Проектирование системного программного обеспечения

Учебно пособие

23.02.08

Лекция 2
Файловый ввод-вывод UNIX

Оглавление

Общие концепции файлового ввода-вывода....2
Открытие файла....3
Закрытие файла....3
Чтение, запись и установка позиции в файле....3
Файловый ввод-вывод в *nix....4
Виртуальная файловая система....4
Виды файловых систем....5
Монтирование, точка монтирования....6
Реализация VFS в Linux....6
Справочник по функциям файлового ввода-вывода Linux....7
Системный вызов open....8
Системный вызов creat....12
Системный вызов close....12
Системный вызов read....13
Системный вызов write....14
Системный вызов lseek....16
Системные вызовы pread и pwrite....17
Системные вызовы dup и dup2....18
Системный вызов sync....19
Системный вызов fsync....20
Системный вызов fdatasync....21
Системный вызов fcntl....22
Системный вызов ioctl....29
Литература....31

Общие концепции файлового ввода-вывода

В данной главе мы попытаемся рассмотреть файловый ввод-вывод без привязки не только к какой-либо операционной системе, но даже без привязки к программной или аппаратной реализации.

Итак, как нам известно - файл является упорядоченной последовательностью байтов имеющей имя. Имя нужно для того, чтобы можно было однозначно выделить нужную информацию из общего информационного массива. Способ отделения данных друг от друга, способ хранения данных и их защиты в данном случае совершенно не важны. Главное то, что зная имя мы можем получить нужную информацию.

С файлами можно осуществлять сравнительно небольшое количество базовых операций:

- Открытие;
- Чтение;
- Запись;
- Перемещение указателя внутри файла;
- Закрытие.

Концепция файлового ввода-вывода универсальна. В виде файла может быть представлены информация находящаяся на каком либо носителе информации, устройство ввода-вывода, очереди (каналы связи) между процессами, сокеты и т.п. Общая идея файлового ввода-вывода базируется на идее сетей процессов.

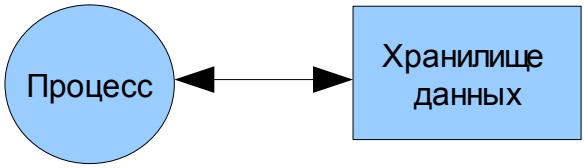

Рисунок 2: Обмен процесс-память

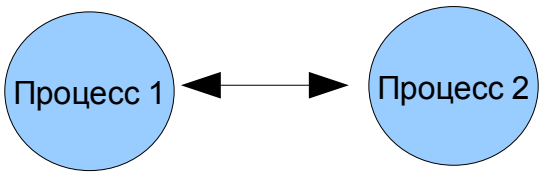

Рисунок 1: Обмен процесс-процесс

Существует два варианта организации обмена:

1. Процесс — устройство памяти (см. рис. 1).
2. Процесс — процесс (см. рис. 2);

Чем отличается процесс от хранилища данных?

Процесс — последовательная смена состояний объекта во времени. Процесс активен, он может делать что-то сам по себе. Хранилище данных пассивно и само по себе оно не меняется никогда. Для изменения состояния хранилища данных нужен внешний процесс.

В настоящее время, большинство файловых обменов можно рассматривать как взаимодействие двух и более процессов и только в некоторых случаях — как взаимодействие процессов и хранилищ данных.

Открытие файла

Под открытием файла понимается создание канала связи между процессами или между процессом и пассивным хранилищем данных. При создании канала связи происходит создание и инициализация инфраструктуры, необходимой для работы с файлом.

В случае файлового обмена между процессами возможно две ситуации:

1. Процесс с противоположной стороны готов к связи;
2. Процесс с противоположной стороны не готов.

В большинстве систем, как правило, выделяют специальный процесс, отвечающий за некий ресурс. Такой процесс называют сервером. Сервер готов к связи, но в отсутствии клиентов он не может создать канала. Канал связи появляется только при открытии файла со стороны клиента.

При открытии канала указывается имя файла. После открытия мы получаем информацию, необходимую и достаточную для доступа к каналу связи именно к нашему файлу. Обычно, эта информация называется дескриптором файла или описателем.

Итак, после успешного открытия файла мы получаем открытый канал связи между нашим процессом и другим процессом или пассивным хранилищем данных.

Открытие файла может занимать существенное время в жизни вычислительной системы. Чтобы система не простаивала, открывающий процесс может быть заблокирован на время открытия канала связи. Возможен и другой вариант. Процесс посылает команду открытия файла и занимается своими делами. Информация о том, что канал успешно открыт или произошла какая либо ошибка может быть доставлена позже.

Закрытие файла

При закрытии файла происходит завершение процесса передачи данных и уничтожение инфраструктуры канала связи. Необходимо понимать, что завершение передачи данных по каналу связи может продолжаться существенное время. Возможно две ситуации:

1. Завершение операции закрытия происходит до завершения передачи всех данных до конечной части цепочки. То есть мы со своей стороны канал закрыли, но данные все еще в пути.
2. Завершение операции после окончания передачи всех данных.

В первом случае мы не знаем, закончилась ли файловая операция успешно. Во втором случае мы вынуждены ждать существенное время.

Чтение, запись и установка позиции в файле

После создания канала связи, файл представляется как непрерывное адресное пространство допускающее три операции:

1. Чтение данных;

2. Запись данных;
3. Установка адреса для последующих операций считывания и записи.

Из всех трех операций над файлом, операция установки адреса обычно самая быстрая. Запись и чтение напротив, могут занимать существенное время. Как обычно, для длительных операций возможно два варианта действий:

1. Блокировка процесса на время выполнения действий;
2. Быстрое завершение операции до её окончания. В дальнейшем, информация о завершении может быть получена путем опроса флагов готовности или по событию.

Если противоположная сторона по какой-либо причине закрыла канал, заблокированный процесс может остаться таковым навсегда. Поэтому необходимо предусмотреть механизм снятия блокировки по какому-либо критерию, например, по тайм-ауту.

Файловый ввод-вывод в *nix

Виртуальная файловая система

В последнее время, возникло достаточно большое количество файловых систем, таких как Ext2, Ext3, NTFS, FAT32, FAT16, NFS и т.д. Для поддержки всего этого многообразия понадобилось бы реализовывать большое количество разнообразных API для работы с каждой из систем. В качестве примера системы, в которой количество API примерно равнялось количеству файловых систем можно привести MS-DOS.

Для упрощения работы с разнообразными файловыми системами была придуман специальный слой в операционных системах, который называют виртуальной файловой системой (Virtual Fils System, далее VFS) или виртуальным файловым коммутатором.

Прикладные процессы используют унифицированный интерфейс для взаимодействия с VFS. Внутри VFS происходит передача параметров функциям, отвечающим за конкретную реализацию файловой системы.

Впервые, концепция VFS была реализована в SunOS фирмой Sun Microsystems в 1985 году. Это нововведение позволяло обращаться к локальной системе UFS и сетевой системе NFS совершенно единообразно. В последствии VFS была реализована в Unix System V Relese 4.

Итак, какие цели преследовались при создании SunOS VFS в 1985 году?

1. Обеспечение разделения зависимых от конкретных файловых систем и независимых частей операционной системы.
2. Должен поддерживаться единый UNIX' образный интерфейс, единый для

доступа к разделам SunOS, NFS, BSD или MS-DOS.

3. Интерфейс должен быть доступным для серверной части удаленной файловой системы для удовлетворения клиентских запросов.
4. Все системные операции должны быть атомарными.

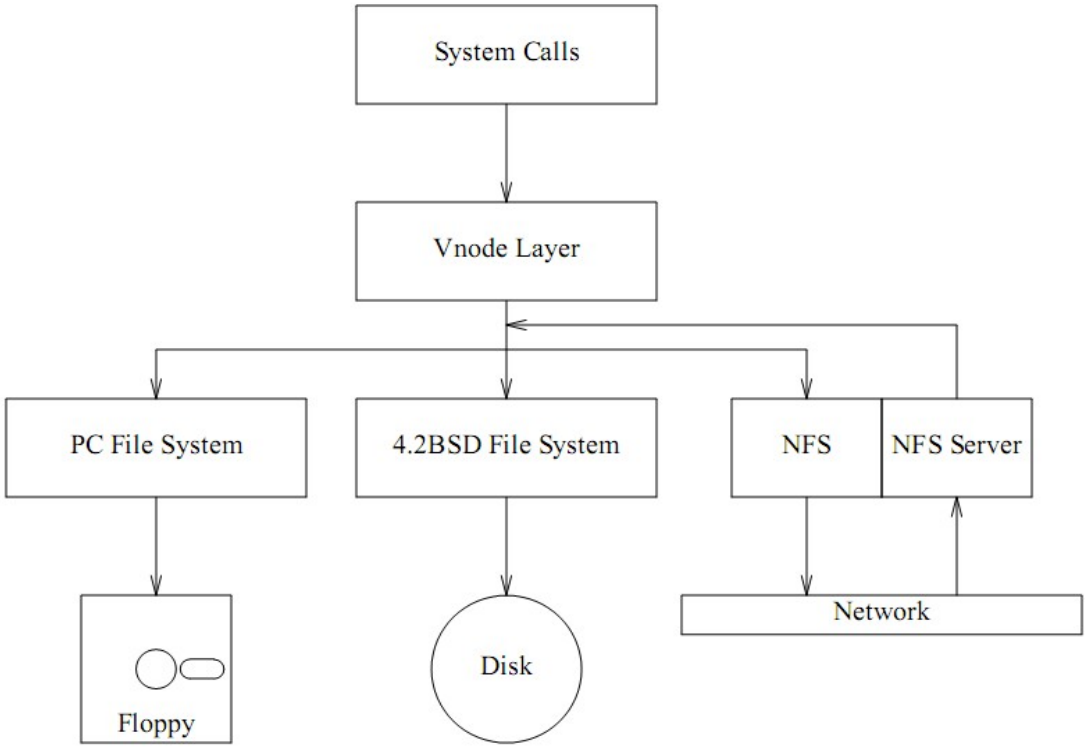

Рисунок 3: Структура VFS SunOS

Виды файловых систем

- Дисковые файловые системы поддерживают структуру файлов на гибких и жестких дисках, USB FLASH накопителях, CD-ROM и DVD.
- Сетевые файловые системы поддерживают прозрачный доступ к файлам находящимся на других машинах, подключенных к компьютерной сети.
- Специальные файловые системы реализуют файловый доступ к различным процессам, которым обычно не свойственен такой способ доступа. Пример специальной файловой системы — каталог /proc в UNIX.

Монтирование, точка монтирования

Монтирование файловой системы представляет собой процесс подключения какой либо частной файловой системы к виртуальной файловой системе. Каталог, в которой смонтирована файловая система называется точкой монтирования. Таким образом, благодаря наличию VFS с унифицированным интерфейсом доступа к файлам мы можем оперировать с различными файловыми системами находящимися в разных каталогах совершенно сходным образом.

Реализация VFS в Linux

Inode (индексный дескриптор) является базовым объектом в файловой системе Linux. Он может быть регулярным файлом, директорией, символической ссылкой или чем либо еще. VFS не делает различий между различными типами объектов. Каждый **inode** представлен структурой *inode*, каждый метод хранится в структуре *inode_operation*.

```
struct inode {
        struct list_head        i_hash;
        struct list_head        i_list;
        struct list_head        i_dentry;

        unsigned long           i_ino;
        unsigned int            i_count;
        kdev_t                  i_dev;
        umode_t                 i_mode;
        nlink_t                 i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        kdev_t                  i_rdev;
        off_t                   i_size;
        time_t                  i_atime;
        time_t                  i_mtime;
        time_t                  i_ctime;
        unsigned long           i_blksize;
        unsigned long           i_blocks;
        unsigned long           i_version;
        unsigned long           i_nrpages;
        struct semaphore        i_sem;
        struct inode_operations *i_op;
        struct super_block      *i_sb;
        wait_queue_head_t       i_wait;
        struct file_lock        *i_flock;
        struct vm_area_struct   *i_mmap;
        struct page             *i_pages;
        spinlock_t              i_shared_lock;
        struct dquot            *i_dquot[MAXQUOTAS];
        struct pipe_inode_info  *i_pipe;


        unsigned long           i_state;

        unsigned int            i_flags;
        unsigned char           i_sock;

        atomic_t                i_writecount;
        unsigned int            i_attr_flags;
        __u32                   i_generation;
        union {
                ....
                struct ext2_inode_info          ext2_i;
                ....
                struct socket                   socket_i;
                void                            *generic_ip;
        } u;
};
```

Файловая система является множеством структур **inode**. Файловая система имеет параметры, общие для всех indode в нее входящих. Например, такими параметрами могут быть флаг *READ_ONLY* и переменная *buffer_size*. Каждая файловая система представлена структурой *super_block*, методы которой содержатся в структуре *super_operations*. Каждая файловая система строго соответствует определенному устройству, с определенным номером (например, /dev/fd0).

Методы входящие в super_block реализуют поведение, специфическое для конкретной файловой системы.

Каждой примонтированной файловой системе соответствует структура super_block.

```
struct vfsmount
{
  kdev_t mnt_dev;                       /* Device this applies to */
  char *mnt_devname;                    /* Name of device e.g. /dev/dsk/hda1 */
  char *mnt_dirname;                    /* Name of directory mounted on */
  unsigned int mnt_flags;               /* Flags of this device */
  struct super_block *mnt_sb;           /* pointer to superblock */
  struct quota_mount_options mnt_dquot; /* Diskquota specific mount options */
  struct vfsmount *mnt_next;            /* pointer to next in linkedlist */
};
```

Структура super_block имеет вид:

```
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        kdev_t                  s_dev;
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_lock;
        unsigned char           s_dirt;
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct dquot_operations *dq_op;
        unsigned long           s_flags;
        unsigned long           s_magic;
        struct dentry           *s_root;
        wait_queue_head_t       s_wait;

        struct inode            *s_ibasket;
        short int               s_ibasket_count;
        short int               s_ibasket_max;
        struct list_head        s_dirty;        /* dirty inodes */
        struct list_head        s_files;

        union {
                /* Configured-in filesystems get entries here */
                void                    *generic_sbp;
        } u;
        /*
         * The next field is for VFS *only*. No filesystems have any business
         * even looking at it. You had been warned.
         */
        struct semaphore s_vfs_rename_sem;      /* Kludge */
};
```

Справочник по функциям файлового ввода-вывода Linux

Материал данной главы можно легко найти в справочной системе Unix. Параметры функций API могут немного меняться, в зависимости от версии системы, которую вы используете. Данные примеры описания функций API даются только с целью общего ознакомления. Для получения наиболее точной справки по какой либо функции в вашей системе вам необходимо набрать следующую команду:

man 2 имя_функции

Полезным может оказаться также небольшое исследование заголовочных файлов для уточнения точного набора параметров той или иной функции.

Таблица 1 Перечень основных функций файлового ввод-вывода

Имя	Назначение
open	Открыть или создать файл
creat	Создать файл
close	Закрыть файловый дескриптор
read	Прочитать данные из файла
write	Записать данные в файл
lseek	Установить смещение для позиционирования операций чтения/записи
pread, pwrite	читает из файлового дескриптора или пишет в него по заданному смещению
dup, dup2	Создать дубликат файлового дескриптора
sync	Записывает буферный кэш на диск
fsync	Синхронизирует состояние файла в памяти с состоянием на диске
fdatasync	Синхронизировать содержимое файла в памяти с содержимым на диске
fcntl	Манипуляции с файловым дескриптором
ioctl	Управление устройствами

Системный вызов open

```
#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 );
```

Системный вызов open превращает имя файла в дескриптор файла (небольшое неотрицательное число, используемое при последующем вводе-выводе, например, с **read** , **write** , и т. п. Если системный вызов завершается успешно, возвращенный файловый дескриптор является самым маленьким дескриптором, который еще не открыт процессом. В результате этого вызова появляется новый открытый файл, не разделяемый ни с каким процессом (разделяемые открытые файлы могут возникнуть в результате системного вызова **fork** (2). Новый файловый дескриптор будет оставаться открытым при выполнении функции **exec** (2) (смотри описание **fcntl** (2)). Указатель в файле устанавливается в начало.

Параметр flags "\" это **O_RDONLY** , **O_WRONLY** или **O_RDWR** , задающие, соответственно, открытие файла только для чтения, только для записи и для чтения и записи, которые можно комбинировать с помощью логического ИЛИ с нулем или более нижеследующих флагов:

- ***O_CREAT*** Если файл не существует, то он будет создан. Владелец (uid) файла устанавливается в фактический идентификатор владельца процесса. Группа (gid) устанавливается либо в фактический идентификатор группы процесса или же в идентификатор группы родительского каталога (зависит от типа файловой системы и флагов программы mount, а также прав доступа к родительскому каталогу. См., например, флаги bsdgroups и sysvgroups в файловой системе ext2, которые описаны в mount (8)). **O_EXCL** Если используется совместно с **O_CREAT** , то если файл уже существует, то open завершится с ошибкой. В этом контексте символьная ссылка существует, независимо от того, куда она указывает. **O_EXCL** не работает на файловых системах NFS, а в программах, полагающихся на этот флаг для реализации блокировки, возникнет race condition. Решение для атомарной блокировки с помощью файла: создать файл с уникальным именем на той же самой файловой системе (это имя может содержать, например, имя машины и идентификатор процесса), использовать **link** (2), чтобы создать ссылку на файл блокировки. Если **link**() возвращает 0, значит, блокировка была успешной. В противном случае используйте **stat** (2), чтобы убедиться, что количество ссылок на уникальный файл возросло до двух, что опять означает, что блокировка была успешной.
- ***O_NOCTTY*** Если pathname ссылается на терминальное устройство (см. tty (4)), то оно не станет контролирующим терминалом процесса, даже если у этого процесса нет такового.

- ***O_TRUNC*** Если файл уже существует, является обычным файлом, а режим открытия позволяет запись (то бишь это **O_RDWR** или **O_WRONLY**), то он будет обрезан до нулевой длины. Если файл является FIFO или устройством терминала, то флаг **O_TRUNC** игнорируется. В противном случае действие **O_TRUNC** не описано. (На многих версиях Linux этот флаг будет игнорирован; на других версиях будет возвращена ошибка.)
- ***O_APPEND*** Файл открывается в режиме добавления. Перед каждым write , файловый указатель перемещается в конец файла, как если бы использовался lseek . **O_APPEND** может привести к повреждению файлов на файловой системе NFS, если несколько процессов одновременно добавляют данные в один файл. Это происходит из-за того, что NFS не поддерживает добавление в файл, поэтому ядро на машине-клиенте должно эмулировать эту поддержку, что не может быть выполнено без race condition.
- ***O_NONBLOCK*** или ***O_NDELAY*** Если возможно, то файл открывается в неблокирующем режиме. Ни open , ни другие последующие операции над возвращенным дескриптором файла не заставят вызывающий процесс ждать. Работа с каналами FIFO также описана в **fifo** (4). Этот режим не обязан как-либо действовать на файлы, не являющиеся FIFO.
- ***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 оборудованием, на котором находится файл. Однако же, смотри в разделе ОГРАНИЧЕНИЯ. **O_NOFOLLOW** Если pathname -- это символическая ссылка, то open завершается с кодом ошибки. Это расширение из FreeBSD, которое было добавлено в Linux в версии 2.1.126. Все прочие символические ссылки в имени будут обработаны как обычно. Заголовочные файлы из glibc 2.0.100 и позже содержат определение этого флага; ядра с версиями меньше, чем 2.1.126, игнорируют этот флаг.
- ***O_DIRECTORY*** Если pathname не является каталогом, то open завершится с кодом ошибки. Этот флаг используется только в Linux, и был добавлен в ядре 2.1.126, чтобы избежать проблем с атаками типа "отказ обслуживания", если **opendir** (2) был вызван с каналом FIFO или ленточным устройством. Этот флаг не следует использовать вне реализации opendir . **O_LARGEFILE** На 32-битных системах, поддерживающих Большие Файловые Системы, этот флаг позволяет открывать файлы, длина которых не помещается в 31 бит.
- Некоторые из вышеописанных флагов могут быть изменены после открытия файла с помощью **fctnl** . Аргумент mode задает права доступа, которые используются в случае создания нового файла. Они модифицируются обычным способом, используя umask процесса: права доступа свежесозданного файла равны (mode & ~umask) . Заметьте, что этот режим доступа относится только к последующим обращениям к свежесозданному файлу; системный вызов open , который создает новый файл только для чтения, вполне может вернуть файловый дескриптор для чтения/записи.

Нижеследующие символические константы можно использовать в mode:

- ***S_IRWXU*** 00700 пользователь (владелец файла) имеет права чтения, записи и выполнения.

- ***S_IRUSR*** (S_IREAD) 00400 пользователь имеет право чтения
- ***S_IWUSR*** (S_IWRITE) 00200 пользователь имеет право записи
- ***S_IXUSR*** (S_IEXEC) 00100 пользователь имеет право выполнения
- ***S_IRWXG*** 00070 группа имеет права чтения, записи и выполнения
- ***S_IRGRP*** 00040 группа имеет право чтения
- ***S_IWGRP*** 00020 группа имеет право записи
- ***S_IXGRP*** 00010 группа имеет право выполнения
- ***S_IRWXO*** 00007 все остальные имеют права чтения, записи и выполнения
- ***S_IROTH*** 00004 все остальные имеют право чтения
- ***S_IWOTH*** 00002 все остальные имеют право записи
- ***S_IXOTH*** 00001 все остальные имеют право выполнения

mode всегда должен быть указан при использовании O_CREAT , во всех остальных случаях этот параметр игнорируется.

- creat эквивалентен open с flags , равными **O_CREAT** | **O_WRONLY** | **O_TRUNC** .

Возвращаемое значение

open и creat возвращают новый дескриптор файла, или -1, если произошла ошибка (в этом случае errno устанавливается должным образом). Заметьте, что open может открывать файлы устройств, но creat не может создавать их -- используйте для этого **mknod** (2).

На файловых системах NFS с включенным преобразованием идентификаторов пользователей open может вернуть файловый дескриптор, но, например, **read** (2) вернет ошибку EACCES . Это из-за того, что клиент выполняет open , проверяя права доступа, но преобразование идентификаторов производится сервером при запросах на чтение и запись.

Если файл только что создан, его atime, ctime и mtime, а также ctime и mtime родительского каталога, установлены в текущее время, В противном случае если файл модифицирован из-за флага **O_TRUNC**, его поля ctime и mtime устанавливаются в текущее время.

Ошибки

- ***EEXIST*** pathname уже существует, и были использованы **O_CREAT** и **O_EXCL** . EISDIR pathname ссылается на каталог, а тип доступа подразумевает запись.
- ***EACCES*** Запрошенный доступ к файлу не разрешен, или же один из каталогов в pathname не позволяет поиск (выполнение), или же файл еще не существует, а доступ для записи в родительский каталог не разрешен. **error_ENAMETOOLONG**(pathname) **error_ENOENT**(pathname)
- ***ENOTDIR*** Компонент, использованный как каталог в pathname , таковым не является, или был указан флаг **O_DIRECTORY** , а pathname не является

каталогом.

- ***ENXIO*** Установлены O_NONB**L**OCK | **O_WRONLY** , файл является каналом FIFO, но нет процессов, которые открыли этот канал для чтения. Возможно также, что файл является файлом устройства, но соответствующее устройство не установлено.
- ***ENODEV*** pathname ссылается на файл устройства, но соответствующего устройства не существует. (Это ошибка в ядре Linux: должен возвращаться ENXIO.)
- ***EROFS*** pathname ссылается на файл, находящийся на файловой системе, смонтированной только для чтения, и был запрошен доступ для записи.
- ***ETXTBSY*** pathname ссылается на исполняемый файл, который в настоящее время исполняется, и был запрошен доступ для записи. error_EFAULT(pathname)
- ***ELOOP*** Слишком много символических ссылок было пройдено при обнаружении pathname , или был указан флаг **O_NOFOLLOW** , а pathname является символической ссылкой.
- ***ENOSPC*** pathname нужно было создать на устройстве, на котором не было места для нового файла. **error_ENOMEM**
- ***EMFILE*** Процесс уже открыл максимально допустимое количество файлов.
- ***ENFILE*** Достигнут предел на максимальное общее количество файлов, открытых в системе.

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, BSD 4.3 Флаги **O_NOFOLLOW** и **O_DIRECTORY** специфичны для Linux. Для того, чтобы получить их определение, может потребоваться определить (через #define) символ _GNU_SOURCE .

Ограничения

В протоколе, на котором работает NFS, существует множество недоработок, в числе прочего, оказывающих влияние на **O_SYNC** и **O_NDELAY** . POSIX предоставляет три разных варианта синхронного ввода-вывода, соответствующего флагам **O_SYNC** , **O_DSYNC** и **O_RSYNC** . В настоящее время (версия 2.1.130), все они являются синонимами под Linux.

Системный вызов creat

int creat (const char *path mode_t mode);

Системный вызов полностью аналогичен системному вызову

open(pathname, O_WRONLY | O_CREAT O_TRUNC, mode).

Системный вызов close

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

close закрывает файловый дескриптор, который после этого не ссылается ни на один и файл и может быть использован повторно. Все блокировки, находящиеся на соответствующем файле, снимаются (независимо от того, был ли использован для установки блокировки именно этот файловый дескриптор).
Если fd является последней копией какого-либо файлового дескриптора, то ресурсы, связанные с ним, освобождаются; если дескриптор был последней ссылкой на файл, удаленный с помощью **unlink** (2), то файл окончательно удаляется.

Возвращаемое значение

close возвращает ноль при успешном завершении или -1, если произошла ошибка.

Ошибки

- ***EBADF*** *fd* является неверным файловым дескриптором.
- ***EINTR*** Системный вызов **close ()** был прерван сигналом.
- ***EIO*** Произошла ошибка ввода/вывода.

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. SVr4 документирует дополнительный код ошибки ENOLINK.

Замечания

Не проверять значение, возвращаемое функцией close - обычная, но от этого не менее серьезная ошибка программирования. Вполне возможно, что ошибка в предыдущей операции **write** (2) впервые даст о себе знать при выполнении close . Отсутствие проверки возвращаемого значение при закрытии файла, может привести к потере данных. Особенно это касается таких аспектов как NFS и дисковые квоты.

Успешное выполнение **close**() не гарантирует, что данные успешно записаны на диск,

потому что ядро откладывает запись. Обычно файловые системы не сбрасывают буфера на диск при закрытии потока. Если вам нужно удостовериться, что данные физически сохранены на диске, используйте **fsync** (2). (В этом случае всё будет зависить от железа и самого диска.)

Системный вызов read

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

Описание

read() пытается читать из файлового дескриптора fd байты в количестве count в буфер, начинающийся по адресу buf . Если count равен нулю, то read() возвращает ноль и ничего не делает. Если count больше, чем **SSIZE_MAX** , то результат неопределен.

Возвращаемое значение

При успешном завершении возвращается количество байт, которые были прочитаны (ноль означает конец файла), а позиция в файле увеличивается на это значение. Если количество прочитанных байт меньше, чем количество запрошенных, то это не считается ошибкой: например, мы могли читать вблизи конца файла, или же из трубы, или с терминала, или же read() был прерван сигналом. В случае ошибки возвращается -1, а errno устанавливается должным образом. В этом случае позиция файла неопределена.

Ошибки

- **EINTR** Системный вызов был прерван сигналом до того, как был прочитан хотя бы байт.
- **EAGAIN** Был запрошен неблокирующий ввод-вывод с помощью **O_NONBLOCK ,** и нет данных, немедленно доступных для чтения.
- **EIO** Ошибка ввода-вывода. Это может произойти, например, если процесс, находящийся в фоновой группе процессов, пытается читать с контролирующего терминала, и игнорирует или блокирует сигнал **SIGTTIN ,** или же его группа процессов осталась без родителя. Это может также случиться, если произошла низкоуровневая ошибка ввода-вывода при чтения с диска или ленты.
- **EISDIR** *fd* ссылается на каталог.
- **EBADF** *fd* не является допустимым дескриптором файлов или не открыт для чтения.
- **EINVAL** *fd* связан с объектом, неприспособленным для чтения.
- **EFAULT** *buf* указывает за пределы доступного адресного пространства.

Могут также возникнуть другие ошибки, в зависимости от объекта, связанного с *fd* . POSIX позволяет системному вызову **read ,** который был прерван после чтения первого куска запрошенных данных, вернуть -1 (устанавливая *errno* в EINTR) или количество уже прочитанных байт.

Соответствие стандартам

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

Ограничения

На файловых системах NFS чтение небольших порций данных обновляет отметки времени только в первый раз, последующие операции чтения не делают этого. Это вызвано кэшированием атрибутов с клиентской стороны, потому что большинство (если не все) клиентов NFS предоставляют серверу обновлять время доступа, а запросы на чтение, которые удовлетворяются из клиентского кэша, не вызывают обновления времени доступа, потому что данные не читаются с сервера. Семантика UNIX может быть достигнута запретом кэширования атрибутов, но в большинстве случаев это увеличит нагрузку на сервер и уменьшит производительность.

Системный вызов write

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

Описание

write пишет count байт в файл, на который ссылается файловый дескриптор fd из буфера, на который указывает buf . POSIX требут, чтобы read(), который может использоваться после write() возвратил новые данные. Заметим, что не все файловые системы соответствуют стандарту POSIX.

Возвращаемое значение

В случае успеха, возвращается количество записанных байт (ноль говорит о том, что ничего записано не было). В случае ошибки, возвращается -1 и значение errno устанавливается соответствующим образом. Если count равен нулю и файловый дескриптор указывает на обычный файл, будет возвращаться 0 без каких-либо других последствий. Для специальных файлов, получившиеся результаты не переносимы на другие платформы.

Ошибки

- ***EBADF*** *fd* не является правильным файловым дескриптором или не открыт для записи.
- ***EINVAL*** *fd* ссылается на объект, который не может быть доступен для записи.
- ***EFAULT*** *buf* находится за пределами доступного вам адресного пространства.
- ***EFBIG*** Попытка записать в файл, который превышает заданное при

реализации ограничение на размер файла или ограничение на размер файла для текущего процесса, или запись в позицию после максимально разрешённого смещения.

- ***EPIPE*** *fd* ссылается на конвеер или сокет, у которого закрыто чтение. Когда такое случается, пишущий процесс также получит сигнал **SIGPIPE .** (Таким образом, возвращаемое значение можно будет увидеть только если программа перехватывает, блокирует или игнорирует этот сигнал).
- ***EAGAIN*** Был выбран неблокирующий ввод/вывод, через **O_NONBLOCK ,** а запись должна произвести блокировку.
- ***EINTR*** Этот вызов был прерван сигналом, перед тем как были записаны какие-либо данные.
- ***ENOSPC*** На устройстве, содержащем файл, на который ссылается *fd* нет свободного места.
- ***EIO*** Во время изменения индексного дескриптора (inode) возникла низкоуровневая ошибка ввода/вывода.

  В зависимости от объекта, на который указывает *fd* могут происходить и другие ошибки.

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, 4.3BSD. SVr4 документирует дополнительные условия ошибок **EDEADLK, ENOLCK, ENOLNK, ENOSR, ENXIO** или **ERANGE.** Под SVr4 вызов write может быть прерван и возвратить **EINTR** в любом месте, а не только перед тем как будут записаны какие-либо данные.

Замечания

Успешный возврат из вызова write не даёт никаких гарантий, что данные сохранены на диске. Фактически, в некоторых некачественных реализациях, даже нет гарантии, что для данных было зарезервировано место. Есть только один способ, чтобы быть уверенным в сохранении данных на диске - это вызов **fsync (**2) после того как вы запишете все ваши данные.

Системный вызов lseek

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

Функция lseek устанавливает смещение для файлового дескриптора fildes в значение аргумента offset в соответствии с директивой whence которая может принимать одно из следующих значений:

- ***SEEK_SET*** Смещение устанавливается в *offset* байт (от начала файла -- прим. пер.).
- ***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.

Ошибки

- ***EBADF*** *fildes* не является дескриптором открытого файла.
- ***ESPIPE*** *fildes* ассоциирован с каналом, сокетом или FIFO.
- ***EINVAL*** *whence* не является одним из значений SEEK_SET, SEEK_CUR, SEEK_END или смещение в файле, которое получилось в результате является отрицательным.
- ***EOVERFLOW*** Получившееся в результате смещение не может быть представлено типом off_t.

Соответствие стандартам

SVr4, POSIX, BSD 4.3

Ограничения

Некоторые устройства несовместимы с позиционированием, а POSIX не определяет какие устройства должны его поддерживать.

Специальные ограничения Linux: при использовании lseek на терминальных

устройствах tty возвращается ESPIPE.

Замечания

Использование в этом документе слова *whence* является некорректным для английского языка, но в силу исторических причин использование этого слова продолжается.

Если вы будете конвертировать старый код, подставляйте вместо значений *whence* следующие макросы:

old	new
0	SEEK_SET
1	SEEK_CUR
2	SEEK_END
L_SET	SEEK_SET
L_INCR	SEEK_CUR
L_XTND	SEEK_END

SVR1-3 возвращает long вместо off_t, BSD возвращает int.
Заметим, что файловые дескрипторы, созданные через dup(2) или fork(2) разделяют указатель текущей позиции в файле, так что позиционирование таких файлов может быть выполнено на тех же условиях.

Системные вызовы pread и pwrite

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
ssize_t pread(int fd , void * buf , size_t count , off_t offset );
ssize_t pwrite(int fd , const void * buf , size_t count , off_t offset );
```

pread() читает *count* байт из файлового дескриптора *fd* по смещению *offset* (от начала файла) в буфер *buf .* Позиция смещения в файле при этом не изменяется.

pwrite() пишет *count* байт из буфера *buf* в файловый дескриптор *fd* по смещению *offset .* Позиция смещения в файле при этом не изменяется.

Файл, на который указывает дескриптор *fd* должен поддерживать позиционирование.

Возвращаемое значение

В случае успеха, возвращается количество прочитанных или записанных байт (ноль говорит о том, что ничего не было записано в случае pwrite или что достигнут конец файла в случае pread), или -1 в случае ошибки, при этом значение *errno* содержит ошибку.

Ошибки

pread может вернуть ошибку и установить *errno* в любое значение из тех, что устанавливаются для read(2) или lseek(2). **pwrite** может вернуть ошибку и установить *errno* в любое значение из тех, что устанавливаются для write(2) или lseek(2).

Соответствие стандартам

Unix98

История

Системные вызовы pread и pwrite были добавлен в Linux начиная с версии 2.1.60; записи в таблице системных вызовов для i386 были добавлены в 2.1.69. Поддержка libc (включая эмуляцию на старых ядрах, без этих системных вызовов) была добавлена в glibc 2.1.

Системные вызовы dup и dup2

```
#include <unistd.h>
int dup(int oldfd ); int dup2(int oldfd , int newfd );
```

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 закрыть-при-exec. **dup** использует самый маленький свободный 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 должным образом).

Ошибки

- ***EBADF*** *oldfd* не является открытым файловым дескриптором, или же *newfd* находится вне допустимого диапазона файловых дескрипторов.
- ***EMFILE*** Процесс уже открыл максимальное количество файлов и пытается открыть еще один.
- ***EINTR*** Вызов **dup2** был прерван каким-либо сигналом.
- ***EBUSY*** (Только в Linux) Может случиться в **dup2** во время при условии состязательных вызовов open() и dup().

Внимание

Ошибка, которую возвращает **dup2 ,** отличается от той, что возвращает **fcntl(** ..., **F_DUPFD** , ... **) ,** когда *newfd* находится вне допустимых пределов. На некоторых системах **dup2** также иногда возвращает **EINVAL ,** как **F_DUPFD .**

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 .*

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. SVr4 документирует дополнительные коды ошибки EINTR и ENOLINK. POSIX.1 добавляет EINTR. Ошибка EBUSY является специфичной для Linux.

Системный вызов sync

```
#include <unistd.h>
void sync(void);
```

Описание

sync сперва записывает индексные дескрипторы (inode) в буферы и затем буферы на диск.

Ошибки

Данная функция всегда выполняется без ошибок.

Соответствие стандартам

SVr4, SVID, X/OPEN, BSD 4.3

Баги

В соответствии со стандартной спецификацией (например SVID), функция sync() выполняет запись, но возврат из неё может произойти перед тем как запись будет фактически завершена. Однако, начиная с версии 1.3.30, Linux выполняет ожидание окончания записи. (Это впрочем не гарантирует целостность данных: современные диски имеют свои собственные большие кэши).

Замечания

Начиная с glibc 2.2.2 прототип функции в Linux такой как показано выше. В libc4,

libc5, и glibc до версии 2.2.1, он выглядит как "int sync(void)" и sync всегда возвращает 0.

Системный вызов fsync

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

Описание

fsync копирует на диск все части файла, находящиеся в памяти и ожидает пока устройство скажет, что все эти части сохранены. Также данный вызов обновляет информацию о состоянии метаданных. Это создаёт уверенность, что запись в каталоге содержит файл, который также находится на диске. Чтобы проделать это явно вызову **fsync** также нужен файловый дескриптор каталога.

fdatasync делает тоже самое что и **fsync ,** но сбрасывает только данный пользователя, а не метаданные, такие как mtime или atime.

Возвращаемое значение

В случае успеха возвращается ноль. При ошибке возвращается -1 и *errno* устанавливается соответствующим образом.

Ошибки

- **EBADF** *fd* не является дескриптором файла, открытого для записи.
- **EROFS** , **EINVAL** *fd* связан со специальным файлом, который не поддерживает синхронизацию.
- **EIO** Во время синхронизации произошла ошибка.

Замечания

В случае если у жёсткого диска разрешена запись кэша, данные могут фактически не быть сохранены после выполнения **fsync / fdatasync .**

Когда файловая система ext2 монтируется с опцией *sync ,* при вызове **fsync** также неявно синхронизируются записи в каталогах.

В ядрах до 2.4, **fsync** на больших файлах может быть неэффективным. В качестве альтернативы можно использовать флаг *O_SYNC* в вызове open (2)**.**

Соответствие стандартам

POSIX.1b (когда-то назывался POSIX.4)

Системный вызов fdatasync

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

Описание

fdatasync записывает на диск содержимое всех буферов данных, связанных с файлом, причем возврат из функции происходит только после того, как это было сделано. Этот системный вызов напоминает **fsync ,** но от него не требуется обновлять метаданные, например, время доступа.

Приложения, которые работают с базами данных или журнальными файлами, часто пишут небольшие фрагменты данных (например, строчку журнала за строчкой), а затем вызывают **fsync ,** чтобы убедиться, что записанные данные сохранены на жестком диске. К сожалению, **fsync** всегда приводит к двум операциям записи: одной для новых данных, и еще одной для того, чтобы обновить информацию, хранящуюся в inode. Если время модификации файла неважно для программы, то можно использовать **fdatasync ,** чтобы избежать ненужной операции записи inode.

Возвращаемое значение

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

Ошибки

- **EBADF** *fd* не является правильным дескриптором файла, открытым для записи.
- **EROFS** , **EINVAL** *fd* связан со специальным файлом, не поддерживающим синхронизацию.
- **EIO** Во время синхронизации произошла ошибка.

Ошибки

В настоящий момент (Linux 2.2) **fdatasync** эквивалентен **fsync .**

Доступность

На POSIX системах, где вызов **fdatasync** недоступен, символ **_POSIX_SYNCHRONIZED_IO ,** определяемый в <unistd.h> устанавливается в значение больше 0. (См. также sysconf (3)**.)**

Соответствие стандартам

POSIX1b (ранее известный как POSIX.4)

Системный вызов fcntl

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

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

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

"Управление close-on-exec"

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.

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. В Linux данная команда может изменять только флаги O_APPEND, O_NONBLOCK, O_ASYNC и O_DIRECT. .P

"Совместная (advisory) блокировка" **F_GETLK** , **F_SETLK** и **F_SETLKW** используются для установки, снятия и тестирования существующих блокировок записи (также известных как блокировки сегмента файла или области файла). Третий аргумент *lock* является указателем на структуру, которая имеет по крайней мере следующие поля (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 " u " l_len* этой структуры задают диапазон байт, который мы хотим заблокировать. *l_start* - это начальное смещение для блокировки, которое интерпретируется как: начало файла (если значение *l_whence* установлено в **SEEK_SET);** текущая позиция в файле (если значение *l_whence* установлено в **SEEK_CUR);** или конец файла (если значение *l_whence* установлено в **SEEK_END).** В последних двух случаях, *l_start* может иметь отрицательное значение, предоставляя смещение, которого не может быть перед началом файла. *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 " u " l_start* до конца файла, не зависимо от того, насколько велик файл. .P Поле *l_type* может быть использовано для указания типа блокировки: чтение (F_RDLCK) или запись (F_WDLCK). Блокировку на чтение (разделяемая блокировка) на область файла может удерживать любое количество процессов, но только один процесс может удерживать блокировку на запись (эксклюзивная блокировка). 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.)

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 " u " 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 " u " l_len* структуры *lock* и устанавливает *l_pid* в значение PID того процесса, который удерживает блокировку. .P 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. .P 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. .P Блокировки не наследуются процессом-потомком, созданным через fork (2)**,** но сохраняются при вызове execve (2)**.** .P 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)**.** .P

"Обязательная (mandatory) блокировка" (Не-POSIX.) Описанные выше блокировки могут быть или совместные или обязательные и по умолчанию используются совместные. Чтобы использовать обязательные блокировки, они должны быть разрешены (с помощью опции "-o mand" в mount (8)**)** для файловой системы, содержащей файл, который должен быть заблокирован, а также разрешены для самого файла (с помощью запрещения прав на выполнение группой и установки set-GID бита в правах доступа к файлу).

Совместные блокировки не являются принудительными и полезны только между сотрудничающими процессами. Обязательные блокировки являются принудительными для всех процессов. .P

"Управление сигналами" **F_GETOWN** , **F_SETOWN** , **F_GETSIG** и **F_SETSIG** используются для управления сигналами доступности ввода/вывода:

F_GETOWN Получить идентификатор процесса или группу процесса, которые в текущий момент принимают сигналы SIGIO и SIGURG для событий на файловом дескрипторе *fd .* Группы процессов возвращаются в виде отрицательных значений.

F_SETOWN Установить идентификатор процесса или группу процесса, которые будут принимать сигналы SIGIO и SIGURG для событий на файловом дескрипторе *fd .* Группы процессов задаются в виде отрицательных значений.

(F_SETSIG может использоваться, чтобы задать другой сигнал вместо SIGIO).

Если вы установили на файловый дескриптор флаг состояния **O_ASYNC** (или через предоставление этого флага в вызове open (2)*,* или используя команду **F_SETFL** вызова **fcntl),** то сигнал SIGIO посылается всякий раз, когда для данного файлового дескриптора становится возможным ввод или вывод.
Процесс или группа процесса, для приёма сигнала могут быть выбраны, используя команду **F_SETOWN** в вызове **fcntl .** Если файловым дескриптором является сокет, то для него также выбирается получатель сигналов SIGURG, которые доставляются,

когда на сокет поступает данных больше, чем его пропускная способность. (SIGURG посылается во всех ситуациях, когда вызов select (2) говорит, что сокет находится в состоянии "исключительной ситуации"). Если файловый дескриптор соответствует терминальному устройству, то группе процессов, которые работают с терминалом не в фоновом режиме будут посылаться сигналы SIGIO.

F_GETSIG Получить сигнал, который будет послан, когда станет возможен ввод или вывод. Значение 0 означает сигнал SIGIO. Любое другое значение (включая SIGIO) является другим сигналам, посылаемым вместо SIGIO и в это случае, для обработчика сигнала доступна дополнительная информация, если он был установлен с SA_SIGINFO.

F_SETSIG Установить сигнал, который будет послан, когда станет возможен ввод или вывол. Значение 0 означает сигнал SIGIO. Любое другое значение (включая SIGIO) является другим сигналам, посылаемым вместо SIGIO и в это случае, для обработчика сигнала доступна дополнительная информация, если он был установлен с SA_SIGINFO.
В случае использования F_SETSIG с ненулевым значением и установкой SA_SIGINFO для обработчика сигнала (см. sigaction (2)**),** обработчику передаётся дополнительная информация о событиях ввода/вывода в структуре *siginfo_t .* Если поле *si_code* показывает, что сигналом является SI_SIGIO, то поле *si_fd* содержит файловый дескриптор, ассоциированный с собрытием вывода/вывода. В противном случае, не существует никакого механизма, чтобы сообщить с каким файловым дескриптором связан полученный сигнал и вы должны использовать

(select (2), poll (2), read (2) с установленным флагом **O_NONBLOCK** и т.д.) чтобы определить какой файловый дескриптор доступен для ввода/вывода.
При выборе сигнала реального времени (value >= SIGRTMIN) как описано в POSIX. 1b, в очередь будут добавляться несколько событий ввода вывода, с теми же номерами сигналов. (Размер очереди зависит от доступной памяти). Дополнительная информация будет доступна как описано выше, если для обработчика сигнала будет установлено SA_SIGINFO.

Используя эти механизмы, программа может реализовать полностью асинхронный ввода/вывод без использования в своей работе select (2) или poll (2). Использование **O_ASYNC , F_GETOWN , F_SETOWN** является специфичным для BSD и Linux. **F_GETSIG** и **F_SETSIG** являются специфичными для Linux. POSIX описывает асинхронный ввод/вывод и структуру *aio_sigevent* используемую для сходных вещей; которые также доступны в Linux как часть библиотеки GNU C (Glibc). .P

"Аренда" **F_SETLEASE** и **F_GETLEASE** (в Linux 2.4 и выше) используются (соответственно) для установки и получения текущих настроек вызывающего процесса, арендующего файл, на который указывает *fd .* Аренда файла предоставляет такой механизм, когда процесс, удерживающий аренду ("держатель аренды") уведомляется (через доставку сигнала), о том, что другой процесс ("конкурент") пытается выполнить open (2) или truncate (2) для этого файла.

F_SETLEASE Устанавливает или удаляет аренду файла, в соответствии со целым значением, установленным в аргументе *arg :*

F_RDLCK Установить аренду чтения. Это приведёт к генерации уведомления, когда другой процесс открывает указанный файл для записи или усекает его.

F_WRLCK Установить аренду записи. Это приведёт к генерации уведомления, когда другой процесс открывает указанный файл (для чтения или записи) или усекает его. Аренда записи может быть установлена на файл, только если этот файл не открыт в этот момент каким-либо другим процессом.

F_UNLCK Удалить аренду с указанного файла.

.P Процесс может удерживтать для файла только один тип аренды. .P Аренда может быть установлена только для обычных файлов. Непривелегированный процесс может установить аренту только для файла, у которого UID совпадает с UID файловой системы этого процесса.

F_GETLEASE Показывает какой тип аренды удерживается для файла, на который указывает *fd* и возвращает одно из значений **F_RDLCK** , **F_WRLCK** или **F_UNLCK,** соответственно показывающих, что вызывающий процесс удерживает аренду чтения, записи или что аренды нет. (Третий аргумент **fcntl ()** при этому опускается.)

Когда конкурент выполняет вызов **open ()** или **truncate (),** который конфликтует с арендой, установленной через **F_SETLEASE ,** системный вызов блокируется ядром (за исключением случая, когда при вызове **open ()** указывается флаг **O_NONBLOCK ;** в этом случае немедленно возращается ошибка **EWOULDBLOCK).** Затем ядро уведомляет держателя аренды, отправляя ему сигнал (по умолчанию SIGIO). Держатель аренты должен при получении этого сигнала выполнить все необходимые действия для подготовки этого файла к использованию другим процессом (например, сбросить буферы кэша) и затем удалить аренду, выполнив операцию **F_SETLEASE** и установив значение аргумента *arg* в **F_UNLCK .** Если держатель аренды не освободит аренду в течении количества секунд, которое задаётся в файле */proc/sys/fs/lease-break-time ,* а системный вызов конкурента останется блокирующим (то, есть конкурент не установит флаг **O_NONBLOCK** для системного вызова **open ()** и этот системный вызов не будет прерван обработчиком события), то ядро принудительно удалит аренду у процесса держателя аренды.

После того как аренда снята держателем аренды или принудительно удалена ядром, ядро снимает блокировку с системного вызова процесса конкуренту и разрешает продолжить его работу.

По умолчанию, для уведомления держателя аренды используется сигнал SIGIO, но его можно изменить, используя команду **F_SETSIG** для **fcntl ().** Если выполняется команда **F_SETSIG** (даже назначая сигнал SIGIO) и при этом обработчик сигнала устанавливается с использованием SA_SIGINFO, то обработчик будет получит в качестве второго аргумента структуру *siginfo_t ,* в которой поле *si_fd* будет содержать дескриптор файла, для которого установлена аренда и которому пытается получить доступ другой процесс. (Это полезно, если вызывающий процесс удерживает аренду на несколько файлов).

"Уведомления об изменении файла и каталога"

F_NOTIFY (Начиная с Linux 2.4) Предоставляет уведомление, когда изменяется каталог, на который указывает *fd* или файлы, которые в нём содержатся. События, о наступлении которых делается уведомление, задаются в аргументе *arg ,* который является битовой маской, получаемой битовым сложением (OR) одной или более следующих масок:

Маска Описание (событие в каталоге)

- ***DN_ACCESS*** Был осуществлён доступ к файлу (read, pread, readv)
- ***DN_MODIFY*** Файл был изменён (write, pwrite, writev, truncate, ftruncate)
- ***DN_CREATE*** Файл был создан (open, creat, mknod, mkdir, link, symlink, rename)
  DN_DELETE Файл был удалён (unlink, rename в другой каталог, rmdir)
- ***DN_RENAME*** Файл был переименован внутри каталога (rename)
- ***DN_ATTRIB*** У файла был изменён атрибут (chown, chmod, utime[s])

(Чтобы включить определения этих масок, нужно задать макроопределение _GNU_SOURCE до подключения <fcntl.h>.)
Уведомления об изменении состояния каталога обычно однократные и приложение должно перерегистрировать установку уведомлений, чтобы и дальше получать их. Однако, если в аргумент *arg* добавить маску **DN_MULTISHOT ,** то уведомления будут происходить до тех пор, пока не будут явно отменены.

Серии запросов **F_NOTIFY** накапливаются, таким образом запросы на уведомления, указываемые в *arg* будут добавляться к тем, что уже установлены. Чтобы запретить уведомления для всех событий, выполните вызов **F_NOTIFY ,** в котором значение *arg* установлено в 0.
Уведомления выполняются через доставку сигнала. По умолчанию - это SIGIO, но вы можете изменить его, используя команду **F_SETSIG** для вызова **fcntl ().** В последнем случае, обработчик сигнала получит в качестве второго аргумента структуру *siginfo_t* (если обработчик был установлен с использованием SA_SIGINFO), а поле *si_fd* в этой структуре будет содержать дескриптор файла, для которого было сегенерировано уведомление (полезно, когда устанавливается уведомление для нескольких каталогов).
Кроме того, когда используется **DN_MULTISHOT ,** для уведомлений должен бы быть использован сигнал реального времени POSIX.1b, так что множественные уведомления могут быть поставлены в очередь.

Возвращаемое значение

При успешном вызове, возвращаемое значение зависит от использованной операции:

0.9i **F_DUPFD** Новый файловый дескриптор.

F_GETFD Значение флага.

F_GETFL Значение флага.

F_GETOWN Значение, представляющее собой владельца дескриптора.

F_GETSIG Значение сигнала, посылаемого когда становится возможным чтение или запись или ноль для традиционного поведения SIGIO.

Все другие команды. Ноль.

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

Ошибки

EACCES или **EAGAIN** Операция запрещена блокировками, которые удерживаются другими процессами. Или, операция запрещена, потому что для этого файла другой процесс использует механизм отображения в память (memory-map).

EBADF *fd* не является открытым файловым дескриптором или команда была **F_SETLK** или **F_SETLKW** и режим открытия файлового дескриптора не совпадает с типом запрошенной блокировки.

EDEADLK Было обнаружено, что указанная команда **F_SETLKW** должна вызывать мёртвую блокировку (deadlock).

EFAULT *lock* находится за пределами доступного адресного пространства.

EINTR Выполнение команды **F_SETLKW ,** было прервана сигналом. Или выполнение команд **F_GETLK** и **F_SETLK ,** было прервано сигналом перед тем как была блокировка была проверка или установлена. Большинство таких ошибок случается при блокировке удалённого файла (например, блокировка через NFS), но иногда такое может случаться и на локальных файлах.

EINVAL Для **F_DUPFD , значение** *arg* отрицательное или же больше, чем максимально возможное значение. Для **F_SETSIG , значение** *arg* не содержит допустимый номер сигнала.

EMFILE Для **F_DUPFD ,** процесс достиг максимального количества открытых файловых дескрипторов.

ENOLCK Открыто слишком много блокировок сегментов, таблица блокировок заполнена или ошибка протокола удалённой блокировки (например, при блокировке через NFS).

EPERM Попытка сбросить флаг **O_APPEND** на файле, который открыт с атрибутом только для добавления.

Замечания

Ошибки, возвращаемые **dup2** отличаются от тех, что возвращаются при **F_DUPFD .** Начиная с ядра 2.0, не существует разницы между типами блокировки, которые осуществляет flock (2) и **fcntl (2).** POSIX 1003.1-2001 разрешает отрицательное значение *l_len .* (И если оно отрицательное, интервал, описывающий блокировку покрывает *l_start* + *l_len* байт, включая *l_start -1.)* Это поддерживатся, начиная с Linux 2.4.21 и 2.5.49.

Некоторые системы имеют больше полей в структуре *"struct flock"* например такие как *l_sysid .* Вообще-то, один *l_pid* не очень-то полезен, если процесс, удерживающий блокировку может работать на другой машине.

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. В стандарте POSIX.1 описываются только F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETLK, F_SETLK и F_SETLKW. F_GETOWN и F_SETOWN в BSD, не поддеживаются в SVr4; F_GETSIG и F_SETSIG являются специфичными для Linux. **F_NOTIFY** , **F_GETLEASE** , и

F_SETLEASE являются специфичными для Linux. (Для подключения этих флагов задавайте макрос _GNU_SOURCE перед подключением заголовочного файла <fcntl.h>.) Флаги, которые являются допустимыми для F_GETFL/F_SETFL также поддерживаются вызовом open (2) и варьируются в зависимости от операционных систем; O_APPEND, O_NONBLOCK, O_RDONLY и O_RDWR описываются в POSIX.1. SVr4 поддерживает несколько других опций и флагов, которые здесь не описываются.

SVr4 документирует дополнительные условия ошибок EIO, ENOLINK и EOVERFLOW.

Системный вызов ioctl

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

Описание

Функция **ioctl** манипулирует базовыми параметрами устройств, представленных в виде специальных файлов. В частности, многими оперативными характеристиками специальных символьных файлов (например терминалов) можно управлять через **ioctl** запросы. В качестве аргумента *d* должен быть указан открытый файловый дескриптор.

Второй аргумент является кодом запроса, который зависит от устройства. Третий аргумент является указателем на память, который не имеет типа. Традиционно это **char *** argp (до тех пор пока в C не появился **void *).** Ioctl запрос *request* кодирует в себе либо аргумент, который является параметром *in* либо аргумент, который является параметром *out* и кроме того размер аргумента *argp* в байтах. Макросы и определения, используемые в специальных ioctl запросах *request* находятся в файле *<sys/ioctl.h> .*

Возвращаемое значение

Обычно в случае успеха возвращается ноль. Некоторые ioctl используют возвращаемое значение как выходной параметр и возвращают в случае успеха неотрицательное значение. В случае ошибки возвращается -1 и значение *errno* устанавливается соответствующим образом.

Ошибки

- **EBADF** *d* не является правильным дескриптором.
- **EFAULT** *argp* ссылается на недоступную область памяти.
- **ENOTTY** *d* не ассоциирован со специальным символьным устройством.
- **ENOTTY** Указанный запрос не применим к данному классу объекта, на

который ссылается дескриптор *d* .

- **EINVAL** *Request* или *argp* заданы неверно.

Соответствие стандартам

Нет единого стандарта. Аргументы, возвращаемые значения и семантики **ioctl (2)** варьируются в соответствии с драйвером устройства (вызов, используется как всеохватывающий, что не полностью соответствует потоковой модели ввода/вывода в Unix). Смотри **ioctl_**list (2) где дан список многих известных **ioctl** вызовов. Функция **ioctl** появилась в AT&T Unix версии 7.

Литература

1. http://ru.gentoo-wiki.com, man файлы, перевод Алексея Махоткина.
2. В.А. Галатенко, Программирование в стандарте POSIX, http://www.intuit.ru/department/se/pposix/
3. http://www.opennet.ru/man_2.shtml
4. У. Ричард Стивенс, Стивен А. Раго, UNIX. Профессиональное программирование.
5. Д. Бовет, М. Чезати. Ядро LINUX.
6. Роберт Лав. Разработка ядра Linux. Второе издание.
7. S.R. Kleiman, Vnodes: An Architecture for Multiple File System Types in Sun UNIX
8. Neil Brown, The Linux Virtual File-system Layer. http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html#toc7
9. Андрей Робачевский. Операционная система UNIX.
10. Андрей Боровский. Программирование для Linux. http://symmetrica.net/uploads/linapi/