Часть 2. Системные вызовы

EXIT

НАЗВАНИЕ _exit, _Exit - функция, завершающая работу программы

СИНТАКСИС

```
#include <unistd.h>
#include <stdlib.h>
void _Exit(int status);
```

ОПИСАНИЕ

_exit "немедленно" завершает работу программы. Все дескрипторы файлов, принадлежащие процессу, закрываются; все его дочерние процессы начинают управляться процессом 1 (init), а родительскому процессу посылается сигнал **SIGCHLD.**

Значение *status* возвращается родительскому процессу как статус завершаемого процесса; он может быть получен с помощью одной из функций семейства **wait.**

Функция **_Exit** эквивалентна функции **_exit.**

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Эти функции никогда не возвращают управление вызвавшей их программе.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. Функция *_Exit()* была представлена C99.

ЗАМЕЧАНИЯ

Для рассмотрения эффектов завершения работы, передачу статуса выхода, зомби-процессов, сигналов и т.п., смотрите документацию по **exit(3).**

Функция **_exit** аналогична **exit(),** но не вызывает никаких функций, зарегистрированных с функцией ANSI C **atexit,** а также не вызывает никаких зарегистрированных обработчиков сигналов. Будет ли выполняться сброс стандартных буферов ввода-вывода и удаление временных файлов, созданных **tmp-file(3),** зависит от реализации. С другой стороны, **_exit** закрывает открытые дескрипторы файлов, а это может привести к неопределенной задержке для завершения вывода данных. Если задержка нежелательна, то может быть полезным перед вызовом **_exit()** вызывать функции типа *tcflush()*. Будет ли завершен ввод-вывод, а также какие именно операции ввода-вывода будут завершены при вызове **_exit(),** зависит от реализации.

СМ. ТАКЖЕ fork(2), execve(2), waitpid(2), wait4(2), kill(2), wait(2), exit(3), termios(3)

Linux 2001-11-17 _EXIT**(2)**

ACCEPT

НАЗВАНИЕ accept - принять соединение на сокете

СИНТАКСИС

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t
*addrlen);
```

ОПИСАНИЕ

Функция **accept** используется с сокетами типов (**SOCK_STREAM, SOCK_SEQPACKET** и **SOCK_RDM**). Эта функция выбирает первый запрос на соединение из очереди ожидающих соединений, создает новый сокет с теми же свойствами, что и у *s*, и выделяет для сокета новый файловый описатель, который

возвратится этой функцией. Новый созданный сокет уже не будет находится в состоянии ожидания соединения. При этом вызове исходный сокет *s* не изменяется. Обратите внимание, что любые флаги описателя файла (например, состояния non-blocking или async, которые могут быть установлены функцией fcntl с командой **F_SETFL**) не наследуются от *accept*.

Аргумент *s* - это сокет, который был создан с помощью функции **socket(2)**, привязан к локальному адресу с помощью функции **bind(2)**, и ожидает входящие соединения после вызова функции **listen(2)**. Аргумент *addr* - указатель на структуру **sockaddr**. В эту структуру записан адрес подключающейся стороны, известный на уровне соединения. Точный формат адреса, передаваемого в аргументе *addr*, определяется семейством сокета (см. **socket(2)** и страницы руководства по соответствующему протоколу). Аргумент *addrlen* - параметр, в который после вызова функции помещается результат. Перед вызовом функции он содержит размер структуры на которую указывает *R addr ;* после вызова он будет содержать фактическую длину (в байтах) адреса. Если в *addr* записано значение NULL, то он не заполняется.

Если очередь ожидающих соединений пуста и сокет отмечен как non-blocking (неблокирующий), тогда **accept** блокирует вызывающий процесс до появления соединения. Если сокет отмечен как non-blocking и в очереди нет никаких ожидающих соединений, тогда **accept** возвращает EAGAIN.

Для того, чтобы получать уведомления о входящих подключениях на сокете, вы можете использовать **select(2)** или **poll(2)**. В этом случае, когда придет запрос на новое соединение, будет получено событие "можно читать", и тогда вы можете вызвать **accept**, чтобы получить сокет для этого соединения. В другом случае, вы можете заставить сокет передавать сигнал **SIGIO**, когда он активируется; см. **socket(7)** для уточнения подробностей.

Для некоторых протоколов, которые требуют явного подтверждения, напр. DECNet, вызов **accept** можно рассматривать просто как извлечение из очереди следующего запроса на подключение без подтверждения. Подтверждение произойдет при чтении или записи в новый файловый описатель, а отказ от соединения может произойти при закрытии нового сокета. В настоящий момент под Linux такую особенность имеет только DECNet.

ЗАМЕЧАНИЯ

Не всегда после получения сигнала **SIGIO** или после возврата из **select(2)** или из **poll(2)** в очереди будет находиться соединение. Оно может быть удалено асинхронной сетевой ошибкой или другим потоком до вызова **accept**. Если такое случается, то вызов будет блокирует ожидание до следующего подключения. Чтобы гарантировать, что **accept** никогда не заблокируется, в сокете *s* должен быть установлен флаг **O_NONBLOCK** (см. **socket(7)**).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае ошибки функция возвращает значение -1. При успешном завершении возвращается неотрицательное целое значение, являющееся описателем сокета.

ОБРАБОТКА ОШИБОК

В Linux **accept** сообщает об ожидающих в очереди сетевых ошибках, возвращая их код при вызове **accept**. Это поведение отличается от других реализаций сокетов BSD. Для надежной работы приложения должны отслеживать сетевые ошибки, определенные для протокола, и обрабатывать их как **EAGAIN**, с помощью повторного выполнения функции. В случае

TCP/IP такими ошибками являются **ENETDOWN**, **EPROTO**, **ENOPROTOOPT**, **EHOSTDOWN**, **ENONET**, **EHOSTUNREACH**, **EOPNOTSUPP**, и **ENETUNREACH**.

КОДЫ ОШИБОК

EAGAIN или **EWOULDBLOCK**
Сокет отмечен как non-blocking, но нет ни одного соединения, которое можно принять.

EBADF Неправильный описатель.

ENOTSOCK
Описатель указывает на файл (не на сокет).

EOPNOTSUPP
Сокет, на который указывает описатель, имеет тип, отличный от **SOCK_STREAM**.

EFAULT Параметр *addr* находится в пространстве адресов с запрещенной записью.

EPERM Правила сетевого экрана (firewall) запрещают соединение.

EINTR Системный вызов был прерван сигналом перехваченным до того, как было установлено корректное соединение. **ENOBUFS**, **ENOMEM** Не хватает свободной памяти. Часто это означает, что распределение памяти ограничено пределами буфера сокета, а не памятью системы, но это не обязательно может быть так.

В дополнение к этим ошибкам, могут также возвращаться сетевые ошибки сокета и ошибки, определенные для протокола. Различные ядра Linux могут вернуть другие ошибки, например, **EMFILE**, **EINVAL**, **ENOSR**, **ENOBUFS**, **EPERM**, **ECONNABORTED**, **ESOCKTNOSUPPORT**, **EPROTONOSUPPORT**, **ETIMEDOUT**. Значение **ERESTARTSYS** может быть получено во время трассировки.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (функция **accept** впервые появилась в BSD 4.2). Руководство BSD описывает пять возможных кодов ошибок (EBADF, ENOTSOCK, EOPNOTSUPP, EWOULDBLOCK, EFAULT). SUSv2 описывает ошибки EAGAIN, EBADF, ECONNABORTED, EFAULT, EINTR, EINVAL, EMFILE, ENFILE, ENOBUFS, ENOMEM, ENOSR, ENOTSOCK, EOPNOTSUPP, EPROTO, EWOULDBLOCK.

Linux _не_ наследует флаги сокета, подобного O_NONBLOCK. Это поведение отличается от других реализаций BSD сокетов. Переносимые программы не должны полагаться на это поведение и всегда устанавливать все требуемые флаги на сокете, возвращенные функцией accept.

ЗАМЕЧАНИЕ

Третий аргумент функции **accept** изначально был определен как **int *** (именно так это сделано в libc4, libc5 и во многих системах, включая BSD 4.*, SunOS и SGI); черновик стандарта POSIX 1003.1g пытался поменять этот тип на **size_t ***, и в SunOS 5 это именно так. Более поздние черновики POSIX содержат **socklen_t ***, и в Single Unix Specification и glibc2 это сделано таким образом. По словам Линуса Торвальдса:

В _любой_ разумной библиотеке размеры "socklen_t" и int _должны_ совпадать. Любой другой вариант несовместим с реализацией сокетов BSD. В POSIX сначала использовали size_t, но я (и, к счастью, кто-то еще, хотя и не слишком многие) сильно возмутились по этому поводу. Такая реализация полностью поломана как раз потому, что size_t очень редко имеет тот же размер, что и "int", например, в 64-битных архитектурах. Это необходимо потому, что интерфейс сокетов BSD таков, каков он есть. В любом случае, люди из POSIX наконец поняли и создали "socklen_t". Вообще, с самого начала они просто не должны были ничего трогать, но по какой-то причине они чувствовали, что должны использовать поименованный тип (вероятно, они не хотели ударить в грязь лицом, сделав глупость, поэтому они тихо переименовали свою грубую

ошибку).

СМ. ТАКЖЕ **bind(2), connect(2), listen(2), select(2), socket(2)**

Linux 2.2 Page 2002-01-17 ACCEPT**(2)**

ACCESS

НАЗВАНИЕ access - проверка прав пользователя для доступа к файлу

СИНТАКСИС

#include <unistd.h>

int access(const char ***_pathname_, int** _mode_**);**

ОПИСАНИЕ

access проверяет, имеет ли процесс права на чтение или запись, или просто проверяет, существует ли файл (или другой объект файловой системы), путь к которому задается переменной <IR> pathname . Если *pathname* является символьной ссылкой, то проверяются права доступа к файлу, на который она указывает. *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** используются для проверки, существует ли файл и можно ли его читать, записывать в него или выполнять, соответственно. **F_OK** просто проверяет существование файла. Результаты проверки зависят от прав доступа к каталогам, находящимся по пути к файлу, заданному параметром <IR> pathname , и от прав доступа к каталогам и файлам, на которые указывают символьные ссылки. Проверка осуществляется, используя <IR> реальные , а не фактические идентификаторы пользователя и группы. Фактические идентификаторы будут использоваться при действительной попытке выполнения операции. Это дает программам setuid простой способ проверить права доступа реального пользователя. Проверяются только биты доступа (без проверки типа файла или содержимого). Таким образом, если имеется возможность записи в каталог, это, вероятно, означает то, что в нем можно создавать файлы, а не то,что в этот каталог можно писать так же, как в обычный файл. Например, файл из DOS может оказаться "исполняемым", но системный вызов **execve(2)** завершится неудачно. Если процесс имеет соответствующие привилегии, то реализация может выдать успех для **X_OK** даже если не установлен ни один из битов, разрешающих исполнение файлов.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном вызове функции (даны все требуемые разрешения), возвращается нулевое значение. При ошибке (по крайней мере один бит в запросе *mode* неудовлетворен или произошла другая ошибка), возвращается -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EACCES Запрошенный тип доступа не удовлетворен или один из каталогов в *pathname* не позволяет произвести поиск.

EROFS Были запрошены права доступа на запись для файла, находящегося на файловой системе, смонтированной только для чтения.

EFAULT *pathname* указывает за пределы доступного адресного пространства.

EINVAL *mode* задан неверно.

ENAMETOOLONG Слишком длинный путь - *pathname*

ENOENT Компонент пути *pathname* не существует или является символьной ссылкой, которая указывает на несуществующий файл или каталог.

ENOTDIR

Компонент пути, использованный как каталог в *pathname* фактически не является каталогом.

ENOMEM Недостаточно памяти.

ELOOP <IR> pathname . является зацикленной символьной ссылкой, то есть при подстановке возникает ссылка на неё саму.

EIO Ошибка ввода-вывода.

ОГРАНИЧЕНИЯ

access возвращает ошибку, если один из запрошенных типов доступа не будет удовлетворен, даже если другие типы прошли бы успешно.

access может работать неверно на файловых системах NFS со включенным преобразованием UID, потому что это преобразование происходит на сервере и спрятано от клиента, который пытается проверить права.

Использование **access** для проверки (например, можно ли пользователю открыть файл перед выполнением **open(2)**) создает бреши в защите, потому что пользователь может в короткий промежуток между проверкой и открытием файла как-то его изменить.

СООТВЕТСТВИЕ СТАНДАРТАМ

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

СМ. ТАКЖЕ stat(2), open(2), chmod(2), chown(2), setuid(2), setgid(2)

Linux 2001-10-16 ACCESS(2)

ACCT

НАЗВАНИЕ acct - включение/выключение режима сбора статистической информации о процессах

СИНТАКСИС

#include <unistd.h>

int acct(const char * *filename*);

ОПИСАНИЕ

Режим сбора статистики включается, когда в качестве аргумента для функции передано имя файла. При завершении работы полученные данные добавляются к файлу *filename*. Если в качестве аргумента передано значение **NULL**, то режим сбора статистики выключается.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении возвращается ноль. Если происходит ошибка, то возвращается -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EACCES Нет прав на запись в указанный файл.

EACCES Аргумент *filename* не является обычным файлом.

EFAULT *filename* указывает на недостижимое для Вас пространство адресов.

EIO Ошибка записи в файл *R filename .*

EISDIR *filename* является каталогом.

ELOOP Обнаружено слишком много символьных ссылок при разрешении *filename*.

ENAMETOOLONG *filename* слишком длинное.

ENOENT Указанное имя файла не существует.

ENOMEM Недостаточно памяти.

ENOSYS Режим сбора статистики процессов BSD не был активирован при компиляции ядра операционной системы. Параметром ядра, который управляет этим значением, является CONFIG_BSD_PROCESS_ACCT.

ENOTDIR

Компонент, используемый как каталог в *filename*, фактически не является каталогом.

EPERM Вызываемый процесс не имеет прав включить режим сбора статистики.

EROFS *filename* обращается к файлу на файловой системе, доступной только для чтения.

EUSERS Нет свободных файловых структур, или нет свободной памяти.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4 (но не POSIX). SVr4 описывает коды ошибок EBUSY, но не EISDIR или ENOSYS. Также AIX и HPUX описывает EBUSY (попытка включить режим сбора статистики, когда этот режим

уже включен), аналогично Solaris (попытка включить режим сбора статистики для уже используемого имени).

ЗАМЕЧАНИЯ

Сбор статистики не ведется для программ, запущенных во время сбоя системы. В частности, сбор статистики не ведется для непрерывающихся программ.

Linux 2.1.126 4 November 1998 ACCT**(2)**

ADJTIMEX

НАЗВАНИЕ adjtimex - функция корректировки системных часов

СИНТАКСИС **#include <sys/timex.h>**

int adjtimex(struct timex ****buf*);**

ОПИСАНИЕ

Linux использует алгоритм, изобретенный David L. Mills для корректировки времени на системных часах. Системный вызов **adjtimex** считывает и устанавливает параметры временных корректировок.

adjtimex Он создает ссылку на структуру *timex*, обновляет параметры ядра с помощью своих значений и возвращает программе ту же структуру с текущими значениями ядра. Структура описывается следующим способом:

```
       struct timex {
           int modes;           /* переключатель режимов
                                   (mode selector) */
           long offset;         /* смещение (мксек)
                                   (time offset) */
           long freq;           /* смещение частоты
                                   (frequency offset) */
           long maxerror;       /* наибольшая ошибка
                                   (maximum error) */
           long esterror;       /* величина коррекции мксек
                                   (estimated error) */
           int status;          /* команда/статус часов
                                   (clock command/status) */
           long constant;       /* константа времени pll
                                   (pll time constant) */
           long precision;      /* точность часов мксек (только для
чтения)
                                   (clock precision) */
           long tolerance;      /* допуск смещения частоты часов
                                   (clock frequency tolerance (ppm))
*/
           struct timeval time; /* текущее время
                                   (current time) */
           long tick;           /* количество мксекунд между
колебаниями часов
                                   (usecs between clock ticks) */
       };
```

Поле *modes* определяет, какие параметры часов (при их наличии) необходимо установить. Оно может быть равно нулю или может содержать комбинацию следующих битов:
Начинающим пользователям рекомендуется оперировать нулевыми значениями *mode* . Опытные пользователи могут устанавливать различные параметры работы часов.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

После успешного завершения операции **adjtimex** возвращает следующие значения часовых установок:
При ошибке **adjtimex** возвращает значение -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EFAULT *buf* указывает на участок памяти, недоступный для записи.

EPERM Значение *buf.mode* не равно нулю, и пользователь не является суперпользователем.

EINVAL Делается попытка установить значение *buf.offset*,

выходящее за пределы от -131071 до +131071, или значение *buf.status*, выходящее за вышеописанные пределы, а также значение *buf.tick*, выходящее за пределы от 900000/**HZ** до 1100000/**HZ**, где **HZ** - частота прерываний в работе системных часов.

СООТВЕТСТВИЕ СТАНДАРТАМ

Команда **adjtimex** является специфической (только для Linux) и не должна запускаться в программах, переносимых на другие платформы. В SVr4 существует похожая команда, но она является более специализированной.

СМ. ТАКЖЕ **settimeofday(2)**

Linux 2.0 30 July 1997 ADJTIMEX**(2)**

ALARM

НАЗВАНИЕ alarm - функция, настраивающая таймер на подачу сигнала

СИНТАКСИС **#include <unistd.h>**

unsigned int alarm(unsigned int *seconds***);**

ОПИСАНИЕ **alarm** настраивает таймер на подачу сигнала **SIGALRM** процессу через *seconds* секунд. Если значение *seconds* равно нулю, то сигнал **alarm** не будет послан. В любом случае, предыдущее значение **alarm** не запоминается.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

alarm возвращает число секунд, оставшихся до запланированного сигнала, или 0, если сигнал не был задан.

ЗАМЕЧАНИЯ Функции **alarm** и **setitimer** используют один и тот же таймер; запросы к одной из из них будут посланы и к другой.

Значение **sleep()** может быть задано при помощи **SIGALRM**; одновременные запросы к **alarm()** и **sleep()** могут привести к сбою и того, и другого. Задержки в подаче сигнала могут замедлить выполнение процессов на заранее определенное количество времени.

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, SVID, POSIX, X/OPEN, BSD 4.3

СМ. ТАКЖЕ **setitimer(2), signal(2), sigaction(2), gettimeofday(2), select(2), pause(2), sleep(3)**

Linux 21 July 1993 ALARM**(2)**

BDFLUSH

НАЗВАНИЕ bdflush - запуск, сброс или настройка демона, записывающего информацию из буфера памяти на диск

СИНТАКСИС

int bdflush(int *func*, **long ***address**);**

int bdflush(int *func*, **long** *data***);**

ОПИСАНИЕ **bdflush** предназначен для запуска, сброса или настройки демона, записывающего информацию из буфера памяти на диск. Вызов **bdflush** может быть сделан только суперпользователем.

Если значение *func* отрицательное или равно 0, а демон не запущен, тогда **bdflush** запускает демона и не возвращается.

Если значение *func* равно 1, информация некоторых буферов будет записана на диск.

Если значение *func* больше или равно двум и является четным числом (младший бит равен 0), тогда *address* является длинным целым числом, а параметр настройки с номером (*func*-2)/2 возвращается вызывавшему по этому адресу.

Если значение *func* больше либо равно 3 и является нечетным числом (младший бит равен 1), тогда *data* является длинным словом, а параметр настройки с номером (*func*-3)/2 приобретает данное значение.

Набор параметров, их значений и диапазонов этих значений определяется в файле *fs/buffer.c*.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Если *func* отрицательна или равна нулю и демон успешно запустился, то **bdflush** никогда не возвращается. В случае успешного выполнения задания возвращаемое значение равно нулю, а в случае ошибки оно равно -1, и переменной *errno*

присваивается соответствующее значение.

КОДЫ ОШИБОК

EPERM Функция вызвана пользователем, который не обладает правами суперпользователя.

EFAULT *address* находится за пределами доступного адресного пространства.

EBUSY Попытка запустить уже запущенный демон.

EINVAL Попытка прочитать или записать параметр с неверным номером или записать неверное его значение.

СООТВЕТСТВИЕ СТАНДАРТАМ

bdflush является функцией, предназначенной только для работы в Linux, поэтому не должна использоваться в программах, переносимых на другие платформы.

СМ. ТАКЖЕ **fsync(2), sync(2), update(8), sync(8)**

Linux 1.2.4 15 April 1995 BDFLUSH**(2)**

BIND

НАЗВАНИЕ bind - функция создания имени сокета

СИНТАКСИС

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

int bind(int sockfd, struct sockaddr *my_addr, socklen_t
addrlen);
```

ОПИСАНИЕ **bind** присваивает сокету *sockfd* локальный адрес *my_addr*. *addrlen* - это длина структуры *my_addr*. Традиционно эта операция называется "присвоение сокету имени." Когда сокет только что создан с помощью **socket(2)**, он существует в пространстве имён (семействе адресов), но не имеет своего имени.

Обычно сокету типа **SOCK_STREAM** требуется присвоить локальный адрес с помощью **bind** перед тем, как он сможет участвовать в соединении (см. **accept(2)**).

Правила, которые используются при создании имён, отличны друг от друга в разных семействах адресов. За более подробной информацией обратитесь к соответствующему разделу под номером 7 руководства. Информацию об **AF_INET** читайте в **ip(7)**, об **AF_UNIX** - в **unix(7)**, об **AF_APPLETALK** - в **ddp(7)**, об **AF_PACKET** - в **packet(7)**, об **AF_X25** - в **x25(7)**, а об **AF_NETLINK** - в **netlink(7)**.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном выполнении возвращаемое значение становится равным нулю. При ошибке оно равно -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EBADF *sockfd* - неверный описатель.

EINVAL Сокет уже имеет определенный адрес. Эта ошибка в будущем может не выводиться (смотрите *linux/unix/sock.c* для уточнения деталей).

EACCES Адрес защищен, или пользователь не является суперпользователем.

ENOTSOCK Аргумент является описателем файла, а не сокета.

Следующие ошибки специфичны для сокетов домена UNIX (**AF_UNIX**):

EINVAL (параметр *addrlen* неверен, или сокет не находится в домене **AF_UNIX**);

EROFS (попытка создать сокет-файл в файловой системе "только для чтения");

EFAULT (*my_addr* находится за пределами доступного адресного пространства);

ENAMETOOLONG

(адрес *my_addr* является слишком длинным);
ENOENT (файла не существует);
ENOMEM (недостаточно памяти);
ENOTDIR
(начальный компонент полного имени (пути) файла не является названием каталога);
EACCES (запрещен поиск в одном из каталогов, указанных в пути);
ELOOP (слишком много символьных ссылок составляют *R my_addr)*.

НАЙДЕННЫЕ ОШИБКИ

Не описаны опции, связанные с работой "прозрачного" прокси.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (функция **bind** впервые появилась в BSD 4.2). SVr4 описывает дополнительные коды ошибок **EADDRNOTAVAIL**, **EADDRINUSE**, и **ENOSR**; а также дополнительные коды ошибок в домене Unix: **EIO** и **EISDIR**.

ЗАМЕЧАНИЯ

Третий аргумент вызова **bind** в действительности имеет тип **int** (это справедливо для BSD 4.*, libc4 и libc5). В существующем socklen_t присутствуют некоторые ошибки POSIX. См. также **accept(2)**.

СМ. ТАКЖЕ accept(2), connect(2), listen(2), socket(2), getsockname(2), ip(7), socket(7)

Linux 2.2 3 Oct 1998 BIND**(2)**

BRK

НАЗВАНИЕ brk, sbrk - функции, изменяющие размер сегмента данных

СИНТАКСИС

```
#include <unistd.h>
int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

ОПИСАНИЕ brk устанавливает величину окончания сегмента данных, равную *end_data_segment*, когда это значение приемлимо, система выделяет достаточно памяти и процесс не превышает свой максимальный размер сегмента данных (см **setrlimit(2)**).

sbrk увеличивает область данных программы на *increment* байтов. **sbrk** не является системным вызовом, она всего лишь часть библиотеки C. Вызов **sbrk** со значением *increment* равным 0, может быть использовано, чтобы найти текущую позицию прерывания программы.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова значение **brk** равно нулю, а **sbrk** устанавливает указатель на начало нового сегмента. При ошибке возвращается значение -1, а переменной *errno* присваивается значение **ENOMEM**.

СООТВЕТСТВИЕ СТАНДАРТАМ

BSD 4.3 **brk** и **sbrk** не определяются стандартом C и заимствованы из стандарта POSIX.1 (см. разделы B.1.1.1.3 и B.8.3.3).

СМ. ТАКЖЕ execve(2), getrlimit(2), malloc(3)

Linux 0.99.11 21 July 1993 BRK**(2)**

CACHEFLUSH

НАЗВАНИЕ cacheflush - функция, сбрасывающая содержимое инструкций и/или данные кэша

СИНТАКСИС

```
#include <asm/cachectl.h>
int cacheflush(char *addr, int nbytes, int cache);
```

ОПИСАНИЕ cacheflush сбрасывает содержимое указанного кэша (кэшей) на адрес пользователя в диапазоне от значения addr до

addr+количество байтов-1 (addr+nbytes-1). Сброс кэша может быть следующих типов:

ICACHE (сбрасывет инструкции кэша);

DCACHE (записывает в память содержимое, таким образом, оно теряет свое значение);

BCACHE (то же, что и **(ICACHE|DCACHE)).**

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

cacheflush при удачной работе возвращаемое значение равно 0, а при ошибке возвращается -1. Если произойдут ошибки, то переменной *errno* будет присвоено значение, указывающее на их тип.

КОДЫ ОШИБОК

EINVAL (параметры кэша не являются одними из ICACHE, DCACHE, или BCACHE);

EFAULT некоторые или все адреса из промежутка от addr до addr+количество байтов-1 (addr+nbytes-1) недоступны.

НАЙДЕННЫЕ ОШИБКИ Текущее приложение не идентифицирует параметры addr и nbytes. Поэтому всегда очищается весь кэш.

ЗАМЕЧАНИЕ Этот системный вызов доступен лишь машинам, основанным на MIPS. Не рекомендуется его использовать в программах, созданных на других платформах.

Linux 2.0.32 27 June 95 CACHEFLUSH**(2)**

CAPGET

НАЗВАНИЕ capget, capset - ункции, устанавливающие/получающие возможности процесса

СИНТАКСИС

#undef _POSIX_SOURCE

#include <sys/capability.h>

int capget(cap_user_header_t *header*, **cap_user_data_t** *data***);**

int capset(cap_user_header_t *header*, **const cap_user_data_t** *data***);**

ОПИСАНИЕ В версии Linux 2.2 возможности пользователя root разделены. Каждый процесс имеет несколько возможностей, которые предопределяют остальные. Процесс наследует возможности, которые передаются через **execve(2)** и которые он может передать своему дочернему процессу.

Эти две функции являются непосредственным интерфейсом ядра для установления и получения возможностей процесса. Эти системные вызовы работают не только с Linux: ядро API может использовать и изменять эти функции (в частности, в формате типа **cap_user_*_t**) в каждой последующей его версии.

Интерфейсы **cap_set_proc(3)** и **cap_get_proc(3)** являются совместимыми с другими системами; используйте их по возможности в приложениях. Если Вы предпочитаете работать с расширениями Linux в приложениях, то для этого существуют другие интерфейсы: **capsetp(3)** и **capgetp(3).**

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном исполнении возвращаемое значение равно нулю. При ошибке оно равно -1, а *errno* принимает соответствующее значение.

КОДЫ ОШИБОК

EINVAL Один из аргументов был неправильным.

EPERM Была попытка добавить возможность в разрешенный набор, или добавить возможность, не являющуюся разрешенной, в действующие или наследуемые наборы.

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Переносной интерфейс для запроса возможностей и установления функций предоставлен библиотекой **libcap**, которую можно отыскать по адресу:

ftp://linux.kernel.org/pub/linux/libs/security/linux-privs

Linux 2.2 1999-09-09 CAPGET**(2)**

CHDIR

НАЗВАНИЕ chdir, fchdir - функции смены рабочего каталога

СИНТАКСИС

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

ОПИСАНИЕ **chdir** устанавливает текущий каталог, указанный в аргументе *path*.

fchdir работает так же, как **chdir**, только в качестве аргумента используется описатель файла.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

После успешного выполнения возвращаемое значение становится нулевым. При ошибке возвращаемое значение равно -1, а переменной *errno* присваивается код ошибки.

КОДЫ ОШИБОК

Наиболее общие коды ошибок для **chdir**:

EFAULT (указатель *path* ссылается на каталоги за пределами доступного адресного пространства);

ENAMETOOLONG (путь *path* является слишком длинным);

ENOENT (указанный файл не существует);

ENOMEM (недостаточно памяти);

ENOTDIR (часть *path* не является каталогом);

EACCES (запрещен поиск в одном из каталогов, находящихся на пути к *path*);

ELOOP (*path* является зацикленной символьной ссылкой, то есть при подстановке возникает ссылка на неё саму);

EIO (ошибка ввода-вывода);

Основные ошибки для **fchdir**:

EBADF (*fd* - неправильный описатель файла);

EACCES (в каталоге, заданном с помощью *fd*, процесс поиска запрещен).

ЗАМЕЧАНИЯ Прототип для **fchdir** доступен, только если определено **_BSD_SOURCE** (либо явно, либо неявно - не определением переменной _POSIX_SOURCE или компилированием с флагом -ansi). В зависимости от файловой системы в работе могут появляться и другие ошибки.

СООТВЕТСТВИЕ СТАНДАРТАМ

Системный вызов **chdir** совместим с SVr4, SVID, POSIX, X/OPEN, 4.4BSD. SVr4 содержит следующие дополнительные коды ошибок: EINTR, ENOLINK и EMULTIHOP,- но не содержит ENOMEM. POSIX.1 не содержит коды ошибок ENOMEM и ELOOP. X/OPEN не определяет коды ошибок EFAULT, ENOMEM и EIO. Системный вызов **fchdir** совместим с SVr4, 4.4BSD и X/OPEN. SVr4 описывает дополнительные коды ошибок: EIO, EINTR, и ENOLINK. X/OPEN определяет дополнительные коды ошибок: EINTR и EIO.

СМ. ТАКЖЕ **getcwd(3), chroot(2)**

Linux 2.0.30 21 August 1997 CHDIR**(2)**

CHMOD

НАЗВАНИЕ chmod, fchmod - функции, изменяющие права доступа к файлу

СИНТАКСИС

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

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

ОПИСАНИЕ

Функции изменяют режим доступа к файлу, заданному параметром *path* или описателем файла *fildes*. С помощью

операции *или* можно сразу задавать следующие режимы:

```
S_ISUID   04000 установить пользователю права на
          выполнение
S_ISGID   02000 установить группе права на
          выполнение
S_ISVTX   01000 бит принадлежности
S_IRUSR (S_IREAD)
          00400 чтение для владельца
S_IWUSR (S_IWRITE)
          00200 запись для владельца
S_IXUSR (S_IEXEC)
          00100 выполнение/поиск для владельца
S_IRGRP   00040 чтение для группы
S_IWGRP   00020 запись для группы
S_IXGRP   00010 выполнение/поиск для группы
S_IROTH   00004 чтение для остальных пользователей
S_IWOTH   00002 запись для остальных пользователей
S_IXOTH   00001 выполнение/поиск для остальных
          пользователей
```

Идентификатор эффективного пользователя (UID) процесса должен быть нулем или должен совпадать с UID файла. Если эффективный UID процесса не равен нулю, а группа-владелец файла не совпадает с фактическим GID процесса или одним из его дополнительных GID, то бит S_ISGID будет сброшен, но ошибки при этом не возникнет. В зависимости от файловой системы биты, установленные для выполнения операции пользователем и группой, могут быть сброшены, когда происходит запись информации в файл. В некоторых файловых системах только суперпользователь имеет возможность устанавливать бит принадлежности, который иногда имеет специальное значение. Для справки о бите принадлежности и об установки идентификаторов пользователя и группы на каталоги смотрите **stat(2)**. В файловых системах NFS отмена некоторых прав доступа немедленно повлияет на открытые файлы, потому что контроль доступа осуществляется сервером, а открытые файлы обрабатываются клиентом. Добавление новых прав доступа может произойти не сразу, если на клиенте запущено кэширование атрибутов.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

После успешного выполнения операции возвращаемое значение равно нулю. При ошибке оно равно -1, а переменной *errno* присваивается соответствующий код ошибки.

КОДЫ ОШИБОК Наиболее общие коды ошибок **chmod**:

EPERM (эффективный UID не совпадает с ID владельца файла и не равен нулю);

EROFS (файл находится в файловой системе, предназначенной только для чтения);

EFAULT (*path* указывает на каталог, находящийся за пределами доступного адресного пространства);

ENAMETOOLONG (полное имя *path* слишком длинное);

ENOENT (файла не существует);

ENOMEM (недостаточно памяти в системе);

ENOTDIR (часть *path* не является каталогом);

EACCES (запрещен поиск в одном из каталогов, который является компонентом *path*);

ELOOP (*path* является зацикленной символьной ссылкой, то есть при соответствующей подстановке возникает ссылка на неё саму);

EIO (ошибка ввода-вывода).

Основные коды ошибок **fchmod**:

EBADF (*fildes* - неверный файловый описатель);

EROFS (см. выше);

EPERM (см. выше);

EIO (см. выше).

В зависимости от файловой системы в работе могут появляться и другие ошибки.

СООТВЕТСТВИЕ СТАНДАРТАМ

Вызов **chmod** соответствует стандартам SVr4, SVID, POSIX, X/OPEN, 4.4BSD. SVr4 описывает EINTR, ENOLINK и EMULTIHOP, но не описывает ENOMEM. POSIX.1 не описывает ни коды ошибок EFAULT, ENOMEM, ELOOP и EIO, ни макросы **S_IREAD**, **S_IWRITE** и **S_IEXEC**. Вызов **fchmod** соответствует 4.4BSD и SVr4. SVr4 описывает дополнительные коды ошибок EINTR и ENOLINK. POSIX требует присутствия функции **fchmod**, если определены символы **_POSIX_MAPPED_FILES** или **_POSIX_SHARED_MEMORY_OBJECTS**, и описывает дополнительные коды ошибок ENOSYS и EINVAL, но не документирует EIO. POSIX и X/OPEN не описывают бит принадлежности.

СМ. ТАКЖЕ **open(2), chown(2), execve(2), stat(2)**

Linux 2.0.32 10 December 1997 CHMOD**(2)**

CHOWN

НАЗВАНИЕ chown, fchown, lchown - функции, изменяющие владельца файла

СИНТАКСИС

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

ОПИСАНИЕ Владелец файла,который задан параметром *path* или *fd*, будет изменен. Только суперпользователь может изменить владельца файла. Владелец файла может изменять группу файла на любую группу, к которой он принадлежит. Суперпользователь может произвольно производить эту замену. Если значение параметров *owner* или *group* равно -1, то соответствующий идентификатор не изменяется. Когда владелец или группа исполняемого файла изменяются не суперпользователем, то биты S_ISUID и S_ISGID будут сброшены. POSIX не требует, чтобы это происходило, когда суперпользователь выполняет функцию *chown*: в этом случае все зависит от версии ядра Linux. Если в правах доступа к файлу не установлен бит исполнения группой (S_IXGRP), то бит S_ISGID означает принудительную блокировку этого файла и не очищается функцией *chown*.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

После успешного выполнения действия возвращаемое значение равно нулю. При ошибке оно равно -1, а переменной *errno* присваивается соответствующий код ошибки.

КОДЫ ОШИБОК Основные коды ошибок **chown**:

EPERM (фактический UID не совпадает с номером владельца файла и не равен нулю; параметры *owner* или *group* заданы неверно);

EROFS (файл находится в файловой системе, предназначенной только для чтения);

EFAULT (*path* указывает на каталог за пределами доступного адресного пространства);

ENAMETOOLONG
(полное имя *path* является слишком длинным);

ENOENT (файла не существует);

ENOMEM (недостаточно памяти в системе);

ENOTDIR
(компонент *path* не является каталогом);

EACCES (запрещен поиск в одном из каталогов, являющихся компонентом пути);

ELOOP (*path* является зацикленной символьной ссылкой, то есть при соответствующей подстановке возникает ссылка на неё саму).

Основные ошибки **fchown**:

EBADF (неправильный описатель файла);
ENOENT (см. выше);
EPERM (см. выше);
EROFS (см. выше);
EIO (при модификации inode произошла ошибка низкоуровневого ввода-вывода).

ЗАМЕЧАНИЯ В версиях Linux до 2.1.81 кроме 2.1.46)**chown** не проходит по символьным ссылками. Начиная с версии Linux 2.1.81, **chown** проходит по символьным ссылкам; кроме того, существует новый системный вызов **lchown**, который не следует по символьным ссылкам. Начиная с Linux 2.1.86, этот новый вызов (имеющий ту же семантику, что и старый **chown**), имеет тот же самый номер системного вызова, а **chown** получил новый номер.

Прототип **fchown** доступен, только если определено **_BSD_SOURCE** (либо явно, либо неявно - не определением переменной _POSIX_SOURCE или компилированием с флагом -ansi).

СООТВЕТСТВИЕ СТАНДАРТАМ

Вызов **chown** соответствует SVr4, SVID, POSIX, X/OPEN. В версии 4.4BSD его может делать только суперпользователь (то есть обычные пользователи не могут передавать файлы). SVr4 описывает EINVAL, EINTR, ENOLINK и EMULTIHOP, но не документирует ENOMEM. POSIX.1 не описывает ENOMEM и ELOOP.

Вызов **fchown** соответствует 4.4BSD и SVr4. SVr4 описывает дополнительные коды ошибок EINVAL, EIO, EINTR и ENOLINK.

ОГРАНИЧЕНИЯ

Семантика **chown**() специально нарушается в файловых системах NFS с разрешенным преобразованием UID. Вдобавок нарушается семантика всех системных вызовов к содержимому файла, потому что выполнение **chown**() может привести к немедленному запрету доступа к уже открытым файлам. Кэширование на клиенте может привести к задержке между сменой владельца и истинным моментом, когда этот пользователь сможет обратиться к файлу на других клиентах.

СМ. ТАКЖЕ **chmod(2), flock(2)**

Linux 2.1.81 May 18, 1997 CHOWN**(2)**

CHROOT

НАЗВАНИЕ chroot - функция установки нового корневого каталога

СИНТАКСИС

```
#include <unistd.h>
int chroot(const char *path);
```

ОПИСАНИЕ chroot изменяет имя корневого каталога на указанный в аргументе *path*. Этот каталог - для файлов, имена которых начинаются со знака "/". Корневой каталог наследуется всеми "потомками" текущего процесса. Только суперпользователь может изменять корневой каталог. Заметьте, что этот системный вызов не изменяет текущий рабочий каталог, поэтому "." может находиться вне дерева каталогов, имя которого начинается со знака "/". В частности, суперпользователь может выйти из созданного корневого каталога, выполнив следующее: `mkdir foo; chroot foo; cd ..'.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

После успешного выполнения действия возвращаемое значение равно нулю. При ошибке возвращаемое значение равно -1, а переменной *errno* присваивается соответствующий код ошибки.

КОДЫ ОШИБОК Самые распространенные ошибки **chroot**:

EPERM (занчение эффективного UID не равено нулю);
EFAULT (*path* указывает на каталог за пределами доступного адресного пространства);
ENAMETOOLONG (имя *path* является слишком длинным);
ENOENT (указанного файла не существует);

ENOMEM (недостаточно памяти в системе);
ENOTDIR (часть имени *path* не является каталогом);
EACCES (запрещен поиск в одном из каталогов, который является компонентом *path*);
ELOOP (*path* является зацикленной символьной ссылкой, то есть при соответствующей подстановке возникает ссылка на нее саму);
EIO (ошибка ввода-вывода).
В зависимости от файловой системы в работе могут появляться и другие ошибки.

СООТВЕТСТВИЕ СТАНДАРТАМ
SVr4, SVID, 4.4BSD, X/OPEN. Эта функция не входит в POSIX.1. SVr4 описывает дополнительные коды ошибок EINTR, ENOLINK и EMULTIHOP. X/OPEN не описывает EIO, ENOMEM и EFAULT. Этот интерфейс помечен как "устаревающий" в X/OPEN.

СМ. ТАКЖЕ **chdir(2)**

CLONE

НАЗВАНИЕ clone - функция, создающая дочерний процесс

СИНТАКСИС

```
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack, int flags,
void *arg);
_syscall2(int, clone, int, flags, void *, child_stack);
```

ОПИСАНИЕ **clone** создает новый процесс аналогично **fork(2)**. **clone** является библиотечной функцией для системного вызова **clone**, переправляющего на **sys_clone**. Описание **sys_clone** приведено далее по тексту. В отличие от **fork(2)**, эти вызовы позволяют дочернему процессу разделять такие части его исполняемого содержания с вызывающими процессами, как память, таблица описателей файлов, таблица обработчиков сигналов. (Заметим, что в данном руководстве "вызывающий процесс" обычно соотвествует "родительскому процессу". Но обязательно посмотрите описание **CLONE_PARENT** далее по тексту.) Основное использование **clone** - это создание множества подпроцессов программы, которые работают в общем пространстве памяти. Когда дочерний процесс создан **clone**, он запускает функцию приложения *fn*(*arg*). (Это отличается от поведения **fork(2)**, где исполнение продолжается в дочернем процессе с точки разделения вызова **fork(2)**. Аргумент *fn* указывает на функцию, вызываемую дочерним процессом в начале работы. Аргумент *arg* передается функции *fn*. Когда функция *fn*(*arg*) выполнена, дочерний процесс закрывается. Целое число, возвращаемое *fn*, является для дочерней программы кодом выхода. Дочерний процесс можно закрыть при помощи **exit(2)**; также он может закрыться при получении сигнала о прерывании работы (fatal signal). Аргумент *child_stack* указывает на местонахождение стека, использованного дочерним процессом. Хотя дочерний и родительский процессы могут разделять память, дочернему процессу нельзя ипспользовать тот-же стек для исполнения, что и вызывающему процессу. Поэтому родительский процесс должен выделить пространство в памяти под стек дочернего процесса и передать указатель на это пространство функции **clone**. На всех процессорах с реализацией Linux (кроме процессоров HP PA) стек "растет вниз". *child_stack* обычно указывает на самый последний адрес памяти для стека дочернего процесса. Последний байт из *flags* содержит номер сигнала, который посылается родительскому процессу при закрытии дочернего. Если этот сигнал отличается от **SIGCHLD**, то родительский процесс должен указать опции **__WALL** или **__WCLONE** для ожидания дочернего процесса через **wait(2)**. Если не указан сигнал,

то родительскому процессу не сообщается о завершении дочернего процесса. Значение *flags* может быть логически сложено побитно, с одной или несколькими константами, для определения того, что является общим между вызывающим и дочерним процессами:

CLONE_PARENT

(Linux 2.4 и далее) Если **CLONE_PARENT** установлен, то родитель нового процесса (возвращаемый от **getpid(2)**) будет такой же, как и у вызывающего процесса. Если значение **CLONE_PARENT** не установлено, то (аналогично **fork(2)**) родитель процесса есть вызывающий процесс.
Заметим, что имменно родительский процесс, как возвращается от **getppid(2)**, выдает сигнал при завершении работы дочернего процесса, поэтому если **CLONE_PARENT** установлен, то сигнализировать будет родитель вызывающего процесса, а не сам вызывающий процесс.

CLONE_FS

Если значение **CLONE_FS** указано, то вызывающий и дочерний процессы используют общую информацию файловой системы. Это касается как корневого каталога, так и рабочего и umask. Любой вызов: **chroot(2)**, **chdir(2)** или **umask(2)**, производится вызывающим или дочерним процессами и отражается с одного на другой. Если значение **CLONE_FS** не указано, то дочерний процесс работает с копией информации файловой системы вызывающего процесса, сделанной сразу после вызова **clone**. Вызовы **chroot(2)**, **chdir(2)**, **umask(2)**, сделанные после этого другими процессами, не влияют на остальные процессы.

CLONE_FILES

Если значение **CLONE_FILES** указано, то вызывающий и дочерний процессы используют один файл с таблицей описателей. Описатели файлов ссылаются на файлы, используемые как вызывающим, так и дочерним процессом. Любой описатель файла, созданный одним из процессов, доступен другому процессу. Если один из процессов закрывает описатель файла или меняет его флаги, то это отражается на другом процессе. Если значение **CLONE_FILES** не указано, то дочерний процесс получает копию всех описателей файла, открытых в вызывающем процессе во время **__clone**. Операции, производимые далее над описателями либо вызывающим, либо дочерним процессом, не влияют на другие процессы.

CLONE_SIGHAND

Если значение **CLONE_SIGHAND** указано, то вызывающий и дочерний процессы используют общую таблицу обработчиков сигнала. Если один из процессов вызывает **sigaction(2)**, чтобы сменить свою реакцию на сигнал, то реакция изменится и у второго. Так или иначе, родительский и дочерний процессы имеют различные маски сигналов и набор ожидаемых сигналов. Один из процессов при задании команды **sigprocmask(2)** может заблокировать или разблокировать сигналы для себя (это не отразится на другом процессе). Если **CLONE_SIGHAND** не указано, то дочерний процесс получает копию таблицы обработчиков сигнала от вызвавшего процесса при вызове **clone** . После этого вызов **sigaction(2)**, сделанный одним из процессов, не будет влиять на другой.

CLONE_PTRACE

Если **CLONE_PTRACE** указано, и отслеживается вызывающий процесс, то отслеживаться будет также и

дочерний процесс (см. **ptrace(2)**).

CLONE_VFORK
Если **CLONE_VFORK** установлен, то выполнение вызывающего процесса задерживается, пока дочерний процесс не раздаст свои виртуальные ресурсы памяти через вызов **execve(2)** или **_exit(2)** (аналогично **vfork(2)**).
Если **CLONE_VFORK** не установлено, то и вызывающий и дочерний процессы после вызова будут распределены, а приложение не должно полагаться на выполнение в каком-либо порядке.

CLONE_VM
Если значение **CLONE_VM** установлено, то вызывающий и дочерний процессы работают в одном пространстве памяти. В частности, записи, сделанные в памяти вызывающим или дочерним процессом, видны из других процессов. Более того, если родительский или дочерний процессы отражают информацию (или снимают ее отражение) с помощью функций **mmap(2)** или **munmap(2)**, то отраженная информация будет влиять на все остальные процессы.
Если **CLONE_VM** не установлено, то дочерний процесс запускается в отдельной копии пространства памяти вызывающего процесса во время **clone**. Записи в память или отражение/снятие отражения выполняемые одним из процессов не влияет на другие, аналогично **fork(2)**.

CLONE_PID
Если значение **CLONE_PID** указано, то дочерний процесс создан с тем же уникальным идентификатором процесса (PID), что и вызывающий процесс. Если значение **CLONE_PID** не указано, то дочерний процесс будет обладать уникальным идентификатором процесса (PID), отличным от вызвавшего его процесса.
Этот флаг может быть указан только системным процессом загрузки (PID 0).

CLONE_THREAD
(Linux 2.4 и далее) Если **CLONE_THREAD** установлено, то дочерний процесс помещается в ту же группу задач, что и вызывающий процесс.
Если **CLONE_THREAD** не установлено, то дочерний процесс помещается в его собственную группу задач, чей идентификатор равен идентификатору процесса.
(Группы задач /- это особенность, добавленная в Linux 2.4 для поддержки понятия задач POSIX, как набора задач, имеющих одинаковый идентификатор процесса. В Linux 2.4, вызовы **getpid(2)** возвращают идентификатор группы задач вызывающего процесса.)

Системный вызов **sys_clone** больше соответствует **fork(2)** в том, что выполнение в дочернем процессе продолжается с точки вызова. Поэтому **sys_clone** требует наличия только аргументов *flags* и *child_stack*, имеющих аналогичные значения и для **clone**. (Заметим, что порядок этих аргументов отличается от **clone**.)
Другое отличие **sys_clone** заключается в том, что аргументом *child_stack* может быть ноль, в этом случае семантика копирования-при-записи проверяет, что дочерний процесс получает отдельные копии страниц стека, если процесс изменяет стек. В этом случае для корректной работы параметр **CLONE_VM** не должен указываться.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ
При удачном завершении работы вызывающему процессу возвращается PID дочернего процесса. При ошибке вместо PID вызывающему процессу возвращается -1, дочерний процесс не создается, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EAGAIN Запущено слишком много процессов.

ENOMEM Невозможно выделить достаточно памяти для размещения в ней структуры задачи для дочернего процесса, или для копирования тех частей вызывающего процесса, что должны быть скопированы.

EINVAL Возвращается от **clone** при нулевых значениях, указанных для *child_stack*.

EPERM **CLONE_PID** было указано процессом с ненулевым идентификатором процесса.

НАЙДЕННЫЕ ОШИБКИ

Как и в версии ядра 2.1.97, флаг **CLONE_PID** не должен быть использован, так как другие части ядра и системы рассматривают PID как уникальный. Нет функции **clone** в libc версии 5. В libc версии 6 (как и в glibc 2) **clone** работает так, как описано в этой инструкции.

СООТВЕТСТВИЕ СТАНДАРТАМ

Вызовы **clone** и **sys_clone** являются специфичными только для Linux, поэтому их не рекомендуется использовать в программах, предназначенных для последующего портирования. Для программирования многозадачных приложений (работающих в одном адресном пространстве) лучше всего использовать библиотеки, имеющие API стандарта POSIX 1003.1c, такие, как LinuxThreads library (включенную в glibc2). Просмотрите **pthread_create(3)**. Это руководство соответствует ядрам 2.0.x, 2.1.x, 2.2.x, 2.4.x и glibc 2.0.x и 2.1.x.

СМ. ТАКЖЕ **fork(2)**, **wait(2)**, **pthread_create(3)**

Linux 2.4 2001-06-26 CLONE**(2)**

CLOSE

НАЗВАНИЕ close - функция, закрывающая описатель файла

СИНТАКСИС

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

ОПИСАНИЕ **close** закрывает описатель файла, который после этого не указывает ни на один файл и может быть использован повторно. все блокировки соответствующего файла снимаются (независимо от того, был ли использован для блокировки именно этот файловый описатель).

Если *fd* является последней копией какого-либо файлового описателя, то ресурсы, связанные с ним, освобождаются; если описатель был последней ссылкой на файл, удаленный с помощью **unlink(2)**, то файл окончательно удаляется.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

После успешного выполнения **close** возвращаемое значение становится равным нулю, а в случае ошибки оно равно -1.

КОДЫ ОШИБОК

EBADF (*fd* является неверным файловым описателем).

EINTR Вызов **close**() был прерван сигналом.

EIO Ошибка ввода-вывода.

СООТВЕТСТВИЕ СТАНДАРТАМ

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

ЗАМЕЧАНИЯ

Не проверять значение, возвращаемое функцией **close**, /-/- обычная, но от этого не менее серьезная ошибка программирования. Весьма вероятно, что ошибки предыдущего вызова **write(2)** проявятся только при вызове конечного **close**. Если значение, возвращаемое при закрытии файла, не проверяется, то данные могут быть незаметно потеряны. В особенности это может проявлятся при работе с NFS и дисковыми квотами.

Успешное закрытие файла не гарантирует, что данные будут

сохранены на диске, т.к. ядро задерживает процесс записи. В файловой системе информация буфера не всегда записывается на диск после закрытия потока. Если Вам необходимо убедится, что данные физически сохранены на диске, то используйте **fsync(2).** (это зависит также и от вашего дискового оборудования.)

СМ. ТАКЖЕ open(2), fcntl(2), shutdown(2), unlink(2), fclose(3), fsync (2)

2001-12-13 CLOSE(**2**)

CONNECT

НАЗВАНИЕ connect - функция, устанавливающая соединение с сокетом

СИНТАКСИС

#include <sys/types.h>
#include <sys/socket.h>
int connect(int *sockfd*, **const struct sockaddr ***_serv_addr_, **socklen_t** *addrlen*);

ОПИСАНИЕ Файловый описатель *sockfd* должен указывать на какой-либо сокет. Если сокет имеет тип **SOCK_DGRAM**, значит, адрес *serv_addr* по умолчанию является тем адресом, на который посылаются датаграммы, и единственным адресом, откуда они принимаются. Если сокет имеет тип **SOCK_STREAM** или **SOCK_SEQPACKET**, то этот вызов попытается установить соединение с другим сокетом. Другой сокет задан параметром *serv_addr*, являющийся адресом (длиной *addrelen*) в пространстве соединений сокета. Каждое пространство коммуникации по своему интерпретирует параметр *serv_addr*. Обычно сокеты с протоколами, основанными на соединении, могут устанавливать соединение только один раз; сокеты с протоколами без соединения могут использовать **connect** многократно, чтобы изменить адрес назначения. Сокеты без поддержки соединения могут прекратить связь с другим сокетом, установив компонент *sa_family* структуры **sockaddr** в положение **AF_UNSPEC**.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Если соединение или "привязка" прошли успешно, возвращаемое значение равно нулю. При ошибке возвращаемое значение равно -1, а в *errno* записывается код ошибки.

КОДЫ ОШИБОК

Ниже описаны только основные ошибки сокетов. Могут также появляться коды ошибок, существующие в конкретном домене.

EBADF Файловый описатель имеет неверный индекс в таблице описателей.

EFAULT Адрес структуры сокета находится за пределами адресного пространства.

ENOTSOCK
Файловый описатель не связан с сокетом.

EISCONN
Соединение с сокетом уже произошло.

ECONNREFUSED
С другой стороны нет "слушающего" сокета.

ETIMEDOUT
Превышен период ожидания соединения. Сервер, возможно, слишком занят и не может принимать новые запросы. Заметьте, что для IP-сокетов время ожидания может быть очень долгим, если на сервере разрешено использование syncookies.

ENETUNREACH
Сеть недоступна.

EADDRINUSE
Локальный адрес уже используется.

EINPROGRESS
Сокет является неблокирующим, а соединение не может

быть установлено прямо сейчас. Можно использовать **select(2)** или **poll(2)**, чтобы закончить соединение, выбрав соответствующий сокет. После того, как **select** сообщит об этом, используйте **getsockopt(2)**, чтобы прочитать флаг **SO_ERROR** на уровне **SOL_SOCKET**. Это необходимо для того, чтобы определить, была ли завершена работа **connect** успешно (в этом случае, **SO_ERROR** равен нулю). При этом значение **SO_ERROR** равно одному из обычных кодов ошибок, перечисленных здесь. Он объясняет причину неудачи.

EALREADY
Сокет является неблокирующим, а предыдущая попытка установить соединение еще не завершилась.

EAGAIN Не осталось свободных локальных портов, или же недостаточно места в кэше маршрутизации. Для домена **PF_INET** смотри описание системной переменной **net.ipv4.ip_local_port_range** в **ip(7)**, где объясняется, как увеличить количество локальных портов.

EAFNOSUPPORT
Адрес находится в некорректном семействе адресов поля *sa_family*.

EACCES, EPERM
Пользователь попытался соединиться с широковещательным адресом, не установив соответствующий флаг на сокете, или же запрос на соединение завершился неудачно из-за локального правила firewall.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (функция **connect** впервые появилась в BSD 4.2). SVr4 описывает дополнительные коды ошибок: **EADDRNOTAVAIL**, **EINVAL**, **EAFNOSUPPORT**, **EALREADY**, **EINTR**, **EPROTOTYPE**, и **ENOSR**. Там также описывается множество дополнительных кодов ошибок, не описанных в этом документе.

ЗАМЕЧАНИЕ

Третий аргумент **connect** в действительности имеет тип *int* (это справедливо для BSD 4.*, libc4 и libc5). Определенное недопонимание привело к появлению *socklen_t*. Черновой вариант этого стандарта еще не принят, но в glibc2 уже содержится socklen_t. См. также **accept(2)**.

НАЙДЕННЫЕ ОШИБКИ

Еще не удалось разорвать соединения с сокетом, вызывая с помощью функции **connect** адрес **AF_UNSPEC**.

СМ. ТАКЖЕ accept(2), bind(2), listen(2), socket(2), getsockname(2).

Linux 2.2 3 Oct 1998 CONNECT**(2)**

DUP

НАЗВАНИЕ dup, dup2 - функции, создающие копию описателя файла

СИНТАКСИС

```
#include <unistd.h>

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

ОПИСАНИЕ dup и **dup2** создают копию файлового описателя *oldfd*. После успешного вызова функции **dup** или **dup2** старый описатель можно использовать вместо нового и наоборот. Они совместно блокируют файл, используют указатели позиции файла и флаги. Например, если позиция файла изменяется с помощью **lseek** в одном из описателей, то она изменяется также и в другом. Однако, два описателя имеют свой собственный флаг "close-on-exec". **dup** предоставляет новому описателю наименьший свободный номер. **dup2** делает *newfd* копией *oldfd*, закрывая *newfd*, если это необходимо.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

dup и **dup2** возвращают новый описатель или значение -1, если произошла ошибка (в этом случае переменной *errno* присваивается значение соответствующего кода ошибки).

КОДЫ ОШИБОК

EBADF *oldfd* не является открытым файловым описателем, или *newfd* находится за пределами допустимого диапазона файловых описателей.

EMFILE Процесс уже открыл максимальное количество файловых описателей и делает попытку открыть новый.

ПРЕДУПРЕЖДЕНИЕ Ошибка, которую возвращает **dup2**, отличается от той, которую возвращает **fcntl(..., F_DUPFD, ...)**, когда *newfd* находится за пределами диапазона. На некоторых системах **dup2** также иногда возвращает значение **EINVAL**, подобное **F_DUPFD**.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. SVr4 описывает дополнительные коды ошибок EINTR и ENOLINK. POSIX.1 содержит EINTR.

СМ. ТАКЖЕ **fcntl(2), open(2), close(2).**

Linux 1.1.46 21 August 1994 DUP(2)

EXECVE

НАЗВАНИЕ execve - функция, осуществляющая выполнение программы

СИНТАКСИС

```
#include <unistd.h>
int execve(const char *filename, char *const argv [], char
*const envp[]);
```

ОПИСАНИЕ

Функция **execve()** отвечает за выполнение программы, заданной параметром *filename*. Программа должна быть либо двоичным исполняемым файлом, либо скриптом, который начинается со строки вида "**#!** *интерпретатор* [аргументы]". В последнем случае интерпретатор - это правильный путь к исполняемому файлу, который не является скриптом; он будет вызван как **интерпретатор** [arg] *filename*. *argv* - массив аргументов (строк), передаваемый запускаемой программе. *envp* - массив строк в формате **параметр=значение**, передаваемый в качестве окружения запускаемой программе. Оба параметра: *argv* и *envp*,- должны завершаться нулевым указателем. К массиву аргументов и к окружению можно обратиться из вызываемой программы с помощью функции *main*, когда она определена как **int main (int** *argc*, **char *****argv*[], **char *****envp*[]). Значение **execve()** не возвращается при успешном выполнении программы, а код, данные, инициализированные данные и стек вызвавшего процесса записываются как код, данные и стек загруженной программы. Новая программа наследует от вызвавшего процесса его идентификатор и открытые файловые описатели, на которых не было флага "close-on-exec". Сигналы, ожидающие обработки, удаляются. Новое значение обработчиков сигналов становится их значением по умолчанию. Сигнал SIGCHLD, в зависимости от установки его в SIG_IGN, может быть сброшен в SIG_DFL. Если текущая программа выполнялась под управлением ptrace, то после успешного выполнения **execve()** ей посылается сигнал **SIGTRAP**. Если в файле программы *filename* установлен set-uid бит, то идентификатор эффективного пользователя вызывающего процесса меняется на идентификатор владельца файла программы. Точно так же, если на файле программы установлен set-gid бит, то идентификатор эффективной группы устанавливается в группе файла программы. Если исполняемый файл является динамически скомпонованным файлом в формате a.out, содержащим "заглушки" для вызова разделяемых библиотек, то

в начале выполнения этого файла вызывается динамический компоновщик **ld.so(8)**, который загружает библиотеки и "связывает" их с исполняемым файлом. Если исполняемый файл является динамически скомпонованным файлом в формате ELF, то для загрузки разделяемых библиотек используется интерпретатор, указанный в сегменте PT_INTERP. Обычно это */lib/ld-linux.so.1* для программ, скомпилированных для работы с Linux libc версии #5, или же */lib/ld-linux.so.2* для программ, скомпилированных для работы с GNU libc версии #2.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении работы значение **execve()** не возвращается, при ошибке возвращаемое значение равно -1, а переменной *errno* присваивается значение соответствующее коду ошибки.

КОДЫ ОШИБОК

EACCES (интерпретатор файла или скрипта не является обычным файлом);

EACCES (нет прав на работу с файлом, скриптом или ELF-интерпретатором);

EACCES (файловая система запущена с флагом *noexec*);

EPERM (файловая система запущена с флагом *nosuid*, пользователь не является суперпользователем, а в файле установлен бит SUID или SGID);

EPERM (процесс отлаживается, пользователь не является суперпользователем, а в файле установлен бит SUID или SGID);

E2BIG (слишком длинный список аргументов);

ENOEXEC(формат исполняемого файла неизвестен, файл предназначен для другой архитектуры или содержит какие-то ошибки, препятствующие его выполнению);

EFAULT (Файл *filename* указывает на файл за пределами доступного адресного пространства);

ENAMETOOLONG (Файл *filename* является слишком длинным);

ENOENT (1. Файла *filename*, скрипта или ELF-интерпретатора не существует. 2. Библиотека, которая используется файлом, или интерпретатор не найдены.);

ENOMEM (недостаточно памяти в системе);

ENOTDIR(компонент имени *filename*, скрипта или ELF-интерпретатора не является каталогом);

EACCES (нет прав на поиск одного из каталогов, являющегося компонентом *filename*, имени скрипта или ELF-интерпретатора);

ELOOP (слишком много символьных ссылок составляют *file-name*, имя скрипта или ELF-интерпретатора);

ETXTBSY(исполняемый файл открыт для записи одним или более процессами);

EIO (ошибка ввода-вывода);

ENFILE (достигнут лимит общего количества открытых файлов);

EMFILE (процесс уже открыл максимальное количество доступных файлов);

EINVAL (исполняемый ELF-файл содержит более одного сегмента PT_INTERP (то есть в нем указано более одного интерпретатора);

EISDIR (указанный ELF-интерпретатор является каталогом);

ELIBBAD (неизвестен формат ELF-интерпретатора).

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, SVID, X/OPEN, BSD 4.3. В POSIX не описывается поведение интерпретатора #!, но, в остальном, в ней все совершенно стандартно. SVr4
описывает дополнительные коды ошибок
EAGAIN, EINTR, ELIBACC, ENOLINK, EMULTIHOP; а POSIX не описывает коды ошибок ETXTBSY, EPERM, EFAULT, ELOOP, EIO, ENFILE, EMFILE, EINVAL, EISDIR и ELIBBAD.

ЗАМЕЧАНИЯ Процессы SUID или SGID не могут быть отлажены с помощью **ptrace()**. Linux игнорирует биты SUID и SGID в скриптах.

Результат подключения файловой системы *nosuid* будет отличаться в разных версиях ядер Linux: некоторые будут отменять выполнение SUID/SGID исполняемых, если это даст пользователю возможности, недопустимых для него (возвращая EPERM), некоторые будут просто игнорировать биты SUID/SGID и нормально выполняться.
Первая строка (строка с #!) исполняемого скрипта не может быть длиннее 127-и символов.

СМ. ТАКЖЕ **chmod(2), fork(2), execl(3), environ(5), ld.so(8)**

Linux 2.0.30 3 September 1997 EXECVE**(2)**

FCNTL

НАЗВАНИЕ fcntl - функция для работы с файловыми описателями

СИНТАКСИС

```
#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);
```

ОПИСАНИЕ

fcntl выполняет различные операции над файловым описателем *fd*. Рассматриваемая операция определяется значением *cmd*:

F_DUPFD ищет первый доступный файловый описатель, больший или равный значению *arg*, и делает его копией *fd*. Это поведение отличается от поведения функции **dup2(2)**, которая использует именно заданный файловый описатель.
Старый и новый описатели могут заменять друг друга. У них общие блокировки, общее положение указателя в файле и флаги; например, если положение указателя изменяется с помощью **lseek** в одном из описателей, то оно также меняется в другом.
Два описателя, однако, не делят флаг "close-on-exec". В копии значение этого флага будет равно нулю; это значит, что он не будет закрыт при запуске exec.
При успешном завершении работы возвращается новый описатель.

F_GETFD Получить состояние флага "close-on-exec". Если бит **FD_CLOEXEC** результата равен нулю, то файл будет оставаться открытым после выполнения **exec**, в противном случае файл будет закрыт.

F_SETFD Установить флаг "close-on-exec", как указано в бите **FD_CLOEXEC** параметра *arg*.

F_GETFL Прочитать флаги описателя (возвращаются все флаги, установленные при помощи **open(2)**).

F_SETFL Установить значение флагов описателя, заданное в *arg*. Можно установить только флаги **O_APPEND**, **O_NONBLOCK** и **O_ASYNC** (прочие флаги не будут затронуты).
Флаги являются общими для копий файлового описателя, сделанных с помощью **dup(2)**, **fork(2)** и т.п.
Флаги и их семантика описаны в **open(2)**.

F_GETLK, **F_SETLK** и **F_SETLKW** используются для управления файловыми блокировками. Третий аргумент, *lock*, указывает на структуру **struct flock** (которая может быть перезаписана этим системным вызовом).

F_GETLK
Возвращает структуру **flock**, которая препятствует своей собственной блокировке или установке значения

поля **l_type** равным **F_UNLCK**, если других блокировок нет.

F_SETLK

Блокировка устанавливается (если поле **l_type** равно **F_RDLCK** или **F_WRLCK**) или снимается (если это поле равно **F_UNLCK**). Если блокировка установлена кем-то ещё, этот вызов возвращает значение -1 и устанавливает значение *errno* равным **EACCESS** или **EAGAIN**.

Значение

F_SETLKW аналогично **F_SETLK**, только вместо того, чтобы вернуть код ошибки, пользователь ждет снятия блокировки. Если во время ожидания к Вам приходит соответствующий

сигнал, то системный вызов прерывается и, после того, как обработан сигнал, значение -1 немедленно возвращается, а значение *errno* становится равным **EINTR**.

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

F_GETOWN

Получает идентификатор процесса или группы процессов, к которым посылаются сигналы **SIGIO** и **SIGURG** о событиях в файловом описателе *fd*. Возвращаемое значение группы процессов становится отрицательным.

F_SETOWN

Устанавливает идентификатор процесса или группы процессов, которые будут получать сигналы **SIGIO** и **SIGURG** о событиях в файловом описателе *fd*. Значение групп процессов становится отрицательным. (**F_SETSIG** может использоваться для задания другого сигнала вместо **SIGIO**). Если Вы установите флаг **O_ASYNC** в файловом описателе (передав этот флаг при вызове **open(2)** или задав команду **F_SETFL** при вызове **fcntl**), то сигнал SIGIO будет посылаться каждый раз, когда в этом файловом описателе становится возможым ввод и вывод.

Процесс или группа процессов, которые будут получать сигнал, могут быть выбраны с помощью команды **F_SETOWN**, заданной функцией **fcntl**. Если файловый описатель - это сокет, то при этом будет также выбран адресат сигналов SIGURG, которые посылаются, когда на сокет приходят данные из пространства вне основного канала. (SIGURG отсылается в любой ситуации, когда функция **select(2)** сообщает о сложившейся в сокете "исключительной ситуации".) Если файловый описатель соответствует терминальному устройству, то сигналы

SIGIO посылаются группе нефоновых процессов на терминале.

F_GETSIG

Дает возможность узнать, какой сигнал посылается, когда становится возможным ввод или вывод. Ноль означает, что посылается SIGIO. Любое другое значение (включая SIGIO) - это сигнал, который посылается вместо SIGIO (в этом случае доступна дополнительная информация, если обработчик сигнала был установлен с помощью функции **SA_SIGINFO**).

F_SETSIG

Задает сигнал, который посылается, когда становится возможным ввод или вывод информации. Значение "ноль" показывает, что нужно посылать стандартный сигнал SIGIO. Любое другое значение (включая SIGIO) означает, что нужно послать другой сигнал, и в этом

случае также доступна дополнительная информация, если обработчик сигнала был установлен с помощью SA_SIGINFO.
Используя F_SETSIG вместе с ненулевым значением и предоставляя обработчику сигнала флаг SA_SIGINFO (см. **sigaction(2)**), можно передать этому обработчику дополнительную информацию о событиях ввода-вывода с помощью структуры *siginfo_t*. Если в поле *si_code* указано, что источником является SI_SIGIO, то поле *si_fd* содержит файловый описатель, в котором произошло событие. В противном случае нет прямого указания, какие именно файловые
описатели участвуют в происходящем, поэтому нужно использовать обычные механизмы (**select(2)**, **poll(2)**, **read(2)** с флагом **O_NONBLOCK**, и так далее), чтобы определить, каким описателям доступен процесс ввода-вывода.
Выбрав сигнал реального времени, соответствующий стандарту POSIX (значение >= SIGRTMIN), можно работать с очередями событий ввода-вывода, использующих один и тот же номер сигнала. (Построение очереди зависит от доступной памяти). Дополнительная информация доступна, если обработчику предоставлен SA_SIGINFO, как описано выше.

При помощи этих механизмов можно реализовать полностью асинхронный ввод-вывод без использования, по большей части, **select(2)** или **poll(2)**.
Использование **O_ASYNC**, **F_GETOWN**, **F_SETOWN** специфично для систем BSD и Linux. **F_GETSIG** и **F_SETSIG** специфичны для Linux. POSIX включает в себя описание асинхронного ввода-вывода и структуру *aio_sigevent*, с помощью которой достигаются эти цели; они также доступны в Linux как часть библиотеки GNU C (Glibc).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении работы возвращаемое значение зависит от того, как действуют следующие установки:

F_DUPFD (новый описатель);
F_GETFD (значение флага);
F_GETFL (значение флагов);
F_GETOWN (владелец описателя);
F_GETSIG (номер сигнала, который посылается, когда появляется возможность чтения или записи, или же ноль, означающий традиционное поведение сигнала SIGIO).

При задании всех остальных команд возвращаемое значение равно нулю, а при ошибке оно равно -1; а переменной *errno* присваивается соответствующий код ошибки.

КОДЫ ОШИБОК

EACCES Выполнение операции невозможно из-за блокировки, установленной другим процессом.
EAGAIN Операция запрещена, потому что файл был записан в память другим процессом.
EBADF *fd* не является открытым файловым описателем или командой была **F_SETLK** или **F_SETLKW** и режим открытия файлового описателя не совпадает с типом запрошенной блокировки.
EDEADLK Обнаружено, что заданная команда **F_SETLKW** вызовет "мертвую" блокировку.
EFAULT *lock* указывает на каталог за пределами доступного адресного пространства.
EINTR Выполнение команды **F_SETLKW** было прервано сигналом. Выполнение команд **F_GETLK** и **F_SETLK** было прервано сигналом, пока блокировка еще не была проверена или

установлена. Это чаще всего случается при блокировке сетевого файла (например, при работе с NFS), но иногда может случиться и локально.

EINVAL Ошибка команды **F_DUPFD**: *arg* отрицателен или значение его больше, чем максимально разрешенное. Команда **F_SETSIG**: *arg* не является разрешенным номером сигнала.

EMFILE Команда **F_DUPFD**: процесс уже открыл максимальное количество файловых описателей.

ENOLCK слишком много сегментов заблокировано, заполнена таблица блокировок или же произошла ошибка при сетевой блокировке (при работе с NFS).

EPERM Попытка сбросить флаг **O_APPEND** с файла, имеющего атрибут "только добавление".

ЗАМЕЧАНИЯ

Коды ошибок, которые возвращает **dup2**, отличаются от тех, которые возвращает **F_DUPFD**.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. В POSIX.1 указаны только операции F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL, F_GETLK, F_SETLK и F_SETLKW. F_GETOWN и F_SETOWN являются специфичными для BSD, которые не поддерживаются в SVr4; F_GETSIG и F_SETSIG специфичны для Linux. Флаги для F_GETFL/F_SETFL - это флаги, которые поддерживаются вызовом **open(2)**, и они отличаются в разных системах; O_APPEND, O_NONBLOCK, O_RDONLY и O_RDWR указаны в POSIX.1. SVr4 поддерживает несколько других опций и флагов, не описанных здесь.

SVr4 описывает дополнительные коды ошибок EIO, ENOLINK и EOVERFLOW.

СМ. ТАКЖЕ **dup2(2), flock(2), open(2), socket(2).**

Linux 12 July 1999 FCNTL**(2)**

FDATASYNC

НАЗВАНИЕ fdatasync - синхронизирует содержимое файла в памяти с содержимым на диске

СИНТАКСИС

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

ОПИСАНИЕ **fdatasync** записывает на диск содержимое всех буферов данных (до того, как завершится вызов функции), связанных с файлом Этот вызов напоминает вызов функции **fsync**, но от него не требуется обновлять метаданные, например, время доступа. Приложения, которые работают с базами данных или файлами журналов, часто пишут небольшие фрагменты данных (например, строку в журнал), а затем вызывают **fsync**, чтобы убедиться, что записанные данные сохранены на жестком диске. К сожалению, **fsync** всегда производит две операции 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 успешного завершения функции возвращается нулевое значение. При ошибке возвращается -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EBADF *fd* не является правильным описателем файла, открытым для записи.

EROFS, EINVAL

fd связан со специальным файлом, не поддерживающим

синхронизацию.

EIO Во время синхронизации произошла ошибка.

НАЙДЕННЫЕ ОШИБКИ

В настоящий момент (Linux 2.0.23) **fdatasync** эквивалентен **fsync**.

СООТВЕТСТВИЕ СТАНДАРТАМ POSIX1b (ранее известный как POSIX.4)

СМ. ТАКЖЕ fsync(2)

Linux 1.3.86 13 April 1996 FDATASYNC**(2)**

FLOCK

НАЗВАНИЕ flock - устанавливает или снимает "мягкую" блокировку (advisory lock) открытого файла

СИНТАКСИС

```
#include <sys/file.h>

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

ОПИСАНИЕ Устанавливает или снимает "мягкую" блокировку открытого файла. Файл задается описателем файла *fd*.

Допустимы следующие операции:

LOCK_SH	(разделяемая блокировка. Несколько процессов одновременно могут держать разделяемую блокировку файла одновременно.);
LOCK_EX	(исключительная блокировка. Только один процесс в каждый момент времени может держать исключительную блокировку файла.);
LOCK_UN	(разблокировать файл);
LOCK_NB	(не блокировать файл, когда он уже заблокирован; это значение может быть задано с помощью операции *логическогосложения* или с помощью других операций).

Файл не может иметь одновременно разделяемую и исключительную блокировку. Блокируется файл (т.е. inode), а *не* описатель файла. Поэтому **dup(2)** и **fork(2)** не создают нескольких копий (instances) блокировки.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успешного завершения возвращается нулевое значение. При ошибке возвращается -1, а переменная *errno* приобретает соответствующее значение.

КОДЫ ОШИБОК

EWOULDBLOCK

Файл заблокирован, а при вызове функции был задан флаг **LOCK_NB**.

СООТВЕТСТВИЕ СТАНДАРТАМ

4.4BSD (вызов **flock(2)** впервые появился в 4.2BSD).

ЗАМЕЧАНИЯ **flock(2)** не блокирует файлы, расположенные в файловой системе NFS; используйте для этого **fcntl(2)**. Требуются более свежие версии Linux и сервера, поддерживающие блокировку.

Блокировки, обеспечиваемые системными вызовами **flock(2)** и **fcntl(2)**, имеют различную относительно порожденных процессов и вызова **dup(2)** семантику.

СМ. ТАКЖЕ **open(2)**, **close(2)**, **dup(2)**, **execve(2)**, **fcntl(2)**, **fork(2)**, **lockf(3)**

Смотрите также *locks.txt* и *mandatory.txt* из каталога */usr/src/linux/Documentation*.

Linux 11 December 1998 FLOCK**(2)**

FORK

НАЗВАНИЕ fork - порождает дочерний процесс

СИНТАКСИС

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

pid_t fork(void);
```

ОПИСАНИЕ

fork порождает дочерний процесс, который отличается от родительского процесса только значениями PID (идентификатора процесса) и PPID (идентификатора родительского процесса), и значение счетчиков использования ресурсов равно 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются.
В Linux **fork** реализован с помощью метода "копирование страниц при записи" (copy-on-write), поэтому расходы на **fork** сводятся к копированию таблицы страниц родителя и созданию уникальной структуры, описывающей задачу порожденного процесса.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении родительскому процессу возвращается PID дочернего, а дочернему процессу возвращается 0. При неудачном завершении родительскому процессу возвращается -1, дочерний процесс не создается, а в переменную *errno* записывается код ошибки.

КОДЫ ОШИБОК

EAGAIN **fork** не может получить достаточно памяти для копирования таблиц страниц родителя и для выделения структуры описания дочернего процесса.

ENOMEM **fork** не может получить необходимые ресурсы ядра, т.к. недостаточно памяти.

СООТВЕТСТВИЕ СТАНДАРТАМ

Вызов **fork** соответствует SVr4, SVID, POSIX, X/OPEN, BSD 4.3.

СМ. ТАКЖЕ **clone(2), execve(2), vfork(2), wait(2)**

Linux 1.2.9 10 June 1995 FORK**(2)**

FSYNC

НАЗВАНИЕ fsync, fdatasync - синхронизирует состояние файла в памяти с состоянием на диске

СИНТАКСИС

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

ОПИСАНИЕ

fsync копирует все части файла, находящиеся в памяти, на устройство (диск) и ждет, пока устройство не доложит о том, что все части нормально сохранены. Также обновляется информация о состоянии метаданных. Нет необходимости убеждаться в том, что элемент каталога, содержащего файл, достиг диска. Для этого необходимо выполнить **fsync** на описатель файла для каталога. **fdatasync** делает то-же, что и **fsync**, но сбрасывает из памяти только данные пользователя, а не метаданные (напр. mtime или atime).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

В случае успешного завершения возвращается нулевое значение. При ошибке возвращается -1, и переменная *errno* устанавливается должным образом.

КОДЫ ОШИБОК

EBADF *fd* неверный описатель файла, например, открытый для записи.

EROFS, EINVAL
fd связан со специальным файлом, который не поддерживает синхронизацию.

EIO Во время синхронизации произошла ошибка.

ЗАМЕЧАНИЯ

В случае, если на жестком диске включено кэширование при записи, данных фактически может не быть на диске, хотя **fsync**/**fdatasync** уже сообщит об их записи на диск и завершит работу.

Когда файловоая система типа ext2 смонтирована с параметром *sync*, то все элементы каталогов также синхронизируются по **fsync**.

В ядрах до версии 2.4 использование **fsync** для больших файлах неэффективно. Альтернативой может быть использование флага *O_SYNC* для **open(2).**

СООТВЕТСТВИЕ СТАНДАРТАМ POSIX.1b (ранее POSIX.4)

СМ. ТАКЖЕ bdflush(2),open(2),sync(2),mount(8) sync(2),update(8), ync(8)

Linux 1.3.85 1994-04-13 FSYNC**(2)**

GETCONTEXT

НАЗВАНИЕ getcontext, setcontext - считывает или устанавливает контекст пользователя

СИНТАКСИС

#include <ucontext.h>
int getcontext(ucontext_t ****ucp*);**
int setcontext(const ucontext_t ****ucp*);**

где:

ucp указывает но структуру, определенную в <ucontext.h> и содержащую маску сигнала, стек исполнения и регистры машины.

ОПИСАНИЕ

getcontext(2) получает текущий контекст вызывающего процесса, сохраняет его в структуре ucontext - на нее указывает *ucp*.

setcontext(2) меняет контекст вызывающего процесса на состояние, сохраненное в структуре ucontext - на нее указывает *ucp*. Структура должна либо быть созданной **get-context(2),** либо быть переданной в качестве третьего параметра обработчиком сигналов **sigaction(2).**

Структура ucontext, создаваемая **getcontext(2),** определена в <ucontext.h> таким образом:

```
typedef struct ucontext
  {
      unsigned long int uc_flags;
      struct ucontext *uc_link;
      stack_t uc_stack;
      mcontext_t uc_mcontext;
      __sigset_t uc_sigmask;
      struct _fpstate __fpregs_mem;
    } ucontext_t;
```

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

getcontext(2) возвращает 0 при нормальном завершении работы и -1 при ошибках. **setcontext(2)** ничего не возвращает при нормальном завершении работы, и возвращает -1 при ошибках.

СООТВЕТСТВИЕ СТАНДАРТАМ

Эти функции соответсвуют стандарту XPG4-UNIX.

ЗАМЕЧАНИЯ Когда работает обработчик сигнала, то текущий контекст пользователя сохраняется и ядром создается новый контекст. Если вызывающий процесс оставляет обработчик сигнала, используя **longjmp(2),** то первоначальный контекст невозможно будет восстановить, а результат последующих вызовов **getcontext(2)** будет непредсказуем. Для избежания этих проблем вместо **longjmp(2)** используйте в обработчике сигналов **siglongjmp(2)** или **setcontext(2).**

СМ. ТАКЖЕ sigaction(2), sigaltstack(2), sigprocmask(2), sigsetjmp(3), setjmp (3).
Red Hat Linux 6.1 20 September 1999 GETCONTEXT**(2)**

GETDENTS

НАЗВАНИЕ getdents - получает записи из каталога
СИНТАКСИС

```
       #include <unistd.h>
       #include <linux/types.h>
       #include <linux/dirent.h>
       #include <linux/unistd.h>

       _syscall3(int, getdents, uint, fd, struct dirent *, dirp, uint,
count);
       int getdents(unsigned int fd, struct dirent *dirp, unsigned int
count);
```

ОПИСАНИЕ

getdents считывает несколько структур *dirent* из каталога, на который указывает описатель *fd*, и записывает их в область памяти, определенной параметром *dirp*. Параметр *count* задает размер области памяти.
Структура *dirent* описана следующим образом:

```
              struct dirent
              {
                  long d_ino;                 /* номер inode */
                  off_t d_off;                /* смещение следующей записи
dirent */
                  unsigned short d_reclen;    /* длина этой записи dirent */
                  char d_name [NAME_MAX+1];   /* имя файла (заканчивающегося
нулем) */
              }
```

d_ino -- номер inode. *d_off* -- расстояние от начала каталога до записи *dirent*. *d_reclen* -- размер этой записи *dirent*. *d_name* -- имя файла, заканчивающегося нулем.
Этот вызов заменяет **readdir(2)**.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращается количество считанных байтов. По окончании записей в каталоге возвращается 0. При ошибке возвращается -1, а в переменную *errno* вносится соответствующее значение.

КОДЫ ОШИБОК

EBADF Некорректный описатель файла *fd*.
EFAULT Аргумент указывает на адрес памяти, находящийся за пределами адресного пространства процесса.
EINVAL Размер буфера результата недостаточен.
ENOENT Каталог не найден.
ENOTDIR В описателе файла отсутствует ссылка на каталог.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, SVID. SVr4 дополнительная документация к ENOLINK, условия ошибок EIO.

СМ. ТАКЖЕ **readdir(2), readdir(3)**
Linux 1.3.6 22 July 1995 GETDENTS**(2)**

GETDOMAINNAME

НАЗВАНИЕ getdomainname, setdomainname - определяет/устанавливает имя домена
СИНТАКСИС

```
       #include <unistd.h>

       int getdomainname(char *name, size_t len);
```

int setdomainname(const char *_name_, size_t _len_);

ОПИСАНИЕ

Эти функции используются для получения или изменения имени домена машины. Если имя домена (завершающееся нулем) занимает более *len* байтов, то **getdomainname** возвращает первые *len* байт (glibc) или ошибку (libc).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращается ноль. В случае возникновения ошибки возвращается -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EINVAL У функции **getdomainname** в libc: значение *name* равно **NULL** или *name* длиннее, чем *len* байтов.

EINVAL У функции **setdomainname**: параметр *len* был отрицателен либо слишком велик.

EPERM Функция **setdomainname**: была вызвана не пользователем root.

EFAULT В функции **setdomainname**: *name* указывает за пределы выделенного пользователю адресного пространства.

СООТВЕТСТВИЕ СТАНДАРТАМ POSIX не содержит описания этих вызовов.

СМ. ТАКЖЕ **gethostname(2), sethostname(2), uname(2)**

Linux 2.0 25 August 1997 GETDOMAINNAME**(2)**

GETDTABLESIZE

НАЗВАНИЕ getdtablesize - определяет размер таблицы описателей

СИНТАКСИС

```
#include <unistd.h>

int getdtablesize(void);
```

ОПИСАНИЕ getdtablesize возвращает максимальное количество файлов, которые процесс может держать открытыми.

ЗАМЕЧАНИЯ getdtablesize является библиотечной функцией в DLL 4.4.1. Эта функция возвращает **OPEN_MAX** (по умолчаню в Linux 2.0.23 равно 256), если **OPEN_MAX** было определено при сборке. Иначе возвращается -1 , а переменной *errno* присваивается соответствующее значение **ENOSYS.**

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (функция **getdtablesize** впервые появилась в BSD 4.2).

СМ. ТАКЖЕ **close(2), dup(2), open(2).**

Linux 0.99.11 22 July 1993 GETDTABLESIZE**(2)**

GETGID

НАЗВАНИЕ getgid, getegid - считывает идентификатор группы процесса

СИНТАКСИС

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

gid_t getgid(void);
gid_t getegid(void);
```

ОПИСАНИЕ getgid возвращает идентификатор действительной (real) группы текущего процесса. **getegid** возвращает идентификатор эффективной (effective) группы текущего процесса. Действительный идентификатор соответствует идентификатору вызывающего процесса. Эффективный идентификатор соответствует установленному биту setuid в правах доступа исполняемого файла.

КОДЫ ОШИБОК Эти вызовы всегда завершаются успешно.

СООТВЕТСТВИЕ СТАНДАРТАМ POSIX, BSD 4.3

СМ. ТАКЖЕ **setregid(2), setgid(2).**

Linux 0.99.11 23 July 1993 GETGID**(2)**

GETGROUPS

НАЗВАНИЕ getgroups, setgroups - получает/устанавливает список дополнительных (supplementary) идентификаторов групп

СИНТАКСИС

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

int getgroups(int size, gid_t list[]);

#include <grp.h>

int setgroups(size_t size, const gid_t *list);
```

ОПИСАНИЕ getgroups

Возвращает в параметре *list* дополнительные идентификаторы групп. Количество получаемых элементов устанавливается в параметре *size*. При этом точно не установлено, включен ли эффективный идентификатор группы вызывающего процесса в возвращаемый список или нет. (Т.о., приложение должно также вызвать **getegid(2)**, чтобы удалить или добавить его в окончательный результат.)

getgroups

Если параметр *size* равен нулю, то *list* не изменяется, а возвращается только количество дополнительных идентификаторов групп для процесса.

setgroups

Устанавливает дополнительные идентификаторы групп для процесса. Эту функцию может использовать только суперпользователь.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

getgroups

При удачном завершении вызова возвращается количество дополнительных идентификаторов групп. В случае возникновения ошибки возвращается -1, а переменной *errno* присваивается соответствующее значение.

setgroups

При удачном завершении вызова возвращается ноль. В случае возникновения ошибки возвращается -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EFAULT Означает, что *list* имеет неправильный адрес.

EPERM В случае **setgroups** означает, что Вы используете ее, не обладая правами суперпользователя.

EINVAL В случае **setgroups** означает, что параметр *size* больше чем величина **NGROUPS** (32 для 2.0.32). В случае **getgroups** означает, что параметр *size* меньше, чем количество дополнительных идентификаторов групп, но и не равно нулю.

ЗАМЕЧАНИЯ

Процесс может иметь по меньшей мере NGROUPS_MAX дополнительных идентификаторов групп в дополнение к эффективному идентификатору группы. Множество дополнительных идентификаторов групп наследуется от родительского процесса и может быть изменено используя вызов **sysconf(3)**:

```
long ngroups_max;
ngroups_max = sysconf(_SC_NGROUPS_MAX);
```

Максимальное возвращаемое значение функцией **getgroups** не может быть больше, чем значение полученное этим способом. Прототип для **setgroups** будет доступен, если определено **_BSD_SOURCE** (либо явно, либо неявно - не определением

_POSIX_SOURCE или компиляцией с флагом -ansi).

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, SVID (только в выпуске 4; эти вызовы отсутствуют в SVr3), X/OPEN, 4.3BSD. Вызов **getgroups** соответствует POSIX.1. Вызов **setgroups** требует прав суперпользователя и не входит в стандарт POSIX.1.

СМ. ТАКЖЕ **initgroups(3), getgid(2), setgid(2)**

Linux 2.0.32 10 December 1997 GETGROUPS**(2)**

GETHOSTID

НАЗВАНИЕ gethostid, sethostid - определяет или устанавливает уникальный идентификатор узла

СИНТАКСИС

```
#include <unistd.h>

long gethostid(void);
int sethostid(long hostid);
```

ОПИСАНИЕ Определяет или устанавливает уникальный 32-битный идентификатор машины. 32-битный идентификатор уникален для каждой существующей системы UNIX. Это напоминает адрес машины в Интернет, возвращаемый функцией **gethostbyname(3)**, который **обычно** не надо устанавливать. Вызов **sethostid** можно произвести только при наличии прав суперпользователя. Параметр *hostid* хранится в файле */etc/hostid*.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

gethostid возвращает 32-битный идентификатор машины, установленный функцией **sethostid(2)**.

СООТВЕТСТВИЕ СТАНДАРТАМ

4.2BSD. Но они были исключены из 4.4BSD. POSIX.1 не охватывает эти функции, но ISO/IEC 9945-1:1990 описывает их в B.4.4.1. SVr4 включает в себя **gethostid** , но не **sethostid**.

ФАЙЛЫ */etc/hostid*

СМ. ТАКЖЕ **hostid(1), gethostbyname(3).**

Linux 0.99.13 29 November 1993 GETHOSTID**(2)**

GETHOSTNAME

НАЗВАНИЕ gethostname, sethostname - определяет/устанавливает имя узла

СИНТАКСИС

```
#include <unistd.h>

int gethostname(char *name, size_t len);
int sethostname(const char *name, size_t len);
```

ОПИСАНИЕ

Эти функции используются для определения или изменения имени машины. Функция **gethostname()** возвращает оканчивающуюся NUL строку с именем машины (установленным ранее функцией **sethostname()**) в массиве *name*, который имеет длину *len* байт. Если имя машины, оканчивающееся NUL, не помещается в массив, то функция не вернет ошибку, хотя имя машины будет обрезано. Не определено, будет ли обрезаное имя оканчиваться NUL.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращается ноль. При ошибке возвращается -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EINVAL *len* отрицательно или больше допустимого размера (для **sethostname**), или меньше действительного значения (для **gethostname** в Linux/i386 в этом случае использует ENAMETOOLONG).

EPERM вызов был произведен без прав суперпользователя (для **sethostname**).

EFAULT *name* не является правильным адресом.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (эта функция появилась в 4.2BSD). POSIX 1003.1-2001 описывает **gethostname**, однако не описывает **sethostname**.

НАЙДЕННЫЕ ОШИБКИ

Для многих сочетаний ядра Linux / libc **gethostname** будет возвращать ошибку вместо обрезанного имени.

ЗАМЕЧАНИЯ SUSv2 гарантирует, что `Имя машины ограничено 255 байтами'. POSIX 1003.1-2001 гарантирует, что `Имя машины (не включая конечный NUL) ограничено HOST_NAME_MAX байтами.

СМ. ТАКЖЕ **getdomainname(2), setdomainname(2), uname(2).**

Linux 2.5.0 2001-12-15 GETHOSTNAME**(2)**

GETITIMER

НАЗВАНИЕ getitimer, setitimer - считывает или устанавливает значение таймера интервалов (interval timer)

СИНТАКСИС

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
      struct itimerval *ovalue);
```

ОПИСАНИЕ

Система предоставляет каждому процессу три таймера, значение каждого из которых уменьшается на единицу по истечении установленного времени. Когда на одном из таймеров заканчивается время, процессу посылается сигнал и таймер обычно перезапускается.

ITIMER_REAL уменьшается постоянно (in real time) и подает сигнал **SIGALRM**, когда значение таймера становится равным 0.

ITIMER_VIRTUAL уменьшается только во время работы процесса и подает сигнал **SIGVTALRM**, когда значение таймера становится равным 0.

ITIMER_PROF уменьшается во время работы процесса и когда система выполняет что-либо по заданию процесса. Совместно с **ITIMER_VIRTUAL** этот таймер обычно используется для профилирования времени работы приложения в пользовательской области и в области ядра. Когда значение таймера становится равным 0, подается сигнал **SIGPROF**.

Величина, на которую устанавливается таймер, определяется следующими структурами:

```
struct itimerval {
    struct timeval it_interval; /* следующее значение */
    struct timeval it_value;    /* текущее значение */
};
struct timeval {
    long tv_sec;                /* секунды */
    long tv_usec;               /* микросекунды */
};
```

Функция **getitimer(2)** заполняет структуру *value* текущим

значением *which* (**ITIMER_REAL**, **ITIMER_VIRTUAL**, или **ITIMER_PROF**). **it_value** устанавливается в соответствии с тем количеством времени, которое осталось на таймере, или приравнивается нулю, если таймер выключен. Аналогично устанавливается **it_interval**. Функция **setitimer(2)** устанавливает значение таймера равным величине, указанной в *value*. Если величина *ovalue* не равна нулю, то в нее записывается прежнее значение таймера.

Значения таймеров уменьшаются от величины *it_value* до нуля, подается сигнал, и значения вновь устанавливаются равными *it_interval*. Таймер, установленный на ноль (его величина it_value равна нулю или время вышло, и величина *it_interval* равна нулю), останавливается.

Величины *tv_sec* и *tv_usec* являются основными при установке таймера.

Время на таймерах никогда не заканчивается ранее указанного срока, за исключением того случая, когда установлено время менее допустимого, в свою очередь константа времени зависит от степени разрешения (обычно 10мсек). По истечению будет послан сигнал, а таймер обнулится. Если таймер заканчивается во время работы процесса (это всегда бывает с **ITIMER_VIRT**), то сигнал будет немедленно послан. Иногда отсылка сигнала откладывается на небольшой промежуток времени, зависящий от степени загруженности системы.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном выполнении возвращается ноль. При ошибке возвращается -1, а переменной *errno* присваиваются соответствующие значения.

КОДЫ ОШИБОК

EFAULT Указатели *value* или *ovalue* являются некорректными.

EINVAL *which* не равно **ITIMER_REAL**, **ITIMER_VIRT**, или **ITIMER_PROF**.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (Впервые этот вызов появился в 4.2BSD).

СМ. ТАКЖЕ **gettimeofday(2), sigaction(2), signal(2)**

НАЙДЕННЫЕ ОШИБКИ

В Linux, все генерируемые сигналы являются уникальными и на каждый из них система отвечает уникальным процессом. При чрезвычайной перегруженности ситемы **ITIMER_REAL** может закончиться до того момента, как сигнал от предыдущего будет доставлен. И таким образом, сигнал от последующего таймера будет потерян.

Linux 0.99.11 5 August 1993 GETITIMER**(2)**

GETPAGESIZE

НАЗВАНИЕ getpagesize - определяет размер страницы памяти

СИНТАКСИС

```
#include <unistd.h>

int getpagesize(void);
```

ОПИСАНИЕ Функция **getpagesize()** Возвращает количество байтов в странице. Под страницей подразумевается то, что имеется ввиду в описании функции **mmap(2)**, когда говорится о том, что файлы отображаются постранично.

Размер страниц, используемых **mmap** определяется через

```
long sz = sysconf(_SC_PAGESIZE);
```

(некоторые системы позволяют вместо _SC_PAGESIZE использовать _SC_PAGE_SIZE) или

```
int sz = getpagesize();
```

ИСТОРИЯ Эта функция появилась в 4.2BSD.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVR4, 4.4BSD, SUSv2. В SUSv2 функция **getpagesize()** помечена как "legacy", а из POSIX 1003.1-2001 она была убрана. HPUX эту функцию не поддерживает.

ЗАМЕЧАНИЯ

Поддерживается ли функция **getpagesize** в Linux зависит от архитектуры. Если поддерживается, то она возвращает символ PAGE_SIZE ядра, который зависит от архитектуры и модели машины. Как правило, создаваемые бинарные файлы используются для всей архитектуры, а не для конкретной одной модели. Поэтому рекомендуется определять PAGE_SIZE не на стадии компиляции из файла заголовка, а при выполнении программы с помощью данной функции, по крайней мере на тех архитектурах (таких как sun4), где зависимость от модели существует. В этом случае функции **getpagesize()** libc4, libc5 и glibc 2.0 не решат проблему, так возвращают статически унаследованное значение, не используя системный вызов. В glibc 2.1 это исправлено.

СМ. ТАКЖЕ **mmap(2), sysconf(3)**

Linux 2.5.0 2001-12-21 GETPAGESIZE**(2)**

GETPEERNAME

НАЗВАНИЕ getpeername - считывает имя подсоединившегося пользователя (машины)

СИНТАКСИС

#include <sys/socket.h>

int getpeername(int *s*, **struct sockaddr *****name*, **socklen_t *****namelen**);

ОПИСАНИЕ Getpeername возвращает имя пользователя машины подсоединившейся на сокет *s*. Параметр *namelen* должен быть инициализирован в целях отображения объема памяти, который занимает *name*. По возвращении он содержит размер памяти, занимаемый именем машины (байт). Имя не считывается если буфер окажется слишком мал.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении возвращается ноль. При ошибке возвращается -1 , а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EBADF Аргумент *s* не является правильным описателем.

ENOTSOCK Аргумент *s* является файлом, а не сокетом.

ENOTCONN Нет соединения.

ENOBUFS Нехватает памяти для производства операции.

EFAULT Параметр *name* указывает на участок недоступного адресного пространства.

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, 4.4BSD (вызов **getpeername** впервые появился в 4.2BSD).

ЗАМЕЧАНИЕ

Третий аргумент функции **getpeername** в действительности является `int *' (то что и описано в BSD 4.*, libc4 и libc5). В результате недоразумения в POSIX появилось socklen_t. Хотя черновой вариант еще не принят, но glibc2 уже следует ему и соответственно включает в себя socklen_t. См. также **accept(2).**

СМ. ТАКЖЕ accept(2), bind(2), getsockname(2).

BSD Man Page 30 July 1993 GETPEERNAME**(2)**

GETPID

НАЗВАНИЕ getpid, getppid - считывает идентификатор процесса

СИНТАКСИС

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

pid_t getpid(void);
pid_t getppid(void);
```

ОПИСАНИЕ getpid возвращает идентификатор текущего процесса (PID). Обычно это используется программами, создающими уникальные временные названия файлов. **getppid** возвращает идентификатор родительского процесса(PPID).

СООТВЕТСТВИЕ СТАНДАРТАМ POSIX, BSD 4.3, SVID

СМ. ТАКЖЕ exec(3), fork(2), kill(2), mkstemp(3), tmpnam(3), tempnam(3), tmpfile(3)

Linux 0.99.11 23 July 1993 GETPID**(2)**

GETPRIORITY

НАЗВАНИЕ getpriority, setpriority - получить/установить приоритеты процессов

СИНТАКСИС

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

ОПИСАНИЕ Приоритеты процессов, групп процессов или пользователей, указанных в *which* и *who*, можно узнать при помощи вызова **getpriority** и установить при помощи вызова **setpriority**, где *which* принимает значения **PRIO_PROCESS**, **PRIO_PGRP** или **PRIO_USER**, а *who* воспринимается в зависимости от значения *which* (.BR PRIO_PROCESS /-/- идентификатор процесса, **PRIO_PGRP** /-/- идентификатор группы прцессов, **PRIO_USER** /-/- идентификатор пользователя). Нулевое значение *who* обозначает, что вызов применяется к текущему процессу, группе процессов или пользователю. *prio* принимает значения от -20 до 20. По умолчанию значение приоритета равно 0; меньшие его значения означают превосходство над другими процессами. Вызов **getpriority** возвращает значение самого высокого из указанных процессов приоритета (наименьшее числовое значение). Вызов **setpriority** устанавливает значение приоритетов для всех указанных процессов на нужную величину. Только суперпользователь имеет право на уменьшение приоритета.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Так как вызов **getpriority** может вернуть значение -1, которое будет означать приоритет, необходимо очистить внешнюю переменную *errno* до вызова, тогда Вы будете знать, что значение -1 является сообщением об ошибке или значением приоритета. Вызов **setpriority** при удачном его завершении возвращает 0 или -1, если произошла ошибка.

КОДЫ ОШИБОК

ESRCH Не найден процесс, указанный в *which* и *who*.

EINVAL *which* не является **PRIO_PROCESS**, **PRIO_PGRP** или **PRIO_USER**.

В добавление к написанному выше, **setpriority** может не выполнится в случае возникновения следующих ошибок:

EPERM Идентификатор эффективного (effective) пользователя вызывающего процесса не соответствует ни идентификатору эффективного, ни реального (real) пользователя найденного процесса;

EACCES попытка уменьшения приоритета без прав суперпользователя.

ЗАМЕЧАНИЯ

Точное поведение при возникновении EPERM зависит от системы. Вышеуказанное описание соответстует SUSv3, и, похоже, действительно для всех систем, подобных SYSV. Linux требует, чтобы реальный или эффективный идентификатор пользователя вызывающего процесса соответствовал реальному (а не эффективному) идентификатору пользователя процесса *who*.

Действительный диапазон приоритетов отличается в разных версиях ядра. В Linux до версии 1.3.36 этот диапазон был от минус бесконечности до 15. С версии 1.3.43 этот диапазон стал от -20 до 19, и системный вызов getpriority возвращает 40..1 для этих значений (так как отрицательные номера являются кодами ошибок). Библиотелный вызов преобразует N в 20-N.

Подключение *<sys/time.h>* сейчас уже не требуется, но улучшает портируемость. (В самом деле, *<sys/resource.h>* определяет структуру *rusage* с полями типа *struct timeval*, определенными в *<sys/time.h>*.)

СМ. ТАКЖЕ **nice(1), fork(2), renice(8).**

BSD Man Page 2001-06-19 GETPRIORITY**(2)**

GETRESUID

НАЗВАНИЕ getresuid, getresgid - считывает идентификаторы действительного (real), эффективного (effective) или сохраненного (saved) пользователя или группы.

СИНТАКСИС

```
#include <unistd.h>

int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
```

ОПИСАНИЕ

getresuid и **getresgid** (представленные в Linux 2.1.44) считывают идентификатор действительного (real), эффективного (effective) или сохраненного (saved) пользователя (соответственно и идентификаторы групп) текущего процесса.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращается ноль. При ошибке возвращается -1, а переменной *errno* присваиваются соответствующие значения.

КОДЫ ОШИБОК

EFAULT Один из аргументов указывает на адрес вне области адресного пространства, принадлежащего вызывающей программе.

СООТВЕТСТВИЕ СТАНДАРТАМ Этот вызов применяется только в Linux.

СМ. ТАКЖЕ **getuid(2), setuid(2), setreuid(2), setresuid(2).**

Linux 2.1.44 16 July 1997 GETRESUID**(2)**

GETRLIMIT

НАЗВАНИЕ getrlimit, getrusage, setrlimit - считывает/устанавливает ограничения использования ресурсов

СИНТАКСИС

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrlimit(int resource, struct rlimit *rlim);
int getrusage(int who, struct rusage *usage);
int setrlimit(int resource, const struct rlimit *rlim);
```

ОПИСАНИЕ **getrlimit** и **setrlimit** соответственно получают и устанавливают ограничения использования ресурсов.

Значение *resource* должно быть одним из:

```
RLIMIT_CPU     /* системное время в секундах */
RLIMIT_FSIZE   /* максимальный размер файла */
RLIMIT_DATA    /* максимальный размер данных */
RLIMIT_STACK   /* максимальный размер стека */
RLIMIT_CORE    /* максимальный размер файла core */
RLIMIT_RSS     /* максимальный rss */
RLIMIT_NPROC   /* максимальное количество процессов */
RLIMIT_NOFILE  /* максимальное количество открытых файлов
*/
RLIMIT_MEMLOCK /* максимальный объем заблокированного
адресного пространства*/
RLIMIT_AS      /* максимальный объем адресного
пространства */
```

Можно снять все ограничения с ресурса путем установки ограничения на **RLIM_INFINITY**. **RLIMIT_OFILE** в BSD называется **RLIMIT_NOFILE.**

Структура **rlimit** описывается следующим образом :

```
struct rlimit {
     rlim_t    rlim_cur;
     rlim_t    rlim_max;
};
```

getrusage возвращает текущие ограничения на ресурсы для *who*, который может быть **RUSAGE_SELF** или **RUSAGE_CHILDREN.** Первое запрашивает информацию о ресурсах используемых текущим процессом, а второе о тех порожденных процессах, которые завершились или завершение которых ожидается.

```
struct rusage {
     struct timeval ru_utime; /* время работы пользователя */
     struct timeval ru_stime; /* использованное системное время
*/
     long ru_maxrss;          /* максимальный rss */
     long ru_ixrss;      /* общий объем разделяемой памяти */
     long ru_idrss;      /* общий объем неразделяемых данных */
     long ru_isrss;      /* общий объем неразделяемых стеков */
     long ru_minflt;          /* количество процессов подгрузки
страницы */
     long ru_majflt;          /* количество ошибок при обращении
к странице */
     long ru_nswap;      /* количество обращений к диску при
подкачке */
     long ru_inblock;         /* количество операций блокового
ввода */
     long ru_oublock;         /* количество операций блокового
вывода */
     long ru_msgsnd;          /* количество отправленных
сообщений */
     long ru_msgrcv;          /* количество принятых сообщений */
     long ru_nsignals;        /* количество принятых сигналов */
     long ru_nvcsw;      /* количество переключений контекста
процессом */
     long ru_nivcsw;          /* количество принудительных
переключений контекста */
};
```

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращается ноль. При ошибке возвращается -1, а переменной *errno* присваиваются соответствующие значения.

КОДЫ ОШИБОК

EFAULT *rlim* или *usage* указывают на недоступную область

адресного пространства.

EINVAL **getrlimit** или **setrlimit** вызывается неправильным *resource*, или **getrusage** вызывается неправильным *who*.

EPERM производится попытка использования **setrlimit()** для установки мягких (advisory) или жестких (mandatory) ограничений поверх текущих жестких при отсутствии прав суперпользователя, или суперпользователь пытается установить значение RLIMIT_NOFILE больше текущего максимума ядра.

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, BSD 4.3

ЗАМЕЧАНИЕ Подключение *<sys/time.h>* сейчас уже не требуется, но улучшает портируемость. (В самом деле, *struct timeval* определяется в *<sys/time.h>*.)

Вышеописанная структура была взята из BSD 4.3 Reno. Не все поля имеют смысл в Linux. Сейчас (в Linux 2.4) поддерживаются только поля **ru_utime**, **ru_stime**, **ru_minflt**, **ru_majflt**, и **ru_nswap**.

СМ. ТАКЖЕ **quotactl(2)**. **ulimit(2)**,

Linux 23 July 1993 GETRLIMIT**(2)**

GETSID

НАЗВАНИЕ getsid - определяет идентификатор сессии

СИНТАКСИС

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

ОПИСАНИЕ **getsid(0)** возвращает идентификатор сессии вызвавшего процесса. **getsid(*p*)** возвращает идентификатор сессии процесса и идентификатор процесса *p*. (Идентификатор сессии процесса - это идентификатор группы процесса, являющегося лидером сессии.) При ошибке возвращается (pid_t) -1 и выставляется соответствующим образом *errno*.

КОДЫ ОШИБОК

EPERM Процесс с идентификатором *p* существует, но не относится к той же сессии, что и текущий процесс, а текущая реализация этой функции считает это ошибкой.

ESRCH Не найдено процесса с идентификатором *p*).

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, POSIX 1003.1-2001.

ЗАМЕЧАНИЯ Linux не возвращает EPERM.

Эта функция появилась в Linux начиная с версии 1.3.44. Поддержка в libc появилась начиная с libc версии 5.2.19. Для получения прототипа в glibc, необходимо определить (define) _XOPEN_SOURCE и _XOPEN_SOURCE_EXTENDED, или использовать "#define _XOPEN_SOURCE *n*", где *n* больше, либо равно 500.

СМ. ТАКЖЕ **getpgid(2)**, **setsid(2)**.

Linux 2.5.0 2001-12-17 GETSID**(2)**

GETSOCKNAME

НАЗВАНИЕ getsockname - считывает адрес сокета

СИНТАКСИС

```
#include <sys/socket.h>

int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

ОПИСАНИЕ **getsockname** возвращает текущее имя указанного сокета. В параметре *namelen* должно быть указано, сколько места выделено под *name*. При возврате в этом параметре передается размер (в байтах).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении вызова возвращается нулевое

значение. При ошибке возвращается -1, а переменной *errno* присваивается соответствующий код ошибки.

КОДЫ ОШИБОК

EBADF некорректный описатель *s*.

ENOTSOCK параметр *s* является файлом, а не сокетом.

ENOBUFS в системе недостаточно ресурсов для выполнения операции.

EFAULT *name* указывает за пределы доступного адресного пространства.

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, 4.4BSD (функция **getsockname** появилась в 4.2BSD).SVr4 документирует дополнительные коды ошибок ENOMEM и ENOSR.

ЗАМЕЧАНИЕ

Третий аргумент функции **getsockname** в действительности имеет тип int * (это именно так в BSD 4.*, libc4 и libc5). Определенное недопонимание привело к тому, что в стандарте POSIX появился тип socklen_t. Черновой вариант стандарта еще не утвержден, но glibc2 уже следует ему и содержит socklen_t. См. также **accept(2)**.

СМ. ТАКЖЕ **bind(2), socket(2)**.

BSD Man Page 24 July 1993 GETSOCKNAME**(2)**

GETSOCKOPT

НАЗВАНИЕ getsockopt, setsockopt - считывает и установливает параметры, связанные с сокетом

СИНТАКСИС

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

int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

ОПИСАНИЕ **getsockopt** и **setsockopt** управляют *параметрами*, установленными в сокете. Параметры могут существовать на нескольких уровнях протоколов; они всегда располагаются на самом верхнем из них. При работе с параметрами сокета должен быть указан уровень протокола, на котором находится этот параметр, и имя этого параметра. Для манипуляции параметрами на уровне сокета *level* задается как **SOL_SOCKET**. Для манипуляции параметрами на любом другом уровне этим функциям передается номер соответствующего протокола, управляющего параметрами. Например, для указания, что флаг должен интерпретироваться протоколом **TCP**, аргумент *level* должен передавать номер протокола **TCP**; см. описание **getprotoent(3)**. Аргументы *optval* и *optlen* используются в функции **setsockopt** для доступа к значениям параметров. В случае вызова **getsockopt** , они задают буфер, в который нужно поместить запрошенное значение параметра. В случае вызова **getsockopt** аргумент *optlen* передается по ссылке. При вызове он содержит размер буфера, на который указывает аргумент *optval*, а после вызова - действительный размер возвращенного значения. Если значение параметра не используется, то аргумент *optval* может быть **NULL**. *optname* и все указанные параметры без изменений передаются для интерпретации соответствующему модулю протоколов. Файл *<sys/socket.h>* содержит определения параметров уровня сокета, описанные ниже. Параметры на других уровнях протоколов различаются по формату и по имени. Обращайтесь к соответствующим пунктам руководства из раздела 4. Большинство параметров на уровне сокета используют тип *int* для аргумента *optval*. Для функции **setsockopt** аргумент должен быть ненулевым, чтобы установить параметр логического типа, или 0, чтобы сбросить этот параметр. Описание доступных параметров

сокетов находится в **socket(7)** и соответствующих протоколам страницах руководства.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном завершении вызова возвращается нулевое значение. При ошибке возвращается -1, а переменной *errno* присваивается соответствующий код ошибки.

КОДЫ ОШИБОК

EBADF Некорректный файловый описатель *s*.

ENOTSOCK
Аргумент *s* является файлом, а не сокетом.

ENOPROTOOPT
Параметр неизвестен на данном уровне.

EFAULT Адрес, на который указывает аргумент *optval*, не находится в доступной части адресного пространства процесса. В случае вызова **getsockopt** эта ошибка может также появиться, если *optlen* выходит за пределы адресного пространства процесса.

СООТВЕТСТВИЕ СТАНДАРТАМ

SVr4, 4.4BSD (эти системные вызовы впервые появились в 4.2BSD). SVr4 описывает дополнительные коды ошибок ENOMEM и ENOSR, но не описывает флаги **SO_SNDLOWAT**, **SO_RCVLOWAT**, **SO_SNDTIMEO**, **SO_RCVTIMEO**.

ЗАМЕЧАНИЕ

Пятый аргумент вызова **getsockopt** и **setsockopt** в действительности имеет тип **int** (это именно так в BSD 4.*, libc4 и libc5). При разработке стандарта POSIX возникло недоразумение, и появился тип **socklen_t**. Черновая версия стандарта ещё не утверждена, но glibc2 уже следует этому стандарту и имеет тип **socklen_t**. Смотри также **accept(2)**.

НАЙДЕННЫЕ ОШИБКИ

Некоторые параметры сокетов должны обрабатываться на более низких уровнях системы.

СМ. ТАКЖЕ ioctl(2), socket(2), getprotoent(3), protocols(5), socket(7), unix(7), tcp(7).

Linux Man Page 24 May 1999 GETSOCKOPT**(2)**

GETTIMEOFDAY

НАЗВАНИЕ gettimeofday, settimeofday - определяет / устанавливает время

СИНТАКСИС

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv , const struct
timezone *tz);
```

ОПИСАНИЕ gettimeofday и **settimeofday** могут получать и устанавливать время и часовой пояс. Аргумент *tv* является структурой **timeval** и описанной в /usr/include/sys/time.h:

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

и задающей количество секунд и микросекунд с начала эпохи (см. **time(2)**).

Аргумент *tz* является **timezone** :

```
struct timezone {
        int  tz_minuteswest; /* количество минут коррекции по
отношению к Гринвичу */
        int  tz_dsttime;     /* тип сезонной коррекции времени */
};
```

Использование структуры timezone является устаревшим

методом: поле *tz_dsttime* никогда не использовалось в Linux, оно не поддерживается и не будет поддерживаться libc или glibc. Любое появление этого поля в исходных версиях ядра (за исключением его описания) является ошибкой. Поэтому все, что описано ниже, представляет собой только исторический интерес. Поле *tz_dsttime* содержит символьную постоянную (значения приведены ниже), которая включает в себя информацию о сезонной коррекции времени (Daylight Saving Time). (Замечание: эта величина постоянна и указывает лишь на алгоритм коррекции.) Алгоритмы сезонной коррекции определяются так :

```
 DST_NONE     /* без коррекции */
 DST_USA      /* коррекция для Америки */
 DST_AUST     /* коррекция для Австралии */
 DST_WET      /* коррекция для Западной Европы */
 DST_MET      /* коррекция для Центральной Европы */
 DST_EET      /* коррекция для Восточной Европы */
 DST_CAN      /* коррекция для Канады */
 DST_GB       /* Коррекция для Великобритании */
 DST_RUM      /* коррекция для Румынии */
 DST_TUR      /* коррекция для Турции */
 DST_AUSTALT  /* коррекция со сдвигом в 1986 для Австралии
*/
```

Разумеется, коррекцию для каждой страны нельзя описать простым алгоритмом, так как этот фактор может зависеть даже от непредсказуемых политических решений. Поэтому этот метод представления часовых поясов больше не используется. В Linux при вызове **settimeofday** поле *tz_dsttime* должно содержать нулевое значение.
В Linux существует специфическое понятие `часовой сдвиг', связанное с функцией **settimeofday** *tz* не равен NULL, а параметр *tv* равен NULL, и значение поля *tz_minuteswest* не равно нулю. В этом случае предполагается, что время аппаратных часов (CMOS clock) местное и к ним дожен быть добавлен этот параметр для того, чтобы получилось время UTC. Но, как мы и говорили, использовать этот метод не рекомендуется.
Для работы со структурой timeval существуют следующие макросы:

```
        ((tvp)->tv_sec || (tvp)->tv_usec)
        ((tvp)->tv_sec cmp (uvp)->tv_sec ||\
        (tvp)->tv_sec == (uvp)->tv_sec &&\
        (tvp)->tv_usec cmp (uvp)->tv_usec)
        ((tvp)->tv_sec = (tvp)->tv_usec = 0)
```

Если *tv* или *tz* равно нулю, то соответствующая структура не заполняется или не возвращается.

Только суперпользователь может работать с **settimeofday**.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

gettimeofday и **settimeofday** при успешном завершении работы возвращают ноль, и возвращают -1 при ошибках (в этом случае *errno* присваиваются соответствующие значения).

КОДЫ ОШИБОК

EPERM **settimeofday** вызвана без прав суперпользователя.

EINVAL Неправильно указан часовой пояс (или какие-то другие параметры).

EFAULT Или *tv* или *tz* указывают на недоступную область адресного пространства.

ЗАМЕЧАНИЯ

Прототип **settimeofday** и определения для **timercmp**, **timerisset**, **timerclear**, **timeradd**, **timersub** доступны (с glibc2.2.2) только если определно **_BSD_SOURCE** (либо явно, либо неявно, неопределением _POSIX_SOURCE или сборкой с флагом -ansi).

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, BSD 4.3
СМ. ТАКЖЕ **date(1), adjtimex(2), time(2), ctime(3), ftime(3)**
Linux 2.0.32 10 December 1997 GETTIMEOFDAY**(2)**

GETUID

НАЗВАНИЕ getuid, geteuid - считывает идентификатор пользователя процесса
СИНТАКСИС

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

uid_t getuid(void);
uid_t geteuid(void);
```

ОПИСАНИЕ **getuid** возвращает идентификатор действительного пользователя текущего процесса. **geteuid** возвращает идентификатор эффективного пользователя текущего процесса. Действительный идентификатор соответствует идентификатору вызвавшего процесса. Фактический идентификатор соответствует биту setuid на исполняемом файле.
СООТВЕТСТВИЕ СТАНДАРТАМ POSIX, BSD 4.3.
СМ. ТАКЖЕ **setreuid(2), setuid(2).**
Linux 0.99.11 23 July 1993 GETUID**(2)**

IDLE

НАЗВАНИЕ idle - заставляет нулевой процесс работать "вхолостую"
СИНТАКСИС

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

ОПИСАНИЕ **idle** является внутренним вызовом, который посылается во время запуска процесса. Он разрешает подкачку своих страниц, уменьшает их приоритет и входит в рабочий цикл. Значение **idle** никогда не возвращается.
Только нулевой процесс может произвести вызов **idle**. В любом пользовательском процессе, даже если есть права суперпользователя, произойдет ошибка **EPERM**.
ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ
idle никогда не возвращается к нулевому процессу, а пользовательскому процессу всегда возвращает -1.
КОДЫ ОШИБОК **EPERM** Всегда возникает, если процесс является пользовательским.
СООТВЕТСТВИЕ СТАНДАРТАМ
Команда является специфичной (только для Linux) и не должна использоваться программами, пернесенными с других платформ, или программами, предназначенными для переноса в другие системы.
ЗАМЕЧАНИЯ В версии 2.3.13 (и более поздних) этого вызова больше не существует.
Linux 1.1.46 21 August 1994 IDLE**(2)**

INTRO

НАЗВАНИЕ intro - введение в системные вызовы
ОПИСАНИЕ В этой главе описываются системные вызовы Linux. Список 164-х системных вызовов в Linux 2.0 находится в **syscalls(2).**

Прямой вызов
В большинстве случаев не требуется посылать системные вызовы напрямую, но бывают случаи, когда Стандартная Библиотека Языка C (libc) не реализует какую-нибудь полезную функцию.

Синтаксис

#include <linux/unistd.h> A _syscall macro желаемый системный вызов

Настройка

Очень важно знать прототип функции. Вам нужно знать, сколько у этого вызова аргументов, каковы их типы и тип, возвращаемый функцией. Есть шесть макросов, облегчающих системные вызовы. Они имеют следующий вид:

_syscall*X*(*type*,*name*,*type1*,*arg1*,*type2*,*arg2*,...)

, где *X* принимает значение от 0 до 5 и указывает на количество аргументов, принимаемых системным вызовом

type - это тип, возвращаемый системным вызовом ;

name - название системного вызова;

typeN - тип N-ого аргумента ;

argN - имя N-ого аргумента

Эти макросы создают функцию с заданными аргументами, которая называется *name*. После того, как Вы включите макрос _syscall() в свой файл с исходным кодом, Вы можете вызывать функции системы, пользуясь именем *name*.

ПРИМЕР

```
_syscall1(int, sysinfo, struct sysinfo *, info);
/* Замечание: если Вы копируете тексты напрямую из исходных
текстов nroff, помните, что необходимо УДАЛИТЬ все лишние
обратные слэши в выражении printf. */
int main(void)
{
     struct sysinfo s_info;
     int error;
     error = sysinfo(&s_info);
     printf("code error = %d\n", error);
        printf("Uptime = %ds\nLoad: 1 min %d / 5 min %d / 15 min %d\n"
                "RAM: total %d / free %d / shared %d\n"
                "Memory in buffers = %d\nSwap: total %d / free %d\n"
                "Number of processes = %d\n",
          s_info.uptime, s_info.loads[0],
          s_info.loads[1], s_info.loads[2],
          s_info.totalram, s_info.freeram,
          s_info.sharedram, s_info.bufferram,
          s_info.totalswap, s_info.freeswap,
          s_info.procs);
     return(0);
}
```

ПРИМЕР ВЫВОДА

```
code error = 0
uptime = 502034s
Load: 1 min 13376 / 5 min 5504 / 15 min 1152
RAM: total 15343616 / free 827392 / shared 8237056
Memory in buffers = 5066752
Swap: total 27881472 / free 24698880
Number of processes = 40
```

ЗАМЕЧАНИЯ

Макросы _syscall() НЕ создают прототипа. Вам может потребоваться создать его вручную, особенно в программе на C++.

Системные вызовы не обязательно возвращают только положительные или отрицательные коды ошибок. Чтобы выяснить настоящее положение дел, может потребоваться обратиться к исходным текстам. Обычно код ошибки - это стандартный код ошибки со знаком минус, например, **-EPERM**.

Макросы _syscall() возвращают результат системного вызова

r, если *r* неотрицательно, а в противном случае возвращают -1 и устанавливают переменную *errno* в значение *-r*. Коды ошибок описаны в **errno(3)**.
Некоторые системные вызовы, например **mmap**, требуют больше пяти аргументов. Они обрабатываются путем помещения аргументов в стек и передачи указателя на блок аргументов. При описании системного вызова аргументы ДОЛЖНЫ передаваться "по значению" или с помощью указателя (для агрегатных типов, например, структур).

СООТВЕТСТВИЕ СТАНДАРТАМ

Для обозначение вариантов Unix и разнообразных стандартов, которым соответствуют системные вызовы, описанные в этой секции руководства, используются различные сокращения:

SVr4 Unix-System V Release 4, описанная в "Programmer's Reference Manual: Operating System API (Intel processors)" (Prentice-Hall 1992, ISBN 0-13-951294-2)

SVID System V Interface Definition (Описание Интерфейса Системы V), описанное в "The System V Interface Definition, Fourth Edition", доступное по адресу ftp://ftp.fpk.novell.com/pub/unix-standards/svid в формате Postscript.

POSIX.1
IEEE 1003.1-1990 часть 1, также известный как ISO/IEC 9945-1:1990s, также известный как "IEEE Portable Operating System Interface for Computing Environments" (Интерфейс Переносимой Операционной Системы для Вычислительных Сред IEEE), разъясненный в книге Donald Lewine "POSIX Programmer's Guide" (O'Reilly & Associates, Inc., 1991, ISBN 0-937175-73-0.

POSIX.1b
IEEE Std 1003.1b-1993 (стандарт POSIX.1b), описывающий возможности работы в реальном времени под переносимыми операционными системами, разъясненный в книге Bill O. Gallmeister "Programming for the real world - POSIX.4" (O'Reilly & Associates, Inc. ISBN ISBN 1-56592-074-0).

4.3BSD/4.4BSD
Версии 4.3 и 4.4. дистрибуции Berkeley Unix. 4.4BSD была обратно совместима с 4.3.

SUS, SUSv2
Single Unix Specification. (Разработана X/Open и The Open Group. Смотри также http://www.UNIX-systems.org/version2/.)

V7 Версия 7, исходная версия Unix от Bell Labs.

ФАЙЛЫ */usr/include/linux/unistd.h*
СМ. ТАКЖЕ **errno(3)**
Linux 1.2.13 22 May 1996 INTRO**(2)**

IOCTL

НАЗВАНИЕ ioctl - управляет устройствами
СИНТАКСИС

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...);
```

["Третий" аргумент обычно **char ***argp*, и далее он будет так называться.]

ОПИСАНИЕ Функция **ioctl** управляет устройствами, на которые ссылаются специальные файлы. При помощи запросов **ioctl** можно управлять многими символьными устройствами (например, терминалами). Аргумент *d* должен представлять собой открытый описатель файла. Параметр *request* определяет, является ли третий аргумент *входным* или *выходным*, а так же размер *argp* в байтах. Макросы и определения, использованные для вызова ioctl *request*, находятся в файле *<sys/ioctl.h>*.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Обычно, при успешном завершении возвращается нулевое значение. Некоторые ioctl используют возвращаемое значение в качестве выводимого параметра и при успешном завершении возвращают неотрицательное значение. При ошибке возвращаемое значение равно -1, а *errno* присваиваются соответствующие значения.

КОДЫ ОШИБОК

EBADF *d* является неправильным описателем.

EFAULT *argp* ссылается на недоступную область адресного пространства.

ENOTTY *d* не соответствует символьному устройству.

ENOTTY указанный запрос не применим к виду объекта, на который ссылается описатель *d.*

EINVAL *Request* или *argp* неправильно указаны.

СООТВЕТСТВИЕ СТАНДАРТАМ

Нет единого стандарта. Аргументы, возвращаемые значения и семантика **ioctl(2)** различны в каждом конкретном случае в зависимости от драйвера устройства (вызов является общим для всех операций, что не совсем соответствует поточной модели ввода/вывода в Unix). Прочтите список **ioctl_list(2)** , в котором указано большинство вызовов **ioctl**. Вызов **ioctl** впервые появился в версии 7 AT&T Unix.

СМ. ТАКЖЕ execve(2), fcntl(2), ioctl_list(2), mt(4), sd(4), tty(4)

BSD Man Page 23 July 1993 IOCTL**(2)**

IOPERM

НАЗВАНИЕ ioperm - устанавливает права на работу с портами ввода/вывода

СИНТАКСИС

#include <unistd.h> /* для libc5 */
#include <sys/io.h> /* для glibc */

int ioperm(unsigned long *from*, **unsigned long** *num*, **int** *turn_on***);**

ОПИСАНИЕ Ioperm устанавливает права доступа процесса к портам ввода-вывода *num*, начиная с порта **from**. Для использования функции **ioperm** необходимы права root. Таким образом, можно задать права доступа только к первым портам 0x3ff. Для работы с другими портами необходимо использовать функцию **iopl** . Права не наследуются по fork, но наследуются по exec. Это можно использовать для предоставления доступа к портам ввода-вывода непривилегированным процессам.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращаемое значение равно 0. При ошибке оно равно -1, а переменная *errno* приобретает соответствующее значение.

СООТВЕТСТВИЕ СТАНДАРТАМ

ioperm - это функция, специфичная для Linux. Не рекомендуется использовать ее в программах, переносимых на другие системы.

ЗАМЕЧАНИЯ Libc5 рассматривает данную функцию как системный вызов, и поэтому в *<unistd.h>* есть ее прототип. В Glibc1 этого прототипа нет. В Glibc2, в *<sys/io.h>* и в *<sys/perm.h>*

этот прототип есть. Не используйте второй вариант, он существует только в версии i386.

СМ. ТАКЖЕ **iopl(2)**

Linux January 21, 1993 IOPERM**(2)**

IOPL

НАЗВАНИЕ iopl - меняет уровень привилегий ввода-вывода

СИНТАКСИС

#include <sys/io.h>

int iopl(int *level***);**

ОПИСАНИЕ **iopl** изменяет уровень привилегий ввода/вывода текущего процесса в соответствии с *level*. Этот вызов необходим для того, чтобы 8514-совместимые X-серверы могли работать под управлением Linux. Этим X-серверам необходим доступ ко всем 65536-и портам ввода/вывода, соответственно, им недостаточно вызова **ioperm** . В дополнение к тому, что на высоком уровне привилегий процессу разрешен неограниченный доступ к портам ввода/вывода, он может также запретить системные прерывания. Скорее всего, это приведет к сбою системы, поэтому использование этой возможности не рекомендуется. Эти права наследуются через fork и exec. Стандартный уровень привилегий ввода/вывода обычного процесса равен 0.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении работы возвращаемое значение равно 0. При ошибке оно равно -1, а переменная *errno* приобретает соответствующее значение.

КОДЫ ОШИБОК

EINVAL Значение *level* больше 3.

EPERM Пользователь процесса - не суперпользователь.

СООТВЕТСТВИЕ СТАНДАРТАМ

iopl - это функция, специфичная для Linux. Не рекомендуется использовать ее в программах, переносимых на другие системы.

ЗАМЕЧАНИЯ

Libc5 воспринимает эту функцию как системный вызов и имеет прототип в *<unistd.h>*. Glibc1 не имеет прототипа этой функции. Glibc2 имеет прототип в *<sys/io.h>* и в *<sys/perm.h>*. Не используйте последнее, этот файл имеется только на i386.

СМ. ТАКЖЕ **ioperm(2)**

Linux 0.99.11 24 July 1993 IOPL**(2)**

IPC

НАЗВАНИЕ ipc - системные вызовы IPC

СИНТАКСИС

int ipc(unsigned int *call*, **int** *first*, **int** *second*, **int** *third*, **void *****ptr*, **long** *fifth*);

ОПИСАНИЕ **ipc()** - это общая точка входа системных вызовов System V IPC в ядре (для сообщений, семафоров и разделяемой памяти). *call* определяет, какую функцию IPC Вы хотите использовать; остальные аргументы передаются соответствующей функции.

Пользовательские программы должны задавать соответствующие функции напрямую. Знать об **ipc()** необходимо только создателям стандартных библиотек и отладчикам ядра.

СООТВЕТСТВИЕ СТАНДАРТАМ

ipc() - это функция, специфичная для Linux. Не рекомендуется использовать ее в программах, переносимых на другие системы.

СМ. ТАКЖЕ **msgctl(2), msgget(2), msgrcv(2), msgsnd(2), semctl(2), semget(2), semop(2), shmat(2), shmctl(2), shmdt(2),**

KILL

НАЗВАНИЕ kill - функция, с помощью которой посылается сигнал процессу

СИНТАКСИС

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

int kill(pid_t pid, int sig);
```

ОПИСАНИЕ Системный вызов **kill** используется для того, чтобы послать любой сигнал любому процессу или группе процессов.

Если *pid* больше 0, то сигнал *sig* посылается процессу *pid*.

Если *pid* равен 0, то сигнал *sig* посылается всем процессам текущей группы.

Если *pid* равен -1, то сигнал *sig* посылается всем процессам, кроме процесса 1 (init). Также смотрите замечания ниже.

Если *pid* меньше -1, то сигнал *sig* посылается всем процессам группы *-pid*.

Если *sig* равен 0, то сигнал не посылается, но выполняется проверка на возникновение ошибок в процессе.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении вызова возвращаемое значение равно 0. При ошибке оно равно -1, а переменной *errno* присваивается соответствующее значение.

КОДЫ ОШИБОК

EINVAL Задан неправильный тип сигнала.

ESRCH Процесс или группа не существуют. Заметьте, что существующий процесс может быть зомби-процессом, который уже подал запрос на завершение работы, но еще не прошел через функцию **wait()**.

EPERM У процесса нет достаточных прав для того, чтобы послать сигналы одному из процессов-получателей. Для того, чтобы процесс мог послать сигнал процессу *pid*, он должен либо иметь привилегии root, либо его реальный или эффективный идентификатор пользователя должен быть равен реальному или сохраненному при помощи set-user-ID идентификатору пользователя процесса-получателя. В случае с SIGCONT, для посылки и приема сигнала процессу достаточно быть частью этой же сессии.

ЗАМЕЧАНИЯ Невозможно послать сигнал процессу номер 1 (init), потому что у него нет обработчика сигналов. Это сделано для того, чтобы не было случайных сбоев системы.

POSIX 1003.1-2001 требует, чтобы *kill(-1,sig)* посылал *sig* всем процессам, которым текущий процесс может послать сигнал, кроме, возможно, системных процессов, специфичных для текущей реализации. Linux позволяет процессу послать сигнал самому себе, однако *kill(-1,sig)* не посылает сигнал текущему процессу.

СООТВЕТСТВИЕ СТАНДАРТАМ SVr4, SVID, POSIX.1, X/OPEN, BSD 4.3, POSIX 1003.1-2001

СМ. ТАКЖЕ **_exit(2), exit(3), signal(2), signal(7)**

Linux 2.5.0 2001-12-18 KILL**(2)**

-