Лабораторная работа №3

Организация взаимодействия процессов через pipe и FIFO в UNIX

Цель: *ознакомиться с особенностями взаимодействия процессов в операционной системе UNIX при помощи инструментов pipe и FIFO.*

1. Теоретический блок

Понятие о потоке ввода-вывода

Среди всех категорий средств коммуникации наиболее употребительными являются каналы связи, обеспечивающие достаточно безопасное и достаточно информативное взаимодействие процессов.

Существует две модели передачи данных по каналам связи — поток ввода-вывода и сообщения. Из них более простой является потоковая модель, в которой операции передачи/приема информации вообще не интересуются содержимым того, что передается или принимается. Вся информация в канале связи рассматривается как непрерывный поток байт, не обладающий никакой внутренней структурой. Изучению механизмов, обеспечивающих потоковую передачу данных в операционной системе UNIX, и будет посвящен этот семинар.

Понятие о работе с файлами через системные вызовы и стандартную библиотеку ввода-вывода для языка C

Потоковая передача информации может осуществляться не только между процессами, но и между процессом и устройством ввода-вывода, например между процессом и диском, на котором данные представляются в виде файла. Поскольку понятие файла должно быть знакомо изучающим этот курс, а системные вызовы, использующиеся для потоковой работы с файлом, во многом соответствуют системным вызовам, применяемым для потокового общения процессов, мы начнем наше рассмотрение именно с механизма потокового обмена между процессом и файлом.

Как мы надеемся, из курса программирования на языке *C* вам известны функции работы с файлами из стандартной библиотеки ввода-вывода, такие как

fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets() и т. д. Эти функции входят как неотъемлемая часть в стандарт ANSI на язык *C* и позволяют программисту получать информацию из файла или записывать ее в файл при условии, что программист обладает определенными знаниями о содержимом передаваемых данных. Так, например, функция *fgets()* используется для ввода из файла последовательности символов, заканчивающейся символом *'\n* — перевод каретки. Функция *fscanf()* производит ввод информации, соответствующей заданному формату, и т.д. С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных.

В операционной системе UNIX эти функции представляют собой надстройку — сервисный интерфейс — над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не требующими никаких знаний о том, что она содержит. Чуть позже мы кратко познакомимся с системными вызовами *open(), read(), write()* и *close()*, которые применяются для такого обмена, но сначала нам нужно ввести еще одно понятие - понятие *файлового дескриптора.*

Файловый дескриптор

Информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления — PCB. В операционной системе UNIX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное

число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода, файловый дескриптор 1 — стандартному потоку вывода, файловый дескриптор 2 — стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок — с текущим терминалом.

Открытие файла. Системный вызов open()

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому прежде чем совершать операции чтения данных из файла и записи их в файл, мы должны поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом *open()*.

Прототип системного вызова

#include <fcntl.h>

*int open(char *path, int flags);*

*int open(char *path, int flags, int mode);*

Описание системного вызова

Системный вызов *open* предназначен для выполнения операции открытия файла и, в случае ее удачного осуществления, возвращает файловый дескриптор открытого файла (небольшое неотрицательное целое число, которое используется в дальнейшем для других операций с этим файлом).

Параметр *path* является указателем на строку, содержащую полное или относительное имя файла.

Параметр *flags* может принимать одно из следующих трех значений:

O_RDONLY - если над файлом в дальнейшем будут совершаться только операции чтения;

O_WRONLY - если над файлом в дальнейшем будут осуществляться только операции записи;

O_RDWR - если над файлом будут осуществляться и операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции «побитовое или (|) » с одним или несколькими флагами:

O_CREAT - если файла с указанным именем не существует, он должен быть создан;

O_EXCL - применяется совместно с флагом *O_CREAT*. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;

O_NDELAY - запрещает перевод процесса в состояние блокировки при выполнении операции открытия и любых последующих операциях над этим файлом;

O_APPEND - при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;

O_TRUNC - если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации. Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

O_SYNC - любая операция записи в файл будет блокироваться до тех пор, пока записанная информация не будет физически помещена на соответсвующий нижестоящий уровень hardware;

O_NOCTTY - если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр *MODE* устанавливает атрибуты прав доступа различных категорий пользователей к новому файлу при его создании. Он обязателен, если среди заданных

флагов присутствует флаг *O_CREAT*, и может быть опущен в противном случае. Этот параметр задается как сумма следующих восьмеричных значений:

0400 - разрешено чтение для пользователя, создавшего файл;

0200 - разрешена запись для пользователя, создавшего файл;

0100 - разрешено исполнение для пользователя, создавшего файл;

0040 - разрешено чтение для группы пользователя, создавшего файл;

0020 - разрешена запись для группы пользователя, создавшего файл;

0010 - разрешено исполнение для группы пользователя, создавшего файл;

0004 - разрешено чтение для всех остальных пользователей;

0002 - разрешена запись для всех остальных пользователей;

0001 - разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра *mode* и маски создания файлов текущего процесса *umask*, а именно - они равны *mode & ~umask*.

При открытии файлов типа FIFO системный вызов имеет некоторые особенности поведения по сравнению с открытием файлов других типов, Если FIFO открывается только для чтения, и не задан флаг *O_NDELAY*, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг *O_NDELAY* задан, то извращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и не задан флаг *O_NDELAY*, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг *O_NDELAY* задан, то констатируется возникновение ошибки и возвращается значение -1.

Возвращаемое значение

Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов *open()* использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент

открытия файла. Из всего возможного набора флагов на текущем уровне знаний нас будут интересовать только флаги *O_RDONLY, O_WRONLY, O_RDWR, O_CREAT* и *O_EXCL*. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись. Как вам известно, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае, когда мы допускаем, что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом *O_CREAT*. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным в параметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в Linux).

В случае, когда мы требуем, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами *O_CREAT* и *O_EXCL*.

Системные вызовы read(), write(), close()

Для совершения потоковых операций чтения информации из файла и ее записи в файл применяются системные вызовы *read()* и *write()*.

Прототипы системных вызовов

```
#include <sys/types.h>
#include <unistd.h>
size_t readfint fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Описание системных вызовов

Системные вызовы *read* и *write* предназначены для осуществления потоковых операций ввода (чтения) и вывода (записи) информации над каналами связи, описываемыми файловыми дескрипторами, т.е. для файлов, pipe, FIFO и socket.

Параметр *fd* является файловым дескриптором созданного ранее потокового канала связи, через который будет отсылаться или получаться информация, т. е. значением, которое вернул один из системных вызовов *open()*, *pipe()* или *socket()*.

Параметр *addr* представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.

Параметр *nbytes* для системного вызова *write* определяет количество байт, которое должно быть передано, начиная с адреса памяти *addr*. Параметр *nbytes* для системного вызова *read* определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса *addr*.

Возвращаемые значения

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Заметим, что это значение (большее или равное 0) может не совпадать с заданным значением параметра *nbytes*, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами

При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов *read* возвращает значение 0, то это означает, что файл прочитан до конца.

Мы сейчас не акцентируем внимание на понятии указателя текущей позиции в файле и взаимном влиянии значения этого указателя и поведения системных вызовов.

После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов *close()*. Надо отметить, что при завершении работы процесса с помощью явного или неявного вызова функции *exit()* происходит автоматическое закрытие всех открытых потоков ввода-вывода.

Прототип системного вызова

#include <unistd.h>

int close(int fd);

Описание системного вызова

Системный вызов *close* предназначен для корректного завершения работы с файлами и другими объектами ввода-вывода, которые описываются в операционной системе через файловые дескрипторы: pipe, FIFO, socket.

Параметр *fd* является дескриптором соответствующего объекта, т. е. значением, которое вернул один из системных вызовов *open(), pipe()* или *socket()*.

Возвращаемые значения

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Для иллюстрации сказанного давайте рассмотрим следующую программу:

```
/* Иллюстрация использования
системных вызовов open(), write() и close() для
записи информации в файл*/
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
int fd;
size_t size;
char string[] = "Hello, world!";
/* Обнуляем маску создания файлов текущего процесса для того, чтобы
права доступа у создаваемого файла точно соответствовали параметру вызова
open() */
(void)umask(0);
/* Попытаемся открыть файл с именем myfile в текущей директории
только для операций вывода. Если файла не существует, попробуем его создать
с правами доступа 0666, т. е. read-write для всех категорий пользователей */
if((fd = open("myfile", 0_WRONLY | 0_CREAT, 0666)) < 0) {
/* Если файл открыть не удалось, печатаем об этом сообщение и
прекращаем работу */
printf("Can\'t open file\n");
exit(-1) ;
}
/* Пробуем записать в файл 14 байт из нашего массива, т.е. всю строку
"Hello, world!" вместе с признаком конца строки */
size = write(fd, string, 14);
if(size != 14){
```

```
/* Если записалось меньшее количество байт,
сообщаем об ошибке */
printf("Can\'t write all  string\n");
exit(-1);
/*  Закрываем файл */
 if (close (fd)- < 0) {
printf("Can\'t  close file\n");
}
return 0 ;
}
```

Обратите внимание на использование системного вызова *umask()* с параметром 0 для того, чтобы права доступа к созданному файлу точно соответствовали указанным в системном вызове *open()*.

Понятие о pipe. Системный вызов pipe()

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является *pipe* (канал, труба, конвейер).

Важное отличие pip'a от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности pipe представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра

такого кольцевого буфера внутри операционной системы используется системный вызов *pipe()*.

Прототип системного вызова

#include <unistd.h>

*int pipe(int *fd);*

Описание системного вызова

Системный вызов *pipe* предназначен для создания pip'a внутри операционной системы.

Параметр *fd* является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива - *fd[0]* - будет занесен файловый дескриптор, соответствующий выходному потоку данных pip'a и позволяющий выполнять только операцию чтения, а во второй элемент массива - *fd[1]* - будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Возвращаемые значения

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибок.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым pip'ом два файловых дескриптора. Для одного из них разрешена только операция чтения из pip'a, а для другого — только операция записи в pipe. Для выполнения этих операций мы можем использовать те же самые системные вызовы *read()* и *write()*, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова *close()* для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие pipe, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует pipe. Таким образом,

время существования pip'a в системе не может превышать время жизни процессов, работающих с ним.

Достаточно яркой иллюстрацией действий по созданию pip'a, записи в него данных, чтению из него и освобождению выделенных ресурсов может служить программа, организующая работу с pip'ом в рамках одного процесса, приведенная ниже.

```
/* Иллюстрация работы с pip'ом
в рамках одного процесса  */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
int  fd[2];
size_t size;
char string[]   =  "Hello,  world!";
char resstring[14];
 /* Попытаемся создать pipe  */
if(pipe(fd)   <  0){
/*  Если создать pipe не удалось,  печатаем об
этом сообщение и прекращаем работу  */
printf("Can\'t  create pipe\n");
exit(-1);
}
/* Пробуем записать в pipe 14  байт из нашего массива,  т.е.  всю строку
"Hello,  world!"  вместе с признаком конца строки */
size = write(fd[l],   string,   14);
if(size   !=  14 ) {
/* Если записалось меньшее количество байт,
сообщаем об ошибке */
printf("Can\'t write all  string\n");
```

```
exit(-1);
}
/* Пробуем прочитать из pip'a 14 байт в другой массив, т.е. всю записанную строку */
size = read(fd[0], resstring, 14);
if(size < 0){
/* Если прочитать не смогли, сообщаем об ошибке */
printf("Can\'t read string\n");
exit(-1) ;
}
/* Печатаем прочитанную строку */
print f("%s\n",resstring);
/* Закрываем входной поток*/
if(close(fd[0]) < 0){
printf("Can\'t close input stream\n");
/* Закрываем выходной поток*/
if (close(fd[l]) < 0){
printf("Can\'t close output stream\n");
}
return 0;
}
```

Организация связи через pipe между процессом-родителем и процессом-потомком. Наследование файловых дескрипторов при вызовах fork() и exec()

Понятно, что если бы все достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через операционную систему, то овчинка не стоила бы выделки. Однако таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом *fork()* и входит в состав неизменяемой части системного контекста процесса при системном вызове *exec()*

(за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении *exec()*, однако их рассмотрение выходит за рамки нашего курса). Это обстоятельство позволяет организовать передачу информации через pipe между родственными процессами, имеющими общего прародителя, создавшего pipe.

Давайте рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком:

```
/* Осуществление однонаправленной связи через pipe между процессом-родителем и процессом-ребенком */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main{){
int  fd[2],   result;
size_t  size;
char resstring[14];
/* Попытаемся создать pipe  */
if(pipe(fd)   <  0) {
/* Если создать pipe не удалось,  печатаем об
этом сообщение и прекращаем работу */
printf{"Can\'t create pipe\n");
exit(-1);
}
/* Порождаем новый процесс  */
result = fork();
if(result){
/*  Если создать процесс не удалось,   сообщаем об этом и завершаем работу */
printf("Can\'t  fork child\n");
```

```
exit(-1);
 } else if (result > 0) {
/* Мы находимся в родительском процессе, который будет передавать информацию процессу-ребенку. В этом процессе выходной поток данных нам не понадобится, поэтому закрываем его.*/
close(fd[0]);
/* Пробуем записать в pipe 14 байт, т.е. всю строку "Hello, world!" вместе с признаком конца строки */
size = write(fd[l], "Hello, world!", 14);
if(size != 14) {
/* Если записалось меньшее количество байт, сообщаем об ошибке и завершаем работу */
printf("Can\'t write all string\n"); exit(-1);
}
/* Закрываем входной поток данных, на этом родитель прекращает работу */
close(fd[l]);
printf("Parent exit\n");
} else {
/* Мы находимся в порожденном процессе, который будет получать информацию от процесса-родителя. Он унаследовал от родителя таблицу открытых файлов и, зная файловые дескрипторы, соответствующие pip'у, может его использовать. В этом процессе входной поток данных нам не понадобится, поэтому закрываем его.*/
close(fd[l]);
/* Пробуем прочитать из pip'a 14 байт в массив,
т.е. всю записанную строку */
size = read(fd[0], resstring, 14);
iffsize < 0){
```

```
/*  Если прочитать не смогли,   сообщаем об ошибке и завершаем работу
*/
printf("Can\'t  read string\n"); exi t(-1);
}
/*  Печатаем прочитанную строку  */
printf("%s\n",resstring);
/* Закрываем входной поток и завершаем работу */ close(fd[0] ) ;
}
return 0;
}
```

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через pipe двустороннюю связь, когда процесс-родитель пишет информацию в pipe, предполагая, что ее получит процесс-ребенок, а затем читает информацию из pipe, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного pipe в двух направлениях необходимы специальные средства синхронизации процессов, о которых речь идет в лекциях «Алгоритмы синхронизации» и «Механизмы синхронизации». Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris 2) реализованы полностью дуплексные pip'ы. В таких системах для обоих файловых дескрипторов, ассоциированных с pipe, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для pip'ов и не является переносимым.

Особенности поведения вызовов read() и write() для pip'а

Системные вызовы *read()* и *write()* имеют определенные особенности поведения при работе с pipe, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов. Организация запрета блокирования этих вызовов для pipe выходит за рамки нашего курса.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. Помните, что за один раз из pipe может прочитаться меньше информации, чем вы запрашивали, и за один раз в pipe может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами! Одна из особенностей поведения блокирующегося системного вызова *read()* связана с попыткой чтения из пустого pipe. Если есть процессы, у которых этот pipe открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости закрытия файлового дескриптора, ассоциированного с входным концом pipe, в процессе, который будет использовать pipe для чтения (*close (fd [1]*) в процессе-ребенке в программе из раздела «Прогон программы для организации однонаправленной связи между родственными процессами через pipe». Аналогичной особенностью поведения при отсутствии процессов, у которых pipe открыт для чтения, обладает и системный вызов *write()*, с чем связана необходимость закрытия файлового дескриптора, ассоциированного с выходным концом pipe, в процессе, который будет использовать pipe для записи (*close(f d[0]*) в процессе-родителе в той же программе).

Системные вызовы *read* и *write* Особенности поведения при работе с pipe, FIFO и socket Системный вызов *read*	
Ситуация	Поведение
Попытка прочитать меньше байт, чем есть в наличии в канале связи.	Читает требуемое количество байт и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.

В канале связи находится меньше байт, чем затребовано, но не нулевое количество.	Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена.	Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение 0. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение 0.
Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена.	Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение *EAGAIN*. Если таких процессов нет, системный вызов возвращает значение 0.
Системный вызов *write*	
Ситуация	Поведение
Попытка записать в канал связи меньше байт, чем осталось до его заполнения.	Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.
Попытка записать в канал связи больше	Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи

байт, чем осталось до его заполнения. Блокировка вызова разрешена.	меньше, чем передаваемое количество информации, то вызов будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт.
Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена.	Системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение *EAGAIN*.
В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена.	Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт.
Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена.	Системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение *EAGAIN*.
Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова.	Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал *SIGPIPE*. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную *errno* в значение *EPIPE*.
Необходимо отметить дополнительную особенность системного вызова write при работе с pap'ами и FIFO. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно - одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.	

Понятие FIFO. Использование системного вызова mknod() для создания FIFO. Функция mkfifo()

Как мы выяснили, доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов-потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов *pipe()*, или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе UNIX существует возможность использования pip'a для взаимодействия других процессов, но ее реализация достаточно сложна и лежит далеко за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от First Input First Output) или именованный pipe. FIFO во всем подобен pip'у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip'a на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов *mknod()* или существующая в некоторых версиях UNIX функция *mkfifo()*. Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам ситемного вызова *open()* . После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы *read()*, *write()* и *close()*. Время существования FIFO в адресном пространстве ядра

операционной системы, как и в случае с pip'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Прототип системного вызова *mknod*

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Описание системного вызова

Нашей целью является не полное описание системного вызова *mknod*, а только описание его использования для создания FIFO. Поэтому мы будем рассматривать не все возможные варианты задания параметров, а только те из них, которые соответствуют этой специфической деятельности.

Параметр *dev* является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0.

Параметр *path* является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно.

Параметр *mode* устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как результат побитовой операции «или» значения *S_FIFO*, указывающего, что системный вызов должен создать FIFO, и некоторой суммы следующих восьмеричных значений:

0400 - разрешено чтение для пользователя, создавшего FIFO;

0200 - разрешена запись для пользователя, создавшего FIFO;

0040 - разрешено чтение для группы пользователя, создавшего FIFO;

0020 - разрешена запись для группы пользователя, создавшего FIFO;

0004 - разрешено чтение для всех остальных пользователей;

0002 - разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно -они равны *(0777 & mode) &~umask*.

Возвращаемые значения

При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном - отрицательное значение.

Прототип функции *mkfifo*

#include <sys/stat.h>

#include <unistd.h>

*int mkfifo(char *path, int mode);*

Описание функции

функция *mkfifo* предназначена для создания FIFO в операционной системе. Параметр *path* является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.

Параметр *mode* устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

0400 - разрешено чтение для пользователя, создавшего FIFO;

0200 - разрешена запись для пользователя, создавшего FIFO:

0040 - разрешено чтение для группы пользователя, создавшего FIFO;

0020 - разрешена запись для группы пользователя, создавшего FIFO;

0004 - разрешено чтение для всех остальных пользователей;

0002 - разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно - они равны *(0777 & mode) & ~umask*.

Возвращаемые значения

При успешном создании FIFO - функция возвращает значение 0, при неуспешном - отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения.

Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Особенности поведения вызова open() при открытии FIFO

Системные вызовы *read()* и *write()* при работе с FIFO имеют те же особенности поведения, что и при работе с pip'ом. Системный вызов *open()* при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и флаг *O_NDELAY* не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг *O_NDELAY* задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и флаг *O_NDELAY* не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг *O_NDELAY* задан, то констатируется возникновение ошибки и возвращается значение -1. Задание флага *O_NDELAY* в параметрах системного вызова *open()* приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

Для иллюстрации взаимодействия процессов через FIFO рассмотрим такую программу:

/ Осуществление однонаправленной связи через FIFO между процессом-родителем и процессом-ребенком */*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
int fd, result;
size_t size;
char resstring[14];
char name[]="aaa.fifo";
/* Обнуляем маску создания файлов текущего процесса для того, чтобы
права доступа у создаваемого FIFO точно соответствовали параметру вызова
mknod() */
(void)umask(0) ;
/* Попытаемся создать FIFO с именем aaa.fifo в текущей директории */
if(mknod(name, S_IFIFO I 0666, 0) < 0){
/* Если создать FIFO не удалось, печатаем об этом сообщение и
прекращаем работу */
printf("Can\'t create FIFO\n"); exit(-1);
}
/* Порождаем новый процесс */
if((result = fork()) < 0) {
/* Если создать процесс не удалось, сообщаем об этом и завершаем
работу */
printf("Can\'t fork child\n"); exit(-1); } else if (result > 0) {
/* Мы находимся в родительском процессе, который будет передавать
информацию процессу-ребенку. В этом процессе открываем FIFO на запись.*/
if((fd = open(name, 0_WRONLY)) < 0){
/* Если открыть FIFO не удалось, печатаем об этом сообщение и
прекращаем работу */
```

printf("Can\'t open FIFO for writing\n");

exit(-1);

}

/ Пробуем записать в FIFO 14 байт, т.е. всю строку "Hello, world!" вместе с признаком конца строки */*

size = write(fd, "Hello, world!", 14); if(size != 14) {

/ Если записалось меньшее количество байт, то сообщаем об ошибке и завершаем работу */*

printf("Can\'t write all string to FIFO\n");

exit(-1);

}

/ Закрываем входной поток данных и на этом родитель прекращает работу */*

close(fd);

printf("Parent exit\n"); } else {

/ Мы находимся в порожденном процессе, который будет получать информацию от процесса-родителя. Открываем FIFO на чтение.*/ if((fd = open(name, 0_RDONLY)) < 0){*

/ Если открыть FIFO не удалось, печатаем об этом сообщение и прекращаем работу */*

printf("Can\'t open FIFO for readingXn"); exit(-1) ;

}

/ Пробуем прочитать из FIFO 14 байт в массив, т.е. всю записанную строку */*

size = read(fd, resstring, 14); if(size < 0){

/ Если прочитать не смогли, сообщаем об ошибке и завершаем работу */*

printf("Can\'t read string\n");

exit(-1);

}

/ Печатаем прочитанную строку */*

```
printf("%s\n",resstring);
/* Закрываем входной поток и завершаем работу */
close(fd);
}
return 0;
}
```

Наберите программу, откомпилируйте ее и запустите на исполнение. В этой программе информацией между собой обмениваются процесс-родитель и процесс-ребенок. Обратим внимание, что повторный запуск этой программы приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом *mknod()*.

Если у вас есть возможность, найдите два компьютера, имеющие разделяемую файловую систему (например, смонтированную с помощью NFS), и запустите на них программы из предыдущего раздела так, чтобы каждая программа работала на своем компьютере, а FIFO создавалось на разделяемой файловой системе. Хотя оба процесса видят один и тот же файл с типом FIFO, взаимодействия между ними не происходит, так как они функционируют в физически разных адресных пространствах и пытаются открыть FIFO внутри различных операционных систем.

2. Практические задания

Задание 1. Связь между родственными процессами через pipe

Напишите программу для организации двусторонней связи между родственными процессами, откомпилируйте ее и запустите на исполнение.

Задание 2. Работа с FIFO

Напишите две программы, одна из которых пишет информацию в FIFO, а вторая — читает из него, так чтобы между ними не было ярко выраженных родственных связей (т. е. чтобы ни одна из них не была потомком другой).