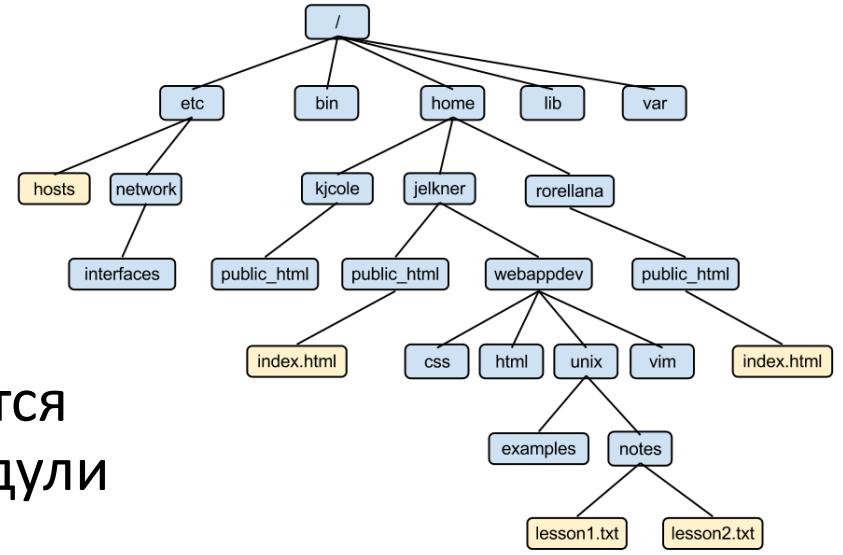
Системное программирование

Тема 2

Файлы в UNIX

Файлы

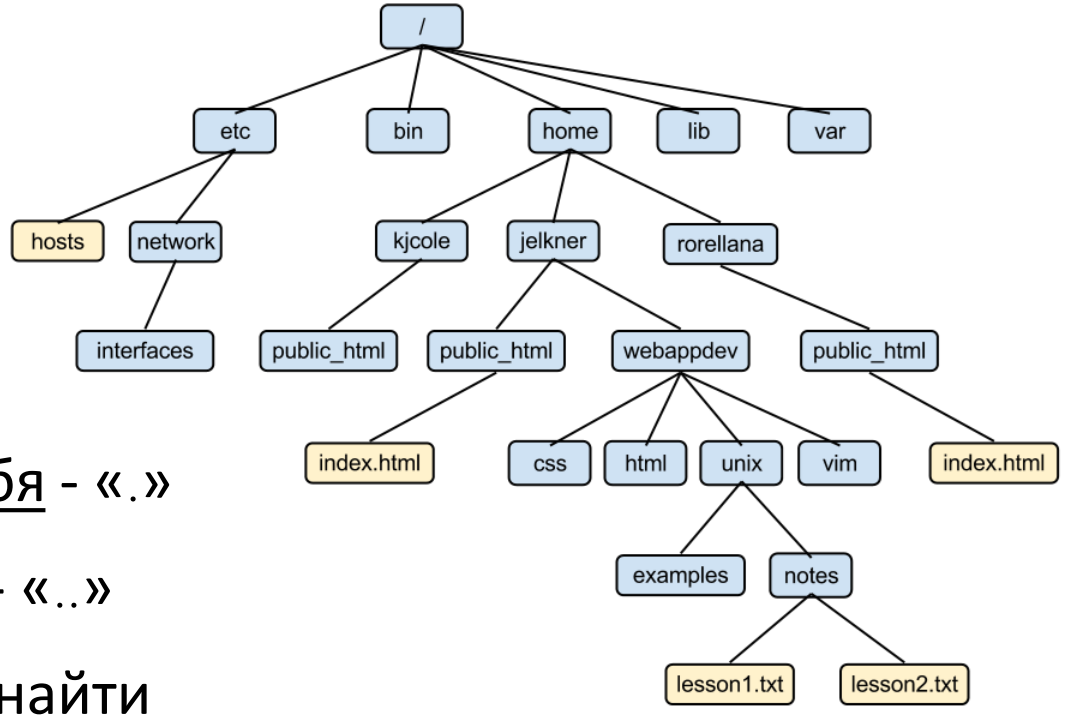


Файл – именованный набор данных.

- В UNIX «**все есть файл» -** в виде файлов представляются устройства, запущенные программы и некоторые модули ядра.

Файловая система UNIX

- Структура файловой системы — дерево.
- Корень дерева — каталог с именем «/».
- Из корневого каталога есть путь до любого файла/каталога (**абсолютный путь**).
- Любой каталог содержит ссылку на самого себя - «.»
- Любой каталог содержит ссылку на родителя - «..»
- Из-за наличия «..» из любого каталога можно найти путь до любого файла (результирующее имя не должно превышать PATH_MAX).



Основные каталоги

/bin – основные исполняемые файлы

/sbin – системные исполняемые файлы

/lib – основные библиотеки

/boot – загрузочные файлы

/dev – файлы устройств

/home – домашние каталогов пользователей

/root – каталог суперпользователя root

/tmp – временные файлы

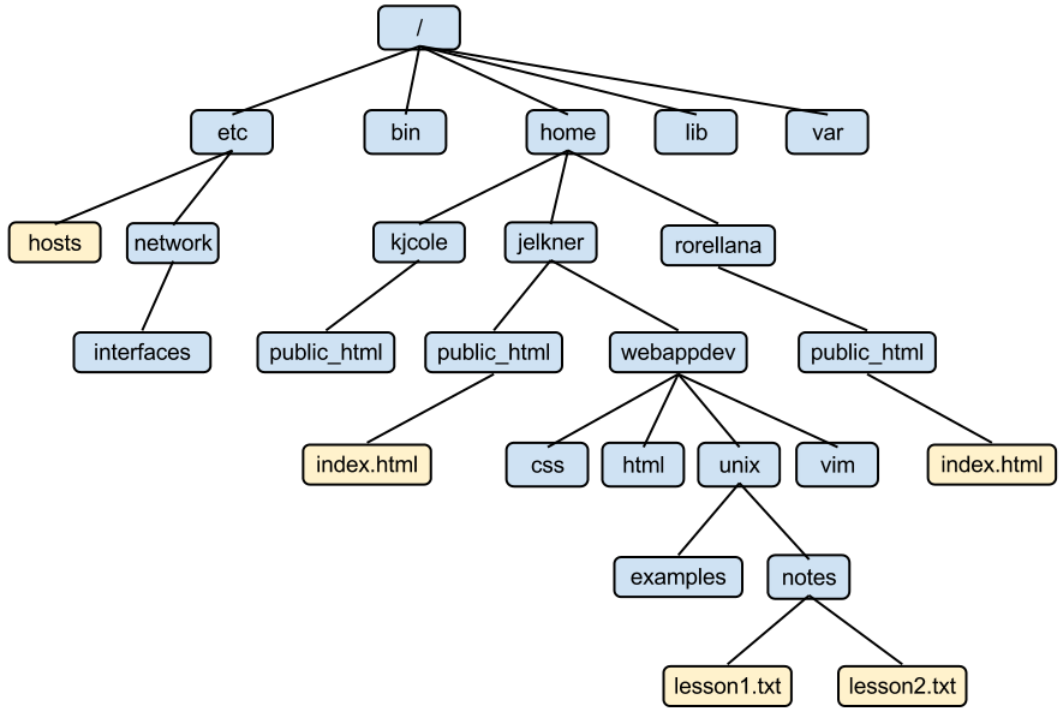
/run – системные временные файлы

/media, /run/media – точки монтирования съемных носителей

/mnt – иные точки монтирования

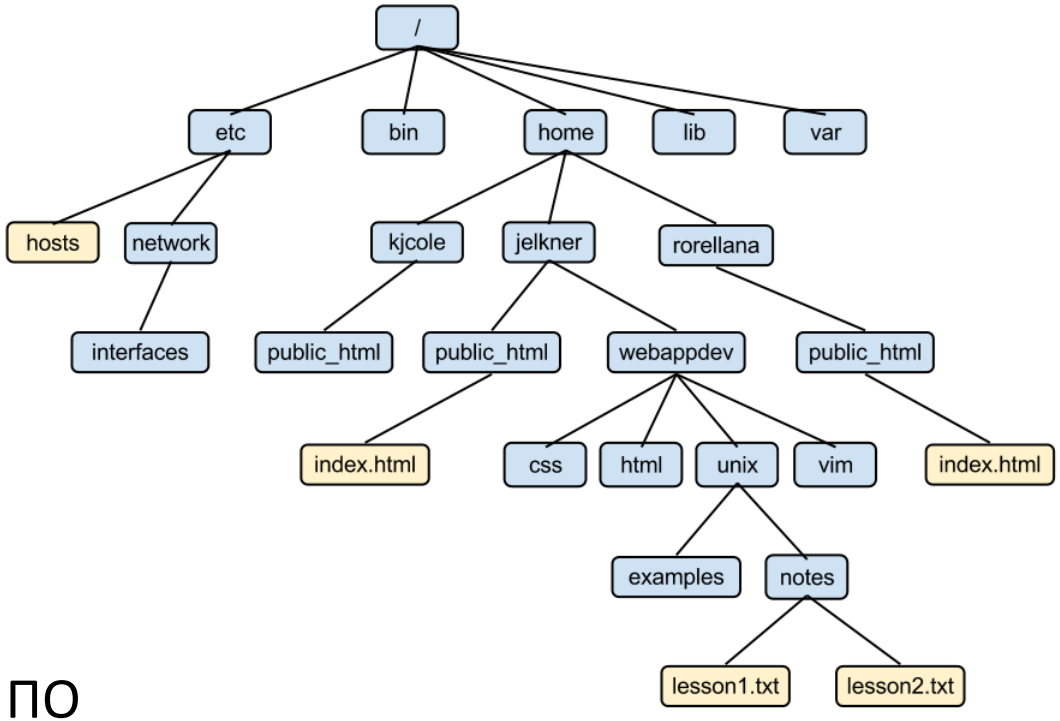
/sys – информация о системе

/proc – информация о процессах



См. стандарт FHS

Основные каталоги



/usr – **каталог установки системного ПО**

/usr/(bin, lib, sbin) – **аналогично** /(bin, lib, sbin)

/usr/share – **неизменяемые данные приложений**

/usr/include – **заголовочные файлы библиотек**

/usr/local/* – **каталог локальной установки ПО**

/usr/local/etc – **файлы конфигурации локального ПО**

/opt – **каталог установки дополнительного ПО**

/var – **данные приложений**

/var/log – **логи**

/var/opt – **данные дополнительного ПО**

/etc – **конфигурационные файлы**

/etc/opt – **конфигурационные файлы дополнительного ПО**

Права доступа к файлу

- Каждый файл имеет пользователя-владельца и группу-владельца.
- Существует 3 основных типа прав доступа к файлу — право на чтение (**r**), право на запись (**w**) и право на выполнение (**x**, запуск программы из файла).

  *Примечание: для каталогов право на выполнение (**x**) означает право на доступ к элементам каталога.*
- Существует 3 группы прав доступа — права владельца файла, права членов группы-владельца файла, права всех остальных пользователей.
- Права кодируются либо в виде последовательности букв (rwxrwxrwx), либо в виде восьмеричного числа (777)

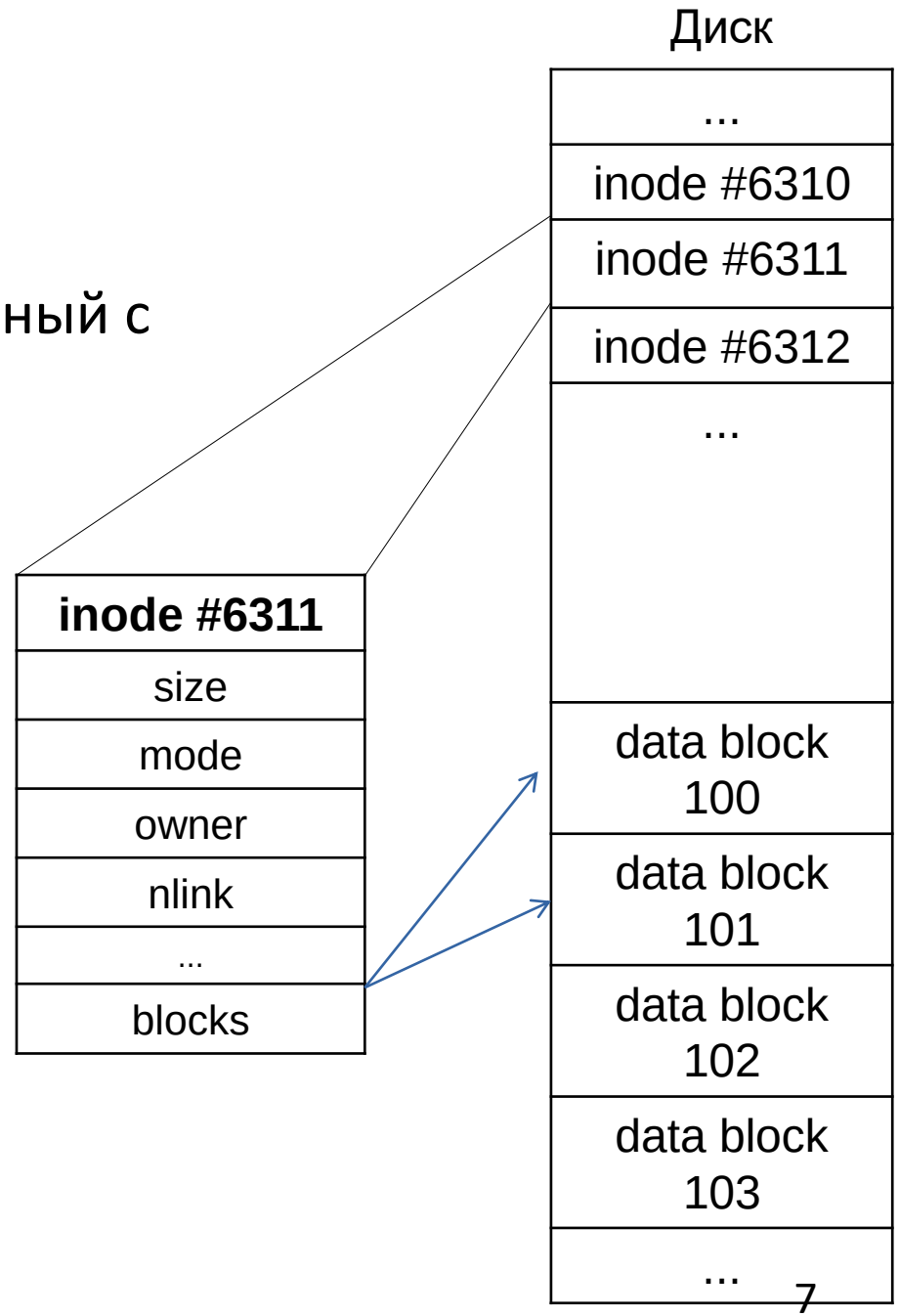
```
$ ls -l
-rwxr-xr-x 5 user group  4096 Sep  7 19:31 файл
   7   5   5
```

inode

Всякий файл имеет единственный ассоциированный с ним **индексный узел** (**inode**).

В inode хранятся основные свойства файла:

- число жестких ссылок (поле nlink);
- размер файла (поле size);
- флаги доступа (поле mode);
- владелец файла и группа-владелец;
- время последнего доступа и изменения
- ссылка на список блоков с данными и др.



Каталоги

Каталог — файл специального вида, хранящий ссылки на другие (вложенные) файлы.

- На диске каталоги хранят пары <имя файла, номер inode файла>.
- Каталоги служат для сопоставления имени файла с данными.
- Сопоставление имени начинается либо с корневого каталога «/», которому соответствует inode с номером 2, либо с рабочего каталога, ассоциированного с процессом и inode которого также известен.

Каталоги

/dir1/foo.txt

/bar.txt

/dir2/tmp

Жесткие ссылки

Жесткая ссылка — запись в каталоге.

- На один и тот же inode могут ссылаться несколько записей в каталогах => на один и тот же файл может быть несколько жестких ссылок.
- Все жесткие ссылки на файл равноправны.
- Множественные жесткие ссылки на каталоги запрещены (риск бесконечных циклов в файловой системе).
- Данные файла удаляются с диска только тогда, когда исчезает последняя жесткая ссылка на файл.

Символические ссылки

Символическая ссылка — специальный файл, хранящий путь к связанному файлу и позволяющий взаимодействовать с ним прозрачно для пользователя.

- В отличие от жесткой ссылки, символическая ссылка — самостоятельный файл.
- Символические ссылки не ограничивают удаление файла => они могут быть “висячими” — указывать на несуществующий файл.

Файлы устройств

Некоторые устройства в UNIX представляются в виде **файлов устройств** (каталог /dev).

- Устройства делятся на блочные (буферизованы, случайный доступ) и символьные (небуферизованы, обычно последовательный доступ).
- Чтение/запись файла устройства соответствуют чтению/записи с устройства напрямую.

Примеры: /dev/sd* - **жесткие диски/SSD,** /dev/nvme* - **NVME-SSD,** /dev/mem – **физическая ОЗУ**

Псевдоустройства

Некоторые файлы, похожие на файлы устройств, на самом деле не связаны ни с одним физическим устройством. Такие файлы называются **файлами псевдоустройсв**. Примеры:

- /dev/random — генератор случайных чисел;
- /dev/urandom — генератор псевдослучайных чисел;
- /dev/zero — бесконечное устройство, заполненное 0;
- /dev/full — “всегда полное” устройство, в которое нельзя ничего записать;
- /dev/null — “всегда пустое” устройство, из которого нельзя ничего прочитать;
- /dev/tty — псевдотерминал процесса.

Специальные файлы

Помимо обычных файлов, каталогов и файлов устройств есть ряд других объектов, которые могут представляться как файлы.

- каналы (FIFO, см. *mkfifo*)
- очереди сообщений
- сокеты
- файлы разделяемой памяти

Вызов open

Для того, чтобы работать с файлом внутри программы, его необходимо сначала открыть вызовом **open**.

```
int fd = open(const char *pathname, int flags, mode_t mode);
```

Параметры:

`pathname` - путь к файлу,
`flags` - режим доступа к файлу и дополнительные флаги,
`mode` - флаги прав доступа к создаваемому файлу.

Вызов open вызвращает -1 в случае ошибки. В случае успешного выполнения возвращается **дескриптор файла**.

Флаги open

Комбинация следующих флагов может быть передана в параметре flags.

- O_RDONLY, O_WRONLY, O_RDWR – открыть файл на чтение/на запись/на чтение-запись (указать можно только один из флагов).
- O_CREAT – создать файл, если он не существует;
- O_EXCL – вернуть ошибку, если указан O_CREAT, но файл уже есть;
- O_TRUNC — если файл существует, то стереть весь его контент;
- O_APPEND — *всегда* производить запись в конец файла.
- O_NOFOLLOW — если файл является символической ссылкой, открыть сам файл ссылки.
- O_SYNC – немедленно записывать все изменения в файле на диск.

Константы доступа

Если указан флаг O_CREAT, то в параметре должны передаваться права доступа к создаваемому файлу. Права задаются комбинацией следующих констант.

Права	**Владельца**	**Группы**	**Остальных**
На чтение	S_IRUSR	S_IRGRP	S_IROTH
На запись	S_IWUSR	S_IWGRP	S_IWOTH
На выполнение	S_IXUSR	S_IXGRP	S_IXOTH
Все	S_IRWXU	S_IRWXG	S_IRWXO

`S_IRWXU | S_IRGRP` — какой набор прав?

Вызов close

После завершения работы с *дескриптором* его необходимо закрыть вызовом close.

```
int close(int fd);
```

Сам файл будет окончательно закрыт тогда, когда будут закрыты все связанные с ним дескрипторы.

Все дескрипторы автоматически закрываются, когда завершается программа.

Таблицы дескрипторов и открытых файлов

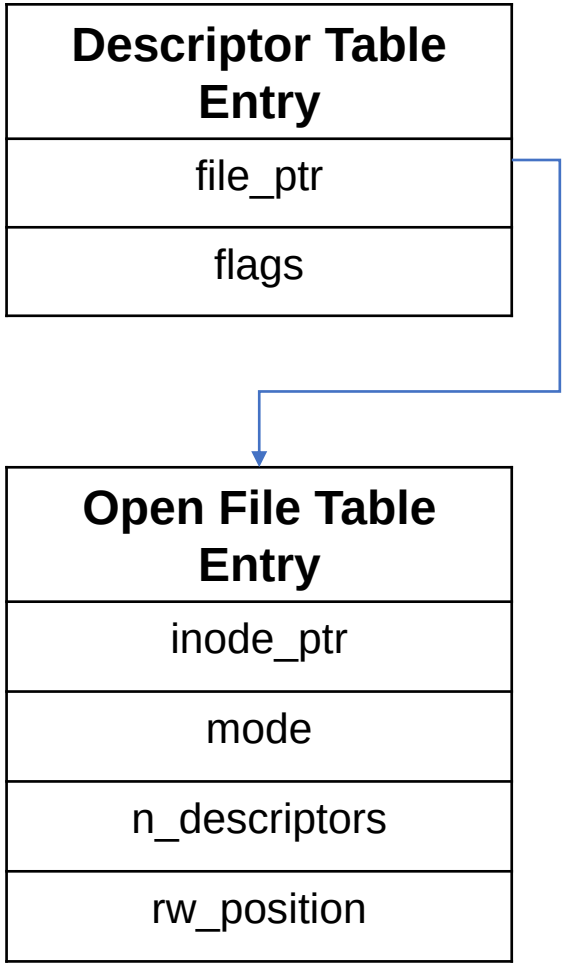
Для каждого процесса в пространстве ядра создается **таблица дескрипторов. Дескриптор**, возвращаемый вызовом open – это индекс в таблице дескрипторов процесса.

Для каждого открытого файла существует запись в общесистемной **таблице открытых файлов**.

Записи в таблице дескрипторов указывают на записи в таблице открытых файлов.

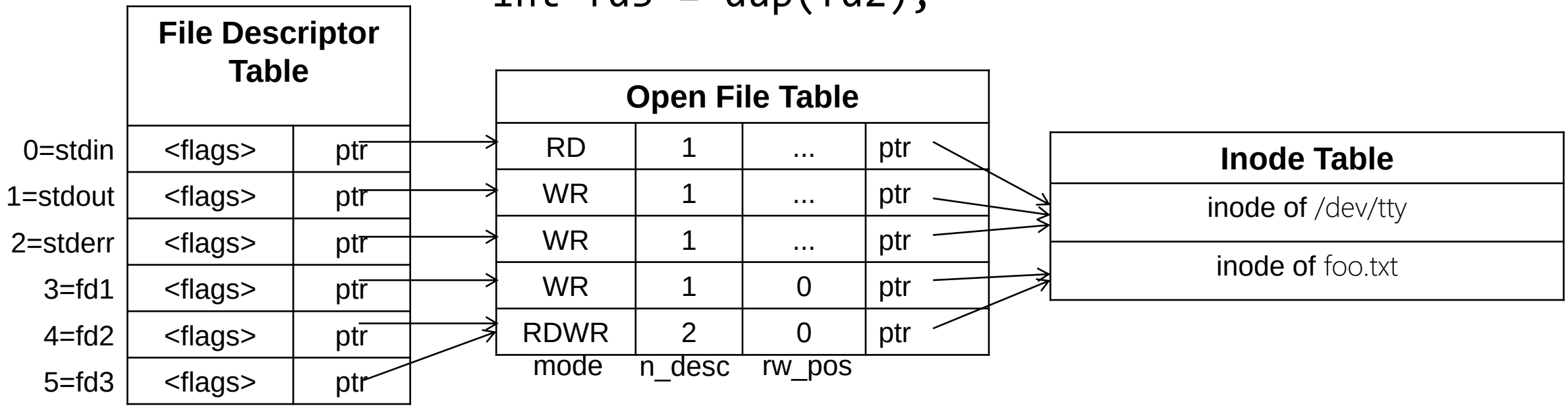
Вызов **`open()`** создает новую запись в таблице открытых файлов и соответствующую ей запись в таблице дескрипторов.

Вызов **`close()`** закрывает запись в таблице дескрипторов. Запись в таблице открытых файлов закрывается только когда не останется связанных с ней дескрипторов.



Таблицы дескрипторов и открытых файлов

```
int fd1 = open(«foo.txt», O_WRONLY);
int fd2 = open(«foo.txt», O_RDWR);
int fd3 = dup(fd2);
```



Маска прав доступа. Вызов umask

В некоторых случаях нежелательно, чтобы программа создавала файлы с определенными разрешениями.

Разрешения, которые *не нужно* устанавливать при создании файла, задаются **маской прав доступа**.

Маска прав доступа по умолчанию имеет значение 022 (соответствует ----wx-wx).

Изменяется маска прав вызовом `umask()` и утилитой *`umask`*.

```
mode_t umask(mode_t mask);
```

Вызов возвращает предыдущую маску прав или -1 в случае ошибки.

```
//с какими правами доступа создастся file?
umask(S_IWOTH|S_IXUSR);
int fd = open("file", O_RDWR|O_CREAT, S_IRWXU|S_IRWXO);
```

Вызовы read и write (примеры 1, 2)

Чтение и запись в файл производятся вызовами `read` и `write`.

```
ssize_t read(int fd, void* buf, size_t count);
ssize_t write(int fd, const void* buf, size_t count);
```

Параметры:

`fd` – дескриптор открытого файла,
`buf` – данные для записи либо буфер для чтения данных,
`count` – размер данных, которые требуется прочитать/записать.

Вызовы возвращают число *фактически* прочитанных/записанных байт (0 означает конец файла, -1 означает ошибку).

Позиция чтения/записи. Вызов lseek (примеры 2,3)

С каждым открытым файлом ассоциирована **позиция чтения/записи** — позиция в файле, с которой начнется следующая операция.

Позиция автоматически изменяется вызовами `read/write`. Изменить позицию можно вызовом `lseek`:

Open File Table Entry
inode_ptr
mode
n_descriptors
rw_position

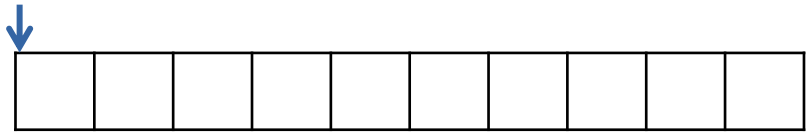
```
off_t lseek(int fd, off_t offset, int whence);
```

Параметры:

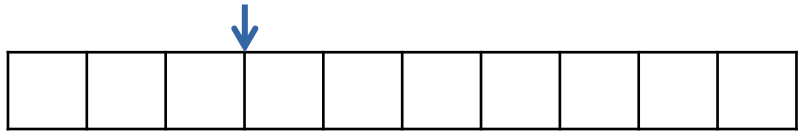
- `fd` – дескриптор файла,
- `offset` – величина смещения позиции,
- `whence` - одно из значений SEEK_SET, SEEK_CUR, SEEK_END

Позиция чтения/записи. Вызов lseek

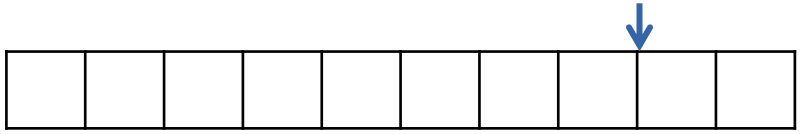
```
int fd = open(“foo.txt”, O_RDWR);
```



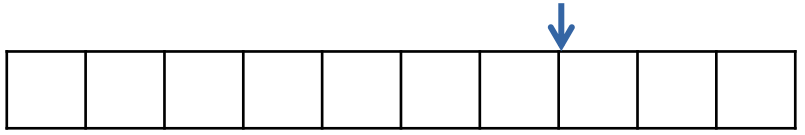
```
off_t off1 = lseek(fd, 3, SEEK_SET);
```



```
off_t off2 = lseek(fd, 5, SEEK_CUR);
```



```
off_t off3 = lseek(fd, -3, SEEK_END)
```

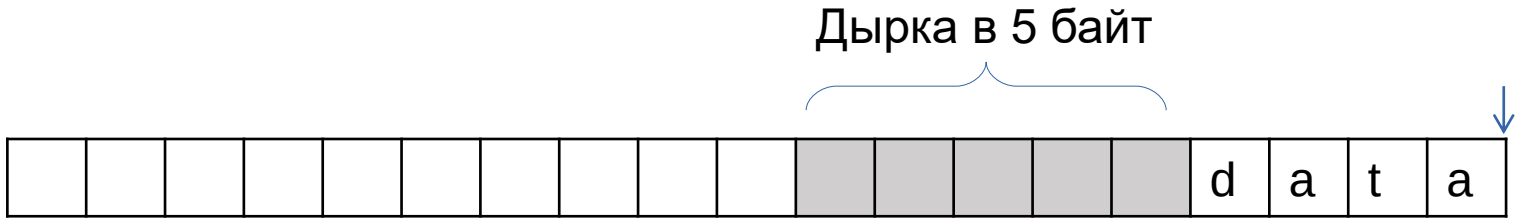


Дырки в файлах

Дырка в файле – область файла, физически отсутствующая на диске.

Чтение в пределах дырки возвращает набор 0, запись – заполняет дырку.

```
int fd = open(“foo.txt”, O_RDWR); // файл размером 10 байт
off_t off = lseek(fd, 5, SEEK_END);
int c = write(fd, “data”, 4);
```



Вызов ftruncate

Для изменения размера файла можно использовать вызов `ftruncate`

```
int ftruncate(int fd, off_t length);
```

Вызов возвращает 0 в случае успеха и -1 в случае ошибки.

Если размер файла увеличивается в большую сторону, в конце файла образуется дырка.

Свойства файла. Вызов fstat (пример 3)

Для получения информации о файле используется вызов `stat/fstat`.

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Вызов возвращает 0 в случае успеха и -1 в случае ошибки.

Вызов заполняет структуру, адрес которой передается в `statbuf`.

Свойства файла. Вызов fstat

```
struct stat {
    dev_t     st_dev;   /* ID устройства хранения */
    ino_t     st_ino;   /* Номер inode файла */
    mode_t    st_mode;  /* Тип файла и флаги доступа */
    nlink_t   st_nlink; /* Количество жестких ссылок */
    uid_t     st_uid;   /* ID пользователя-владельца */
    gid_t     st_gid;   /* ID группы-владельца */
    off_t     st_size;      /* Размер */
    off_t     st_blksize;  /* Размер в блоках по 512 байт*/
    time_t    st_atime; /* Время последнего доступа */
    time_t    st_mtime; /* Время последнего изменения */
    time_t    st_ctime; /* Время последнего изменения inode */
}
```

Свойства файла. Вызов fstat

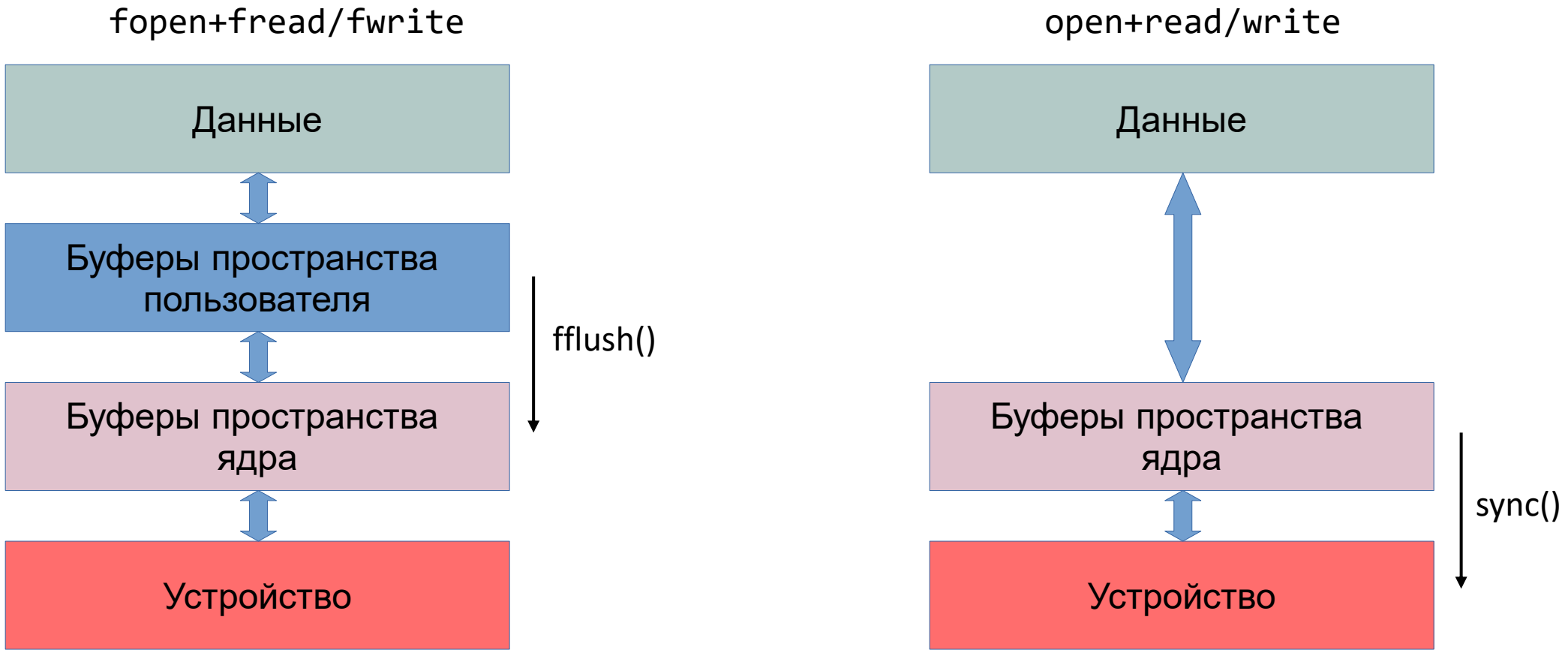
Для проверки типа файла используется ряд функций-макросов:

- `S_ISREG(st_mode)` — обычный файл;
- `S_ISDIR(st_mode )` — каталог;
- `S_ISCHR(st_mode )` — файл символьного устройства;
- `S_ISBLK(st_mode )` — файл блочного устройства;
- `S_ISLNK(st_mode )` — символическая ссылка.
- `S_ISFIFO(st_mode )` — канал.

Для извлечения только флагов доступа используется выражение `st_mode & 0777`

```
struct stat s;
stat("foo.txt", &s); //аналогично open+fstat+close
printf(«Is dir? %d", S_ISDIR(s.st_mode));
printf(«Acess mode %o", (s.st_mode & 0777));
```

Буферизация ввода-вывода (пример 4)



Переименование файлов. Вызов rename

Файл может быть переименован вызовом rename. С помощью этого же вызова файл может быть перемещен в другой каталог.

```
int rename(const char *oldpath, const char *newpath);
```

Вызов возвращает 0 в случае успеха, -1 в случае ошибки.

При перемещении в пределах одной файловой системы (например, в пределах одного раздела диска), rename не производит копирования данных.

Перемещение уже открытого файла не сказывается на работе с ним.

Удаление файлов. Вызов unlink

Файл (*но не каталог*) может быть удален вызовом `unlink`.

```
int unlink(const char *pathname);
```

Вызов возвращает 0 в случае успеха, -1 в случае ошибки.

Файл не будет удален с диска, пока на него есть жесткие ссылки или пока он открыт в программе.

Ссылки. Вызовы link и symlink

Жесткая ссылка на файл создается вызовом `link`.

```
int link(const char *target, const char *linkpath);
```

Символическая (мягкая) ссылка создается вызовом `symlink`.

```
int symlink(const char *target, const char *linkpath);
```

Вызовы возвращают 0 в случае успеха, -1 в случае ошибки.

В случае `link` путь в `target` обязан существовать, в случае `symlink` — не обязан.

Создание и удаление каталогов

Для создания каталога используется вызов `mkdir()`.

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

Параметр `mode` указывает флаги доступа к создаваемому каталогу (S_IRXU, S_IRWXG, ...).

Для удаления каталога используется вызов `rmdir()`.

```
int rmdir(const char* path);
```

Каталог должен быть пустым.

Рабочий каталог. Вызов chdir

У каждой выполняющейся программы есть **рабочий каталог** — каталог, от которого отсчитываются относительные пути.

Относительный путь — путь, не начинающийся с корня файловой системы (с /).

Узнать рабочий каталог можно вызовом `getcwd()`, изменить — вызовом `chdir()`:

```
char* getcwd(char* buf, size_t size);
int chdir(const char *path);
```

Вызов `chdir` возвращает -1 в случае ошибки и 0 — в случае успеха.
Вызов `getcwd` возвращает `buf` в случае успеха и NULL в случае ошибки.

Просмотр каталогов. Функция scandir

Для получения набора элементов каталога используется функция `scandir()` (см. след слайд).

```
int scandir(…);                          /* см. след. слайд */
struct dirent{
        ino_t  d_ino;                    /* Номер inode файла*/
        char d_name[];                   /* Имя файла */
};
```

Функция возвращает количество полученных элементов каталога или -1.

Стоит посмотреть: функции opendir, readdir, telldir, seekdir, rewinddir, getdents64

Просмотр каталогов. Функция scandir

```
int scandir(const char *path,
      dirent*** namelist,
      int (*filter)(const dirent *),
      int (*compar)(const dirent **, const dirent **)    );
```

Параметры:

`path` – путь к каталогу,
`namelist` – адрес, по которому будет записан результат,
`filter` – функция, фильтрующая элементы каталога (можно передать NULL),
`compar` – функция, сортирующая элементы каталога.

В качестве параметра `compar` можно передать функцию `alphasort()`.
Массив, возвращаемый в `namelist`, нужно удалить после использования.

Права доступа к файлу. Вызов chmod

Изменение флагов доступа к существующему файлу осуществляется вызовом chmod.

```
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

В качестве параметра mode указываются те же константы, что и в вызове open.

Вызов возвращает -1 в случае ошибки и 0 – в случае успеха.

Примечание 1: нет отдельного системного вызова для получения флагов доступа к файлу, получить флаги можно вызовом `stat()`.

Примечание 2: маска прав доступа не оказывает влияния на `chmod()`.

Константы доступа

Если указан флаг O_CREAT, то в параметре должны передаваться права доступа к создаваемому файлу. Права задаются комбинацией следующих констант.

Права	**Владельца**	**Группы**	**Остальных**
На чтение	S_IRUSR	S_IRGRP	S_IROTH
На запись	S_IWUSR	S_IRGRP	S_IWOTH
На выполнение	S_IXUSR	S_IXGRP	S_IXOTH
Все	S_IRWXU	S_IRWXG	S_IRWXO

Блокировка файлов

Блокировка файла – ограничение доступа к файлу или его части.

В POSIX определен единственный тип блокировок – advisory locks (рекомендательные/необязательные блокировки, далее – просто блокировки).

Данные блокировки *не позволяют* полностью запретить доступ к файлу другим процессам. Но другой процесс может проверить блокировку и узнать, наложена блокировка или нет.

Блокировка снимается автоматически, если файл закрывается.

Примечание: с помощью вызова fcntl можно блокировать не весь файл, а только его часть.

В некоторых ОС (в т.ч. Linux) вводится еще один тип блокировок – mandatory locks (обязательные блокировки). Данные блокировки полностью запрещают доступ к файлу другим процессам, однако их реализация на уровне ОС имеет ряд недостатков, и потому их использование нежелательно. Подобные блокировки могут налагаться только через `fcntl`.

Блокировка файлов. Вызов flock (пример 5)

Наложить блокировку на файл можно вызовом `flock()`.

```
int flock(int fd, int operation);
```

В поле `operation` должна передаваться одна из констант, определяющих тип операции:

- `LOCK_SH` – наложить разделяемую блокировку (блокировку для чтения);
- `LOCK_EX` – наложить исключительную блокировку (блокировку для записи);
- `LOCK_UN` – снять блокировку.

Если на файл уже наложена несовместимая блокировка, вызов приостанавливает программу до получения блокировки.

Если дополнительно передать флаг `LOCK_NB`, то, вместо приостановки программы, вызов вернет -1 с ошибкой `EWOULDBLOCK`.

Предотвращение параллельного запуска

Вызов `flock()` используется для запрета параллельного запуска программ.

Системные программы обычно поступают следующим образом:

1. Открывают или создают файл /run/<имя программы>.pid (или по другому фиксированному пути).
2. Вызывают `flock()` с флагами `LOCK_EX|LOCK_NB` .
3. Если блокировка наложена успешно – записывают в файл текущий идентификатор процесса (см. вызов `getpid()`) и продолжают выполнение.
4. Если блокировка не наложена – читают из файла идентификатор блокирующего процесса, выводят сообщение об ошибке и завершаются.

Вызов fcntl

Для получения и изменения некоторых свойств открытого файла используется вызов `fcntl`.

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

Параметр cmd определяет одновременно и свойство, и действие, которое будет производиться (получение или изменение значения свойства).

Возвращаемое значение также зависит от cmd, но в любом случае при ошибке возвращается -1.

Возможные значения *cmd* перечислены в документации (*man fcntl*). Среди возможностей - наложение блокировок на часть файл, наложение строгих блокировок, управление асинхронным вводом/выводом и т.д.

Файлы устройств

Файлы устройств соответствуют реальным устройствам и обычно располагаются в каталоге */dev*.

Каждый файл устройства имеет старший и младший номер (см. вывод *stat*).

Старший номер устройства идентифицирует модуль ядра, который отвечает за работу с устройством.

Младший номер устройства идентифицирует устройство.

```
$ stat /dev/sda

  File: /dev/sda
  Size: 0               Blocks: 0          IO Block: 4096   block special file
Device: 5h/5d   Inode: 152         Links: 1     Device type: 8,0
Access: (0660/brw-rw----)  Uid: (    0/    root)   Gid: (    6/    disk)
Access: 2024-02-11 21:22:57.939572272 +0400
Modify: 2024-02-11 21:22:57.939572272 +0400
Change: 2024-02-11 21:22:57.939572272 +0400
```

Взаимодействие с устройствами (пример 6)

Для взаимодействия с устройством используется вызов `ioctl`().

```
int ioctl(int fd, unsigned long request, ...);
```

Параметры:

`fd` – дескриптор открытого файла устройства;
`request` – константа, определяющая операцию;

Дополнительные параметры, которые принимает вызов, зависят от конкретного устройства и указываются в документации на него.

Монтирование файловых систем

Файловая система, находящаяся на устройстве, может быть *смонтирована в каталог* с помощью вызова `mount()`. Содержимое каталога становится недоступным, но не теряется.

```
int mount(const char *source, const char *target,
          const char *filesystemtype,
          unsigned long mountflags, const void *data);
```

Параметры:

`source`	– путь к файлу устройства;
`target`	– путь к целевому каталогу;
`filesystemtype`	– имя монтируемой ФС (из */proc/filesystems*);
`mountflags`	– флаги монтирования (MS_NOEXEC, MS_RDONLY и др);
`data`	– дополнительные данные, передаваемые модулю ядра.

Для размонтирования используется вызов `int umount(const char *target);`

См. также: */proc/self/mounts*

Файловая система tmpfs

Интересной возможностью является создание файловой системы в оперативной памяти. Работа с файлами в такой файловой системе происходит намного быстрее.

Монтирование в консоли:

$ sudo mount –t tmpfs –o size=<размер>,mode=777 tmpfs <каталог>

Монтирование в программе:

```
mount(NULL, “<каталог>”,  “tmpfs”, 0, “size=<размер>,mode=777”);
```

Отслеживание изменений в файлах (пример 7)

Для отслеживания изменений в каталогах используются системные вызовы набора inotify:

```
int fd = inotify_init();

int wd = inotify_add_watch(int fd, const char *pathname,
                                                uint32_t mask);

int inotify_rm_watch(int fd, int wd);
```

Данные функции не входят в стандарт POSIX.

Инициализация отслеживания (пример 7)

```
int fd = inotify_init();
```

Вызов инициализирует очередь событий inotify и возвращает файловый дескриптор, связанный с ней (или -1 в случае ошибки).

Для чтения событий по данному дескриптору используется вызов read().

Отслеживание изменений в файлах (пример 7)

Для отслеживания событий связанных с заданным файлом используется вызов inotify_add_watch().

```
int wd = inotify_add_watch(int fd, const char *pathname,
                                                uint32_t mask);
```

Параметры:

`fd` – дескриптор очереди событий;
`pathname` – путь, события по которому отслежтиваются
`mask` – маска отслеживаемых событий.

Вызов возвращает дескриптор отслеживания, который используется для сопоставления событий. *Данный дескриптор нельзя передавать в read().*

Для отмены отслеживания используется вызов `inotify_rm_watch(int fd, int wd)`.

Типы событий inotify(пример 7)

Комбинация следующих констант может быть передана в аргументе `mask`.

`IN_OPEN` – файл/каталог или элемент каталога был открыт
`IN_CLOSE` – файл/каталог или элемент каталога закрыт
`IN_MODIFY` – файл/каталог или элемент каталога изменен
`IN_ACCESS` – файл/каталог или элемент каталога был прочитан
`IN_ATTRIB` – свойства файла/каталога или элемента каталога были изменены
`IN_CREATE` – новый элемент каталога создан
`IN_DELETE` – элемент каталога удален
`IN_DELETE_SELF` – отслеживаемый файл/каталог удален
`IN_MOVE_SELF` – отслеживаемый файл/каталог удален
`IN_MOVED_FROM` – элемент каталога перемещен из отслеживаемого каталога
`IN_MOVED_TO` – новый элемент каталога перемещен в каталог

Чтение событий inotify(пример 7)

Для чтения событий отслеживания используется системный вызов read().

Вызов считывает одну или несколько структур `inotify_event` :

```
struct inotify_event {
    int      wd;       /* Дескриптор отслеживания */
    uint32_t mask;     /* Маска, определяющая тип события*/
    uint32_t cookie;   /* Число для сопоставления событий
                          переименования*/
    uint32_t len;      /* Размер имени файла в байтах*/
    char     name[];   /* Имя файла */
};
```

Чтение событий inotify(пример 7)

Так как общий размер структуры заранее не известен, буфер для чтения событий следует резервировать с запасом. Достаточно зарезервировать буфер в виде:

```
const char buffer_size = sizeof(inotify_event) + PATH_MAX + 1;
char* buf = malloc(buffer_size);
int read_size = read(fd, buf, buffer_size);
```

Вызовы `read()` при успешном завершении считывает целиком как минимум 1 событие. При этом следует учитывать, что в буфер может быть считано более 1 события.

Если размер буфера недостаточен, вызов `read()` провалится с ошибкой `EINVAL`.

Если в очереди нет событий, то вызов `read()` заблокируется до появления события.