Содержание

1 open, creat

open, creat — открыть и возможно создать файл.

Использование

```
#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);
```

Описание

Системный вызов `open()` используется для преобразования пути к файлу в файловый дескриптор (небольшое неотрицательное число, которое используется в операциях ввода/-вывода `read`, `write` и др.). Если вызов завершился успешно, возвращается файловый дескриптор с минимальным номером, не открытый данным процессом. Вызов создаёт новый открытый файл, не разделяемый с другими процессами (однако разделяемые открытые файлы могут возникнуть вследствие вызова `fork()`). Новый файловый дескриптор установлен как не закрываемый в результате операции `exec()` (см. также `fcntl()`). Текущее положение в файле установлено на начало файла.

`flags` может быть один из `O_RDONLY`, `O_WRONLY` или `O_RDWR`, что означает открытие только на чтение, только на запись, либо и на чтение, и на запись.

`flags` также могут быть побитово объединены с одним или несколькими из следующих.

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

`O_EXCL` — когда используется с `O_CREAT`, если файл с таким именем уже существует, вызов завершается с ошибкой.

`O_TRUNC` — если файл уже существует, он очищается.

`O_APPEND` — файл открывается в режиме добавления. Перед каждой записью, указатель текущего положения в файле позиционируется на конец файла, как при `lseek()`.

Некоторые из этих необязательных флагов могут быть изменены с использованием `fcntl` после того, как файл был открыт.

`mode` определяет права доступа к файлу, если создаётся новый файл. Права доступа модифицируются с использованием `umask` процесса как обычно (`mode & ~umask`).

Для `mode` существуют символические константы:

S_IRWXU	00700	пользователь (владелец файла) имеет права на запись, чтение и выполнение.
S_IRUSR(S_IREAD)	00400	пользователь имеет права на чтение.
S_IWUSR(S_IWRITE)	00200	пользователь имеет права на запись.
S_IXUSR(S_IEXEC)	00100	пользователь имеет права на выполнение.
S_IRWXG	00070	группа имеет права на чтение, запись и выполнение.
S_IRGRP	00040	группа имеет права на чтение.
S_IWGRP	00020	группа имеет права на запись.
S_IXGRP	00010	группа имеет права на выполнение.
S_IRWXO	00007	прочие имеют права на чтение, запись и выполнение.
S_IROTH	00004	прочие имеют права на чтение.
S_IWOTH	00002	прочие имеют права на запись.
S_IXOTH	00001	прочие имеют права на выполнение.

mode должен всегда быть указан, когда O_CREAT указан в flags. mode игнорируется в противном случае.

creat эквивалентен open с флагами O_CREAT|O_WRONLY|O_TRUNC.

Возвращаемое значение

open и creat возвращают новый файловый дескриптор, или -1 в случае ошибки, тогда errno устанавливается в код ошибки. open может открывать специальные файлы устройств, но не может создавать их, для этого используется mknod.

Коды ошибок

EEXIST	файл с данным путём существует и O_CREAT и O_EXCL были заданы в флагах.
EISDIR	данный путь указывает на каталог, а запрошенный метод доступа предполагает запись.
EACCES	запрошенный режим доступа не разрешён, либо один из каталогов в пути к файлу не позволяет поиск в нем, либо файл не существует и в доступе на запись к родительскому каталогу отказано.
ENAMETOOLONG	путь слишком длинный.
ENOENT	компонента имени каталога в пути не существует, либо висячая символическая ссылка.
ENOTDIR	компонента пути, используемая как каталог, не является каталогом.
EROFS	путь указывает на файл на файловой системе, доступной только на чтение, и режим записи был запрошен.
ETXTBSY	путь указывает на исполняемый файл, который запущен на выполнение, и режим записи был запрошен.
EFAULT	pathname представляет собой недопустимый адрес.
ELOOP	слишком много символических ссылок при трансляции пути.
ENOSPC	на устройстве не осталось свободного места.
ENOMEM	недостаточно памяти для ядра.
EMFILE	процесс уже открыл максимальное количество файлов.
ENFILE	достигнут лимит на число открытых файлов в системе.

2 close

close — закрыть файловый дескриптор.

Использование

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

Описание

close закрывает файловый дескриптор, который после этого не ссылается ни на какой файл и может быть использован повторно.

Если fd — последняя копия некоторого файлового дескриптора, все ресурсы, ассоциированные с ним, освобождаются; если данный дескриптор был последним дескриптором, ссылающимся на некоторый файл, который был удалён с помощью unlink, файл и все его данные удаляются.

Возвращаемое значение

`close` возвращает 0 при успешном завершении и -1 в случае ошибки.

Коды ошибок

`EBADF`	`fd` — неверный файловый дескриптор.

Замечания

Отсутствие проверки возвращаемого `close` значения является очень частой, но тем не менее серьёзной ошибкой. В файловых системах, которые используют буферизацию записи для увеличения производительности, `write` может успешно завершаться, хотя данные ещё не были записаны на диск. Ошибка может быть возвращена при последующих операциях `write`, и гарантировано, что ошибка будет возвращена при закрытии файла. Отсутствие проверки на возвращаемое `close` значение может вести к незамеченной потере данных.

3 dup, dup2

`dup`, `dup2` — скопировать файловый дескриптор.

Использование

```c
#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 файловые дескрипторы полностью эквивалентны. Они разделяют указатели позиции в файле и флаги. Например, если позиция в файле модифицирована с помощью `lseek` у одного из файловых дескрипторов, она также изменяется и у другого файлового дескриптора.

Старый и новый дескрипторы не разделяют флаг закрытия при `exec`.

Для нового дескриптора `dup` использует свободный дескриптор с минимальным номером.

`dup2` делает дескриптор `newfd` копией `oldfd`, закрывая `newfd` при необходимости.

Возвращаемое значение

`dup` и `dup2` возвращают новый дескриптор, или -1 в случае ошибки, и в этом случае `errno` устанавливается в код ошибки.

Коды ошибок

`EBADF`	неверный файловый дескриптор `oldfd` или `newfd`.
`EMFILE`	процесс уже открыл максимальное количество файлов.

4 read

read — читать из файлового дескриптора.

Использование

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

Описание

read пытается считать до count байт включительно из файлового дескриптора fd в буфер, начинающийся с buf.

Если count равен 0, read возвращает 0 без других последствий.

Возвращаемое значение

При успешном завершении возвращается число считанных байт (ноль означает конец файла), и текущая позиция в файле продвигается на это число. Не является ошибкой ситуация, когда число считанных байт меньше числа запрошенных байт, это может иметь место, например, при чтении с терминала, или когда read был прерван приходом сигнала. При ошибке возвращается -1 и переменная errno устанавливается в код ошибки.

Коды ошибок

EINTR	Системный вызов был прерван приходом сигнала, и не было считано никаких данных.
EAGAIN	Для файлового дескриптора установлен неблокирующийся режим (O_NONBLOCK), и нет данных доступных немедленно.
EIO	Ошибка ввода/вывода. Может случиться, например, если фоновый процесс пытается считать данные с управляющего терминала, и процесс блокирует SIGTTIN.
EISDIR	fd ссылается на каталог.
EBADF	fd — неверный файловый дескриптор или не открыт на чтение.
EINVAL	fd связан с объектом, который не допускает чтения.
EFAULT	buf представляет собой недопустимый адрес.

5 write

write — записать данные в файловый дескриптор.

Использование

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

Описание

write записывает до count байт включительно в файл, на который ссылается файловый дескриптор fd, из буфера, начинающегося по адресу buf.

Возвращаемое значение

При успешном завершении возвращается количество записанных байт (0 обозначает, что ничего не было записано). При ошибке возвращается -1, и переменная `errno` устанавливается в код ошибки. Если `count` равен 0, и файловый дескриптор ссылается на обычный файл, возвращается 0 без других эффектов.

Коды ошибок

`EINTR`	Системный вызов был прерван приходом сигнала, и не было записано никаких данных.
`EAGAIN`	Для файлового дескриптора установлен неблокирующийся режим (`O_NONBLOCK`), и данные не могут быть записаны немедленно.
`EIO`	Ошибка ввода/вывода.
`EISDIR`	`fd` ссылается на каталог.
`EBADF`	`fd` — неверный файловый дескриптор или не открыт на запись.
`EINVAL`	`fd` связан с объектом, который не допускает записи.
`EFAULT`	`buf` представляет собой недопустимый адрес.
`EPIPE`	`fd` связан с каналом или сокетом, другой конец которого закрыт. В этом случае процесс получает сигнал `SIGPIPE`, после чего возвращается эта ошибка.
`ENOSPC`	На устройстве не осталось места для данных.

6 lseek

`lseek` — переместить указатель текущего положения в файле.

Использование

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

Описание

Системный вызов `lseek` перемещает указатель текущего положения в файловом дескрипторе `fildes` на значение `offset` согласно директиве `whence`.

`SEEK_SET`	0	`offset` отсчитывается от начала файла.
`SEEK_CUR`	1	`offset` отсчитывается от текущего положения.
`SEEK_END`	2	`offset` отсчитывается от текущего размера файла.

`lseek` позволяет устанавливать указатель положения за текущий конец файла. Если после этого в это место будут записаны данные, чтения в «дыре» будут возвращать нули до тех пор, пока в дыру не будут записаны данные.

Возвращаемое значение

При успешном завершении `lseek` возвращает новое положение указателя относительно начала файла, измеренное в байтах. При ошибке возвращается значение `(off_t) -1`, и переменная `errno` устанавливается в код ошибки.

Коды ошибок

EBADF	fildes — неверный файловый дескриптор.
EINVAL	whence имеет недопустимое значение.
ESPIPE	fildes ассоциирован с каналом, сокетом или FIFO.

Ограничения

Не все устройства (например терминалы), поддерживают операцию lseek. Поведение lseek в этом случае неопределено.

7 stat, fstat, lstat

stat, fstat, lstat — получить информацию о файле.

Использование

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

Описание

Эти системные вызовы возвращают информацию о заданных файлах. Для получения информации о файле достаточно иметь права поиска на все каталоги, указанные в пути к файлу.

stat возвращает информацию о файле с путём file_name и заполняет структуру buf.

lstat идентичен stat, кроме того, что если указанное имя является символической ссылкой, возвращается информация о самой символической ссылке, а не о файле, на который она указывает.

fstat идентичен stat, кроме того, что возвращается информация о файле по файловому дескриптору filedes, открытому, например, с помощью open.

Системные вызовы заполняют структуру stat, содержащую следующие поля.

```
struct stat
{
  dev_t         st_dev;      /* устройство */
  ino_t         st_ino;      /* индексный дескриптор (inode) */
  mode_t        st_mode;     /* права доступа и флаги */
  nlink_t       st_nlink;    /* число ссылок */
  uid_t         st_uid;      /* идентификатор владельца */
  gid_t         st_gid;      /* идентификатор группы */
  dev_t         st_rdev;     /* тип устройства */
  off_t         st_size;     /* размер в байтах */
  unsigned long st_blksize;  /* эффективный размер блока */
  unsigned long st_blocks;   /* число занятых блоков */
  time_t        st_atime;    /* время последнего доступа */
  time_t        st_mtime;    /* время последней модификации */
  time_t        st_ctime;    /* время последнего изменения inode */
};
```

Поле `st_blocks` даёт размер файла в блоках по 512 байт. Поле `st_blksize` даёт предпочтительный размер блока для эффективных операций ввода-вывода с файловой системой.

Для проверки типа файла определены следующие макросы:

`S_ISLNK(m)`	символическая ссылка?
`S_ISREG(m)`	обычный файл?
`S_ISDIR(m)`	каталог?
`S_ISCHR(m)`	устройство символьного типа?
`S_ISBLK(m)`	устройство блочного типа?
`S_ISFIFO(m)`	fifo?
`S_ISSOCK(m)`	сокет?

Для поля `st_mode` определены следующие флаги:

`S_IFMT`	0170000	битовая маска для поля типа файла
`S_IFSOCK`	0140000	сокет
`S_IFLNK`	0120000	символическая ссылка
`S_IFREG`	0100000	обычный файл
`S_IFBLK`	0060000	устройство блочного типа
`S_IFDIR`	0040000	каталог
`S_IFCHR`	0020000	устройство символьного типа
`S_IFIFO`	0010000	fifo
`S_ISUID`	0004000	бит suid
`S_ISGID`	0002000	бит sgid
`S_ISVTX`	0001000	sticky bit
`S_IRWXU`	00700	маска для прав доступа владельца
`S_IRUSR`	00400	владелец имеет права чтения
`S_IWUSR`	00200	владелец имеет права записи
`S_IXUSR`	00100	владелец имеет права 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 для прав доступа прочих пользователей
`S_IROTH`	00004	прочие имеют права чтения
`S_IWOTH`	00002	прочие имеют права записи
`S_IXOTH`	00001	прочие имеют права выполнения

Бит sgid имеет несколько назначений: для каталогов он указывает, что в данном каталоге используется семантика создания файлов BSD, то есть файлы, создаваемые в этом каталоге, наследуют идентификатор группы от каталога, а не от идентификатора группы процесса, создающего файл, каталоги созданные в этом каталоге также будут иметь установленный бит sgid.

'sticky' bit (`S_ISVTX`) на каталоге означает, что файл в данном каталоге может быть переименован или удалён только владельцем файла, владельцем каталога или суперпользователем (root).

Возвращаемое значение

При успешном завершении возвращается 0. При ошибке возвращается `-1`, и переменная `errno` устанавливается в код ошибки.

Коды ошибок

EBADF	неверный файловый дескриптор filedes.
ENOENT	компонента имени каталога в пути не существует, либо висячая символическая ссылка.
ENOTDIR	компонента пути, используемая как каталог, не является каталогом.
ELOOP	слишком много символических ссылок при трансляции пути.
EFAULT	file_name представляет собой недопустимый адрес.
EACCES	один из каталогов в пути к файлу не позволяет поиск в нем.
ENOMEM	недостаточно памяти для ядра.
ENAMETOOLONG	путь слишком длинный.

8 access

access — проверить права доступа пользователя к файлу.

Использование

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

Описание

access проверяет, имеет ли процесс права на чтение или запись, либо существует ли файл, путь к которому указан в pathname. Если pathname — символическая ссылка, проверяются права доступа к файлу, на который указывает эта символическая ссылка.

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 проверку на существование файла и права на чтение, запись и выполнение соответственно. F_OK запрашивает проверку на существование файла.

Результат проверки зависит от прав доступа к каталогам, указанным в пути к файлу, и от прав доступа к каталогам и файлам, на которые ссылаются символические ссылки, использованные при разборе пути к файлу.

Проверки производятся с реальным идентификатором пользователя и группы процесса, а не с эффективными идентификаторами, в отличие от команд доступа к файлам. Это позволяет программам с битом suid проверять полномочия вызвавшего её пользователя.

Проверяются только биты доступа, но не содержимое файлов, то есть если каталог обнаружен как доступный на запись, это может означать, что в этом каталоге могут создаваться файлы, а не то, что каталог может быть записан как файл.

Возвращаемое значение

При успешном завершении (все запрошенные права могут быть предоставлены) возвращается 0. При ошибке (хотя бы одно из запрошенных прав не предоставлено, или в случае другой ошибки) возвращается -1, и errno устанавливается в код ошибки.

Коды ошибок

EACCES	запрошенный режим доступа не разрешён, либо один из каталогов в пути к файлу не позволяет поиск в нем.

EROFS	путь указывает на файл на файловой системе, доступной только на чтение, и режим записи был запрошен.
EFAULT	pathname представляет собой недопустимый адрес.
EINVAL	mode задан некорректно.
ENAMETOOLONG	путь слишком длинный.
ENOENT	компонента имени каталога в пути не существует, либо висячая символическая ссылка.
ENOTDIR	компонента пути, используемая как каталог, не является каталогом.
ENOMEM	недостаточно памяти для ядра.
ELOOP	слишком много символических ссылок при трансляции пути.
EIO	ошибка ввода/вывода.

9 link

link — создать новое имя для файла.

Использование

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

Описание

link создаёт новую связь (жёсткую связь) на существующий файл. Если имя newpath существует, оно переписывается.

Вновь созданное имя может использоваться для любой операции точно так же, как и старое. Оба имени ссылаются на тот же самый файл (имеют одинаковых владельцев и права доступа), после создания жёсткой связи невозможно отличить, какое имя было «оригинальным».

Жёсткие связи не могут пересекать границы файловых систем. Если это необходимо, используйте symlink.

Возвращаемое значение

При успешном завершении возвращается 0. При ошибке возвращается -1, и переменная errno устанавливается в код ошибки.

Коды ошибок

EXDEV	oldpath и newpath не располагаются на одной файловой системе
EPERM	файловая система не поддерживает создание жёстких связей; либо oldpath является каталогом
EFAULT	oldpath или newpath указывают за пределы адресного пространства процесса
EACCES	каталог, содержащий newpath не допускает запись в него; либо один из каталогов в oldpath не допускает поиск
ENAMETOOLONG	слишком длинный путь oldpath или newpath
ENOENT	компонента каталога в oldpath или newpath не существует, либо является «висящей» символической ссылкой
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым

ENOMEM	недостаточно памяти ядра
EROFS	файловая система работает в режиме «только чтение»
EMLINK	файл, на который указывает oldpath, уже имеет слишком много жёстких связей
ELOOP	слишком много символических связей было встречено при прослеживании пути newpath или oldpath
ENOSPC	на устройстве нет свободного места для новой записи в каталоге
EIO	ошибка ввода/вывода на устройстве

10 symlink

symlink — создать новое имя для файла.

Использование

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

Описание

symlink создаёт символическую связь с именем newpath, которая содержит строку oldpath.

Символические связи интерпретируются во время выполнения как если бы содержимое ссылки было подставлено в прослеживаемый путь для поиска файла или каталога.

Символические связи могут содержать компоненту пути .., которая (если использована в начале строки связи), обозначает родительский каталог того каталога, в котором находится символическая связь.

Символическая связь может указывать на существующий файл или каталог, а может — на несуществующий. Последний случай ещё называется «висячей ссылкой».

Права доступа у символической связи игнорируются, когда символическая связь прослеживается, но проверяются при переименовании или удалении в каталогах с установленным битом t.

Если newpath существует, он не переписывается. oldpath не проверяется на допустимость.

Удаление имени, на который указывает символическая связь, ведёт к действительному удалению файла (если только этот файл не имел ещё жёстких связей). Если такое поведение нежелательно, нужно использовать link.

Возвращаемое значение

При успешном завершении возвращается 0. При ошибке возвращается -1 и переменная errno устанавливается в код ошибки.

Коды ошибок

EPERM	файловая система не поддерживает создание жёстких связей
EFAULT	oldpath или newpath указывают за пределы адресного пространства процесса
EACCES	каталог, содержащий newpath не допускает запись в него; либо один из каталогов в oldpath не допускает поиск

ENAMETOOLONG	слишком длинный путь oldpath или newpath
ENOENT	компонента каталога в oldpath или newpath не существует, либо является «висящей» символической ссылкой
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым
ENOMEM	недостаточно памяти ядра
EROFS	файловая система работает в режиме «только чтение»
EEXIST	newpath уже существует
ELOOP	слишком много символических связей было встречено при прослеживании пути newpath или oldpath
ENOSPC	на устройстве нет свободного места для новой записи в каталоге
EIO	ошибка ввода/вывода на устройстве

11 readlink

readlink — считать значение символической связи.

Использование

```
#include <unistd.h>
int readlink(const char *path, char *buf, size_t bufsiz);
```

Описание

readlink помещает содержимое символической связи path в буфер buf, который имеет размер bufsiz. readlink не добавляет символ '\0' в конец строки. Содержимое символической связи ограничивается bufsiz символами, если размер буфера недостаточен для размещения всей связи.

Возвращаемое значение

Системный вызов возвращает число символов, записанное в буфер в случае успешного завершения. В случае ошибки возвращается -1 и переменная errno устанавливается в код ошибки.

Коды ошибок

ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым
EINVAL	bufsiz неположителен; либо path не является символической связью
ENAMETOOLONG	слишком длинный путь path
ENOENT	компонента каталога в path не существует, либо является «висящей» символической ссылкой; символическая связь path не существует
EACCES	один из каталогов в path не допускает поиск
ELOOP	слишком много символических связей было встречено при прослеживании пути path
EIO	ошибка ввода/вывода на устройстве
EFAULT	path или buf указывают за пределы адресного пространства процесса
ENOMEM	недостаточно памяти ядра

12 unlink

unlink — уничтожить имя и, возможно, файл, на который оно указывает.

Использование

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

Описание

unlink уничтожает имя в файловой системе. Если это имя было последним, которое ссылалось на файл, и файл не был открыт никаким процессом, файл уничтожается и место на диске, которое он занимал, становится доступным для повторного использования.

Если имя pathname было последней ссылкой на файл, но какие-либо процессы держат этот файл открытым, файл продолжает существование до момента, когда будет закрыт последний файловый дескриптор, ссылающийся на файл.

Если имя pathname является именем символической связи, уничтожается сама символическая связь, а не файл, на который она указывает.

Возвращаемое значение

При успешном завершении возвращается 0. В случае ошибки возвращается -1, и переменная errno устанавливается в код ошибки.

Коды ошибок

EFAULT	path или buf указывают за пределы адресного пространства процесса
EACCES	каталог, содержащий pathname не допускает запись в него; либо один из каталогов не допускает поиск
EPERM	у каталога, содержащего удаляемый файл, установлен бит t, и uid процесса не совпадает с uid удаляемого файла или родительского каталога; pathname является каталогом
ENAMETOOLONG	слишком длинный путь pathname
ENOENT	компонента каталога в pathname не существует, либо является «висящей» символической ссылкой
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым
EISDIR	pathname является каталогом
ENOMEM	недостаточно памяти ядра
EROFS	pathname находится в файловой системе, которая работает в режиме «только чтение»
ELOOP	слишком много символических связей было встречено при прослеживании пути path
EIO	ошибка ввода/вывода на устройстве

13 rename

rename — изменить имя или расположение файла.

Использование

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

Описание

rename переименовывает файл, перемещая его между каталогами, если необходимо. Другие жёсткие связи (созданные с помощью link) не затрагиваются.

Если newpath уже существует, он будет атомарно заменён так, что для другого процесса, который работает с newpath, он всегда будет существовать. Однако может существовать промежуток времени, в течение которого oldpath и newpath будут указывать на один и тот же файл.

Если newpath существует, но операция по какой-либо причине завершается неудачно, rename гарантирует, что newpath останется неизменным.

Если oldpath является символической связью, связь переименовывается, а если newpath является символической связью, она уничтожается.

Возвращаемое значение

При успешном завершении операции возвращается 0. При ошибке возвращается -1, и переменная errno устанавливается в код ошибки.

Коды ошибок

EISDIR	newpath существует и является каталогом, а oldpath не является каталогом
EXDEV	oldpath и newpath не находятся на одной файловой системе
ENOTEMPTY, EEXIST	newpath — непустой каталог, то есть содержит записи, отличные от . и ..
EBUSY	oldpath или newpath используются другим процессом или операционной системой таким образом, что делает операцию невозможной
EINVAL	попытка сделать каталог подкаталогом самого себя
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым; либо oldpath является каталогом, а newpath существует и не является каталогом
EFAULT	oldpath или newpath указывают за пределы адресного пространства процесса
EACCES	закрыт доступ на запись к необходимым каталогам; каталог в пути oldpath или newpath не позволяют вести поиск в нем; файловая система не допускает такую операцию
ENAMETOOLONG	слишком длинный путь oldpath или newpath
ENOENT	компонента каталога существует, либо является «висящей» символической ссылкой
ENOMEM	недостаточно памяти ядра
EROFS	pathname находится в файловой системе, которая работает в режиме «только чтение»
ELOOP	слишком много символических связей было встречено при прослеживании пути path
ENOSPC	на устройстве не осталось места для новой записи в каталоге

14 mkdir

`mkdir` — создать каталог.

Использование

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

Описание

`mkdir` создаёт новый каталог с именем `pathname`.

`mode` задаёт права доступа к создаваемому каталогу. Параметр `mode` модифицируется значением `umask` процесса стандартным способом (`mode & ~umask`).

Вновь созданный каталог будет иметь такого владельца, каков euid процесса. Каталог унаследует идентификатор группы от родительского каталога.

Возвращаемое значение

При успешном завершении `mkdir` возвращает 0. При ошибке возвращается -1 и переменная `errno` устанавливается в код ошибки.

Коды ошибок

EEXIST	`pathname` уже существует (не обязательно каталог)
EFAULT	`pathname` указывает за пределы адресного пространства процесса
EACCES	каталог, содержащий `pathname` не допускает запись в него; либо один из каталогов не допускает поиск
ENAMETOOLONG	слишком длинный путь `pathname`
ENOENT	компонента каталога в `pathname` не существует, либо является «висящей» символической ссылкой
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым
ENOMEM	недостаточно памяти ядра
EROFS	`pathname` находится в файловой системе, которая работает в режиме «только чтение»
ELOOP	слишком много символических связей было встречено при прослеживании пути `path`
ENOSPC	на устройстве не осталось свободного места; либо исчерпана квота пользователя

15 rmdir

`rmdir` — удалить каталог.

Использование

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

Описание

rmdir удаляет каталог, который должен быть пустым.

Возвращаемое значение

При успешном завершении возвращается 0. При ошибке возвращается -1, и переменная errno устанавливается в код ошибки.

Коды ошибок

EPERM	файловая система не поддерживает удаление каталогов; у каталога, содержащего удаляемый файл, установлен бит t, и uid процесса не совпадает с uid удаляемого файла или родительского каталога;
EFAULT	pathname указывает за пределы адресного пространства процесса
EACCES	каталог, содержащий pathname не допускает запись в него; либо один из каталогов не допускает поиск
ENAMETOOLONG	слишком длинный путь pathname
ENOENT	компонента каталога в pathname не существует, либо является «висящей» символической ссылкой
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым
ENOTEMPTY	каталог содержит записи, отличные от . и ..
EBUSY	каталог является текущим или корневым каталогом некоторого процесса
ENOMEM	недостаточно памяти ядра
EROFS	pathname находится в файловой системе, которая работает в режиме «только чтение»
ELOOP	слишком много символических связей было встречено при прослеживании пути path

16 chdir, fchdir

chdir, fchdir — сменить текущий каталог.

Использование

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

Описание

chdir устанавливает заданный параметром каталог path как текущий. fchdir устанавливает текущим каталог, заданный открытым файловым дескриптором.

Возвращаемое значение

При успешном завершении возвращается 0. При ошибке возвращается -1, и переменная errno устанавливается в код ошибки.

Коды ошибок

Коды ошибок зависят от файловой системы. Наиболее общие приведены ниже.

EFAULT	path указывает за пределы адресного пространства процесса
ENAMETOOLONG	слишком длинный путь path
ENOENT	каталог path не существует
ENOMEM	недостаточно памяти ядра
ENOTDIR	компонента пути, заявленная в пути как каталог, на самом деле не является таковым
EACCES	один из каталогов-компонент пути не допускает поиск в нем
ELOOP	слишком много символических связей было встречено при прослеживании пути path
EBADF	fd не является допустимым файловым дескриптором

17 getcwd

getcwd — получить имя текущего каталога.

Использование

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Описание

getcwd копирует абсолютное имя текущего рабочего каталога в массив, на который указывает buf, размера size. Если имя текущего каталога требует буфера размера большего, чем size, возвращается NULL и errno устанавливается в ERANGE.

Возвращаемое значение

При успешном завершении возвращается buf. При ошибке возвращается NULL, и переменная errno устанавливается в код ошибки.

18 opendir

opendir — открыть каталог на чтение.

Использование

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

Описание

Функция `opendir` открывает каталог с заданным именем и возвращает указатель на структуру `DIR`, используемую для доступа к элементам каталога. Указатель текущего положения позиционируется на начало файла.

Возвращаемое значение

Указатель на структуру `DIR` при успешном завершении, или `NULL` при ошибке, тогда `errno` устанавливается в код ошибки.

Коды ошибок

EACCES	каталог закрыт на чтение, либо один из каталогов в пути к файлу не позволяет поиск в нем.
EMFILE	процесс уже открыл максимальное количество файлов.
ENFILE	достигнут лимит на число открытых файлов в системе.
ENOENT	компонента имени каталога в пути не существует, либо висячая символическая ссылка, либо пустое имя каталога.
ENOMEM	недостаточно памяти для ядра.
ENOTDIR	компонента пути, используемая как каталог, не является каталогом.

19 readdir

`readdir` — читать из каталога.

Использование

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Описание

Функция `readdir` возвращает указатель на структуру `dirent`, которая содержит информацию о следующей записи в каталоге, доступ к которому производится по указателю `dir`. Функция возвращает `NULL` при ошибке или конце каталога.

Данные, возвращённые `readdir`, будут переписаны последующими вызовами `readdir` с тем же указателем `dir`.

Структура `struct dirent` содержит поле `char d_name[]`, в котором хранится текущее имя в каталоге, заканчивающееся нулевым символом. Структура может содержать поле `ino_t d_ino`, хранящее номер индексного дескриптора, но пользоваться им нельзя, поскольку оно хранит неправильное значение в точках монтирования.

Возвращаемое значение

При успешном завершении `readdir` возвращает указатель на структуру `dirent`. В случае ошибки или конца файла возвращается NULL.

Коды ошибок

EBADF неверный указатель dir.

20 telldir

telldir — получить текущую позицию чтения в каталоге.

Использование

```
#include <dirent.h>
off_t telldir(DIR *dir);
```

Описание

Функция telldir возвращает текущую позицию в потоке чтения каталога dir.

Возвращаемое значение

При успешном завершении возвращается текущая позиция в потоке чтения каталога. При ошибке возвращается -1, и переменная errno устанавливается в код ошибки.

Коды ошибок

EBADF Неверный дескриптор потока каталога dir.

21 seekdir

seekdir — установить позицию чтения из каталога для последующего вызова readdir.

Использование

```
#include <dirent.h>
void seekdir(DIR *dir, off_t offset);
```

Описание

Функция seekdir устанавливает положение указателя чтения из дескриптора каталога dir. Следующий вызов readdir считает очередную запись, начиная с установленной позиции. Значение offset должно быть получено вызовом telldir или быть равным 0.

Возвращаемое значение

Функция не возвращает значения.

22 closedir

closedir — закрыть каталог.

Использование

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir);
```

Описание

Функция closedir закрывает каталог, ассоциированный с dir. Этот указатель dir после закрытия использовать нельзя.

Возвращаемое значение

При успешном завершении возвращается 0, в случае ошибки возвращается -1.

Коды ошибок

EBADF неверный указатель dir.

23 chmod, fchmod

chmod, fchmod — изменить права доступа к файлу.

Использование

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

Описание

Системные вызовы изменяют права доступа к файлу по заданному пути или файлу, на который ссылается открытый файловый дескриптор.

Права доступа определяются побитовым объединением следующих флагов.

S_ISUID	04000	setuid бит
S_ISGID	02000	setgid бит
S_ISVTX	01000	‘sticky’ бит
S_IRUSR (S_IREAD)	00400	чтение владельцем
S_IWUSR (S_IWRITE)	00200	запись владельцем
S_IXUSR (S_IEXEC)	00100	выполнение/поиск владельцем
S_IRGRP	00040	чтение группой
S_IWGRP	00020	запись группой
S_IXGRP	00010	выполнение/поиск группой
S_IROTH	00004	чтение остальными
S_IWOTH	00002	запись остальными
S_IXOTH	00001	выполнение/поиск остальными

Эффективный идентификатор процесса должен быть нулевым или совпадать с идентификатором владельца файла.

В файловой системе NFS ограничение прав доступа немедленно повлияет на уже открытые файлы, потому что контроль доступа производится сервером, а открытые файлы поддерживаются клиентом. Расширение прав доступа может быть отложено для клиентов, у которых включено кеширование атрибутов.

Возвращаемое значение

В случае успешного завершения возвращается 0. При ошибке возвращается -1 и переменная `errno` устанавливается в код ошибки.

Коды ошибок

В зависимости от файловой системы могут возвращаться другие ошибки. Наиболее общие ошибки приведены ниже.

`EPERM`	Эффективный идентификатор процесса не совпадает с идентификатором владельца файла и не равен 0.
`EROFS`	Указанный файл находится на файловой системе, доступной только на чтение.
`EFAULT`	`path` — неверный адрес.
`ENAMETOOLONG`	`path` слишком длинный.
`ENOENT`	Файл не существует.
`ENOMEM`	Недостаточно памяти ядра, чтобы выполнить операцию.
`ENOTDIR`	Компонент пути не является каталогом.
`EACCES`	Компонент пути не допускает поиск в нем.
`ELOOP`	Слишком много символических ссылок.
`EIO`	Ошибка ввода/вывода.
`EBADF`	Недопустимый файловый дескриптор `fildes`.

24 utime

`utime` — изменить времена последнего доступа и/или модификации индексного дескриптора.

Использование

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

Описание

`utime` изменяет времена доступа и модификации индексного дескриптора, заданного с помощью `filename`, на значения полей `actime` и `modtime` соответственно. Если `buf` равен `NULL`, время доступа и модификации устанавливается в текущее время. Структура `utimbuf` определена следующим образом.

```
struct utimbuf {
        time_t actime;  /* access time */
        time_t modtime; /* modification time */
};
```

Возвращаемое значение

В случае успешного выполнения возвращается 0. При ошибке возвращается -1, и переменная `errno` устанавливается в код ошибки.

Коды ошибок

Могут возвращаться другие коды ошибок.

`EACCES` Нет прав записи в файл.
`ENOENT` Файл `filename` не существует.

25 umask

`umask` — установить маску создания файлов.

Использование

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

Описание

`umask` устанавливает маску создания файлов `umask` в значение `mask & 0777`.

Значение `umask` используется системными вызовами open(2), creat(2), mkdir(2) для установки прав доступа ко вновь создаваемому файлу. Более точно, права доступа, установленные в `umask`, сбрасываются в аргументе `mode` системного вызова open(2). Если, например, значение `umask` равно `022` (часто используемое значение по умолчанию), а новый файл создаётся с правами 0666, файл будет создан с правами `0666 & ~022` = `0644` = `rw-r--r--`.

Возвращаемое значение

Системный вызов всегда завершается успешно и возвращает предыдущее значение маски создания файлов.

26 truncate, ftruncate

`truncate`, `ftruncate` — обрезать файл на заданную длину.

Использование

```
#include <unistd.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Описание

Эти системные вызовы обрезают файл, именованный с помощью `path`, или на который ссылается `fd` до длины `length`. Если файл был изначально короче, его длина не изменяется. Если файл был длиннее, остаток файла теряется. При использовании `ftruncate` файл должен быть открыт на запись.

Возвращаемое значение

В случае успешного завершения возвращается 0. При ошибке возвращается -1 и переменная `errno` устанавливается в код ошибки.

Коды ошибок

ENOTDIR	Компонент пути не является каталогом.
ENAMETOOLONG	`path` слишком длинный.
ENOENT	Файл не существует.
EACCES	Компонент пути не допускает поиск в нем.
EACCES	Указанный файл недоступен на запись.
EISDIR	Указанный файл является каталогом.
ELOOP	Слишком много символических ссылок.
EIO	Ошибка ввода/вывода.
EROFS	Указанный файл находится на файловой системе, доступной только на чтение.
EFAULT	`path` — неверный адрес.
EBADF	Недопустимый файловый дескриптор `fd`.
EINVAL	Файловый дескриптор `fd` ссылается не на файл, или файл не открыт на запись.
ETXTBSY	Данный файл является программой, которая в данный момент исполняется.