**Лекция №5. Каналы.**

**Вопрос 1. Анонимные каналы.**

Вспомните синтаксис организации программных каналов при работе в командной строке *shell*:

#*cat myfile | wc*.

При этом (стандартный) вывод программы *cat*(1), которая выводит содержимое файла *myfile*, передается на (стандартный) ввод программы *wc*(1), которая, в свою очередь подсчитывает количество строк, слов и символов.

В результате мы получим что-то вроде 12 45 260, что будет означать количество строк, слов и символов в файле *myfile*.

Таким образом, два процесса обменялись данными. При этом использовался программный канал, обеспечивающий однонаправленную передачу данных между двумя задачами.

Для создания канала используется системный вызов pipe(2) int pipe(int\*filedes), который возвращает два файловых дескриптора – filedes[0] для записи вканал и filedes [1] для чтения из канала. Теперь, если один процесс записывает данные в filedes[0] другой сможет получить эти данные из filedes[1]. В случае неудачи вызов pipe вернет значение -1. Это может произойти, если в момент вызова произойдет превышение максимально возможного числа дескрипторов файлов, которые могут быть одновременно открыты процессами пользователя (в этом случае переменная errno будет содержать значение EMFILE), или если произойдет переполнение таблицы открытых файлов в ядре (в этом случае переменная errno будет содержать значение ENFILE).

Каналы не могут использоваться в качестве средства межпроцессного взаимодействия между независимыми процессами.

Хотя в приведенном примере может показаться, что процессы *cat*(1) и *wc*(1) независимы, на самом деле оба этих процесса создаются процессом *shell* и являются родственными (рис.1). После создания канала с ним можно работать просто при помощи вызовов *read* и *write*.

Обратите внимание, что сообщения считываются в том же порядке, в каком они были записаны. Каналы обращаются с данными в порядке «первый вошел/ первым вышел» (*first-in first-out*, или сокращенно *FIFO*). Другими словами, данные, которые помещаются в канал первыми, первыми и считываются на другом конце канала. Этот порядок нельзя изменить, поскольку вызов *lseek* не работает с каналами.

Размеры блоков при записи в канал и чтении из него необязательно должны быть одинаковыми, хотя в нашем примере это и было так. Можно, например, писать в канал блоками по 512 байт, а затем считывать из него по одному символу, так же, как и в случае обычного файла. Тем не менее, использование блоков фиксированного размера дает определенные преимущества.

Рис. 1. Создание канала между задачами *cat*(1) и *wc*(1)

Работа примера показана графически на рис. 2. Эта диаграмма позволяет более ясно представить, что процесс только посылает данные сам себе, используя канал в качестве некой разновидности механизма обратной связи. Это может показаться бессмысленным, поскольку процесс общается только сам с собой.



Рис. 2 Пример работы с каналами

Настоящее значение каналов проявляется при использовании вместе с системным вызовом *fork*, тогда можно воспользоваться тем фактом, что файловые дескрипторы остаются открытыми в обоих процессах.

На рис. 3 показано соединение двух процессов по каналу. Здесь видно, что и в родительском, и в дочернем процессах открыто по два дескриптора файла, позволяя выполнять запись в канал и чтение из него. Поэтому любой из процессов может выполнять запись в файл с дескриптором *р*(1) и чтение из файла с дескриптором *р*(0). Это создает определенную проблему. Каналы предназначены для использования в качестве однонаправленного средства связи. Если оба процесса будут одновременно выполнять чтение из канала и запись в него, то это приведет к путанице.

Чтобы избежать этого, каждый процесс должен выполнять либо чтение из канала, либо запись в него и закрывать дескриптор файла, как только он стал не нужен. Фактически программа должна выполнять это для того, чтобы избежать неприятностей, если посылающий данные процесс закроет дескриптор файла, открытого на запись.



Рис. 3 Пример работы с каналами

В конечном итоге получится однонаправленный поток данных от дочернего процесса к родительскому процессу. Данная упрощенная ситуация показана на рис. 4.



Рис. 4 Пример работы с каналами

**Закрытие каналов**. Что произойдет, если дескриптор файла, соответствующий одному из концов канала, будет закрыт?

Возможны два случая:

1. закрывается дескриптор файла, открытого только на запись. Если существуют другие процессы, в которых канал открыт на запись, то ничего не произойдет. Если же больше не существует процессов, которые могли бы выполнять запись в канал, и канал при этом пуст, то любой процесс, который попытается выполнить чтение из канала, получит пустой блок данных. Процессы, которые были приостановлены и ожидали чтения из канала, продолжат свою работу, вызовы read вер-

нут нулевое значение. Для процесса, выполняющего чтение, результат будет похож на достижение конца файла;

1. закрывается дескриптор файла, открытого только на чтение. Если еще есть процессы, в которых канал открыт на чтение, то снова ничего не произойдет. Если же больше не существует процессов, выполняющих чтение из канала, то ядро посылает всем процессам, ожидающим записи в канал, сигнал *SIGPIPE*. Если этот сигнал не перехватывается в процессе, то процесс при этом завершит свою работу. Если же сигнал перехватывается, то после завершения процедуры обработчика прерывания вызов *write* вернет значение -1, и переменная errno после этого будет содержать значение *EPIPE*. Процессам, которые будут пытаться после этого выполнить запись в канал, также будет посылаться сигнал *SIGPIPE*.

Вопрос 2. Именованные каналы.

Каналы являются изящным и мощным механизмом межпроцессного взаимодействия. Тем не менее, они имеют ряд недостатков.

*Первый*, и наиболее серьезный из них, заключается в том, что каналы могут использоваться только для связи процессов, имеющих общее происхождение, таких как родительский процесс и его потомок. Это ограничение становится очевидным при попытке разработать настоящую «серверную» программу, которая выполняется постоянно, обеспечивая системный сервис. Примерами таких программ являются серверы управления сетью и спулеры печати. В идеале клиентские процессы должны иметь возможность стартовать, подключаться к не связанному с ними серверному процессу при помощи канала, а затем снова отключаться от него. К сожалению, такую модель при помощи обычных каналов реализовать нельзя.

*Второй* недостаток каналов заключается в том, что они не могут существовать постоянно. Они каждый раз должны создаваться заново, а после завершения обращающегося к ним процесса уничтожаются.

Для восполнения этих недостатков существует разновидность канала, называемая именованным каналом, или файлом типа *FIFO* (сокращение от *firstin first-out*, то есть «первый вошел/первым вышел»). В отношении вызовов *read* и *write* именованные каналы идентичны обычным. Тем не менее, в отличие от обычных каналов, именованные каналы являются постоянными и им присвоено имя файла системы *Unix*. Именованный канал также имеет владельца, размер и связанные с ним права доступа. Он может быть открыт, закрыт и удален, как и любой файл *Unix*, но при чтении или записи ведет себя аналогично каналу.

Прежде чем рассматривать применение каналов *FIFO* на программном уровне, рассмотрим их использование на уровне команд. Для создания именованного канала используется команда *mknod*: */etc/mknod channel p*.

Первый аргумент channel является именем канала *FIFO* (в качестве него можно задать любое допустимое имя *Unix*). Параметр *р* команды *mknod* указывает, что нужно создать именованный канал. Этот параметр необходим, так как команда *mknod* также используется для создания файлов устройств.

Некоторые атрибуты вновь созданного канала *FIFO* можно вывести при помощи команды *ls*:

*ls -l channel*;

*prw-rw-r1 ben usr 0 Aug 1 21: 05 channel*.

Символ *р* в первой колонке обозначает, что channel является файлом типа *FIFO*. Обратите внимание на права доступа к именованному каналу *channel* (чтение/запись для владельца и группы владельца, только чтение для всех остальных пользователей); владельца и группу владельца (*ben, usr*); размер (0 байт, то есть в настоящий момент канал пуст) и время создания.

При помощи стандартных команд *Unix* можно выполнять чтение из канала *FIFO* и запись в него, например: *cat < channel.*

Если выполнить эту команду сразу же после создания именованного канала *channel*, то она «зависнет». Это происходит из-за того, что процесс, открывающий канал *FIFO* на чтение, по умолчанию будет блокирован до тех пор, пока другой процесс не попытается открыть канал *FIFO* для записи. Аналогично процесс, пытающийся открыть канал *FIFO* для записи, будет блокирован до тех пор, пока другой процесс не попытается открыть его для чтения. Это благоразумный подход, так как он экономит системные ресурсы и облегчает координацию работы программы. Вследствие этого, при необходимости создания одновременно как записывающего, так и читающего процессов, потребуется запустить один из них в фоновом режиме.

Программирование при помощи каналов *FIFO*, в основном, идентично программированию с использованием обычных каналов. Единственное существенное различие заключается в их инициализации. Вместо использования вызова *pipe* канал *FIFO* создается при помощи вызова *mkfifо*. В старых версиях *Unix* может потребоваться использование более общего вызова *mknod*:

Описание

#*include <sys/types. h>*

#*include <sys/stat. h*>

*int mkfifo*(*const char \*pathname, mode\_t mode*);

Системный вызов *mkfifо* создает файл *FIFO* с именем, заданным первым параметром *pathname*. Канал *FIFO* будет иметь права доступа, заданные параметром *mode* и измененные в соответствии со значением *umask* процесса.

После создания канала *FIFO* он должен быть открыт при помощи вызова

*open*. Поэтому, например, фрагмент кода

#*include <sys/types. h> #include <sys/stat. h>*

*#include <fcntl. h> mkfifo("/tmp/fifo", 0666);*

*fd = open("/tmp/fifo", 0\_WRONLY*);

открывает канал FIFO для записи. Вызов *open* будет заблокирован до тех пор, пока другой процесс не откроет канал *FIFO* для чтения (конечно же, если канал *FIFO* уже был открыт для чтения, то возврат из вызова *open* произойдет немедленно).

Можно выполнить не блокирующий вызов open для канала *FIFO*. Для этого во время вызова должен быть установлен флаг *O\_NONBLOCK* (определенный в файле <*fcntl. h*>) и один из флагов *O\_RDONLY* или *O\_WRONLY*, например:

*if*((*fd = open*("/*tmp/fifo*", *O\_WRONLY | O\_NONBLOCK*)) == -1)

*perror*("Ошибка вызова *open* для канала *FIFO*").

Если не существует процесс, в котором канал *FIFO* открыт для чтения, то этот вызов *open* вернет значение -1 вместо блокировки выполнения, а переменная *errno* будет содержать значение *ENXIO*. В случае же успешного вызова *open* последующие вызовы *write* для канала *FIFO* также будут не блокирующими.

Обратите внимание на то, что канал *FIFO* открывается одновременно для чтения и записи (при помощи задания флага *0\_RDWR*). Чтобы понять, для чего это сделано, предположим, что канал *FIFO* был открыт только для чтения при помощи задания флага *0\_RDONLY*. Тогда выполнение программы rcvmessage будет сразу заблокировано в момент вызова *open*. Когда после старта программы *sendmessage* в канал *FIFO* будет произведена запись, вызов *open* будет разблокирован, программа *rcvmessage* будет читать все посылаемые сообщения. Когда же канал *FIFO* станет пустым, а процесс *sendmessage* завершит работу, вызов *read* начнет возвращать нулевое значение, так как канал *FIFO* уже не будет открыт на запись ни в одном процессе. При этом программа *rcvmessage* войдет в бесконечный цикл.

Использование флага *0\_RDWR* позволяет гарантировать, что, по крайней мере, в одном процессе, то есть самом процессе программы *rcvmessage*, канал *FIFO* будет открыт для записи. В результате вызов *open* всегда будет блокироваться то тех пор, пока в канал *FIFO* снова не будут записаны данные.

Следующий диалог показывает, как можно использовать эти две программы. Программа *rcvmessage* выполняется в фоновом режиме для получения сообщений от разных процессов, выполняющих программу *sendmessage*.

Вопрос 3. API каналов

Для простых приложений применение неблокирующих операций чтения и записи работает прекрасно. Для работы с множеством каналов одновременно существует другое решение, которое заключается в использовании системного вызова *select*.

Представьте ситуацию, когда родительский процесс выступает в качестве серверного процесса и может иметь произвольное число связанных с ним клиентских (дочерних) процессов. В этом случае серверный процесс должен как-то справляться с ситуацией, когда одновременно в нескольких каналах может находиться информация, ожидающая обработки. Кроме того, если ни в одном из каналов нет ожидающих данных, то может иметь смысл приостановить работу серверного процесса до их появления, а не опрашивать постоянно каналы. Если информация поступает более чем по одному каналу, то серверный процесс должен знать обо всех таких каналах для того, чтобы работать с ними в правильном порядке (например, согласно их приоритетам).

Это можно сделать при помощи системного вызова *select* (существует также аналогичный вызов *poll*). Системный вызов *select* используется не только для каналов, но и для обычных файлов, терминальных устройств, именованных каналов и сокетов. Системный вызов *select* показывает, какие дескрипторы файлов из заданных наборов готовы для чтения, записи или ожидают обработки ошибок. Иногда серверный процесс не должен совсем прекращать работу, даже если не происходит никаких событий, поэтому в вызове *select* также можно задать предельное время ожидания:

Описание

#*include <sys/time.h*>

*int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*errorfds, struct timeval \*timeout*).

*Первый* параметр *nfds* задает число дескрипторов файлов, которые могут представлять интерес для сервера.

Например, если дескрипторы файлов с номерами 0, 1 и 2 присвоены потокам *stdin, stdout* и *stderr* соответственно, и открыты еще два файла с дескрипторами 3 и 4, то можно присвоить параметру *nfds* значение 5.

Программист может определять это значение самостоятельно или воспользоваться постоянной *FD\_SETSIZE*, которая определена в файле <*sys/time.h*>. Значение постоянной *FD\_SETSIZE* равно максимальному числу дескрипторов файлов, которые могут быть использованы вызовом *select*.

*Второй, третий* и *четвертый* параметры вызова select являются указателями на битовые маски (*bit mask*), в которых каждый бит, соответствует дескриптору файла.

Если бит включен, то это обозначает интерес к соответствующему дескриптору файла:

* + набор *readfs* определяет дескрипторы, для которых сервер ожидает возможности чтения;
	+ набор *writefs* – дескрипторы, для которых ожидается возможность выполнить запись;
	+ набор *errorfds* определяет дескрипторы, для которых сервер ожидает появление ошибки или исключительной ситуации, например, по сетевому соединению могут поступить внеочередные данные.

Так как работа с битами довольно неприятна и приводит к немобильности программ, существует абстрактный тип данных *fd\_set*, а также макросы или функции (в зависимости от конкретной реализации системы) для работы с объектами этого типа.