Системное программирование

Inter-process communication

IPC

"The last thing one knows in constructing a work is what to put first". -- Blaise Pascal

«Лишь в конце работы мы обычно узнаем, с чего ее нужно было начать». – Блез Паскаль

Каналы и межпроцессное взаимодействие

Лабораторная работа #1

Сергей Станкевич, Минск

Лабораторная работа #6.
РАБОТА С КАНАЛАМИ (Pipes)

Цель. Освоить механизм взаимодействия между процессами на основе каналов (pipes).

Норма времени выполнения: 2 академических часа.
Оценка работы: 3 зачетных единицы.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ.

Межпроцессное взаимодействие

В экосистеме Linux (UNIX) процессы работают каждый в отдельном адресном пространстве. Процессы никак между собой никак не связаны. Поэтому, можно сказать, что каждый процесс «думает», что он в системе один единственный, и вокруг него больше никого нет. Информацию, которую процесс может получить, это информация от операционной системы. Это примерно, как жители многоквартирного дома, ни каждый знает кото его сосед по лестничной клетке, или кто там живет на этаж ниже или три этажа выше. Каждый квартиросъемщик живет своей личной жизнью. Однако жители большого дома имеют некоторые способы передавать информацию своим соседям. Можно постучать по стене, потолку или радиатору отопления чтобы другие услышали сигналы. Можно позвонить по телефону или написать письмо. Наконец-то что-нибудь «запостить» в социальной сети сообщества домоуправления.

Точно также и у процессов существует много способов передачи каких-то данных другим процессам. Это разнообразие способов межпроцессного взаимодействия предоставляет операционная система. Выбор способа взаимодействия зависит от поставленных задач.

Межпроцессное взаимодействие (Inter-process communication), сокращенно, IPC – это набор методов для обмена данными между процессами.

Это обеспечение ядром обмена информацией между процессами и предоставление процессам возможности уведомлять друг друга о событиях.

Ядро Linux реализует большинство механизмов межпроцессного взаимодействия UNIX, в частности определенных и стандартизированных в SystemV и POSIX, а также собственные механизмы взаимодействия.

К основными средствам межпроцессного взаимодействия относятся:

- Сигналы;
- Каналы;
- FIFO (именованные каналы);
- Сокеты;
- сообщения (очереди сообщений);
- семафоры и мьютексы;
- разделяемую память.

На протяжении всего курса обучения мы поработаем со всеми этими средствами, но начнем с каналов.

Каналы в межпроцессном взаимодействии

Канал – это механизм взаимодействия процессов. По каналу производится обмен данными между процессами. Канал (также иногда называемый трубой – **pipe**) представляет собой объект (псевдофайл), который можно использовать для связывания двух процессов.

Каналы осуществляют в Linux родственное локальное межпроцессное взаимодействие.

Рис. 1.8. Два процесса, соединенные каналом

Если процессы A и B захотят пообщаться с помощью канала, они должны установить его заранее. Когда процесс A решает отправить данные процессу B, он пишет их в канал, как если бы это был выходной файл. Процесс B может прочесть данные, читая их из канала, как если бы он был файлом с входными

данными. Таким образом, взаимодействие между процессами в UNIX очень *похоже на обычные чтение и запись файлов*.

Два типа каналов Linux:

- Терминальный канал (безымянный, временный);
- Именованный канал.

Каналы, как, впрочем, и все *объекты операционной системы*, могут использоваться интерфейсом командной оболочки (в терминалах и скриптах), или в программах. Соответственно на программном уровне каналы могут быть реализованы с помощью *системных вызовов* и *библиотечных функций* языка Си.

Терминальный канал

> Командная строка Linux. Полное руководство. — СПб.: Питер, 2017. — 480 с.: Стр. 76

Терминальный канал в Linux – «безымянен», потому что существует анонимно и только во время выполнения процесса.

«Умение» команд Linux читать данные со стандартного ввода и выводить результаты в стандартный вывод используется механизмом командной оболочки, который называется *конвейером*.

Каналы-конвейеры могут перенаправлять стандартный вывод, ввод или ошибку одного процесса другому для дальнейшей обработки.

Синтаксис системного вызова pipe или pipe без имени «спрятан» в символе (вертикальная черта) | между любыми двумя командами:

```
Command-1 | Command-2 | …| Command-N
```

Поэтому часто этот оператор, | , называют также оператором *канала*.

Канал (конвейер) не может быть доступен в другом сеансе; он создается временно, чтобы обеспечить выполнение Command-1 и перенаправить стандартный вывод. Он удаляется после успешного выполнения.

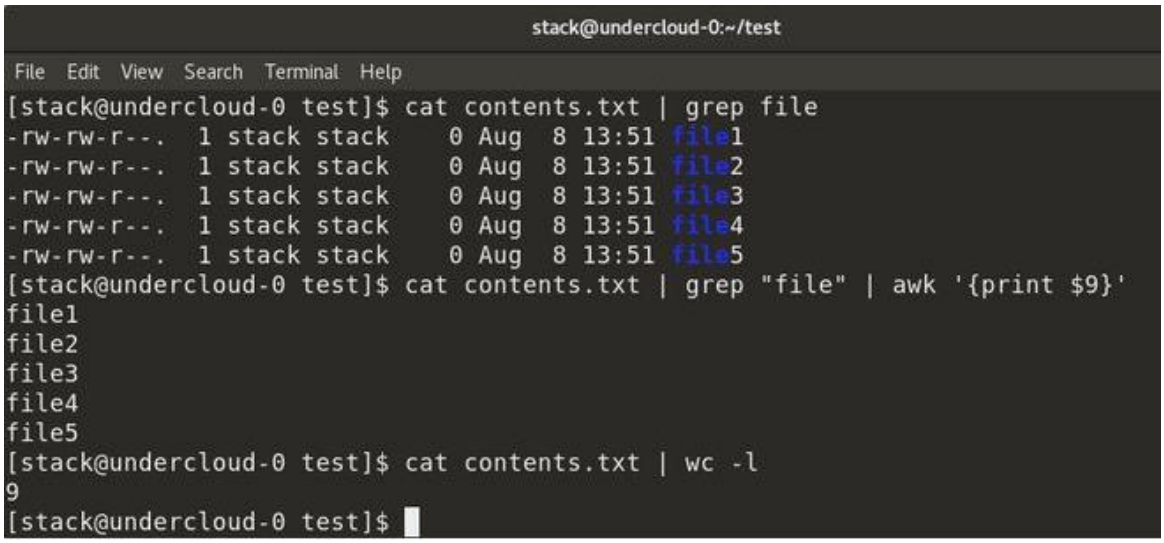
Для демонстрации этого механизма канала-конвейера нам понадобится несколько команд и регулярные выражения.

Команда less может получать данные со стандартного ввода. Используем less в конвейере для постраничного отображения вывода любой команды, которая посылает свои результаты в стандартный вывод:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

Это очень удобно! С помощью этого приема можно со всем комфортом исследовать вывод любой команды, посылающей результаты на стандартный вывод.

Вот еще один пример.



Для проведения такого эксперимента создайте соответствующие файлы.

Использование каналов в программах

Создание канала: pipe()

Интерфейс канала представляет собой два связанных файловых дескриптора, один из которых предназначен для записи данных, другой – для чтения. Возможность родственного межпроцессного взаимодействия через канал обусловлена тем, что дочерние процессы наследуют от родителей открытые файловые дескрипторы.

Для создания канала служит системный вызов pipe(), который объявлен в заголовочном файле unistd.h следующим образом:

```
int pipe (int PFDS[2]);
```

При успешном окончании системного вызова pipe() в текущем процессе появляется канал, запись в который осуществляется через дескриптор PFDS[0], а чтение – через PFDS[1]. При удачном завершении pipe() возвращает 0. В случае ошибки возвращается –1.

Данные, передаваемые через канал, практически не нуждаются в разделении доступа. Канал автоматически блокирует читающий или пишущий процесс, когда это необходимо. Если читающий процесс запрашивает больше данных,

чем есть в настоящий момент в канале, то процесс блокируется до тех пор, пока в канале не появятся новые данные.

Закрытие "концов" канала осуществляется при помощи системного вызова close(). Следует отметить, что снять блокировку с читающего процесса можно, закрыв входной "конец" канала. И наоборот: если пишущий процесс заблокирован операцией write(), например, то закрытие читающего конца канала снимет блокировку. Здесь нужно помнить: закрытие дескриптора происходит только тогда, когда все процессы, разделяющие этот дескриптор, вызовут close().

Перенаправление ввода-вывода: dup2()

Системный вызов dup2() позволяет перенаправлять ввод-вывод. Он объявлен в заголовочном файле unistd.h следующим образом:

```
int dup2 (int FD1, int FD2);
```

Проще можно сказать так: системный вызов dup2() перенаправляет ввод-вывод с дескриптора FD2 на дескриптор FD1. Или так: системный вызов dup2() связывает дескриптор FD2 с ресурсом (файлом, потоком, устройством и т. п.) дескриптора FD1.

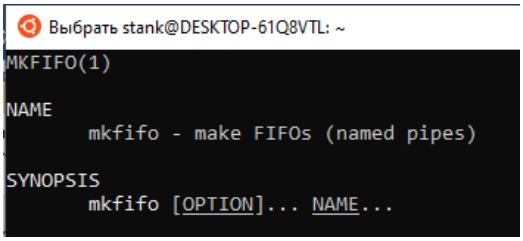
При успешном окончании dup2() возвращает 0, в случае ошибки –1.

Однако, *отсутствие имен* у каналов делает их недоступными для независимых процессов. Этот недостаток устранен в *каналах FIFO*, у которых имеются *имена*.

Именованные каналы (каналы FIFO)

Это *специальный файл*, который использует механизм FIFO (первым пришел, первым вышел).

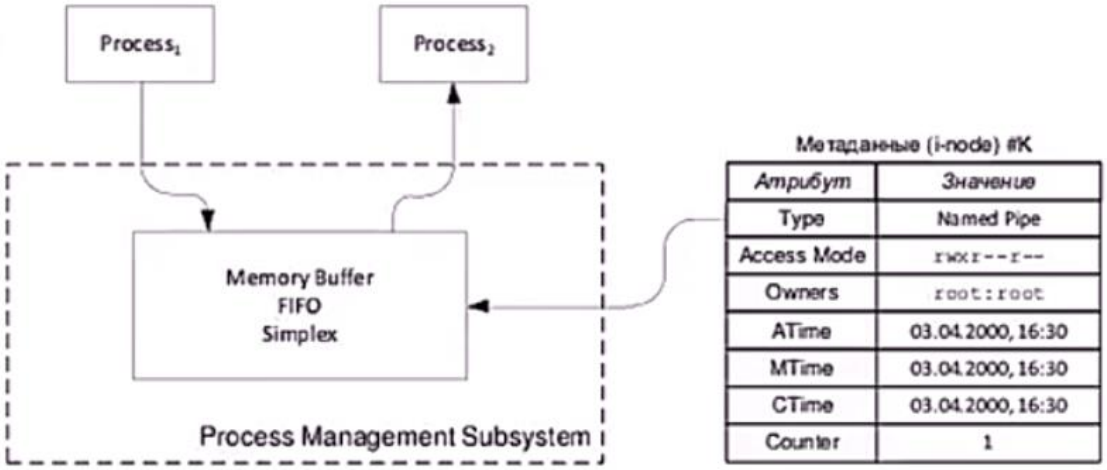
Именованные каналы FIFO работают аналогично безымянным каналам-конвейерам. Особенность именованных каналов в том, что они представлены файлами специального типа (p) и "*видны*" в файловой системе. Обратите внимание, что в расширенном выводе программы ls каналы FIFO обозначаются символом ‘p’ (от англ. *pipe*).



Это, как правило, позволяет использовать их вместо обычных файлов. Следует также отметить, что через FIFO могут взаимодействовать процессы, не являющиеся *ближайшими родственниками*.

Именованный канал может использоваться как обычный файл; то есть вы можете писать в него, читать из него, а также открывать или закрывать его. Он может работать, пока система не будет перезагружена, или пока он не будет удален.

Структура файла-канала



Файл не содержит статической информации, поэтому в нем отсутствуют блоки данных. По этой причине его размер всегда равен 0 байт.

Чтобы создать именованный канал с помощью командной строки, используется команда (shell): **`mkfifo <pipe-name>`** .

Другой способ создать именованный канал FIFO - использовать следующую команду: **`mknod p <pipe-name>`** .

Именованные каналы удаляются командой rm, как обычный файл.

С применением именованных каналов можно, например, выполнять следующие команды: *процесс1 > именованный_канал* и *процесс2 < именованный_канал*, и такая пара команд будет действовать подобно конвейеру *процесс1 | процесс2*.

Файл именованного канала, можно использовать в нескольких сеансах оболочки.

Использование именованных каналов в программах

Программа может самостоятельно создавать именованные каналы FIFO. Для этого предусмотрен *системный вызов* mkfifo(), который объявлен в заголовочном файле sys/stat.h следующим образом:

```
int mkfifo (const char * FILENAME, mode_t MODE);
```

Аргументы: FILENAME — это имя файла, MODE — права доступа.

При успешном окончании mkfifo() возвращает 0, в случае ошибки: –1.

Работа с этими файлами осуществляется обычным образом при помощи системных вызовов open(), close(), read(), write() и т. д. Это, как правило, позволяет использовать их вместо обычных файлов.

Именованный канал существует в системе и после окончания процесса. Он должен быть «отсоединён» или удалён, когда уже не используется. Процессы обычно подсоединяются к каналу для осуществления взаимодействия между ними. Удаление FIFO осуществляется системным вызовом unlink().

Организовать межпроцессное взаимодействие посредством именованных каналов также можно с помощью *библиотечных функций* языка Си.

Есть две библиотечные функции popen, pclose, которые соответственно открывают и закрывают процесс. Они объявлены в заголовочном файле stdio.h следующим образом соответственно:

```
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

Аргумент command – это указатель на С-строку, содержащую командную строку для оболочки. Эта команда передается /bin/sh с помощью флага -c . Интерпретация, если она необходима, выполняется самой оболочкой. Аргумент type – это указатель на С-строку, содержащую символ ‘r’ для чтения или ‘w’ для записи.

Возвращаемое значение popen() – это обычный поток ввода-вывода (за исключением того, что он должен быть закрыт только функцией pclose(), а не fclose()). Запись в канал передается на стандартный поток ввода команды, стандартный поток вывода команды передается в канал, кроме случаев, когда потоки вывода-вывода переопределены самой командой.

Функция popen возвращает NULL, если вызовы fork(2) или pipe(2) закончились ошибкой или если невозможно выделить необходимый для этого объем памяти. Функция pclose возвращает -1, если wait4 возвращает ошибку или если была обнаружена какая-либо другая ошибка.

Заметьте, что выходной поток, возвращаемый popen, по умолчанию полностью *буферизирован*.

Функция pclose ожидает завершения ассоциированного процесса и возвращает код выхода так же, как и функция wait4.

Best of LUCK with it, and remember to HAVE FUN
while you're learning :)

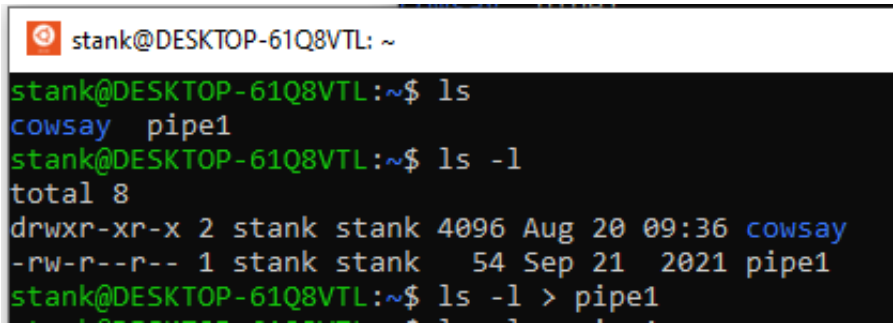
УПРАЖНЕНИЯ

Упражнение 1

Работа с каналами в консольном терминале

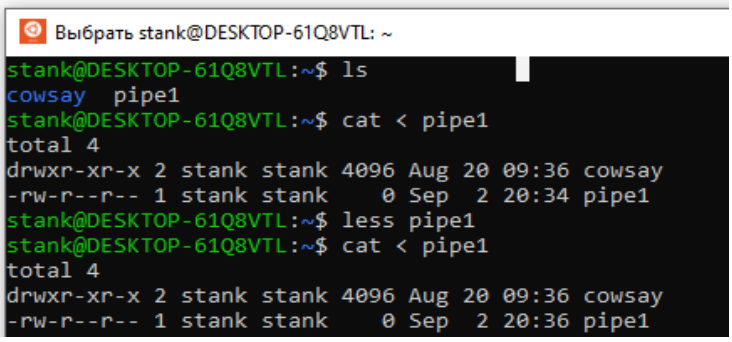
Используем передачу данных с помощью каналов в командной строке. Сначала произведем перенаправление ввода-вывода с помощью обычного регулярного файла.

Откроем два окна терминала. В первом терминале введите простую команду и перенаправьте ее вывод в обычный, *регулярный* файл: `$ ls -l > pipe1`



Команда **ls** в первом окне терминала благополучно закончит работу.

Во втором окне терминала введите следующую команду: `$ cat < pipe1`. Здесь появится список содержимого каталога, созданный в первом окне, как результат работы команды **cat**.



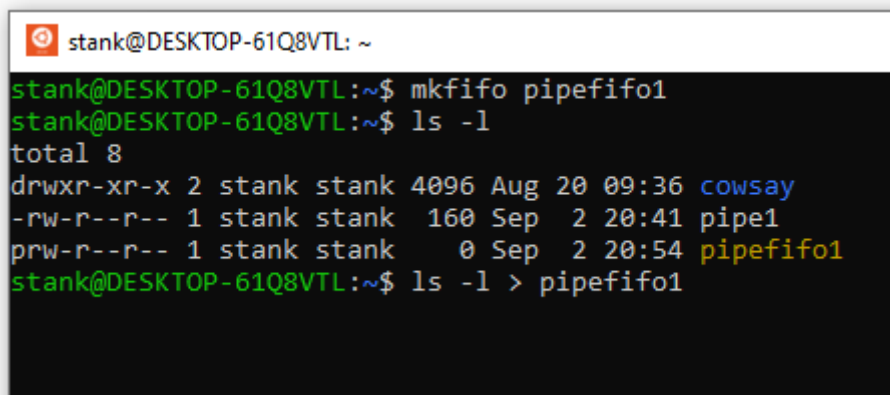
Проведите вышеприведенную операцию, и результаты представьте в отчете скриншотами.

Затем рассмотрим пример создания файла *именованного канала в консольном терминале* и работы механизма передачи данных.

> Командная строка Linux. Полное руководство. — СПб.: Питер, 2017. — 480 с.: Стр. 477

Создадим специальный файл-канал (тип «pipe») с помощью команды mkfifo.

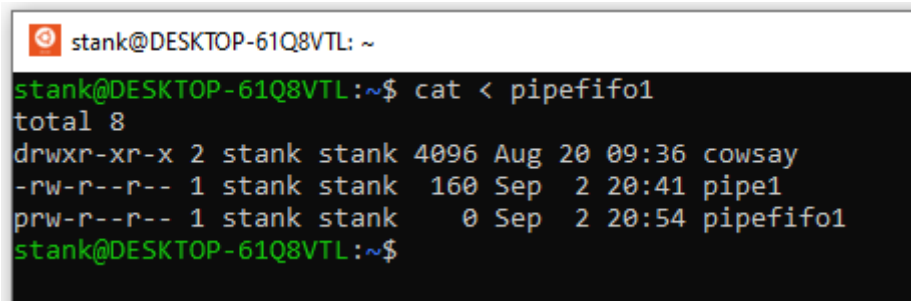
С помощью команды ls -al передадим данные в файл-канал. Здесь мы создали именованный канал pipefifo1 и перенаправили вывод команды ls -al в именованный канал. Обратите внимание на атрибуты файла-каналы, и как подсвечено имя файла. Смотрите следующий скриншот.



После нажатия клавиши ENTER появится ощущение, что команда «зависла». Это объясняется тем, что с другого конца канала данные еще не были прочитаны. В таких ситуациях говорят, что канал заблокирован. Разблокировка произойдет автоматически, как только мы подключим процесс с другого конца канала и прочитаем данные из него.

Мы можем открыть новый сеанс оболочки (терминал) и отследить содержимое именованного канала, в котором показан вывод команды **ls -al**, как было указано ранее. Следите за поведением первого терминала, что там происходит.

Во втором окне терминала введите следующую команду:



Получим результат, здесь обратите внимание, что размер именованного канала равен нулю и имеет обозначение типа файла – ‘**p**’.

> Чтобы перенаправить стандартный вывод любой команды другому процессу, используйте символ ‘>’.
>
> Чтобы перенаправить стандартный ввод любой команды, используйте символ ‘<’.

Продолжаем работу с терминалом.

Проведем сжатие и упаковку коллекции файлов текущей директории и передадим архив в другую директорию с помощью именованного канала и там распакуем архив.

> Одной из основных задач администратора компьютерных систем является обеспечение безопасности данных, а одним из способов решения этой задачи – своевременное создание резервных копий системных файлов.
>
> Даже если вы не являетесь системным администратором, вам все равно пригодится *умение* создавать копии и перемещать *большие коллекции файлов* из одного места в другое и с одного устройства на другое.

Технологии управления коллекциями файлов

- Технология сжатия файлов
- Технология архивирования файлов
- Технология синхронизации файлов в удаленных системах

Программа ***.zip** одновременно является и инструментом *сжатия*, и *архиватором*.

> Windows — программа читает и создает файлы с расширением ***.zip**. Однако в Linux чаще других используется программа сжатия **gzip**, а второе место занимает **bzip2**. Пользователи Linux используют **zip** в основном для обмена файлами с системами Windows, а не как основной инструмент сжатия и архивирования.

В простейшем случае программа **zip** имеет следующий синтаксис:

gzip параметры сжатый_файл файл

Пример как создать *zip-архив* нашей песочницы с каталогом LinuxLab7:

```
$ gzip -r LinuxLab7.zip LinuxLab7
```

Без параметра **-r** (отвечает за рекурсивный обход каталогов) в архив будет включен только каталог LinuxLab7 (без своего содержимого). Расширение ***.zip** добавляется к имени выходного файла автоматически (мы включили его в пример для наглядности).

Извлечение содержимого из zip-архива выполняется просто – с помощью программы **gunzip**:

```
$ cd Desktop
$ gunzip ../playground.zip
```

Проведите подобные операции, но с использованием именованных каналов.

Результаты упражнения представьте в отчете скриншотами.

> Итак, в следующий раз, когда вы будете работать с командами в терминале Linux и обнаружите, что перемещаете данные между командами, каналы сделают этот процесс для вас быстрым и простым.

Упражнение 2

Использование каналов в программах

Сначала рассмотрим примеры с созданием *неименованного* канала.

Пример №1. Реализуем взаимодействия двух процессов-братьев через канал, созданный в родительском процессе, с использованием системных вызовов.

pipe1-parent.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

#define STR_SIZE      32

int main (void)
{
        int pf[2];
        int pid1, pid2;
        char spf [2][STR_SIZE];

if (pipe (pf) == -1) { fprintf (stderr, "pipe() error\n"); return 1; }

sprintf (spf[0], "%d", pf[0]);
sprintf (spf[1], "%d", pf[1]);

if ((pid1 = fork ()) == 0) { close (pf[0]);
        execl ("./pipe1-src", "pipe1-src", spf[1], NULL);
        fprintf (stderr, "exec() [src] error\n"); return 1;
}
```

```
// продолжение

if ((pid2 = fork ()) == 0) { close (pf[1]);
        execl ("./pipe1-dst", "pipe1-dst", spf[0], NULL);
        fprintf (stderr, "exec() [dst] error\n"); return 1;
}
```

```
        waitpid (pid1, NULL, 0);
        close (pf[0]);
        close (pf[1]);
        waitpid (pid2, NULL, 0);
        return 0;
}
```

pipe1-src.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PP_MESSAGE    "Hello World\n"
#define WAIT_SECS     5

int main (int argc, char ** argv)
{
int i, fd;
if (argc < 2) {
fprintf (stderr, "src: Too few arguments\n"); return 1;
}

fd = atoi (argv[1]);

fprintf (stderr, "Wait please");
for (i = 0; i < WAIT_SECS; i++, sleep (1)) fprintf (stderr,
".");
fprintf (stderr, "\n");

if (write (fd, PP_MESSAGE, strlen (PP_MESSAGE)) == -1)
{
fprintf (stderr, "src: write() error\n"); return 1;
}
close (fd);
return 0;
}
```

pipe1-dst.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char ** argv)
{
int fd;
char ch;

if (argc < 2) {
fprintf (stderr, "dst: Too few arguments\n");
return 1;
}
fd = atoi (argv[1]);
while (read (fd, &ch, 1) > 0)
write (1, &ch, 1);

close (fd);
return 0;
}
```

Произведите автосборку многофайлового проекта.

```
$ gcc -o pipe1-parent pipe1-parent.c
$ gcc -o pipe1-src pipe1-src.c
$ gcc -o pipe1-dst pipe1-dst.c
$ ./pipe1-parent
```

Объясните принцип работы приложения.

Пример №2. Перенаправление стандартного ввода-вывода.

Перенаправление стандартного вывода в файл. **Программа dup01.c**	Перенаправление стандартного ввода. **Программа dup02.c**

#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#define FILENAME "myfile"
 	
int main (void)
{
char ch;
int fd = open (FILENAME, O_WRONLY | O_CREAT | O_TRUNC, 0640);
if (fd == -1) {
fprintf (stderr, "open() error\n"); return 1;
}
 	
if (dup2 (fd, 1) == -1) {
fprintf (stderr, "dup2() error\n"); return 1;
}
 	
printf ("Hello World!\n");
 	
close (fd);
return 0;
}
	#include <unistd.h>
	#include <stdio.h>
	#include <fcntl.h>
	#define FILENAME "myfile"
	
	int main (void)
	{
	char ch;
	int fd = open (FILENAME, O_RDONLY);
	if (fd == -1) {
	fprintf (stderr, "open() error\n"); return 1;
	}
	
	if (dup2 (fd, 0) == -1) {
	fprintf (stderr, "dup2() error\n"); return 1;
	}
	
	while (read (0, &ch, 1) > 0)
	write (1, &ch, 1);
	
	close (fd);
	return 0;
	}

Сборка проектов:
 	
$ gcc -o dup01 dup01.c
$ ./dup01
$ cat myfile
	
	$ gcc -o dup02 dup02.c
	$ echo "Hello World" > myfile
	$ ./dup02

Пример №3.

Программа принимает два аргумента: имя каталога и произвольную строку. Затем программа вызывает команду ls для первого аргумента и grep -i для второго. При этом стандартный вывод ls связывается со стандартным вводом grep.

Программа pipe2.c

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
 	
int main (int argc, char ** argv)
{
int pf[2];
int pid1, pid2;
	if (argc < 3) { fprintf (stderr, "Too few arguments\n"); return 1; }
	if (pipe (pf) == -1) { fprintf (stderr, "pipe() error\n"); return 1; }
	
	if ((pid1 = fork ()) == 0) {
	dup2 (pf[1], 1); close (pf[0]);
	execlp ("ls", "ls", argv[1], NULL);
	fprintf (stderr, "exec() [1] error\n");
	return 1;
	}

```
// продолжение

 if ((pid1 = fork ()) == 0) {
 dup2 (pf[0], 0);
 close (pf[1]);
 execlp ("grep", "grep", "-i", argv[2], NULL);
 fprintf (stderr, "exec() [2] error\n");
 return 1;
 }
```

```
 close (pf[1]);
 waitpid (pid1, NULL, 0);
 close (pf[0]);
 waitpid (pid2, NULL, 0);

 return 0;
}
```

Проверяем:

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2 / bin
```

Пример №4. Две программы, которые взаимодействуют через FIFO. Каналы FIFO функционируют аналогично обычным каналам. Если канал опустел, то читающий процесс блокируется до тех пор, пока не будет прочитано заданное количество данных или пишущий процесс не вызовет close(). И наоборот, пишущий процесс "засыпает", если данные не успевают считываться из FIFO.

Программа **fifo2-server.c**

```
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>

#define FIFO_NAME "myfifo"
#define BUF_SIZE 512

int main (void)
{
 FILE * fifo; char * buf;

 if (mkfifo ("myfifo", 0640) == -1) {
 fprintf (stderr, "Can't create fifo\n"); return 1;
 }

 fifo = fopen (FIFO_NAME, "r");
 if (fifo == NULL) {
 fprintf (stderr, "Cannot open fifo\n"); return 1;
 }

buf = (char *) malloc (BUF_SIZE);
 if (buf == NULL) {
 fprintf (stderr, "malloc () error\n"); return 1;
 }
```

Программа **fifo2-client.c**

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

#define FIFO_NAME "myfifo"

int main (int argc, char ** argv)
{
 int fifo;

 if (argc < 2) {
 fprintf (stderr, "Too few arguments\n"); return 1;
 }

 fifo = open (FIFO_NAME, O_WRONLY);
 if (fifo == -1) {
 fprintf (stderr, "Cannot open fifo\n"); return 1;
 }

 if (write (fifo, argv[1], strlen (argv[1])) == -1) {
 fprintf (stderr, "write() error\n");
 return 1;
 }
```

// Программа fifo2-server.c	// Программа fifo2-client.c
fscanf (fifo, "%s", buf); printf ("%s\n", buf); fclose (fifo); free (buf); unlink (FIFO_NAME); return 0; }	close (fifo); return 0; }
Запустим "сервер", он создаст FIFO и "замрет" в ожидании поступления данных: $ gcc -o fifo2-server fifo2-server.c $ ./fifo2-server	Запустим в другом терминальном окне клиентскую программу: $ gcc -o fifo2-client fifo2-client.c $ ./fifo2-client Hello Серверный процесс тут же "проснется", выведет сообщение "Hello" и закончит работу.

Упражнение 3

Использование библиотечных функций для создания именованных каналов

Пример №1. Листинг программы показывает, как процесс получает данные от команды uname. Эта команда предоставляет информацию о пользователе.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output was:-\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Канал открывается по команде библиотечной функции

```
popen ("uname -a","r");
```

Аргумент “uname -a” указывает, к кому обращается данный процесс, т.е. имя другого процесса или команда. В нашем примере – это команда uname. Второй операнд “r” означает, что данные будут считываться (read). Таким образом, наша программа получит по каналу информацию о пользователе. Команда popen (“uname -a”, ”r”) возвращает номер (точнее системный идентификатор) канала. Поэтому мы пишем read_fp = popen (“uname -a”, ”r”). Переменная будет хранить номер канала.

Системные вызовы функции **`popen ("uname -a","r");`**

1. Создание канала pipe(),
2. Создание дочернего процесса fork(),
3. Вызывает командную оболочку.

Далее производится считывание данных из канала по команде

```
chars_read = fread(buffer,sizeof(char), BUFSIZ, read_fp);
```

Здесь chars_read представляет число прочитанных символов. Данные считываются в массив buffer. Размер массива равен BUFSIZ. Последний аргумент – это номер канала.

Пример №2. Получите через канал содержимое текстового файла.

Канал можно создать и для вывода. Вот пример, разберитесь в нем.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define BUFSIZZ 256
  void main(){
   FILE* write_fp;
   char buffer[BUFSIZZ+1];
    sprintf(buffer,"We are studying the Linux...\n");
    write_fp=popen ("cat ","w");
    fwrite(buffer,sizeof(char),strlen(buffer),write_fp);
    pclose(write_fp);
      exit(0);
}
```

Здесь вывод процесса осуществляется на вход команды popen(“cat “,”w”); Таким образом содержимое буферного массива buffer отображается на экране. Исправьте код примера так, чтобы в выводе на консоль заменить слово Linux на Linux Labs.

We hope you enjoy working with Linux!

ЗАДАНИЯ

Задание 1

Передайте через неименованный и именованный каналы следующую информацию связанную с периодами выполнения задания:

1-4 неделя семестра системное время и дату.
5-7 неделя семестра получить календарь, системное время и дату.
8-10 неделя семестра получить имя ядра и название операционной системы, а также системную дату и время.
11-16 неделя семестра получить имя операционной системы, версии и реализацию ядра, системную дату и время, а также календарь.

```
$ Command –help
```

Бонус. Для автоматизации процесса напишите скрипт.

Задание 2

Реализуйте взаимодействия двух процессов-братьев через канал, созданный в родительском процессе, с использованием системных вызовов (см. упражнение 2, пример №1).

Разработайте универсальную программу перенаправления стандартного ввода-вывода (см. упражнение 2, пример №2).

Выполните программу, указанную в упражнении 2, пример №3 с выводом содержимого каталогов на выбор: proc, lib, mnt, usr, etc, home или dev.

Выполните программу, указанную в упражнении 2, пример №4 с выводом с выводом данных, указанных в задании 1.

> * Во всех программах отобразить данные, указанные в задании 1, а также сведения, касающиеся **разработчиков, выполнивших данное задание**.

Задание 3

Разработайте универсальную программу передачи данных с помощью именованного канала с выводом данных о разработчике программы и параметров, указанных в задании №1.

> * при работе с именованными каналами посмотрите файловый дескриптор канала в системе (бонус).

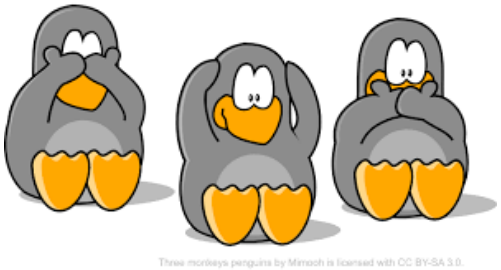
> Дополнительную информацию можно получить в книге Иванов Н. Н. «Программирование в Linux. Самоучитель» стр. 295 – 298.

«Easy things should be easy and hard things should be possible»
«Простые вещи должны быть простыми, а сложные вещи должны быть возможными»

Контрольные вопросы

1. Что такое межпроцессное взаимодействие?
2. Что такое каналы?

3. Какие типы каналов существуют?
4. Что такое именованные каналы?
5. К какому виду межпроцесного взаимодействия относятся каналы? По широте охвата, по направлению передачи данных, по характеру доступа, пояснить.
6. Команды, системные вызовы и библиотечные функции, относящиеся к каналам, и библиотеки их подключения.
7. Команды и системные вызовы, относящиеся к именованным каналам.
8. Атрибуты системных вызовов.
9. Какие задачи выполняют библиотечные функции popen(), pclose().
10. Что такое однонаправленное межпроцессное взаимодействие?
11. Что такое архитектура приложений *клиент*/*сервер*, и в чем ее отличие от архитектуры взаимодействия процессов «источник» - «приёмник».



ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Литература

Шоттс У. «Командная строка Linux. Полное руководство.» — СПб.: Питер, 2017. — 480 с.: ил. — (Серия «Для профессионалов») Подробоно песочница придставлена к книге на страницах 76, 477-480.

Иванов Н. Н. Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2012. — 400 с.: ил. Стр. 287 – 298.

Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил. Стр. 254 – 298.

Справка

man (от англ. manual — руководство) — команда Unix, предназначенная для форматирования и вывода справочных страниц.

Интернет источники