ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 04 — Файловая система

Преподаватель: Поденок Леонид Петрович
prep@lsi.bas-net.by
БГУИР 505а 5 к. +375 17 293 8039
ОИПИ, 116/118 +375 17 320 7402

БГУИР 2023

2023.02.15

Оглавление

Файловая система и файловый ввод-вывод

Общие понятия

Файл (Именованный файл) — некий набор хранимых на внешнем носителе данных, имеющий имя (набор данных — data set).

Это не строгое определение файла — оно лишь отражает те свойства, которые мы обычно используем на прикладном уровне.

Термин «файловая система» имеет 2 значения:

1) подсистема ОС, отвечающая за хранение информации на внешних запоминающих устройствах (ЗУ) в виде именованных файлов;

2) обозначение структур данных, создаваемых на внешнем ЗУ с целью организации хранения на этом устройстве данных в виде именованных файлов.

Имя файла — определяется алфавитом и длиной.

Каталог (Directiry) — особый тип файла, хранящий имена файлов, некоторые из которых, возможно, сами являются каталогами.

При создании на диске файловой системы создается один каталог — **корневой каталог**.

При необходимости в корневом каталоге можно создавать другие каталоги, а их имена записать в корневой — **каталоги первого уровня вложености**.

В них, в свою очередь, тоже можно создавать каталоги.

При использовании каталогов говорят о **кратком** или **локальном** имени файла (строка символов, уникальная в пределах каталога) и о **полном имени файла** или **полном пути к файлу** (строка символов, включающая имена всех каталогов, начиная с корневого и заканчивая локальным).

Имена каталогов разделяются символом «**/**».

(*URI — Uniform Resource Identificator*)

В каждом каталоге существует файл со специальным именем, обозначающим каталог более высокого уровня (родительский). Его имя «**..**».

Еще одно специальное имя — файл, описывающий сам каталог. Его имя «**.**».

При работе с файлами один из каталогов тем или иным способом объявляется **текущим (current directory)**.

К файлам из текущего каталога можно обращаться по короткому имени.

Для обращения к файлам вне текущего каталога можно использовать **полный (абсолютный)** или **относительный** путь.

Относительный путь начинается с текущего каталога.

Операционные системы различают полные и относительные пути по наличию символа-разделителя «**/**» перед первым именем в пути.

```
/home/user/music/ACDC/Back in Black/Back in Black.flac
music/ACDC/Back in Black/Back in Black.flac
```

Примечание:

При передаче пути и имени, содержащих пробелы, в качестве параметров, пробелы следует экранировать (Back\ in\ Black.flac) или заключать параметр в кавычки ('Back in Black.flac').

Файловая система ОС UNIX

- единое дерево каталогов.

- в имя файла ни в каком виде не входит имя устройства, на котором файл находится.

- если в системе присутствует несколько устройств прямого доступа (dasd), один их них объявляется **корневым (root)**, и все остальные «**монтируются» (mount)** в тот или иной каталог, называемый точкой **монтирования (mount point)**.

- для указания полного пути к файлу на примонтированном устройстве спереди к имени файла добавляется полный путь к точке монтирования. Например, если у нас есть дискета, смонтированная в **`/media/myfloppy`**, а на ней есть каталог **`work`**, а в нем файл fo**`o.c`**, то полный путь к этому файлу будет **`/media/myfloppy/work/foo.c`**.

В ОС UNIX каталоги хранят только имя файла и некоторый номер, позволяющий идентифицировать соответствующий файл.

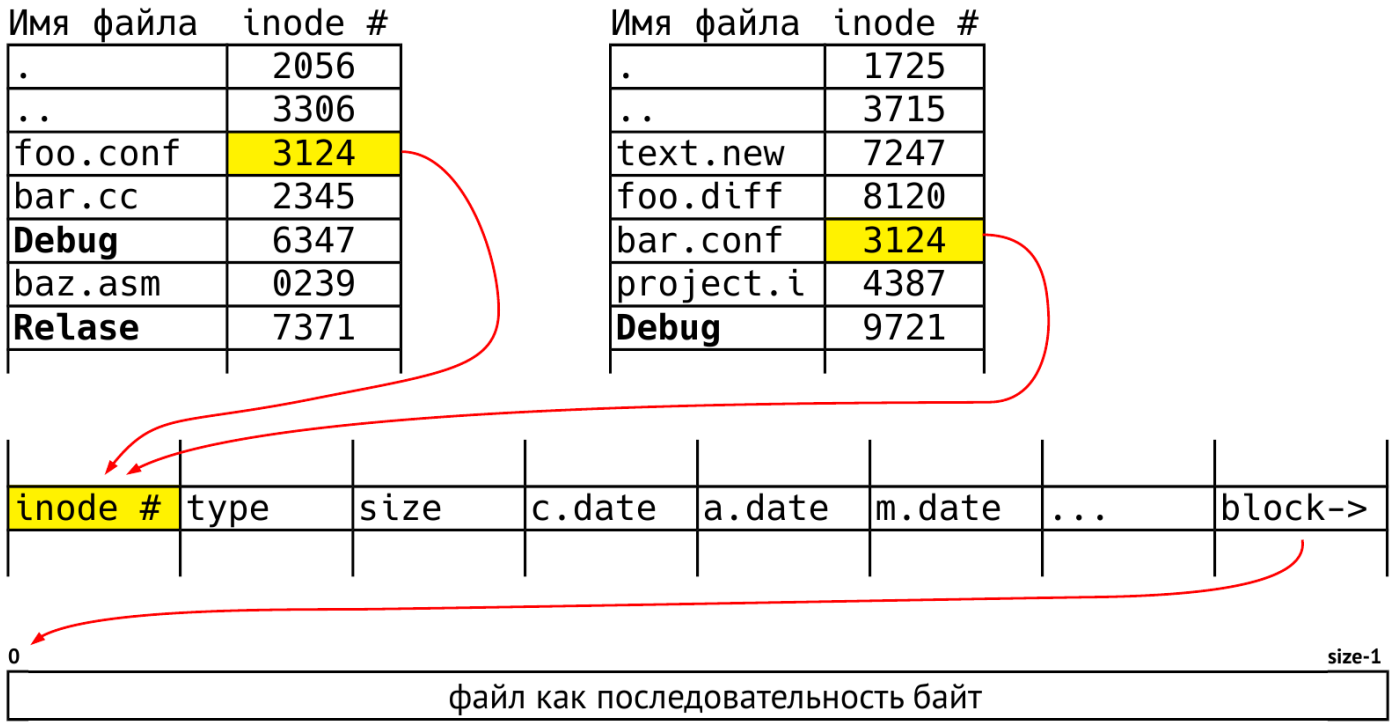
Вся остальная информация о файле, как то размер, расположение на диске, даты создания, модификации и последнего обращения, данные о владельце и о правах доступа к нему связываются не с именем файла, а с этим самым номером, который связан индексным дескриптором файла.

ИНДЕКСНЫЙ ДЕСКРИПТОР — хранимая на внешнем ЗУ (диске) структура данных, содержащая всю информацию о файле, исключая его имя.

Размер индексного дескриптора обычно составляет 128 байтов.

Допускается, чтобы несколько имен файлов, в том числе и в разных каталогах, ссылались на один и тот же номер индексного дескриптора.

Индексный дескриптор содержит:
- номер (уникальный в рамках файловой системы данного диска);
- тип файла;
- права доступа к файлу;
- количество связей (ссылок на файл в каталогах) файла;
- идентификатор пользователя и группы-владельца;
- размер файла в байтах;
- время последнего доступа к файлу;
- время последнего изменения файла;
- время последнего изменения индексного дескриптора файла;
- указатели на блоки данных файла (обычно 10);
- указатели на косвенные блоки (обычно 3).



Ограничения на имя файла

В имени файла допустимы любые символы ASCII за исключением разделителя каталогов «/» и нулевого символа '`\0`'.

Ограничения на длину имени существуют — обычно 255.

Многие современные ФС допускают использование UNICODE.

Единственная кодировка UNICODE, совместимая с ASCII — utf8.

Стандарт POSIX **(Portable Operating System Interface)** требует реализации поддержки двух типов связей имен с индексными дескрипторами

- жестких;

- символических.

Жесткой ссылкой (hard link) считается элемент каталога, указывающий непосредственно на некоторый индексный дескриптор. Это «настоящее имя файла».

Жесткие ссылки очень эффективны, но у них существуют определенные ограничения, так как они могут создаваться только в пределах одной физической файловой системы.

Когда создается такая ссылка, связываемый файл должен уже существовать.

Кроме того, жесткой ссылкой не могут связываться каталоги.

Таким образом, файл может иметь несколько совершенно равноправных имен.

Файл создается с помощью вызовов **`open()`** или **`create()`** всегда с одним именем. Дополнительные имена можно назначить файлу используя системный вызов **`link()`**. Этот вызов является проекцией команды оболочки **`ln`**. на

В индексном дескрипторе содержится счетчик жестких ссылок на этот дескриптор.

Создание жестких ссылок на каталоги не допускается (на уровне ядра) из-за вероятности возникновения рекурсии и зацикливания.

Создание жестких ссылок на файлы, расположенные на других устройствах, не допускается (уникальность номеров индексных дескрипторов имеет место только в пределах устройства).

Файл удаляется функцией **`unlink()`**, которая удаляет одну из жестких ссылок. Когда будет удалена последняя, файл считается удаленным и ресурсы, которые он занимает на носителе освобождаются.

Типы файлов. Символические ссылки

Каталоги — это файлы специального типа. Они хранят имена и номера индексных дескрипторов. Все, чем отличается каталог от обычного файла на низком уровне — это значение признака типа в индексном дескрипторе.

В остальном хранение каталогов на диске не отличается от хранения обычных файлов.

Кроме обычных файлов и каталогов операционные системы поддерживают и другие специальные типы файлов. FAT (MS DOS, Windows), например, поддерживает тип «метка тома» (volume label).

UNIX поддерживает достаточно большое количество разновидностей файлов специального типа:

- файлы байт-ориентированных устройств;
- файлы блок-ориентированных устройств;
- имена сокетов;
- именованные каналы (FIFO);
- символические ссылки.

Файловая система обеспечивает перенаправление запросов, адресованных периферийным устройствам, к соответствующим модулям подсистемы ввода-вывода.

Символическая ссылка (Symbolic link) — файл специального типа, содержащий путь к другому файлу.

Указание на то, что данный элемент каталога является символической ссылкой, находится в индексном дескрипторе. О*бычные* команды доступа к файлу вместо получения данных из физического файла, берут их из файла, имя которого приведено в ссылке.

Этот путь может указывать на что угодно — это может быть каталог, он может даже находиться в другой физической файловой системе, более того, указанного файла может и вовсе не быть.

В описании пути можно использовать любое сочетание символических и жестких ссылок.

Операция открытия символической ссылки на чтение или запись приводит к открытию на чтение или запись того файла, на который она ссылается, а не ее самой.

Символическая ссылка в отличие от жесткой имеет свой собственный номер индексного дескриптора и имеет свой тип.

Создание и/или удаление символической ссылки никогда не затрагивают ни имени файла, на который она ссылается, ни его индексного дескриптора.

Файл может быть удален, а ссылка остается («висящая»).

Файл может не существовать в момент создания символической ссылки.

Символическая ссылка создается вызовом **symlink()**.

Интерфейсы **link()** и **symlink()** похожи.

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

Создает новое имя для файла **oldpath**.

Если **newpath** существует, он перезаписан не будет.

Новое имя может использоваться точно так же, как и старое, для любых операций — оба имени ссылаются на один и тот же файл (то есть имеют те же права доступа и владельца) и невозможно сказать, какое имя было «настоящим».

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

Создает символьную ссылку, которая называется **frompath** и содержит строку **topath**.

Символьные ссылки интерпретируются «на лету», как будто бы содержимое ссылки было подставлено вместо пути, по которому идет поиск файла или каталога.

Если **frompath** существует, он перезаписан не будет.

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

Удаляет имя и возможно файл, на который оно ссылается.

Права доступа к файлам

Файловая система контролирует права доступа к файлам, выполняет операции создания и удаления файлов, а также выполняет запись/чтение данных файла.

Каждый файл в ОС UNIX содержит набор прав доступа, по которому определяется, как пользователь взаимодействует с данным файлом. Этот набор хранится в индексном дескрипторе данного файла в виде 16-разрядного целого значения, из которого обычно используется 12 битов. Каждый бит используется как переключатель, разрешая (значение 1) или запрещая (значение 0) тот или иной доступ.

Три первых бита устанавливают различные виды поведения при выполнении.

Оставшиеся девять делятся на три группы по три, определяя права доступа для владельца, группы и остальных пользователей. Каждая группа задает права на чтение, запись и выполнение.

Базовые биты прав доступа

Восьмеричное значение	Вид в столбце прав доступа	Право или назначение бита
4000	**---s------**	Установленный эффективный идентификатор владельца (бит SUID)
2000	**------s---**	Установленный эффективный идентификатор группы (бит SGID)
1000	**---------t** **---------T**	*Клейкий* (sticky) бит. Вид для каталогов и выполняемых файлов, соответственно.
0400	**-r--------**	Право владельца на чтение
0200	**--w-------**	Право владельца на запись
0100	**---x------**	Право владельца на выполнение
0040	**----r-----**	Право группы на чтение
0020	**-----w----**	Право группы на запись
0010	**------x---**	Право группы на выполнение
0004	**-------r--**	Право всех прочих на чтение
0002	**--------w-**	Право всех прочих на запись
0001	**---------x**	Право всех прочих на выполнение

```
$ ls -l /etc
итого 2608
drwxr-xr-x. 3 root root   4096 дек 17 16:35 abrt
-rw-r--r--. 1 root root     18 фев 21  2020 adjtime
-rw-r--r--. 1 root root   1529 окт  9  2019 aliases

lrwxrwxrwx. 1 root root   30 сен 27  2019 extlinux.conf -> ../boot/extlinux/extlinux.conf
lrwxrwxrwx.  1 root root  21 ноя  4 22:13 os-release -> ../usr/lib/os-release
```

Бит чтения для всех типов файлов имеет одно и то же значение: он позволяет читать содержимое файла (получать листинг каталога командой **`ls`**).

Бит записи также имеет одно и то же значение — он позволяет писать в этот файл, включая и перезапись содержимого. Если у пользователя отсутствует право доступа на запись в каталоге, где находится данный файл, то пользователь не сможет его удалить.

Аналогично, без этого же права пользователь не создаст новый файл в каталоге, хотя может сократить длину доступного на запись файла до нуля.

Бит выполнения. Если для некоторого файла установлен бит выполнения, то файл может выполняться как команда. В случае установки этого бита для каталога, этот каталог можно сделать текущим (перейти в него командой **cd**).

Бит SUID. Установленный бит SUID означает, что доступный пользователю на выполнение файл будет выполняться с правами (с эффективным идентификатором) владельца, а не пользователя, вызвавшего файл (как это обычно происходит).

Бит SGID. Установленный бит SGID означает, что доступный пользователю на выполнение файл будет выполняться с правами (с эффективным идентификатором) группы-владельца, а не пользователя, вызвавшего файл (как это обычно происходит).

Если бит SGID установлен для файла, не доступного для выполнения, он означает обязательное блокирование, т.е. неизменность прав доступа на чтение и запись пока файл открыт определенной программой.

Клейкий/липкий бит. Установленный клейкий бит для обычных файлов ранее (во времена PDP-11) означал необходимость сохранить образ программы в памяти после выполнения (для ускорения повторной загрузки). Сейчас при установке обычным пользователем он сбрасывается.

Значение этого бита при установке пользователем **`root`** зависит от версии ОС и иногда необходимо. Так, в ОС Solaris необходимо устанавливать клейкий бит для обычных файлов, используемых в качестве области подкачки.

Установка клейкого бита для каталога означает, что файл в этом каталоге может быть удален или переименован только пользователем-владельцем файла, пользователем-владельцем каталога, если файл доступен пользователю на запись и пользователем root.

Расчет прав. Для расчета прав доступа необходимо сложить восьмеричные значения всех необходимых установленных битов. В результате получится четырехзначное восьмеричное число. Например:

Чтение для владельца:	0400
Запись для владельца:	0200
Выполнение для владельца:	0100
Чтение для группы:	0040
Выполнение для группы:	0010
Выполнение для прочих:	0001
Сумма:	**0751**

Структура файловой системы

Каждый жесткий диск состоит из одной или нескольких логических частей (групп цилиндров), называемых разделами (partitions). Расположение и размер раздела определяется при форматировании диска.

В ОС UNIX разделы выступают в качестве независимых устройств, доступ к которым осуществляется как к различным носителям данных. Обычно в разделе может располагаться только одна физическая файловая система.

Имеется много типов физических файловых систем, например FAT16, VFAT, NTFS, HPFS, HFS с разной структурой.

Имеется множество типов физических файловых систем UNIX (ufs, s5fs, ext234, vxfs, jfs, ffs и т.д.).

Физическая файловая система UNIX

Физическая файловая система UNIX занимает раздел диска и состоит из следующих основных компонентов:

Суперблок (superblock).
Содержит общую информацию о файловой системе.

Массив индексных дескрипторов (ilist).
Содержит метаданные всех файлов файловой системы.
Индексный дескриптор (***inode***) содержит информацию о статусе файла и указывает на расположение данных этого файла. Ядро обращается к индексному дескриптору по индексу в массиве.

Один дескриптор является корневым для физической файловой системы, через него обеспечивается доступ к структуре каталогов и файлов после монтирования файловой системы.
Размер массива индексных дескрипторов является фиксированным и задается при создании физической файловой системы.

Блоки хранения данных.
Данные обычных файлов и каталогов хранятся в блоках. Обработка файла осуществляется через индексный дескриптор, содержащий ссылки на блоки данных.

Суперблок

Суперблок содержит информацию, необходимую для монтирования и управления файловой системой в целом. В каждой файловой системе существует только один суперблок, который располагается в начале раздела. Суперблок считывается в память ядра при монтировании файловой системы и находиться там до ее отключения - демонтирования.

Суперблок содержит:

– тип файловой системы;

– размер файловой системы в логических блоках, включая сам суперблок, массив индексных дескрипторов и блоки хранения данных;

– размер массива индексных дескрипторов;

– количество свободных блоков;

– количество свободных индексных дескрипторов;

– флаги;

– размер логического блока файловой системы (512, 1024, 2048, 4096, 8192).

– список номеров свободных индексных дескрипторов;

– список адресов свободных блоков.

Количество свободных индексных дескрипторов и блоков хранения данных может быть значительным, поэтому хранение списка номеров свободных индексных дескрипторов и списка адресов свободных блоков целиком в суперблоке непрактично.

Для индексных дескрипторов храниться только часть списка. Когда число свободных дескрипторов приближается к 0, ядро просматривает список и вновь формирует список свободных дескрипторов.

Такой подход неприемлем в отношении свободных блоков хранения данных, поскольку по содержимому блока нельзя определить, свободен он или нет. Поэтому необходимо хранить список адресов свободных блоков целиком.

(§) Список адресов свободных блоков может занимать несколько блоков хранения данных, но суперблок содержит только один блок этого списка.

Первый элемент этого блока указывает на блок, хранящий продолжение списка.

Выделение свободных блоков для размещения файла производиться с конца списка суперблока. Когда в списке остается единственный элемент, ядро интерпретирует его как указатель на блок, содержащий продолжение списка. В этом случае содержимое этого блока считывается в суперблок, и блок становится свободным. Такой подход позволяет использовать дисковое пространство под списки, пропорциональное свободному месту в файловой системе. Когда свободного места практически не остается, список адресов свободных блоков целиком помещается в суперблоке.

Индексные дескрипторы

Индексный дескриптор, или **inode**, содержит информацию о файле, необходимую для обработки данных, т.е. метаданные файла. Каждый файл ассоциирован с одним индексным дескриптором, хотя может иметь несколько имен (жестких ссылок) в файловой системе, каждое из которых указывает на один и тот же индексный дескриптор.

Индексный дескриптор не содержит:

- имени файла, которое содержится в блоках хранения данных каталога;
- содержимого файла, которое размещено в блоках хранения данных.

Индексный дескриптор содержит:

- номер;
- тип файла;
- права доступа к файлу;
- количество связей (ссылок на файл в каталогах) файла;
- идентификатор пользователя и группы-владельца;
- размер файла в байтах;
- время последнего доступа к файлу;
- время последнего изменения файла;
- время последнего изменения индексного дескриптора файла;
- указатели на блоки данных файла (обычно 10);
- указатели на косвенные блоки (обычно 3).

Размер индексного дескриптора обычно составляет 128 байтов.

Индексный дескриптор содержит информацию о расположении данных файла. Поскольку дисковые блоки хранения данных, в общем случае, располагаются не последовательно, индексный дескриптор должен хранить физические адреса всех блоков, принадлежащих данному файлу.

Каждый дескриптор содержит 13 указателей.

Первые 10 указателей непосредственно ссылаются на блоки данных файла.

Если файл большего размера — 11-ый указатель ссылается на первый косвенный блок (indirection block) из 128 (256) ссылок на блоки данных.

Если и этого недостаточно, 12-ый указатель ссылается на дважды косвенный блок, содержащий 128 (256) ссылок на косвенные блоки.

Наконец последний, 13-ый указатель ссылается на трижды косвенный блок из 128 (256) ссылок на дважды косвенные блоки. Количество элементов в косвенном блоке зависит от его размера.

Поддерживая множественные уровни косвенности, индексные дескрипторы позволяют отслеживать огромные файлы, не растрачивая дисковое пространство для небольших файлов.

Файлы устройств

ОС UNIX обобщает концепцию файла как универсальную абстракцию.

Практически любое внешнее устройство может быть представлено на пользовательском уровне, как файл специального типа. Это справедливо для жестких дисков, всевозможных параллельных и последовательных портов и виртуальных терминалов.

Например, для создания образов CD/DVD, резервных копий дисков и дисковых разделов в ОС Windows необходимо использовать специальное ПО.

В ОС UNIX этот вопрос решается крайне просто.

```
$ cat /dev/cdrom > image.iso
$ cat myfile.ps > /dev/lp0
$ dd if=/dev/sdb of=sdb.backup
$ dd if=sdb.backup of=/dev/sdb
```

Работа с такими устройствами осуществляется с помощью тех же системных вызовов, которые используются для работы с файлами.

Два типа устройств

Устройства, которые могут быть представлены в виде файлов, делятся на два типа — **символьные** (или потоковые) и **блочные**. Иногда используются термины «байт-ориентированные» и «блок-ориентированные» устройства.

Основными операциями над байт-ориентированными устройствами являются запись и чтение одного (очередного) символа (байта).

Блок-ориентированные устройства воспринимаются как хранилища данных, разделенных на блоки фиксированного размера. Основными операциями, соответственно, являются чтение и запись заданного блока.

Байт-ориентированные — терминал (`/dev/tty0`, `/dev/tty1`, `...`), клавиатура, мышь, принтер (`/dev/lp0`, `...`), звуковая карта, `/dev/null`, `/dev/zero`, `/dev/random`, ...

Блок-ориентированные — диски (/dev/sda,/dev/sdb,...), разделы дисков (/dev/sda1, /dev/sda2,...).

Операции над устройствами

Файлы устройств могут быть открыты на чтение и запись. В результате, используя полученный дескриптор, можно писать на устройство и читать с него.

Блок-ориентированные устройства поддерживают позиционирование с помощью вызова lseek().

Позиционироваться можно в любую точку, которая необязательно должна находиться точно на границе блока или сектора.

Прочитать и записать можно произвольное количество данных. ОС позаботится, чтобы прочитать соответсвующий блок, если начало и/или конец порции не находятся на границах, модифицировать их содержимое и записать обратно.

Байт-ориентированные устройства операцию позиционирования не поддерживают.

Это основное отличие устройств одного типа от другого.

Над устройствами определены дополнительные операции, специфические для конкретного устройства. Например, открытие/закрытие лотка привода, установку скорости обмена и канального протокола последовательного порта, низкоуровневую разметку гибких дисков, управление громкостью воспроизведения звука, и т.п.

Все операции данной категории выполняются с помощью системного вызова управления устройствами -- ioctl()

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

Функция **ioctl** манипулирует базовыми параметрами устройств, представленных в виде специальных файлов.

В частности, многими оперативными характеристиками специальных символьных файлов (например терминалов) можно управлять через **ioctl** запросы. В качестве аргумента **d** должен быть указан открытый файловый дескриптор.

Второй аргумент является кодом запроса, который зависит от устройства.

Третий аргумент является указателем на память, который не имеет типа.

Традиционно это **char *argp** или **void *argp**.

Ioctl запрос **request** кодирует в себе либо аргумент, который является параметром **in** либо аргумент, который является параметром **out** и кроме того размер аргумента **argp** в байтах.

Макросы и определения, используемые в специальных **ioctl** запросах **request** находятся в файле **<sys/ioctl.h>**.

Примеры кода

```
#include <linux/cdrom.h>
int fd = open("/dev/cdrom", O_RDONLY | O_NOBLOCK);

ioctl(fd, CDROMEJECT);       // откроем лоток
ioctl(fd, CDROMCLOSETRAY);   // и закроем его
```

O_NONBLOCK используется, чтобы избежать поиска диска в устройстве и ошибки, если диска в лотке нет.

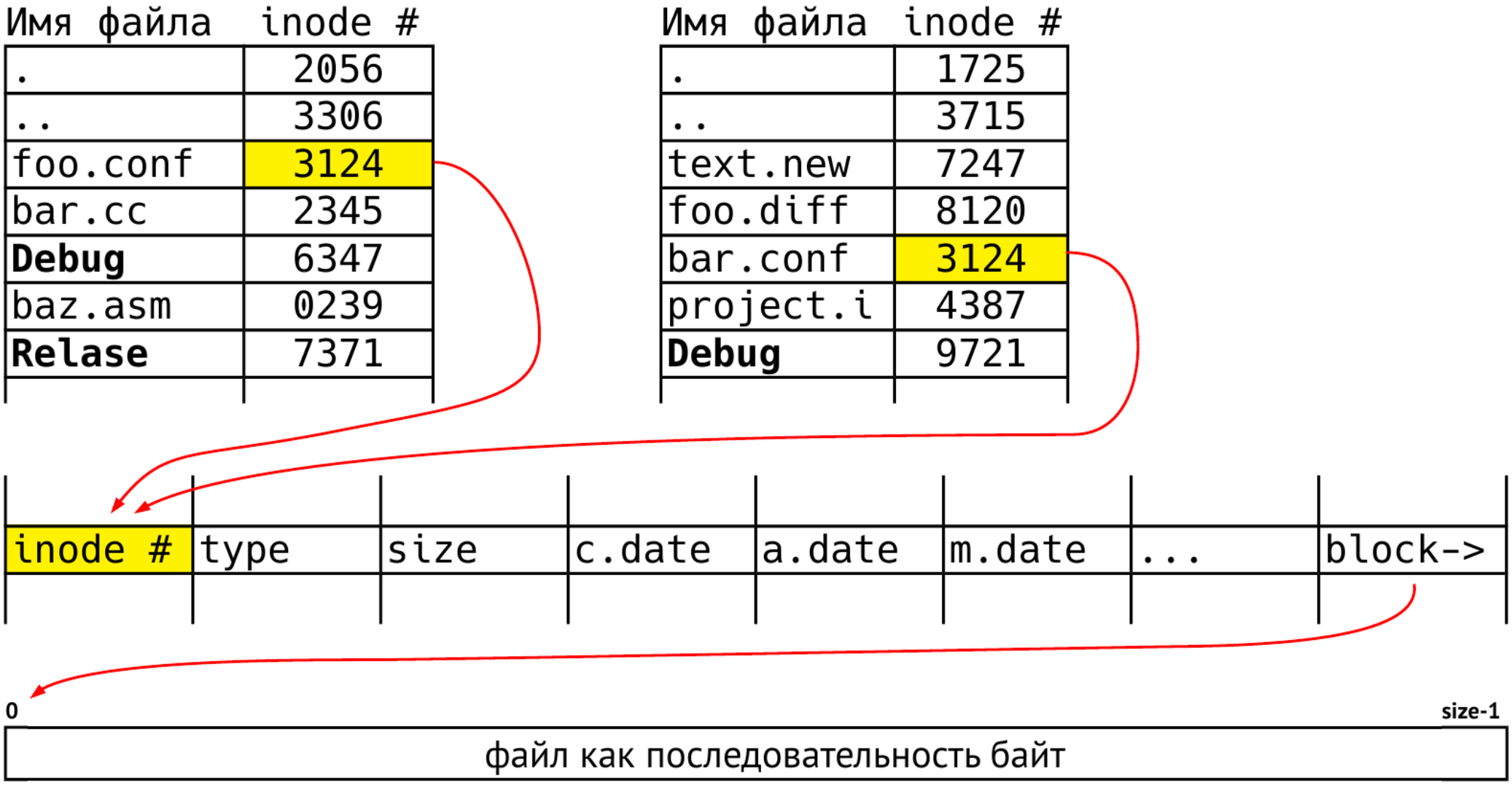
Чтобы заставить проиграть вторую дорожку музыкального диска, следует написать:

```
struct cdrom_ti  cti;
cti.cdti_trk0 = 2;
cti.cdti_ind0 = 0;
cti.cdti_trk1 = 2;
cti.cdti_ind1 = 0;
ioctl(fd, CDROMPLAYTRAKIND, &cti);
```

Системные вызовы для работы с каталогами

Как все устроено — от записи в каталоге к содержимому файла

Как путь преобразуется в файл — **path_resolution(7)**



opendir(3) — открывает/закрывает каталог;
closedir(3) — открывает/закрывает каталог;
dirfd(3) — возвращает файловый дескриптор потока каталога;
readdir(3) — читает каталог в **struct dirent**;
rewinddir(3) — меняет позицию потока каталога;
scandir(3) — поиск совпадающих элементов в каталоге;
seekdir(3) — устанавливает позицию следующего вызова **readdir()**;
telldir(3) — возвращает текущее положение в каталоге.
chdir(2) — изменить рабочий каталог

Открытие каталога

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);

[dirent.h]
...
/* This is the data type of directory stream objects.
   The actual structure is opaque to users.
   Это тип данных объекта потока каталога.
   Действительная структура скрыта от программиста */
typedef struct __dirstream DIR;
```

Функция **opendir()** открывает поток каталога, соответствующий каталогу **name**, и возвращает указатель на этот поток. Поток устанавливается на первой записи в каталоге.

Функция **opendir()** возвращает указатель на поток каталога или NULL в случае ошибок.

ОШИБКИ

EACCES — доступ запрещен;

EMFILE — процесс использует слишком много открытых потоков;

ENFILE — система использует слишком много открытых потоков;

ENOENT — каталога не существует или **name** - пустая строка;

ENOMEM — недостаточно памяти для выполнения операции;

ENOTDIR — **name** не является каталогом.

Соответствующий файловый дескриптор потока каталога может быть получен с помощью **dirfd(3)**.

```
DIR *dir;

dir = opendir("/"); // Открываем корневой каталог
if (!dir) {
    perror("opendir(""/"")");
    exit(errno);
}
```

Закрытие каталога

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dirp);
```

Функция **closedir()** закрывает поток каталога, связанный с **dirp**. При успешном вызове **closedir()** также закрывается файловый дескриптор, связанный с **dirp**. После данного вызова дескриптор потока каталога **dirp** будет недоступен.

Функция **closedir()** возвращает 0 при успешном выполнении. В случае ошибки возвращается **-1**, и устанавливается соответствующее значение **errno**.

ОШИБКИ

EBADF — неверный дескриптор потока каталога **dirp**.

Чтение каталога

```
#include <sys/types.h>
#include <dirent.h>

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

Функция **readdir()** возвращает указатель на структуру **dirent**, представляющую запись каталога в потоке каталога, указанного в **dir**.

Структура **dirent** операционной системой может выделяться статически, поэтому не стоит пытаться освобождать ее вызовом **free()**.

Функция **readdir()** возвращает **NULL** и не меняет **errno** по достижении последней записи. Если произошла ошибка возвращается **NULL**, а **errno** устанавливается в соответствующее значение. Чтобы отличить конец потока от ошибки, перед вызовом **readdir()** следует присваивать **errno** значение нуля и проверять значение **errno**, если возвращается **NULL**.

В соответствие с POSIX, структура **dirent** содержит поле **char d_name[]** неопределенной длины, с максимальным количеством символов, предшествующих конечному нулевому символу, равным **NAME_MAX**.

Использование других полей отрицательно влияет на переносимость программы.

ОШИБКИ

EBADF — неверный описатель потока каталога **dir**.

Структура dirent

В реализации **glibc** структура **dirent** определена следующим образом:

```
struct dirent {
    ino_t          d_ino;       // номер иноды файла
    off_t          d_off;       // это не смещение!!!
    unsigned short d_reclen;    // длина этой записи
    unsigned char  d_type;      // тип файла; поддерживается не во всех ФС
    char           d_name[256]; // имя файла с null в конце
}
```

d_off — значение, возвращаемое в **d_off**, тоже самое что и после вызова **telldir(3)** в текущем положении курсора в потоке каталога. Не смотря на тип и имя, в современных файловых системах поле **d_off** мало похоже на смещение в каталоге. Приложения должны считать, что это поле неизвестного типа (чёрный ящик) и не делать предположений о его содержимом.

d_reclen — размер (в байтах) возвращаемой записи. Может не совпадать с размером структуры **dirent**.

d_name — это поле содержит имя файла с завершающим **null**.

d_type — это поле содержит значение типа файла, что позволяет не делать дополнительный вызов **lstat(2)**, если дальнейшие действия зависят от типа файла.

Если определён соответсвующий макрос тестирования свойств **_DEFAULT_SOURCE**, то для значения, возвращаемого в **d_type**, glibc определяет следующие макросы-константы:

DT_BLK — блочное устройство;
DT_CHR — символьное устройство;
DT_DIR — каталог;
DT_FIFO — именованный канал (FIFO);
DT_LNK — символическая ссылка;
DT_REG — обычный файл;
DT_SOCK — доменный сокет UNIX;
DT_UNKNOWN — тип файла невозможно определить.

В настоящее время, только некоторые файловые системы (Btrfs, ext2, ext3 и ext4) поддерживают возврат типа файла в **d_type**. Все приложения должны правильно обрабатывать возвращаемое значение **DT_UNKNOWN**.

Данные, возвращаемые **readdir()**, могут быть переписаны последующими вызовами **readdir()** для того же потока каталога.

ЗАМЕЧАНИЯ

Поток каталога открывается с помощью **opendir(3)**.

Порядок последовательно читаемых имён файлов вызовом **readdir()** зависит от реализации файловой системы и не следует ожидать, что имена будут отсортированы предсказуемым образом.

В стандарте POSIX.1 определены только поля **d_name** и **d_ino**.

Кроме Linux, поле **d_type** доступно, преимущественно только в системах BSD. Остальные поля доступны во многих, но не во всех системах.

В glibc программы могут определить доступность полей, не определённых в POSIX.1, по наличию макросов:

_DIRENT_HAVE_D_NAMLEN
_DIRENT_HAVE_D_RECLEN
_DIRENT_HAVE_D_OFF
_DIRENT_HAVE_D_TYPE

Определение структуры **dirent**, показанное выше, взято из заголовочных файлов **glibc** и имеет поле **d_name** постоянного размера.

Предупреждение: приложения не должны зависеть от размера поля **d_name**.

В POSIX оно определено как **char d_name[]** — массив символов неопределённого размера не более **NAME_MAX** символов с конечным байтом **null** (**'\0'**).

В POSIX.1 явно сказано, что это поле не должно использоваться как **lvalue**.

В стандарте также отмечено, что использование **sizeof(d_name)** некорректно — вместо этого следует использовать **strlen(d_name)**.

В некоторых системах это поле определено как **char d_name[1]**[1]) — использовать **sizeof(struct dirent)** для получения размера записи, включающей размер **d_name** также неправильно.

Следует также отметить, что вызов **fpathconf(fd,_PC_NAME_MAX)** (получить параметры настроек для файлов) возвращает значение 255 в большинстве файловых систем, но в некоторых файловых системах, например, CIFS и серверы Windows SMB длина имени файла с **null** конце, возвращаемое в **d_name**, может превышать этот размер. В таких случаях поле **d_reclen** будет содержать значение, превышающее размер структуры **dirent**, взятой из glibc .

---

1) Пластичный массив. Как особый случай, последний элемент структуры, содержащей более чем один именованный элемент может иметь незавершенный тип массива — этот элемент называется членом пластичного массива. Размер структуры остается такой, как если бы член пластичного массива отсутствовал, за исключением того, что такой член может иметь большее завершающее заполнение, чем подразумевает его отсутствие.

Пример использования:

```
#include <stdio.h>
#include <sys/types.h>
#include dirent.h>
#include stdlib.h>
#include errno.h>

int main (int argc, char *argv[]) {

    DIR *dir = opendir(".");
    if (!dir) {
        perror("opendir(""."")");
        exit(errno);
    }
    struct dirent *dir_entry;
    for (;;) {
        dir_entry = readdir(dir);
        if (NULL == dir_entry) { // конец каталога или ошибка
            if (errno) {
                perror("readdir()");
                exit(errno);
            }
           break;
        }
        process_direntry(dir_entry); // обработка очередного элемента
    }
    return(0);
}
```

Системные вызовы для работы с файлами

Получить информацию о файле

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

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

#include <fcntl.h>    /* определения констант AT_* */
#include <sys/stat.h>

int fstatat(int dirfd, const char *pathname, struct stat *statbuf, int flags);
```

Данные системные вызовы возвращают информацию о файле в буфер, на который указывает **statbuf**.

Для этого не требуется иметь права доступа к самому файлу, но в случае **stat()**, **fstatat()** и **lstat()** – потребуются права выполнения (поиска) на все каталоги, указанные в полном имени файла **pathname**.

Вызовы **stat()** и **fstatat()** возвращают информацию о файле, указанном в **pathname**.

Вызов **lstat()** идентичен **stat()**, но в случае, если **pathname** является символьной ссылкой, то возвращается информация о самой ссылке, а не о файле, на который она указывает.

Вызов **fstat()** идентичен **stat()**, но файл задаётся файловым дескриптором **fd**.

Структура `stat`

Все эти системные вызовы возвращают структуру **`stat`**, которая содержит следующие поля:

```
struct stat {
    dev_t     st_dev;         // ID устройства с файлом
    ino_t     st_ino;         // номер иноды
    mode_t    st_mode;        // тип файла и режим доступа
    nlink_t   st_nlink;       // количество жёстких ссылок
    uid_t     st_uid;         // идентификатор пользователя-владельца
    gid_t     st_gid;         // идентификатор группы-владельца
    dev_t     st_rdev;        // идентификатор устройства
    off_t     st_size;        // общий размер в байтах
    blksize_t st_blksize;     // размер блока ввода-вывода ФС
    blkcnt_t  st_blocks;      // количество выделенных 512Б блоков

    struct timespec st_atim; // время последнего доступа
    struct timespec st_mtim; // время последнего изменения
    struct timespec st_ctim; // время последней смены состояния
};
```

Замечание: порядок полей структуры **`stat`** для разных архитектур отличается.

Также, в определении выше не показаны дополняющие байты, которые для различных архитектур могут присутствовать между некоторыми полями.

Поля структуры `stat`

`st_dev` — устройство, на котором расположен файл (для разбора идентификатора этого поля могут пригодиться макросы major(3) и minor(3)).

`st_ino` — номер иноды файла.

`st_mode` — тип файла и режим доступа ().

`st_nlink` — количество жёстких ссылок на файл.

`st_uid` — пользовательский идентификатор владельца файла.

`st_gid` — групповой идентификатор владельца файла.

`st_rdev` — устройство, который этот файл (инод) представляет.

`st_size` — размер файла (если он обычный или является символьной ссылкой) в байтах. Размер символьной ссылки равен длине пути файла, на который она ссылается, без конечного нулевого байта.

`st_blksize` — «предпочтительный» размер блока для эффективного ввода/вывода в файловой системе.

`st_blocks` — количество блоков (по 512 байт), выделенных для файла (может быть меньше, чем **`st_size/512`**, когда в файле есть пропуски (holes)).

`st_atime` — метка времени последнего доступа к файлу.

`st_mtime` — метка времени последнего изменения файла.

`st_ctime` — метка времени последнего изменения состояния файла.

Системный вызов `fstatat()`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>    /* определения констант AT_* */
#include <sys/stat.h>

int fstatat(int dirfd,
            const char *pathname,
            struct stat *statbuf,
            int flags);
```

Системный вызов **`fstatat()`** представляет собой обобщённый интерфейс доступа к файловой информации, и может выполнить работу за системные вызовы **`stat()`**, **`lstat()`** и **`fstat()`**.

Если в **`pathname`** задан относительный путь, то он считается относительно каталога, на который ссылается файловый дескриптор **`dirfd`** (а не относительно текущего рабочего каталога вызывающего процесса, как это имеет место в **`stat()`** и **`lstat()`**).

Если в **`pathname`** задан относительный путь и значение **`dirfd`** равно **`AT_FDCWD`**, то **`pathname`** рассматривается относительно текущего рабочего каталога вызывающего процесса, как в случае **`stat()`** и **`lstat()`**.

Если в **`pathname`** задан абсолютный путь, то **`dirfd`** игнорируется.

Значение **`flags`** может быть 0, или включать один или более флагов (man stat(3)).

При успешном выполнении возвращается 0.

В случае ошибки возвращается -1, а **`errno`** устанавливается в соответствующее значение.

ОШИБКИ

EACCES — запрещён поиск в одном из каталогов пути **pathname**.

EBADF — значение **fd** не является правильным открытым файловым дескриптором.

EFAULT — неправильный адрес.

ELOOP — во время определения пути встретилось слишком много симв. ссылок.

ENAMETOOLONG — слишком длинное значение аргумента **pathname**.

ENOENT — компонент пути **pathname** не существует или в и указана пустая строка, а в **flags** не указан **AT_EMPTY_PATH**.

ENOMEM — не хватает памяти (например, памяти ядра).

ENOTDIR — компонент в префиксе пути **pathname** не является каталогом.

EOVERFLOW — значение **pathname** или **fd** ссылаются на файл, чей размер, номер **inode** или количество блоков не может быть представлено с помощью типов **off_t**, **ino_t** или **blkcnt_t**, соответственно.

Эта ошибка может возникнуть, если, например, приложение собрано на 32-битной платформе без флага **-D_FILE_OFFSET_BITS=64** , а размер файла, для которого вызван **stat()**, превышает (1<<31)-1 байт.

В **fstatat()** дополнительно могут возникнуть следующие ошибки:

EBADF — значение **dirfd** не является правильным файловым дескриптором.

EINVAL — указано неверное значение в **flags**.

ENOTDIR — значение **pathname** содержит относительный путь и **dirfd** содержит файловый дескриптор, указывающий на файл, а не на каталог.

Указанные в **stat.h** макросы POSIX проверяют, является ли файл:

S_ISLNK(st_mode) — символьной ссылкой;
S_ISREG(st_mode) — обычным файлом;
S_ISDIR(st_mode) — каталогом;
S_ISCHR(st_mode) — символьным устройством;
S_ISBLK(st_mode) — блочным устройством;
S_ISFIFO(st_mode) — каналом FIFO;
S_ISSOCK(st_mode) — сокетом.

Битовые маски для полей типа файла

S_IFMT **0170000** — битовая маска для полей типа файла

S_IFSOCK **0140000** — сокет
S_IFLNK **0120000** — символьная ссылка
S_IFREG **0100000** — обычный файл
S_IFBLK **0060000** — блочное устройство
S_IFDIR **0040000** — каталог
S_IFCHR **0020000** — символьное устройство
S_IFIFO **0010000** — канал FIFO
S_ISUID **0004000** — бит setuid
S_ISGID **0002000** — бит setgid
S_ISVTX **0001000** — бит принадлежности

Битовые маски для полей прав доступа

S_IRWXU 00700 — маска для прав доступа пользователя
S_IRUSR 00400 — пользователь имеет право чтения
S_IWUSR 00200 — пользователь имеет право записи
S_IXUSR 00100 — пользователь имеет право выполнения
S_IRWXG 00070 — маска для прав доступа группы
S_IRGRP 00040 — группа имеет права чтения
S_IWGRP 00020 — группа имеет права записи
S_IXGRP 00010 — группа имеет права выполнения
S_IRWXO 00007 — маска прав доступа всех прочих
S_IROTH 00004 — все прочие имеют права чтения
S_IWOTH 00002 — все прочие имеют права записи
S_IXOTH 00001 — все прочие имеют права выполнения

Пример

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysmacros.h>

int main(int argc, char *argv[]) {

    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Использование: %s <путь>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (lstat(argv[1], &sb) == -1) {
        perror("lstat");
        exit(EXIT_FAILURE);
    }

    printf("ID содержащего устройства: [%lx,%lx]\n",
           (long)major(sb.st_dev), (long)minor(sb.st_dev));
```

```
    printf("Тип файла:");

    switch (sb.st_mode & S_IFMT) {
    case S_IFBLK:  printf("блочное устройство\n");     break;
    case S_IFCHR:  printf("символьное устройство\n");  break;
    case S_IFDIR:  printf("каталог\n");                break;
    case S_IFIFO:  printf("FIFO/канал\n");             break;
    case S_IFLNK:  printf("символьная ссылка\n");      break;
    case S_IFREG:  printf("обычный файл\n");           break;
    case S_IFSOCK: printf("сокет\n");                 break;
    default:       printf("неизвестно?\n");            break;
    }
    printf("Кол-во ссылок: %ld\n", (long) sb.st_nlink);
    printf("Размер файла: %lld байт\n", (long long) sb.st_size);
    printf("Посл. изм. состояния: %s", ctime(&sb.st_ctime));
    printf("Посл. доступ к файлу: %s", ctime(&sb.st_atime));
    printf("Посл. изм. файла:     %s", ctime(&sb.st_mtime));
    exit(EXIT_SUCCESS);
}
```

Вывод

```
ID содержащего устройства: [8,3]
Тип файла:каталог
Кол-во ссылок: 19
Размер файла: 4096 байт
Посл. изм. состояния: Mon Nov  8 14:13:50 2021
Посл. доступ к файлу: Sun Feb 27 03:33:06 2022
Посл. изм. файла:     Mon Nov  8 14:13:50 2021
```

Системные вызовы ввода/вывода

Файл (File) [POSIX.1-2017]

Объект, в который можно писать или читать, или и то и другое.
Файл имеет определенные атрибуты, включая права доступа и тип.
Типы файлов включают:
- обычный файл;
- специальный символьный файл;
- специальный файл блока;
- специальный файл FIFO;
- символическую ссылку;
- сокет;
- каталог.

Реализация может поддерживать другие типы файлов.

Специальный файл FIFO — тип файла со свойством чтения данных, записанных в такой файл, в порядке очереди записи.

Открытие файла

Чтобы начать работу с файлом, его необходимо открыть. Операция объявляет системе, что программа собирается работать с данным файлом.

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

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

Вызов **open( )** используется, чтобы преобразовать путь к файлу в дескриптор файла.

Дескриптор файла (File Descriptor/Описатель файла) – уникальное неотрицательное целое число для каждого процесса, используемое для идентификации открытого файла с целью доступа к нему с помощью вызовов **read( )**, **write( )** и т.п. при последующем вводе-выводе.

Значение вновь созданного файлового дескриптора составляет от нуля до **OPEN_MAX**[2] **– 1**. Дескриптор файла может иметь значение больше или равное **OPEN_MAX**, если значение **OPEN_MAX** уменьшилось (**sysconf( )**) с момента открытия дескриптора файла. Дескрипторы файлов также могут использоваться для реализации дескрипторов каталога сообщений и потоков каталогов.

Дескриптор файла — это не индексный дескриптор, а идентификатор объекта данных ядра ОС.

Дескрипторы с номерами 0, 1, 2 обычно связаны со стандартными потоками ввода-вывода (**stdin**, **stdout**, **stderr**). На момент передачи управления пользовательскому коду программы обычно эти дескрипторы уже открыты. Тем не менее, их можно перенаправить.

2) Определено в <limits.h>

Если системный вызов завершается успешно, возвращенный файловый дескриптор является наименьшим дескриптором, который еще не открыт процессом. В результате этого вызова появляется новый открытый файл, не используемый совместно никакими процессами (совместно используемые открытые файлы могут возникнуть, когда вызывается **fork(2)**).

Указатель устанавливается в начале файла.

Параметр **flags** — это битовые флаги, определяющие режим открытия файла.

Флаги собираются с помощью побитовой операции OR.

O_RDONLY — открывается в режиме «только для чтения»

O_WRONLY — открывается в режиме «только для записи»

O_RDWR — открывается в режиме «для чтения и записи»

O_CREAT — если файл не существует, то он будет создан.

Владелец (идентификатор пользователя) файла устанавливается в значение эффективного идентификатора пользователя процесса.

Группа (идентификатор группы) устанавливается либо в значение эффективного идентификатора группы процесса, либо в значение идентификатора группы родительского каталога (зависит от типа файловой системы, параметров подсоединения (mount) и режима родительского каталога.

O_EXCL — если флаг используется совместно с **O_CREAT**, то при наличии уже созданного файла вызов **open()** завершится с ошибкой. В этом состоянии, при существующей символьной ссылке не обращается внимание, на что она указывает.

O_EXCL не работает в файловых системах NFS, а в программах, использующих этот флаг для блокировки, возникнет состояние гонки («race condition»).

O_NOCTTY — если **pathname** указывает на терминальное устройство, то оно не станет терминалом управления процесса, даже если процесс такового не имеет;

O_TRUNC — если файл уже существует, он является обычным файлом и режим позволяет записывать в этот файл (т.е. установлено **O_RDWR** или **O_WRONLY**), то его длина будет урезана до нуля.

Если файл является каналом FIFO или терминальным устройством, то этот флаг игнорируется. В остальных случаях действие флага **O_TRUNC** не определено.

O_APPEND — файл открывается в режиме добавления. Перед каждой операцией **write** файловый указатель будет устанавливаться в конце файла, как если бы использовался **lseek()**).

O_APPEND может привести к повреждению файлов в системе NFS, если несколько процессов одновременно добавляют данные в один файл. Это происходит из-за того, что NFS не поддерживает добавление в файл данных, поэтому ядро на машине-клиенте должно эмулировать эту поддержку).

O_NONBLOCK или **O_NDELAY** — если возможно, то файл открывается в режиме **non-blocking**. Ни **open()**, ни другие последующие операции над возвращаемым дескриптором файла не заставляют вызывающий процесс ждать. Используется для работы с каналами FIFO.

O_SYNC — файл открывается в режиме синхронного ввода-вывода. Все вызовы **write()** для соответствующего описателя файла блокируют вызывающий процесс до тех пор, пока данные не будут физически записаны.

O_NOFOLLOW — если **pathname** — это символьная ссылка, то **open()** вернет код ошибки. Это расширение FreeBSD. Все прочие символьные ссылки в имени будут обработаны как обычно.

O_DIRECTORY — если **pathname** не является каталогом, то **open()** выдаст ошибку. Этот флаг используется только в Linux чтобы избежать проблем с «отказом от обслуживания» при вызове **opendir(2)** для канала FIFO или ленточного устройства.

O_LARGEFILE — на 32-битных системах, поддерживающих файловые системы (Large), этот флаг позволяет открывать файлы, длина которых больше, чем 31 бит.

O_CLOEXEC — устанавливает флаг ***close-on-exec*** на новом файловом дескрипторе.

Указание данного флага позволяет программе избежать дополнительной операции **fcntl(2) F_SETFD** для установки флага **FD_CLOEXEC**.

Использование этого флага обязательно для некоторых многонитиевых программ, поскольку использование отдельной операции **fcntl(2) F_SETFD** для установки флага **FD_CLOEXEC** недостаточно для избежания состязательности, когда одна нить открывает файловый дескриптор, а в тоже время другая нить может выполнять **fork(2)** и **execve(2)**. В зависимости от порядка выполнения, состязательность может привести к тому, что файловый дескриптор, возвращённый **open()**, будет ненамеренно передан программе, выполняющейся в созданном с помощью **fork(2)** потомке (такого рода состязательность, в принципе, возможна для любых системных вызовов, создающих файловый дескриптор, у которого должен быть установлен флаг *close-on-exec*, и различные другие системные вызовы Linux предоставляют эквивалент флагу **O_CLOEXEC**, чтобы избежать этой проблемы).

Символьные константы, используемые в mode

S_IRWXU 00700 владелец файла имеет права на чтение, запись и выполнение
S_IRUSR 00400 пользователь имеет права на чтение файла
S_IWUSR 00200 пользователь имеет права на запись в файл
S_IXUSR 00100 пользователь имеет права на выполнение файла

S_IRWXG 00070 группа имеет права на чтение, запись и выполнение файла
S_IRGRP 00040 группа имеет права на чтение файла
S_IWGRP 00020 группа имеет права на запись в файл
S_IXGRP 00010 группа имеет права на выполнение файла

S_IRWXO 00007 все остальные имеют права на чтение, запись и выполнение
S_IROTH 00004 все остальные имеют права на чтение файла
S_IWOTH 00002 все остальные имеют права на запись в файл
S_IXOTH 00001 все остальные имеют права на выполнение файла

Чтение и запись

Операции чтения и записи автоматически обеспечивают доступ к последовательным порциям файла.

```
#include <unistd.h>

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

Пытается записать **count** байт из фала, с которым связан файловый дескриптор **fd** в буфер, адрес которого начинается с **buf**.

Если количество **count** равно нулю, то **read()** возвращает это нулевое значение и завершает свою работу.

Если **count** больше, чем **SSIZE_MAX**, то результат не определен.

При успешном завершении вызова возвращается количество байтов, которые были считаны (нулевое значение означает конец файла), а позиция файла увеличивается на это значение.

Если количество прочитанных байтов меньше, чем количество запрошенных, то это не считается ошибкой – данные, например, могли быть почти в конце файла, в канале, на терминале, или **read()** был прерван сигналом.

В случае ошибки возвращаемое значение равно -1, а переменной **errno** присваивается номер ошибки. В этом случае позиция файла не определена.

Особый случай – вызов возвратил 0. Это значит **EOF**.

```
#include <unistd.h>

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

Записывает до **count** байтов из буфера **buf** в файл, на который ссылается файловый дескриптор **fd**.

В случае успешного завершения возвращается количество байтов, которые были записаны (ноль означает, что не было записано ни одного байта).

В случае ошибки возвращается -1, а переменной **errno** присваивается соответствующее значение.

Если **count** равен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не будет произведено никаких действий.

Закрытие файла

После окончания работы с файлом его следует закрыть. Это важно, поскольку дескрипторы файла — ограниченный ресурс.

Общее количество файловых дескрипторов в системе ограничено.

Количество дескрипторов открытых одним процессом файлов ограничено.

```
#include <unistd.h>

int close(int fd);
```

Закрывает файловый дескриптор, который после этого не ссылается ни на один и файл и может быть использован повторно. Все блокировки, находящиеся на соответствующем файле, снимаются (независимо от того, был ли использован для установки блокировки именно этот файловый дескриптор).

Если **fd** является последней копией какого-либо файлового дескриптора, то ресурсы, связанные с ним, освобождаются; если дескриптор был последней ссылкой на файл, удаленный с помощью **unlink()**, то файл окончательно удаляется.

Позиционирование

Операции чтения и записи автоматически обеспечивают доступ к последовательным порциям файла.

Для изменения порядка доступа с дескриптором связано значение текущей позиции, которое можно изменять с помощью системного вызова

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

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

Устанавливает смещение для файлового дескриптора **fd** в значение аргумента offset в соответствии с параметром **whence** который может принимать одно из следующих значений:

SEEK_SET Смещение устанавливается в **offset** байт от начала файла.

SEEK_CUR Смещение устанавливается как текущее смещение плюс **offset** байт.

SEEK_END Смещение устанавливается как размер файла плюс **offset** байт.

Функция **lseek()** позволяет задавать смещения, которые будут находиться за существующим концом файла (но это не изменяет размер файла). Если позднее по этому смещению будут записаны данные, то последующее чтение в промежутке от конца файла до этого смещения, будет возвращать нулевые байты (до тех пор, пока в этот промежуток не будут фактически записаны данные).

SEEK_DATA Подогнать файловое смещение к следующему расположению, большему или равному значению **offset**, по которому в файле есть данные. Если значение **offset** указывает на данные, то файловое смещение устанавливается в **offset**.

SEEK_HOLE Подогнать файловое смещение к следующему промежутку, большему или равному значению **offset**. Если значение **offset** указывает в середину промежутка, то файловое смещение устанавливается в **offset**. Если перед **offset** нет промежутка, то файловое смещение подгоняется к концу файла (т.е., это скрытый промежуток, который есть в конце любого файла).

Если **offset** указывает за конец файла, в обоих вышеуказанных случаях, **lseek()** завершится с ошибкой.

Эти операции позволяют приложениям отображать промежутки в разреженном файле. Это может быть полезно для таких приложений, как инструменты резервного копирования файлов, которые могут выиграть в месте при создании резервных копий и сохранить промежутки, если у них есть механизм их обнаружения.

Для поддержки этих операций промежуток представляется последовательностью нулей, которые (обычно) физически не занимают места на носителе. Однако файловая система может не сообщать о промежутках, поэтому эти операции — не являются гарантируемым механизмом отображения пространства носителя в файл (более того, последовательность нулей, которая на самом деле была записана на носитель, может не посчитаться промежутком). В простейшей реализации, файловая система может поддержать эти операции так — при **SEEK_HOLE** всегда возвращать смещение конца файла, а при **SEEK_DATA** всегда возвращать значение **offset**.

Чтобы получить определения **SEEK_DATA** и **SEEK_HOLE** из **<unistd.h>**, нужно задать макрос тестирования свойств **_GNU_SOURCE**.

Возвращает

При успешном выполнении **`lseek()`** возвращает получившееся в результате смещение в байтах от начала файла. При ошибке возвращается значение **`(off_t) -1`** и в **`errno`** записывается код ошибки.

Ошибки

EBADF – **`fd`** не является открытым файловым дескриптором.

EINVAL – Неправильное значение **`whence`**. Получается, что возвращаемое файловое смещение стало бы отрицательным или указывало бы за конец поверхности носителя.

ENXIO – Значение **`whence`** равно **`SEEK_DATA`** или **`SEEK_HOLE`**, а файловое смещение указывает за конец файла.

EOVERFLOW – Результирующие файловое смещение не может быть представлено типом **`off_t`**.

ESPIPE – Значение **`fd`** связано с каналом, сокетом или FIFO.