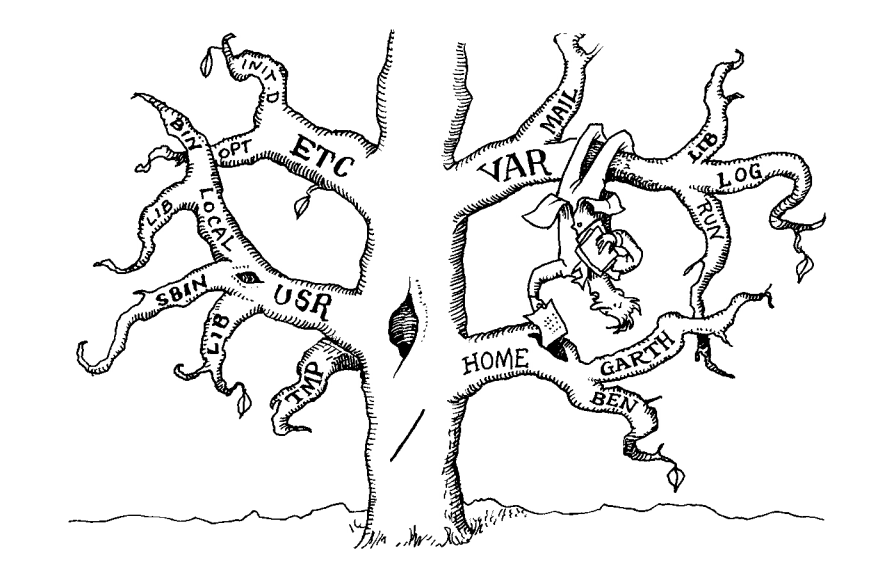
***5 Файловая система***



Быстрый опрос: что из нижеперечисленного вы ожидаете найти в "файловой системе"?

* Процессы
* Аудио устройства
* Структуры данных ядра и параметры настройки
* Каналы связи между процессами

Если система UNIX или Linux, правильный ответ "все вышеперечисленное и даже больше!". И да, там вы также можете найти некоторые файлы1.

*1. Возможно, точнее будет сказать, что эти объекты представлены в файловой системе. В большинстве случаев файловая система используется в качестве точки рандеву для соединения клиентов с драйверами, которые они ищут.*

Основная цель файловой системы - представлять и организовывать ресурсы системы хранения. Тем не менее, программисты стремились избегать изобретения велосипеда, когда дело доходило до управления другими типами объектов. Часто оказывается удобным отобразить эти объекты в пространстве имен файловой системы. Эта унификация имеет некоторые преимущества (согласованный программный интерфейс, легкий доступ из оболочки) и некоторые недостатки (реализации файловой системы, похожие на чудовище Франкенштейна), но, нравится нам это или нет, это путь UNIX (и, следовательно, Linux).

Файловую систему можно представить четырьмя основными компонентами:

* Пространство имен - способ назвать вещи и организовать их в иерархии
* API2 - набор системных вызовов для навигации и управления объектами
* Модели безопасности - схемы защиты, сокрытия и обмена некоторыми объектами
* Реализация - программное обеспечение, чтобы связать логическую модель с оборудованием

*2. API или интерфейс программирования приложений - это общий термин для набора процедур, которые могут быть вызваны программистом благодаря библиотекам, операционной системе или программным пакетам.*

Современные ядра определяют абстрактный интерфейс, который поддерживает множество различных back-end-файловых систем. Некоторые части файлового дерева обрабатываются традиционными реализациями на основе дисков. Другие - отдельными драйверами в ядре. Например, сетевые файловые системы обрабатываются драйвером, который перенаправляет запрошенные операции на удаленный сервер в сети.

К сожалению, архитектурные границы не четко очерчены, и существует довольно много особых случаев. Например, "файлы устройств" определяют способ взаимодействия программ с драйверами внутри ядра. Файлы устройств на самом деле не являются файлами данных, но они обрабатываются файловой системой, а их характеристики хранятся на диске.

Еще одним усложняющим фактором является то, что ядро поддерживает более одного типа дисковой файловой системы. Преобладающими стандартами являются файловые системы ext4, XFS и UFS, а также ZFS от Oracle и Btrfs. Тем не менее, доступно много других, в том числе Verita VxFS и JFS от IBM.

Также широко поддерживаются "чужие" файловые системы, включая файловые системы FAT и NTFS, используемые в Microsoft Windows, и файловую систему ISO 9660, используемую на старых компакт-дисках.

Файловая система - это большая тема, которую мы рассматриваем с разных сторон. В этой главе рассказывается, где искать элементы в вашей системе, и описываются характеристики файлов, значения битов прав доступа и использование некоторых основных команд, которые просматривают и устанавливают атрибуты. В Главе 20 "Хранилище" вы найдете более технически сложные темы о файловой системе, такие как разбиение диска.

Глава 21, Сетевая файловая система, описывает NFS, систему общего доступа к файлам, которая обычно используется для удаленного доступа к файлам между системами UNIX и Linux. Глава 22, SMB, описывает аналогичную систему из мира Windows.

*См. Разделы, начиная со стр. 762, для получения дополнительной информации о конкретных файловых системах.*

Имея так много различных реализаций файловых систем, может показаться странным, что эта глава называется так, как если бы существовала только одна файловая система. Мы можем быть неопределенными в отношении исходного кода, потому что большинство современных файловых систем либо пытаются реализовать традиционные функциональные возможности файловой системы более быстрым и более надежным способом, либо добавляют дополнительные функции в качестве слоя поверх стандартной семантики файловой системы. Некоторые файловые системы реализуют оба подхода. Хорошо это или плохо, но слишком большое количество существующего программного обеспечения зависит от модели, описанной в этой главе, поэтому эту модель невозможно просто отбросить.

**5.1 Имя пути (pathnames)**

Файловая система представлена в виде единой унифицированной иерархии, которая начинается с каталога **/** и продолжается через произвольное количество подкаталогов. **/** также называется корневым каталогом. Эта система с одной иерархией отличается от используемой в Windows, в которой сохраняется концепция пространств имен для конкретных разделов.

Графические пользовательские интерфейсы часто называют каталоги "папками", даже в системах Linux. Папки и каталоги - это одно и то же; "папка" - это всего лишь языковая утечка из миров Windows и macOS. Тем не менее, стоит отметить, что слово "папка" имеет тенденцию вызывать раздражение у некоторых технарей. Не используйте его в техническом контексте, если вы не готовы к усмешкам в вашу сторону.

Список каталогов, которые необходимо просмотреть, чтобы найти конкретный файл, а также имя файла этого файла образуют путь. Имена путей могут быть либо абсолютными (например, **/tmp/foo**), либо относительными (например, **book4/filesystem**). Относительные имена путей интерпретируются, начиная с текущего каталога. Возможно, вы привыкли думать о текущем каталоге как о функции оболочки, но она есть у каждого процесса.

Термины *filename*, *pathname* и *path* более или менее взаимозаменяемы или, по крайней мере, мы используем их взаимозаменяемо в этой книге. *Filename* и *path* могут использоваться как для абсолютных, так и для относительных путей; *pathname* обычно предполагает абсолютный путь.

Файловая система может быть сколь угодно глубокой. Однако каждый компонент имени пути (т.е. каждый каталог) должен иметь имя длиной не более 255 символов. Существует также ограничение на общую длину пути, которую вы можете передать в ядро в качестве аргумента системного вызова (4095 байт в Linux, 1024 байт в BSD). Чтобы получить доступ к файлу с более длинным именем пути, чем эти, вы должны перейти в промежуточный каталог и использовать относительное имя пути.

**5.2 Монтирование и демонтирование файловой системы**

Файловая система состоит из небольших блоков, также называемых файловыми системами, каждая из которых состоит из одной директории и ее поддиректорий и файлов. Из контекста обычно очевидно, какой тип "файловой системы" обсуждается, но для ясности в следующем обсуждении мы используем термин "файловое дерево" для обозначения общей структуры и резервируем слово "файловая система" для отдельных ветвей файлового дерева.

Некоторые файловые системы живут на дисковых разделах или на логических томах, поддерживаемых физическими дисками, но, как упоминалось ранее, файловые системы могут представлять собой все, что подчиняется соответствующему API: сетевой файловый сервер, компонент ядра, эмулятор диска на основе памяти и т.д. Большинство ядер имеют изящную "петлевую" файловую систему, которая позволяет монтировать отдельные файлы, как если бы они были отдельными устройствами. Это полезно для монтирования образов DVD-ROM, хранящихся на диске, или для разработки образов файловой системы без необходимости ее переразбиения. Системы Linux могут даже обрабатывать существующие части файлового дерева как файловые системы. Этот трюк позволяет дублировать, перемещать или скрывать части файлового файлов.

В большинстве случаев файловые системы присоединяются к дереву с помощью команды **mount**3. **mount** подключает каталог в существующем файловом дереве, называемом точкой монтирования, к корню вновь присоединенной файловой системы. Предыдущее содержимое точки монтирования становится временно недоступным, пока там смонтирована другая файловая система. Однако точки монтирования обычно являются пустыми каталогами.

*3. Мы говорим "в большинстве случаев", потому что ZFS использует несколько иной подход к монтированию и демонтированию, не говоря уже о многих других аспектах администрирования файловых систем. Смотрите стр. 773 для подробной информации.*

Например:

**$ sudo mount /dev/sda4 /users**

устанавливает файловую систему, хранящуюся на разделе диска, представленном **/dev/sda4** по пути **/users**. Затем вы можете использовать **ls /users** для просмотра содержимого этой файловой системы.

В некоторых системах **mount** - это просто обертка, которая вызывает специфичные для файловой системы команды, такие как **mount.ntfs** или **mount\_smbfs**. Вы можете вызывать эти вспомогательные команды напрямую, если вам нужно; иногда они предлагают дополнительные опции, которые не понимает обёртка **mount**. С другой стороны, универсальной команды **mount** достаточно для решения повседневных задач.

Вы можете запустить команду **mount** без каких-либо аргументов, чтобы увидеть все файловые системы, которые смонтированы в настоящий момент. В системах Linux их может быть 30 или более, большинство из которых представляют различные интерфейсы для ядра.

Файл **/etc/fstab** содержит список файловых систем, которые обычно монтируются в системе. Информация в этом файле позволяет автоматически проверять файловые системы (с помощью **fsck**) и монтировать их (с помощью **mount**) во время загрузки с указанными вами параметрами. Файл **fstab** также служит документацией для размещения файловых систем на диске и включает короткие команды, такие как **mount /usr**. Смотрите страницу 768 для обсуждения **fstab**.

Вы можете отключить файловые системы с помощью команды **umount**. **umount** пожалуется, если вы попытаетесь демонтировать файловую систему, которая используется в данный момент. Отсоединяемая файловая система не должна иметь открытых файлов или процессов, чьи текущие каталоги находятся в ней; если файловая система содержит исполняемые программы, ни одна из них не должна быть запущена.

В Linux есть "ленивая" опция демонтирования (**umount -l**), которая удаляет файловую систему, но не отключает ее по-настоящему, пока все существующие ссылки на файлы не будут закрыты. Это спорный вопрос, является ли эта опция полезной. Начнем с того, что нет никакой гарантии, что существующие ссылки когда-либо закроются сами по себе. Кроме того, состояние "полуотключенный" может представлять несогласованную семантику файловой системы программам, которые ее используют; они могут читать и записывать с помощью существующих файловых дескрипторов, но не могут открывать новые файлы или выполнять другие операции с файловой системой.

**umount -f** принудительно демонтирует занятую файловую систему и поддерживается во всех наших примерах систем. Тем не менее, использование этой опции на монтируемых не NFS файловых системах почти всегда является плохой идеей. Более того, она может не работать на определенных типах файловых систем (например тех, которые хранят журналы, таких как XFS или ext4).

Вместо использования **umount -f**, когда файловая система, которую вы пытаетесь демонтировать, оказывается занятой, выполните команду **fuser**, чтобы выяснить, какие процессы содержат ссылки на эту файловую систему. **fuser -c** *точка\_монтирования* выводит на экран PID каждого процесса, который использует файл или каталог в этой файловой системе, а также серию буквенных кодов, которые показывают активность.

Например:

|  |
| --- |
| freebsd$ **fuser -c /usr/home**  /usr/home: 15897c 87787c 67124x 11201x 11199x 11198x 972x |

Точные буквенные коды варьируются от системы к системе. В этом примере из системы FreeBSD **c** указывает, что процесс имеет свой текущий рабочий каталог в файловой системе, а **x** указывает на выполняемую программу. Тем не менее, детали, как правило, не важны - PID - это то, что вам нужно.

Чтобы исследовать проблемные процессы, просто запустите **ps** со списком PID, возвращаемых **fuser**. Например:

nutrient:~$ **ps up "87787 11201"**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| USER | PID | %CPU | %MEM | STARTED | TIME | COMMAND |
| fnd  fnd | 11201  87787 | 0.0  0.0 | 0.2  0.0 | 14Jul16  Thu07PM | 2:32.49  0:00.93 | ruby: slave\_audiochannelbackend  -bash (bash) |

Здесь кавычки заставляют оболочку передавать список PID в **ps** в качестве единственного аргумента.

В системах Linux вы можете избежать необходимости фильтровать PID через **ps**, запустив **fuser** с флагом **-v**. Эта опция создает более читаемый вывод, который включает имя команды.

$ fuser -cv /usr

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | USER | PID | ACCESS | COMMAND |
| /usr | root  root  root | 444  499  520 | ....m  ....m  ....m | atd  sshd  lpd |
| ... |  |  |  |  |

Буквенные коды в столбце ACCESS те же, что используются в неподробном выводе команды **fuser**.

Более сложной альтернативой **fuser** является утилита **lsof**. **lsof** - более сложная и изощренная программа, чем **fuser**, и ее вывод, соответственно, более подробный. По умолчанию **lsof** установлен во всех наших примерах систем Linux и доступен в виде пакета во FreeBSD.

В Linux скрипты в поисках конкретной информации об использовании файловых систем процессами также могут напрямую читать файлы в **/proc**. Однако **lsof -F**, который форматирует вывод **lsof** для простого анализа, является более простым и мобильным решением. Используйте дополнительные флаги командной строки для запроса только необходимой информации.

**5.3 Организация файлового дерева**

Системы UNIX никогда не были хорошо организованы. Одновременно используются различные несовместимые соглашения об именах, и различные типы файлов случайным образом разбросаны по пространству имен. Во многих случаях файлы делятся по функциям, а не по вероятности их изменения, что затрудняет обновление операционной системы. Каталог **/etc**, например, содержит некоторые файлы, которые никогда не настраиваются, а некоторые - полностью локальные. Как узнать, какие файлы следует сохранить при обновлении? Ну, вам просто нужно знать ... или доверять установочному программному обеспечению для принятия правильных решений.

Как у логически мыслящего системного администратора, у вас может возникнуть желание улучшить текущую организацию. К сожалению, в файловом дереве много скрытых зависимостей, поэтому такие усилия обычно приводят к возникновению проблем. Просто позвольте всему оставаться там, куда его устанавливают ОС и системные пакеты. Когда предлагается выбор мест установки, всегда выбирайте дефолтное значение, если у вас нет особых и веских причин поступить иначе.

*См. Главу 11 для получения дополнительной информации о настройке ядра.*

Корневая файловая система включает в себя как минимум корневой каталог и минимальный набор файлов и подкаталогов. Файл, содержащий ядро ОС, обычно находится в **/boot**, но его точное имя и местоположение могут отличаться. В BSD и некоторых других системах UNIX ядро на самом деле представляет собой не один файл, а набор компонентов.

Также частью корневой файловой системы являются **/etc** - для критических системных и конфигурационных файлов, **/sbin** и **/bin** - для важных утилит, и иногда **/tmp** - для временных файлов. Каталог **/dev** традиционно был частью корневой файловой системы, но в наши дни это виртуальная файловая система, которая монтируется отдельно (см. стр. 331 для получения дополнительной информации по этой теме).

Некоторые системы хранят файлы совместно используемых библиотек и некоторые другие дополнения, такие как препроцессор C, в каталоге **/lib** или **/lib64**. Другие переместили эти элементы в /usr/lib, иногда оставляя их в **/lib** в качестве символической ссылки.

Каталоги **/usr** и **/var** также имеют большое значение. /usr - это место, где хранится большинство стандартных, но не критичных для системы программ, а также различное добро в виде онлайн руководств и большинства библиотек. FreeBSD хранит немного локальной конфигурации в **/usr/local**. **/var** содержит спул каталоги, файлы журналов, учетную информацию и различные другие элементы, которые быстро растут или изменяются и различаются на каждом хосте. Чтобы система могла перейти в многопользовательский режим - должны быть доступны и **/usr**, и **/var**.

*Чтобы узнать несколько причин, по которым разделение может быть желательным, и некоторые практические правила для его руководства см. стр. 742.*

В прошлом было стандартной практикой разделить системный диск и поместить некоторые части файлового дерева в свои собственные разделы, чаще всего в **/usr**, **/var** и **/tmp**. Это не редкость даже сейчас, но существует более долговременная тенденция - иметь одну большую корневую файловую систему. Большие жесткие диски и все более изощренные реализации файловой системы снизили ценность создания разделов.

Разбиение чаще всего используется в попытке предотвратить использование одной частью файлового дерева всего доступного пространства и не допустить остановку всей системы. Соответственно, **/var** (который содержит файлы журналов, которые могут разрастаться в трудные времена), **/tmp** и домашние каталоги пользователей являются одними из первых кандидатов на создание и использование собственных разделов. Выделенные файловые системы также могут хранить громоздкие элементы, такие как библиотеки исходного кода и базы данных.

В таблице 5.1 перечислены некоторые из наиболее важных стандартных каталогов (альтернативные строки были скрыты для улучшения читаемости).

В большинстве систем в man странице **hier** более подробно описаны некоторые общие принципы компоновки файловой системы. Однако не ожидайте, что действующая система будет соответствовать генеральному плану во всех отношениях.

Для систем Linux Стандарт Иерархии Файловых Систем пытается систематизировать, рационализировать и объяснить стандартные каталоги4. Это отличный ресурс для ознакомления, когда вы сталкиваетесь с необычной ситуацией и вам необходимо выяснить, куда что-то поместить. Несмотря на свой статус "стандарта", он является больше отражением реальной мировой практики, чем предписывающим документом. Кроме того, в последнее время он не подвергался особому обновлению, поэтому он не описывает точную структуру файловой системы, находящуюся в современных дистрибутивах.

*4. См. wiki.linuxfoundation.org/en/FHS*

**5.4 Типы файлов**

Большинство реализаций файловой системы определяют семь типов файлов. Даже когда разработчики добавляют что-то новое и замечательное в файловое дерево (например, информацию о процессах в **/proc**), оно все равно должно быть сделано и выглядеть так, как если бы оно было одним из следующих семи типов:

* Обычные файлы
* Каталоги
* Символьные файлы устройств
* Файлы блочных устройств
* Сокеты локального домена
* Именованные конвейеры (FIFO)
* Символьные ссылки

Вы можете определить тип существующего файла с помощью команды **file**. **file** не только знает о стандартных типах файлов, но также знает кое-что о распространенных форматах, используемых в обычных файлах.

$ **file /usr/include**

|  |
| --- |
| /usr/include: directory |

$ **file /bin/sh**

|  |
| --- |
| /bin/sh: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),  dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 11.0  (1100122), FreeBSD-style, stripped |

Вся эта информация о **/bin/sh** означает что "это исполняемая команда".

**Таблица 5.1 - Стандартные каталоги и их содержимое**

|  |  |
| --- | --- |
| **Имя пути** | **Содержимое** |
| **/bin** | Основные команды операционной системы |
| **/boot** | Загрузчик, ядро и файлы, необходимые ядру |
| **/compat** | Во FreeBSD, файлы и библиотеки для бинарной совместимости с Linux |
| **/dev** | Записи устройств для дисков, принтеров, псевдо-терминалов и т.д. |
| **/etc** | Критические файлы запуска и конфигурации |
| **/home** | Стандартные домашние каталоги пользователей |
| **/lib** | Библиотеки, общие библиотеки и команды, используемые в **/bin** и **/sbin** |
| **/media** | Точки монтирования для файловых систем съемных носителей |
| **/mnt** | Временные точки монтирования съемных носителей |
| **/opt** | Дополнительные пакеты ПО (используется редко, для совместимости) |
| **/proc** | Информация обо всех запущенных процессах |
| **/root** | Домашний каталог суперпользователя (иногда просто /) |
| **/run** | Место встречи для работающих программ (PID, сокеты и т.д.) |
| **/sbin** | Основные команды операционной системыа |
| **/srv** | Файлы, предназначенные для распространения через веб или другие серверы |
| **/sys** | Множество различных интерфейсов ядра (Linux) |
| **/tmp** | Временные файлы, которые могут исчезнуть после перезагрузки |
| **/usr** | Иерархия вторичных файлов и команд |
| **/usr/bin** | Большинство команд и исполняемых файлов |
| **/usr/include** | Файлы заголовков для компиляции С программ |
| **/usr/lib** | Библиотеки; также поддерживает файлы для стандартных программ |
| **/usr/local** | Локальное ПО или конфигурационные данные; зеркалирует **/usr** |
| **/usr/sbin** | Менее важные команды для администрирования и ремонта |
| **/usr/share** | Элементы, которые могут быть общими для множества систем |
| **/usr/share/man** | Он-лайн man страницы |
| **/usr/src** | Исходный код для нелокального ПО (используется не широко) |
| **/usr/tmp** | Еще одно временное пространство (содержимое сохраняется после перезагрузки) |
| **/var** | Системные данные и некоторые файлы конфигурации |
| **/var/adm** | Варьируется: журналы, записи настройки, странные административные биты |
| **/var/log** | Файлы системного журнала |
| **/var/run** | Та же функция, что и у **/run**; в настоящее время часто является символической ссылкой |
| **/var/spool** | Буферные (то есть хранилища) каталоги для принтеров, почты и т.д. |
| **/var/tmp** | Еще одно временное пространство (содержимое сохраняется после перезагрузки) |

1. Отличительной чертой **/sbin** изначально было то, что его содержимое было статически связано и поэтому имело меньше зависимостей от других частей системы. В настоящее время все бинарные файлы динамически связаны, и между **/bin** и **/sbin** нет реальной разницы.

Другой вариант для исследования файлов **ls -ld**. Флаг **-l** показывает подробную информацию, а флаг **-d** заставляет **ls** показывать информацию для каталога, а не показывать его содержимое.

Первый символ вывода **ls** показывает тип. Например, первый символ d в следующем выводе показывает, что **/usr/include** является каталогом:

$ **ls -ld /usr/include**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| drwxr-xr-x | 27 | root | root | 4096 | Jul 15 | 20:57 | /usr/include |

В таблице 5.2 показаны коды, используемые **ls** для представления файлов различных типов.

**Таблица 5.2 - Кодировка типов файлов, используемая ls**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип файла** | **Символ** | **Как создать** | **Как удалить** |
| Обычный файл | - | редактором, **cp**, и т.д. | **rm** |
| Каталог | d | **mkdir** | **rmdir, rm -r** |
| Символьный файл устройства | c | **mknod** | **rm** |
| Файл блочного устройства | b | **mknod** | **rm** |
| Локальный доменный сокет | s | системный вызов **socket** | **rm** |
| Именованный канал | p | **mknod** | **rm** |
| Символьная ссылка | l | **ln -s** | **rm** |

Как показано в таблице 5.2, **rm** является универсальным инструментом для удаления файлов. Но как бы вы удалили файл с именем, скажем, **-f**? Это допустимое имя файла в большинстве файловых систем, но в данном случае **rm -f** не сработает, потому что **rm** интерпретирует **-f** как флаг. Ответ заключается либо в том, чтобы ссылаться на файл по пути, имя которого не начинается с тире (например, **./-f**), либо с помощью аргумента **--**, который сообщает **rm**’у, что все, что следует за ним, является именем файла, а не параметром (то есть, **rm -- -f**).

Имена файлов, содержащие управляющие символы или символы Unicode, представляют аналогичную проблему, поскольку воспроизведение этих имен с клавиатуры может быть затруднено или невозможно. В этой ситуации вы можете использовать шаблоны shell, чтобы определить файлы, которые нужно удалить. Когда вы используете шаблоны, рекомендуется использовать опцию **-i** команды **rm**, чтобы **rm** подтвердил удаление каждого файла. Эта функция защищает вас от удаления любых "хороших" файлов, которые случайно совпадают с вашим шаблоном. Чтобы удалить файл с именем **foo**<Control-D>**bar** в следующем примере, вы можете использовать

$ **ls**

|  |  |  |
| --- | --- | --- |
| foo?bar | foose | kde-root |

$ **rm -i foo\***

rm: remove 'foo\004bar'? **y**

rm: remove 'foose'? **n**

Обратите внимание, что **ls** показывает управляющий символ в виде знака вопроса, что может быть немного обманчиво. Если вы этого не помните - **?** является символом совпадения с шаблоном оболочки. Попробуйте выполнить команду **rm foo?bar**, потенциально вы можете удалить более одного файла (хотя и не в этом примере). **-i** ваш друг!

**ls -b** показывает контрольные символы в виде восьмеричных чисел, которые могут быть полезны, если вам нужно их конкретно идентифицировать. <Control-A> равен 1 (\001 в восьмеричном представлении), <Control-B> равен 2 и т.д. в алфавитном порядке. **man ascii** и страница в Wikipedia о ASCII содержат хорошую таблицу управляющих символов и их восьмеричные эквиваленты.

Чтобы удалить файлы с наиболее ужасными именами, вам может потребоваться использовать команду **rm -i \*.**

Другим вариантом удаления файлов с ненормальными именами является использование альтернативного интерфейса файловой системы, такого как режим dired **emacs** для работы в режиме реального времени, или визуального инструмента, такого как Nautilus.

**Обычные файлы**

Обычные файлы состоят из серии байтов; файловые системы не накладывают никакой структуры на их содержимое. Текстовые файлы, файлы данных, исполняемые программы и общие библиотеки хранятся как обычные файлы. Разрешен как последовательный, так и произвольный доступ.

**Каталоги**

Каталог содержит именованные ссылки на другие файлы. Вы можете создавать каталоги с помощью **mkdir** и удалять их с помощью **rmdir**, если они пусты. Вы можете рекурсивно удалить непустые каталоги, включая все их содержимое, с помощью **rm -r**.

Специальные элементы "**.**" и "**..**" относятся к самому каталогу и его родительскому каталогу; они не могут быть удалены. Поскольку корневой каталог не имеет реального родительского каталога, путь "**/..**" эквивалентен пути "**/.**" (и оба эквивалентны **/**).

**Жесткие ссылки**

Имя файла хранится в его родительском каталоге, а не в самом файле. Фактически, более одного каталога (или более одной записи в одном каталоге) могут ссылаться на файл одновременно, и ссылки могут иметь разные имена. Такой механизм создает иллюзию, что файл существует в более чем одном месте одновременно.

Эти дополнительные ссылки ("ссылки" или "жесткие ссылки", чтобы отличать их от символьных ссылок, обсуждаемых ниже) являются синонимами исходного файла. Что касается файловой системы, все ссылки на файл эквивалентны. Файловая система ведет подсчет количества ссылок, указывающих на каждый файл, и не освобождает блоки данных файла до тех пор, пока его последняя ссылка не будет удалена. Жесткие ссылки не могут пересекать границы файловой системы.

Вы можете создать жесткие ссылки с помощью **ln** и удалить их с помощью **rm**. Синтаксис **ln** легко запомнить, если вы помните, что он отражает синтаксис **cp**. Команда **cp oldfile newfile** создает копию **oldfile** с именем **newfile**, а **ln oldfile newfile** делает имя **newfile** дополнительной ссылкой на **oldfile**.

В большинстве реализаций файловых систем технически возможно создавать жесткие ссылки как на каталоги, так и на простые файлы. Однако ссылки на каталоги часто приводят к вырожденным условиям, таким как петли файловой системы и каталоги, которые не имеют единственного однозначного родителя. В большинстве случаев лучше использовать символическую ссылку (см. стр. 131).

Вы можете использовать **ls -l**, чтобы увидеть, сколько существует ссылок на данный файл. Смотрите пример вывода **ls** на стр. 134 для получения некоторых дополнительных деталей. Также обратите внимание на комментарии относительно **ls -i** на стр. 135, так как этот параметр особенно полезен для определения жестких ссылок.

Жесткие ссылки не являются отдельным типом файлов. Вместо того, чтобы определять отдельную "вещь", называемую жесткой ссылкой, файловая система просто позволяет нескольким элементам каталога указывать на один и тот же файл. В дополнение к содержимому файла, базовые атрибуты файла (такие как права собственности и разрешения) также являются общими.

**Файлы символьных и блочных устройств**

*См. Главу 11 для получения дополнительной информации об устройствах и драйверах.*

Файлы устройств позволяют программам взаимодействовать с аппаратным и периферийным оборудованием системы. Ядро включает (или загружает) драйвер для каждого из устройств системы. Это программное обеспечение заботится о мельчайших деталях управления каждым устройством, так что само ядро может оставаться относительно абстрактным и аппаратно независимым.

Драйверы устройств представляют стандартный интерфейс связи, который выглядит как обычный файл. Когда файловая система получает запрос, который ссылается на файл символьного или блочного устройства, она просто передает запрос соответствующему драйверу устройства. Однако важно отличать файлы устройств от драйверов устройств. Файлы - это просто точки рандеву, которые взаимодействуют с драйверами. Они сами не являются драйверами.

Различие между символьными и блочными устройствами неуловимо и не стоит подробного описания. В прошлом несколько типов аппаратного обеспечения были представлены как блочными, так и символьными файлами устройств, но сегодня такая конфигурация встречается редко. На практике FreeBSD полностью избавилась от блочных устройств, хотя их спектральное присутствие все еще можно увидеть в **man-**страницах и файлах заголовков.

Файлы устройств характеризуются двумя номерами, называемыми мажорным и минорным номерами устройств. Мажорный номер устройства сообщает ядру, к какому драйверу относится файл, а минорный номер устройства обычно сообщает драйверу, к какому физическому устройству обращаться. Например, мажорный номер устройства 4 в системе Linux обозначает последовательный драйвер. Первый последовательный порт (**/dev/tty0**) будет иметь мажорный номер устройства 4 и минорный номер устройства 0.

Драйверы могут интерпретировать минорные номера устройств, которые им передаются, любым удобным для них способом. Например, драйверы ленты используют минорный номер устройства, чтобы определить, следует ли перематывать ленту при закрытии файла устройства.

В далеком прошлом каталог **/dev** был универсальным каталогом, и файлы устройств в нем создавались с помощью **mknod** и удалялись с помощью **rm**. К сожалению, эта грубая система была плохо приспособлена для работы с бесконечным морем драйверов и типов устройств, которые появились за последние несколько десятилетий. Она также способствовала появлению разного рода потенциальных несоответствий конфигурации: файлам устройств, которые не ссылались на фактическое устройство, устройства, недоступные из-за отсутствия файлов устройств, и так далее.

В наши дни каталог **/dev** обычно монтируется как специальный тип файловой системы, и его содержимое автоматически поддерживается ядром совместно с демоном пользовательского уровня. Есть несколько разных версий этой же базовой системы. См. Главу 11, Драйверы и Ядро, для получения дополнительной информации о подходе каждой системы к этой задаче.

**Сокеты локального домена**

Сокеты - это соединения между процессами, которые позволяют им общаться напрямую. UNIX определяет несколько видов сокетов, большинство из которых включают в себя сеть.

*См. Главу 10 для получения дополнительной информации о syslog.*

Сокеты локального домена доступны только с локального хоста, и на них ссылаются через объект файловой системы, а не через сетевой порт. Иногда их называют "доменными сокетами UNIX". Syslog и X Window System являются примерами стандартных средств, которые используют сокеты локального домена, но их гораздо больше, включая многие базы данных и серверы приложений.

Сокеты локального домена создаются с помощью системного вызова **socket** и удаляются с помощью команды **rm** или системного вызова **unlink**, когда у них больше нет пользователей.

**Именованные каналы**

Как и локальные доменные сокеты, именованные каналы обеспечивают связь между двумя процессами, работающими на одном хосте. Они также известны как "файлы FIFO" (как и в финансовом учете, FIFO - это сокращение фразы "первым пришел, первым вышел"). Вы можете создавать именованные каналы с помощью **mknod** и удалять их с помощью **rm**.

Именованные каналы и локальные доменные сокеты служат похожим целям, и тот факт, что оба существуют, по сути является историческим артефактом. Скорее всего, ни один из них не существовал бы, если бы UNIX и Linux были разработаны сегодня; сетевые сокеты в результате заменят обоих.

**Символьные ссылки**

Символьная или "мягкая" ссылка указывает на файл по имени. Когда ядро сталкивается с символической ссылкой в процессе поиска имени пути, оно перенаправляет свое внимание на имя путь, сохраненное как содержимое ссылки. Разница между жесткими и мягкими ссылками заключается в том, что жесткая ссылка является прямой ссылкой, тогда как символьная ссылка является ссылкой по имени. Символьные ссылки отличаются от файлов, на которые они указывают.

Вы можете создать символьные ссылки с помощью **ln -s** и удалить их с помощью **rm**. Поскольку символьные ссылки могут содержать произвольные пути, они могут ссылаться на файлы в других файловых системах или на несуществующие файлы. Ряд символьных ссылок также может образовывать петли.

Символьная ссылка может содержать абсолютный или относительный путь. Например команда:

$ **sudo ln -s archived/secure /var/data/secure**

делает ссылку **/var/data/secure** на **/var/data/archived/secure** используя относительный путь. Она создает символьную ссылку **/var/data/secure** на **archived/secure**, что демонстрируется этим выводом из **ls**:

$ **ls -l /var/data/secure**

lrwxrwxrwx 1 root root 18 Aug 3 12:54 /var/data/secure -> archived/secure

Затем весь каталог **/var/data** можно переместить в другое место, не прерывая работу символьной ссылки.

Разрешения файла, которые **ls** показывает для символической ссылки, lrwxrwxrwx, являются фиктивными значениями. Разрешение на создание, удаление или переход по ссылке контролируется содержащим каталогом, тогда как разрешение на чтение, запись и выполнение для цели ссылки предоставляется собственными разрешениями цели. Следовательно, символьные ссылки не нуждаются (и не имеют) никакой собственной информации о разрешениях.

Распространенная ошибка - думать, что первый аргумент **ln -s** интерпретируется относительно текущего рабочего каталога. Однако этот аргумент фактически не разрешается **ln** как имя файла: это просто буквенная строка, которая становится целью символьной ссылки.

**5.5 Атрибуты файлов**

Согласно традиционной модели файловой системы UNIX и Linux каждый файл имеет набор из девяти битов разрешений, которые контролируют, кто может читать, записывать и выполнять содержимое файла. Вместе с тремя другими битами, которые в первую очередь влияют на работу исполняемых программ, эти биты составляют "режим работы" файла.

Двенадцать битов режима сохраняются вместе с четырьмя битами информации о типе файла. Четыре бита типа файла устанавливаются при первом создании файла и не могут быть изменены, но владелец файла и суперпользователь могут изменить двенадцать битов режима с помощью команды **chmod** (change mode). Используйте **ls -l** (или **ls -ld** для каталога), чтобы проверить значения этих битов. Смотрите страницу 134 для примера.

**Биты разрешений**

Девять битов разрешений определяют, какие операции могут выполняться над файлом и кем. Традиционная UNIX не позволяет устанавливать разрешения для отдельных пользователей (хотя все системы теперь поддерживают списки контроля доступа того или иного типа; см. стр. 140). Вместо этого три набора разрешений определяют доступ для владельца файла, группы владельцев файла и всех остальных (именно в этом порядке)5. Каждый набор имеет три бита: бит чтения, бит записи и бит выполнения (также именно в таком порядке).

*5. Если вы думаете о владельце как о "пользователе", а обо всех остальных как о "других", вы можете запомнить порядок наборов разрешений, на примере имени Hugo. u, g и o также являются буквенными кодами, используемыми мнемонической версией* ***chmod****.*

Удобно рассматривать права доступа к файлам в виде восьмеричных (основание 8) чисел, поскольку каждая цифра восьмеричного числа представляет три бита, а каждая группа битов разрешения состоит из трех битов. Три старших бита (с восьмеричными значениями 400, 200 и 100) контролируют доступ для владельца. Вторые три (40, 20 и 10) контролируют доступ для группы. Последние три (4, 2 и 1) контролируют доступ для всех остальных ("остального мира"). В каждом триплете старший бит - это бит чтения, средний бит - бит записи, а младший бит - бит выполнения.

Хотя пользователь может вписаться в две из трех категорий разрешений, применяются только самые конкретные разрешения. Например, владелец файла всегда имеет доступ, определяемый битами разрешения владельца, а не битами разрешения группы. Категории "другие" и "группа владельцев" могут иметь больший доступ, чем владелец, и такая конфигурация будет весьма необычной.

В обычном файле бит чтения позволяет открывать и читать файл. Бит записи позволяет изменять или удалять содержимое файла; однако возможность удаления или переименования (или удаления, а затем повторного создания!) файла контролируется разрешениями в его родительском каталоге, где фактически хранится сопоставление имени с пространством данных.

Бит выполнения позволяет файлу быть исполняемым. Существуют два типа исполняемых файлов: бинарные файлы, которые непосредственно запускает процессор, и скрипты, которые должны интерпретироваться оболочкой или какой-либо другой программой. По соглашению скрипты начинаются со строки, аналогичной следующей:

#!/usr/bin/perl

Эта строка указывает на соответствующего интерпретатора. Предполагается, что небинарные исполняемые файлы, в которых не указан интерпретатор, являются **sh** скриптами6.

*6. Ядро понимает #! ("shebang") синтаксис и действует на него напрямую. Однако, если интерпретатор не указан полностью и правильно, ядро откажется выполнять файл. Затем оболочка делает вторую попытку выполнить скрипт, вызывая* ***/bin/sh****, который обычно является ссылкой на оболочку Almquist или* ***bash****; см. стр. 198. Свен Масчек ведет чрезвычайно детальную страницу об истории, реализации и кроссплатформенном поведении шебанга по адресу goo.gl/J7izhL.*

Для каталога бит выполнения (часто называемый в этом контексте битом "поиск" или "сканирование") позволяет вводить каталог или передавать его по мере вычисления имени пути, но не выводить его содержимое. Комбинация битов чтения и выполнения позволяет просматривать содержимое каталога. Комбинация битов записи и выполнения позволяет создавать, удалять и переименовывать файлы в каталоге.

Различные расширения, такие как списки контроля доступа (см. стр. 140), SELinux (см. стр. 85) и "бонусные" биты разрешений, определенные отдельными файловыми системами (см. Стр. 139), усложняют или отменяют традиционную 9-битную модель разрешений. Если у вас возникли проблемы с объяснением наблюдаемого поведения системы, проверьте, не мешает ли один из этих факторов.

**Биты setuid и setgid**

Биты с восьмеричными значениями 4000 и 2000 являются битами setuid и setgid. При установке в исполняемых файлах эти биты позволяют программам получать доступ к файлам и процессам, которые в противном случае были бы недоступны для пользователя, который их запускает. Механизм setuid / setgid для исполняемых файлов описан на странице 68.

Когда он установлен на каталоге, бит setgid заставляет вновь созданные файлы в каталоге принимать группу, которой принадлежит каталог, а не группу пользователя, создавшего файл. Это соглашение облегчает совместное использование каталога с файлами несколькими пользователями, до тех пор пока они принадлежат к одной группе. Эта интерпретация бита setgid не связана с его значением, когда он установлен на исполняемом файле, но не может быть никакой двусмысленности относительно того, какое значение является подходящим.

**Sticky bit**

Бит с восьмеричным значением 1000 называется sticky (липким) битом. Он имел историческое значение как модификатор для исполняемых файлов в ранних системах UNIX. Однако его значение в настоящее время устарело, и современные системы молча игнорируют sticky бит, когда он установлен на обычных файлах.

Если для каталога установлен sticky бит, файловая система не позволит вам удалить или переименовать файл, если вы не являетесь владельцем каталога, владельцем файла или суперпользователем. Наличие разрешения на запись в каталоге недостаточно. Это соглашение помогает сделать каталоги вроде **/tmp** немного более приватными и безопасными.

**ls: список и проверка файлов**

Файловая система содержит около сорока отдельных частей информации для каждого файла, но большинство из них полезны только для самой файловой системы. Как системного администратора, вас больше всего заинтересует: количество ссылок, владелец, группа, режим, размер, время последнего доступа, время последней модификации и тип. Вы можете проверить все это с помощью **ls -l** (или **ls -ld** для каталога; без флага **-d ls** перечисляет содержимое каталога).

Атрибут - время изменения - также поддерживается для каждого файла. Общепринятое название этого времени ("ctime", сокращение от "время изменения") заставляет некоторых людей верить, что это время создания файла. К сожалению, это не так; он просто записывает время последнего изменения атрибутов файла (владельца, режима и т. д.), в отличие от времени, когда содержимое файла было изменено.

Рассмотрим следующий пример:

$ **ls -l /usr/bin/gzip**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -rwxr-xr-x | 4 | root | wheel | 37432 | Nov 11 2016 | | /usr/bin/gzip |

Первое поле указывает тип и режим файла. Первый символ - тире, поэтому файл является обычным файлом (см. Таблицу 5.2 для ознакомления с другими кодами).

Следующие девять символов в этом поле представляют собой три набора битов разрешения. Порядок следующий владелец-группа-другие, а порядок битов в каждом наборе - чтение-запись-выполнение. Хотя эти биты имеют только двоичные значения, **ls** показывает их символически буквами r, w и x - чтение, запись и выполнение. В этом случае владелец имеет все разрешения для файла, а все остальные имеют права на чтение и выполнение.

Если бы бит setuid был установлен, x, представляющий разрешение на выполнение для владельца, был бы заменен на s, а если бы бит setgid был установлен, x для группы также был бы заменен на s. Последний символ разрешений (разрешение на выполнение для "всех остальных") отображается как t, если включен sticky бит для файла. Если установлен бит setuid/setgid или sticky бит, а соответствующий бит выполнения - нет, эти биты отображаются как S или T.

Следующее поле в списке - это количество ссылок на файл. В данном случае это 4, что указывает на то, что **/usr/bin/gzip** - это только одно из четырех имен этого файла (остальные в этой системе - **gunzip**, **gzcat** и **zcat**, все находятся в **/usr/bin**). Каждый раз, когда создается жесткая ссылка на файл, счетчик ссылок файла увеличивается на 1. Символьные ссылки не влияют на счетчик ссылок.

Все каталоги имеют как минимум две жесткие ссылки: ссылку из родительского каталога и ссылку из специального файла с именем "." внутри самого каталога.

Следующие два поля в выводе **ls** являются поле владельца и группа владельцев файла. В этом примере владельцем файла является root, а файл принадлежит группе с именем wheel. Файловая система на самом деле хранит их как идентификаторы пользователя и группы, а не как имена. Если текстовые версии (имена) не могут быть определены, **ls** показывает эти поля в виде чисел. Это может произойти, если пользователь или группа, которой принадлежит файл, был удален из файла **/etc/passwd** или **/etc/group**. Это также может указывать на проблему с вашей базой данных LDAP (если вы ее используете); см. Главу 17.

Следующее поле - это размер файла в байтах. Этот файл имеет длину 37,432 байта. Далее следует дата последнего изменения: 11 ноября 2016 г. Последнее поле в списке - это имя файла, **/usr/bin/gzip**.

Вывод **ls** немного отличается для файла устройства. Например:

$ **ls -l /dev/tty0**

crw--w----. 1 root tty 4, 0 Aug 3 15:12 /dev/tty0

Большинство полей такие же, но вместо размера в байтах **ls** показывает мажорные и минорные номера устройства. **/dev/tty0** является первой виртуальной консолью в этой системе (Red Hat) и управляется драйвером устройства 4 (драйвером терминала). Точка в конце поля режима указывает на отсутствие списка управления доступом (ACL, обсуждается, начиная со страницы 140). Некоторые системы показывают эту информацию по умолчанию, а некоторые нет.

Одна из опций **ls**, полезная для определения жестких ссылок, **-i**, которая указывает **ls** показывать "inode каждого файла". Вкратце, inode (номер индекса) - это целое число, связанное с содержимым файла. Inode - это "вещи", на которые указывают записи каталога; записи, которые являются жесткими ссылками на один и тот же файл, имеют одинаковый inode. Чтобы выяснить сложную сеть ссылок, вам нужно использовать как **ls -li**, чтобы показать количество ссылок и inode, так и **find** для поиска совпадений7.

*7. Попробуйте использовать* ***find*** *mountpoint* ***-xdev -inum*** *inode* ***-print****.*

Перечислим некоторые другие параметры **ls**, которые важно знать: **-a** - показать все записи в каталоге (даже файлы, имена которых начинаются с точки), **-t** - отсортировать файлы по времени изменения (или **-tr** отсортировать в обратном хронологическом порядке), **-F** - для отображения имен файлов таким образом, чтобы различать каталоги и исполняемые файлы, **-R** - для рекурсивного отображения и **-h** для отображения размеров файлов в удобочитаемой форме (например, 8K или 53M).

Большинство версий **ls** теперь по умолчанию используют файлы с цветовым кодированием, если ваша терминальная программа поддерживает эту функциональность (большинство поддерживает). **ls** определяет цвета в соответствии с ограниченной и абстрактной палитрой ("красный", "синий" и т. д.), и программа терминала может сама сопоставить эти запросы с конкретными цветами. Вам может понадобиться настроить как **ls** (переменная окружения LSCOLORS или LS\_COLORS), так и эмулятор терминала, чтобы получить цвета, которые будут читабельными и ненавязчивыми. Кроме того, вы можете просто удалить дефолтную конфигурацию для раскрашивания (обычно **/etc/profile.d/colorls\***), чтобы полностью отключить цвета.

**chmod: изменение разрешений**

Команда **chmod** изменяет права доступа к файлу. Только владелец файла и суперпользователь могут изменять права доступа к файлу. Чтобы использовать эту команду в ранних системах UNIX, вам пришлось бы немного изучить восьмеричные записи, но в текущих версиях допускаются как восьмеричная запись, так и мнемонический синтаксис. Восьмеричный синтаксис обычно более удобен для администраторов, но его можно использовать только для указания абсолютного значения для битов разрешения. Мнемонический синтаксис может изменять отдельные биты, оставляя другие в покое.

Первый аргумент **chmod** - это спецификация назначаемых разрешений, а второй и последующие аргументы - это имена файлов, для которых необходимо изменить разрешения. В восьмеричном представлении первая восьмеричная цифра спецификации предназначена для владельца, вторая - для группы, а третья - для всех остальных. Если вы хотите включить биты setuid, setgid или sticky, вы должны использовать четыре восьмеричные цифры, а не три, причем три специальных бита образуют первую цифру.

Таблица 5.3 иллюстрирует восемь возможных комбинаций для каждого набора из трех битов, где r, w и x означают чтение, запись и выполнение.

**Таблица 5.3 Кодирование разрешений для chmod**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Восьмеричное представление** | **Двоичное представление** | **Разрешение** | **Восьмеричное представление** | **Двоичное представление** | **Разрешение** |
| 0 | 000 | --- | 4 | 100 | r-- |
| 1 | 001 | --x | 5 | 101 | r-x |
| 2 | 010 | -w- | 6 | 110 | rw- |
| 3 | 011 | -wx | 7 | 111 | rwx |

Например, **chmod 711 myprog** предоставляет все разрешения пользователю (владельцу) и разрешает только выполнение всем остальным8.

*8. Если бы* ***myprog*** *был shell скриптом, ему нужно было бы включить биты на чтение и выполнение. Чтобы скрипт выполнялся интерпретатором, его нужно открыть и прочитать как текстовый файл. Бинарные файлы выполняются непосредственно ядром и поэтому не требуют разрешения на чтение.*

В мнемоническом синтаксисе вы объединяете набор целей (**u**, **g** или **o** для пользователя, группы, всех остальных или **a** для всех трех) с оператором (**+**, **-**, **=** для добавления, удаления или установки) и набором разрешений. Страница руководства **chmod** содержит подробности, но синтаксис, вероятно, лучше всего изучить на примерах. Таблица 5.4 иллюстрирует некоторые мнемонические операции.

Сложной частью использования мнемонического синтаксиса является запоминание, что означает **o** - "владелец" или "все остальные"; правильные ответ - "все остальные". Вспомните порядок букв в имени H**ugo**.

**Таблица 5.4 Примеры мнемонического синтаксиса chmod**

|  |  |
| --- | --- |
| **Спецификация** | **Значение** |
| **u+w** | Добавляет разрешение на запись для владельца файла |
| **ug=rw,o=r** | Дает разрешение на чтение и запись владельцу и группе, а также разрешение на чтение всем остальным |
| **a-x** | Удаляет разрешение на выполнение для всех категорий (владелец/группа/остальные) |
| **ug=srx,o=** | Создает setuid / setgid и дает разрешение на чтение и запись только владельцу и группе |
| **g=u** | Делает разрешения группы такими же, как разрешения для владельца |

В системах Linux вы также можете указать режимы, которые нужно назначить, скопировав их из существующего файла. Например, **chmod --reference=filea fileb** делает права **fileb** таким же, как у **filea**.

**chmod** с опцией **-R** рекурсивно обновляет права доступа к файлам в каталоге. Однако эта опция сложнее, чем кажется, потому что вложенные файлы и каталоги могут иметь неодинаковые атрибуты; например, некоторые файлы могут быть исполняемыми; другие - текстовыми. Мнемонический синтаксис особенно полезен с опцией **-R**, потому что он сохраняет биты, значения которых вы не устанавливаете явно. Например,

$ **chmod -R g+w mydir**

добавляет разрешение на запись для группы в **mydir** и все его содержимое, не путая биты выполнения каталогов и программ.

Если вы хотите настроить биты выполнения, будьте осторожны с **chmod -R**. Он не учитывает тот факт, что бит выполнения имеет другую интерпретацию для каталога, чем для простого файла. Следовательно, **chmod -R a-x**, вероятно, не сделает того, что вы хотите. Используйте **find**, чтобы выбрать только обычные файлы:

$ **find mydir -type f -exec chmod a-x {} ';'**

**chown и chgrp: смена владельца и группы**

Команда **chown** меняет владельца файла, а команда **chgrp** меняет владельца группы. Синтаксис **chown** и **chgrp** повторяют синтаксис **chmod**, за исключением того, что первым аргументом является новый владелец или группа соответственно.

Чтобы изменить группу файла, вы должны быть суперпользователем или владельцем файла и принадлежать к группе, на которую меняете разрешение. Старые системы, использующие SysV позволяли пользователям отдавать свои собственные файлы с помощью **chown**, но в наши дни такое поведение является необычным; сейчас **chown** является привилегированной операцией.

Как и **chmod**, **chown** и **chgrp** могут использовать рекурсивный флаг **-R** для изменения настроек каталога и всех файлов, находящихся в нем. Например, последовательность

$ **sudo chown -R matt ~matt/restore**

$ **sudo chgrp -R staff ~matt/restore**

может сбросить владельца и группу файлов, восстановленных из резервной копии для пользователя matt. Не пытайтесь использовать **chown** на файлах с точкой в имени с помощью такой команды, как

$ **sudo chown -R matt ~matt/.\***

поскольку шаблон соответствует **~matt/..** и, следовательно, в конечном итоге меняет владельца родительского каталога и, возможно, домашних каталогов других пользователей.

**chown** может сразу менять владельца и группу файла с помощью синтаксиса

**chown** *user:group file ...*

Например,

$ **sudo chown -R matt:staff ~matt/restore**

На самом деле вы можете не указывать ни пользователя, ни группу, что делает команду **chgrp** излишней. Если вы включаете двоеточие, но не называете конкретную группу, версия **chown** для Linux использует дефолтную пользовательскую группу.

Некоторые системы принимают обозначение *user.group* как эквивалент *user:group*. Это просто дань историческим вариациям среди систем; оно означает то же самое.

**umask: присваивание дефолтных разрешений**

Вы можете использовать встроенную shell команду **umask**, чтобы влиять на дефолтные разрешения, присваиваемые для создаваемых вами файлов. Каждый процесс имеет свой собственный атрибут **umask**; встроенная shell команда **umask** устанавливает собственный **umask** оболочки, который затем наследуется командами, которые вы запускаете.

**umask** указывается в виде восьмеричного значения из трех цифр, которое представляет разрешения на изъятие. Когда файл создается, его разрешения устанавливаются на то, что запрашивает программа создания, минус все, что запрещает **umask**. Таким образом, отдельные цифры umask допускают разрешения, показанные в таблице 5.5.

**Таблица 5.5 Кодировка прав доступа umask**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Восьмеричное представление** | **Двоичное представление** | **Разрешение** | **Восьмеричное представление** | **Двоичное представление** | **Разрешение** |
| 0 | 000 | rwx | 4 | 100 | -wx |
| 1 | 001 | rw- | 5 | 101 | -w- |
| 2 | 010 | r-x | 6 | 110 | --x |
| 3 | 011 | r-- | 7 | 111 | --- |

Например, **umask 027** предоставляет все разрешения для владельца, но запрещает права на запись для группы и не допускает никаких разрешений для всех остальных. Частое дефолтное значение **umask** - 022, которое запрещает запись для группы и всех остальных, но разрешает чтение.

*См. Главу 8 для получения дополнительной информации о файлах запуска.*

В стандартной модели управления доступом вы не можете заставить пользователей иметь конкретное значение **umask**, потому что они всегда могут изменить его состояние до того, что они хотят. Однако вы можете указать подходящее дефолтное значение в примерах файлов запуска, которые вы даете новым пользователям. Если вам требуется больший контроль над разрешениями создаваемых пользователем файлов, вам необходимо перейти на систему обязательного контроля доступа, такую как SELinux; см. стр. 84.

**Бонусные флаги Linux**

Linux определяет набор дополнительных флагов, которые могут быть установлены в файлах для запроса специальной обработки. Например, флаг **a** делает файл доступным только для добавления, а флаг **i** делает его неизменяемым и неудаляемым.

Флаги имеют двоичные значения, поэтому они либо присутствуют, либо отсутствуют для данного файла. Базовая реализация файловой системы должна поддерживать соответствующую функцию, поэтому не все флаги могут использоваться на всех типах файловых систем. В дополнение, некоторые флаги являются экспериментальными, не реализованными или только для чтения.

Linux использует команды **lsattr** и **chattr** для просмотра и изменения атрибутов файла. В таблице 5.6 перечислены некоторые из наиболее распространенных флагов.

**Таблица 5.6. Флаги атрибутов файлов Linux**

|  |  |  |
| --- | --- | --- |
| **Флаг** | **ФСа** | **Значение** |
| **A** | **XBE** | Никогда не обновлять время доступа (st\_atime; для производительности) |
| **a** | **XBE** | Разрешить запись только в режиме добавленияb |
| **C** | **B** | Отключить обновления copy-on-write |
| **c** | **B** | Сжать содержимое |
| **D** | **BE** | Принудительная синхронная запись обновлений каталога |
| **d** | **XBE** | Не резервировать; утилиты резервного копирования должны игнорировать этот файл |
| **i** | **XBE** | Сделать файл неизменным и неудаляемымb |
| **j** | **E** | Вести журнал для изменений данных, а также метаданных |
| **S** | **XBE** | Принудительная синхронная запись изменений (без буферизации) |
| **X** | **B** | Избегать сжатия данных, если оно установлено по умолчанию |

1. X = XFS, B = Btrfs, E = ext3 и ext4
2. Может быть установлен только пользователем root

Как и следовало ожидать от такого случайного набора функций, значение этих флагов для администраторов варьируется. Главное, что нужно помнить, это то, что если какой-то конкретный файл ведет себя странно, проверьте его с помощью **lsattr**, чтобы убедиться, что у него включен один или несколько флагов.

Отказ от поддержки времени последнего доступа (флаг **A**) может повысить производительность в некоторых ситуациях. Однако это значение зависит от реализации файловой системы и схемы доступа; вам придется сделать свой собственный сравнительный анализ. Кроме того, современные ядра теперь по умолчанию монтируют файловые системы с параметром *relatime*, который сводит к минимуму обновления *st\_atime* и делает флаг **A** в значительной степени устаревшим.

Неизменяемые флаги и флаги только для добавления (**i** и **a**) были в основном задуманы как способы сделать систему более устойчивой к взлому хакеров или враждебному коду. К сожалению, они могут запутать программное обеспечение и защитить только от хакеров, которые не знают достаточно, чтобы использовать **chattr -ia**. Реальный опыт показывает, что эти флаги чаще используются *самими* хакерами, чем против них.

*См. Главу 23 для получения дополнительной информации об управлении конфигурациями.*

Мы видели несколько случаев, когда администраторы использовали флаг **i** (immutable), чтобы предотвратить изменения, которые иначе были бы внесены системами управления конфигурациями, такими как Ansible или Salt. Излишне говорить, что этот хак создает путаницу, когда детали забыты, и никто не может понять, почему не работает управление конфигурациями. Никогда не делай этого - просто подумай о стыде, который почувствовала бы твоя мать, если бы она знала, что ты задумал. Исправьте проблему в системе управления конфигурациями, как хотелось бы маме.

Флаг "не резервировать" (**d**) потенциально представляет интерес для администраторов, но поскольку он является рекомендательным, убедитесь, что ваша система резервного копирования соблюдает его.

Флаги, которые влияют на синхронизацию журналирования и записи (**D**, **j** и **S**), существуют главным образом для поддержки баз данных. В основном они не используются администраторами. Все эти параметры могут значительно снизить производительность файловой системы. Кроме того, вмешательство в синхронность записи, как известно, приводит к путанице **fsck** в файловых системах ext\*.

**5.6 Списки контроля доступа**

Традиционная 9-битная система контроля доступа владелец/группа/остальные достаточно мощна, чтобы удовлетворить подавляющее большинство административных потребностей. Хотя система имеет четкие ограничения, она в значительной степени соответствует традициям UNIX (некоторые могут сказать, "былые традиции") простоты и предсказуемости.

Списки контроля доступа, или ACL, являются более мощным, но и более сложным способом регулирования доступа к файлам. С каждым файлом или каталогом может быть связан ACL, в котором перечислены правила разрешений, которые необходимо применить к нему. Каждое из правил в ACL называется записью контроля доступа или ACE.

Запись контроля доступа идентифицирует пользователя или группу, к которой она применяется, и указывает набор разрешений, которые должны применяться к этим объектам. Списки ACL не имеют заданной длины и могут включать спецификации разрешений для нескольких пользователей или групп. Большинство ОС ограничивают длину отдельных ACL, но она достаточно велика (обычно не менее 32 записей), и редко используется.

Более сложные системы ACL позволяют администраторам указывать частичные наборы разрешений или отрицательные разрешения. Большинство из них также имеют функции наследования, которые позволяют спецификациям доступа распространяться на вновь создаваемые объекты файловой системы.

**Предупреждение**

ACL широко поддерживаются и занимают наше внимание до конца этой главы. Тем не менее, ни один из этих фактов не должен толковаться как побуждение к их принятию. ACL имеют свою нишу, но она лежит за пределами основного направления администрирования UNIX и Linux.

ACL существуют главным образом для обеспечения совместимости с Windows и удовлетворения потребностей небольшого сегмента предприятий, которым фактически требуется гибкость на уровне ACL. Они не являются блестящим следующим поколением контролем доступа и не предназначены для замены традиционной модели.

Сложность ACL создает несколько потенциальных проблем. ACL не только утомительны в использовании, но также могут вызвать неожиданные взаимодействия с системами резервного копирования, не поддерживающими ACL, сетевыми файловыми службами и даже простыми программами, такими как текстовые редакторы.

ACL также имеют тенденцию становиться все труднее в обслуживании по мере роста количества записей. Реальные ACL часто включают в себя рудиментарные записи и записи, которые служат только для компенсации проблем, вызванных предыдущими записями. Можно реорганизовать и упростить эти сложные ACL, но это рискованно и требует много времени, поэтому это редко делается.

В прошлом копии этой главы, которые мы рассылали профессиональным администраторам для проверки, часто возвращались с примечаниями, такими как "Эта часть выглядит хорошо, но я не могу ее подтвердить, потому что я никогда не использовал ACL".

**Типы ACL**

В качестве преобладающих стандартов для UNIX и Linux появились два типа ACL: POSIX ACL и NFSv4 ACL.

Версия POSIX датируется работой над спецификациями, выполненной в середине 1990-х годов. К сожалению, фактический стандарт не был выпущен, и начальные реализации сильно различались. В наши дни мы в гораздо лучшей форме. Системы в основном сошлись на общем фрейме для POSIX ACL и общем наборе команд, **getfacl** и **setfacl**, для управления ими.

С первого взгляда модель POSIX ACL просто расширяет традиционную систему разрешений *rwx* в UNIX для учета разрешений для нескольких групп и пользователей.

По мере того, как POSIX ACL становились все более популярными, для UNIX и Linux стало все более распространенным совместное использование файловых систем с Windows, которая имеет собственный набор соглашений ACL. Здесь сюжет усложняется, потому что в Windows есть множество различий, которых нет ни в традиционной модели UNIX, ни в ее эквиваленте POSIX ACL. ACL в Windows также семантически более сложны; например, они допускают отрицательные разрешения ("deny" записи) и имеют сложную схему наследования.

*См. Главу 21 для получения дополнительной информации о NFS.*

Архитекторы 4 версии NFS - общего протокола обмена файлами - хотели включить ACL в качестве первоклассного объекта. Из-за разделения UNIX / Windows и несоответствий между реализациями UNIX ACL было ясно, что системы на концах соединения NFSv4 часто могут быть разных типов. Каждая система может понимать NFSv4 ACL, POSIX ACL, Windows ACL или вообще не иметь ACL. Стандарт NFSv4 должен был бы взаимодействовать со всеми этими различными мирами, не вызывая слишком много сюрпризов или проблем безопасности.

Учитывая это ограничение, возможно, неудивительно, что NFSv4 ACL, по сути, являются объединением всех существующих систем. Они являются расширенным набором POSIX ACL, поэтому любой POSIX ACL может быть представлен как NFSv4 ACL без потери информации. В то же время NFSv4 ACL вмещают в себя все биты разрешений, имеющиеся в системах Windows, а также имеют большинство семантических функций Windows.

**Реализация ACL**

Теоретически, ответственность за поддержание и применение ACL может быть возложена на несколько различных компонентов операционной системы. ACL могут быть реализованы ядром от имени всех файловых систем системы, отдельными файловыми системами или, возможно, программным обеспечением более высокого уровня, таким как серверы NFS и SMB.

На практике поддержка ACL зависит как от ОС, так и от файловой системы. Файловая система, которая поддерживает ACL в одной системе, может не поддерживать их в другой, или она может иметь несколько иную реализацию, управляемую другими командами.

*См. Главу 22 для получения дополнительной информации о SMB.*

Демоны файловой службы сопоставляют собственную схему (или схемы) ACL своего хоста с соглашениями, соответствующими протоколу подачи: NFSv4 ACL для NFS и Windows ACL для SMB. Детали этого сопоставления зависят от реализации файлового сервера. Обычно правила являются сложными и отчасти настраиваются с помощью параметров конфигурации.

Поскольку реализации ACL зависят от файловой системы, и потому что системы поддерживают несколько реализаций файловых систем, некоторые системы в конечном итоге поддерживают несколько типов ACL. Даже данная файловая система может предлагать несколько вариантов ACL, как видно из различных портов ZFS. Если доступно несколько систем ACL, команды для управления ими могут быть одинаковыми или разными; это зависит от системы. Добро пожаловать в ад сисадминов.

**Поддержка ACL в Linux**

Linux стандартизирован на ACL в стиле POSIX. NFSv4 ACL не поддерживаются на уровне файловой системы, хотя, конечно, системы Linux могут монтировать и совместно использовать файловые системы NFSv4 по сети.

Преимущество этой стандартизации состоит в том, что почти все файловые системы Linux теперь поддерживают POSIX ACL, включая XFS, Btrfs и семейство ext \*. Даже ZFS, чья собственная система ACL - это NFSv4, была портирована на Linux с POSIX ACL. Стандартные команды **getfacl** и **setfacl** могут использоваться везде, независимо от используемого типа файловой системы. Однако вам может потребоваться убедиться, что для монтирования файловой системы была использована правильная опция **mount**. Файловые системы обычно поддерживают опцию *acl*, опцию *noacl* или сразу обе, в зависимости от их дефолтных настроек.

В Linux есть набор команд (**nfs4\_getfacl**, **nfs4\_setfacl** и **nfs4\_editfacl**) для обработки NFSv4 ACL файлов, смонтированных с серверов NFS. Однако эти команды нельзя использовать для локально хранимых файлов. Более того, они редко включаются в стандартный перечень программного обеспечения дистрибутивов; вам придется устанавливать их отдельно.

**Поддержка ACL в FreeBSD** 

FreeBSD поддерживает как POSIX ACL, так и NFSv4 ACL. Его собственные команды **getfacl** и **setfacl** были расширены, чтобы включить в себя wrangling в стиле NFSv4 ACL. Поддержка NSFv4 ACL является относительно недавней (по состоянию на 2017 год) разработкой.

На уровне файловой системы UFS и ZFS поддерживают ACL в стиле NFSv4, а UFS также поддерживает POSIX ACL. Потенциальная путаница здесь - это ZFS, которая поддерживает NFSv4 ACL только в BSD (также в Solaris и ее производных) и POSIX ACL только в Linux.

Для UFS используйте один из параметров **mount** *acls* или *nfsv4acls*, чтобы указать, в каком мире вы хотите жить. Эти параметры являются взаимоисключающими.

**POSIX ACL**

POSIX ACL - это в основном прямое расширение стандартной 9-битной модели разрешений UNIX. Разрешение на чтение, запись и выполнение-это единственные возможности, с которыми имеет дело система ACL. Украшения, такие как setuid и sticky-биты, обрабатываются исключительно через биты традиционной модели.

ACL позволяют независимо устанавливать биты *rwx* для любой комбинации пользователей и групп. Таблица 5.7 показывает, как могут выглядеть отдельные записи в ACL.

**Таблица 5.7. Записи, которые могут появляться в ACL POSIX**

|  |  |  |
| --- | --- | --- |
| **Формат** | **Пример** | **Устанавливает разрешения для** |
| user::*perms* | user::rw- | Владельца файла |
| user:*username:perms* | user:trent:rw- | Определенного пользователя |
| group::*perms* | group::r-x | Группы владельцев файла |
| group:*groupname:perms* | group:staff:rw- | Определенной группы |
| other::*perms* | other::--- | Всех остальных |
| mask::*perms* | mask::rwx | Всех, кроме владельца и остальныхa |

1. Использование масок несколько хитро и объясняются далее в этом разделе.

Пользователи и группы могут быть идентифицированы по имени или по UID/GID. Точное количество записей, которые может содержать ACL, зависит от реализации файловой системы, но обычно составляет не менее 32. В любом случае, это, вероятно, практический предел управляемости.

*Взаимодействие традиционных моделей и ACL*

Файлы с ACL сохраняют свои исходные биты режима, но согласованность обеспечивается автоматически, и два набора разрешений никогда не могут конфликтовать. В следующем примере показано, что записи ACL автоматически обновляются в ответ на изменения, внесенные стандартной командой **chmod**:

$ **touch example**

$ **ls -l example**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -rw-rw-r-- | 1 | garth | garth | 0 | Jun 14 | 15:57 | example |

$ **getfacl example**

# file: example

# owner: garth

# group: garth

user::rw-

group::rw-

other::r--

$ **chmod 640 example**

$ **ls -l example**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -rw-r----- | 1 | garth | garth | 0 | Jun 14 | 15:57 | example |

$ **getfacl --omit-header example9**

user::rw-

group::r--

other::---

*9. Этот пример взят из Linux. FreeBSD-версия* ***getfacl*** *использует* ***-q*** *вместо* ***--omit-header*** *для скрытия подобных строк в выводе.*

Такая принудительная согласованность позволяет старому программному обеспечению, не знающему ACL, достаточно хорошо работать в мире ACL. Тем не менее, есть нюанс. Хотя запись *group::* в ACL в приведенном выше примере, кажется, отслеживает средний набор битов традиционного режима, это не всегда так.

Чтобы понять почему так происходит, предположим, что устаревшая программа очищает биты записи во всех трех наборах разрешений традиционного режима (например, **chmod ugo-w** *file*). Намерение явно состоит в том, чтобы убрать разрешение на запись для всех. Но что, если если полученный ACL будет выглядеть так?

user::r--

group::r--

group:staff:rw-

other::r--

С точки зрения устаревших программ, файл кажется неизменным, но на самом деле он доступен для записи любому участнику группы staff. Не хорошо. Чтобы уменьшить вероятность двусмысленности и недопонимания, применяются следующие правила:

* Записи user:: и other:: в ACL по определению идентичны битам разрешения "owner" и "other" из традиционного режима. Изменение режима изменяет соответствующие записи ACL, и наоборот.
* Во всех случаях действительными правами доступа, предоставляемыми владельцу файла и пользователям, не упомянутым иным образом, являются права, указанные в записях user:: и other:: ACL соответственно.
* Если файл не имеет явно определенного ACL или ACL, состоящего только из одной записи для user::, group:: и other::, эти записи ACL идентичны трем наборам традиционных битов разрешений. Это тот случай, который показан в примере **getfacl** выше. Такой ACL называется "минимальным" и не обязательно должен быть реализован как логически отдельный ACL.
* В более сложных ACL традиционные биты разрешения группы соответствуют специальной записи ACL, называемой *mask*, а не ACL запись *group::*. Маска ограничивает доступ, который ACL может предоставить всем указанным пользователям, всем указанным группам и группе по умолчанию.

Другими словами, маска определяет верхнюю границу прав доступа, которые ACL может назначить отдельным группам и пользователям. Концептуально она похожа на **umask**, за исключением того, что маска ACL действует всегда и что она определяет разрешения, а не запреты. Записи ACL для указанных пользователей, указанных групп и дефолтных групп могут включать биты разрешений, которых нет в маске, но файловые системы просто игнорируют их.

В результате биты традиционного режима никогда не могут преуменьшить доступ, разрешенный ACL в целом. Кроме того, удаление бита из групповой части традиционного режима удаляет соответствующий бит в маске ACL и тем самым запрещает это разрешение всем, кроме владельца файла и тех, кто попадает в категорию "все остальные".

Когда ACL, показанный в предыдущем примере, расширяется и включает записи для определенного пользователя и группы, **setfacl** автоматически предоставляет соответствующую маску:

$ **ls -l example**

-rw-r----- 1 garth garth 0 Jun 14 15:57 example

$ **setfacl -m user::r,user:trent:rw,group:admin:rw example**

$ **ls -l example**

-rw-r-----+ 1 garth garth 0 Jun 14 15:57 example

$ **getfacl --omit-header example**

user::r--

user:trent:rw-

group::r--

group:admin:rw-

mask::rw-

other::---

Параметр **-m** в **setfacl** означает "modify" (изменить): он добавляет записи, которых еще нет, и корректирует те, которые уже есть. Обратите внимание, что **setfacl** автоматически создает маску, которая позволяет всем разрешениям, предоставленным в ACL, вступать в силу. Если вы хотите установить маску вручную, включите ее в список записей ACL, передаваемый для **setfacl**, или используйте опцию **-n**, чтобы **setfacl** не генерировал ее заново.

Обратите внимание, что после команды **setfacl**, **ls -l** показывает знак "+" в конце режима файла, чтобы обозначить, что с ним теперь связан реальный ACL. Первый вывод команды **ls -l** не показывает +, потому что в тот момент ACL являлся "минимальным".

Если вы используете традиционную команду chmod для манипулирования файлом, содержащим ACL, учтите, что ваши настройки для разрешений "группа" влияют только на маску. Продолжим предыдущий пример:

$ **chmod 770 example**

$ **ls -l example**

-rwxrwx---+ 1 garth staff 0 Jun 14 15:57 example

$ **getfacl --omit-header example**

user::rwx

user:trent:rw-

group::r--

group:admin:rw-

mask::rwx

other::---

Вывод **ls** в этом случае вводит в заблуждение. Несмотря на очевидно щедрые разрешения для группы, на самом деле никто не имеет разрешения на запуск файла по причине членства в группе. Чтобы предоставить такое разрешение, вы должны отредактировать сам ACL.

Чтобы полностью удалить ACL и вернуться к стандартной системе разрешений UNIX, используйте **setfacl -bn**. Строго говоря, флаг **-n** необходим только во FreeBSD. Без него **setfacl** в FreeBSD оставляет вам запись рудиментарной маски, которая испортит последующие изменения разрешений для группы. Однако вы можете использовать опцию **-n** в Linux без каких-либо проблем.

*Определение доступа в POSIX*

Когда процесс пытается получить доступ к файлу, его эффективный UID сравнивается с UID, которому принадлежит файл. Если они одинаковы, доступ определяется в разрешениях ACL *user::*. В противном случае, если существует соответствующая пользовательская запись ACL, разрешения определяются этой записью в сочетании с маской ACL.

Если нет никакой доступной записи для пользователя, файловая система пытается найти действительную связанную с группой запись, которая авторизует запрос доступа; эти записи обрабатываются вместе с маской ACL. Если подходящей записи не найдено, превалирует запись *other::*.

*Наследование в POSIX ACL*

В дополнение к типам записей ACL, перечисленных в таблице 5.7, ACL для каталогов могут включать дефолтные записи, которые распространяются на ACL вновь созданных файлов и подкаталогов, созданных в них. Подкаталоги получают эти записи как в виде активных записей ACL, так и в виде копий дефолтных записей. Следовательно, исходные дефолтные записи могут в конечном итоге распространяться вниз по нескольким уровням иерархии каталогов.

После того, как дефолтные записи были скопированы в новые подкаталоги, между родительским и дочерним ACL-списками нет постоянной связи. Если родительские дефолтные записи изменяются, эти изменения не отражаются в ACL существующих подкаталогов.

Вы можете установить дефолтные записи ACL с помощью **setfacl -dm**. В качестве альтернативы, вы можете включить дефолтные записи в постоянный список записей управления доступом, добавив к ним префикс *default:*.

Если каталог имеет некоторые дефолтные записи, он должен включать полный набор дефолтных значений для *user::*, *group::*, *other::* и *mask::*. **setfacl** будет заполнять любые дефолтные записи, которые вы не укажете, копируя их из текущего списка разрешений ACL, как обычно генерируя итоговую маску.

**NFSv4 ACL**  

В этом разделе мы обсудим характеристики списков NFSv4 ACL и кратко рассмотрим синтаксис команды, используемый для установки и проверки их во FreeBSD. Они не поддерживаются в Linux (кроме демонов службы NFS).

Со структурной точки зрения списки NFSv4 ACL аналогичны спискам Windows ACL. Основное различие между ними заключается в спецификации объекта, к которому относится запись контроля доступа.

В обеих системах ACL хранит эту сущность в виде строки. В Windows ACL строка обычно содержит идентификатор безопасности Windows (SID), тогда как в NFSv4 строка обычно имеет вид *user:username* или *group:groupname*. Это также может быть один из специальных токенов *owner@*, *group@* или *everyone@*. Эти последние записи являются наиболее распространенными, потому что они соответствуют битам режима, находящимся в каждом файле.

Системы, такие как Samba, которые предоставляют общий доступ к файлам в системах UNIX и Windows, должны обеспечивать какой-либо способ сопоставления между пользователями Windows и NFSv4.

Модели разрешений NFSv4 и Windows более гранулярны, чем традиционная модель чтение-запись-выполнение в UNIX. В случае NFSv4 основные преимущества заключаются в следующем:

* NFSv4 отличает разрешение на создание файлов в каталоге от разрешения на создание подкаталогов.
* NFSv4 имеет отдельный бит разрешения "append" (добавить/присоединить).
* NFSv4 имеет отдельные разрешения на чтение и запись для данных, атрибутов файлов, расширенных атрибутов и списков ACL.
* NFSv4 контролирует способность пользователя изменять владельца файла через стандартную систему ACL. В традиционной UNIX возможность смены владельца файлов обычно зарезервирована для root.

Таблица 5.8 показывает различные разрешения, которые могут быть назначены в системе NFSv4. Она также показывает односимвольные коды, используемые для их представления, и более подробные канонические имена.

**Таблица 5.8 Права доступа к файлам в NFSv4**

|  |  |  |
| --- | --- | --- |
| **Код** | **Полное имя** | **Разрешения** |
| r | read\_data | Чтение данных (файла) или вывод списка содержимого каталога |
| w | write\_data | Записать данные (файл) или создать файл (каталог) |
| x | execute | Выполнить как программу |
| p | append\_data | Добавить данные (файл) или создать подкаталог (каталог) |
| D | delete\_child | Удалить дочерний объект в каталоге |
| d | delete | Удалить |
| a | read\_attributes | Читать нерасширенные атрибуты |
| A | write\_attributes | Записать нерасширенные атрибуты |
| R | read\_xattr | Чтение именованных ("расширенных") атрибутов |
| W | write\_xattr | Запись именованных ("расширенных") атрибутов |
| c | read\_acl | Читать список контроля доступа |
| C | write\_acl | Записать список контроля доступа |
| o | write\_owner | Смена владельца |
| s | synchronize | Разрешить запросы на синхронный ввод/вывод (обычно игнорируется) |

Хотя модель разрешений NFSv4 довольно подробная, отдельные разрешения в основном должны быть понятны сами по себе. Разрешение "synchronize" позволяет клиенту указывать, что его изменения в файле должны быть синхронными, то есть вызовы для записи не должны возвращаться до тех пор, пока данные не будут фактически сохранены на диске.

Расширенный атрибут - это именованный фрагмент данных, который хранится вместе с файлом; большинство современных файловых систем поддерживают такие атрибуты. На этом этапе преимущественное использование расширенных атрибутов заключается в хранении самих ACL. Однако модель разрешений NFSv4 обрабатывает ACL отдельно от других расширенных атрибутов.

В реализации FreeBSD владелец файла всегда имеет права доступа *read\_acl*, *write\_acl*, *read\_attributes* и *write\_attributes*, даже если сам ACL файла указывает иное.

*Сущности NFSv4, для которых могут определяться разрешения*

В дополнение к большому количеству спецификаторов *user:username* и *group:groupname*, NFSv4 определяет несколько специальных сущностей, которым могут быть назначены разрешения в ACL. Наиболее важными среди них являются *owner@*, *group@* и *everyone@*, которые соответствуют традиционным категориям в 9-битной модели разрешений.

NFSv4 имеет несколько отличий от POSIX. Во-первых, у него нет *дефолтной* сущности, используемой в POSIX для управления наследованием ACL. Вместо этого любая отдельная запись управления доступом (access control entry - ACE) может быть помечена как наследуемая (см. *Наследование ACL в NFSv4* ниже). NFSv4 также не использует маску для согласования разрешений, указанных в режиме файла, с его ACL. Режим должен соответствовать параметрам, указанным для *owner@*, *group@* и *everyone@*, а файловые системы, которые реализуют NFSv4 ACL, должны сохранять эту согласованность при обновлении режима или ACL.

*Определение доступа в NFSv4*

Система NFSv4 отличается от POSIX тем, что ACE определяет только частичный набор разрешений. Каждая ACE является либо "разрешающей" ACE, либо "запрещающей" ACE; она действует скорее как маска, чем как достоверная спецификация всех возможных разрешений. Несколько ACE могут применяться к любой конкретной ситуации.

При принятии решения о разрешении конкретной операции, файловая система считывает ACL по порядку, обрабатывая ACE, пока либо не будут предоставлены все запрошенные разрешения, либо не будет отказано в некоторых запрошенных разрешениях. Рассматриваются только те записи ACE, строки сущностей которых совместимы с личностью текущего пользователя.

Этот итеративный процесс оценки означает, что *owner@*, *group@* и *everyone@* не являются точными аналогами соответствующих битов традиционного режима. ACL может содержать несколько копий этих элементов, и их приоритет определяется порядком их появления в ACL, а не соглашением. В частности, *everyone@* действительно относится ко всем, а не только к пользователям, определенным конкретно.

Файловая система может достичь конца ACL, не получив однозначного ответа на запрос разрешения. Стандарт NFSv4 считает, что результат не определен, но реальные реализации запрещают доступ, потому что это соглашение используется в Windows, и потому что это единственный вариант, который имеет смысл.

*Наследование ACL в NFSv4*

Как и POSIX ACL, NFSv4 ACL позволяют вновь созданным объектам наследовать записи управления доступом из родительского каталога. Тем не менее, система NFSv4 является немного более мощной и намного более запутанной. Вот важные моменты:

* Вы можете пометить любой ACE как наследуемый. Наследование для вновь созданных подкаталогов (*dir\_inherit* или *d*) и наследование для вновь созданных файлов (*file\_inherit* или *f*) помечаются отдельно.
* Вы можете применить различные записи контроля доступа к новым файлам и новым каталогам, создав отдельные записи контроля доступа в родительском каталоге и пометив их соответствующим образом. Вы также можете применить один ACE ко всем новым дочерним объектам (любого типа), включив флаги *d* и *f*.
* С точки зрения определения доступа, записи управления доступом одинаково влияют на родительский (исходный) каталог независимо от того, являются ли они наследуемыми. Если вы хотите, чтобы запись применялась к дочерним элементам, но не к самому родительскому каталогу, включите ACE флаг *inherit\_only* (*i*).
* Новые подкаталоги обычно наследуют две копии каждого ACE: один с отключенными флагами наследования, который применяется к самому подкаталогу; и один с включенным флагом inherit\_only, который устанавливает новый подкаталог для распространения его унаследованных ACE. Вы можете отключить создание этого второго ACE, включив флаг no\_propagate (n) на копии ACE родительского каталога. Конечным результатом является то, что ACE распространяется только на непосредственных потомков исходного каталога.
* Не путайте распространение записей контроля доступа с истинным наследованием. Ваша установка флага для ACE, связанного с наследованием, просто означает, что ACE будет скопирован в новые объекты. Это не создает никаких постоянных отношений между родителем и его дочерними объектами. Если позднее вы измените записи ACE в родительском каталоге, дочерние элементы не будут обновлены.

Таблица 5.9 суммирует эти различные флаги наследования.

**Таблица 5.9. Флаги наследования NFSv4 ACE**

|  |  |  |
| --- | --- | --- |
| **Код** | **Полное имя** | **Значение** |
| f | file\_inherit | Распространить этот ACE на вновь созданные файлы. |
| d | dir\_inherit | Распространить этот ACE на вновь созданные директории. |
| i | inherit\_only | Распространить, но не применять к текущему каталогу. |
| n | no\_propagate | Распространить на новые подкаталоги, но отключить наследование. |

*Просмотр NFSv4 ACL*

FreeBSD расширила стандартные команды **setfacl** и **getfacl**, используемые с POSIX ACL, для управления NFSv4 ACL. Например, здесь представлен ACL для вновь созданного каталога:

freebsd$ **mkdir example**

freebsd$ **ls -ld example**

drwxr-xr-x 2 garth staff 2 Aug 16 18:52 example/

$ **getfacl -q example**

owner@: rwxp--aARWcCos:------:allow

group@: r-x---a-R-c--s:------:allow

everyone@: r-x---a-R-c--s:------:allow

Флаг **-v** запрашивает полные имена разрешений:

freebsd$ **getfacl -qv example**

owner@:read\_data/write\_data/execute/append\_data/read\_attributes/

write\_attributes/read\_xattr/write\_xattr/read\_acl/write\_acl/

write\_owner/synchronize::allow

group@:read\_data/execute/read\_attributes/read\_xattr/read\_acl/

synchronize::allow

everyone@:read\_data/execute/read\_attributes/read\_xattr/read\_acl/

synchronize::allow

*Мы сделали отступы в этих строках вывода и обернули их косыми чертами, чтобы четче проследить структуру.*

Этот недавно созданный каталог, похоже, имеет сложный ACL, но на самом деле это всего лишь 9-битный режим, "переведенный на язык" ACL. Для файловой системы не обязательно хранить фактический ACL, потому что ACL и режим эквивалентны (как и в POSIX ACL, такие списки называются "минимальными" или "тривиальными"). Если бы в каталоге был действительный ACL, то ls показывал бы биты режима с "+" в конце (т. е. *drwxr-xr-x+*) чтобы показать его присутствие.

Каждый пункт представляет одну запись управления доступом. Формат следующий:

*entity:permissions:inheritance\_flags:type*

*entity* может быть ключевым словом *owner@*, *group@* или *everyone@* или формой, такой как *user:username* или *group:groupname*. И *permissions* и *inheritance\_flags* представляют собой разделенные слэшем списки опций в подробном выводе или побитовое представление в стиле **ls** в коротком выводе. Значение *type* ACE может быть *allow* или *deny*.

Использование двоеточия в качестве разделителя в поле *entity* затрудняет скриптам синтаксический анализ выходных данных **getfacl** независимо от используемого формата вывода. Если вам нужно обрабатывать списки ACL программным способом, лучше всего делать это через модульный API, а не анализируя вывод команды.

*Взаимодействие между ACL и режимами*

Режим и ACL должны оставаться согласованными, поэтому всякий раз, когда вы настраиваете одну из этих сущностей, другая автоматически обновляется, чтобы соответствовать ей. Системе легко определить соответствующий режим для данного ACL. Однако эмулировать традиционное поведение режима с помощью ряда записей управления доступом сложнее, особенно в контексте существующего ACL. Система часто должна генерировать несколько и, казалось бы, несовместимых наборов записей для *owner@*, *group@* и *everyone@*, которые зависят от порядка оценки их совокупного эффекта.

Как правило, лучше всего избегать вмешательства в режим файлов или каталогов после применения ACL.

*Настройка NFSv4 ACL*

Поскольку система разрешений обеспечивает согласованность между режимом файла и его ACL, все файлы имеют как минимум тривиальный ACL. Следовательно, изменения ACL всегда являются обновлениями.

Вы вносите изменения в ACL с помощью команды **setfacl**, так же, как и в режиме POSIX ACL. Основное отличие состоит в том, что порядок записей контроля доступа имеет значение для NFSv4 ACL, поэтому вам может потребоваться вставить новые записи в конкретное место в существующем ACL. Вы можете сделать это используя флаг **-a**:

**setfacl -a** *position entries file* …

Здесь *position* - это индекс существующей записи контроля доступа (нумеруется начиная с нуля), перед которой должны быть вставлены новые записи. Например, команда

$ **setfacl -a 0 user:ben:full\_set::deny ben\_keep\_out**

устанавливает запись контроля доступа в файл **ben\_keep\_out**, которая запрещает все разрешения пользователю ben. **full\_set** является сокращенной записью, которая включает в себя все возможные разрешения (напишем, что в настоящее время это будут **rwxpDdaARWcCos;** сравните с таблицей 5.8).

Поскольку новая запись контроля доступа вставлена в нулевую позицию, она является первой записью, к которой обращаются, и имеет приоритет над более "низкими" записями. Бену будет отказано в доступе к файлу, даже если, например, разрешения *everyone@* предоставят доступ остальным пользователям.

Вы также можете использовать длинные имена, такие как **write\_data**, для определения разрешений. Разделите несколько длинных имен косыми чертами. Вы не можете смешивать односимвольные коды и длинные имена в одной команде.

Как и в POSIX ACL, вы можете использовать флаг **-m** для добавления новых записей в конец существующего ACL.

Что касается сложных изменений в существующих ACL, вы можете лучше всего их достичь, выгрузив ACL в текстовый файл, отредактировав записи контроля доступа в текстовом редакторе, а затем перезагрузив весь ACL. Например:

$ **getfacl -q file > /tmp/file.acl**

|  |  |
| --- | --- |
| $ **vi /tmp/file.acl**  $ **setfacl -b -M /tmp/file.acl file** | *# Сделайте необходимые изменения* |

Опция **-b** команды **setfacl** удаляет существующий ACL перед добавлением записей контроля доступа, перечисленных в **file.acl**. Его включение позволяет удалять записи, просто удаляя их из текстового файла.