## Язык С как стандарт взаимодействия с ОС

**Замечание**

Хотя 32-битное ПО может работать на процессорах х86-64, *64-битное ПО не может работать на процессорах х86*.

**Общеизвестно**, что язык ***С*** рассматривается в двух контекстах:

* *как высокоуровневый инструмент* для написания самих ОС;
* *как низкоуровневый инструмент* для написания прикладных программ, взаимодействующих с ядром ОС.

Синтаксис языка не содержит даже собственных конструкций ввода-вывода, функцио- нальное окружение его, реализованное прежде всего в виде библиотеки *libc*, явля- ется неотъемлемой частью ОС и *изменяется после перекомпиляции ее ядра*.

**Как следствие** указанных особенностей языка ***С***, его реализация для целей программирования сильно привязана как к аппаратной платформе ЭВМ, так и к архитектуре ПО ОС. Это, в свою очередь, порождает проблемы переносимости уже разработанного ПО на разные архитектуры, а также разработку инструментальных средств для этого языка.

**Стремление повысить мобильность ПО**, которое пришется на языке С, привело к разработе стандарта **POSIX** (*Portable Operating System Interface*), призванного стабилизировать описание интерфейсов различных операционных систем.

Само название **POSIX** (1988 год) было предложено известным специалистом, являющимся основателем «*Фонда свободного программного обеспечения*», - Ричар- дом Столмэном. Наиболее современная версия стандарта POSIX, ***в редакции 2003 г.***, основана на Техническом стандарте *Open Group IEEE Std 1003.1* и на междуна- родном стандарте *ISO/IEC 9945* (версии до 2001 года являются устаревшими).

**По состоянию на 2001 год**, стандарт содержал следующие *четыре части*:

1. *Основные определения* (термины, концепции и интерфейсы, общие для всех частей);
2. *Описание прикладного программного C-интерфейса* к системным сервисам;
3. *описание интерфейса* к системным сервисам на уровне командного языка и служебных программ;
4. *Детальное разъяснение* положений стандарта, обоснование принятых реше- ний;
5. *В дальнейшем*, многие мелкие исправления накапливались и были внесены *в редакцию 2003 года*.

Основные идеи этого стандарта описываются *множеством базовых, системных сервисов*, необходимых для функционирования прикладных программ. **Доступ к этим сервисам** предоставляется посредством интерфейса, который был специфи- цирован для языка ***C***, командного языка и других общеупотребительных служебных программ. Здесь следует подчеркнуть, что у каждого интерфейса есть две стороны: *вызывающая* и *вызываемая*.

**Стандарт POSIX** ориентирован в первую очередь на *вызывающую сторону*, что делает его полезным как для системных, так и прикладных программистов.

**Цель стандарта POSIX** - сделать приложения мобильными на уровне исход- ного кода языка. Это значит, что, при переносе программ языка ***С*** на другую опера-ционную платформу, *потребуется только новая компиляция исходных текстов программ*.

**Замечание** Поскольку, стандарт POSIX определяет только интерфейс к системным сервисам, то *он оставляет за рамками рассмотрения саму реализацию интерфейса*. В частности:

* *не различаются* системные вызовы и библиотечные функции;
* *не являются* объектом стандартизации средства администрирования, а также аппаратные ограничения и функции, которые необходимы только суперполь-

зователю.

**Ориентация POSIX** на международный стандарт языка ***C*** определила также и направление развития спецификаций POSIX, в плане синхронизации обоих стан- дартов:

* В стандарте проведено разделение на *обязательные* и *дополнительные функ- ции*.
* Особое внимание уделяется *способам реализации стандартизуемых функ- ций*, как в "классической" Unix-среде, так и на других операционных плат- формах, в сетевых и распределенных конфигурациях.

**В редакции 2003-го года**, стандарт POSIX рассматривает следующие катего- рии системных компонентов:

* средства разработки;
* сетевые средства;
* средства реального времени;
* потоки управления;
* математические интерфейсы;
* пакетные сервисы;
* заголовочные файлы;
* унаследованные интерфейсы.

Именно такой, общий перечень интерфейсов должна предоставлять каждая

операционная система для работы любого приложения.

*Реализация* или *операционная система*, соответствующая стандарту POSIX, должна поддерживать *все обязательные служебные программы, функции, а также заголо-вочные файлы с обеспечением специфицированного в стандарте поведения*.

Для этих целей используется константа ***\_POSIX\_VERSION***, которая имеет значение ***200112L***.

*ОС также может* предоставлять *возможности, помеченные стандартом в качестве дополнительных* или *содержать нестандартные функции*. Если утверж- дается, что поддерживается некоторое расширение, то это должно производиться *непротиворечивым образом*, для всех необходимых частей и так, *как описано в стандарте*. Для этого, в заголовочном файле **<unistd.h>** должны быть *определены константы, соответствующие всем поддерживаемым необязательным возмож- ностям*. Например, константа ***\_POSIX2\_C\_DEV*** обслуживает средства разработки программ на языке C.

**Анализируя** эти константы, *во время компиляции*, система разработки выяс- нит возможности используемой ОС и подстроится под них.

**Аналогичные действия** могут быть выполнены с помощью:

* функции *long sysconf(int name)*, - во время выполнения программы;
* служебной программой *getconf*, - посредством запуска ее в командной строке или в сценарии языка shell.

**Для минимизации** размеров ОС и приложений, стандартом POSIX была пре- дусмотрена весьма мелкая гранулярность *необязательных возможностей*. Было проведено объединение *взаимосвязанных необязательных возможностей* в группы:

* шифрование;
* средства реального времени;
* продвинутые средства реального времени;
* потоки реального времени;
* продвинутые потоки реального времени;
* трассировка;
* ПОТОКИ;
* унаследованные возможности.

**В документации** на ОС должны быть отражены вопросы соответствия стан-

дарту POSIX, описаны поддерживаемые дополнительные и нестандартные возмож- ности.

**Применительно непосредственно к ОС**, определены ряд основных поня- тий, соответствующих стандарту POSIX:

* пользователь;
* файл;
* процесс;
* терминал;
* хост;
* узел сети;
* время;
* языково-культурная среда.

**Это** - первичные понятия, которые строго не определяются, а поясняются с

помощью других понятий и отношений. Для каждого из них описаны присущие им атрибуты и применимые к ним операции. Содержатся пояснения следующих *ос- новных понятий*:

1. *У пользователя* есть имя и числовой идентификатор.
2. *Файл* - объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. К числу последних относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.
3. *Процесс* - адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
4. *Терминал* (или терминальное устройство) - символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.
5. *Сеть* - совокупность взаимосвязанных хостов.
6. *Языково-культурная среда* - часть пользовательского окружения, которая зависит от языковых и культурных соглашений.

**Для работы**, с большим числом сущностей, предоставляются *механизмы группирования* и *построения иерархий*. Существует:

* иерархия файлов;
* группы пользователей и процессов;
* подсети и другие.

**Для написания программ**, оперирующих с сущностями POSIX-совмести-

мых систем, применяются или командный интерпретатор (языка shell) и/или компи- лируемый язык C:

* *в первом случае*, приложение может пользоваться разными служебными прог- раммами (утилитами);
* *во втором*, - функциями.

**Функциональный интерфейс ОС** всегда считается *первичным*, но, в POSIX- совместимых ОС, определены объекты, которые считаются *вспомогательными*. Такие объекты обеспечивают организацию взаимодействия между основными сущ- ностями. Примерами таких объектов являются *средства межпроцессного взаимо- действия*, которые *выполняются в определенном окружении*. Частью такого окру- жения является *языково-культурная среда (Locale)*, которая образованная такими категориями, как:

* *символы и их свойства;*
* *форматы сообщений;*
* *дата и время;*
* *числовые и денежные величины*.

*Каждый пользовательский процесс ОС ассоциирован*, по крайней мере, на три файла:

* стандартный ввод;
* стандартный вывод;
* стандартный протокол.

**Обычно**, стандартный ввод назначается на клавиатуру терминала, а стан-

дартный вывод и стандартный протокол - на экран:

* со стандартного ввода читаются команды и (иногда) исходные данные для них;
* на стандартный вывод поступают результаты выполнения команд;
* в стандартный протокол помещаются диагностические сообщения.

**Замечание** К ОС, также могут предъявляться качественные требования, например, требование под- держки реального времени: способность обеспечить необходимый сервис в течение за- данного отрезка времени.

**Стандарт POSIX** также определяет ряд требований к среде компиляции POSIX-совместимых приложений. Часто, разработка приложений ведется в *кросс- режиме*. Поэтому на каждой инструментальной платформе создается такая среда компи-ляции приложений, чтобы результат этой компиляции можно было перенес- ти для последующего выполнения на целевую платформу.

**Важнейшая часть среды компиляции** - заголовочные (или включаемые) файлы, содержащие прототипы функций, определения символических констант, макросов, типов данных, структур и т.п.

**Для каждой**, описанной в стандарте POSIX функции, определено, где и какие заголовочные файлы должны быть включены использующим ее приложением. Пос- редством символических констант, определенных в заголовочном файле ***<****unistd.h****>***, операционная система предоставляет приложению информацию о поддерживаемых возможностях.

**Стандартом POSIX**, также предусмотрен симметричный механизм, называе- мый механизмом макросов проверки возможностей. Он позволяет приложениям объявлять о своем желании получить доступ к определенным прототипам и име- нам.

**С целью** не допустить пересечения имен, в заголовочных файлах исполь- зуются префиксы ***posix\_***, ***POSIX\_*** и ***\_POSIX\_***, которые зарезервированы для нужд стандарта.

**Замечание**

С подчеркивания, за которым следует еще одно подчеркивание или заглавная латинская буква, могут начинаться только системные, но не прикладные имена.

**Для включаемых файлов** описаны префиксы используемых в них имен.

Например:

* *для операций управления файлами*, фигурирующих в *<fcntl.h>*, в качестве префиксов задействованы ***F\_, O\_, S\_***;
* *у средств межпроцессного взаимодействия*, описанных в файле *<sys/ipc.h>*, префиксом служит ***IPC\_***;
* *для манипулирования характеристиками терминалов* в файле *<termios.h>* определено множество разнообразных имен: EXTB, VDSUSP, DEFECHO, FLUSHO и другие;
* еще имеется *четыреста семнадцать* имен типа *\_Exit, abort, abs, acos* и дру- гие, которые могут участвовать в редактировании внешних связей приклад- ной программы.

*Мобильность приложений*, соответствующих стандарту POSIX, принципи- ально достижима благодаря двум основным факторам:

* *во-первых*, - наличию огромного числа стандартизованных системных серви- сов;
* *во-вторых*, - возможности динамического выяснения характеристик целевой платформы и подстройки под них приложения.

Приложения, соответствующие стандарту POSIX, могут быть *одно****-*** и *много****-*** *процессными*, с возможностью динамической адаптации конфигурации к свойствам целевой платформы. Далее перечислим основные важные элементы стандартиза- ции без подробного объяснения.

**Стандартизованы средства** *порождения и завершения процессов, смены их программ, опроса и/или изменения разнообразных характеристик*. Процессы мож- но приостанавливать и активизировать в заданное время.

Необходимая *степень детерминизма выполнения приложений* достигается благодаря *средствам поддержки реального времени*, к которым относятся управ- ление дисциплиной выделения процессоров, сигналы реального времени, удержа- ние страниц в оперативной памяти, таймеры высокого разрешения и другие.

**Функции для работы с файлами** удовлетворяют потребности приложений в чтении и записи долговременных данных, защите таких данных от несанкциониро- ванного доступа:

* *Механизм блокировки фрагментов файлов* позволяет обеспечить *атомар- ность транзакций*.
* *Асинхронный ввод/вывод* дает возможность совмещать операции обмена, оп- тимизируя тем самым приложения.
* *С помощью множества служебных программ* можно относительно легко организовать сложную обработку данных.
* *Тщательно проработаны вопросы доступа* к внешним устройствам, которые подсоединены по последовательным линиям, особенно к терминалам.

**Стандартизованный командный язык** shell обеспечивает адекватное сред- ство для написания небольших мобильных процедур и их быстрой интерактивной отладки.

**Выделены механизмы конвейеров**, позволяющие объединять команды в цепочки с фильтрацией промежуточных результатов.

**Служебные программы** образуют развитую среду выполнения для shell- процедур.

**Фоновый режим** выполнения программ позволяет организовать одновре- менное выполнение нескольких программ и взаимодействие с ними.

**Стандарт POSIX** стандартизует интерфейс командной строки. Вероятно, в будущих версиях стандарта будет регламентирован графический интерфейс.

**Для многопользовательских систем**, POSIX регламентирует различные средства *непосредственного и почтового обмена информацией*.

**Стандарт POSIX** - обязательный элемент современной дисциплины разра- ботки прикладных систем.

## Системные операции для работы с файловой системой

Теперь перейдем к изучению основных *системных вызовов*, разделенных на две группы

* управление файлами;
* управление каталогами и файловыми системами.

Изучение назначения и возможностей этих функций является предметной целью

данной темы.

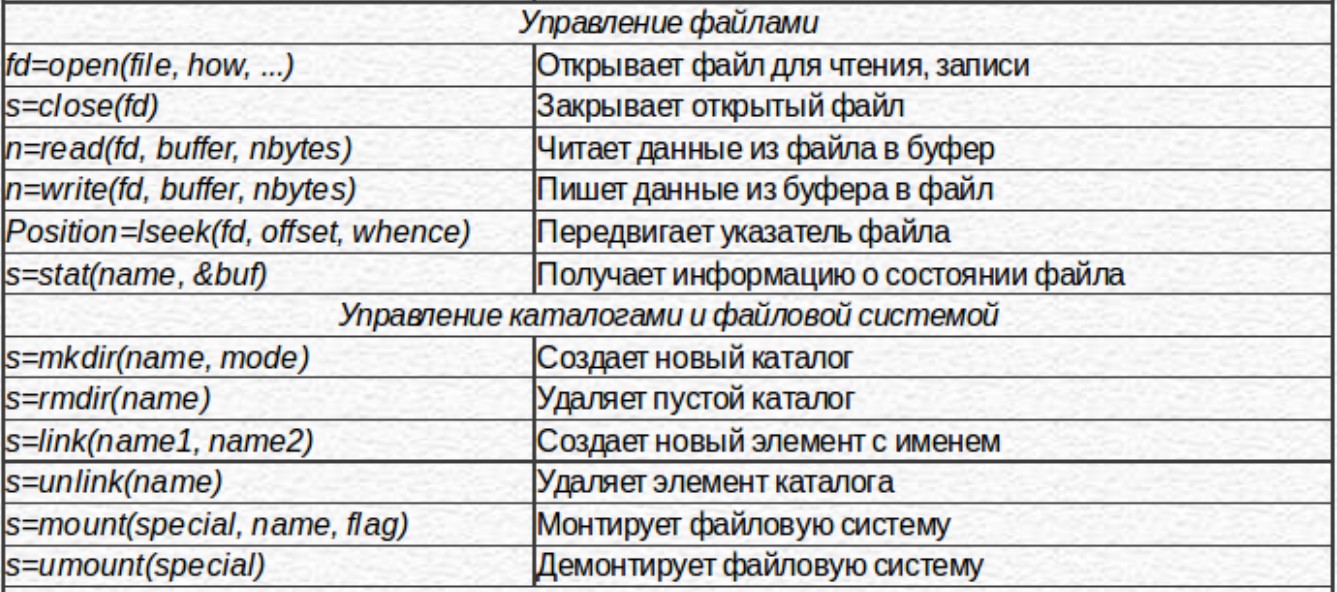


Рисунок 1.5 — Системные функции подсистемы ввода-вывода

**Замечание**

Отметим, что наряду с *низкоуровневыми системными вызовами*, приведенными на ри- сунке 1.5, для работы с файлами и файловыми системами используются *более высо- коуровневые функции*, которые обеспечивают буферизованный ввод-вывод и подробно изложены в любом учебнике по языку С. Считается, что студент уже имеет опыт работы с ними и, при необходимости, использует учебник [5] как справочное пособие.

В данном подразделе будут изучены только первые шесть функций, которые касаются *непосредственно управления файлами*.

Все системные функции ввода-вывода используют *целочисленное значение дескриптора файла*. Это значение формируется в ядре ОС и индивидуально привя- зывается к конкретному процессу.

Нумерация значений дескрипторов для каждого процесса начинается с нуля и определяется в момент открытия файлов, из условия *минимального значения дос- тупного (незанятого) дескриптора*.

Когда файл закрывается, то соответсвующий ему *номер дескриптора осво- бождается* и *используется для открытия других файлов*.

Все остальные системные функции, кроме функции *stat(...)*, используют уже открытый дескриптор файла.

### Системные вызовы open() и close()

Системный вызов *open(...)* открывает некоторый файл и, в случае успешного завершения, возвращает целочисленный номер дескриптора.

Допускается использование следующих аргументов:

*pathname* — строковый указатель на имя файла, заданного в соответсвии с требова- ниями конкретной ОС;

*flags* — целочисленное значение, определяющее режимы открытия файла, которые обычно задаются *конкатенацией битовых масок*:

* *O\_APPEND* — добавление информации в конец файла;
* *O\_CREAT* — создание файла, если он отсутсвует;
* *O\_RDONLY* — только для чтения из файла;
* *O\_WRONLY* — только для записи в файл;
* *O\_RDWR* — для чтения и записи в файл;
* *O\_TMPFILE* — после закрытия файл будет удален, а также *многие другие*. *mode* — целочисленное значение, определяющее режимы доступа к файлу,

например, восьмеричное значение *0666* разрешает доступ на чтение и запись всем пользователям.

В случае ошибочного завершения, функция *open(...)* возвращает отрицатель- ное значение -1, а переменная *errno* содержит код ошибки, котрый следует анали- зировать отдельно.

**Замечание**

Функция *open(...)* содержит множество режимов открытия файлов, которые могут быть несовместимы друг с другом, а также множество вариантов сообщений об ошибках,

Системный вызов *close(...)* закрывает открытый файл и освобождает номер дескриптора для последующего использования.

ние 0.

В случае успешного завершения вызова, возвращается целочисленное значе-

В случае ошибки — целочисленное значение -1, а переменная *errno* содержит

код ошибки:

* *EBADF* — аргумент функции не является правильным дескриптором файла;
* *EINTR* — закрытие файла было прервано сигналом;
* *EIO* — общая ошибка ввода-вывода.

**Замечание**

Несмотря на кажущуюся простоту, системный вызов *close(...)* выполняет очень важные

функции:

* *он универсален*, поскольку применим ко всем видам файлов;
* *корректно завершает работу с файлом*, записывая при необходимости все данные на устройство ввода-вывода;
* *неявно вызывается* ядром ОС, когда процесс завершает свою работу.

### Системные вызовы read() и write()

Системный вызов *read(...)* читает заданное число байт из файла, определен- ного его дескриптором. В случае успешного завершения операции чтения, данные помещаются во внешний буфер и возвращается реальное число прочитанных байт.

Для вызова этой функции используется три аргумента:

* *fd* — целочисленный номер дескриптора открытого файла;
* *buf* — указатель на внешний буфер, в который записываюся читаемые байты;
* *count* — количество байт запрашиваемых для чтения из файла. Функция *read(...)* читает байты с текущей позиции файла:
* если смещение текущей позиции равно или больше размера файла, то возвра- щается значение 0;
* если данные недоступны, но конец файла не достигнут, то ожидается поступ- ление данных, что в общем случае определяется режимом открытия файла;
* в случае ошибки чтения, возвращается значение -1 и необходимо анализиро- вать переменную *errno*;
* если размер параметра *count* превышает значение константы *SSIZE\_MAX*, то результат выполнения операции не специфицирован.

**Замечание**

При использовании функции *read(...)* необходимо анализировать системные ограниче- ния конкретной ОС: максимальный размер выделяемого буфера для записи читаемых данных и максимальное число читаемых данных за одну операци.

Системный вызов *write(...)* записывает заданное число байт из буфера в файл, определенный его дескриптором. В случае успешного завершения операции, дан- ные помещаются в файл, начиная с текущей позиции, и возвращается реальное чис- ло записанных байт.

В этом системном вызове используются теже аргументы, что и в рассмот- ренной ранее функции *read(...)*. В случае ошибки записи, возвращается значение -1 и необходимо анализировать переменную *errno*.

Если данные нужно поместить в файл с нужной позиции, то предварительно нужно использовать системный вызов

Дополнительно, на результат вывода могут влиять различные системные ограничения ОС, например, *максимальный размер файлов*, поддерживаемых той или иной файловой системой. Ряд таких значений можно посмотреть с помощью утилиты ***prlimit***, которая позволяет как получать так и устанавливать ряд лимитов процессов ОС.

Демонстрацию применения изученных в данном подразделе системных функций проведем на примере задачи «*Чтение и анализ структуры MBR*».

Напомним, что ***MBR*** является *заголовочной структурой* каждого накопителя типа «*винчестер*», совместимая с функциональными возможностями BIOS ЭВМ. В новых системах, которые cодерживают ***UEFI*** вместо ***BIOS***, возможна поддержка накопителей со структурой ***GPT***, куда MBR входит как вспомогательная структура, предназначенная для защиты блочного устройства.

MBR содержит *6 основных частей*:

* *код* System Bootstrap — 446 байт;
* *четыре описателя* разделов по 16 байт;
* *сигнатуру* MBR — 2 байта.

**Замечание** Запуск всех программ для работы с устройствами должен проводиться от имени пользователя *root*, иначе они будут завершаться из-за отсутствия прав доступа к этим устройствам.

### Системный вызов lseek()

Системный вызов *lseek(...)* устанавливает позицирование в файле, опреде- ленном его дескриптором. В случае успешного завершения операции, возвращается *место новой позиции в байтах* от начала файла.

Для вызова этой функции используется три аргумента: *fd* — целочисленный номер дескриптора открытого файла; *offset* — целочисленное значение смещения в байтах; *whence* — директива, задающая интерпретацию смещения:

* *SEEK\_SET* = 0 - смещение устанавливается в *offset* байт;
* *SEEK\_CUR* = 1 - смещение устанавливается как текущее смещение плюс

*offset* байт;

* *SEEK\_END* = 2 - смещение устанавливается как размер файла плюс *offset*

байт.

В случае ошибки, функция *lseek(...)* возвращает значение -1 и необходимо анализировать переменную *errno*:

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

## Создание специальных файлов

Сама ***VFS*** реализована в ядре ОС и все операции с файлами осуществляются в пространстве ее представлений. Фактически, ***VFS***, является *виртуальным коммутатором* всех внешних файловых систем, подклю- ченных (монтированных) к ней.

**В идейном плане**, структура ***VFS*** очень близка к структуре ***EXT2FS***, которая и является прямым ее прототипом.

**Напомним**, что структурную основу ***EXT2FS***, как и других файловых систем Linux/UNIX, играет *таблица индексных дескрипторов* (Inode Tables), каждая строка которой содержит полное описание файла, кроме его имени:

* *когда файл создается* в файловой системе, то ему выделяется отдельная строка в Inode Tables;
* *номер этой строки* — *i-узел* (*inode*) и используется для дальнейшей работы с файлом, а *имя файла* и *значение его i-узла* записываются в соответствующий

файл директории, имеющий записи переменной длинны (см. далее таблицу 1.1);

* *когда файл открывается*, то ***VFS*** находит по имени файла значение *i-узла*,

запоминает его значение и возвращает пользователю *номер дескриптора файла*;

* *пользователь*, обращаясь к файлу, указывает его дескриптор файла, а ***VFS*** по номеру дескриптора находит значение *i-узла* файла.

Наряду с изученной ранее функцией *open(...)*, которая позволяет как откры- вать, так и создавать файлы, в стандарте POSIX имеется функция *mknod(...)*, обес- печивающая возможность создавать и различные специальные файлы.

**Замечание** Специальные файлы, после их создания, получают свои номера *i-узлов*, но, в отличие от обычных файлов, они в действительности есть только указатели на соответствующие драйверы устройств в ядре. По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствую- щее данному файлу: *класс устройства*, *старший номер устройства* и *младший номер устройства*.

*Класс устройства* фактически сообвествует понятию тип устройства, опре- деленному выше.

*Старший номер устройства*, группирует типы устройств, например, жест- кий диск или звуковая плата.

Файлы устройств одного типа имеют одинаковые имена и различаются по но- меру, прибавляемому к имени. Например, все файлы сетевых плат Ethernet имеют имена, начинающиеся на **eth:** eth0, eth1 и другие.

*Младший номер устройства* применяется для нумерации устройств одного типа, другими словами, - устройств с одинаковыми старшими номерами.

## Запрос информации о статусе файлов

Изучив основные особенности файлов, перейдем к рассмотрению системного вызова *stat(...)*, который *по заданному имени файла* позволяет получить информа- цию о свойствах любых типов файлов.

Дополнительно, стандарт POSIX предоставляет функции:

* *fstat(...)* - которая вместо имени файла использует *номер его дескриптора*;
* *lstat(...)* - которая использует *имя ссылки на файл* и возвращяет информацию о самой ссылке, а не о файле, на который она ссылается.

Ряд полей этой структуры требует уточнения:

* *st\_dev* — описывает устройство, на котором находится этот файл; поскольку это поле имеет тип *dev\_t*, то для работы с ним необходимо пользоваться мак- росами *major(…)*, *miner(...)* и *mkdev(...)*, описанными ниже;
* *st\_rdev* - описывает устройство, которое представляет этот файл (inode);
* *st\_size* - задает размер файла в байтах, если он обычный или является сим- вольной ссылкой; *размер символьной ссылки* - длина пути файла на который

она сылается, без конечного NUL;

* *st\_blocks* - задает размер файла *в 512-байтных блоках*: оно может быть мень- ше, чем *st\_size*/512, например, когда в файле есть пропуски;
* *st\_blksize* - задает "предпочтительный" размер блока для эффективного ввода/ вывода в файловой системе: запись в файл более мелкими порциями может

привести к некорректному *чтению/изменению/повторной* записи информации;

* *более точную семантику других полей* следует изучать по руководству ***man***.

**В стандарте** POSIX предусмотрены макросы, которые проверяют, является ли файл:

* S\_ISLNK(m) — символьной ссылкой;
* S\_ISREG(m) — обычным файлом;
* S\_ISDIR(m) — каталогом;
* S\_ISCHR(m) — символьным устройством;
* S\_ISBLK(m) — блочным устройством;
* S\_ISFIFO(m) — каналом FIFO;
* S\_ISSOCK(m) — сокетом.

**Для поля** *st\_mode*, в стандарте POSIX для программирования на языке С пре- дусмотрены также флаги

**В случае** успеха, все функции возвращают *ноль*.

**При обнаружении** ошибки возвращается *-1*, а переменной *errno* присваива- ется номер ошибки:

Поскольку поля *st\_dev* и *st\_rdev* имеют тип *dev\_t*, который может отличаться для разных аппаратных платформ, следует пользоваться макросами, присутству- ющими в система разработки большинства ОС:

## Каналы

Операционные системы предоставляют программисту два типа каналов:

* *полудуплексные каналы* UNIX;
* *именованные каналы* FIFO.

### Полудуплексные каналы UNIX

Изучая командный интерпретатор shell, мы уже сталкивались с понятием каналов. Например, если в командной строке запустить: ***ls -l | grep src***, то ин- терпретатор *sh* или *bash* будут выполнять следующую последовательность дейст- вий:

* будут одновременно запускатся утилиты *ls*, обеспечивающая вывод подроб- ного списка файлов текущей директории, и утилита *grep*, фильтрующая вход-

ной поток данных по наличию слова *src*, в каждой строке;

* перед вызовом утилит, shell создаст полудуплексный каканал, обеспечив воз- можность записи в него для утилиты *ls* и чтение из него для утилиты *grep*.

**Каналы** - старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX.

Они предоставляют метод односторонних коммуникаций между процессами: отсюда появился термин *half-duplex channel*.

Сам канал создается в ядре ОС. Когда процесс создает канал, ядро уста- навливает *два файловых дескриптора* для пользования этим каналом. Один такой дескриптор используется, чтобы открыть *путь ввода в канал* (запись), в то время как другой применяется *для получения данных из канала* (чтение).

Если открыт массив дескрипторов *int fd[2]* и процесс посылает данные через канал *fd[1]*, то он имеет возможность получить эту информацию из деск- риптора *fd[0]*.

Создание неименованного канала выполняется с помощью функции *pipe(...)*

Основное назначение *неименованных полудуплексных каналов* — *взаимодей- ствие между родительским и дочерними процессами*. В этом случае используется то свойство, что дочерний процесс наследует все ресурсы, открытые родительским

процессом, в том числе и открытые каналы.

**Если предположить**, что родительский процесс должен передать некоторое сообщение дочернему процессу, а потом дождаться его завершения, то в этом случае:

* родителскому процессу *не нужет дескриптор чтения из канала*;
* дочернему процессу *не нужен дескриптор записи в канал*.

**Замечание**

Следует помнить:

* с каналами не работает функция *lseek(...)*;
* системные вызовы не всегда корректно работают с консолями систем разработки, поэтому программы лучше запускать в терминале из командной строки.

### Именованные каналы FIFO

*Неименованные каналы* UNIX позволяют общаться только родственным друг другу процессам, которые получены с помощью *fork()*.

Для целей взаимодействия не родственных («*чужих*») процессов предназ- начены *именованные каналы FIFO*, которые также создаются в ядре ОС, но имеют *имена, отображаемые в файловой системе*.

Типичное применение каналов FIFO - разработка приложений «*клиент — сервер*», когда:

* *несколько процессов* могут записывать или читать FIFO *одновременно*;
* *режим работы* с FIFO - *полудуплексный*, что позволяет процессам общаться только в одном из направлений.

**Создать** канал FIFO можно с помощью функции *mknod(...)*, если во втором параметре *mode* установить тип файла *S\_IFIFO*, а третьему параметру *dev* присвоить значение *0*.

Стандарт POSIX предоставляет для создания именованных каналов специ- альный системный вызов *mkfifo(...)*

Перед использованием каналы FIFO требуют своего открытия с помощью системного вызова *open(…)*. Если при этом не указать режим *O\_NONBLOCK*, то:

* *открытие* FIFO блокируется и для записи, и для чтения;
* *при записи* канал блокируется до тех пор, пока другой процесс не откроет FIFO для чтения;
* *при чтении* канал снова блокируется до тех пор, пока другой процесс не запишет данные;
* *если FIFO закрыть для записи* через *close(...)* или *fclose(...)*, то это значит, что для чтения в FIFO помещается *EOF*.

Флаг открытия канала *O\_NONBLOCK* может использоваться *только при доступе для чтения*. При попытке открыть FIFO с *O\_NONBLOCK* для записи возникает ошибка открытия.

Если несколько процессов пишут в один и тот же FIFO, необходимо обратить внимание на то, чтобы сразу не записывалось больше, чем *PIPE\_BUF* байтов. Это необходимо, чтобы данные не смешивались друг с другом.

**Чтобы** не запускать множество программ, рассмотрим работу каналов FIFO на примере двух родственных процессов:

* *программа запускается* и создает канал FIFO с именем */dev/home/cfifo*, а за- тем выполняет *fork()*, создавая дочерний процесс;
* *родительский процесс* открывает канал на чтение, читает данные и выводит их на терминал; если прочитано менее 2-х байт, то — ожидает завершение дочернего процесса, а затем завершается сама;
* *дочерний процесс*, в цикле, читает строку данных с клавиатуры, открывает канал для записи, пишет в него данные и закрывает канал; если с клавиатуры

прочитано менее 2-х байт, то, после передачи их в канал, завершает свою ра- боту.

## Дублирование дескрипторов файлов

Известно, что:

* *интерпретатор shell*, запуская процесс, передает ему три дескриптора фай- лов: 0 — для чтения; 1 и 2 — для записи;
* *функция open(...)*, в случае нормального завершения, возвращает наименьший доступный номер дескриптора;

В практических задачах часто возникает необходимость дублирования деск- рипторов или открытие дескриптора с заданным номером. Например, серверные *программы*, принимая внешние к ним запросы, могут сами не иметь алгоритмов обработки файлов, а обращаются к другим программам. В этом случае:

* *программа обработки алгоритма* пищется и отлаживается при условии, что ввод осущесвляется из дескриптора с номером 0, а вывод осуществляется в

файл с дескриптором 1;

* *серверная программа*, для обработки запроса, создает дочерний процесс, ко- торый: открывает файл ввода информации, закрывает дескриптор 0 и дубли-

рует дескриптор открытого файла, обеспечивая ввод данных по дескриптору 0; аналогичные действия проводятся с дескриптором файла вывода; после этого, посредством одного из системных вызовов *exec(…)*, осуществляется запуск *программы обработки алгоритма*.

Стандарт POSIX предоставляет два системных вызова дублирования деск- рипторов файлов:

#include <unistd.h>

int dup (int *oldfd*);

int dup2(int *oldfd*, int *newfd*);

где oldfd — старый дескриптор файла; newfd — новый дескриптор файла.

Обе функции, в случае нормального завершения вызова:

* возвращают новый дескриптор файла;
* функция *dup2(…)*, если требуется, предварительно закрывает старый деск- риптор *oldfd*.

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

Если произошла ошибка, то возвращается значение -1 и переменная *errno*

устанавливается должным образом..

## Монтирование и демонтирование ФС

Все ОС имеют системные вызовы, предназначенные для *подключения* и *отключения* внешних файловых систем к *корневой файловой системе*. Стандарт POSIX определяет эти функции как:

#include <sys/mount.h>

int mount(const char \*source, const char \*target, unsigned mountflags);

int umount(const char \*special);

где *source* — раздел блочного устройства или файл, содержащий файловую систе- му, известную ядру ОС;

*target* — директория, присутствующая в файловой системе, к которой монтируется внешняя файловая система;

*mountflags* — опции, с которыми монтируется файловая система;

*special* — это *source* или *target*, которые специфицируют точку монтирования.

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

**Замечание** В предыдущих темах были подробно изучены утилиты ***mount*** и ***umount***, предназначен- ные для аналогичных целей.

Программисту, *без большой необходимости*, не следует не следует использовать эти системные вызовы, потому что:

* *основные привилегии* создания структуры ФС и защита такой структуры обеспе- чивается администратором ОС, что исключает необходимость оперативного мон-

тирования и демонтирования из программы;

* *синтаксис и реализация* этих системных вызовов всегда специфичен для конк- ретной ОС, можно убедится при изучении ***man 2 mount*** и ***man 2 umount***;
* *в крайнем случае*, можно с помощью дочернего процесса вызвать «родные» ути- литы ОС, что повысит переносимость разработанного ПО.

## Ссылки на имена файлов

Программисту и обычному пользователю, любая отдельная файловая система ОС представляется в виде иерархии имен файлов, в которой особое место зани- мают *файлы типа директорий* (*каталогов*):

* именно *с корневого каталога* начинается каждая файловая система, разме- щенная на внешних носителях информации;
* именно *к каталогам монтируются* разные файловые системы, образуя еди- ное дерево файловой системы ОС, вершина которой (*корневой каталог VFS*)

формируется в ядре ОС;

* именно *в записях каталогов* имя файла (*name*) связано с i-узлом (*inode*).

В стандарте POSIX для создания и удаления каталогов определяет два сис-

темных вызова: *mkdir(...)* и *rmdir(...)*.

Системный вызов *mkdir(...)* имеет синтаксис:

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

int mkdir(const char \*pathname, mode\_t mode);

где *pathname* — имя создаваемой каталога;

*mode* — задает права доступа, которые получит зданный каталог; эти права стан- дартным образом модифицируются с помощью *umask*: права доступа оказываются равны *mode & ~umask*.

Созданный каталог принадлежит фактическому владельцу процесса и насле- дует его права. При успешном завершении возвращается ноль или -1, если про- изошла ошибка. В этом случае *errno* устанавливается следующим образом:

**Замечание** Создать файл каталог можно и спомощью системного вызова *open(...)* или, напри-мер, *mknod(...)*,но такой каталог будет «*неполноценным*», поскольку в нем будут от- сутствовать файлы «.» и «..».

Системный вызов *rmdir(...)* имеет синтаксис:

#include <unistd.h>

int rmdir(const char \*pathname);

где *pathname* — имя удаляемого каталога, который должен быть пустым, кроме файлы «.» и «..».

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

устанавливается следующим образом:

**Для удаления имен файлов**, в стандарте POSIX, предусмотрен системный вызов *unlink(...)*, имеющий следующий синтаксис:

#include <unistd.h>

int unlink(const char \*pathname);

где *pathname* — имя удаляемого файла.

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

устанавливается следующим образом:

**Замечание**

Удаляя имя из файловой системы, *unlink(...)* учитывает, что:

* если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он зани- мает освобождается для дальнейшего использования;
* если имя было посленей ссылкой на файл, но какие-либо процессы все еще дер- жат этот файл открым, файл будет оставлен пока последний файловый дескрип- тор, указывающий на него, не будет закрыт;
* если имя указывает на символьную ссылку, ссылка будет удалена;
* если имя указывает на сокет, FIFO или устройство, имя будет удалено, но про- цессы, которые открыли любой из этих объектов могут продолжать его использо-

вать.

**Таким образом**, мы видим, что в общем случае, *один файл может иметь несколько имен*.

Для создания новых имен, которые интерпретируются как имена файлов, стандарт POSIX предоставляет два системных вызова:

* *link()* - создает *новую ссылку* на существующий файл (*на i-узел*) внутри одной файловой системы, известную также как "*жесткая*" ссылка; она имеет теже

свойства, что и старое имя, поэтому их оригинальность их установить невоз- можно;

* *symlink()* - создает *символьную ссылку на имя* предположительно существую- щего файла, независимо от файловой системы, в которой оно должно нахо-

диться; если такого файла не существует, то ссылка называется «*висячей*». Системный вызов *link(...)* имеет следующий синтаксис:

#include <unistd.h>

int link(const char \*oldpath, const char \*newpath);

где *oldpath* — имя существующего файла;

*newpath* — новое имя файла, в тойже файловой системе, что и старое.

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

устанавливается следующим образом:

Системный вызов *symlink(...)* имеет следующий синтаксис:

#include <unistd.h>

int symlink(const char \*oldpath, const char \*newpath);

где *oldpath* — имя существующего файла;

*newpath* — новое имя файла, в тойже файловой системе, что и старое.

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

устанавливается следующим образом:

**Замечание**

При использовании *symlink(...)*:

* *не производится* никакой проверки *oldpath;*
* *удаление файла*, на который ссылается символьная ссылка, действительно уда- лит файл, если только у него нет других жестких ссылок; если такое поведение

нежелательно, то следует использовать *link(...)*.

**Замечание**

На практике, часто используются и другие функции для работы с файловой системой ОС, которые можно отнести к системным вызовам. Например, *chdir(...)*, *chroot(...)*, *rename(…)* и другие.