04. Файловый ввод-вывод

Егор Орлов

Курс: UNIX-DEV-SYS. Системное программирование в среде UNIX (Linux/FreeBSD). ВИШ СПбПУ, 2021

Содержание

1 Вводная информация **2**
1.1 Задачи в проекте . . . 2
1.2 Инструменты . . . 2

2 Системные вызовы **2**
2.1 Общая информация . . . 2
2.2 О системных вызовах . . . 3
2.3 Пример системного вызова (examples/sleep.c) . . . 5
2.4 Прямой вызов сисемных вызовов (examples/syscall.c) . . . 6

3 Работа с файлами в ОС Linux на низком уровне **6**
3.1 Хранение файлов в ФС . . . 6
3.2 Индексные дескрипторы . . . 7
3.3 Просмотр индексного дескриптора . . . 7
3.4 Структура inode . . . 7
3.5 Основные системные вызовы для работы с файлами . . . 7
3.6 Файловый указатель . . . 8

4 Низкоуровневый ввод-вывод **8**
4.1 Заголовочные файлы функций-оберток . . . 8
4.2 Открытие файла (open/openat) . . . 8
4.3 Создание файла (creat) . . . 9
4.4 Закрытие файла (close) . . . 9
4.5 Чтение из файла (read) . . . 9
4.6 Запись в файл (write) . . . 10
4.7 Перемещение файлового указателя (lseek) . . . 10
4.8 Интерфейс управления файловым дескриптором (fcntl) . . . 10
4.9 Проверка прав доступа к файлу (access) . . . 10
4.10 Управление дисковым кэшем (fsync) . . . 11
4.11 Пример - ведение журнала (journal.c) . . . 11

5 Управление объектами файловой системы **12**
5.1 Создание жесткой ссылки (link) . . . 12
5.2 Создание символической ссылки (symlink) . . . 12
5.3 Удаление файла (unlink) . . . 12

5.4 Переименование (rename) . . . 12
5.5 Работа с каталогами (dirent.h) . . . 12

6 Высокоуровневый ввод-вывод (stdio.h) 13
6.1 Возможности stdio.h . . . 13
6.2 Ввод одиночного символа - getchar . . . 14
6.2.1 Пример . . . 14
6.3 Вывод одиночного символа . . . 14
6.3.1 Пример . . . 14
6.4 Форматированный вывод - printf . . . 15
6.5 Спецификаторы форматированного ввода-вывода . . . 15
6.6 Форматированный ввод - scanf . . . 15
6.7 Ввод-вывод с текстовыми файлами . . . 15
6.8 Открытие файла (высокоуровневое) . . . 16
6.9 Закрытие файла (высокоуровневое) . . . 16
6.10 Обработка ошибок для функций обращения к ОС (errno.h) . . . 16
6.10.1 Пример . . . 17
6.11 Ввод-вывод отдельных символов в файлах . . . 17
6.12 Форматированный ввод-вывод в файлах . . . 18
6.13 Ввод-вывод строк в потоках . . . 18
6.14 Блочный ввод-вывод . . . 18

1. Вводная информация

1.1. Задачи в проекте

- Реализовать ведение журнала
- Реализовать выдачу статических файлов в соответствии с путем в HTTP-запросе

1.2. Инструменты

- Низкоуровневые и высокоуровневые функции работы с файлами

2. Системные вызовы

2.1. Общая информация

- **Системные вызовы** - API ядра ОС для использованияя из пользовательского пространства (приложений). При этом как и при вызове обучных функций в системные вызовы передаются параметры, а вызовы возвращают значения.
- **Режим ядра (kernel mode)** — привилегированный режим, используемый ядром операционной системы.
- **Пользовательский режим (user mode)** — режим, в котором выполняется большинство пользовательских приложений.
- Переключение из пользоватльского режима в режим ядра происходит посредством механизма **программных прерываний**.

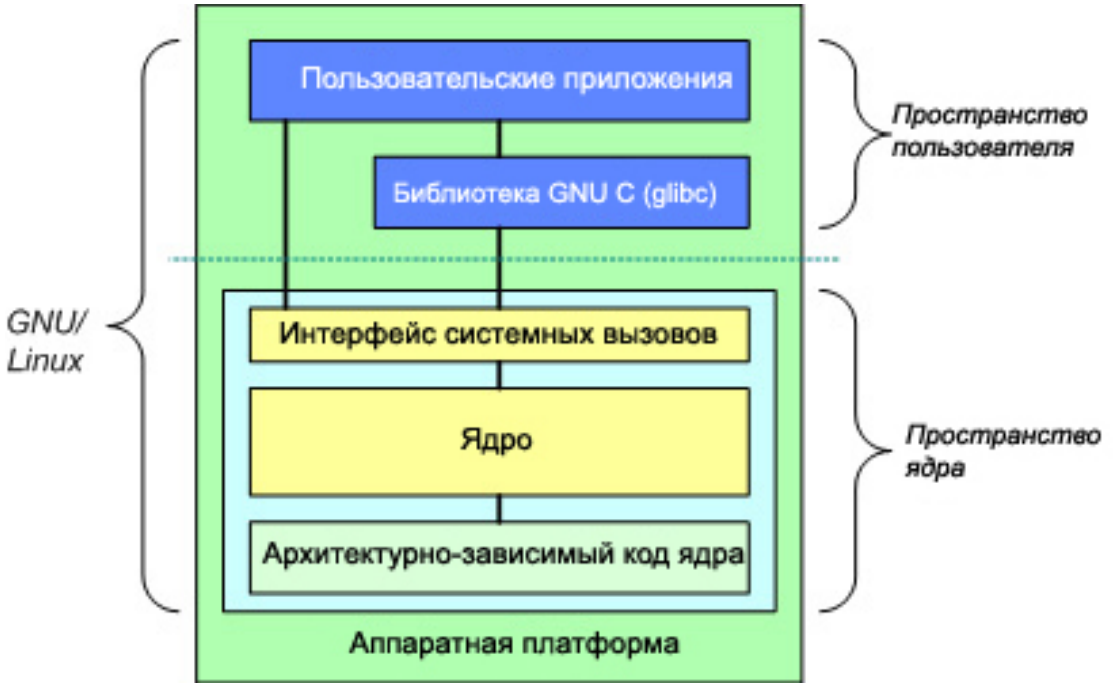

Рис. 1: Обращение приложения к функциям ядра

  - происходит копирование данных из области памяти пользовательского процесса в область памяти ядра и наоборот
  - происходит переход потока выполнения от процесса к функции ядра и наоборот
  - достаточно затратная процедура
- Без системных вызовов практически нельзя сделать ничего осмысленного.
- Инструмент анализа - утилита **strace**.

2.2. О системных вызовах

- Основные группы системных вызовов:
  - Управление процессами
  - Управление файлами
  - Управление каталогами и файловой системой
  - Прочие
- Типы системных вызовов:
  - **блокирующие** - полностью обрабатываются в ядре, возвращение вызывающему процессу только после завершения, так же назывется синхронным режимом выполнения, процесс на время обработки его блокирующего вызова переводится планировщиком в режим ожидания; таких вызовов большинство, ибо проще в реализации и проще использовать,
  - **неблокирующие** - достаточно быстро возвращают управление вызывающему процессу, но требуется выполнять дополнительные действия, чтобы понять, завершена ли операция целиком (асинхронным режимом выполнения)

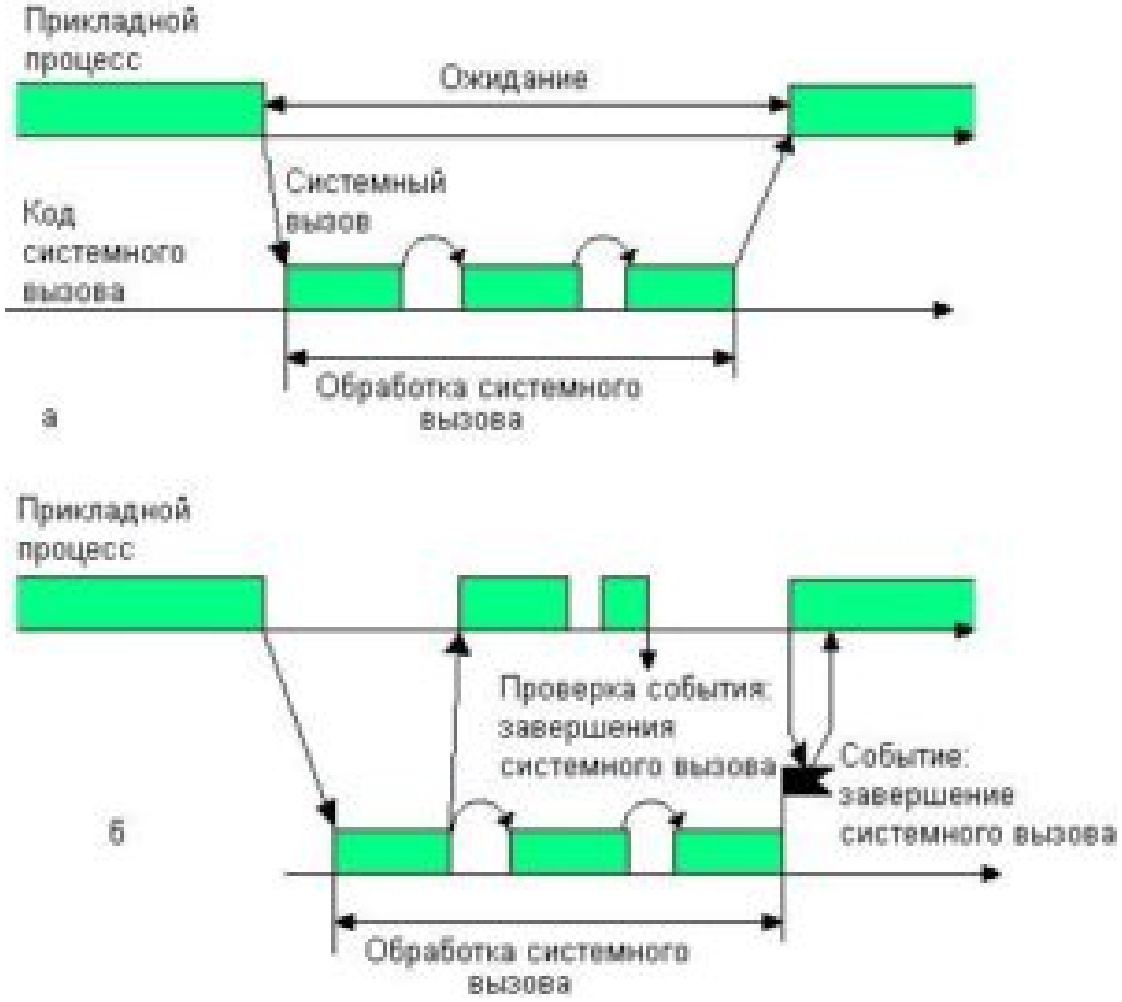

Рис. 2: Блокирующие и неблокирующие системные вызовы

- Доступ к системным вызовам:
  - напрямую
  - через функции-обертки в unistd.h
  - через более высокоуровневые функции стандартной и прочих библиотек
- Системные вызовы стандартизованы в стандартах:
  - SUS
  - POSIX

2.3. Пример системного вызова (examples/sleep.c)

```
#include <unistd.h>
int main() {
        sleep(1);
        return 0;
}
```

- **unistd.h** - заголовочный файл стандартной библиотеки C, описывающий функции-обертки системных вызовов
- **sleep** - функция-обертка системного вызова **clock_nanosleep**

```
$ cc -c sleep.c -o sleep.o
$ nm sleep.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U sleep
$ cc sleep.c -o sleep
$ nm ./sleep
. . .
0000000000001135 T main
00000000000010b0 t register_tm_clones
                 U sleep@GLIBC_2.2.5
0000000000001050 T _start
0000000000004008 D __TMC_END__
```

Функция **sleep** в стандартной библиотеке C (GLibC - реализация libc от GNU)

```
$ ltrace sleep
sleep(1)                                                  = 0
+++ exited (status 0) +++
```

Трассировка вызовов библиотечных функций

```
$ strace ./sleep
. . .
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffd698802c0) = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Отображение в системный вызов

2.4. Прямой вызов сисемных вызовов (examples/syscall.c)

- Возможности, которые не входят в стандарты SUS, POSIX, в каждой UNIX-системе для этого могут быть свои механизмы, в Linux - функция **syscall** из glibc, заголовок в **sys/syscall.h**

```
#include <unistd.h>
#include <sys/syscall.h>
int main() {
        char msg[9] = "Syscall!\n";
        syscall(__NR_write, 1, msg, 9);
        return 0;
}
```

- Параметры **syscall**
  - код системного вызова (в виде константы из unistd.h)
  - дескриптор потока вывода - STDOUT
  - указатель на буфер
  - количество символов в буфере

  Вывод на экран путем дергания напрямую системного вызова **write**

```
$ ltrace ./syscall
syscall(1, 1, 0x7ffef2300e3f, 9Syscall!
)                               = 9
+++ exited (status 0) +++
```

  Смотрим библиотечные вызовы

```
$ strace ./sycall
. . .
write(1, "Syscall!\n", 9Syscall!
)   = 9
exit_group(0)                           = ?
+++ exited with 0 +++
```

  Просмотр системных вызовов. Системный вызов возвращает кол-во переданных байт

3. Работа с файлами в ОС Linux на низком уровне

3.1. Хранение файлов в ФС

- В каталоге для каждого файла каталога хранится:
  - имя файла
  - номер индексного дескриптора **inode**
- Вся остальная информация о файле - в **inode**

```
$ ls -il /bin/ls
2753245 -rwxr-xr-x 1 root root 137888 янв 14 15:17 /bin/ls
```

- Записи в каталогах ФС об именах файлов создают иерархическую структуру
- Фактически запись о файле в каталоге - это ссылка на **inode**
- Ссылок на конкретный **inode** может быть несколько
  - первая ссылка создается при создании файла (вместе с самим inode)
  - дополнительные можно создавать командой **ln**

3.2. Индексные дескрипторы

- **inode** (index node) - индексный дескриптор
- Содержит метаданные файла, а именно:
  - расположении (физические адреса на диске) **всех блоков данных**, принадлежащих файлу
  - размер файла, кол-во занимаемых блоков на диске
  - тип файла
  - права доступа к файлу
  - владельца файла
  - даты последнего доступа, создания, изменения
  - кол-во ссылок на inode
- Хранятся в спец. области файловой системы, часто ограниченной по размеру

- Удаление файла - это удаление его метаданных (**inode**)

3.3. Просмотр индексного дескриптора

- Команда **stat**

```
$ stat /etc/passwd
  Файл: /etc/passwd
  Размер: 2737  Блоков: 8  Блок В/В: 4096   обычный файл
Устройство: fd00h/64768d   Inode: 1575711     Ссылки: 1
Доступ: (0644/-rw-r--r--)  Uid: (0/root)   Gid: (0/root)
Доступ: 2019-06-30 22:48:12.626850670 +0300
Модифицирован: 2019-06-28 20:59:17.394701734 +0300
Изменён: 2019-06-28 20:59:17.494701738 +0300
```

3.4. Структура inode

3.5. Основные системные вызовы для работы с файлами

Вызов	Описание
open	Принимает имя и флаги открытия, возвращает файловый дескриптор
write	Принимает файловый дескриптор, данные для записи, их количество
read	Принимает файловый дескриптор и куда сложить прочитанные данные, их количество
close	Принимает файловый дескриптор
creat	Создание файла
lseek	Устанавливает файловый указатель на определенное место в файле
unlink	Удаление файла
fcntl	Управление файловым дескриптором
access	Проверка доступа к файлу
fsync	Управление кэшированием
stat	Получение индексного дескриптора для файла по имени

Вызов	Описание
fstat	Получение индексного дескриптора для файла по файловому дескриптору

3.6. Файловый указатель

- Смещается по файлу по мере выполнения операций ввода вывода
- При открытии файла устанавливается на начало файла.
- По мере выполнения операции **чтения** смещается на объем считываемых данных, т.е. всегда указывает на еще непрочитанные данные
- При выполнении операции **запись** указатель смещается на количество записанных данных.

4. Низкоуровневый ввод-вывод

4.1. Заголовочные файлы функций-оберток

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

4.2. Открытие файла (open/openat)

- Возвращает файловый дескриптор, или -1 в случае возникновения ошибки. Проверка ошибки - через глобальную переменную **errno**

```
#include <fcntl.h>
int open(const char *pathname, int flags);
```

- Поле **flags** определяет режим открытия файла, может включать один из основных режимов и несколько дополнительных

Основные режимы	Описание
O_RDONLY	Режим только чтения
O_WRONLY	Режим только записи
O_RDWR	Режим чтения и записи

Дополнительные режимы	Описание
O_SYNC	Отключение буферного кэша, данные попадают на диск до возврата из write
O_CREAT	Если файл есть, то игнорируется, иначе создается
O_APPEND	Установить файловый указатель на конец файла
O_TRUNC	Если файл существует, уничтожить его содержимое

Дополнительные режимы	Описание
O_EXCL	Вместе с O_CREAT создаем файл, если есть, то НЕ открываем

- Использование функции **open** для создания файла

```
#include <fcntl.h>
int open(const char *pathname, int flags, mode_t mode);
```

Параметр mode имеет смысл только при вызове **open** с включением режима O_CREAT

- Пример - открытие на запись и при отсутсвии создание файла с установкой указателя в конец

```
int fd = open ("logfile", O_WRONLY | O_CREAT | O_APPEND, 0600);
```

- **openat** - открывает файл, добираясь к нему по относительному пути от файлового дескриптора текущего каталога

```
#include <fcntl.h>
int openat(int fd, const char *path, int oflag, ...);
```

4.3. Создание файла (creat)

```
#include <fcntl.h>
int creat(const char *pathname mode_t mode);
```

Создание файла с разрешениями доступа, заданными в **mode**

- Итоговые разрешения создаваемого файла определяются путем наложения на запрашиваемые процессом разрешения переменной **umask**

4.4. Закрытие файла (close)

```
# include <unistd.h>
int close(int fd);
```

Разрывает связь файлового дескриптора и файла на ФС. Возвращает 0 в случае успеха, -1 при ошибке, но файловый дескриптор даже при неудачном закрытии не остается открытым.

4.5. Чтение из файла (read)

- Пытается прочитать из потока заданное количество данных. Если данные отсутствуют - будет выполнена блокировка процесса до того момента, как данные появятся. Если данные есть - но их меньше, чем запрошено - вызов возвращает выполнение процессу.

```
#include <unistd.h>
ssize_t read(int fs, void *buf, size_t count);
```

Возвращает количество на самом деле прочитанных данных или 0, если достигнут конец файла. Поэтому результат выполнения **read** стоит проверять. При возникновении ошибки возвращает -1, а саму ошибку необходимо анализировать через переменную **errno**

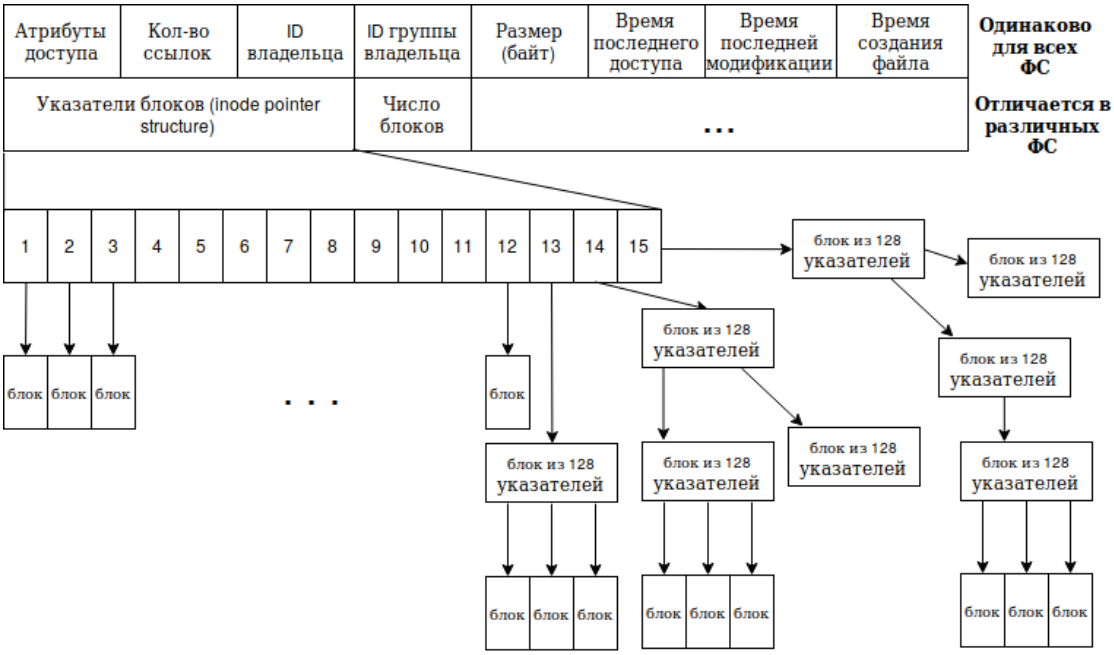

Рис. 3: Структура индексного дескриптора

4.6. Запись в файл (write)

- Записывает заданное кол-во байт, находящихся по указанному адресу в поток.

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Возвращает так же количество фактически записанных данных. Обычно это то же значение, что и **nbyte**, но возможны и исключения (заполнение диска). При возникновении ошибки возвращает -1, а саму ошибку необходимо анализировать через переменную **errno**

4.7. Перемещение файлового указателя (lseek)

- Изменение текущей позиции в файле, возвращает установленную позицию

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Перемещает позицию в файле на **offset** байт, начиная с места, определяемого параметром **whence**

Значение whence	Описание
SEEK_SET	От начала файла
SEEK_CUR	От текущей позиции
SEEK_END	От конца файла (для использования отрицательных смещений)

- Пример

```
pos = lseek(fd, 0, SEEK_CUR)
```

Ничего не поменялось, но мы узнали текущую позицию.

- Если файл открыт на запись и при помощи **lseek** мы вышли за текущие пределы файла - то его размер увеличивается, пропущенное место считается содержащим нули (дырка - hole). Место на диске на хранение нулей при этом не занимается (sparse-файлы).
- Вызов **lseek** может применяться только к т.н. **позиционируемым(seekable)** потокам, которыми например являются файлы на диске, но не относятся стандартные потоки ввода-вывода.

4.8. Интерфейс управления файловым дескриптором (fcntl)

Проверяет файл, заданный путем **path** на предмет возможности доступа методами, заданными битовой маской **amode**

Значения amode	Описание
R_OK	Файл доступен для чтения
W_OK	Файл доступен для записи
X_OK	Файл доступен для выполнения
F_OK	Файл есть

- Пример

```
if (access(path,R_OK) == 0 )
```

4.10. Управление дисковым кэшем (fsync)

- Запись данных в реальности происходит асинхронным образом, т.е. ОС принимает от вас все, что вы хотите записать, помещает в буфер и возвращает управление вызвавшему процессу. Далее через дисковый кэш эти данные когда-нибудь дойдут до накопителя.
- Вызов **fsync** запрашивает запись на диск всех данных, ассоциированных с файловым дескриптором

```
#include <unistd.h>
int fsync(int fildes);
```

- Вызов возвращает управление потоку только после завершения всех операций записи.

  Вместо этого можно открывать файл с режимом O_SYNC

```
#include <unistd.h>
int fdatasync(int fildes);
```

  Аналогично, но сбрасывает на диск только данные, без изменения метаданных файла в индексном дескрипторе.

4.11. Пример - ведение журнала (journal.c)

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

const char *logfile = "access.log";

int write_to_journal(char *entry) {
        int fd = open(logfile, O_WRONLY | O_CREAT | O_APPEND, 0600);
        write(fd, entry, strlen(entry));
        write(fd, "\n", 1);
        fsync(fd);
        close(fd);
        return 0;
```

```
}
int main() {
        if (access(logfile,F_OK|W_OK) == 0 )
                write(1,"File exists\n",12);
        else
                write(1, "No file\n",8);
        return write_to_journal("test journal message");
}
```

5. Управление объектами файловой системы

5.1. Создание жесткой ссылки (link)

```
#include <unistd.h>
int link(const char *path1, const char *path2);
```

5.2. Создание символической ссылки (symlink)

```
#include <unistd.h>
int symlink(const char *path1, const char *path2);
```

5.3. Удаление файла (unlink)

- Удаление жесткой ссылки/файла

```
#include <unistd.h>
int unlink(const char *path);
```

5.4. Переименование (rename)

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

5.5. Работа с каталогами (dirent.h)

- Открытие каталога и получение ссылки на **описатель каталога** структуру DIR по имени

```
#include <dirent.h>
DIR *opendir(const char *dirname);
```

- По файловому дискриптору

```
#include <dirent.h>
DIR *fdopendir(int fd);
```

- Путешествие по записям каталога

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

- Закрытие описателя каталога

```c
#include <dirent.h>
int closedir(DIR *dirp);
```

- Пример

```c
#include <dirent.h>
...
DIR *dir;
struct dirent *dp;
...
if ((dir = opendir (".")) == NULL) {
   perror ("Cannot open .");
   exit (1);
}

while ((dp = readdir (dir)) != NULL) {
   ...
}
```

- Структура **dirent**

```c
struct dirent
  {
    __ino_t d_ino;                // индексный дескриптор
    __off_t d_off;
    unsigned short int d_reclen;  // длина записи
    unsigned char d_type;         // тип файла
    char d_name[256];             // имя файла
  };
```

- Переименование файлов/каталогов

```c
#include <stdio.h>
int rename(const char *old, const char *new);
```

- Удаление каталога

```c
#include <unistd.h>
int rmdir(const char *path);
```

6. Высокоуровневый ввод-вывод (stdio.h)

6.1. Возможности stdio.h

- Заголовочный файл, содержащий **функции стандартного буферизованного ввода/вывода** (POSIX), включая:
  - Посимвольный ввод-вывод в стандартных потоках
  - Форматированный ввод-вывод в стандартных потоках
  - Посимвольный ввод-вывод в текстовых файлах
  - Форматированный ввод-вывод в текстовых файлах
  - Ввод-вывод отдельных строк в файлах и в памяти
  - Блочный ввод-вывод

- При использовании функций **stdio.h** буферизация, в дополнении к механизмам ОС, организуется в **стандартной библиотеке Си**, т.е. в самой программе, использующей функции буферизированного ввода-вывода.
- В сравнении с низкоуровневыми функциями ввода-вывода (обертками над системными вызовами) высокоуровневые функции позволяют переводить числа из внутреннего представления в текстовое, что полезно при работе с текстовыми файлами (чтение/запись строк и т.п.).

6.2. Ввод одиночного символа - getchar

- Читает одиночный символ, возвращает его код (от 0 до 255) или EOF=-1 если конец файла (Ctrl-D)

```
int getchar();
```

Примечание: возвращаемое значение нельзя сразу присваивать переменной типа char, нужно вначале проверить на EOF

6.2.1. Пример

- Подсчет кол-ва срок с stdin, вывод результат на stdout

```
// strcount.c
int main() {
    int ch;
    int lines = 0;
    while((ch = getchar()) != EOF) {
        if(ch == '\n')
            lines += 1;
    }
    printf("%d\n",lines);
    return 0;
}
```

Примечание: Строка, не заканчивающаяся символом переноса строки “\n” не считается.

6.3. Вывод одиночного символа

- Выводит на **stdout** символ, код которого передан в параметре, биты за пределами диапазона 0..255 будут отброшены. Возвращает код введенного символа, или EOF при ошибке.

```
int putchar(int c);
```

6.3.1. Пример

- Дублирует вводимые с stdin символы на stdout

```
// chardup.c
int main() {
    int ch;
    while ((ch = getchar()) != EOF) {
        putchar(ch);
    }
```

```
    return 0;
}
```

- Преобразование XOR над входными данными

```
// xor.c
int main() {
    char enc_key[] = "SecretKey";
        int key_len = strlen(enc_key);
        int ch, num = 0;
        while((ch = getchar()) != EOF) {
                num += 1;
                putchar(ch ^ enc_key[num%key_len]);
        }
        return 0;
}
```

6.4. Форматированный вывод - printf

- Функция - интерпретатор форматной строки

```
int printf(const char *format, . . .)
```

6.5. Спецификаторы форматированного ввода-вывода

Символ	Описание
d	знаковое целое десятичное
s	строка (параметр - адрес)

6.6. Форматированный ввод - scanf

```
int scanf(const char *format, . . .)
```

6.7. Ввод-вывод с текстовыми файлами

- Принципы ввода-вывода с текстовыми файлами аналогичны вводу-выводу в стандартные потоки, за исключением того, что перед началом работы файл нужно открыть, создав тем самым новый поток ввода-вывода.
- Тип FILE* - идентификатор потока (файловая переменная), можно работать как с обычной переменной - присваивать, передавать, возвращать.
- Любую функцию, расчитанную на работу с потоком типа FILE. *В заголовочном файле **stdio.h** определены для этого 3 следующие глобальные переменные типа FILE*:
  - **stdio**
  - **stdout**
  - **stderr**

6.8. Открытие файла (высокоуровневое)

- Открытие файла, заданного по имени *name* абсолютным или относительным путем и с режимом *mode*. Возвращает файловую переменную или NULL в случае ошибки

```
FILE* fopen(const char *name, const char *mode);
```

Режим fopen()	Описание
r	только для чтения
r+	чтение и запись с начала файла
w	запись, файл создается, существующий уничтожается
w+	запись и чтение, файл создается, существующий уничтожается
a	запись добавлением информации в конец
a+	чтение с начала файла, запись в конец файла

Для совместимости с стандартами ISO C поддерживается добавление буквы **b** к спецификации режима, например **rb+** или **r+b**. В UNIX-системах ни на что не влияет

6.9. Закрытие файла (высокоуровневое)

- Операция, обратная открытию файла

```
int fclose(FILE *f);
```

Высвобождаются ресурсы ОС, обеспечивающие работу данного потока ввода-вывода. Возвращает 0 в случае успеха и -1 в случае неудачи. И в том и в другом случае поток закрывается.

6.10. Обработка ошибок для функций обращения к ОС (errno.h)

- Принцип обработки ошибочных ситуаций в стандартной библиотеке языка С:
  - в **libc**, т.е. в каждой Си-шной программе есть глобальная переменная **errno**
  - в нее в виде целого числа заносится код ошибки
  - для доступа к этой переменной необходимо использовать заголовочный файл
- Значения **errno**
  - имеют символические имена (константы)
  - свои для каждой библиотечной функции или системного вызова
  - описаны в справке (man)

errno для fopen()	Описание
ENOENT	Файл отсутствует, а открывался по r или r+, или отсуствует каталог в пути
EACCES	Не хватает прав для запрошенного открытия
EROFS	Пытаемся открыть на запись файл на read-only ФС

- Обработка ошибок
  - для системного ПО: чем ближе к возникновению ошибки тем лучше

– обычно подразумевается индивидуальная обработка нескольких наиболее типичных кодов, остальное - общие действия (вывод сообщения, завыершения работы или текущего пакета операций и т.п.)

```
char *strerror(int errnum)
```

Возвращает адрес строки описания для указанного кода ошибки

- Поскольку ошибки принято писать на стандартный поток ошибок **stderr**, то для задач вывода строки с ошибкой лучше использовать функции, которые работают именно с ним (perror), или самостоятельно выводить в **stderr**
- Выдает в поток **stderr** объект(сущность) с которой произошла ошибка и строку сообщения на основании переменной **errno**. Выдаваемые сообщения локализованы

```
#include <stdio.h>
void perror(const char *s);
```

s - та сущность, с которой произошла ошибка (имя файла, функции)

6.10.1. Пример

- Открываем файл на чтение и запись, если такого файла нет, то создаем и открываем на чтение и запись

```
int main() {
        FILE *f;
        f = fopen("file.txt", "r+");
        if (!f) {
                if (errno == ENOENT) {
                        f = fopen("file.txt", "w+");
                        printf("No file, creating\n");
                } else {
                        perror("file.txt");
                        return 1;
                }
        }
        fclose(f);
        return 0;
}
```

6.11. Ввод-вывод отдельных символов в файлах

- После открытия потока ввода-вывода, над ним можно выполнять тот же набор действий, что и со стандартными потоками
- Для ввода и вывода отдельных символов используются 2 функции, аналогичные **getchar** и **putchar**,но принимающие идентификатор потока в качестве параметра

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

- При посимвольной обработке информации полезной так же является функция, которая возвращает в поток только-что прочитанный символ, т.е. возвращает указатель на поток на один символ назад

```
int ungetc(int c, FILE *stream);
```

Например, данная функция может быть полезна при написании ПО типа лексических анализаторов

6.12. Форматированный ввод-вывод в файлах

- Отличаются от **printf** и **scanf** добавлением первого параметра - идентификатора файла

```
int fprintf(FILE *f, const char *format, . . .)
int fscanf(FILE *f, const char *format, . . .)
```

6.13. Ввод-вывод строк в потоках

```
int fputs(const char *s, FILE *stream);
int *fgets(char *s, int s, FILE *stream);
```

6.14. Блочный ввод-вывод

- Предназначены для работы с файлами, рассматриваемыми как массивы произвольных данных
- Чтение по адресу **ptr** данных в количестве **ntimes**, размера **size** каждый.

```
#include <stdio.h>
size_t fread(void *restrict ptr, size_t size, size_t nitems,
           FILE *restrict stream);
```