Системные вызовы в Linux

С точки зрения организации операционная система Linux и другие UNIX системы относятся к стандарту POSIX операционных систем. *POSIX* (англ. *Portable Operating System Interface* — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный *API*), библиотеку языка *C* и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных *UNIX*-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-*Unix* систем. Текущий стандарт *POSIX IEEE Std 1003.1-2017 (ISO/IEC/IEEE 9945:2009/Cor 2:2017).*

Стандарт POSIX рассматривает следующие категории системных компонентов:

- средства разработки;
- сетевые средства;
- средства реального времени;
- потоки управления;
- математические интерфейсы;
- пакетные сервисы;
- заголовочные файлы;
- унаследованные интерфейсы.

Применительно непосредственно к ОС, определены ряд основных понятий, соответствующих стандарту POSIX.

1) У пользователя есть имя и числовой идентификатор.

2) Файл - объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. К числу последних относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.

3) Процесс - адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.

4) Терминал (или терминальное устройство) - символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.

5) Сеть - совокупность взаимосвязанных хостов.

6) Языково-культурная среда - часть пользовательского окружения, которая зависит от языковых и культурных соглашений, а также время и дата.

Для работы, с большим числом сущностей, предоставляются механизмы группирования и построения иерархий. Существует:

• иерархия файлов;

• группы пользователей и процессов;

• подсети и другие.

Особенности стандарта *POSIX* могут быть получены при помощи команды

getconf –a

Для работы с приложениями в системах *POSIX* предусмотрены командный интерпретатор (языка *shell*) и/или компилируемый язык С. В первом случае, взаимодействие пользователя с ОС предусмотрено через различные служебные утилиты, сервисы и т.д. (программы), а во втором случае через функции стандартных библиотек (функции системных вызовов).

Системные вызовы предоставляют безопасный интерфейс обращения к системным (привилегированным) ресурсам операционной системы. Системные вызовы передают управление ядру операционной системы, которое определяет, предоставлять ли приложению запрашиваемые ресурсы. Если ресурсы доступны, то ядро выполняет запрошенное действие, затем возвращает управление приложению.

Отметим, так часть системных вызовов может быть переопределена в более высокоуровневых библиотеках (например, *fopen* библиотеки *glibc* переопределен с функции *open* для *UNIX*). Однако, системные вызовы являются наиболее быстрым и безопасным методом доступа к системным ресурсам.

Важно понимать, что каждая функция, например, каждая функция *bash* представляет собой утилиту, работающую или с системными вызовами напрямую или работающую с функциями *glibc,* которые в свою очередь работают с системными вызовами.

Анализ системных вызовов можно провести при помощи библиотек *ltrace* и *strace*.

Задание 1.

1. Проверьте наличие утилит при помощи запросов командной строки

   strace –V

2. Создайте каталог *testdir* с двумя файлами *file1* и *file2*.
3. Изучите работу утилиты *ltrace* при помощи следующего запроса

   ltrace ls testdir/

   Отметим, что каталог с именем *testdir* открывается с помощью библиотечной функции *opendir*, после чего следуют вызовы функций *readdir*, читающих содержимое каталога. В конце происходит вызов функции *closedir*, которая закрывает каталог, открытый ранее.
4. Для просмотра системных вызовов используйте *strace* с командой *ls testdir*.
5. Запишите вызовы в файл при помощи вызова *strace* с флагом *–o*

strace -o trace.log ls testdir/

6. Проверьте записанное в *trace.log*.
7. Найдите все записи *trace.log*, соответствующие *testdir*, например, команда
   grep testdir trace.log
8. Найдите запись execve, например,
   execve("/usr/bin/ls", ["ls", "testdir/"], [/ 40 vars */]) = 0*
9. Найдите вызов *execve* среди вторых страниц *man* и дайте комментарии к данному вызову.
10. Изучите остальные системные вызовы, которые работают с *testdir*.
11. Откройте файл *trace.log* и найдите вызов *getdents*, этот вызов читает записи каталога.
12. Найдите вызов *write* и прокомментируйте его работу, ответьте на вопрос: почему первый аргумент 1 и в каком случае аргумент был бы равен 2?.
13. Отметим, что *trace.log* включает массу других записей, большая часть таких записей – это то инициализация и очистка процесса.
14. Изучите работу утилиты *strace* с флагом *strace –f*.
    strace -f ls testdir
15. Изучите работу *strace* с ключом *–e*, например,
    strace -e write, getdents ls testdir

Задание 2.

1. Создайте два терминала, в первом из них запустите команду *cat* без указания аргументов.
2. Во втором терминале запустите команду
   ps -ef | grep cat
3. Запустите *strace* с опцией *-p* и *PID*, который вы нашли с помощью *ps*.
   Первый системный вызов, который вы увидите — это *read*, ожидающий ввода от потока с номером 0, то есть от стандартного ввода, который сейчас является терминалом, на котором запущена команда *cat*.
4. Вернитесь к терминалу, где вы оставили запущенную команду *cat*, и введите какой-нибудь текст. Проверьте изменения во втором терминале.
5. Завершите работу утилиты cat при помощи комбинации клавиш *Ctrl+C*, ответьте на вопрос, что произошло с треминалом, в котором был запущен *strace*.

Для каждой, описанной в стандарте *POSIX* функции, определено, где и какие заголовочные файлы должны быть включены использующим ее приложением.

Таким образом, на уровне языка С в системах *POSIX* определены стандартные библиотеки со стандартными функциями системных вызовов. Например, следующие:

- <*unistd.h*> различные базовые функции;
- <*complex.h*> комплексная арифметика;
- <*ctype.h*> определение типа символов;
- <*fcntl.h*> открытие и вывод каталогов;
- <*ftw.h*> обход дерева файлов;
- <*sys/ipc.h*> межпроцессное взаимодействие (*IPC*);
- <*sys/mman.h*> отображение файлов в память;
- <*sys/stat.h*> информация о файле (stat и пр.).

Для справки по системным вызовам и вызовам библиотек можно найти на соответствующих страницах утилиты man. Однако, вероятно их придется установить отдельно.

sudo apt install manpages-dev manpages-posix-dev

Например, изучите системный вызов

man 2 open

Man page	***Description***	***Examples***
1	*Executable programs or shell commands*	*man 1 cat; man vi*
2	*System calls (functions provided by the kernel)*	*man 2 sendmsg*
3	*Library calls (functions within program libraries)*	*man 3 abort*
4	*Special files (usually found in /dev)*	*man 4 intel* *man 4 amdgpu*
5	*File formats and conventions eg /etc/passwd*	*man 5 shadow*
6	*Games*	*man 6 gti man 6 sl*
7	*Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)*	*man 7 inode*
8	*System administration commands (usually only for root)*	*man 8 ip*
9	*Kernel routines [Non standard]*	*man 9 vmxnet*

Задание 3.

1. Изучите функцию *system* из 3 листа *man.*
2. Создайте приложение, выполняющее следующий код.

```
#include <stdlib.h>
int main() {
int return_value;
return_value = system("ls –l /");
```

```
return return_value;
}
```

3. Разберитесь, как он работает и напишите комментарии к коду.

   Отметим, что на практике часто функция main задается в следующем виде

   ```
   int main( int argc[] , char *argv[ ] [, char *envp[ ] ] );
   ```

   данное объявление позволяет удобно передавать аргументы командной строки и переменные окружения. Определение аргументов:

   - *argc* - количество аргументов, которые содержатся в *argv[]* (всегда больше либо равен 1);
   - *argv* - в массиве строки представляют собой параметры из командной строки, введенные пользователем программы. По соглашению, *argv[0]* – это команда, которой была запущена программа, *argv[1]* – первый параметр из командной строки и так далее до *argv[argc]* – элемент, всегда равный *NULL*;
   - *envp* - это массив строк, которые представляют собой переменные окружения. Массив заканчивается значением *NULL*.

   Таким образом при вызове приложения функции *main* в качестве аргументов начиная со второго (*argv[1]*) будут поданы аргументы, указанные при вызове.

4. Перепишите программу так, чтобы в ней первый аргумент (*argv[1]*) указывал на действие в оболочке *shell*.

Работа с файлами в системе *POSIX*

В системе *POSIX* понятие файл – это одна из базовых сущностей, которая определяется как - объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип, дата создания, дата последнего изменения и другие. В системе *POSIX* к файлам относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. При этом каждая конкретная реализация ОС может поддерживать и другие типы файлов.

Также для работы с файлами в системах *POSIX* предусмотрена единая иерархия файлов. Таким образом, что полное имя каждого файла (полный путь) представляет собой единую древовидную структуру, начиная от корневого каталога и включая все остальные. Наиболее современный стандарт организации такой структуры *FHS (Filesystem Hierarchy Standard)*.

Система не накладывает на информацию, хранимую в файле, никаких ограничений или структурных требований, например, каталог - это файл, осуществляющий связь между именами файлов и собственно файлами, создавая тем самым структуру. Другим приёмом является системное внешнее запоминающее устройство. Системное внешнее запоминающее устройство - это устройство, к которому обращается аппаратный загрузчик

для загрузки операционной системы. Для операционной системы *Unix* корневой каталог файловой системы располагается на ее системном устройстве. При этом файловая система также включает файлы расположенные на других носителях. Для связи иерархий файлов, расположенных на разных носителях, применяется процедура монтирование файловой системы (системный вызов *mount(char *special, *name, rwflag)*).

Стандартная библиотека *C (glibc)* представляет библиотеку стандартного ввода-вывода (*stdio*) с пользовательской буферизацией, включающей операции: открытие, закрытие, чтение и запись файлов. Помимо этих функций существуют и прямые системные вызовы *Linux* (мы будем применять первый тип вызовов). На языке стандартного ввода-вывода открытый файл называется потоком данных (*stream*). Потоки данных могут открываться для чтения (входные потоки), записи (выходные потоки) или для обеих операций (потоки ввода-вывода).

Основные функции работы с файлами в системе *POSIX* определены в библиотеках

```
#include <fcntl.h>      /* open() and O_XXX flags */
#include <sys/stat.h>   /* S_IXXX flags */
#include <sys/types.h>  /* mode_t */
#include <unistd.h>     /* close() */
```

Такими функциями являются следующие:

Управление файлами	
fd=open(file, how, ...)	Открывает файл для чтения, записи
s=close(fd)	Закрывает открытый файл
n=read(fd, buffer, nbytes)	Читает данные из файла в буфер
n=write(fd, buffer, nbytes)	Пишет данные из буфера в файл
Position=lseek(fd, offset, whence)	Передвигает указатель файла
s=stat(name, &buf)	Получает информацию о состоянии файла
Управление каталогами и файловой системой	
s=mkdir(name, mode)	Создает новый каталог
s=rmdir(name)	Удаляет пустой каталог
s=link(name1, name2)	Создает новый элемент с именем
s=unlink(name)	Удаляет элемент каталога
s=mount(special, name, flag)	Монтирует файловую систему
s=umount(special)	Демонтирует файловую систему

Все системные функции работы с файлами используют т.н. файловый дескриптор (целочисленное значение). Это значение формируется в ОС и индивидуально привязывается к конкретному процессу. Нумерация значений дескрипторов для каждого процесса начинается с нуля и определяется в момент открытия файлов, из условия минимального значения доступного (незанятого) дескриптора. Когда файл закрывается, то соответствующий ему номер дескриптора освобождается и используется для открытия других файлов.

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.

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, необходимо должны поместить информацию о файле в таблицу файловых дескрипторов (таблица открытых файлов) и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом

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

из библиотеки <*fcntl.h*>. Параметр *flags* функции *open* может принимать одно из следующих трех значений:

- *O_RDONLY* – только операции чтения;
- *O_WRONLY* – только операции записи;
- *O_RDWR* – операции чтения, и операции записи.

Каждое из этих значений может быть скомбинировано посредством операции "побитовое или

(|)" с одним или несколькими флагами, например:

- *O_CREAT* – если файла с указанным именем не существует, он должен быть создан;
- *O_EXCL* – ошибка, если файл уже создан (совместно с флагом *O_CREAT*).
- *O_NDELAY* – запрещает перевод процесса в состояние ожидание при выполнении операции открытия и любых последующих операциях над этим файлом; ·
- *O_APPEND* – при открытии файла и перед выполнением каждой операции записи (если она, 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, времен последнего доступа к файлу и его последней модификации.

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. Параметр *mode* 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, (mode & ~umask*).

Другими важными операциями работы с файлами являются следующие.

- Функция *ssize_t read(int fd, void *buf, size_t count),* возвращает число прочитанных слов *(<unistd.h>*). Функция *read(...)* читает байты с текущей позиции файла, однако если смещение текущей позиции равно или больше размера файла, то возвращается значение 0. Если данные недоступны, но конец файла не достигнут, то ожидается поступление данных, что в общем случае определяется режимом открытия файла.
- Функция *ssize_t write(int fd, const void *buf, size_t count),* возвращает число записанных слов *(<unistd.h>*).
- Функция *off_t lseek(int fd, off_t offset, int whence)* – сдвиг позиции начала считывания/записи в файл *(<unistd.h>, <sys/types.h>).*

  Флаг *whence* — директива, задающая интерпретацию смещения: *SEEK_SET = 0* - смещение устанавливается в *offset* байт; *SEEK_CUR = 1* - смещение устанавливается как текущее смещение плюс *offset* байт; *SEEK_END = 2* - смещение устанавливается как размер файла плюс *offset* байт. Напрмер:
  - *lseek(fd,0,0)* текущая позиция на начале файла;
  - *lseek(fd,0,2)* текущая позиция в конце файла;
  - *lseek(fd,-10,1)* текущая позиция -10 байт.
- Функция *int close(int fd),* возвращает ошибку или ее отсутствие *(<unistd.h>).*

Задание 4.

1. Создайте проект, в котором должен быть один файл main.cpp и один файл *hello*, с записью «*Hello world*!» Внутри.
2. В файле *main.cpp* запишите следующую программу

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

```
#define BUFFER_SIZE 100

char buffer2read[BUFFER_SIZE];
const char buffer2write[]="HELLO WORLD!\n";
size_t sizeofwrite = strlen(buffer2write);
size_t size=0;

int main(int argc, char **argv){
if(argc != 2)  {
      printf("please, call the function with filepath");
      return 1;}
int fd = open(v[1], O_RDWR);

perror("fd");

printf("fd=%d\n",fd);
size = read(fd, buffer2read, BUFFER_SIZE);

printf("size = %ld %d\n", size, (int)sizeof(buffer2read));
size = write(fd, buffer2write, sizeofwrite );

printf("size of write %d written %ld \n",(int) sizeofwrite, size);

size = read(fd, buffer2read, BUFFER_SIZE);
printf("size = %ld %d\n", size, (int)sizeof(buffer2read));
close(fd);
}
```

3. Напишет комментарии в коде, касательно того, что делает каждая строка.
4. Модифицируйте программу так, чтобы можно было создавать новый файл, если он не существует.
5. Модифицируйте программу так, чтобы можно было отдельным аргументом задавать создание файла на чтение и запись или только на чтение.
6. Модифицируйте программу так, чтобы новая фраза записывалась в файл начиная с 3 позиции текста.
7. Ознакомьтесь с командой *chmod*() (*man* 2; *int chmod(const char *pathname, mode_t mode) #include <sys/types.h>; #include <sys/stat.h>)* и добавьте ее в программу для изменения прав доступа к файлу.
8. Ознакомьтесь с функциями link и softlink добавьте их в приложение. Проверьте и ответьте на вопрос, чем отличаются результаты их работы.

Одной из наиболее важных функций для работы с файлами является системный вызов *int stat (const char *pathname, struct stat *buf); (#include <sys/types.h>; #include <sys/stat.h>; #include <sys/time.h>).* В данная функция заполняет структуру *stat* настройками для выбранного файла. Структура имеет следующий вид

```
struct stat {
   dev_t st_dev;     /* логическое устройство, где находится файл */
   ino_t st_ino;      /* номер индексного дескриптора */
   mode_t st_mode;         /* права доступа к файлу */
   nlink_t st_nlink;       /* количество жестких ссылок на файл */
   uid_t st_uid;           /* ID пользователя-владельца */
   gid_t st_gid;           /* ID группы-владельца */
   dev_t st_rdev;          /* тип устройства */
```

```
    off_t st_size;            /* общий размер в байтах */
    unsigned long st_blksize;  /* размер блока ввода-вывода */
    unsigned long st_blocks;   /* число блоков, занимаемых файлом */
    time_t st_atime;          /* время последнего доступа */
    time_t st_mtime;          /* время последней модификации */
    time_t st_ctime;          /* время последнего изменения */
};
```

https://fitu.npi-tu.ru/assets/fitu/iist/files/metod/laboratornyie-pi-api-linux-kovalevskij(bakalavryi)..pdf

По мимо *stat()* в *Linux* предусмотрены функции *fstat*() возвращает сведения о файле, представляемом дескриптором файла *fd*; *lstat*() идентична *stat*(), за исключением того, что если передать ей символическую ссылку, то *lstat*() вернет информацию о самой ссылке, а не о целевом файле.

Отметим, что Время в *Linux* отсчитывается в секундах, прошедшее с начала этой эпохи (*00:00:00 UTC,* 1 Января 1970 года). Для получения системного времени можно использовать функции:

```
time_t time (time_t *tt); int gettimeofday (struct timeval *tv, struct timezone *tz);
```

из библиотеки *#include <sys/time.h>*, где

```
struct timeval {
    long tv_sec; /* секунды */
    long tv_usec; /* микросекунды */
};
```

Задание 5.

1. Создайте проект, в котором должен быть один файл *main.cpp* и один файл *hello*, с записью «*Hello world*!» Внутри.
2. Файл *main.cpp* должен содержать следующий код

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char **argv){
    if(argc != 2)  {
        printf("please, call the function with filepath");
        return 1;
     }

    struct stat fileStat;
    if(stat(argv[1],&fileStat) < 0)
        return 1;

    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n",(int)fileStat.st_size);
    printf("Number of Links: \t%d\n",(int)fileStat.st_nlink);
    printf("File inode: \t\t%d\n",(int)fileStat.st_ino);

    printf("File Permissions: \t");
    printf((S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf((fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf((fileStat.st_mode & S_IWUSR) ? "w" : "-");
```

```
    printf((fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf((fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf((fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf((fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf((fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf((fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf((fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n");

  printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode)) ?
  "is" : "is not");

    return 0;
}
```

3. Создайте приложение для файла *main.cpp* и изучите его работу.
4. Дополните приложение выводом информации об *ID* пользователя-владельца и группы владельца.
5. Дополните приложение выводом информации о времени последнего изменения.

Задание 6.

1. Разработать и отладить процедуру, выполняющую следующие действия:
   a. последовательное открытие трех существующих файлов;
   b. вывод на консоль метаданных этих файлов соответственно системными вызовами *stat, fstat , lstat*;
   c. закрытие этих файлов.

Работа с каталогами.

В системах *POSIX* каталоги являются особыми типами файлов. Для работы с каталогами предусмотрены ряд системных вызовов. Логика таких вызовов аналогична предыдущим примерам работы с файлами. Для открытия и закрытия каталогов используются следующие функции из библиотеки *<dirent.h>:*

```
DIR *opendir (const char *dirname);
```

Успешный вызов *opendir*() создает поток каталога, который представляет каталог, указанный при помощи параметра *dirname*. Поток каталога — это дескриптор файла, представляющий открытый каталог, некоторое количество метаданных и буфер для записи содержимого каталога. Зная дескриптор можно имея поток каталога, можно получить дескриптор файла для соответствующего каталога *dirfd (DIR *dir)*.

Для чтения записей каталога существует вызов:

```
struct dirent *readdir(DIR *dirptr);
```

который возвращает структуру вида

```
struct dirent {
long d_ino;   /* уникальное число для каждого файла */
off_t d_off;    /* смещение данного элемента в реальном каталоге */
unsigned short d_reclen;    /* перемменная длина структуры */
char d_name[1]; /* начало массива символов, задающего имя элемента каталога; данное
имя ограничено нулевым байтом и может содержать не более MAXNAMLEN символов */
};
```

в системе *Linux* обязательным является только поле *dirent* структуры - это *d_name*, то есть имя файла в каталоге.

Функция *readdir* имеет следующую особенность: при первом вызове в структуру *dirent* будет считана первая запись каталога. Затем, при втором вызове вторая запись и так далее. После прочтения всего каталога в результате последующих вызовов *readdir* будет возвращено значение *NULL*. Для возврата указателя в начало каталога на первую запись предусмотрен вызов *void rewindir(DIR *dirptr).*

Для работы с каталогами существуют системные вызовы:

```
int mkdir (const char *pathname, mode_t mode);
int rmdir (const char *pathname);
```

Также для работы с каталогами в Linux предусмотрены такие функции, как:

- *char * getcwd(char *name, size_t size)* – получение текущего каталога;
- *int chdir (const char *path); int fchdir (int fd);* - смена текущего каталога;
- *int mkdir (const char *pathname, mode_t mode);*
- *int rmdir (const char *pathname);*
- *int closedir (DIR *dirptr)* – Закрыть каталог.

где *mode* – права доступа. Следует отметить, что что вызов *rmdir*() будет успешен, только если удаляемый каталог пуст, т.е. содержит записи "точка" (.) и "двойная точка" (..).

Задание №7

1. Изучите следующий код и дайте комментарии к его работе

```
int find_flin_in_dir (const char *user, const char * flin) {
    struct dirent *entry;
    int ret = 1;
    DIR *dir; dir = opendir (user);
    errno = 0;
    while ((entry = readdir (dir)) != NULL) {
        if (!strcmp(entry->d_name, flin)) {
        ret = 0; break;
        }
    }
    if (errno && !entry)
            perror ("readdir");
    closedir (dir); return ret;
}
```

2. Написать программу вывода на экран содержимого заданного пользователем каталога с использованием изученных функций.

Задание №8

1. Разработать и отладить процедуру, выполняющую следующие действия:
    a. создание нового каталога *DIR4*;
    b. поменять на другой существующий каталог, заполненный файлами;

c. вывод на консоль содержимого этого каталога - файлов с их характеристиками (см. задание 6);
d. закрытие текущего рабочего каталога и уничтожение каталога *DIR4*.