**Лабораторная работа № 6**

# ПЕРЕДАЧА ИНФОРМАЦИИ МЕЖДУ ПРОЦЕССАМИ. СИСТЕМНЫЙ ВЫЗОВ PIPE

## **1. Цель работы**

Целью работы является изучение методов программирования по взаимодействию процессов через системные вызовы и стандартную библиотеку ввода-вывода.

## **2. Задачи работы**

– Закрепление, углубление и расширение знаний студентов при использовании процессов и файлов в прикладных программах.

– Приобретение умений и навыков работы с системой программирования на языках С и С++ в операционной системе Linux.

– Выработка способности логического мышления, осмысления полученных результатов при применении набора функций для работы с процессами через системные вызовы и стандартную библиотеку ввода-вывода для языка C.

## **3. Теоретическая часть**

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

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

*Pipe* можно представить 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. 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'а* 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’а* 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. 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’а*, а для другого – только операция записи в *pipe*. Для выполнения этих операций используются 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. Когда все процессы, использующие *pipe*, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует *pipe*. Таким образом, время существования *pip’а* в системе не может превышать время жизни процессов, работающих с ним.

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:

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include<stdlib.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[1], 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'а 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 \*/

printf("%s\n",resstring);

/\* Закрываем входной поток\*/

if(close(fd[0]) < 0){

printf("Can\'t close input stream\n");

}

/\* Закрываем выходной поток\*/

if(close(fd[1]) < 0){

printf("Can\'t close output stream\n");

}

return 0;

}

**Организация связи через *pipe* между процессом-родителем и процессом-потомком**

Достоинство *pip'ов* не сводится к замене функции копирования из памяти в память внутри одного процесса для пересылки информации через операционную систему. 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().* Это обстоятельство позволяет организовать передачу информации через *pipe* между родственными процессами, имеющими общего прародителя, создавшего *pipe*.

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

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include<stdlib.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 < 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

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[1], "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[1]);

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 pipe, может их использовать.

В этом процессе входной поток данных нам не

понадобится, поэтому закрываем его.\*/

close(fd[1]);

/\* Пробуем прочитать из pip'а 14 байт в массив, т.е. всю

записанную строку \*/

size = read(fd[0], 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[0]);

}

return 0;

}

**Организации двунаправленной связи между родственными процессами через pipe.**

*Pipe* служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере попытаться организовать через *pipe* двустороннюю связь, когда процесс-родитель пишет информацию в *pipe* , предполагая, что ее получит процесс-ребенок, а затем читает информацию из *pip’а*, 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 *pip’а* в двух направлениях необходимы специальные средства синхронизации процессов. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух *pipe*. Модифицируйте программу из предыдущего примера для организации такой двусторонней связи, откомпилируйте ее и запустите на исполнение.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris2) реализованы полностью дуплексные *pip’ы*. В таких системах для обоих файловых дескрипторов, ассоциированных с *pip'ом*, 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 *pip’ом*, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через *pipe*. Помните, что за один раз из *pip’а* 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 *pip'а*. 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 *pip'a*, в процессе, который будет использовать *pipe* для чтения (*close (fd[1]*). 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 *pip'a*, в процессе, который будет использовать *pipe* для записи (*close (fd[0]*) в процессе-родителе.

Попытка прочитать меньше байт, чем есть в наличии в канале связи приводит к чтению требуемого количества байт. При этом возвращается 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.

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 байт, чем осталось до его заполнения. Блокировка вызова разрешена. Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов тем самым 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 *pip’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.

## **4. Задание на лабораторную работу**

1. Прочитать методический материал.

2. Изучить характеристики и синтаксис функций и системных вызовов.

3. Набрать код примеров в текстовые файлы и произвести компиляцию программ.

4. Проверить работоспособность программ.

## **Контрольные вопросы**

1. Что представляет собой в действительности канал *pipe*?
2. Почему в Linux не реализованы полностью дуплексные *pip’ы*?
3. В чем заключается отличие *pip’а* от файла?
4. Что происходит в системе при работе системного вызова *pipe*?
5. Какую информацию содержит переменная *errno*?