию, что данные готовы, либо в основном потоке программы. Ожидание может выполняться в цикле, который выполняется в основном потоке.

Сигнал SIGIO для каждого процесса может быть только один. В результате, за один раз можно обработать только один файловый дескриптор. Во время выполнения обработчика сигнала сигнал блокируется. Если за время блокировки сигнал доставляется несколько раз, то они теряются. Если маска сигнала sa\_mask = NULL, то во время выполнения обработчика другие сигналы не блокируются.

## Модель асинхронного ввода-вывода

Асинхронный ввод-вывод был добавлен только в редакции стандарта POSIX.1g в 1993 году, как одно из расширений реального времени. Спецификация согласовала различия в функциях реального времени, которые появились в разных стандартах, объединив их. В основном эти функции работают так, что сообщают ядру о начале операции и уведомляют приложение, когда вся операция, включая копирование данных из ядра в буфер приложения, завершена (рис. 6).

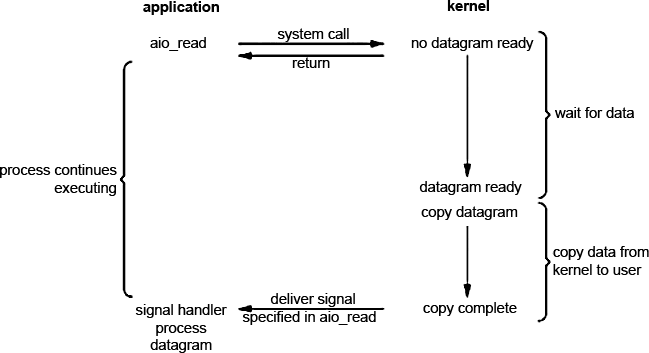


Рис.6

В вызове **aio\_read()**  даётся указание ядру начать операцию ввода-вывода, и указывается, каким сигналом уведомить процесс о завершении операции (включая копирование данных в пользовательский буфер). При этом вызывающий процесс не блокируется, а результат операции (например, полученная UDP-дейтаграмма) может быть обработан в обработчике сигнала. Разница с предыдущей моделью ввода-вывода, который управляется сигналом, состоит в том, что в предыдущей модели сигнал уведомляет о возможности **начала операции** (вызове операции чтения), а в асинхронной модели сигнал уведомляет уже о **завершении операции** копирования данных в буфер пользователя.

Вся функциональность, относящаяся к асинхронному вводу-выводу в Linux, описана в файле <**aio.h**>. Информация, необходимая для работы с асинхронным вводом-выводом, хранится в структуре aiocb (для 64-х битных операций существует аналогичная структура aiocb64):

|  |  |
| --- | --- |
|  | struct aiocb {     /\* структура управления асинхронным вводом-выводом.  \*/     int aio\_fildes;               /\* дескриптор файла  \*/     int aio\_lio\_opcode;           /\* выполняемая операция.  \*/     int aio\_reqprio;              /\* смещение  \*/     volatile void \*aio\_buf;       /\* адрес буфера.  \*/     size\_t aio\_nbytes;            /\* длина передаваемой последовательности  \*/  **struct sigevent aio\_sigevent; /\* номер и значение сигнала.  \*/**  ...  }  Важную роль в асинхронном вводе-выводе играют сигналы.  #include <signal.h>  union sigval { /\* Data passed with notification \*/  int sival\_int; /\* Integer value \*/  void \*sival\_ptr; /\* Pointer value \*/  };  struct **sigevent** {  int sigev\_notify; /\* Notification method \*/  **int** **sigev\_signo**; /\* Notification signal \*/  union sigval sigev\_value; /\* Data passed with  notification \*/  **void (\*sigev\_notify\_function) (union sigval);**  **/\* Function used for thread notification (SIGEV\_THREAD) \*/**  void \*sigev\_notify\_attributes;  /\* Attributes for notification thread  (SIGEV\_THREAD) \*/  pid\_t sigev\_notify\_thread\_id;  /\* ID of thread to signal (SIGEV\_THREAD\_ID) \*/  }; |

Некоторые функции асинхронного ввода-вывода:

|  |  |
| --- | --- |
|  | int aio\_read( struct aiocb \*\_\_aiocbp );  int aio\_write( struct aiocb \*\_\_aiocbp );  int lio\_listio( int \_\_mode, struct aiocb\* const list[ \_\_restrict\_arr ],                  int \_\_nent, struct sigevent \*\_\_restrict \_\_sig ) ;  int aio\_cancel( int \_\_fildes, struct aiocb \*\_\_aiocbp ); |

Параметры функций: файловый дескриптор, адрес буфера, номер сигнала **sigev\_signo** и функция, используемая для уведомления потока.

Функция lio\_listio() позволяет инициализировать выполнение целой цепочки асинхронных операций (длиной **\_\_nent**).

Как и в случае с потоками, асинхронные операции проще запустить, чем позже остановить, поэтому для их остановки предлагается специальная функция aio\_cancel().

Также можно предположить, что каждая асинхронная операция выполняется как отдельный поток.

В основном в асинхронном вводе-выводе внимание сосредотачивается на двух моментах:

1. на возможности определить, что ввод-вывод можно выполнить быстро;
2. на завершении операции ввода-вывода в любом случае: при невозможности выполнения ввода-вывода сразу возвращается ошибка.

Проблема асинхронного ввода-вывода состоит в том, что необходимо получать результаты асинхронных действий синхронно, т.е. асинхронные операции в итоге должны доставлять процессам данные, а процессы должны иметь возможность получать эти данные по необходимости.

***Классификация моделей ввода-вывода***

Операции ввода-вывода могут быть блокирующими и неблокирующими, синхронными и асинхронными (рис.7).



Рис.7

В соответствии с данной классификацией:

* блокирующий ввод-вывод – блокирующий синхронный;
* неблокирующий ввод-вывод или опрос (polling) – синхронный неблокирующий;
* мультиплексированный ввод-вывод – блокирующий асинхронный;
* асинхронный ввод-вывод – неблокирующий асинхронный

Ввод-вывод, управляемый сигналами, относится к неблокирующему асинхронному, но при получении сигнала о готовности данных вызывается блокирующий системный вызов.

Данная классификация может рассматриваться как избыточная, но она подчеркивает очень важные вещи.

Блокирующий синхронный ввод-вывод является наиболее обычным типом ввода-вывода: при обращении процесса к внешнему устройству с помощью системного вызова ввода-вывода процесс блокируется в ожидании завершения операции ввода-вывода; для информирования процессора о завершении операции ввода-вывода контроллер внешнего устройства формирует сигнал, который по линии IRQ№ поступает на контроллер прерывания; контроллер формирует сигнал прерывания, который по линии шины данных поступает на выделенный pin процессора; в конце цикла выполнения каждой команды процессор проверяет наличие сигнала прерывания и при его поступлении переходит на выполнение обработчика данного прерывания; обработчик прерывания должен сохранить данные, пришедшие от контроллера устройства, в буфере ядра, чтобы затем функции обратного вызова драйвера устройства доставили полученные данные в буфер приложения. Для этого приложение разблокируется.

Неблокирующем синхронном вводе-выводе приложение, выдавшее запрос ввода-вывода не блокируется, а делает многочисленные запросы, проверяя готовность данных.

Другая парадигма блокировки – это неблокирующий ввод-вывод с блокировкой уведомления. В этой модели устанавливается неблокирующий ввод-вывод, а затем вызывается блокирующий системный вызов мультиплексора (select или poll), чтобы определить какую-либо активность для файловых дескрипторов. Достоинством мультиплексора является то, что он используется для уведомления о многих дескрипторах.

Асинхронный ввод-вывод (AIO) - это метод выполнения операций ввода-вывода таким образом, что процесс, выдавший запрос ввода-вывода, не блокируется до завершения операции. Вместо этого после отправки запроса ввода-вывода процесс продолжает выполняться его код и можно позже проверить статус поданного запроса. Для завершения транзакции ввода-вывода может использоваться сигнал, на который устанавливается обработчик сигнала, или поток на основе обратного вызова.

**Блокирующий синхронный ввод-вывод**

Известный пример, приведенный в книге Шоу А. [5] демонстрирует блокирующий синхронный ввод-вывод. Процесс блокируется запросив ввод-вывод и пробуждается, когда операция ввода-вывода, выполняемая внешним устройством завершена, для получения результатов операции ввода-вывода.

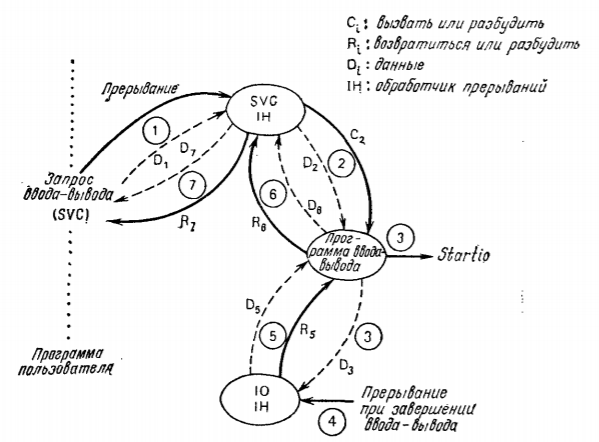
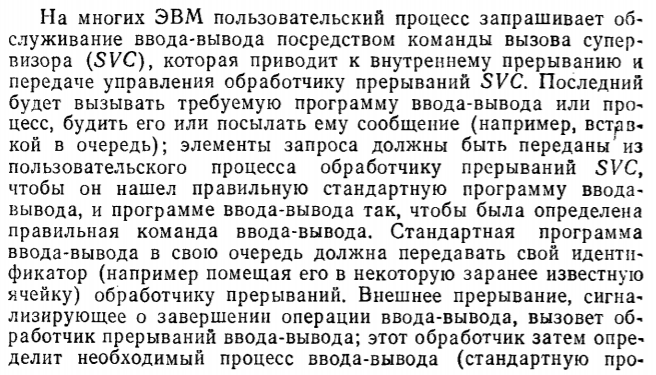
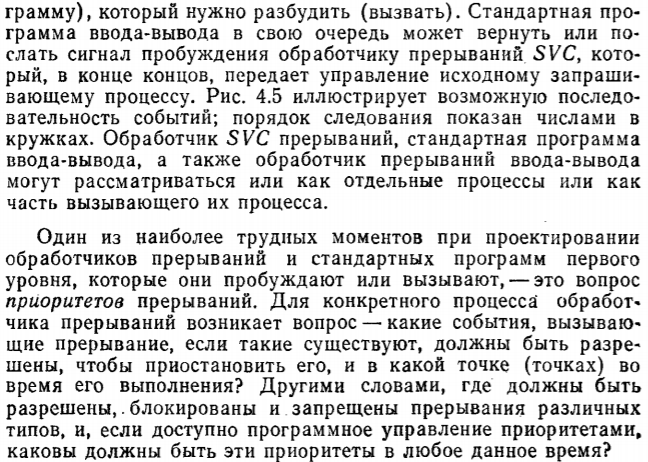
****

Рис.8 Запрос ввода-вывода приложения





Список используемых источников

1. W. Richard Stevens, Bill Fenner, Andrew M. Rudoff «Unix Network Programming Volume 1, Third edition: The sockets networking API»
2. Мультиплексированный ввод/вывод [Voltt](https://habr.com/ru/users/Voltt/) 6 января 2011 в 23:34 <https://habr.com/post/111357/>
3. Олег Цилюрик Инструменты ОС Linux для разработчиков приложений для ОС Windows. Часть 17. Расширенные операции ввода-вывода   
   Опубликовано 24.04.2014 https://www.ibm.com/developerworks/ru/library/linux\_windows\_17/
4. [Андрей Боровский](mailto:anb@symmetrica.net), [symmetrica.net](http://symmetrica.net/) Потоки Статья из серии "[Программирование для Linux](http://citforum.ru/programming/unix/borovsky/)", журнал [Linux Format](http://linuxformat.ru/) ([**citforum.ru**](http://citforum.ru/)›[Программирование](http://citforum.ru/programming)›[Программирование в Unix/Linux](http://citforum.ru/programming/unix)›[threads](http://citforum.ru/programming/unix/threads/)) <http://citforum.ru/programming/unix/threads/>
5. Шоу А. Логическое проектирование операционных систем. Пер. с англ. М.: Мир, 1981г. – 360 с.,ил.

1. В электронике: *мультипле́ксор* — устройство, имеющее несколько сигнальных входов, один или более управляющих входов и один выход. Схематически мультиплексор можно изобразить в виде коммутатора, обеспечивающего подключение одного из нескольких входов (их называют информационными) к одному выходу устройства. Коммутатор обслуживает управляющая схема, в которой имеются адресные входы и, как правило, разрешающие ([стробирующие](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D1%80%D0%BE%D0%B1&action=edit&redlink=1)). Сигналы на **адресных** входах определяют, какой конкретно информационный канал подключен к выходу.

   В вычислительных системах: *мультиплексор; коммутатор* – устройство, которое объединяет информацию, поступающую по нескольким каналам ввода, и выдает ее по одному выходному каналу; *мультиплексирование; уплотнение* – совмещение нескольких сообщений, передаваемых одновременно в одной физической или логической среде; существует два основных типа мультиплексирования: временное мультиплексирование и частотное мультиплексирование [↑](#footnote-ref-1)