Сергей Юрьевич Шилов

Системное программирование для современных платформ

Системное программирование для современных платформ

4. Низкоуровневый файловый ввод-вывод

Системное программирование для современных платформ

Файлы:

Регулярные (- или r)

- Каталоги (d)
- Символьные ссылки (l)
- Каналы (p)
  - Именованные (n)
  - Неименованные (u)
- Специальные файлы устройств
  - Блочные (b)
  - Символьные (c)
- Сокеты (s)

Системное программирование для современных платформ

Информация и метаинформация файлов

- Информация - «содержимое» файлов
- Метаинформация — атрибуты файлов (имя, размер, права доступа, временные метки)

Системное программирование для современных платформ

Права доступа (UNIX ACL):

файл	A — All (все)		
	U — user (владелец)	G — Group (группа)	O — Other (остальные)
	R W X	R – X	R – –
	R — Read	W — Wright	X — eXecute

Каждой группе соответствует своя битовая маска:

- rwx — 111=07
- rw — 110=06
- rx — 011=03
- r — 010=02 etc.

Системное программирование для современных платформ

Особенности прав доступа к каталогам:

• R — разрешено только узнать список файлов, содержащихся в этом каталоге и те метаданные, которые хранятся в файле каталога

• X — разрешено заходить в каталог и просматривать содержимое файлов, если это разрешено правами доступа к файлам, узнавать атрибуты файлов. Можно изменить содержимое файла, если это разрешено правами доступа к файлу, но не имя файла.

• W — разрешено изменять атрибуты файлов, их имена, удалять их вне зависимости от прав доступа к файлам.

Системное программирование для современных платформ

Особенности UNIX ACL

- Невозможно установить разные права доступа для двух различных групп пользователей (решается через создание нескольких имён у файла — hardlinks)
- Возможность удаления файлов, при отсутствии прав доступа к самому файлу (достаточно прав записи в каталог) (решается через установку специальных прав доступа)

Системное программирование для современных платформ

Специальные права доступа

- Биты смены эффективного идентификатора пользователя и группы (u:S - -, g:- S -)
- Специальный бит доступа к директориям (o:- - T): позволяет пользователю изменять в каталогах только свои файлы. Устанавливается на каталоги «коллективного доступа» (почтовые каталоги, спулеры, каталоги временных файлов)
- Специальный бит файла (стики-бит o:- - S). Запрещает своппинг файла (не является POSIX-совместимым)

Системное программирование для современных платформ

Временные метки (timestamps)

- Время создания, модификации, доступа к файлам
- Считаются в секундах от 00:00:00 1 января 1970 UTC (эра UNIX, UNIX epoch)
- Тип данных long (32-bit) или extra long (64-bit)

16:00 23 сентября 2015 — 1 443 628 800

Системное программирование для современных платформ

Утилиты и API

- ls/stat, access
- chmod
- chown, chgrp/chown
- -/utime
- ln/link, symlink
- rm/unlink
- umask

Системное программирование для современных платформ

ls [опции] [файл...]
dir [файл...]
vdir [файл…]

ОПИСАНИЕ

Программа **ls** выводит список всех файлов (не каталогов), перечисленных в командной строке, а затем выводит список всех файлов, находящихся в каталогах, перечисленных в командной строке. Если не указано ни одного файла, то по умолчанию аргументом назначается `.' (текущий каталог). Опция **-d** заставляет **ls** не считать аргументы-каталоги каталогами. Будут отображаться только файлы, имя которых не начинаются с `.' или все файлы, если задана опция **-a**.
Каждый список файлов (для файлов, которые не являются каталогами и для каждого каталога, содержащего список файлов) сортируется отдельно в алфавитной последовательности текущих региональных настроек (locale). Когда указана опция **-l**, то перед каждым списком выводится итоговая строка с общим размером всех файлов в списке, который измеряется в полу-килобайтах (512 байт).

Системное программирование для современных платформ

ОПЦИИ POSIX

-C – напечатать список файлов в колонках с вертикальной сортировкой
-F – для каждого имени каталога добавлять суффикс `**/**', для каждого имени FIFO – `**|**' и для каждого имени исполняемого файла `***'**
-R – включить рекурсивную выдачу списка каталогов
-a – включать в список файлы с именем, начинающимся с `.'
-c – использовать при сортировке (при задании опции **-t** или **-l**) время изменения состояния файла вместо времени последней модификации файла
-d – выдавать имена каталогов, как будто они обычные файлы, вместо того, чтобы показывать их содержимое
-i – предварять вывод для каждого файла его серийным номером (номером **inode**).
-l – выдавать (в одноколоночном формате) тип файла, права доступа к файлу, количество ссылок на файл, имя владельца, имя группы, размер файла (в байтах), временной штамп и имя файла. Типы файлов могут принимать следующие значения: **-** для обычного файла, **d** для каталога, **b** для блочного устройства, **c** для символьного устройства, **l** для символической ссылки, **p** для **FIFO** и **s** для гнезда (**socket**)

По умолчанию, временная метка является временем последней модификации; опции **-c** и **-u** позволяют выбрать две другие метки. Для файлов устройств, поле размера обычно указывает на старший и младший номера устройства

Системное программирование для современных платформ

ОПЦИИ POSIX (продолжение)

-q - вместо непечатаемых символов в имени файла при выводе будут ставиться знаки вопроса. (Эта опция включена по умолчанию при выводе на терминал)
-r - производить сортировку в обратном порядке
-t - сортировать по показываемой временной метке
-u - использовать при сортировке (опция **-t**) или перечислении (опция **-l**) время последнего доступа к файлу вместо времени последней модификации файла
-1 - вывод в одну колонку
-- - завершает список опций

Системное программирование для современных платформ

chmod [опции] режим файл...

ОПИСАНИЕ

chmod изменяет права доступа каждого указанного файла в соответствии с правами доступа, указанными в параметре **режим**, который может быть представлен как в символьном виде, так и в виде восьмеричного числа, представляющего битовую маску новых прав доступа.
Формат символьного режима таков:
`**[ugoa...][[+-=][rwxXstugo...]...][,...]**'.

Каждый аргумент - это список команд изменения прав доступа, разделенных запятыми. Каждая команда начинается с нуля или более букв `**ugoa**', комбинация которых указывает, чьи права доступа к файлу будут изменены: владельца **(u)**; группы **(g)**; остальных пользователей **(o)** или же всех пользователей **(a)**. Буква `**a**' эквивалентна `**ugo**'. Если не задана ни одна буква, то автоматически будет использоваться буква `**a**', но биты, установленные в **umask**, не будут затронуты.

Оператор `**+**' добавляет выбранные права доступа к уже имеющимся; `**-**' удаляет эти права; а `**=**' присваивает только эти права каждому указанному файлу.

Буквы `**rwxXstugo**' выбирают новые права доступа: чтение **(r)**; запись **(w)**; выполнение (или доступ к каталогу) **(x)**; выполнение, если файл является каталогом или уже имеет право на выполнение для какого-нибудь пользователя **(X)**; **setuid-** или **setgid-**биты **(s)**; **sticky-**бит **(t)**; установка для остальных таких же прав доступа, которые имеет пользователь, владеющий этим файлом **(u)**; установка для остальных таких же прав доступа, которые имеет группа файла **(g)**; установка для остальных таких же прав доступа, которые имеют остальные пользователи (не входящие в группу файла) **(o)**.

Системное программирование для современных платформ

`sticky-бит' не описывается в POSIX. Такое специфическое название он получил из-за первоначальной функции, которую он выполнял: сохранял исполняемый код программы на устройстве подкачки. В настоящее время установка **sticky**-бита для каталога, приводит к тому, что только владелец файла и владелец этого каталога могут удалять этот файл из каталога. (Обычно это используется в каталогах типа **/tmp**, куда все имеют права на запись).

Числовой режим состоит из не более четырех восьмеричных цифр (от нуля до семи), которые складываются из битовых масок **4**, **2** и **1**. Любые пропущенные разряды дополняются лидирующими нулями. Первая цифра выбирает установку идентификатора пользователя **(setuid) (4)** или идентификатора группы **(setgid) (2)** или **sticky-бита (1)**. Вторая цифра выбирает права доступа для пользователя, владеющего данным файлом: чтение**(4)**, запись **(2)** и выполнение **(1)**; третья цифра выбирает права доступа для пользователей, входящих в данную группу, с тем же смыслом, что и у второй цифры; и четвертый разряд выбирает права доступа для остальных пользователей (не входящих в данную группу), опять с тем же смыслом.

chmod никогда не изменяет права на символьные ссылки, так как этого не умеет делать системный вызов **chmod**. Это не является проблемой, так как права символьных ссылок никогда не используются. Однако, для каждой символьной ссылки, заданной в командной строке, **chmod** изменяет права доступа связанного с ней файла. При этом **chmod** игнорирует символьные ссылки, встречающиеся во время рекурсивной обработки каталогов.

Системное программирование для современных платформ

ОПЦИИ POSIX

-R - Рекурсивное изменение прав доступа для каталогов и их содержимого.

ПРИМЕРЫ

```
chmod g-s file снимает бит set-group-ID (sgid),
chmod ug+s file устанавливает биты suid и sgid,
chmod o+s file ничего не делает вне UNIX-систем
```

Системное программирование для современных платформ

chown [опции] пользователь[:группа] файл…
chgrp [опции] группа файл...

ОПИСАНИЕ

chown изменяет владельца и/или группу для каждого заданного файла.
В качестве имени владельца/группы берется первый аргумент, не являющийся опцией. Если задано только имя пользователя (или числовой идентификатор пользователя), то данный пользователь становится владельцем каждого из указанных файлов, а группа этих файлов не изменяется. Если за именем пользователя через двоеточие следует имя группы (или числовой идентификатор группы), без пробелов между ними, то изменяется также и группа файла. Идентификаторы пользователя могут быть заданы как символьными именами, так и числовыми идентификаторами **UID** и **GID**.

chgrp изменяет группу каждого заданного файла на группу, которая может быть представлена как именем группы, так и ее числовым идентификатором (**GID**).

ОПЦИИ POSIX

-R - Рекурсивное изменение владельца для каталогов и их содержимого.
-- - Завершает список опций.

Системное программирование для современных платформ

ln [опции] исходный [куда]
ln [опции] исходный... каталог

ОПИСАНИЕ

В **Unix** существует два вида ссылок, называемых жесткие ссылки и символьные ссылки. Жесткая ссылка является именем какого-либо файла. Все имена имеют одинаковый статус.

Символьная ссылка отличается от жесткой ссылки: она является специальным файлом, который содержит путь к другому файлу. Таким образом, символическая ссылка может указывать на файлы, которые находятся на других файловых системах и не нуждается в наличии того файла, на который она указывает. Когда происходит попытка доступа к файлу, ядро операционной системы заменяет ссылку на тот путь, который она содержит. (Кроме команды **rm** и системного вызова **unlink**). Для чтения состояния символьной ссылки, а также имени файла, на который она указывает, используются системные вызовы **lstat** и **readlink**. Для других системных вызовов, которые зависят и различаются для разных операционных систем, может осуществляться работа как с самой символьной ссылкой, так и с файлом, на который она указывает.

ln создаёт жёсткую ссылку на файл. При указании опции **-s**, создаётся символьная ссылка.

Системное программирование для современных платформ

Если задан только один файл, то для него делается ссылка в текущем каталоге с таким же именем, как у этого файла. В противном случае, если последний аргумент является именем существующего каталога, то **ln** создаст ссылки в этом каталоге для каждого из исходных файлов, с такими же именами как и у исходных файлов. В противном случае, если задано два файла, то создается ссылка с именем «куда» для исходного файла. Если последний аргумент не является каталогом и задано более чем два аргумента, то будет выдаваться сообщение об ошибке.

Невозможно создать символическую ссылку на каталог.

Конкретные операционные системы могут накладывать дополнительные ограничения.

ОПЦИИ POSIX

-f - Удалять существующие файлы 'куда'.
-- - Завершает список опций.

Системное программирование для современных платформ

rm [опции] файл…

ОПИСАНИЕ

rm удаляет каждый заданный файл. По умолчанию каталоги не удаляются, но если заданы опции **-r** или **-R**, то будет удаляться все дерево каталогов ниже заданного каталога, включая и его самого (без ограничения на глубину этого дерева). Будет выдано сообщение об ошибке, если последний компонент файла -- это `.' или `..' (чтобы избежать неприятных сюрпризов при задании команды `**rm -r .***' или ей подобных). Если задана опция **-i** или файл является недоступным на запись и при этом стандартный вывод -- это терминал и не задана опция **-f**, то **rm** выводит на стандартный вывод ошибок запрос на подтверждение удаления этого файла и читает ответ из стандартного ввода. Если ответ не утвердительный, то файл пропускается.

ОПЦИИ POSIX

-f - Не запрашивать подтверждения операции. Не выдавать диагностических сообщений. Не возвращать код ошибочного завершения, если ошибки были вызваны несуществующими файлами.
-i - Выводить запрос на подтверждение операции удаления (если заданы одновременно опции **-f** и **-i**, то срабатывает последняя указанная).
-r или **-R** - Рекурсивное удаление дерева каталогов.
-- - Завершает список опций.

Системное программирование для современных платформ

umask [-S] [маска]

Утилита **umask** задает маску создания файла в текущей среде командного интерпретатора равной значению, задаваемому операндом маска. Эта маска влияет на начальное значение битов прав доступа всех создаваемых далее файлов. Если **umask** вызвана в порожденном командном интерпретаторе или в отдельной среде выполнения, она не влияет на маску режима создания файлов в вызывающей среде. Поэтому утилиту **umask** нельзя использовать для изменения маски в текущем сеансе. Ее действие ограниичивается проверкой значения маски вызывающего сеанса. Чтобы изменить маску текущего сеанса, необходимо использовать одну из встроенных команд интерпретаторов.

Если операнд **маска** не указан, утилита **umask** выдает в стандартный выходной поток значение маски режима создания файлов вызывающего процесса.

ОПЦИИ

-S - Выдает результаты в символьном виде.

ПРИМЕРЫ

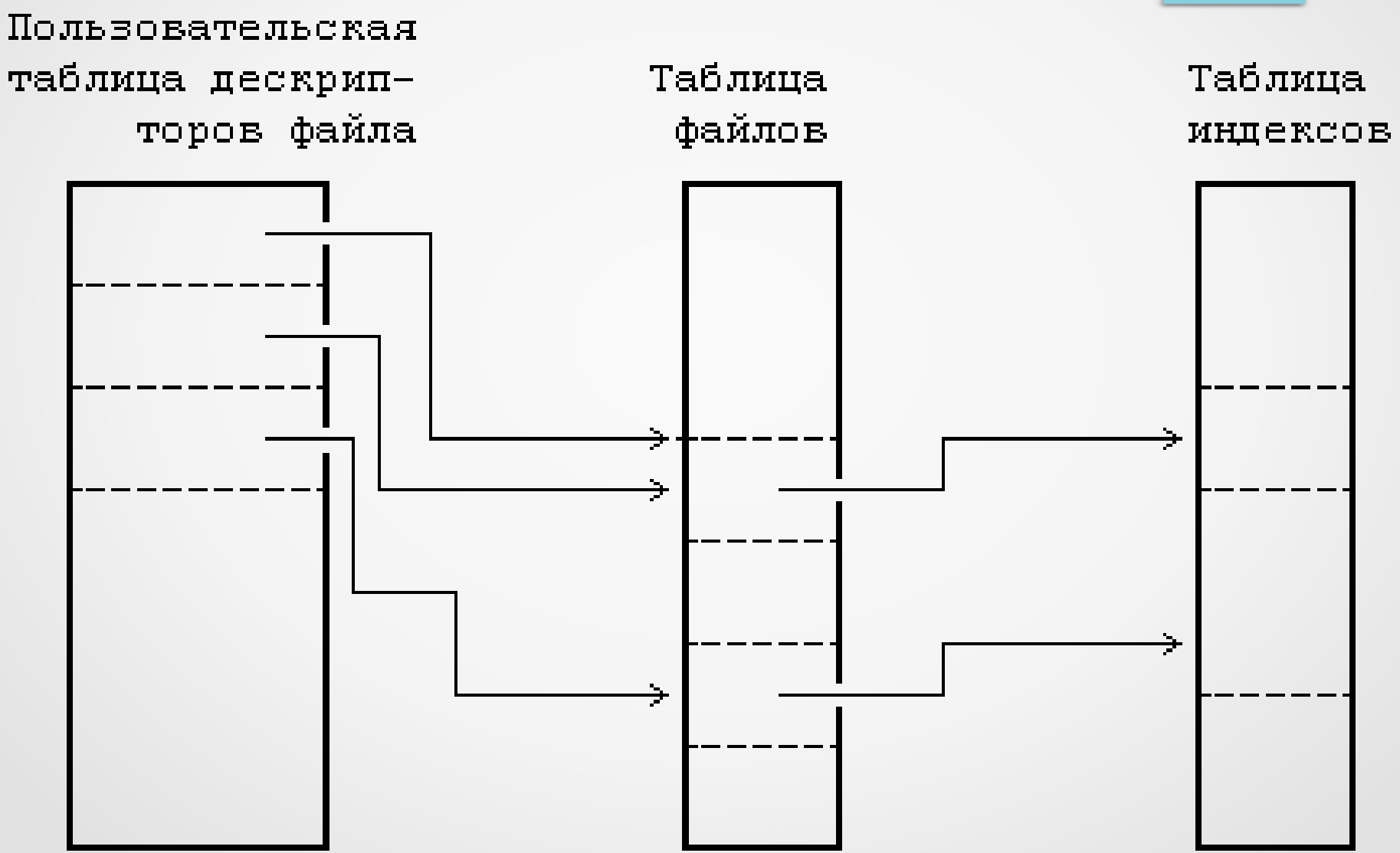
umask a=rx,ug+w или **umask 002**
устанавливает маску так, что у создаваемых файлов бит записи others сброшен
umask g-w
изменяет маску так, что у создаваемых файлов биты group и others сброшены
umask - -w
утанавливает маску так, что у создаваемых файлов все биты записи сброшены
umask -S u=rwx,g=rwx,o=rx

Системное программирование для современных платформ

Дескрипторы файлов

- Индекс файла в таблице открытых файлов
- Пользовательские и системные дескрипторы
- Стандартные дескрипторы 0,1,2 связаны с потоками ввода-вывода STDIN, STDOUT, STDERR
- Дескриптор является основным аргументом, указывающим на файл при низкоуровневом вводе-выводе (вместо struct FILE)

Системное программирование для современных платформ



Системное программирование для современных платформ

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *fname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *fname, struct stat *buf);
```

Функции возвращают информацию об указанном файле. Для этого не требуется иметь права доступа к файлу, но требуются права поиска во всех каталогах, указанных в полном имени файла.

stat возвращает информацию о файле fname и заполняет буфер buf. lstat идентична stat, но в случае символьных ссылок она возвращает информацию о самой ссылке, а не о файле, на который она указывает. fstat идентична stat, только возвращается информация об открытом файле, на который указывает fd (возвращаемый open(2)).

Системное программирование для современных платформ

```
struct stat {
    dev_t         st_dev;      /* устройство */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* режим доступа */
    nlink_t       st_nlink;    /* количество жестких ссылок */
    uid_t         st_uid;      /* идентификатор владельца */
    gid_t         st_gid;      /* идентификатор группы */
    dev_t         st_rdev;     /* тип устройства */
                               /* (если это устройство) */
    off_t         st_size;     /* общий размер в байтах */
    blksize_t     st_blksize;  /* размер блока ввода-вывода */
                               /* в файловой системе */
    blkcnt_t      st_blocks;   /* количество выделенных блоков */
    time_t        st_atime;    /* время последнего доступа */
    time_t        st_mtime;    /* время последней модификации */
    time_t        st_ctime;    /* время последнего изменения */
};
```

Системное программирование для современных платформ

Поле **st_size** задает размер файла (если он обычный или является символьной ссылкой) в байтах. Размер символьной ссылки - длина пути файла на который она ссылается, без конечного **NUL**.

Поле **st_blocks** задает размер файла в **512**-байтных блоках. (Оно может быть меньше, чем **st_size/512** например, когда в файле есть пропуски и это поддерживается файловой системой.) **st_blksize** задает "предпочтительный" размер блока для эффективного ввода/вывода в конкретной файловой и/или операционной системе. (Запись в файл более мелкими порциями может привести к некорректному чтению/изменению/повторной записи информации).

Некоторые файловые системы не реализуют все метки времени. Некоторые файловые системы позволяют обращаться к файлам так, что не происходит никаких изменений в поле **st_atime**. (См. **'noatime'** в **mount(8)**. Поле **st_atime** изменяется при доступе к файлу, например, при **execve(2)**, **mknod(2)**, **pipe(2)**, **utime(2)** и **read(2)** (если прочитано больше нуля байтов). Другие функции, например, **mmap(2)**, могут изменять, а могут и не изменять **st_atime** в зависимости от реализации. Поле **st_mtime** изменяется при модификациях файла, например, при выполнении **mknod(2)**, **truncate(2)**, **utime(2)** и **write(2)** (если записано больше нуля байтов). Более того, поле **st_mtime** каталога изменяется при создании и удалении файлов в этом каталоге. Поле st_mtime не изменяется при изменении владельца, группы, количества жёстких ссылок файла или режима доступа к нему. Поле **st_ctime** изменяется при записи или установке информации об **inode** (владельце, группе, количестве ссылок, режиме и т.д.).

Указанные далее макросы POSIX проверяют, является ли файл (поле **st_mode**):

S_ISLNK 0120000 - символьной ссылкой (Нет в POSIX.1-1996.)
S_ISREG 0100000 - обычным файлом
S_ISDIR 0040000 - каталогом
S_ISCHR 0020000 - символьным устройством
S_ISBLK 0060000 - блочным устройством
S_ISFIFO 0010000 - каналом FIFO
S_ISSOCK 0140000 — сокетом
S_IFMT 0170000 - битовая маска для полей типа файла

Системное программирование для современных платформ

Биты доступа (поле **st_mode**)

S_ISUID	0004000	бит **setuid**
S_ISGID	0002000	бит **setgid** (смотри ниже)
S_ISVTX	0001000	бит принадлежности (смотри ниже)
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 (не находящихся в группе)
S_IROTH	00004	все прочие имеют права чтения
S_IWOTH	00002	все прочие имеют права записи
S_IXOTH	00001	все прочие имеют права выполнения

Бит **S_ISGID** используется в нескольких случаях: указывает, что для данного каталога используется семантика **BSD** - файлы, создаваемые в нем, наследуют группу-владельца от этого каталога, а не от **GID** процесса, создавшего файл; в подкаталогах данного каталога также будет установлен бит **S_ISGID**. Если файл не имеет бита выполнения группой (**S_IXGRP**), то бит **setgid** означает жёсткую блокировку файла. Бит принадлежности (**S_ISVTX**) каталога означает, что файлы в этом каталоге могут быть удалены или переименованы только владельцем файла, владельцем каталога и суперпользователем. Конкретные операционные системы могут определять дополнительные значения поля st_mode, поэтому для переносимости программ настоятельно рекомендуется использовать мнемонические константы.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

В случае успеха возвращается **0**. При ошибке возвращается **-1**, а переменной **errno** присваивается номер ошибки.

Системное программирование для современных платформ

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

ОПИСАНИЕ

access проверяет, имеет ли процесс права на чтение или запись, или же просто проверяет, существует ли файл (или другой объект файловой системы), с именем **pathname**. Если **pathname** является символьной ссылкой, то проверяются права доступа к файлу, на который она ссылается.
mode - это маска, состоящая из одного или более флагов **R_OK**, **W_OK**, **X_OK** и **F_OK**.

R_OK, **W_OK** и **X_OK** запрашивают соответственно проверку существования файла и возможности его чтения, записи или выполнения. **F_OK** просто проверяет существование файла. Результаты проверки зависят от прав доступа к каталогам, находящимся по пути к файлу, заданному параметром **pathname**, и от прав доступа к каталогам и файлам, на которые ссылаются символьные ссылки, встреченные по пути.

Проверка осуществляется, используя реальные, а не эффективные идентификаторы пользователя и группы. Эффективные идентификаторы будут использоваться при действительной попытке выполнения той или иной операции. Это дает **setuid**-программам простой способ проверить права доступа настоящего пользователя.

Проверяются только биты прав доступа, а не тип файла или его содержимое. Таким образом, если каталог имеет "возможность записи", это, вероятно, означает, что в нем можно создавать файлы, а не что в этот каталог можно писать так же, как в обычный файл. Подобно этому файл из DOS может показаться "выполняемым", но системный вызов execve(2) завершится неудачно.

Если процесс имеет соответствующие привилегии, то некоторые реализации могут показать успех для **X_OK** даже если права на файл не содержат бит, разрешающий выполнение.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха (есть все запрошенные права) возвращается 0. При ошибке (по крайней мере один запрос прав из **mode** был неудовлетворен, или случилась другая ошибка), возвращается -1, а errno устанавливается должным образом.

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

ОПИСАНИЕ

Изменяет права доступа к файлу, заданному параметром **path** или файловым дескриптором **fd**.
Права задаются применением логической операции OR к следующим константам:

S_ISUID - 04000 установить при выполнении бит смены идентификатора пользователя
S_ISGID - 02000 установить при выполнении бит смены идентификатора группы
S_ISVTX - 01000 sticky бит
S_IRUSR (S_IREAD) - 00400 владелец может читать
S_IWUSR (S_IWRITE) - 00200 владелец может писать
S_IXUSR (S_IEXEC) - 00100 владелец может выполнять файл или искать в каталоге
S_IRGRP - 00040 группа-владелец может читать
S_IWGRP - 00020 группа-владелец может писать
S_IXGRP - 00010 группа-владелец может выполнять файл или искать в каталоге
S_IROTH - 00004 все остальные могут читать
S_IWOTH - 00002 все остальные могут писать
S_IXOTH - 00001 все остальные могут выполнять файл или искать в каталоге

Системное программирование для современных платформ

Эффективный идентификатор пользователя **(UID)** для вызывающего процесса должен быть нулем или совпадать с **UID** владельца файла.

Если эффективный **UID** процесса не равен нулю, а группа-владелец файла не совпадает с фактическим **GID** процесса или одним из его дополнительных **GID**'ов, то бит **S_ISGID** будет сброшен, но ошибки при этом не возникнет.

В зависимости от файловой системы, **suid** и **sgid** биты могут быть сброшены, когда происходит запись в файл. На некоторых файловых системах только суперпользователь может устанавливать **sticky** бит, который может иметь специальное значение. О значении **sticky** бита, а также **suid** и **sgid** битов на каталоги, см. **stat(2)**. На файловых системах **NFS** отмена некоторых прав доступа немедленно повлияет на открытые файлы, потому что контроль доступа осуществляется сервером, а открытые файлы обрабатываются клиентом. Добавление новых прав доступа может произойти не сразу, если на клиенте включено кэширование атрибутов.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается **0**. При ошибке возвращается **-1**, а **errno** устанавливается должным образом.

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

ОПИСАНИЕ

Изменяет владельца для файла, задаваемого параметрами **path** или **fd**. Только суперпользователь может изменять владельца файла. Владелец файла может изменять группу файла на любую группу, к которой он принадлежит. Суперпользователь может произвольно изменять группу. Если параметр **owner** или **group** заданы как **-1**, то соответствующий идентификатор не изменяется.

Когда владелец или группа исполняемого файла изменяются не-суперпользователем, то очищаются биты **S_ISUID** и **S_ISGID**. **POSIX** не требует, чтобы это происходило, когда суперпользователь выполняет **chown**; в этом случае поведение зависит от операционной системы. Если в правах доступа к файлу не установлен бит исполнения группой (**S_IXGRP**), то бит **S_ISGID** означает принудительную блокировку на этом файле и не очищается функцией **chown**.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается **0**. При ошибке возвращается **-1**, а **errno** устанавливается должным образом.

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *path, struct utimbuf *buf);

#include <sys/time.h>
int utimes(char *path, struct timeval *tvp[]);
```

ОПИСАНИЕ

utime изменяет время доступа или модификации **inode**, указанного с помощью **path**, делая его равным полям **actime** и **modtime** буфера **buf**, соответственно. Если **buf** равен **NULL**, то время доступа и модификации устанавливаются в текущее время. Структура **utimbuf** выглядит так:

```
struct utimbuf {
        time_t actime;  /* время доступа */
        time_t modtime; /* время модификации */};
```

В библиотеках **Linux (libc 4.4.1** и выше) **utimes** является просто оберткой для **utime**: **timeval[0].tv_sec** соответствует **actime**, а **timeval[1].tv_sec** соответствует **modtime**. Структура **timeval** выглядит так:

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

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается **0**. При ошибке возвращается **-1**, а **errno** устанавливается должным образом.

Системное программирование для современных платформ

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

ОПИСАНИЕ

link создает новую ссылку (также известную как "жесткая" ссылка) на существующий файл. Если **newpath** существует, он не будет перезаписан.

Это 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".

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается **0**. При ошибке возвращается **-1**, а **errno** устанавливается должным образом.

Системное программирование для современных платформ

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

ОПИСАНИЕ

symlink 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.

Символьные ссылки могут содержать такие компоненты пути, как .. которые, (если используются в начале ссылки), ссылаются на родительский каталог того каталога, в котором находится ссылка.

Символьная ссылка может указывать как на существующий, так и на несуществующий файлы; в последнем случае такая ссылка называется "висячей".

Права доступа к символьной ссылке не используются; её владелец игнорируется при поиске по ссылке, но проверяется при удалении или переименовании ссылки, находящейся в каталоге с установленным **sticky** битом.

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

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается **0**. При ошибке возвращается **-1**, а значение **errno** устанавливается должным образом.

Системное программирование для современных платформ

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

ОПИСАНИЕ

unlink удаляет имя из файловой системы. Если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он занимает освобождается для дальнейшего использования.
Если имя было последней ссылкой на файл, но какие-либо процессы всё ещё держат этот файл открытым, файл будет оставлен пока последний файловый дескриптор, указывающий на него, не будет закрыт.

Если имя указывает на символьную ссылку, ссылка будет удалена.

Если имя указывает на **сокет**, **FIFO** или устройство, имя будет удалено, но процессы, которые открыли любой из этих объектов могут продолжать его использовать.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успеха возвращается **0**. В случае ошибки возвращается **-1** и значение **errno** устанавливается соответствующим образом.

Системное программирование для современных платформ

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

ОПИСАНИЕ

rename переименовывает файл и, если требуется, перемещает его из одного каталога в другой. Все прочие "жесткие" ссылки на файл (созданные с помощью **link(2)**), не изменяются. Если **newpath** уже существует, то он будет вновь записан при наличии соответствующих прав, так что неизвестны условия, при которых другой процесс, пытающийся обратиться к **newpath**, не обнаружит его. Если **newpath** существует, но операция завершается ошибкой или система аварийно завершает работу, **rename** гарантирует, что **newpath** останется нетронутым. При повторной записи, однако, есть вероятность, что **oldpath** и **newpath** будут ссылаться на один и тот же файл. Если **oldpath** является символьной ссылкой, то она переименовывается; если **newpath** является символьной ссылкой, то будет вновь записан файл, на который она указывает.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

В случае успешного завершения вызова возвращается **0**. При ошибке возвращается **-1**, а переменной **errno** присваивается номер ошибки.

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

ОПИСАНИЕ

umask устанавливает значение **umask** равным **mask & 0777. umask** используется функцией **open(2)** для установки стандартных прав только что созданного файла. Если быть точнее, права в **umask** исключаются из прав доступа параметра **mode** в функции **open(2)** (так, например, стандартное значение **umask**, равное **022**, приводит к тому, что устанавливаемое значение прав доступа ко вновь созданному файлу становятся равными **0666 & ~022 = 0644 = rw-r--r--** в обычном случае, если установленное значение **mode** равно **0666**-и).

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Этот системный вызов всегда завершается успешно и возвращает предыдущее значение **umask.**

Системное программирование для современных платформ

Типичные значения errno при файловых операциях

EISDIR — попытка скопировать каталог в файл и т.п.
EXDEV — попытка работы с типами файлов, неподдерживаемых данной файловой системой, тж. Попытка записи в файловую систему, физически защищённую от записи
ENOTEMPTY — попытка удалить непустой каталог
EBUSY — попытка удалить или переместить текущий или корневой каталог какого-то процесса
EEXIST - новое имя пути совпадает с началом старого имени.
EINVAL - попытка сделать каталог своим собственным подкаталогом.
EMLINK - достигнуто максимальное значение счётчика ссылок
ENOTDIR - компонент, используемый как каталог, в действительности не является каталогом.
EFAULT - аргумент указывающий на каталог за пределами доступного адресного пространства.

Системное программирование для современных платформ

Типичные значения errno при файловых операциях

EACCES — нарушение прав доступа для каталогов
EPERM — нарушение прав доступа для файлов
ENAMETOOLONG - имена являются слишком длинными
ENOENT — операции не существующим каталогом или там, где подразумевается каталог, используется "висячая" символьная ссылка
ENOMEM - ядру не хватило памяти для выполнения операции
EROFS - файл находится в файловой системе, предназначенной только для чтения
ELOOP — операции с «висячей» символьной ссылкой
ENOSPC - В устройстве, содержащем файл, нет места для новой записи

Системное программирование для современных платформ

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

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

ОПИСАНИЕ

Вызов **open()** используется, чтобы преобразовать путь к файлу в файловый дескриптор. Если системный вызов завершается успешно, возвращённый дескриптор является наименьшим дескриптором, который еще не открыт процессом. В результате этого вызова появляется новый открытый файл, не разделяемый никакими процессами (разделяемые открытые файлы могут возникнуть, когда посылается системный вызов **fork(2)**). Новый описатель файла будет оставаться открытым при выполнении функции **exec(2)**, если не указан флаг **close-on-exec** (смотри описание **fcntl(2)**). Маркер чтения-записи устанавливается в начале файла. Параметр **flags** - это флаги **O_RDONLY**, **O_WRONLY** или **O_RDWR**, открывающие файлы "только для чтения", "только для записи" и для чтения и записи соответственно, которые собираются с помощью побитовой операции **OR** из следующих значений: **O_CREAT**, **O_EXCL**, **O_NOCTTY**, **O_TRUNC**, **O_APPEND**, **O_NONBLOCK**, **O_SYNC**, **O_NOFOLLOW**, **O_DIRECTORY**, **O_LARGEFILE**

Системное программирование для современных платформ

O_CREAT — если файл не существует, то он будет создан. Владелец файла устанавливается в значение **EUID** процесса. Группа устанавливается либо в значение **EGID** процесса, либо в значение идентификатора группы родительского каталога (зависит от типа файловой системы, параметров подсоединения (**mount**) и режима родительского каталога);
O_EXCL — если он 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_NOCTTY — если **path** указывает на терминальное устройство, то оно не станет терминалом управления процесса, даже если процесс такового не имеет, используется для создания процессов-демонов;
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 других случаях действие флага O_TRUNC не определено;
O_APPEND — файл открывается в режиме добавления. Перед каждой операцией **write** маркер будет устанавливаться в конце файла, как если бы использовался **lseek**);
O_NONBLOCK или **O_NDELAY** — если есть поддержка со стороны ОС, то файл открывается в режиме **non-blocking**. Ни **open**, ни другие последующие операции над возвращаемым описателем файла не заставляют вызывающий процесс ждать. Программист должен самостоятельно написать собственный механизм блокировок. **POSIX** определяет этот флаг только для **FIFO**-файлов;
O_SYNC — файл открывается в режиме синхронного ввода-вывода. Все вызовы **write** для соответствующего дескриптора файла блокируют вызывающий процесс до тех пор, пока данные не будут физически записаны;
O_DSYNC — гарантирует целостность данных; данные и любая другая информация, которую операционная система должна найти, записываются на диск до возвращения **write()**. Однако, вспомогательные данные, такие, как время модификации или доступа к файлу, записываются асинхронно;
O_RSYNC — если данные находятся в кэше, то файл синхронизируется до выполнения функции **read()**;
O_NOFOLLOW — если **path** - это символьная ссылка, то **open** содержит код ошибки. Не является **POSIX**-совместимым;
O_DIRECTORY — если **path** не является каталогом, то open укажет на ошибку. Не является **POSIX**-совместимым. Этот флаг не следует использовать вне реализации **opendir**;
O_LARGEFILE — на **32**-битных системах, поддерживающих файловые системы объёмом более 4Тб, этот флаг позволяет открывать файлы, длина которых больше **31**-ого бита

Системное программирование для современных платформ

Некоторые из вышеописанных флагов могут быть изменены с помощью **fctnl** после открытия файла. Аргумент **mode** задает права доступа, которые используются в случае создания нового файла. Они модифицируются обычным способом, с помощью **umask** процесса; права доступа созданного файла равны (**mode & ~umask**). Обратите внимание, что этот режим применяется только к правам создаваемого файла; **open** создает файл только для чтения, но может вернуть дескриптор с установленными флагами для чтения и записи.

В **mode** используются такие же мнемонические константы, как и функциях семейства **chmod**:
S_IRWXU - 00700 владелец файла имеет права на чтение, запись и выполнение;
S_IRUSR (S_IREAD) - 00400 владелец файла имеет права на чтение файла;
S_IWUSR (S_IWRITE) - 00200 владелец файла имеет права на запись в файл;
S_IXUSR (S_IEXEC) - 00100 владелец файла имеет права на исполнение файла;
S_IRWXG - 00070 группа имеет права на чтение, запись и исполнение 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 исполнение файла;
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.

Аргумент **mode** всегда должен быть указан при использовании **O_CREAT**; во всех остальных случаях этот параметр игнорируется. **creat** эквивалентен **open** с **flags**, которые равны **O_CREAT | O_WRONLY | O_TRUNC.**

Системное программирование для современных платформ

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

open и **creat** возвращают новый дескриптор файла или **-1** в случае ошибки (в этом случае значение переменной **errno** устанавливается должным образом). Заметьте, что **open** может открывать файлы устройств, но **creat** не может создавать их, поэтому используйте для создания функцию **mknod(2)**.

Если создается файл, то его время последнего доступа, создания и модификации устанавливаются в значение текущего времени, а также устанавливаются поля времени модификации и создания родительского каталога. Иначе, если файл изменяется с флагом **O_TRUNC**, то его время создания и время изменения устанавливаются в значение текущего времени.

ПРИМЕЧАНИЯ

Эффект (неопределенный изначально) от **O_RDONLY | O_TRUNC** отличается в разных реализациях. Во многих системах файл в действительности обрезается.

Системное программирование для современных платформ

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

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

ОПИСАНИЕ

read() пытается записать **count** байтов файлового дескриптора **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 может быть определен.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

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: например, данные 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 маркера ввода-вывода не определена.

Системное программирование для современных платформ

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

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

ОПИСАНИЕ

write записывает до **count** байтов из буфера **buf** в файл, на который ссылается файловый дескриптор **fd**. **POSIX** указывает на то, что вызов **write()**, произошедший после вызова **read()** возвращает уже новое значение. Заметьте, что не все файловые системы соответствуют стандарту **POSIX**.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

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. Для специальных файлов результаты не могут быть перенесены на другую платформу.

Системное программирование для современных платформ

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

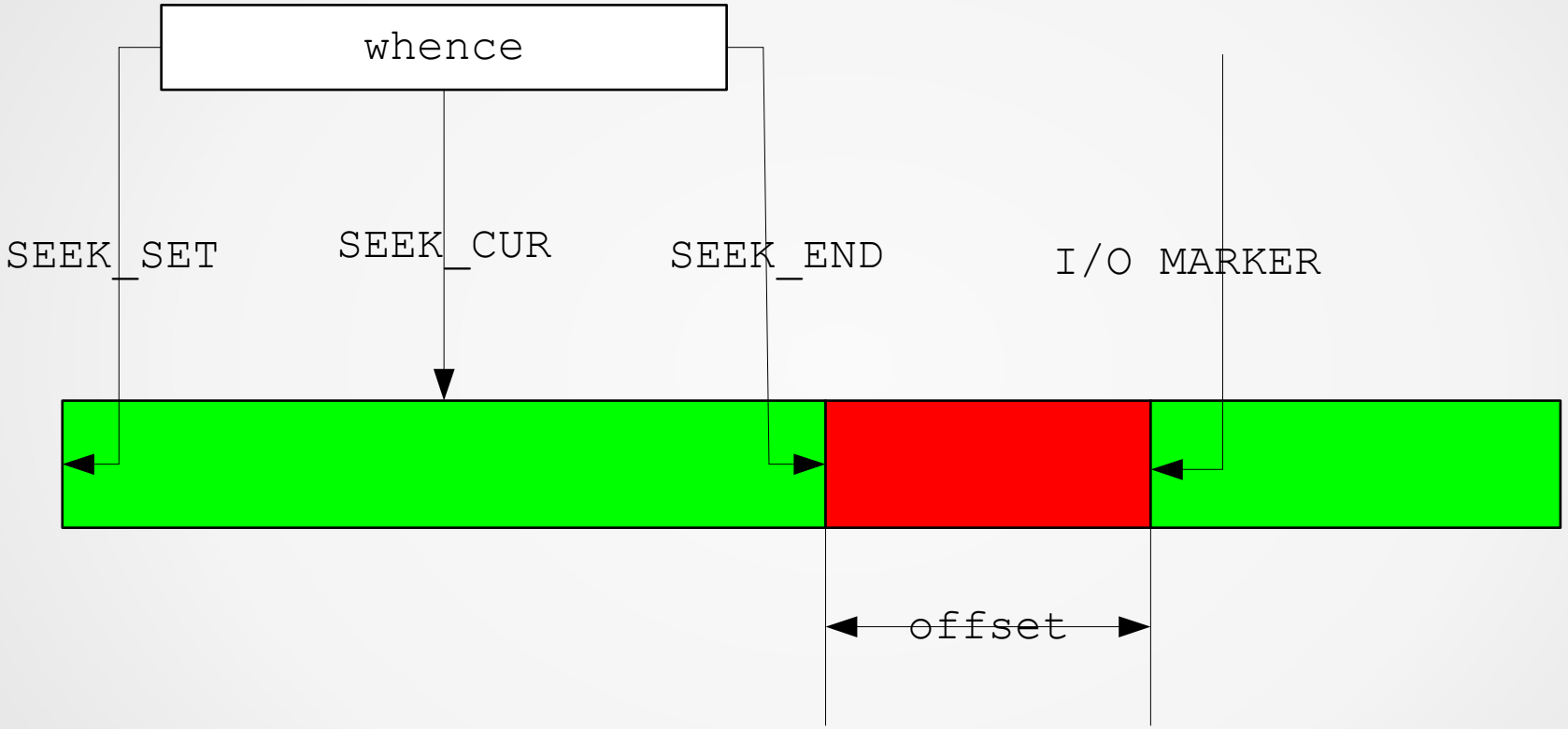
ОПИСАНИЕ

Функция **lseek** устанавливает смещение для файлового дескриптора **fd** в значение аргумента **offset** в соответствии с указателем 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 (пока в этот промежуток не будут фактически записаны данные).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном выполнеии **lseek** возвращает получившееся в результате смещение в батах от начала файла. В противном случае, возвращается значение **(off_t)-1** и **errno** показывает ошибку.

Системное программирование для современных платформ



Системное программирование для современных платформ

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

int fcntl(int fd, int cmd, ...);
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

ОПИСАНИЕ

fcntl выполняет одну из различных дополнительных операций над файловым дескриптором **fd**. Эта операция определяется содержимым аргумента **cmd**.

F_DUPFD
Ищет наименьший доступный номер файлового дескриптора, который больше **arg** и делает его копией дескриптора **fd**. Фактически это другая форма вызова **dup2(2)** которая используется с явным указанием файлового дескриптора. Старый и новый дескрипторы могут использоваться равнозначно. Они разделяют одни и те же блокировки, указатели на позиции в файле и флаги; например, если позция в файле изменяется с помощью **lseek** для одного из дескрипторов, то эта же позиция также будет изменена и для другого.

Однако, данные два дескриптора не разделяют флаг **close-on-exec**. Флаг **close-on-exec** в копии выключен. Это означает, что дескриптор не будет закрыт в случае вызова **exec**.

При успешном выполнении этой операции, возвращается новый файловый дескриптор

Системное программирование для современных платформ

F_GETFD - Читает флаг **close-on-exec**. Если бит **FD_CLOEXEC** установлен в **0**, то файл будет оставлен открытым при вызове **exec**, в противном случае он будет закрыт.
F_SETFD - Устанавливает флаг close-on-exec в значение, заданное битом **FD_CLOEXEC** аргумента **arg**.

Флаги состояния файла

Любой файловый дескриптор имеет несколько связанных с ним флагов, которые инициализируются вызовом **open(2)** и, возможно, изменяются затем вызовом **fcntl(2)**. Эти флаги разделяются между копиями (сделанными с помощью **dup(2)**, **fork(2)**, и других вызовов) этого же файлового дескриптора.
Эти флаги и их смысл описываются на странице руководства **open(2)**.

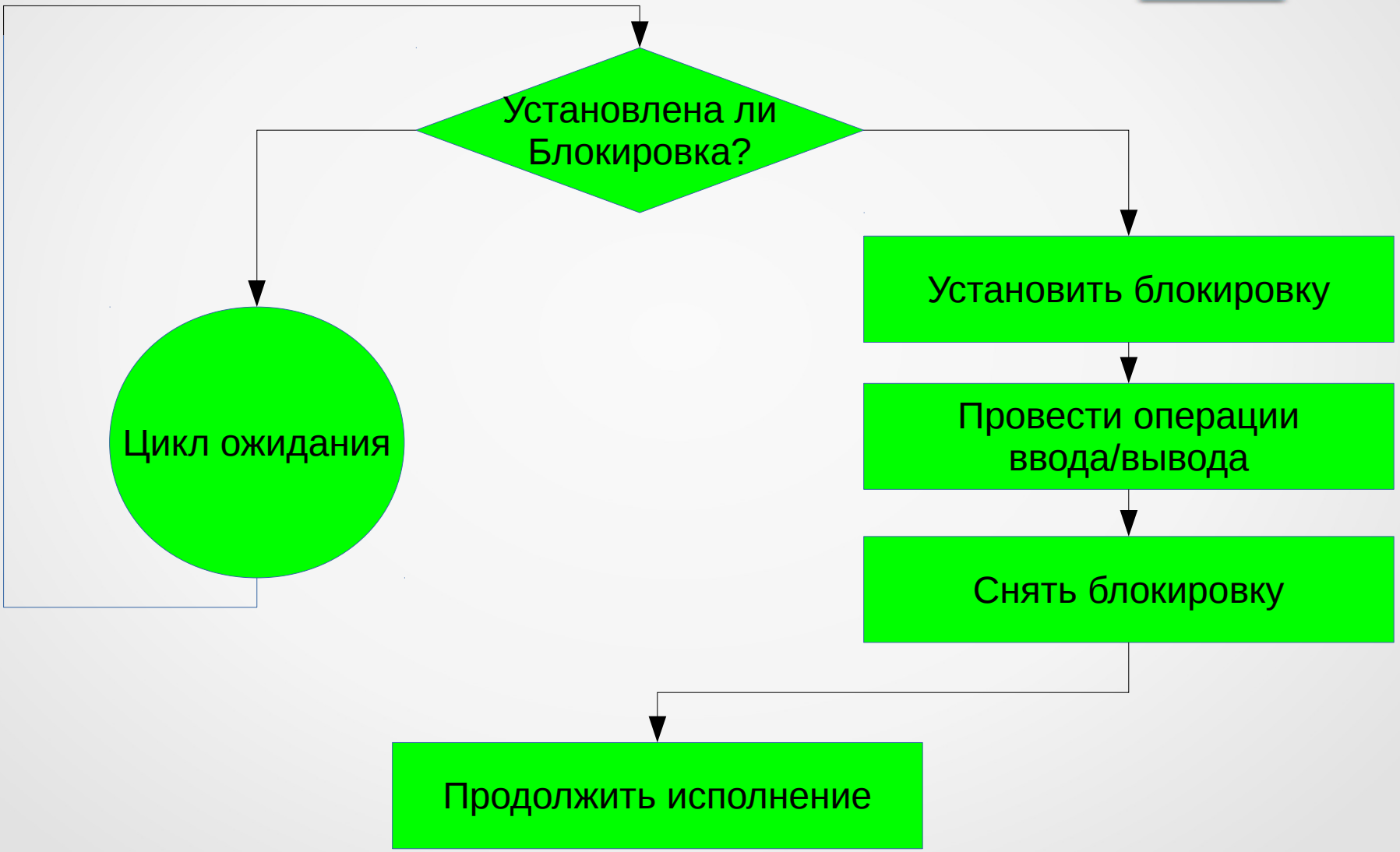
F_GETFL - Читает флаги файлового дескриптора.
F_SETFL - Устанавливает часть флагов, относящихся к состоянию файла, согласно значению, указанному в аргументе **arg**. Оставшиеся биты (режим доступа, флаги создания файла) в значении **arg** игнорируются. Доступнтые для изменеия флаги могут различаться в зависимости от используемой операционной системы. В **Linux** данная команда может изменять только флаги **O_APPEND**, **O_NONBLOCK**, **O_ASYNC** и **O_DIRECT**.

Системное программирование для современных платформ

Блокировки файлов

- механизм избегания race condition и повреждения структур данных при доступе к файлам
- может устанавливаться на файл целиком (устарело) или на область файла
- блокировка записи (эксклюзивная): один процесс может записывать, никто больше не может ни читать, ни записывать файл
- блокировка чтения (разделяемая): сколько угодно процессов могут читать файл, никто не может записывать
- делятся на обязательные (ядерные, mandatory) и факультативные (пользовательские, advisory) блокировки
- обязательные блокировки могут приводить к состоянию dead-locks
- факультативные блокировки – соглашение «safe programming»
- для управления факультативными блокировками используется системный вызов fcntl

Системное программирование для современных платформ



Системное программирование для современных платформ

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

int fcntl(int fd, int cmd, struct flock *lock);
```

F_GETLK, **F_SETLK** и **F_SETLKW** используются для установки, снятия и тестирования существующих блокировок записи. Третий аргумент **lock** является указателем на структуру **flock**, служащую для описания блокировки. **POSIX** гарантирует наличие следующих полей (в произвольном порядке).

```
struct flock {
    ...
    short l_type;    /* Тип блокировки: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;  /* Как интерпретировать l_start: SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Начальное смещение для блокировки */
    off_t l_len;     /* Количество байт для блокировки */
    pid_t l_pid;     /* PID процесса блокирующего нашу блокировку (F_GETLK only) */
    ...};
```

Поля **l_whence**, **l_start** и **l_len** определяют область, который мы хотим заблокировать. **l_whence** и **l_start** — аналогичны соответствующим аргументам системного вызова **lseek(2)**. **l_len** - это неотрицательное целое число (но см. ЗАМЕЧАНИЯ ниже), которое задаёт количество байт, которые будут заблокированы. Байты следующие после конца файла могут быть заблокированы, но это нельзя сделать для байтов, которые находятся перед началом файла. Значение **0** для **l_len** имеет специальное назначение: блокировка всех байтов, начиная от позиции, заданной **l_whence** и **l_start** до конца файла, не зависимо от того, насколько велик файл. Поле **l_type** может быть использовано для указания типа блокировки: чтение (**F_RDLCK**) или запись (**F_WDLCK**). Блокировку чтения (разделяемая) на область файла может удерживать любое количество процессов, но только один процесс может удерживать блокировку записи (эксклюзивная). Любая эксклюзивная блокировка исключает все другие блокировки, как разделяемые так и эксклюзивные. Один процесс может удерживать только один тип блокировки области файла; если происходит новая блокировка на уже заблокированную область, то существующая блокировка преобразуется в новый тип блокировки. (Такие преобразования могут привести к разбиению, уменьшению или срастанию с существующей блокировкой, если диапазон байт, заданный для новой блокировки неточно совпадает с диапазоном существующей блокировки.)

Системное программирование для современных платформ

F_SETLK - Установить блокировку (когда **l_type** установлено в значение **F_RDLCK** или **F_WRLCK**) или снять блокировку (когда **l_type** установлено в значение **F_UNLCK**) или байты, заданны с помощью полей **l_whence**, **l_start** и **l_len** структуры **lock**. Если конфликтующая блокировка удерживается другим процессом, то данный вызов вернёт **-1** и установит значение **errno** в **EACCES** или **EAGAIN**.

F_SETLKW - Если устнаовлен **F_SETLK**, но для файла удерживается конфликтующая блокировка, то происходит ожидание снятия блокировки. Если во время ожидания поступил сигнал, то данный вызов прерывается и (после возврата из обработчика сигнала) из него происходит немедленный возрат (где возвращаемое значение установлено в **-1**, а **errno** установлено в значение **EINTR**).

F_GETLK - При входе в этот вызов, **lock** описывает блокировку, которую мы должны бы установить на файл. Если такая блокировка не может быть установлена, **fcntl()** по факту не устанавливает её, но возвращает **F_UNLCK** в поле **l_type** структуры **lock** и оставляет другие поля структуры неизменёнными. Если одна или более несовместимых блокировок мешают установке нашей блокировки, то **fcntl()** возвращает подробности об одной из этих блокировок в полях **l_type**, **l_whence**, **l_start** и **l_len** структуры **lock** и устанавливает **l_pid** в значение **PID** того процесса, который удерживает блокировку. Для того, чтобы установить блокировку на чтение, **fd** должен быть открыт на чтение. Для того, чтобы установить блокировку на запись, **fd** должен быть открыт на запись. Чтобы установить оба типа блокировки, дескриптор должен быть открыт на запись и на чтение. Также, как и при снятии блокировки через явное указание **F_UNLCK**, блокировка автоматически снимается, когда процесс завершается или если он закрывает любой файловый дескриптор, ссылающийся на файл, на котором удерживается блокировка. Это плохо: это означает, что процесс может потерять блокировки на файлах типа **/etc/passwd** или **/etc/mtab**, когда по какой-либо причине библиотечная функция производит их открытие, чтение и закрытие. Блокировки не наследуются процессом-потомком, созданным через **fork(2)**, но сохраняются при вызове **execve(2)**. Поскольку буферизация, выполняется через библиотеку **stdio(3)**, использование блокировок с функциями в этом пакете нужно избегать; вместо этих функций используйте **read(2)** и **write(2)**.

Системное программирование для современных платформ

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

int readv(int fd, const struct iovec *vector, size_t count);
int writev(int fd, const struct iovec *vector, size_t count);
```

ОПИСАНИЕ

Функция **readv()** считывает **count** блоков из файла, ассоциированного с дескриптором **fd**, в несколько буферов, описанных в **vector**. Функция **writev()** записывает минимум count блоков, описанных в **vector**, в файл, ассоциированный с дескриптором **fd**.

vector указывает на структуру **struct iovec**, заданную в **<sys/uio.h>** как

```
struct iovec {
void *iov_base;    /* Начальный адрес */
size_t iov_len;    /* Количество байтов */};
```

Буферы обрабатываются в следующем порядке: **vector[0]**, **vector[1]**, ... **vector[count]**.

Функция **readv()** работает так же, как и **read(2)**, за исключением того, что заполняет несколько буферов. Функция **writev()** работает так же, как и **write(2)**, за исключением того, что будет записано несколько буферов.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Функции **readv()** и **writev()** возвращает количество прочитанных или записанных байтов или **-1** в случае ошибки.

Системное программирование для современных платформ

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

ОПИСАНИЕ

dup и **dup2** создают копию файлового дескриптора **oldfd**.
Старый и новый дескрипторы можно использовать друг вместо друга. Они имеют общие блокировки, указатель позиции в файле и флаги; например, если позиция в файле была изменена с помощью **lseek**, на одном из дескрипторов, то эта позиция также меняется и на втором.

Два дескриптора, однако, каждый имеют свой собственный флаг **close-on-exec**.
dup использует первый доступный номер дескриптора.
dup2 делает **newfd** копией **oldfd**, закрывая **newfd**, если требуется.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

dup и **dup2** возвращают новый дескриптор или **-1**, если произошла ошибка (в этом случае **errno** устанавливается соответствующим образом).

ОШИБКИ

Если **newfd** был открыт, любые ошибки, которые могли бы случиться во время **close()**, теряются. Осторожный программист не будет использовать **dup2** не закрыв сперва **newfd**.

Системное программирование для современных платформ

```
#include <unistd.h>
int pipe(int fd[2]);

#include <fcntl.h>
#include <unistd.h>
int pipe2(int fd[2], int flags);
```

ОПИСАНИЕ

pipe создает пару файловых дескрипторов, указывающих на неименованный канал, и помещает их в массив, на который указывает **fd**. **fd[0]** предназначен для чтения, а **fd[1]** предназначен для записи.
pipe2 действует аналогично **pipe()**, однако позволяет устанавливать флаги **O_NONBLOCK** и **O_CLOEXEC** без дополнительного вызова **fcntl()**. Не является **POSIX**-совместимым.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

При удачном завершении вызова возвращаемое значение равно нулю. При ошибке возвращается **-1**, а переменной **errno** присваивается номер ошибки.

Системное программирование для современных платформ

Спасибо за внимание!

:-)

Системное программирование для современных платформ

Системное программирование для современных платформ

Системное программирование для современных платформ