Linux не для идиотов

«Сборник рассказов и рецептов»

Составитель – К.А.Е. (Dalth)

Ядро и модули

Ядро Linux является единственным процессом, имеющим непосредственный доступ к аппаратуре – все остальные процессы обращаются к устройствам только через ядро. В ядре Linux можно выделить несколько важных подсистем: подсистему управления памятью, планировщик задач, подсистему VFS – виртуальную файловую систему и драйверы.

Подсистема управления памятью управляет распределением оперативной памяти между задачами, а также обслуживает файл подкачки, планировщик задач управляет разделением процессорного времени между задачами (процессами и нитями), подсистема VFS предназначена для обслуживания файловых операций.

Драйверы предназначены для управления устройствами и поддержки различных протоколов. Существует две разновидности драйверов – статически подключенные в ядро драйверы и загружаемые модули, первые всегда загружены, вторые могут быть загружены при необходимости и выгружены когда необходимость в них отпала. Каждый модуль и само ядро содержат сигнатуру версии – специальную метку, которая описывает версию ядра и некоторые опции, использованные при компиляции ядра. Кроме того, ядра версии 2.6 могут поддерживать цифровую подпись модулей. Это было сделано для повышения надежности системы – по умолчанию ядро не будет загружать и использовать драйверы, предназначенные для другой версии ядра, или собранные с другими опциями, поскольку это может привести к возникновению проблем. Версию ядра можно узнать с помощью команды uname:

```
[dalth@inferno dalth]$ uname -r
2.6.8.1
```

В принципе, утилиты для работы с модулями поддерживают возможность загрузки модулей, собранных для другого ядра – но пользоваться этой возможностью следует с крайней осторожностью, поскольку это может привести к непредсказуемым последствиями – от ошибок ядра (kernel panic) и вплоть до странных потерь данных и непонятных ошибок, взявшихся на пустом месте.

В большинстве дистрибутивов образ ядра располагается в каталоге /boot, а загружаемые модули ядра располагаются в /lib/modules/<версия_ядра>, там же располагается таблица зависимостей модулей, поскольку некоторые модули могут нуждаться для своей работы в других модулях (например, драйвер поддержки SCSI-дисков нуждается в драйвере поддержки SCSI – как следствие этого, если объекты какого-либо модуля используются другим драйвером, такой модуль невозможно выгрузить). Следующий листинг демонстрирует достаточно типичное

содержание каталога модулей для ядер линейки 2.6:

```
[root@viking dev]# ls -l /lib/modules/2.6.8.1/
total 616
lrwxrwxrwx   1 root root     18 Авг 27 15:36 build -> /usr/src/linux
drwxr-xr-x  10 root root   4096 Окт  1 13:55 kernel
-rw-r--r--   1 root root 108680 Окт  1 13:56 modules.alias
-rw-r--r--   1 root root     69 Окт  1 13:56 modules.ccwmap
-rw-r--r--   1 root root 153967 Окт  1 13:56 modules.dep
-rw-r--r--   1 root root     73 Окт  1 13:56 modules.ieee1394map
-rw-r--r--   1 root root    357 Окт  1 13:56 modules.inputmap
-rw-r--r--   1 root root  16658 Окт  1 13:56 modules.isapnpmap
-rw-r--r--   1 root root  85093 Окт  1 13:56 modules.pcimap
-rw-r--r--   1 root root  68078 Окт  1 13:56 modules.symbols
-rw-r--r--   1 root root 150781 Окт  1 13:56 modules.usbmap
lrwxrwxrwx   1 root root     18 Окт  1 13:22 source -> /usr/src/linux
[root@viking dev]# find /lib/modules/2.6.8.1/kernel -type f | head -20
/lib/modules/2.6.8.1/kernel/arch/i386/kernel/cpuid.ko
/lib/modules/2.6.8.1/kernel/arch/i386/kernel/microcode.ko
/lib/modules/2.6.8.1/kernel/arch/i386/kernel/msr.ko
/lib/modules/2.6.8.1/kernel/crypto/blowfish.ko
/lib/modules/2.6.8.1/kernel/crypto/deflate.ko
/lib/modules/2.6.8.1/kernel/crypto/md5.ko
/lib/modules/2.6.8.1/kernel/crypto/twofish.ko
/lib/modules/2.6.8.1/kernel/drivers/acpi/fan.ko
/lib/modules/2.6.8.1/kernel/drivers/acpi/processor.ko
/lib/modules/2.6.8.1/kernel/drivers/acpi/thermal.ko
/lib/modules/2.6.8.1/kernel/drivers/base/firmware_class.ko
/lib/modules/2.6.8.1/kernel/drivers/block/cryptoloop.ko
/lib/modules/2.6.8.1/kernel/drivers/block/loop.ko
/lib/modules/2.6.8.1/kernel/drivers/block/nbd.ko
/lib/modules/2.6.8.1/kernel/drivers/block/paride/epat.ko
/lib/modules/2.6.8.1/kernel/drivers/block/paride/paride.ko
/lib/modules/2.6.8.1/kernel/drivers/block/paride/pd.ko
/lib/modules/2.6.8.1/kernel/drivers/block/paride/pg.ko
/lib/modules/2.6.8.1/kernel/drivers/bluetooth/bcm203x.ko
/lib/modules/2.6.8.1/kernel/drivers/bluetooth/bfusb.ko
[root@viking dev]#
```

Бинарные файлы модулей содержатся в подкаталоге *kernel*, и имеют расширение "**.o**" для ядер линейки 2.4 и расширение "**.ko**" для ядер линейки 2.6. В файлах *modules.***map* перечисляются символы (функции и переменные), экспортируемые модулями.

Часто в одном каталоге с модулями содержатся ссылки на каталоги, в которых хранились исходные тексты ядра и в котором производилась сборка ядра (это ссылки source и build, соответственно). Эти ссылки, как правило, используются для того, чтобы скомпилировать модули или программы, которые зависят от версии ядра (например, эти ссылки используются при инсталляции модуля поддержки видеокарт nvidia).

Учет взаимосвязей между загруженными модулями производится с помощью счетчика ссылок – модуль увеличивает свой счетчик ссылок как только какой-либо его объект начинает использоваться другими драйверами. Когда объекты модуля освобождаются, счетчик ссылок

уменьшается. Модуль может быть выгружен, если число ссылок на него станет равно 0. В ядрах версии 2.6 существует возможность произвести принудительную выгрузку модуля даже если он используется, но этим пользоваться без крайней необходимости не рекомендуется, поскольку очень возможно возникновение ошибок.

Ядро содержит множество переменных и функций, которые используются различными драйверами, и соответственно, если какой-либо драйвер должен обратиться к такому объекту, он должен знать его адрес. Некоторые драйверы также содержат переменные и функции, которые должны быть доступны другим драйверам, и адреса таких объектов тоже размещаются в специальной таблице. При загрузке модуля ядро и программа загрузки модулей устанавливает адреса всех объектов, в которых нуждается загружаемый модуль, и только после этого модуль может начать инициализацию.

Загрузка модулей и их выгрузка осуществляются утилитами *modprobe*, *insmod* и *rmmod*. Программы *modinfo* и *depmod* предназначены для получения служебной информации о загружаемых модулях. В процессе своей работы эти утилиты опираются на конфигурационные файлы /etc/modprobe.conf (для ядер 2.6.X) или modules.conf (для ядер 2.4.X).

Загрузка операционной системы

Для компьютеров архитектуры x86 последовательность загрузки хорошо описана в специализированной литературе, но мы все-таки кратко ее повторим. После включения компьютера первым загружается BIOS. Он тестирует аппаратуру и инициализирует устройства. После этого BIOS прочитывает начальный сектор загрузочного жесткого диска, убеждается что он содержит загрузчик операционной системы, и передает управление прочитанному коду. В большинстве случаев этот код является кодом загрузчика операционной системы. Кроме загрузчика операционной системы, начальный сектор также может содержать таблицу разделов жесткого диска.

В задачи загрузчика операционной системы входит допрочтение своего кода (и, возможно, конфигурационного файла), загрузка ядра операционной системы, установка параметров для ядра и передача управления ядру. Ядро инициализирует драйверы, проверяет параметры и, опираясь на параметры, пытается смонтировать корневую файловую систему, после чего (если не было проинструктировано об ином) запускает программу /sbin/init. Дальнейшая работа init подробна описана во множестве книг и статей.

Нередко случаются ситуации, когда корневая файловая система располагается на устройстве, чей драйвер скомпилирован в виде модуля, или драйвер корневой файловой системы

скомпилирован в виде модуля. Получается замкнутый круг – чтобы смонтировать корневую файловую систему систему, необходимо прочесть драйвер, а чтобы прочесть драйвер – нужно смонтировать корневую файловую систему. Чтобы разорвать этот порочный круг, в Linux была введен поддержка *initrd* – INITial RamDisk.

Initial ramdisk – это файл, который прочитывается загрузчиком ОС и загружается в память вместе с ядром. Ядро интерпретирует фрагмент памяти, куда загружен этот файл, как блочное устройство с помощью специального драйвера, статически вкомпилированного в ядро. После инициализации статически скомпилированных драйверов ядро монтирует файловую систему, хранящуюся в initrd и загружает с нее драйверы и запускает программы, необходимые для монтирования корневой файловой системы.

Обычно файл с образом ядра хранится в каталоге /boot и называется *vmlinuz-<версия>*, там же располагается файл *initrd-<версия>.img*, содержащий образ файловой системы initrd. Для каждой версии ядра необходим свой образ initrd, в который включены модули для этой версии ядра. В большинстве случаев образ initrd поставляется в бинарном пакете вмсте с ядром, или автоматически создается в процессе построения ядра из исходных текстов в момент выполнения команды *make install*, если же возникает ситуация, когда необходимо повторно собрать образ initrd (например, в сервере сменили SCSI-контроллер), можно воспользоваться специальной командой mkinitrd, позволяющей произвести повторную генерацию initrd:

```
[root@viking dalth]# mkinitrd /tmp/initrd-2.4.8.1.img 2.6.8.1
[root@viking dalth]# cp /tmp/initrd-2.4.8.1.img /boot
[root@viking dalth]# reboot
```

Для Linux существует два основных загрузчика – LILO и GRUB. Второй является более поздней разработкой и немного удобней в использовании, а LILO используется по историческим или личным причинам (например, он нравится системному администратору), либо в некоторых случаях, когда требуются специфичные для LILO функции. Для более подробной справки лучше обратиться к справочному руководству (man grub, man lilo).

Из интересных особенностей GRUB и LILO следует отметить то, что и оба этих загрузчика, и ядро оперируют термином корневой файловой системы – но если с точки зрения ядра эта та файловая система, которая содержит программу /sbin/init, то с точки зрения обоих загрузчиков корневой файловой системой является та, которая содержит образ ядра и файл initrd.

Организация памяти

Подсистема виртуальной памяти управляет распределением оперативной памяти между задачами (процессами). Каждая задача считает, что ей выделен непрерывный участок памяти

максимального размера, поддерживаемого на соответствующей архитектуре (для архитектуры x86 это 4GB). Из них первый гигабайт резервируется для себя ядром, второй отдается под код программы и разделяемые библиотеки (оба этих фрагмента ядром защищаются), а два последних гигабайта отдаются собственно программе под ее данные – но это только то, как видит это все программа.

На самом же деле программа занимает только тот объем памяти, с которым она реально работает. Большинство памяти существует только “на бумаге”, т.е. будет предоставлена программе в тот момент, когда она обратится в эту область. Ядро распределяет память страницами фиксированного размера. Процедура, когда страница оперативной памяти объявляется частью адресного пространства процесса, называется *отображением* этой страницы в адресное пространство процесса.

Соответственно, ядро отображает реально используемые страницы в виртуальное адресное пространство процесса. Когда процесс обращается к некоторой странице своего адресного пространства, ядро проверяет, имеет ли он право на доступа к этой странице, и если проверка пройдена и доступ получен, то ядро переадресовывает обращение на реальный адрес этой страницы. Если это первое обращение к странице, ядро попытается найти свободную страницу и в случае успеха отобразит ее в адресное пространство соответствующего процесса. Размер страницы фиксирован архитектурой процессора, и для x86 ее размер составляет 4096 байт.

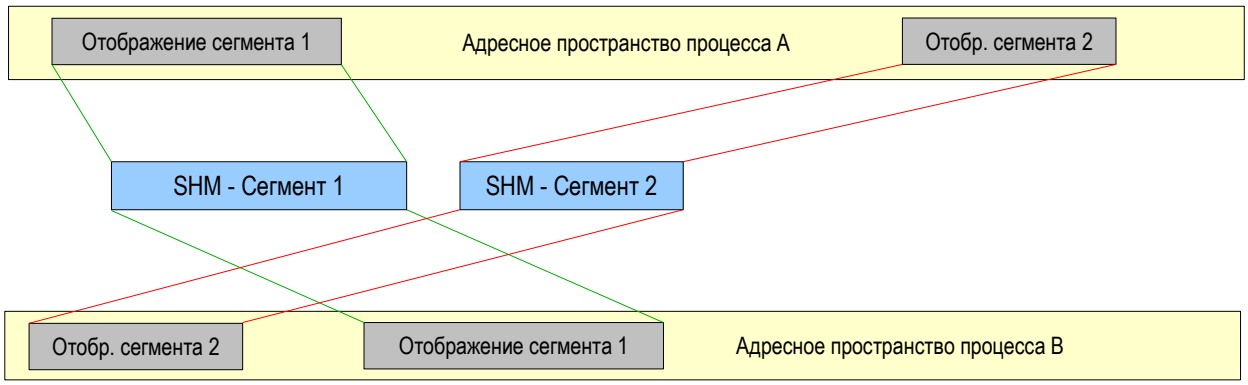
Если случается ситуация, когда свободных страниц больше нет, но существует файл подкачки, ядро может убрать одну из наиболее долго не использовавшихся страниц в файл подкачки, и освободившуюся физическую страницу отдать запросившему память процессу. Если же нет ни незанятого пространства в файле подкачки, ни свободных страниц RAM, то развитие событий может быть следующим: либо запросивший память процесс прерван и “убит” системой, либо какой-то другой из процессов (это определяется специфическими алгоритмами) будет “убит” ядром, и освободившаяся память будет передана запросившему память процессу.

Ограничение адресного пространства в 4GB не означает, что система не сможет адресовать более этого объема памяти. На платформе x86 ядро Linux может использовать до 64GB, а ограничение в 4GB накладывается лишь на размер адресного пространства процесса.

System V shared memory

Linux поддерживает стандартную для всех UNIX-подобных операционных систем организацию разделяемой памяти. Пользовательские приложения могут создавать *сегменты* разделяемой памяти, которые могут быть присоединены к некоторому фрагменту адресного пространства

процесса. Любой процесс, имеющий достаточные права доступа, может присоединиться к сегменту разделяемой памяти, и отобразить его в свое адресное пространство, начиная с некоторого адреса.



Если в приведенной схеме любой из процессов изменит содержимое памяти в области, занимаемой отображением одного из сегментов, то же самое изменение произойдет в адресном пространстве другого процесса, поскольку соответствующий сегмент существует в одно экземпляре и отображен в адресное пространство обоих процессов.

Кроме System V IPC ядро Linux также поддерживает другие объекты IPC, в частности семафоры и очереди сообщений. Каждый объект System V IPC идентифицируется уникальным ключем. Просмотреть список всех объектов IPC можно командой *ipcs*. Команда ipcrm позволяет удалять объекты IPC, которые по каким-либо причинам остались неосвобожденными после завершения создавшего их процесса – например, такая ситуация может возникнуть после аварийного завершения работы СУБД Oracle, Informix или DB2.

Соответственно, перед перезапуском процесса системный администратор с помощью команды ipcrm должен освободить неиспользуемые объекты IPC, поскольку стартующее приложение не сможет их повторно создать и не будет корректно работать.

Для каждого объекта IPC система устанавливает права доступа, как если бы это был файл (т.е. для кажого объекта IPC можно устанавливать набор прав ugo/rwx, но в отличие от обычных файлов сменить права доступа для IPC-объектов можно только вызывая специализированные функции, предназначенные для работы с такими объектами.

```
[dalth@viking dalth]$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          oracle    640        4194304    10
0x00000000 32769      oracle    640        20971520   10
0x00000000 65538      oracle    640        29360128   10
0x0d3c24a0 98307      oracle    640        29360128   50
0x00000000 13697028   root      777        49152      1
0x00000000 13729797   root      777        16384      1
0x000004d2 13795334   dalth     666        1008       2
0x00000000 14286866   root      644        790528     2          dest
0x00000000 21823507   dalth     600        393216     2          dest
0x00000000 21921814   root      644        122880     2          dest
0x00000000 14516249   root      644        151552     1          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0b4f657c 262147     oracle    640        154
0x000004d2 458756     dalth     666        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

[dalth@viking dalth]$
```

Поддержка System V IPC позволяет сравнительно легко переносить на Linux приложения, написанные для других UNIX-систем.

Файловая система

VFS и драйверы файловых систем являются одной из важнейших составляющих ядра. Для того, чтобы получить доступ к файлам, хранящимся на каком-либо устройстве хранения данных, необходимо, чтобы в ядро был загружен драйвер соответствующей файловой системы, и файловая система была смонтирована. Драйвера всех файловых систем поддерживают набор стандартных функций: открыть файл по имени, записать данные в файл, прочитать данные из файла, закрыть файл, удалить файл и т.д.

Уточним, что драйвера файловых систем не занимаются кэшированием, этим занимается VFS.

При первоначальной загрузке драйвер файловой системы регистрирует в VFS имя файловой системы и те функции, которые предназначены для выполнения стандартных файловых операций. Впоследствии при обращении к файлу на какой-либо файловой системе VFS будет переадресовывать обращение на соответствующую функцию, если таковая была зарегистрирована драйвером. Посмотреть список обслуживаемых ядром файловых систем можно в файле */proc/filesystems*:

```
[dalth@viking proc]$ cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   sockfs
nodev   usbfs
nodev   usbdevfs
nodev   futexfs
nodev   tmpfs
nodev   pipefs
nodev   eventpollfs
nodev   devpts
        ext2
nodev   ramfs
nodev   hugetlbfs
        iso9660
nodev   devfs
nodev   mqueue
        ext3
nodev   rpc_pipefs
nodev   nfsd
nodev   smbfs
```

Операция монтирования предназначена для того, чтобы сделать доступной файловую систему, расположенную на каком-либо блочном устройстве. Суть операции монтирования заключается в том, что ядро ассоциирует некоторый каталог (называемый точкой монтирования) с блочным устройством и драйвером файловой системы. Для этого оно передает ссылку на блочное устройство драйверу файловой системы, и в случае, если драйвер успешно проидентифицировал эту файловую систему, ядро заносит в специальную таблицу монтирования информацию о том, что все файлы и каталоги, чей полный путь начинается с указанной точки монтирования, обслуживаются соответствующим драйвером файловой системы и расположены на указанном блочном устройстве.

Некоторые файловые системы не нуждаются в блочном устройстве, поскольку хранят свои данные исключительно в памяти, например файловая система *procfs*, через файлы которой можно получить доступ к различным системным параметрам и таблицам.

Очень часто при монтировании файловой системы системный администратор имеет возможность задать *опции монтирования*. Опции монтирования – это специальные параметры, которые влияют на работу драйвера файловой системы, когда он работает с файловой системой на соответствующем блочном устройстве – например, с помощью опций монтирования можно управлять режимом кэширования данных, преобразованиями имен файлов и данных, включать и отключать поддержку ACL и т.д.

Посмотреть таблицу примонтированных файловых систем можно через файл */proc/mounts*:

```
[dalth@viking proc]$ cat /proc/mounts
rootfs / rootfs rw 0 0
/dev/root / ext3 rw 0 0
none /dev devfs rw 0 0
/proc /proc proc rw,nodiratime 0 0
/sys /sys sysfs rw 0 0
none /dev/pts devpts rw 0 0
usbdevfs /proc/bus/usb usbdevfs rw 0 0
/dev/chimera/var /var ext3 rw 0 0
/dev/chimera/temp /tmp ext3 rw 0 0
/dev/chimera/usr /usr ext3 rw 0 0
/dev/chimera/home /home ext3 rw 0 0
/dev/chimera/opt /opt ext3 rw 0 0
none /dev/shm tmpfs rw 0 0
```

Виртуальная файловая система Linux различает несколько типов файлов: каталоги, обычные файлы, именованные каналы, символьные ссылки, сокеты и специальные файлы. Каждая из этих разновидностей обрабатывается своим собственным образом:

- Обычные файлы могут быть прочитаны, записаны, удалены или отображены в память.
- Каталог можно представить как список имен файлов, и для этого списка определены операции добавления элемента в список, удаление элемента из списка, переименование элемента списка.
- Именованные каналы являются просто буферами – в него можно записать некоторый объем данных, и прочесть их в том же порядке, в каком они были записаны.
- Сокеты являются вариантом именованных каналов, но если у именованного канала буфер только один, то есть нельзя определить какой процесс записал данные в канал, то у сокетов может быть несколько клиентов, один из которых осуществляет управление сокетом и может обмениваться данными с любым из остальных клиентов, а те в свою очередь могут обмениваться данными с диспетчером канала – то есть сокет поддерживает множество независимых буферов, по одному на каждую пару сервер+клиент
- Специальные файлы предназначены для того, чтобы осуществлять прямой доступ к устройствам. Подробнее о специальных файлах будет говориться в главе ***Секреты /dev***
- Символьные ссылки являются “ярлыками”, которые могут содержать имя другого файла – и тогда операции чтения, записи и открытия/закрытия файла

автоматически переадресовываются к файлу, на который указывает символьная ссылка, но в отличие от “ярлыков” Windows (shortcuts), символьные ссылки (symbolic links) не требуют специальных функций при работе – все программы (кроме тех, которые специально предназначены для работы с ними) видят их как обычные файлы, и также их открывают, читают и записывают данные и т.д.

В некоторых файловых системах, которые изначально проектировались для UNIX-подобных систем, есть возможность создавать кроме символьных ссылок еще и жесткие ссылки. Фактически, жесткая ссылка – это второе имя для файла. Жесткие ссылки возможно создавать только в пределах одной файловой системы.

Из-за того, что в VFS присутсвует понятие кэширования, перед отключением системы необходимо делать обязательный сброс изменений на диск. Сброс кэша на диск осуществляется в момент размонтирования файловой системы. Кроме того, с помощью команды *sync* можно в любой момент принудительно сбросить на диск все закэшированные изменения в файловой системе (например, системные администраторы часто делают sync перед загрузкой нового драйвера). Размонтировать файловую систему можно только тогда, когда ни одна процесс не удерживает в открытом состоянии файлов с этой файловой системы, а также не находится ни один процесс не имеет рабочим каталогом каталога с размонтируемой файловой системы. При невыполнении этого условия размонтировать файловую систему не удастся:

```
[root@viking dalth]# umount /home/ftp/pub/linux/fedora/cd1
umount: /home/ftp/pub/linux/fedora/cd1: device is busy
umount: /home/ftp/pub/linux/fedora/cd1: device is busy
[root@viking dalth]#
```

Некоторые файловые системы поддерживают специальные опции, позволяющие принудительно синхронизировать файловую систему при каждой операции чтения или записи. Обычно опции, влияющие на синхронизацию файловой системы, содержат в своеме названии слово **sync**, например приведенная ниже команда инструктирует операционную систему примонтировать некоторый раздел в режиме принудительной синхронизации:

```
[root@viking dalth]# mount -t ext3 -o sync,dirsync /dev/hda9 /home
```

Следует учесть, что принудительная синхронизация – это удар по производительности операций записи для файловой системы, смонтированной в таком режиме, поэтому испольовать такой его следует осторожно.

Права доступа

Кроме стандартных наборов прав доступа к файлам некоторые файловые системы Linux

поддерживают т.н. *POSIX ACL* – списки контроля доступа POSIX. Эта возможность позволяет гибко управлять доступом к файлу, не ограничиваясь “класическим” набором ugo/rwx. Для того, чтобы использовать на файловой системе POSIX ACL, необходимо смонтировать файловую систему с опцией *acl*:

```
[root@inferno root]# mount -t ext3 -o acl /dev/inferno/opt /opt
```

Возможно также настроить соответствующий параметр для файловой системы, чтобы она поддерживала POSIX ACL по умолчанию:

```
[root@inferno root]# tune2fs -o acl /dev/inferno/opt
```

После установки соответствующей опции можно приступать к работе с POSIX ACL. Для работы с ними существует две базовых утилиты: *getfacl* для получения списка дополнительных атрибутов доступа, и *setfacl* для установки расширенных атрибутов контроля доступа. Если в выводе команды ls вы видите символ “+” рядом со списком стандартных прав доступа, это означает, что для файла также установлены расширенные атрибуты котнроля доступа:

```
[root@inferno root]# ls -l /home/dalth/.bash_???????
-rw-r-----+ 1 dalth dalth 20034 Окт 11 22:48 /home/dalth/.bash_history
-rw-r--r--  1 dalth dalth   191 Авг 23 21:51 /home/dalth/.bash_profile
[root@inferno root]#
```

Для просмотра значений расширенных атрибутов можно воспользоваться утилитой getfac. Жирным шрифтом выделен дополнительный атрибут контроля доступа, позволяющий пользователю kiki получить доступ на чтение к файлу .bash_history:

```
[root@inferno dalth]# getfacl .bash_history
# file: .bash_history
# owner: dalth
# group: dalth
user::rw-
user:kiki:r--
group::---
mask::r--
other::---
```

Добавим пользователю oracle права на чтение и запись файла .bash_history с помощью команды setfacl, и затем отберем дополнительные права на доступ к указанному файлу у пользователя kiki:

```
[root@inferno dalth]# setfacl -m u:oracle:rw .bash_history
[root@inferno dalth]# setfacl -x u:kiki .bash_history
[root@inferno dalth]# getfacl .bash_history
# file: .bash_history
# owner: dalth
# group: dalth
user::rw-
user:oracle:rw-
group::---
mask::rw-
other::---
```

Последним шагом сбросим все расширенные атрибуты с файла с файла .bash_history:

```
[root@inferno dalth]# setfacl -b .bash_history
[root@inferno dalth]# ls -l .bash_history
-rw-------  1 dalth dalth 20034 Окт 11 22:48 .bash_history
```

Расширенные атрибуты позволяют гибко контролировать доступ к файловых объектам, обходя стратегию ugo/rwx пришедшую из “классического UNIX”. Права доступ на файловые объекты могут быть выданы не только пользователю, но и группе.

К сожалению, далеко не все утилиты и файловые системы поддерживают ACL, поэтому при резервном копировании или восстановлении файлов необходимо проверять корректность установки расширенных атрибутов и правильность их переноса.

Журналируемые файловые системы

Для обеспечения сохранности данных и обеспечения целостности файловых систем при неожиданных сбоях были разработаны *журналируемые файловые системы*. Как правило, у этих файловых систем существует специальная область данных, называемая *журналом*. Все изменения, которые необходимо произвести с файловой системой, сначала записываются в журнал, и уже из журнала попадают в основную часть файловой системы.

В большинстве случаев журналируются только *метаданные* файловых систем (служебная информация, обеспечивающая целостность структуры файловой системы – например, изменения в каталогах или служебных таблицах размещения файлов). Некоторые файловые системы позволяют журналировать не только метаданные, но и данные файлов – такой шаг позволяет повысить надежность, но уменьшает скорость записи данных на файловую систему.

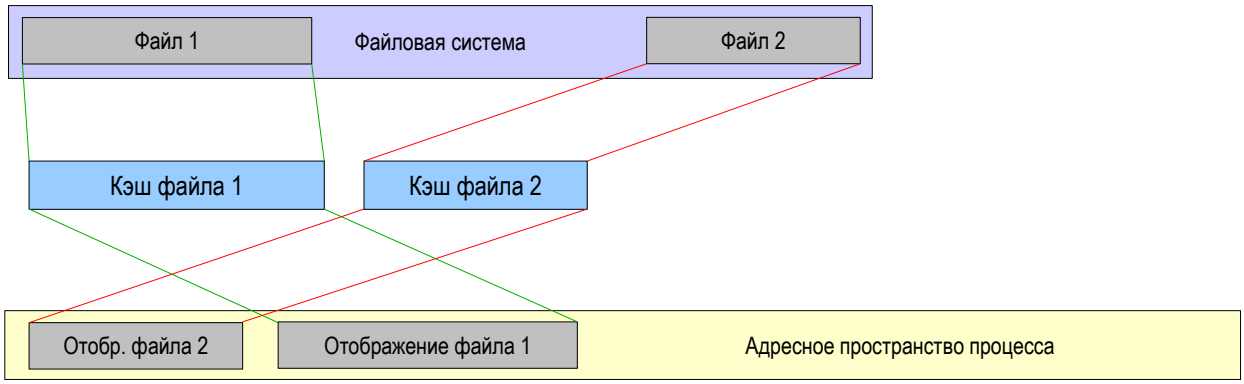
В большинстве случаев, журналируемые файловые системы способны решить проблемы с надежностью при неожиданных сбоях без тех потерь производительности, к которым может привести использование опций **sync** при монтировании.

В частности, к журналируемым файловым системам, например, относятся EXT3, ReiserFS, XFS,

JFS и некоторые другие.

Отображенные в память файлы

Объединение кэширования файлов и разделяемой памяти позволяет реализовать такое действие, как отображение файла в память. Для простоты можно представить, что файл загружается в кэш, и страницы кэша отображаются в адресное пространство процесса, и в результате любое изменение в том фрагменте адресного пространства, которое занято отображением файла, автоматически попадает в закэшированные данные файла. Когда файл закрывается, закэшированные изменения сбрасываются на диск, изменяя сам файл. Кроме того, в свободное время ядро также постепенно сбрасывает изменившиеся кэшированные данные на диск.



> На самом деле, механизм отображения файлов в память куда “хитрее” - при обращении по записи к странице, которая является отображением некоторого файла, ядро перехватывает обращение, производит запись в файл (в подавляющем большинстве случаев эта операция попадает в кэш). При обращении по чтению к такой странице ядро опять же перехватывает обращение и производит чтение из файла – в большинстве случае это чтение производится из кэша. Для наших же целей проще будет считать, что страницы кэша отображены в память процесса.

Такая методика часто используется для загрузки разделяемых библиотек, когда выполняемый код библиотек и исполняемого кода программы хранится в кэше и через отображение файла в память становится виден в адресном пространстве процесса:

```
[dalth@viking dalth]$ cat /proc/1/maps
08048000-08050000 r-xp 00000000 03:01 75813      /sbin/init
08050000-08051000 rw-p 00008000 03:01 75813      /sbin/init
08051000-08072000 rw-p 08051000 00:00 0
40015000-40016000 rw-p 40015000 00:00 0
4c8ee000-4c903000 r-xp 00000000 03:01 92869      /lib/ld-2.3.3.so
4c903000-4c904000 r--p 00014000 03:01 92869      /lib/ld-2.3.3.so
4c904000-4c905000 rw-p 00015000 03:01 92869      /lib/ld-2.3.3.so
4c907000-4ca1c000 r-xp 00000000 03:01 92857      /lib/tls/libc-2.3.3.so
4ca1c000-4ca1e000 r--p 00115000 03:01 92857      /lib/tls/libc-2.3.3.so
4ca1e000-4ca20000 rw-p 00117000 03:01 92857      /lib/tls/libc-2.3.3.so
4ca20000-4ca22000 rw-p 4ca20000 00:00 0
4d201000-4d20f000 r-xp 00000000 03:01 92965      /lib/libselinux.so.1
4d20f000-4d211000 rw-p 0000d000 03:01 92965      /lib/libselinux.so.1
bfffd000-c0000000 rw-p bfffd000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

На самом деле, драйверы любого устройства и любой файловой системы могут по-своему реализовывать операцию mmap, но большинство драйверов файловых систем полагаются в этом на VFS.

Специальные файловые системы

Некоторые типы файловых систем являются специальными и предназначаются для выполнения или реализации специфических задач операционной системы. К таким файловым системам относятся файловые системы *procfs* и *sysfs*, предоставляющие доступ к различным параметрам системы и системным объектам, “живущим” в ядре.

Файловая система sysfs в основном предоставляет доступ к объектам ядра и отображает взаимосвязи между ними. Файловая система procfs предоставляет доступ к различным параметрам ядра и драйверов и к пользовательским процессам, позволяя тем самым реализовать такие команды как ps или sysctl. Большинство файлов в sysfs двоичные, в procfs – текстовые.

Драйверы файловых системы *ramfs*, *tmpfs* и *shmfs* очень похожи, и после монтирования файловой системы такого типа на ней можно создавать файлы, хранящиеся в памяти и отличаются в основном небольшими особенностями работы (например, страницы, используемые ramfs под данные файлов, не вытесняются в swap-файл в отличие от shmfs и tmpfs). В ядре 2.6 shmfs была заменена на tmpfs.

Сетевые файловые системы

Сетевые файловые системы предназначены для получения доступа к файловым системах других компьютеров с использованием сетевых протоколов.

Наиболее часто используются сетевые файловые системы NCPFS (для доступа к серверам Novell NetWare), SMBFS (для доступа к серверам Windows) и NFS (для доступа к файловым

системам других UNIX-систем).

Как правило, процедура монтирования сетевых файловых систем схожа с процедурой монтирования обычных файловых систем на блочных устройствах с тем отличием, что вместо блочного устройства указывается адрес сервера, чья файловая система монтируется, и имя монтируемого ресурса. Для примера рассмотрим процедуры монтирования ресурсов, доступных по SMB и по NFS:

```
# mount -t smbfs -o username=usr,workgroup=tst //server/share_name /mnt/smb_target
Password: ********
# mount -t nfs -o timeout=4 server@/export/home /mnt/nfs_target
```

В данном примере опция *-t* команды *mount* указывает тип файловой системы, опция *-o* позволяет задать дополнительные параметры для монтирования – для SMB мы задаем, например, имя пользователя, с правами которого производится подключение к серверу и имя рабочей группы или домена, для NFS мы указывает таймаут, по истечении которого операция ввода/вывода считается неудавшейся. Вместо блочного устройства мы указываем адрес сервера, ресурс которого хотим использовать, и имя ресурса на сервере. Последним параметром идет точка монтирования.

Создание файловых систем

Для создания файловых систем в Linux используется команда mkfs:

```
    [root@inefrno root]# mkfs -t ext3 /dev/hda6
```

На самом деле, mkfs является просто “оберткой” к реальным программам создания файловых систем, которые обычно именуются как *mkfs.<имя_ФС>*, например mksf.ext2 или mkfs.reiserfs.

В большинстве случаев программы группы mkfs просто инициализируют специальную область раздела, называемую *суперблоком файловой системы*. Суперблок содержит ссылки на все значимые элементы файловой системы (например, ссылку на оглавление корневого каталога, ссылку список свободных блоков, ссылку на список файлов и т.д.)

Наличие суперблока (который практически всегда содержит в своем теле некоторое характерное значение) позволяет производить монтирование файловых систем без указаниях их типа. К сожалению, некоторые файловые системы вследствие своей архитектуры не содержат суперблока (в частности, FAT и большинство ее разновидностей) и для блочных устройств, содержащих такие файловые системы, автоматическое определение типа файловой системы затруднено.

Интерфейс sysctl

Ядро содержит очень много параметров, от которых зависит его производительность и которые могут изменять алгоритмы его работы. Для того, чтобы иметь возможность узнавать и изменять эти параметры, в UNIX был разработан интерфейс sysctl.

Виртуальная файловая система procfs содержит каталог sys с деревом подкаталогов и файлов. Содержимое каждого из этих файлов можно прочесть, например, командой *cat*, или записать в такой файл новое значение командой echo:

```
[root@inferno root]# cat /proc/sys/kernel/shmmax
33554432
[root@inferno root]# echo 67108864 >/proc/sys/kernel/shmmax
[root@inferno root]# cat /proc/sys/kernel/shmmax
67108864
[root@inferno root]#
```

Команда sysctl предназначена для того, чтобы избежать необходимости использовать прямой доступ к этим файлам, и предоставить возможность автоматизации установки таких параметров при загрузке системы. На самом же деле, команда sysctl просто читает или записывает значения в файлы из каталога /proc/sys, т.е. если системный администратор устанавливает с помощью команды sysctl значение некоторого параметра, фактически он просто записывает это значение в соответствующий файл. Существует однозначное соответствие между именем параметра и именем файла, через который его можно изменить: если посмотреть вывод sysctl -a, можно увидеть, что параметры в большинстве своем именуются несколькими мнемоническими аббревиатурами, разделенными точками:

```
[root@inferno root]# sysctl -a | grep mem
net.ipv4.tcp_rmem = 4096        87380   174760
net.ipv4.tcp_wmem = 4096        16384   131072
net.ipv4.tcp_mem = 24576        32768   49152
net.ipv4.igmp_max_memberships = 20
net.core.optmem_max = 10240
net.core.rmem_default = 108544
net.core.wmem_default = 108544
```

Если в имени параметра заменить точки на символ разделителя пути (символ “/”), и к началу получившейся строки добавить **/proc/sys/** - то мы получим имя файла, через который можно изменить или прочесть значение соответствующего параметра.

Если системному администратору необходимо при каждой загрузке изменять некоторые параметры через интерфейс sysctl, то список параметров и их значений можно записать в конфигурационный файл **/etc/sysctl.conf**, который прочитывается при каждой загрузке системы.

Статически и динамически собранные программы

В Linux исполняемые файлы можно условно поделить на две группы – те, которые содержат в себе весь код, необходимые для работы, и те, которым необходимы разделяемые библиотеки. Первые называют статически собранными бинарными файлами, вторые называют динамически собранными исполняемыми файлами.

Статически собранные программы характеризуются тем, что могут корректно функционировать в любых условиях, и не зависят от наличия или отсутствия разделяемых билиотек, что может оказаться полезным в ситуациях, когда возникают конфликты версий разделяемых билиотек, или когда системные билиотеки повреждены или недоступны (например во время восстановления операционной системы после серьезного сбоя). К недостаткам таких исполняемых файлов следует отнести то, что они имеют значительный размер и для обновления программы необходимо полностью заменить ее исполняемый файл – например, если несколько статически собранных программ, которые работают с архивами ZIP, содержат ошибку, то для исправления ошибки необходимо заменить все эти программы, что может быть затруднено (например, будет трудно точно установить, какие именно программы содержат ошибочный код и нуждаются в обновлении). Кроме того, статически собранные программы не умеют совместно использовать совпадающие участки кода, что ведет к излишнему расходу системных ресурсов.

Динамически собранные исполняемые файлы для корректной работы требуют наличия файлов разделяемых библиотек, и соответственно при их отсутствии/повреждении не могут корректно функционировать, но зато для обновления программы и исправления ошибки часто оказывается достаточным просто заменить соответствующую разделяемую библиотеку, после чего ошибка исчезает во всех программах, которые эту библиотеку используют динамически. Динамически связанные программы также значительно меньше по объему, чем статически связанные, и код разделяемых библиотек может использоваться одновременно многими программами – что позволяет экономить системные ресурсы.

Подавляющее большинство программ в современных дистрибутивах Linux являются динамически собранными. Определить тип исполняемого фала (статический ли он либо с динамическим связыванием) можно, например, с помощью команды *ldd*:

```
[dalth@viking dalth]$ ldd /bin/su
        linux-gate.so.1 =>  (0xffffe000)
        libpam.so.0 => /lib/libpam.so.0 (0x4ce08000)
        libpam_misc.so.0 => /lib/libpam_misc.so.0 (0x4cb3c000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x4e3a2000)
        libdl.so.2 => /lib/libdl.so.2 (0x4ca49000)
        libc.so.6 => /lib/tls/libc.so.6 (0x4c907000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x4c8ee000)
[dalth@viking dalth]$ ldd /sbin/devlabel
        not a dynamic executable
[dalth@viking dalth]$
```

В данном случае мы видим, что исполняемый файл /bin/su использует динамическое связывание, а исполняемый файл /sbin/devlabel собран статическим образом.

Системная библиотека GNU libc

Основная системная библиотека, которая так или иначе используется практически всеми программами, называется glibc (GNU libc). Основными задачами glibc являются обеспечение взаимодействия между ядром и пользовательскими процессами, поддержка локализации и многие другие распространенные действия.

На нижнем уровне прикладные процессы могут обратиться к функциям ядра посредством системных вызовов (syscall). Фактически syscall – это вызов прерывания 80h с установленными параметрами, описывающими параметры для этого системного вызова. Те функции glibc, которые должны обратиться к ядру, в большинстве случаев просто устанавливают параметры для соответствующего системного вызова и вызывают int80h.

Большинство программ используют динамически загружаемую библиотеку glibc, но некоторые приложения, которые должны работать вне зависсимости от наличия файловой системы, где расположена динамически загружаемая реализация libc, используют статическое связывание, когда весь программный код, необходимый для их работы, содержится в исполняемом файле программы. В основном к таким программам относятся утилиты, используемые при загрузке системы совместно с initrd – например, статические варианты утилит insmod, lvm или devlabel, а также командные оболочки для первичной зарузки или восстановления системы – например sash – standalone shell, часто используемый при восстановлении системы после серьезного сбоя или nash, используемый при выполнении сценариев загрузки после монтирования initrd, но до монтирования корневой файловой системы, когда разделяемая версия glibc еще недоступна.

LD, Shared Library, SO и много страшных слов

Существует набор базовых действий, которые практически любая программа выполняет одинаково – открытие файла, чтение и запись данных и тому подобное. Разделяемые

библиотеки предназначены для того, чтобы предоставить прикладным программам готовые интерфейсы функций для выполнения каких-либо более-менее стандартных действий. Разделяемая библиотека, как понятно из названия, может использоваться множеством программ. В настоящий момент стандартным форматом для разделяемых библиотек в Linux является ELF (Executable Linked Format).

Каждый файл ELF имеет заголовок, в котором описывается, какие секции содержит этот файл. Секции объединяют однотипные данные, и их детальное описание можно прочитать в справочном руководстве [man elf]. Мы же выделим следующую информацию: каждая библиотека содержит список имен переменных и функций, которые она содержит и предоставляет другим (экспортирует) и список переменных и функций, которые необходимо взять в других библиотеках, а также секции инициализации и деинициализации. Экспортируемые и импортируемые объекты (переменные и функции) называют *символами* библиотеки.

Большинство исполняемых файлов программ также имеют формат ELF, и на самом деле отличаются от библиотек в основном тем, что не имеют экспортируемых функций. Загрузчик ELF (он же dl, dynamic linker и dynamic loader) умеет загружать в память код ELF-файла, анализировать его структуру для определения списков экспортируемых и импортируемых символов и загружать необходимые для работы программы библиотеки.

Когда пользователь пытается запустить какую–либо программу, первым начинает работу загрузчик ELF. Он загружает в память процесса бинарный файл и выделяет, какие символы и из каких библиотек необходимо догрузить в память. После дозагрузки каждой библиотеки загрузчик связывает символы (проставляет реальные адреса) из загруженной библиотеки и повторяет цикл анализа на предмет того, какую библиотеку нужно загрузить. Когда все нужные библиотеки загружены, загрузчик передает управление коду инициализации каждой из загруженных библиотек в порядке, обратном загрузке, после чего передает управление коду программы. По завершении программы загрузчик снова “проходится” по всем библиотекам и вызывает их функции деинициализации. Если на этапе загрузки какой – либо библиотеке возникает ошибка, загрузчик сообщит об этом пользователю. Наиболее типичные ошибки dl – это не найденный файл библиотеки или неразрешимый символ (символ не был найден в библиотеке, в которой ожидался).

Вполне естественно, что загрузчик ищет библиотеки не по всей файловой системе, а только в определенных каталогах. Это каталоги /lib, /usr/lib и те, которые были перечислены системным администратором в файле /etc/ld.so.conf. Уточним, что этот файл на самом деле используется

только системной утилитой ldconfig, сам же загрузчик использует кэш-файл /etc/ld.so.cache. Обновить этот кэш-файл можно путем простого запуска ldconfig без параметров. Следствием этого является то, что если вы установили в систему новые библиотеки, не мешает вызвать ldconfig.

В некоторых дистрибутивах есть возможность включать в ld.so.conf дополнительные файлы без его изменения. Для этого в ld.so.conf включается специальная строка вида:

```
include ld.so.conf.d/*.conf
```

Это приводит к тому, что каталоги, перечисленные в файлах с расширением conf, расположенных в каталоге */etc/ld.so.conf.d* будут использованы для поиска разделяемых библиотек:

```
[root@viking dalth]# cat /etc/ld.so.conf
include ld.so.conf.d/*.conf
/usr/lib/mysql
/usr/X11R6/lib
/usr/lib/qt-3.3/lib
[root@viking dalth]# ls /etc/ld.so.conf.d/
oracle
[root@viking dalth]# cat /etc/ld.so.conf.d/oracle
/opt/oracle/9i/lib
[root@viking dalth]#
```

Нередко возникает ситуация, когда пользователю необходимо запустить какую-либо программу, которая не находится в каталогах, описанных в /etc/ld.so.conf. В таких ситуациях можно воспользоваться специальным “люком”, оставленным разработчиками dl специально для таких случаев: дело в том, что кроме загрузки библиотек с использованием данных из ld.so.cache загрузчик проверяет факт наличия библиотеки с указанным именем в каталогах, перечисленных в переменной среды LD_LIBRARY_PATH.

Разработчики часто используют еще одну возможность ld: если файл некоторой разделяемой библиотеки указан в переменной LD_PRELOAD, эта библиотека принудительно загружается и ее символы считаются более “приоритетными” и перекрывают одноименные символы, если таковые существуют в других библиотеках, загружаемых ld при запуске на выполнение бинарного файла ELF.

Попробуем рассмотреть примеры использования указанных возможностей dl: пусть есть некоторый программный продукт, в состав которого кроме собственно исполняемых программ входят разделяемый библиотеки (например, таковы практически все продукты, разработанные с помощью Borland Kylix). Если мы установим такой пакет, например, в */opt/program*, его исполняемые файлы в */opt/program/bin* а разделяемые библиотеки в */opt/program/lib*, то

программа, скорее всего, не будет запускаться, поскольку не сможет загрузить необходимых библиотек. Для того, чтобы программы пакета начали запускаться, мы должны “объяснить” ld где именно искать библиотеки. Рассмотрим возможные способы, которыми мы можем воздействовать на ld чтобы добиться нужного нам результата.

Первый способ – указать каталог с библиотеками перед запуском программы и уже затем запустить программу (ld воспользуется значением переменной для того, чтобы попытаться найти библиотеки по указанному пути):

```
$ export LD_LIBRARY_PATH=/opt/program/lib
$ /opt/program/bin/filename
```

Второй способ – добавить каталог */opt/program/lib* в файл */etc/ld.so.conf* и запустить *ldconfig*, решив проблему с невозможностью нахождения этих библиотек для всех программ сразу:

```
$ su -
# echo /opt/program/lib >>/etc/ld.so.conf
# ldconfig
# exit
$ /opr/program/bin/filename
```

Можно также воспользоваться возможностью принудительной загрузки тех библиотек, которые необходимы программе для запуска:

```
$ export LD_PRELOAD=/opt/program/lib/*
$ /opr/program/bin/filename
```

Большая часть кода разделяемых библиотек находится в кэше и становится доступна процессам через отображение файла в память. Это отображение делается с правами доступа “только чтение”, что защизает код библиотек от переписывания его неправильно работающими или просто злонамренными программами.

Информация о процессах и файловая систем /proc

Ядро и его подсистемы очень важны, но большинство пользы приносят прикладные задачи, поэтому мониторинг состояния задач (процессов) – очень важная часть работы системного администратора. В Linux получить информацию о процессах можно через файлы и каталоги файловой системы procfs, как правило монтируемой к каталогу /proc.

Каждому процессу сопоставляется в /proc отдельный каталог, имя которого совпадает со значением PID процесса. Файлы в этом каталоге предоставляют информацию о соответствующем процессе. Таблица приводит список файлов и их назначение:

Имя файла	Формат	Назначение
cmdline	строка, разделенная символами \0	Представляет командную строку, которой был запущен процесс. параметры командной строки отделаются друг от друга символами \0
environ	строка, разделенная символами \0	Представляет список переменных окружения для указанного процесса
exe	символьная ссылка	Ссылается на исполняемый файл процесса
maps	несколько строк	Список отображенных в память процесса файлов
mem	бинарный	Прямой доступ к адресному пространству процесса
mounts	несколько строк	Список примонтированных файловых систем, доступных процессу
stat	строка числовых значений	Статистика активности процесса
statm	строка числовых значений	Статистика по использованию памяти процессом
cwd	символьная ссылка	Ссылается на каталог, который является текущим для процесса
fd/*	символьные ссылки	Имена файлов подкаталога fd соответсвуют открытым процессом дескрипторам файлов. Символьные ссылки указывают на соответствующие файлы
root	символьная ссылка	Ссылается на каталог, который процесс считает корневым
status	несколько строк	Описание состояния процесса

Все указанные данные полностью соответствуют тому, что показала бы программа ps, будучи запущеной в тот момент, когда просматривается соответствующий файл, поскольку утилита *ps* на самом деле просто читает данные из соответствующих файлов в /proc.

Создание процессов

Linux на самом деле поддерживает только один внутренний механизм создания процессов – механизм fork+exec. Любой процесс, который хочет создать еще один процесс, должен сначала создать свою копию с помощью системного вызова *fork*, после чего порожденный процесс, который является полной копией предыдущего за исключением нескольких параметров, таких как PID (Process ID) и PPID (Parent Process ID) использует системный вызов exec для того, чтобы загрузить в свое адресное пространство код новой программы и начать его выполнение. Соответственно, все процессы организуют дерево, когда у каждого процесса есть родительский процесс (исключение составляет процесс init, запущеный ядром на этапе загрузки).

Когда процесс завершается, код его завершения возвращается родительскому процессу. До тех пор, пока код завершения процесса не будет прочитан родительским процессом, запись об этом процессе продолжает существовать в таблице процессов в ядре. Такой процесс (уже завершившийся, но еще числящийся в таблице процессов) называют процессом – зомби (zombie process). Завершить zombie process может родительский процесс, прочтя код его завершения.

Если у вас в системе появилось множество зомби – процессов, это скорее всего означает ошибку в программе, породившей этот процесс. Удалить процесс – зомби можно только удалив его родительский процесс.

Нередко бывает, что родительский процесс завершается раньше, чем дочерний, и тогда для дочернего процесса объявляется родительским процесс init, и поэтому в системе никогда не бывает процессов-”сирот”, т.е. тех, кто не имеет родителя и чей код завершения некому прочесть.

Секреты /dev

Ядро Linux реализует поддержку двух типов устройств – символьных и блочных. Основное их отличие в том, что для блочных устройств операции ввода вывода осуществляются не отдельными байтами (символами), а блоками фиксированного размера.

В Linux вся работа с устройствами ведется через специальные файлы, которые обычно расположены в каталоге /dev. Специальные файлы не содержат данных, а просто служат точками, через которые можно обратиться к драйверу соответствующего устройства. У каждого специального файла есть три характеристики – тип устройства (character или block), старший номер устройства (major number) и младший номер (minor number). Для примера, посмотрим на содержимое каталога /dev:

```
[dalth@viking proc]$ ls -lL /dev/hd* /dev/ttyS*
brw-------  1 root  root  3,  0 Окт  1 20:16 /dev/hda
brw-------  1 root  root  3,  1 Окт  1 20:16 /dev/hda1
brw-------  1 root  root  3,  2 Окт  1 20:16 /dev/hda2
brw-------  1 dalth disk 22,  0 Янв  1  1970 /dev/hdc
crw-------  1 root  root  4, 64 Янв  1  1970 /dev/ttyS0
crw-------  1 root  root  4, 65 Янв  1  1970 /dev/ttyS1
crw-------  1 root  root  4, 66 Янв  1  1970 /dev/ttyS2
```

Как видно, в листинге присутствует описание семи устройств, четырех блочных и трех символьных. Для каждого файла можно увидеть его тип (первая буква в списке прав доступа), пользователя-владельца, группу-владельца, major number, minor number, дату модификации и имя файла.

Для поддержки работы с устройствами в ядре хранятся две таблицы, одна для списка символьных устройств, другая для списка блочных устройств. Каждая строка таблицы сопоставлена какой-то разновидности устройств соответствующего типа – например, для типа “символьные устройства” можно выделить следующие разновидности: COM-порты, LPT-порты, PS/2-мыши, USB-мыши и т.д., для типа “блочные устройства” можно выделить SCSI-диски, IDE-диски, SCSI-CD-приводы, виртуальные диски которыми представляются RAID-контролеры

и т.п.

Каждая ячейка в этих системных таблицах сопоставляется конкретному *экземпляру устройства*. Таким образом, с точки зрения ядра каждое устройство оказывается однозначно проидентифицировано тремя параметрами – типом устройства (блочное или символьное) и двумя числами – номерами строки и номером столбца таблицы, в которой хранится ссылка на драйвер этого устройства.

Пример таблицы символьных и устройств

	0	**1**	**...**	**63**	**64**	**65**	**66**	**...**	**175**	**...**
4					COM1	COM2	COM3			
6	LPT1	LPT2								
10		Мышь PS/2		Диспетчер томов LVM					Слот AGP	
14	Микшер первой зв. карты									
195	Первая видеокарта NVidia									

Пример таблицы блочных устройств

	0	**1**	**2**	**...**	**16**	**...**	**64**	**65**	**...**
3	IDE Primary Master	Раздел 1 на IDE Primary Master	Раздел 2 на IDE Primary Master		Раздел 16 на IDE Primary Master		IDE Primary Slave	Раздел 1 на IDE Primary Slave	
13	SCSI диск 1	Раздел 1 на SCSI-диске 1	Раздел 2 на SCSI-диске 1		SCSI диск 2		SCSI диск 4	Раздел 1 на SCSI-диске 4	

При попытке обращения к такому специальному файлу ядро переадресует обращение через нужный драйвер на устройство в соответствии с теми данными, которые указаны в таблице устройств, причем конкретная таблица устройств будет выбрана в зависимости от типа устройства, строка из таблицы будет выбрана по major number, и столбец будет выбран по minor number. Если мы посмотрим на примеры наших таблиц, то увидим, что обращение на специальный файл /dev/ttyS1, который представляет символьное устройство со старшим номером 4 и младшим номером 65 будет адресовано на последовательный порт COM2, а обращение к файлу /dev/hda2 (блочное устройство со старшим номером 3 и младшим номером 2) будет адресовано на 2-й раздел жесткого диска IDE, работающего в режиме primary master.

В настоящее время существует два подхода к организации /dev – статическая организация и динамическая организация. В первом случае в каталоге /dev заранее создаются специальные файлы для всех возможных устройств вне зависимости от того, загружен драйвер соответствующего устройства или нет. Во втором случае специальные файлы в /dev создаются по мере инициализации устройств и загрузки драйверов, и удаляются при выгрузке соответствующего драйвера или удалении устройства.

Процесс работы со статическим /dev особых проблем не вызывает – системный администратор при необходимости просто создает отсутствующие файлы командой ***mknod*** или ***MAKEDEV***. В том случае, когда какая-либо программа обращается к устройству, чей драйвер не загружен (или загружен, но ни одного соответствующего устройства не было обнаружено), операционная система возвращает ошибку при попытке открытия файла такого "неверного" устройства. Ниже приведен пример создания специального файла, соответствующего блочному устройству с мажором 8 и минором 33 и попытка его использования (отметим, что этот специальный файл соответствует разделу на жестком диске, который не существует на тестовой машине, где выполнялись эти команды):

```
[root@viking root]# cd /dev
[root@viking dev]# ls -l /dev/hda33
ls: /dev/hda33: No such file or directory
[root@viking dev]#
[root@viking dev]# mknod hda33 b 8 33
[root@viking dev]#
[root@viking dev]# ls -l hda33
brw-r--r--  1 root root 8, 33 Окт 11 09:27 hda33
[root@viking dev]#
[root@viking dev]# dd if=hda33 of=/dev/null
dd: opening `hda33': No such device or address
[root@viking dev]#
```

Сообщение **No such device or address** как раз и означает, что записи для данного устройства в таблице блочных устройств не существует.

Ядро Linux совместно с некоторыми системными утилитами поддерживает такую интересную возможность, как загрузка драйверов "по требованию". Реализуется это следующим образом – в момент, когда какая-либо программа пытается открыть специальный файл, не связанный ни с каким драйвером, ядро делает попытку подобрать соответствующий драйвер самостоятельно. Необходимый драйвер для каждого специального файла определяется в файле /etc/modules.conf путем задания специального алиаса (alias) для модуля. Для активизации автоматической загрузки драйвера какого-либо символьного устройства в большинстве случаев достаточно просто записать в /etc/modules.conf строку следующего вида:

```
alias char-major-X-Y имя_драйвера
```

Для блочных устройств соответствующая запись слегка меняет свою форму:

```
alias block-major-X-Y имя_драйвера
```

X и Y – это major и minor специального файла, попытка открыть который должна активизировать автоматическую загрузку драйвера. Владельцы видеокарт на чипе nVidia могут

увидеть этот подход в действии – программа инсталляции драйвера nVidia автоматически прописывает в modules.conf запись для загрузки по требования той части драйвера, которая работает в режиме ядра.

Вместо X или Y может также быть подставлен символ “*” , означающий “любое число”. Например, пусть в modules.conf будет написана следующий текст:

```
alias char-major-81-* bttv
```

Тогда при обращении к любому символьному устройству с major number равным 81 и которое не ассоциировано ни с каким драйвером, система попытается загрузить драйвер bttv (драйвер TV-тюнера на основе чипа bt848).

Эта возможность обеспечивает Linux возможность плавной загрузки и эффективного использования ресурсов – драйвер не загружается, пока в нем не возникнет необходимости. К сожалению, за простоту этой схемы приходится платить большим количеством специальных файлов в /dev.

Для того, чтобы избавить администратора от ручного создания специальных файлов и для уменьшения количества файлов в /dev был реализован второй способ организации /dev – динамическое создание специальных файлов процессе загрузки драйверов. Реализовано это было следующим образом:

Ядро монтирует к каталогу /dev специальную файловую систему, называемую ***devfs*** – эта файловая система хранится целиком в оперативной памяти и не занимает никакого места на диске. Когда какой-либо драйвер в процессе загрузки или работы обнаруживает обслуживаемое им устройство, он регистрирует это устройство и сообщает о нем драйверу devfs. Драйвер devfs создает специальный файл, который виден прикладным программам и может быть корректно открыт. При выгрузке же драйвер устройства сообщает devfs о том, что соответствующее устройство уже не активно, и драйвер devfs удаляет запись о соответствующем специальном файле из файловой системы devfs.

Файловая система devfs отличается тем, что как правило специальный файл для устройства создается с длинным путем – например, для раздела на scsi-диске путь может выглядеть примерно так: ***/dev/scsi/host1/bus1/target3/lun4/partition2***

Эта особенность является весьма важным плюсом devfs, поскольку она позволяет адресовать дисковые устройства путем указания логического пути их подключения и избежать смены имен SCSI-дисков в некоторых случаях (об этих случаях будет рассказано позднее).

Для того, чтобы организовать более прозрачную структуру каталогов и файлов устройств, используется специальный демон ***devfsd***. Он взаимодействует с драйвером devfs и ядром и в процессе активизации и деактивизации устройств он создает и удаляет символьные ссылки вида /dev/disks/disc0 или /dev/hda1.

Надо отметить, что схема динамического /dev в некотором смысле близка к той организации кталога /dev, которая используется некоторыми коммерческими UNIX-системами (например, в Solaris), когда есть виртуальная файловая система /devices, и на ее файлы создаются ссылки из /dev, только в Linux роль программы cfgadm играет демон devfsd, и все изменения в состав /dev вносятся автоматически.

С помощью devfsd файловая система devfs также реализует автоматическую загрузку модулей, но в этом случае выбор модуля идет не через комбинацию type/major/minor, а путем указания имени запрошенного файла – когда приложение пытается открыть несуществующий файл устройства, devfs передает имя запрошенного файла демону devfsd, и последний загружает необходимые модули, например такой код в файле modules.devfs:

```
    alias /dev/nvidia* nvidia
```

Приведет к тому, что при попытке обращения к любому файлу, чей полный путь начинается строкой */dev/nvidia*, будет произведена попытка загрузить драйвер *nvidia.o* (для ядра 2.6 *nvidia.ko*)

В принципе, на сегодняшний день выбор того, каким именно образом необходимо организовывать /dev, остается за пользователем и создателем дистрибутива. Например, в Mandrake Linux используется devfs, а в RedHat, Fedora и SUSE каталог /dev организован статическим образом, а опытные пользователи часто меняют способ организации /dev в зависимости от своих предпочтений.

В современных дистрибутивах и ядрах поддержка devfs/devfsd отключена, и на смену этой паре пришел специальный демон, называемый udev. В отличие от devsfd, который требовал поддержки со стороны ядра, udev такой поддержки не требует. При инициализации устройства ядро подает сигнал через файловую систему sysfs, и udev, получив сигнал об этом событии, самостоятельно создает соответствующий специальный файл устройства в каталоге /dev.

Блочные устройства

Любое устройство, подключенное к компьютеру, имеет свое назначение, и блочные устройства в большинстве своем предназначаются для хранения информации. Как организована работа с блочными устройствами в Linux?

Во-первых, следует определиться с типами блочных устройств. Их следует поделить на две категории: к первой отнесем логические (виртуальные) устройства (loop-устройства, software RAID-устройства, устройства Volume Management, поддержка различных таблиц разделов), ко второй категории - физические устройства (SCSI диски и CD-ROM'ы, IDE-диски, USB-storage, RAM-диск).

Виртуальные устройства являются на самом деле просто оберткой, дополнительным слоем. В реальности драйверы логических устройств не работают с периферийными устройствами напрямую, они лишь переадресовывают запросы на драйверы других логических или физических устройств.

Драйверы физических устройств работают совместно с драйверами контроллеров, позволяя производить доступ к соответствующим устройствам на блочном уровне и предоставляя тем самым фактически прямой доступ к носителю – но, поскольку большинстве случаев дисковые устройства имеют значительный объем, они часто делятся на *разделы*. Раздел является постоянным непрерывным фрагментом дискового пространства, местоположение которого на жестком диске записано в специальной области диска – таблице разделов.

Существует множество различных форматов разбиения диска на разделы – например, DOS partition table, BSD disklabels, UnixWare slices и многие другие. Как правило, во всех случаях соответствующая спецификация предусматривает возможность перечисления ограниченного количества разделов путем указания номеров первой и последней дорожек, занимаемых каждым из разделов. Каждый раздел видится как отдельное блочное устройство.

По традиции имена блочных устройств, соответствующих IDE-дискам и созданым на них разделам начинаются с *hd* и имеют вид **/dev/hd<N>[<M>]** где **N** – это буква, зависящая от контроллера и канала IDE, к которому подключено устройство, и режима устройства (master/slave). **M** – это некоторое число от 1 до 63 (фактически номер раздела на диске). Если число не указано, подразумевается весь диск. SCSI-дискам в /dev присваиваются имена sda, sdb, sdc и т.д. Ниже приводится небольшая таблица соответствия устройств и имен специальных файлов для IDE-дисков:

Контроллер	Канал	Режим		Имя файла
1	1	master	primary master	hda
1	1	slave	primary slave	hdb
1	2	master	secondary master	hdc
1	2	slave	secondary slave	hdd
2	1	master	tertiary master	hde
2	1	slave	tertiary slave	hdf
2	2	master	quaternary master	hdg
2	2	slave	quaternary slave	hdh

Когда драйвер блочного устройства, поддерживающего разбиение на разделы, в процессе загрузки или работы обнаруживает обслуживаемое им устройство, он считывает с него таблицу разделов, определяет ее разновидность и составляет таблицу разделов, запоминая начало и конец каждого из разделов. Впоследствии, если какая-либо программа производит обращение не непосредственно к физическому устройству, а к разделу на этом устройстве, драйвер с использованием построенной таблицы разделов определяет реальный адрес блока, над которым нужно произвести запрошенную операцию ввода/вывода.

Особой разновидностью раздела можно назвать расширенный (*extended*) раздел DOS. Расширенный разделы DOS может быть разбит на произвольное количество вложенных разделов, но в настоящий момент без использования LVM ядро Linux поддерживает до 63 разделов на IDE-диске и до 15 разделов на SCSI-диске. Такое ограничение связано с распределением мажоров и миноров блочных устройств.

Когда используется разбиение диска на разделы с использованием таблицы разделов DOS, следует помнить, что на жестком диске может быть не более 4 первичных разделов. Если администратору нужно, чтобы на жестком диске было более 4 разделов, необходимо объявить один из первичных разделов как расширенный раздел. Первичные разделы при использовании таблицы разделов DOS нумеруются от 1 до 4, логические разделы нумеруются начиная с 5, вне зависимости от количества первичных разделов. Только один из первичных разделов может быть объявлен расширенным.

Рассмотрим вывод команды fdisk, которая обычно используется в Linux для разбиения диска на разделы:

```
[root@stend root]# fdisk /dev/hda -l

Disk /dev/hda: 6442 MB, 6442450944 bytes
16 heads, 63 sectors/track, 12483 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1   *           1         203      102280+  83  Linux
/dev/hda2             204        2032      921816   83  Linux
/dev/hda3            2033        3072      524160   82  Linux swap
/dev/hda4            3073       12483     4743144    f  W95 Ext'd (LBA)
/dev/hda5            3073        4088      512032+  83  Linux
/dev/hda6            4089       12483     4231048+  83  Linux

[root@stend root]#
```

Как видно, на жестком диске IDE созданы 6 разделов, из них 4 первичных (разделы с номерами от 1 до 4) и два логических раздела с номерами 5 и 6, созданых внутри extended-раздела hda4.

Таблица разделов диска не может быть изменена в том случае, если хотя-бы один из разделов этого диска используется. В этом случае ядро продолжает использовать “старую” разметку (с которой оно работало до изменения), а изменения записываются на диск и вступают в силу после перезагрузки компьютера.

Если нет возможности запустить программу fdisk, то для получения данных о разметке блочных устройств на разделы и о состоянии блочных устройств, доступных в настоящий момент, можно использовать некоторые файлы из файловой системы /proc:

```
[dalth@viking dalth]$ cat /proc/partitions
major minor  #blocks  name

   3     0   78150744 hda
   3     1    1084356 hda1
   3     2   77063805 hda2
 253     0    1048576 dm-0
 253     1    1048576 dm-1
 253     2   10485760 dm-2
 253     3   10485760 dm-3
 253     4    1048576 dm-4
 253     5   41943040 dm-5
 253     6     360448 dm-6
 253     7     258048 dm-7
 253     8     258048 dm-8
 253     9     258048 dm-9
 253    10      53248 dm-10
   7     0     651884 loop0
   7     1     650198 loop1
   7     2     653336 loop2
   7     3     198962 loop3
```

Например, эти данные могут быть использованы для восстановления таблицы разделов, если системный администратор по ошибке ее исправил.

Интересной особенностью таблицы разделов диска является то, что она не всегда изменяема. Это приводит к тому, что во многих случаях невозможно изменить размер какого-либо раздела или внести какие-либо другие изменения в таблицу разделов без перезагрузки – т.е. системный администратор может делать любые изменения, но они вступят в силу только после перезагрузки системы.

Распределение мажоров и миноров IDE дисков

IDE-диски в настоящее время наиболее часто используются в оффисных и домашних компьютерах, поэтому знать особенности распределения мажоров и миноров для этих типов устройств достаточно важно. IDE-устройства отличаются низкой ценой и неплохой скоростью предачи данных, но у это шины есть архитектурные недостатки – например, все устройства работают со скоростью самого медленного из них, на один шлейф (на один канал) можно подключить только 2 устройства.

Всем устройствам, находщимся на одном канале IDE присвоен один мажор. В настоящий момент ядро Linux выделяет для каждого устройства 64 минора, из которых первый минор зарезервирован для всего диска, и 63 минора остается для идентификации разделов. Таким образом, для IDE-дисков мажор идентифицирует канал, а по минору можно определить номер устройства на канале (режим master/slave) и номер раздела. Более детально это можно увидеть в следующей таблице:

Распределение номеров устройств для IDE-дисков					
Канал	**Устройство**	**Раздел**	**Major number**	**Minor number**	**Имя в /dev**
1	1	Весь диск	3	0	/dev/hda
		Раздел 1		1	/dev/hda1
		Раздел 2		2	/dev/hda2
		Раздел 3		3	/dev/hda3
		Раздел 4		4	/dev/hda4
		...			
		Раздел 63		63	/dev/hda63
	2	Весь диск		64	/dev/hdb
		Раздел 1		65	/dev/hdb1
		Раздел 2		66	/dev/hdb2
		Раздел 3		67	/dev/hdb3
		...			
		Раздел 63		127	/dev/hdb63
2	1	Весь диск	22	0	/dev/hdc
		Разделы		1-63	/dev/hdc[1..63]
	2	Весь диск		64	/dev/hdd
		Разделы		65-127	/dev/hdd[1..63]

По таблице становится видно, что на каждый 64-й минор происходит смена физического диска. Драйвер IDE в текущей версии ядра Linux поддерживает до четырех каналов IDE – т.е. до 8 устройств, по два устройства на канал..

Рекомендации, которую можно дать владельцам компьютеров с интерфейсом IDE: устройства по возможности рекомендуется держать на разных каналах. Если нет свободных каналов, то “быстрые” устройства лучше подключать на один канал, медленные – на другой.

Распределение мажоров и миноров для SCSI-дисков

Она свободна от некоторых недостатков IDE, напрмиер количество устройств на одном канале может быть 15 (на самомо деле 16, но одним устройством считается сам контроллер), все SCSI устройства работают на своей максимальной скорости, и ограничены только возможностями шины и контроллера – но SCSI-устройства и дороже, и поэтому шина SCSI используется в основном на серверах и рабочих станциях. Для SCSI-дисков ситуация немного меняется – мажоры не привязаны к контроллерам (т.е. один major number может использоваться дисками с разных хост-адаптеров). На каждом SCSI-диске система поддерживает до 16 разделов, а нумерация дисков производится в порядке их подключения по схеме, аналогичной IDE-дискам – но только переход на следующий диск происходит на каждом 16-м миноре (т.е. разделы на SCSI-дисках нумеруются от1 до 15). Увидеть это можно в следующей таблице и листинге /dev:

Распределение номеров устройств для SCSI-дисков				
Номер диска в порядке подключения	**Major number**	**Minor number**	**Раздел**	**Имя файла**
1	8	0	Весь диск	sda
		1	Первый первичный	sda1
		2	Второй первичный	sda2
		3	Третий первичный	sda3
		4	Четвертый первичный	sda4
		5	Первый логический	sda5
		...		
		15	Одиннадцатый логический	sda15
2	8	16	Весь диск	sdb
		17	Первый первичный	sdb1
		18	Второй первичный	sdb2
		19	Третий первичный	sdb3
		20	Четвертый первичный	sdb4
		21	Первый логический	sdb5
		...		
		31	Одиннадцатый логический	sdb15

В текущей версии ядро Linux поддерживает до 4096 SCSI-дисков – и для абсолютного большинства компьютеров этого должно быть достаточно. Очень важной особенностью SCSI-дисков является то, что мажор устройства, соответствующего физическому диску, не зависит от контроллера. В результате, если у вас есть 3 диска SCSI, то они всегда именуются sda, sdb и sdc, и если вы выключите компьютер и отключите первый диск, то после перезагрузки второй диск (который ранее назывался sdb) станет называться sda, а третий диск (который назывался sdc) станет называться sdb, поэтому при работе со SCSI-устройствами можно использовать devfs (которая позволяет адресовать диски через путь их подключения), либо использовать специализированные средства управления дисковым пространством, которые помечают носители и впоследствии правильно их идентифицируют даже после переименования устройств - например, средства md (software RAID) или LVM. Надо заметить, что в свете удаления devfs из основной ветви ядра, использование LVM стало фактически обязательным на серверах.

Logical Volume Manager

Использование таблиц разделов для управления дисковым пространством – достаточно часто используемое решение. К сожалению, оно не свободно от определенных недостатков – например, нет возможности расширить раздел или уменьшить его размер, нет возможности создать один раздела на нескольких дисках и т.д. Решить эту задачу призван LVM (Logical

Volume Manager).

LVM работает следующим образом: пользователь может пометить какие-либо блочные устройства как разделы, используемые LVM. Каждое из таких помеченых блочных устройств (их называют физическими томам,и или physical volumes) может быть присоединено к какой либо группе логических томов (logical volume groups). Внутри групп логических томов могут создаваться уже собственно логические тома (logical volumes). Дисковое пространство любого физического тома из некоторой группы может быть выделено для любому логическому тому из этой группы. Реализовано это через так называемые “экстенты” (extents) дискового пространства. Физические тома LVM разбиваются на экстенты, после чего из экстентов и составляются логические тома. Именно за счет этого можно динамически менять конфигурацию дискового пространства – экстент может быть удален из одного тома, и добавлен к другому. Каждый объект LVM – будь то логический том, физический том или группа томов, имеет свой уникальный идентификатор.

Осуществляется это комплексно драйвером *device mapper* и специализированными программами из пакета lvm2. Эти программы читают файлы конфигурации и служебную информацию из заголовков физических томов, и на основании этой информации сообщают драйверу инструкции о том, из каких фрагментов каких блочных устройств каким именно образом должны быть скомбинированы логические тома, после чего драйвер для каждого логического тома создает отдельное блочное устройство. При обращении к такому блочному устройству device mapper определяет, на основании ранее переданных параметров к какому блоку какого блочного устройства на самом деле должен быть переадресован запрос, и запрашивает соответствующее блочное устройство для окончательного выполнения операции, после чего возвращает результат выполнения операции (прочитанные данные, сообщение об ошибке, код завершения операции) обратившейся программе.

Использование LVM позволяет гибко управлять распределением дискового пространства и избежать ограничений, связанных с классическим распределением дискового пространства путем создания разделов на жестких дисках. Единственное правило, которое я бы советовал соблюдать при использовании LVM – не создавать корневой раздел системы на логическом томе LVM: инициализации тома LVM, на котором находится корневая файловая система, необходимо вмешательство некоторых утилит, которые находятся на еще не смонтированной корневой файловой системы. Это решаемая проблема, но она потребует некоторого опыта.

Ниже идет пример создания и инициализации физического тома, группы томов и примеры нескольких операций с логическими томами. На первом фрагменте протокола

продемонстрирована инициализация таблицы разделов для использования LVM. Порядок действий для использования LVM в общем случае следующий: один или несколько разделов жесткого диска с помощью fdisk помечаются как разделы LVM. Затем эти разделы инициализируются и передаются в группы томов, после чего их дисковое пространство можно использовать для создания логических томов:

```
[root@inferno dalth]# fdisk /dev/hdb

The number of cylinders for this disk is set to 79408.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/hdb: 40.9 GB, 40982151168 bytes
16 heads, 63 sectors/track, 79408 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/hdb1               1       79408    40021600+  8e  Linux LVM

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
[root@inferno dalth]#
```

В приведенном выше выводе fdisk видно, что на IDE-диске primary slave создан один раздел типа LVM (код типа раздела 0x8E). В следующем листинге показан процесс инициализации физического тома и создания группы томов **aurora**, в которую включается инициализированный командой *pvcreate* физический том /dev/hdb1:

```
[root@inferno dalth]# pvcreate /dev/hdb1
  No physical volume label read from /dev/hdb1
  Physical volume "/dev/hdb1" successfully created
[root@inferno dalth]# vgcreate aurora /dev/hdb1
  Volume group "aurora" successfully created
[root@inferno dalth]#
```

Третий фрагмент демонстрирует создание нескольких томов и изменение размеров томов, проделанное с помощью LVM. В этом примере создается логический том размеров в 20GB, затем размер этого тома увеличивается до 30GB, создается еще один логический том, и после этого оба созданных логических тома удаляются.

```
[root@inferno dalth]#
[root@inferno dalth]# lvcreate -L 20G -n ftpdata aurora
  Logical volume "ftpdata" created
[root@inferno dalth]#
[root@inferno dalth]# lvscan
  ACTIVE            '/dev/aurora/ftpdata' [20,00 GB] next free (default)
[root@inferno dalth]#
[root@inferno dalth]# lvresize -L +10G /dev/aurora/ftpdata
  Extending logical volume ftpdata to 30,00 GB
  Logical volume ftpdata successfully resized
[root@inferno dalth]#
[root@inferno dalth]# lvscan
  ACTIVE            '/dev/aurora/ftpdata' [30,00 GB] next free (default)
[root@inferno dalth]# lvcreate -L 8G -n home_dirs aurora
  Logical volume "home_dirs" created
[root@inferno dalth]#
[root@inferno dalth]# lvscan
  ACTIVE            '/dev/aurora/ftpdata' [30,00 GB] next free (default)
  ACTIVE            '/dev/aurora/home_dirs' [8,00 GB] next free (default)
[root@inferno dalth]#
[root@inferno dalth]# lvremove /dev/aurora/ftpdata
Do you really want to remove active logical volume "ftpdata"? [y/n]: y
  Logical volume "ftpdata" successfully removed
[root@inferno dalth]#
[root@inferno dalth]# lvremove /dev/aurora/home_dirs
Do you really want to remove active logical volume "home_dirs"? [y/n]: y
  Logical volume "home_dirs" successfully removed
[root@inferno dalth]#
```

Еще один фрагмент демонстрирует удаление группы томов и очистку физического тома:

```
[root@inferno dalth]# vgremove aurora
  Volume group "aurora" successfully removed
[root@inferno dalth]# pvremove /dev/hdb1
  Labels on physical volume "/dev/hdb1" successfully wiped
[root@inferno dalth]#
```

Каждый логический том LVM имеет свой собственный minor number, a major number для всех томов LVM равен 253. Для доступа к томам LVM можно создавать блочные устройства с помощью команды *mknod*, а можно воспользоваться возможностями, предоставляемыми утилитой devlabel. Эта утилита создает символьные ссылки и каталоги внутри подкаталога /dev, причем для каждой группы томов в /dev создается каталог с именем этой группы, а логические тома представляются символьными ссылками из этих каталогов на блочные устройства, обслуживаемые драйвером device mapper, и тогда любой том LVM можно адресовать следующим путем: */dev/<имя_группы>/<имя_тома>*. На листинге ниже показан пример того, как можно распределить дисковое пространство с помощью LVM:

```
[root@viking root]#
[root@viking root]# ls -la /dev/chimera
lr-xr-xr-x  1 root root 23 Окт  8 13:53 opt -> /dev/mapper/chimera-opt
lr-xr-xr-x  1 root root 24 Окт  8 13:53 swap -> /dev/mapper/chimera-swap
lr-xr-xr-x  1 root root 24 Окт  8 13:53 temp -> /dev/mapper/chimera-temp
lr-xr-xr-x  1 root root 23 Окт  8 13:53 usr -> /dev/mapper/chimera-usr
lr-xr-xr-x  1 root root 23 Окт  8 13:53 var -> /dev/mapper/chimera-var
[root@viking root]#
[root@viking root]# lvscan
  ACTIVE          '/dev/chimera/swap' [1,00 GB] next free (default)
  ACTIVE          '/dev/chimera/temp' [1,00 GB] next free (default)
  ACTIVE          '/dev/chimera/usr' [10,00 GB] next free (default)
  ACTIVE          '/dev/chimera/opt' [10,00 GB] next free (default)
  ACTIVE          '/dev/chimera/var' [1,00 GB] next free (default)
[root@viking root]#
[root@viking root]# mount | grep chimera
/dev/mapper/chimera-var on /var type ext3 (rw)
/dev/mapper/chimera-temp on /tmp type ext3 (rw)
/dev/mapper/chimera-usr on /usr type ext3 (rw)
/dev/mapper/chimera-opt on /opt type ext3 (rw)
[root@viking root]#
```

Таким образом, возможности LVM позволяют системному администратору максимально эффективно использовать дисковое пространство, оперативно реагируя на меняющиеся условия эксплуатации. Еще одной интересной возможностью LVM является так называемый multipath I/O. В случае активации соответствующей опции в ядре device mapper знает о том, что физический том с некоторым UUID может быть доступен через несколько контроллеров, и в случае отказа одного контроллера динамически происходит переключение ввода-вывода на другой. Опытные системные администраторы также оценят такую возможность, как создание снимка (snapshot) логического тома: при создании снимка создается моментальная копия логического тома, которая начинает «жить» независимо от того тома, на основе которого она была создана:

```
# xfs_freeze /home
# lvcreate -s -L 10G -n home_snapshot /dev/chimera/home
# xfs_freeze -u /home
# dd if=/dev/chimera/home_snapshot of=/dev/st0
# lvremove /dev/chimera/home_snapshot
```

В приведенном примере системный администратор «замораживает» файловую систему XFS, при этом драйвер XFS сбрасывает все закэшированные операции на диск, и после этого блокирует все процессы, которые пытаются писать на «замороженную» файловую систему. Затем системный администратор создает снимок тома home из группы томов chimera, на котором «живет» файловая система /home, и этот снимок называет home_snapshot, при этом на удержание копии измененных данных выделяется 10 гигабайт дискового пространства. После создания снимка файловая система /home размораживается, но каждый раз, когда будет

переписываться какой-либо блок логического тома home, первоначальная версия изменяемого блока будет копироваться в те 10GB дискового пространства, которые мы выделили под снимок тома, и мы можем считать содержимое тома home_snapshot неизменным, и скопировать его на ленту. В прцессе чтения, если читаемый блок не изменялся с момента создания снимка, то он читается из исходного тома (home), если же блок менялся с момента создания снимка, то используется его копия, хранимая в зарезервированном при создании снимка пространстве. После окончания копирования мы удаляем снимок командой lvremove.

Sotware RAID

Ядро Linux содержит средства для организации software raid (программных RAID-устройств). Эта возможность поддерживается драйвером устройств *md*. В отличие от device mapper, драйвер md умеет работать в “самостоятельном” режиме, получая конфигурацию из параметров, которые пользователь указал ядру при загрузке системы, что позволяет организовывать загрузку системы с RAID-устройств. Все устройства md имеют мажор 254 и миноры от 0 и до 16383.

В отличие от LVM, основной задачей которого является динамическое распределение дискового пространства (деление разделов на фрагменты и построение из фрагментов новых блочных устройств), задачей подсистемы RAID является построение новых блочных устройств путем объединения существующих.

Каждое из устройств, входящих в создаваемый дисковый массив, может определенным образом помечаться. Впоследствии эти метки (их также называют *array superblocks*) используются для повторной сборки массива. В частности, например, суперблок массива содержит его уникальный идентификатор, который можно использовать при сборке ранее созданного массива после перезагрузки. Если программный RAID-массив был помечен в процессе создания (т.е. на нем был создан суперблок массива), это дает возможность автоматической сборки массива вне зависимости от того, поменялся или нет порядок следования устройств. Например, такая необходимость может возникнуть в ситуации, когда порядок нумерации блочных устройств изменился – например, один из SCSI-дисков, участвовавших в построении массива, был удален.

Естественно, суперблок не является обязательным – то есть можно создавать массивы без суперблока, но управления ими может быть затруднено вследствие необходимости «руками» контролировать корректность указания устройств при переконфигурации массива.

Из интересных особенностей драйвера *md* стоит отметить то, что он поддерживает разбиение md-устройств на разделы, при этом минорные номера присваиваются разделам аналогично

тому, как они присваиваются разделам на дисках IDE, т.е на каждом RAID-устройстве можно создать до 63 разделов. Определить минор раздела, созданного на RAID-устройстве, можно с помощью вычисления значения следующего выражения: 64 * N + M, где N – это номер массива (номер RAID-устройства) из диапазона 0 ... 255, а M – это номер раздела из диапазона 1 ... 63.

Следует сказать, что по умолчанию в большинстве дистрибутивов специальные файлы для разделов на md-устройствах не создаются, и их необходимо создать вручную командой mknod. В настоящий момент оптимальным, наверное, следует считать комбинирование использования LVM и md, что позволяет достигнуть надежности за счет дублирования данных средствами md, и гибкости распределения дискового пространства за счет возможностей LVM.

Драйвер md хорошо подходит для создания RAID-устройств уровней 0, 1 или 0+1, но не будет являться оптимальным вариантом в случае использования, например RAID уровня 5 (чередование данных по устройствам с вычислением контрольной суммы и кодом исправления ошибок), поскольку это создаст значительную нагрузку на процессор при большом объеме передаваемых данных. Возможно, что в таких случаях стоит подумать о приобретении аппаратного контроллера RAID - например, HP NetRaid (сделан на основе AMI MegaRAID) или Compaq Smart Array (сейчас называется HP Smart Array).

На листинге демонстрируется пример создания, активизации и остановки программных RAID-устройств уровня 0 и уровня 1 средствами драйвера md и системной утилиты mdadm. Утилита mdadm имеет конфигурационный файл /etc/mdadm.conf, но для того, чтобы проделать некоторые тесты и демонстрационные примеры нет необходимости его изменять.

В первом примере будем считать, что на жестком диске hdb создано два раздела, с которыми мы и будем экспериментировать. Для начала необходимо произвести инициализацию md-устройства. Соответственно, для успешной необходимо указать тип RAID-массива, специальный файл md-устройства, которое мы хотим инициализировать, и список блочных устройств, на которых будет располагаться получившийся массив:

```
[root@inferno dalth]#
[root@inferno dalth]# mdadm --create \
>                          /dev/md0 --level=0 \
>                          --raid-devices=2 /dev/hdb1 /dev/hdb2
mdadm: array /dev/md0 started.
[root@inferno dalth]#
[root@inferno dalth]# mdadm -Q /dev/md0
/dev/md/d0: 983.25MiB raid0 2 devices, 0 spares.
[root@inferno dalth]#
[root@inferno dalth]# mdadm -S /dev/md0
[root@inferno dalth]#
[root@inferno dalth]# mdadm --create \
>                          /dev/md0 --level=1 \
>                          --raid-devices=2 /dev/hdb1 /dev/hdb2
mdadm: array /dev/md0 started.
[root@inferno dalth]#
[root@inferno dalth]# mdadm -Q /dev/md0
/dev/md0: 491.63MiB raid1 2 devices, 0 spares.
[root@inferno dalth]#
[root@inferno dalth]# mdadm -S /dev/md0
[root@inferno dalth]#
[root@inferno dalth]# mdadm -assemble /dev/md0 /dev/hdb1 /dev/hdb2
mdadm: /dev/md0 has been started with 2 drives.
[root@inferno dalth]#
```

Последняя команда демонстрирует активизацию массив путем указания имени md-устройства и нескольки блочных устройств, на которых оно базируется. Все остальные параметры (размеры блоков, разновидность RAID и т.д.) утилита mdadm извлекла из суперблока массива. Как уже отмечалось, суперблок массива содержит еще и уникальный идентификатор массива, что дает возможность проидентифицировать каждое исходное блочное устройство на предмет его принадлежности к какому-либо массиву. Ниже приведен пример вывода утилиты **mdadm**, демонстрирующий как можно получить некоторые полезные данные о массиве, а строка, содержащая UID массива выделена жирным текстом:

```
[root@viking root]# mdadm -Q -D /dev/md0
/dev/md0:
        Version : 00.90.01
  Creation Time : Fri Oct  8 14:29:21 2004
     Raid Level : raid1
     Array Size : 102336 (99.94 MiB 104.79 MB)
    Device Size : 102336 (99.94 MiB 104.79 MB)
   Raid Devices : 2
  Total Devices : 2
Preferred Minor : 0
    Persistence : Superblock is persistent

    Update Time : Fri Oct  8 14:32:22 2004
          State : clean, no-errors
 Active Devices : 2
Working Devices : 2
 Failed Devices : 0
  Spare Devices : 0

    Number   Major   Minor   RaidDevice State
       0     253       24        0      active sync   /dev/hdb1
       1     253       25        1      active sync   /dev/hdb2
           UUID : 8696ffc0:52547452:ba369881:d1b252d0
         Events : 0.3
[root@viking root]#
```

Впоследствии этот идентификатор может быть использован в файле конфигурации для утилиты mdadm. В конфигурационном файле /etc/mdadm.conf можно указать список устройств и правила их построения, после чего описанные в нем md-устройства будут автоматически собираться и разбираться без указания списка исходных устройств:

```
[root@viking root]# cat /etc/mdadm.conf
MAILADDR root
ARRAY /dev/md0 UUID=8696ffc0:52547452:ba369881:d1b252d0
DEVICE /dev/hdb*
[root@viking root]#
```

В листинге видно, что устройство массив md0 имеет указанный идентификатор, а также указано, что для построения массивов могут быть использованы все раделы диска hdb. В этом примере если системный администратор напишет команду **mdadm –assemble /dev/md0**, то mdadm просканирует все файлы устройств с именами, совпадающими с шаблоном /dev/hdb* и подключит к массиву md0 те из них, на которых будет найдена суперблок массива с тем идентификатором, который указан в параметре *ARRAY* для устройства /dev/md0.

Системный администратор, который хочет расположить корневую файловую систему на md-устройстве, должен указать ядру при загрузке какие именно физические блочные устройства должны входить в md-устройство, на котором содержится корневая файловая система. Обычно это делается путем загрузки ядра командной строкой с опциями следующего вида:

```
linux md0=0,/dev/hdb1,/dev/hdb2 root=/dev/md0
```

Данный пример приведен скорее как иллюстративный, поскольку в зависимости от опций, использованных при создании RAID-устройства, на котором расположена корневая файловая система, командная строка ядра может меняться.

Суперблок массива записывается не в начале блочного устройства, а ближе к его середине или концу. Сделано это было для того, чтобы можно было создать RAID-массив с boot-сектором, который сможет быть прочитан не только ядром Linux с драйвером *md*, но и базовым загрузчиком BIOS, вследствие чего можно объединить в RAID-массив не *разделы* жестких дисков, а непосредственно *физические диски.* Тогда загрузчик, установленный в начало RAID-устройства, окажется установленным в начало жесткого диска, после чего можно использовать при загрузке ядра следующую командную строку:

```
linux md0=d0,/dev/hda,/dev/hdb root=/dev/md_d0p1
```

Драйвер *md* также поддерживает возможность задания hotswap-устройств для массивов, т.е. резервных устройств, которые могут быть активизированы в при сбое одного из основных устройств в массиве.

Поддержка устройств software RAID в Linux дает возможность создавать серверы с высокой отказоустойчивостью и быстродействием.

Маршрутизация IP

Маршрутизация IP является опциональной возможностью IP-стека Linux. По умолчанию функция маршрутизации не активируется, и система не маршрутизирует пакеты через свои интерфейсы, а только обрабатывает адресованные ей пакеты. Включение маршрутизации IP-пакетов производится через параметр *net.ipv4.ip_forward* интерфейса sysctl. Если значение этого параметра равно 0, то маршрутизация отключена, если же значение параметра не равно 0, маршрутизация включена:

```
[dalth@viking dalth]$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
[dalth@viking dalth]$
```

Кроме того, возможно разрешать или запрещать участие в маршрутизации для каждого интерфейса индивидуально:

```
[dalth@viking dalth]$ sysctl -a | grep forward | grep v4
net.ipv4.conf.vmnet1.mc_forwarding = 0
net.ipv4.conf.vmnet1.forwarding = 0
net.ipv4.conf.eth0.mc_forwarding = 0
net.ipv4.conf.eth0.forwarding = 0
net.ipv4.conf.lo.mc_forwarding = 0
net.ipv4.conf.lo.forwarding = 0
net.ipv4.conf.default.mc_forwarding = 0
net.ipv4.conf.default.forwarding = 0
net.ipv4.conf.all.mc_forwarding = 0
net.ipv4.conf.all.forwarding = 0
net.ipv4.ip_forward = 0
[dalth@viking dalth]$
```

По умолчанию маршрутизация включается и выключается для всех интерфейсов одновременно, но для отдельных интерфейсов возможно сменить флаг участия в маршрутизации. Изменять параметры маршрутизации может только системный администратор или пользователь, который имеет право записи в необходимые файлы интерфейса sysctl. Следующий листинг демонстрирует включение маршрутизации через все интерфейсы путем вызова программы sysctl:

```
[root@viking dalth]# sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
[root@viking dalth]# sysctl -a | grep forward | sort
net.ipv4.conf.all.forwarding = 1
net.ipv4.conf.all.mc_forwarding = 0
net.ipv4.conf.default.forwarding = 1
net.ipv4.conf.default.mc_forwarding = 0
net.ipv4.conf.eth0.forwarding = 1
net.ipv4.conf.eth0.mc_forwarding = 0
net.ipv4.conf.lo.forwarding = 1
net.ipv4.conf.lo.mc_forwarding = 0
net.ipv4.conf.vmnet1.forwarding = 1
net.ipv4.conf.vmnet1.mc_forwarding = 0
net.ipv4.ip_forward = 1
[root@viking dalth]#
```

В процессе маршрутизации для выбора интерфейса и следующего узла для доставки пакета (next hop) ядро использует *таблицу маршрутизации*. Эта таблица представляет список критериев, в соответствии с которыми выбирается следующий узел. В частности, в таблице маршрутизации фигурируют следующие условия: адрес сети получателя пакета, маска подсети получателя пакета, IP-адрес следующего узла, метрика маршрута и служебные поля (например, тип и возраст записи).

Запись о сети с адресом 0.0.0.0 и маской подсети 0.0.0.0 называют маршрутом по умолчанию, или *default route*. Узел, чей адрес указан в поле gateway для маршрута по умолчанию, называют маршрутизатором по умолчанию, или *default gateway* или *default router*. В системе может быть произвольное количество маршрутов по умолчанию, но они должны быть как минимум с

разными метриками. Для просмотра таблицы маршрутизации можно воспользоваться командой *route*. Эта команда позволяет оперировать с таблицей машрутов, добавляя и удаляя из нее записи.

```
[root@inferno dalth]# route -n
Kernel IP routing table
Destination   Gateway      Genmask         Flags Metric Ref  Use Iface
10.80.1.113   0.0.0.0      255.255.255.255 UH    0      0      0 ppp0
127.0.0.0     0.0.0.0      255.0.0.0       U     0      0      0 lo
0.0.0.0       10.80.1.113  0.0.0.0         UG    0      0      0 ppp0
[root@inferno dalth]# route del default
[root@inferno dalth]# route -n
Kernel IP routing table
Destination   Gateway      Genmask         Flags Metric Ref  Use Iface
10.80.1.113   0.0.0.0      255.255.255.255 UH    0      0      0 ppp0
127.0.0.0     0.0.0.0      255.0.0.0       U     0      0      0 lo
[root@inferno dalth]# route add default dev ppp0
[root@inferno dalth]#
```

В данном выводе таблица упорядочена по маске подсети, что соответствует порядку ее просмотра ядром. Столбцы *Destination* и *Genmask* содержат адрес и маску сети получателя пакета, столбец *Metri*c фактически указывает приоритет маршрута (маршрут с меньшей метрикой более приоритетен), поле *Gateway* указывает IP-адрес следующего узла для передачи пакета. Некоторые типы интерфейсов (в частности, интерфейсы типа точка-точка, или point-to-point) подразумевают, что на принимающем конце линии связи всегда находится не более одного узла, и поэтому в этой ситуации IP-адрес следующего узла можно не указывать. В данном случае мы видим, что в приведенном примере некоторые узлы доступны через интерфейс ppp0 типа точка-точка. В частности, именно из-за этого свойства приведенная выше таблица оказывается эквивалентна следующей ниже. Жирным шрифтом помечена измененная строка, демонстрирующая “точечную” природу PPP-соединения:

```
[root@inferno dalth]# route -n
Kernel IP routing table
Destination   Gateway     Genmask         Flags Metric Ref    Use Iface
10.80.1.113   0.0.0.0     255.255.255.255 UH    0      0        0 ppp0
127.0.0.0     0.0.0.0     255.0.0.0       U     0      0        0 lo
0.0.0.0       0.0.0.0     0.0.0.0         U     0      0        0 ppp0
```

Специфика использования протокола PPP (обычно используемого при модемых соединениях) такова, что любой PPP-интерфейс является интерфейсом типа точка-точка, более того – PPP и расшифровывается как *Point-to-Point Protocol*. Также интерфейсами точка-точка являются интерфейсы *SLIP* (Serial Line IP) и практически все разновидности туннельных интерфейсов.

При деактивизации интерфейса из таблицы маршрутизации автоматически исключаются все маршруты, для которых в поле *Iface* был указан отключившийся интерфейс. Для некоторых

типов интерфейсов при активизации в таблице маршрутизации также создаются служебные записи о маршрутах, которые нельзя удалить.

В большинстве случаев таблица маршрутизации имеет не слишком большой размер, но в некоторых ситуациях (в частности, на шлюзовых машинах в больших сетях) таблица может иметь весьма значительный размер и изменяется не “вручную” с помощью команды route, а специальными программами – демонами поддержки протоколов динамической маршрутизации. С некоторыми упрощениями алгоритм работы этих демонов можно описать следующим образом: демон “слушает” приходящие пакеты для обслуживаемых протоколов динамической маршрутизации, и по получении (или неполучении) такого пакета отдает ядру команду на изменение таблицы маршрутизации.

Следует также заметить, что команда route в режиме вывода таблицы маршрутизации фактически просто фильтрует и форматирует данные, содержащиеся в специальном файле, называемом /proc/net/route, используемом для доступа к таблице маршрутизации, ведущейся ядром.

Фильтры пакетов

Возможность инсталляции фильтров пакетов является очень интересной возможностью, предоставляемой стеком TCP/IP ядра Linux. Не углубляясь в детали просто отметим, что IP-стек Linux позволяет различным модулям установить “ловушки” (hooks) для пакетов. При этом каждый пакет, попадающий в такую ловушку, передается для обработки драйверу, установившему эту ловушку. Драйвер, в свою очередь, может проанализировать пакет, проделать какие-либо действия с пакетом, после чего вернуть код обработки, инструктируя таким образом ядро о том, что следует делать с пакетом дальше – вернуть ли отправителю сообщение об ошибке, прервать ли обработку и уничтожить пакет, либо продолжать обработку пакета обычным образом.

В настоящее время для фильтрации пакетов наиболее часто используются средства *iptables*. iptables – это набор различных драйверов, позволяющих осуществить анализ, произвести преобразование, или изменить обработку IP-пакетов. Основным объектом в iptables является цепочка правил (*chain*). Каждое правило в цепочке содержит набор условий совпадения (condition matches) и действие (*action*). Цепочки сгруппированы в таблицы (*tables*).

Действий есть две разновидности – прерывающие обработку пакета в цепочке, например действия DROP или ACCEPT, и не прерывающие обработку пакета цепочкой – например, LOG или MARK.

Цепочки используются для проверки пакетов – то есть пакет поочередно последовательно сравнивается с каждым из правил цепочки, и если он удовлетворяет всем условиям в правиле, к пакету применяется действие, указанное в этом правиле. Если действие является прерывающем, то на этом обработка пакета этой цепочкой заканчивается, если действие не прерывающее, то пакет продолжает проверяться этой же цепочкой.

Стандартные цепочки также содержат специальное неявное действие по умолчанию, называемое политикой цепочки (chain policy). Действие, указанное как политика цепочки, применяется ко всем пакетам, которые не попали ни под одно правило с “прерывающим” действием.

Стандартные таблицы и цепочки

Подсистема пакетного фильтра содержит три таблицы, в каждой из которых содержатся несколько цепочек (наборов правил). Кроме того, администратор может создавать собственные цепочки правил. Ниже перечисляются стандартные таблицы и цепочки:

Таблица	Назначение	Цепочки	Назначение
mangle	Модификация пакетов	PREROUTING	Модификация всех пришедших пакетов
		INPUT	Модификация пакетов, пришедших на адрес компьютера
		FORWARD	Модификация пакетов, которые должны быть отмаршрутизированы (пересланы на другой хост)
		OUTPUT	Модификация пакетов, сгенерированных процессами данного хоста
		POSTROUTING	Модификация всех переданых пакетов
filter	Фильтрация пакетов – принятие решения об их дальнейшей обработке или отказе от обработки)	INPUT	Фильтрация адресованных этому компьютеру пакетов
		OUTPUT	Фильтрация сгенерированных этим компьютером пакетов
		FORWARD	Фильтрация маршрутизируемых (транзитных) пакетов
nat	Трансляция адресов	PREROUTING	Трансляция адресов всех принимаемых пакетов
		FORWARD	Трансляция адресов транзитных пакетов
		POSTROUTING	Трансляция адресов всех передаваемых пакетов

Таблица *nat* обладает “двойственным” действием, т.е. если вы включите преобразование для входящих пакетов, исходящие пакеты также будут модифицироваться, и наоборот. Таблицы mangle и nat изменяют пакеты, но mangle не ведет список изменений, т.е. является “однонаправленой” таблицей.

Порядок применения стандартных таблиц и цепочек

Рассмотрим, какой путь проходит пакет в цепочках и таблицах iptables. Для входящих пакетов (адресованных компьютеру, на котором активизирована поддержка iptables) верна следующая последовательность применения цепочек:

```
1. mangle.PREROUTING
2. nat.PREROUTING
3. mangle.INPUT
4. filter.INPUT
```

Для пакетов, отправляемых с компьютера, реализуется следующая цепочка обработки:

```
1. filter.OUTPUT
2. mangle.OUTPUT
3. nat.POSTROUTING
4. mangle.POSTROUTING
```

Пакеты, являющиеся транзитными (маршрутизируемыми), т.е. не адресованные фильтрующему компьютеру и не сгенерированные фильтрующим компьютером, проходят по следующей последовательности цепочек и таблиц:

```
1. mangle.PREROUTING
2. nat.PREROUTING
3. mangle.FORWARD
4. filter.FORWARD
5. nat.FORWARD
6. nat.POSTROUTING
7. mangle.POSTROUTING
```

Стандартные действия

Каждое правило в любой цепочке может ссылаться на одно из стандартных или дополнительных действий, либо на какую-либо пользовательскую цепочку. Основное различие между стандартными и дополнительными действиями в том, что стандартные действия могут указываться в правилах любых цепочек любых таблиц, а дополнительные действия можно указывать только для некоторых цепочек некоторых таблиц. Перечислим стандартные действия:

ACCEPT	Прервать проверку пакета цепочкой и перейти к следующей в порядке обработки пакета стандартной цепочке
DROP	Прервать обработку пакета, сам пакет удалить
RETURN	Прервать проверку пакета цепочкой и вернуться к проверке вышестоящей цепочкой, а если действие встретилось в одно из стандартных цепочек, поступить с пакетом так, как предписано в политике цепочки (chain policy)
QUEUE	Передать пакет некоторому процессу для дальнейшей обработки

Интересной особенностью также является то, что существуют так называемые target extensions, которые реализованы как модули и также могут использоваться для указания проводимого над пакетом действия. В частности, к таким действиям, например, относятся действия LOG – запротоколировать факт получения пакета, MASQUERADE – подменить IP-адрес отправителя пакета, MARK – пометить пакет и многие другие. Некоторые действия могут встречаться не во всех цепочках, а только в некоторых цепочках некоторых таблиц.

Еще одним специфическим действием можно назвать переход к пользовательской цепочке. При этом, если пакет в процессе обработки попадает под действие правила, у которого в качестве действия указан переход к другой цепочке, то пакет начинает проверяться ее правилами. Это часто используется, чтобы сходным образом обрабатывать некоторые виды пакетов.

Условия отбора

Условия отбора делятся на две группы – стандартные условия, которые применимы ко всем пакетам, и расширенные условия, называемые также match extensions. Расширенные условия могут применяться не для всех пакетов, а только для некоторых из них – например, дополниетельные условия для протокола UDP включают в себя адреса портов отправителя и получателя, а для ICMP – тип и код ICMP-сообщения.

Примеры конфигураций iptables

Попробуем рассмотреть несколько простых примеров, достаточно часто используемых в реальных конфигурациях. Стоит заметить, что самостоятельная конфигурация пакетного фильтра требует некоторых (точнее, достаточно значительных) знаний сетевых протоколов, поскольку в конфигурации необходимо задавать множество критериев, которые сильно зависят от ситуации и от используемых сервисов.

Пример 1: простейшая конфигурация iptables для домашнего компьютера, подключенного к Internet. В этой конфигурации мы запретим все входящие соединения, а также все исходящие пакеты UDP кроме тех, которые необходимы для нормальной работы в Internet с

использованием PPP-соединения. В этой конфигурации мы разрешаем передачу всех пакетов в рамках локальной машины, разрешаем исходящие TCP-пакеты, разрешаем входящие пакеты TCP для уже установленных соединений, и разрешаем передачу и прием UDP-пакетов для службы DNS и пакетов автоматической конфигурации соединения PPP (пакетов DHCP). Кроме того, следует разрешить прием управляющих пакетов ICMP и отправку запросов и прием ответов PING:

```
# iptables -P INPUT DROP
# iptables -A INPUT -j ACCEPT -i lo
# iptables -A INPUT -j ACCEP -p tcp ! --syn
# iptables -A INPUT -j ACCEPT -p udp --source-port 53
# iptables -A INPUT -j ACCEPT -p udp --source-port 67 --destination-port 68
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type destination-unreachable
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type time-exceeded
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type parameter-problem
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type echo-reply
# iptables -P OUTPUT DROP
# iptables -A OUTPUT -j ACCEPT -p tcp
# iptables -A OUTPUT -j ACCEPT -p udp --destination-port 53
# iptables -A OUTPUT -j ACCEPT -p udp --destination-port 67 --source-port 68
# iptables -A OUTPUT -j ACCEPT -p icmp --icmp-type echo-request
```

Пример 2: то же самое, что в примере 1, но все “отбитые” пакеты протоколируются. Для того, чтобы добиться такого эффекта, нужно создать дополнительную цепочку, которая будет протоколировать и удалять пакеты. Эту цепочку мы назовем KILLER – вполне обоснованно, не так ли? Кроме того, мы исправим политики стандартных цепочек так, чтобы запрещенные пакеты не удалялись, а “забрасывались” в созданную нами цепочку KILLER, а нашей основной цели (сначала протоколировать, потом удалить пакет) можно добиться просто указав два действия – сначала LOG, затем DROP. Поскольку действие LOG не является прерывающим обработку, мы получим требуемый нам эффект:

```
# iptables -N KILLER
# iptables -A KILLER -j LOG
# iptables -A KILLER -j DROP
# iptables -P INPUT KILLER
# iptables -A INPUT -j ACCEPT -i lo
# iptables -A INPUT -j ACCEP -p tcp ! --syn
# iptables -A INPUT -j ACCEPT -p udp --source-port 53
# iptables -A INPUT -j ACCEPT -p udp --source-port 67 --destination-port 68
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type destination-unreachable
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type time-exceeded
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type parameter-problem
# iptables -A INPUT -j ACCEPT -p icmp --icmp-type echo-reply
# iptables -P OUTPUT KILLER
# iptables -A OUTPUT -j ACCEPT -p tcp
# iptables -A OUTPUT -j ACCEPT -p udp --destination-port 53
# iptables -A OUTPUT -j ACCEPT -p udp --destination-port 67 --source-port 68
# iptables -A OUTPUT -j ACCEPT -p icmp --icmp-type echo-request
```

Пример 3: маскарад пакетов. Маскарадом называют преобразование IP-адресов проходящих пакетов так, чтобы они выглядели как отправленные с системы-маршрутизатора, а не с какого-либо узла “за” маршрутизатором. Достигается это путем изменения IP-адреса (и, возможно, номера порта) в “транзитных” пакетах. Собственно преобразование задается путем указания действий SNAT – замена адреса отправителя, DNAT – замена адреса получателя, или MASQUERADE – аналогично SNAT, но без указания конкретного IP-адреса. Предположим, что наш “внешний” интерфейс имеет адрес 193.267.14.6, а внутренняя сеть имеет адрес 192.168.0.0/24. Тогда для того, чтобы дать всем компьютерам нашей сети доступ по протоколу TCP “наружу”, мы должны подать примерно следующую команду:

```
# iptables -A POSTROUTING -t nat -j SNAT -o ppp0 \
>   --to-source 193.267.14.6 -p tcp \
>   --source 192.168.0.0/24 \
>   --destination ! 192.168.0.0/24
```

Если у нас внешний адрес динамический, а не статический (мы работаем по dialup – соединению), то мы можем использовать динамический маскарад без привязки к внешнему адресу – ну или с использованием динамической привязки, кому как больше нравится:

```
# iptables -A POSTROUTING -t nat -j MASQUERADE -o ppp0 \
>   --source 192.168.0.0/24 \
>   --destination ! 192.168.0.0/24
```

Действие SNAT более эффективно, MASQUERADE гибче и проще в использовании. Особое внимание нужно обратить на указание -o ppp0, то есть действие применяется ТОЛЬКО для пакетов, отправляемых через интерфейс ppp0. Еще вы можете увидеть, что мы указываем это правило только один раз, и обратного к нему правила не строим - об этом позаботится функция connection tracking (отслеживание состояния соединений), и обратная замена адресов в отправляемых в ответ на наши запросы пакетах будет произведена системой автоматически.

Пример 4: “проброс” пакетов во внутреннюю сеть. Обычно это используется, если мы хотим перебросить пакеты, пришедшие на адрес маршрутизатора, на какую-либо из машин внутренней сети (например, так можно предоставить доступ ко внутреннему WWW-серверу). Достигается это использованием действия DNAT (destination NAT). В нашем случае мы перебрасываем все TCP-пакеты, пришедшие на интерфейс маршрутизатора ppp0 на порт 80, на порт 85 компьютера с адресом 192.168.0.6:

```
# iptables -A PREROUTING -t nat -j DNAT -i ppp0 \
>   --to-destination 192.168.0.6:80 -p tcp --destination-ports 80
```

Конечно, приведенными примерами возможности iptables не исчерпываются, скорее даже

наоборот – это лишь малая часть того, что умеет эта подсистема. Приведенные же примеры демонстрируют наиболее простые случаи, позволяющие составить некоторое представление об iptables и возможностях этой технологии.

В системах, основанных на RedHat Linux и его вариациях, есть специальный стартовый сценарий загрузки, также называемый iptables. Расположен он как правило в каталоге **/etc/rc.d/init.d/**. Этот сценарий при загрузке системы инсталлирует правила, созданные администратором, и его можно использовать для сохранения конфигурации iptables. В процессе конфигурации системный администратор задает конфигурацию подсистемы iptables, используя утилиту **/sbin/iptables**, а после окончания настройки дает команду **/etc/rc.d/init.d/iptables save**, после чего текущая конфигурация сохраняется в файл **/etc/sysconfig/iptables**. Существует также множество “фронтендов” для настройки iptables, которые могут использоваться начинающими пользователями и не слишком опытными администраторами, но “ручной” способ настройки все-таки предпочтительней, поскольку позволяет очень точно настроить подсистему iptables.