**Лекция 1:**

**Сеанс работы в Linux**

**Пользователи системы**

Прежде, чем система будет готова к работе с пользователем, происходит **процедура загрузки** системы. В процессе загрузки будет запущена основная управляющая *программа* ( *ядро* ), определено и инициализировано имеющееся оборудование, активизированы сетевые соединения, запущены системные службы. В Linux во время загрузки на экран выводятся диагностические сообщения о происходящих событиях, и если все в порядке и не возникло никаких ошибок, *загрузка* завершится выводом на экран приглашения " login:". Оно может выглядеть по-разному, в зависимости от настройки системы: может отображаться в красиво оформленном окне или в виде простой текстовой строки вверху экрана. Это приглашение к *регистрации в системе*: система ожидает, что в ответ на это приглашение будет введено *входное имя* пользователя, который начинает работу. Естественно, имеет смысл вводить такое имя, которое уже известно системе, чтобы она могла "узнать", с кем предстоит работать - выполнять команды неизвестного пользователя Linux откажется.

**Многопользовательская модель разграничения доступа**

Процедура *регистрации в системе* для Linux **обязательна**: работать в системе, не зарегистрировавшись под тем или иным *именем пользователя*, просто **невозможно** 1. Для каждого пользователя определена сфера его полномочий в системе: программы, которые он может запускать, файлы, которые он имеет право просматривать, изменять, удалять. При попытке сделать что-то, выходящее за рамки полномочий, пользователь получит сообщение об ошибке. Такая строгость может показаться излишней, если пользователи компьютера доверяют друг другу, и особенно если у компьютера только один пользователь. Эта ситуация очень распространена в настоящее время, когда слово "компьютер" означает в первую очередь " *персональный компьютер* ".

Однако *персональный компьютер* - довольно-таки позднее явление в мире вычислительной техники, получившее широкое распространение только в последние два десятилетия. Раньше слово "компьютер" ассоциировалось с огромным и дорогостоящим (занимавшим целые залы) вычислительным центром, предназначенным в первую очередь для решения разного рода научных задач. Машинное время такого центра стоит очень недешево, и при этом его возможности необходимы одновременно многим сотрудникам, которые могут ничего не знать о работе друг друга. Требуется следить за тем, чтобы не произошло случайного вмешательства пользователей в чужую работу и повреждения данных (файлов), выделять каждому машинное время (по возможности избежав простаивания) и пространство на диске и при этом не допускать захвата всех ресурсов одним пользователем и его задачей, а равномерно распределять ресурсы между всеми. Для такой системы принципиально важно знать, кому принадлежат задачи и файлы, поэтому и возникла необходимость предоставлять доступ к ресурсам системы только после того, как пользователь **зарегистрируется в системе** под тем или иным именем.

Такая модель была реализована в *многопользовательской операционной системе* UNIX. Именно от нее Linux - также *многопользовательская операционная система* - унаследовала принципы работы с пользователями. Но это не просто дань традиции или стремление к универсальности: многопользовательская модель позволяет решить ряд задач, весьма актуальных и для современных *персональных компьютеров*, и для серверов, работающих в локальных и глобальных сетях, и вообще в любых системах, одновременно выполняющих **разные** задачи, за которые отвечают **разные** люди.

Компьютер - это всего лишь инструмент для решения разного рода прикладных задач: от набора и распечатывания текста до вычислений. Сложность состоит в том, что для настройки этого инструмента и для работы с его помощью используются одни и те же операции: изменение файлов и выполнение программ. Получается, что, если не соблюдать осторожность, побочным результатом работы может стать выход системы из строя. Поэтому первоочередная задача для систем любого масштаба - разделять повседневную работу и изменение самой системы. В многопользовательской модели эта задача решается очень просто: разделяются *"обычные" пользователи* и *администратор(ы)*. В полномочия *обычного пользователя* входит все необходимое для выполнения прикладных задач, попросту говоря, для работы, однако ему запрещено выполнять действия, изменяющие саму систему. Таким образом можно избежать повреждения cистемы в результате ошибки пользователя (нажал не ту кнопку) или ошибки в программе, или даже по злому умыслу (например, вредительской программой-вирусом). Полномочия *администратора* обычно не ограничены.

Для *персонального компьютера*, с которым работают несколько человек, важно обеспечить каждому пользователю независимую рабочую среду. Это снижает вероятность случайного повреждения чужих данных, а также позволяет каждому пользователю настроить внешний вид рабочей среды по своему вкусу и, например, сохранить расположение открытых окон между сеансами работы. Эта задача очевидным образом решается в многопользовательской модели: организуется *домашний каталог*, где хранятся данные пользователя, настройки внешнего вида и поведения его системы и т.п, а доступ остальных пользователей к этому каталогу ограничивается.

Если компьютер подключен к глобальной или локальной сети, то вполне вероятно, что какую-то часть хранящихся на нем ресурсов имеет смысл сделать публичной и доступной по сети. И напротив, часть данных, скорее всего, делать публичными не следует (например, личную переписку). Ограничив доступ пользователей к персональным данным друг друга, мы решим и эту задачу.

Именно благодаря гибкости многопользовательской модели разграничения доступа она используется сегодня не только на серверах, но и на домашних *персональных компьютерах*. В самом простом варианте - для *персонального компьютера*, на котором работает только один человек - эта модель сводится к двум пользователям: *обычному пользователю* для повседневной работы и *администратору* - для настройки, обновления, дополнения системы и исправления неполадок. Но даже в таком сокращенном варианте это дает целый ряд преимуществ.

**Учетные записи**

Конечно, система может быть "знакома" с человеком только в переносном смысле: в ней должна храниться запись о пользователе с таким именем и о связанной с ним системной информации - ***учетная запись***. Английский эквивалент термина *учетная запись* - ***account***, "счет". Именно с *учетными записями*, а не с самими пользователями, и работает система. В действительности, соотношение *учетных записей* и пользователей в Linux обычно не является однозначным: несколько человек могут использовать одну *учетную запись* - система не может их различить. И в то же время в Linux имеются *учетные записи* для *системных пользователей*, от имени которых работают некоторые программы, но не люди.

*Учетная запись* ( **account** ) - объект системы, при помощи которого Linux ведет учет работы пользователя в системе. *Учетная запись* содержит данные о пользователе, необходимые для *регистрации в системе* и дальнейшей работы с ней.

*Учетные записи* могут быть созданы во время установки системы или после установки. Подробно процедура создания *учетных записей* (добавления пользователей) описана в лекции 12.

Главное для человека в *учетной записи* - ее название, *входное имя* пользователя. Именно о нем спрашивает система, когда выводит приглашение " login:". Помимо *входного имени* в *учетной записи* содержатся некоторые сведения о пользователе, необходимые системе для работы с ним. Ниже приведен список этих сведений.

***Входное имя (login name)*** - название *учетной записи* пользователя, которое нужно вводить при *регистрации в системе* .

**Идентификатор пользователя**

Linux связывает *входное имя* c *идентификатором пользователя* в системе - *UID* (User ID). ***UID*** - это положительное целое число, по которому система и отслеживает пользователей2. Обычно это число выбирается автоматически при регистрации *учетной записи*, однако оно не может быть произвольным. В Linux есть некоторые соглашения относительно того, какому типу пользователей могут быть выданы идентификаторы из того или иного диапазона. В частности, *UID* от " 0 " до " 100 " зарезервированы для псевдопользователей3.

*Идентификатор пользователя*, *UID* - уникальное число, однозначно идентифицирующее *учетную запись* пользователя в Linux. Таким числом снабжены все *процессы* Linux и все объекты *файловой системы*. Используется для персонального учета действий пользователя и определения *прав доступа* к другим объектам системы.

**Идентификатор группы**

Кроме идентификационного номера пользователя, с *учетной записью* связан *идентификатор группы*. *Группы пользователей* применяются для организации доступа нескольких пользователей к некоторым ресурсам. У группы, так же, как и у пользователя, есть имя и идентификационный номер - ***GID*** (Group ID). В Linux пользователь должен принадлежать как минимум к одной группе - *группе по умолчанию*. При создании *учетной записи* пользователя обычно создается и группа, имя которой совпадает с входным именем4, именно эта группа будет использоваться как ***группа по умолчанию*** для данного пользователя. Пользователь может входить более чем в одну группу, но в *учетной записи* указывается только номер *группы по умолчанию*.

**Полное имя**

Помимо *входного имени* в *учетной записи* содержится и *полное имя* (имя и фамилия) использующего данную *учетную запись* человека. Конечно, пользователь может указать что угодно в качестве своего имени и фамилии. *Полное имя* необходимо не столько системе, сколько людям - чтобы иметь возможность определить, кому принадлежит *учетная запись*.

**Домашний каталог**

Файлы всех пользователей в Linux хранятся раздельно, у каждого пользователя есть собственный *домашний каталог*, в котором он может хранить свои данные. Доступ других пользователей к *домашнему каталогу* пользователя может быть ограничен. Информация о *домашнем каталоге* обязательно должна присутствовать в *учетной записи*, потому что именно с него начинает работу пользователь, зарегистрировавшийся в системе.

**Командная оболочка**

Каждому пользователю нужно предоставить способ взаимодействия с системой: передача ей команд и получение от нее ответов. Для этой цели служит специальная программа - ***командная оболочка*** (или ***интерпретатор командной строки*** ). Она должна быть запущена для каждого пользователя, который зарегистрировался в системе. Поскольку в Linux доступно несколько разных *интерпретаторов командной строки*, в *учетной записи* указано, какой из них нужно запустить для данного пользователя. Если специально не указывать *командную оболочку* при создании *учетной записи*, она будет назначена по умолчанию, вероятнее всего это будет bash.

***Интерпретатор командной строки*** ( **командный интерпретатор**, *командная оболочка*, **оболочка** ) - это программа, используемая в Linux для организации "диалога" человека и системы. Командный интерпретатор имеет три основных ипостаси: (1) редактор и анализатор команд в *командной строке*, (2) высокоуровневый системно-ориентированный язык программирования, (3) средство организации взаимодействия команд друг с другом и с системой.

**Понятие "администратор"**

В Linux есть только один пользователь, полномочия которого в системе принципиально отличаются от полномочий остальных пользователей - это пользователь с идентификатором " 0 ". Обычно *учетная запись* пользователя с UID=0 называется root (англ., "корень"). Пользователь root - это *"администратор"* системы Linux, *учетная запись* для root обязательно присутствует в любой системе Linux, даже если в ней нет никаких других *учетных записей*. Пользователю с таким *UID* разрешено выполнять **любые** действия в системе, а значит, любая ошибка или неправильное действие может повредить систему, уничтожить данные и привести к другим печальным последствиям. Поэтому **категорически** не рекомендуется регистрироваться в системе под именем root для повседневной работы. Работать в root следует только тогда, когда это действительно необходимо: при настройке и обновлении системы или восстановлении после сбоев.

**Регистрация в системе**

Вернемся теперь к нашей загруженной операционной системе Linux, которая по-прежнему ожидает ответа на свое приглашение " login:". Если система настроена таким образом, что это приглашение оформлено в виде графического окна в центре экрана, следует нажать комбинацию клавиш Ctrl+Alt+F1 - произойдет переключение видеорежима и на экране на черном фоне появится примерно следующий текст:

Welcome to Some Linux / tty1

localhost login:

Пример 1.1. Начальное приглашение к регистрации

Мы переключились в так называемый *текстовый режим*, в котором нам недоступны возможности графических интерфейсов: рисование окон произвольной формы и размера, *поддержка* миллионов цветов, отрисовка изображений. Все возможности *текстового режима* ограничены набором текстовых и псевдографических символов и несколькими десятками базовых цветов. Однако в Linux в *текстовом режиме* можно выполнять практически любые действия в системе (кроме тех, которые требуют непосредственного **просмотра** изображений). *Текстовый режим* в Linux - это полнофункциональный способ управления системой. В различных реализациях Linux работа в графическом режиме может выглядеть по-разному5, более того, *графический режим* может быть даже недоступен после установки системы без специальной настройки. Текстовый же режим доступен в любой реализации Linux и всегда выглядит практически одинаково. Именно поэтому все дальнейшие примеры и упражнения мы будем рассматривать и выполнять в *текстовом режиме*, возможностей которого будет достаточно для освоения излагаемого в курсе материала.

Первая строка в примере - это просто приглашение, она носит информационный характер. Существует очень много различных реализаций Linux (называемых дистрибутивами, они будут обсуждаться в лекции 18), и в каждом из них принят свой формат первой строки приглашения, хотя почти наверняка там будет указано, с какой именно версией Linux *пользователь* имеет дело, и, возможно, будут присутствовать еще некоторые параметры. В наших примерах мы будем использовать условную реализацию Linux - "Some Linux".

Вторая строка начинается с *имени хоста* - собственного имени системы, установленной на данном компьютере. Это имя существенно в том случае, если *компьютер* подключен к локальной или глобальной сети, если же он ни с кем более не связан, оно может быть любым. Обычно *имя хоста* определяется уже при установке системы, однако в нашем случае используется вариант по умолчанию - " localhost ". Заканчивается эта строка собственно приглашением к *регистрации в системе* - словом " login:".

Теперь понятно, что в ответ на данное приглашение мы должны ввести *входное имя*, для которого есть соответствующая *учетная запись* в системе. В правильно установленной операционной системе Linux должна существовать как *минимум* одна *учетная запись* для *обычного пользователя*. Во всех дальнейших примерах у нас будет участвовать Мефодий Кашин, владелец *учетной записи* " methody " в системе "Some Linux". Вы можете пользоваться для выполнения примеров любой *учетной записью*, которая создана в Вашей системе (естественно, кроме root ).

Итак, Мефодий вводит свое *входное имя* в ответ на приглашение системы:

Welcome to Some Linux / tty1

localhost login: Methody

Password:

Login incorrect

login:

Пример 1.2. Регистрация в системе

В ответ на это система запрашивает *пароль*. *Пароль* Мефодия нам неизвестен, поскольку он его никому не говорит. Когда Мефодий вводил свой *пароль*, на экране монитора он не отображался (это сделано, чтобы *пароль* нельзя было подсмотреть), однако Мефодий точно знает, что не сделал опечатки. Тем не менее, система отказала ему в регистрации, выдав *сообщение об ошибке* (" Login *incorrect* "). Если же внимательно посмотреть на введенное *имя пользователя*, можно заметить, что оно начинается с заглавной буквы, в то время как *учетная запись* называется " methody ". Linux всегда делает различие между заглавными и строчными буквами, поэтому " Methody " для него - уже другое имя. Теперь Мефодий повторит попытку:

login: methody

Password:

[methody@localhost methody]$

Пример 1.3. Успешная регистрация в системе

На этот раз *регистрация* прошла успешно, о чем свидетельствует последняя строка примера - *приглашение командной строки*. Приглашение - это подсказка, выводимая *командной оболочкой* и свидетельствующая о том, что система готова принимать команды пользователя. Приглашение может быть оформлено по-разному, более того, *пользователь* может сам управлять видом приглашения (подробнее это будет рассмотрено в лекции 7), но почти наверняка в приглашении содержатся *входное имя* и *имя хоста* - в нашем примере это " methody " и " localhost " соответственно. Заканчивается приглашение чаще всего символом " $ ". Это *командная строка*, в которой будут отображаться все введенные пользователем с клавиатуры команды, а при нажатии на клавишу *Enter* содержимое *командной строки* будет пеpедано для исполнения системе.

**Идентификация (authentication)**

Когда система выводит на экран *приглашение командной строки* после правильного введения *имени пользователя* и *пароля*, это означает, что произошла *идентификация пользователя*. *Пароль* может показаться излишне сложным, но у системы нет другого способа удостовериться, что за монитором находится именно тот человек, который имеет право на использование данной *учетной записи*. Конечно, процедура *идентификации* имеет очевидное значение для систем, к которым имеют непосредственный или сетевой доступ многие не связанные друг с другом пользователи. Процедура *идентификации* гарантирует, что к такой системе не получит доступ случайный человек, не имеющий права использовать ее ресурсы и хранящуюся в ней информацию. Одновременно она дает определенную гарантию защиты от злонамеренного вмешательства: даже если навредить попытается пользователь, имеющий *учетную запись*, его действия будут зарегистрированы в системе (поскольку системе всегда известно, от имени какой *учетной записи* выполняются те или иные действия), и злоумышленника можно будет найти. Для тех пользователей, которым процедура *идентификации* кажется утомительной и необязательной (например, единственным пользователям *персональных компьютеров* ), существует возможность получить доступ к системе, минуя процедуру *идентификации*. Для этого применяется программа autologin. Она предоставляет доступ к работе с графическим интерфейсом сразу после загрузки системы, не запрашивая *имя пользователя* и *пароль*. В действительности, autologin запускает все программы от имени одного пользователя, зарегистрированного в системе. Например, Мефодий мог бы использовать свою учетную запись methody для автоматического входа в систему. Однако у этого подхода есть свои минусы:

* Невозможно определить, кто, что и когда делал в системе, потому что все реальные пользователи работают с одной учетной записью, то есть с точки зрения системы все они - один и тот же пользователь.
* Вся личная информация этого пользователя становится достоянием общественности.
* *Пароль* легко забывается ( *пароль* все равно есть у любого пользователя), потому что его не нужно вводить каждый день. При этом autologin дает доступ только человеку, сидящему перед монитором, и только к работе с графическим интерфейсом. Если же потребуется, например, скопировать файлы с данного компьютера по сети, *пароль* все равно придется вводить.

Учитывая все перечисленные минусы, можно заключить, что использовать autologin разумно только в тех системах, которые не подключены к локальной или глобальной сети, и к которым при этом открыт публичный доступ (например, в библиотеке).

**Смена пароля**

Если *учетная запись* была создана не самим пользователем, а *администратором* многопользовательской системы (скажем, *администратором* компьютерного класса), скорее всего, был выбран тривиальный *пароль* с тем расчетом, что пользователь его изменит при первом же входе в систему. Распространены тривиальные пароли " 123456 "., " empty " и т. п. Поскольку *пароль* - это единственная гарантия, что вашей *учетной записью* не воспользуется никто, кроме вас, есть смысл выбирать в качестве *пароля* неочевидные последовательности символов. В Linux нет существенных ограничений на длину *пароля* или входящие в него символы (в частности, использовать пробел **можно** ), но нет смысла делать *пароль* слишком длинным - велика вероятность его забыть. Надежность *паролю* придает его непредсказуемость, а не длина. Например, *пароль*, представляющий собой *имя пользователя* или повторяющий название его *учетной записи*, **очень предсказуем**. Наименее предсказуемы *пароли*, представляющие собой случайную комбинацию прописных и строчных букв, цифр, знаков препинания, но их и труднее всего запомнить.

Пользователь может в любой момент поменять свой *пароль*. Единственное, что требуется для смены *пароля* - знать текущий *пароль*. Допустим, Мефодий придумал более удачный *пароль* и решил его поменять. Он уже зарегистрирован в системе, поэтому ему нужно только набрать в *командной строке* команду passwd и нажать Enter.

[methody@localhost methody]$ passwd

Changing password for methody.

Enter current password:

You can now choose the new password or passphrase.

A valid password should be a mix of upper and lower

case letters, digits, and other characters. You can

use an 8 character long password with characters from

at least 3 of these 4 classes, or a 7 character long

password containing characters from all the classes.

An upper case letter that begins the password and a

digit that ends it do not count towards

the number of character classes used.

A passphrase should be of at least 3 words, 12 to 40

characters long and contain enough different characters.

Alternatively, if noone else can see your terminal now,

you can pick this as your password: "spinal&state:buy".

Enter new password:

Пример 1.4. Смена пароля

Набрав в командной строке " passwd ", Мефодий запустил программу passwd, которая предназначена именно для замены информации о *пароле* в *учетной записи* пользователя. Она вывела приглашение ввести текущий *пароль* (" Enter current password "), а затем, в ответ на правильно введенный *пароль*, предложила подсказку относительно грамотного составления *пароля* и даже вариант надежного *пароля*, который Мефодий вполне может использовать, если никто в данный момент не видит его монитора. При каждом запуске passwd генерирует новый случайный *пароль* и предлагает его пользователю. Однако Мефодий не воспользовался подсказкой и придумал *пароль* сам:

Enter new password:

Weak password: not enough different characters

or classes for this length.

Try again.

. . .

Enter new password:

Пример 1.5. Смена пароля (продолжение)

В данном случае операция не удалась, поскольку с точки зрения passwd *пароль*, придуманный Мефодием, оказался слишком простым6. В следующий раз ему придется ввести более сложный *пароль*. passwd запрашивает новый *пароль* дважды, чтобы удостовериться, что в первый раз не было опечатки, и если все в порядке, выведет сообщение о том, что операция смены *пароля* прошла успешно, а затем завершит работу, вернув Мефодию *приглашение командной строки:*

Enter new password:

Re-type new password:

passwd: All authentication tokens updated successfully

[methody@localhost methody]$

Пример 1.6. Пароль изменен

Придирчивость, с которой passwd относится к *паролю* пользователя, не случайна. *Пароль* пользователя - одно из самых важных и зачастую одно из самых слабых мест безопасности системы. Отгадавший *пароль* пользователя (причем не имеет значения, сделал это человек или программа) получит доступ к ресурсам системы ровно в том объеме, в котором он предоставляется пользователю, сможет читать и удалять файлы и т. п. Особенно это важно в случае *пароля* администратора, потому что его полномочия в системе гораздо шире, а действия от его имени могут повредить и саму систему. Обычному пользователю в некоторых обстоятельствах также могут быть переданы полномочия *администратора* (этот вопрос будет подробно обсуждаться в лекции 4), в таком случае не менее важно, чтобы и его *пароль* был надежным.

*Пароль* пользователя root изначально назначается при установке системы, однако он может быть изменен в любой момент впоследствии точно так же, как и *пароль* обычного пользователя.

**Одновременный доступ к системе**

То, что Linux - многопользовательская и многозадачная система, проявляется не только в разграничении *прав доступа*, но и в организации рабочего места. Каждый *компьютер*, на котором работает Linux, предоставляет возможность зарегистрироваться и получить *доступ* к системе нескольким пользователям одновременно. Даже если в распоряжении всех пользователей есть только один *монитор* и одна системная клавиатура, эта возможность небесполезна: одновременная *регистрация в системе* нескольких пользователей позволяет работать по очереди без необходимости каждый раз завершать все начатые задачи (закрывать все окна, прерывать *исполнение* всех программ) и затем возобновлять их. Более того, ничто не препятствует зарегистрироваться в системе несколько раз под одним и тем же *входным именем*. Таким образом, можно получить *доступ* к одним и тем же ресурсам (своим файлам) и организовать параллельную работу над несколькими задачами.

**Виртуальные консоли**

Характерный для Linux способ организации параллельной работы пользователей - *виртуальные консоли*.

Допустим, Мефодий хочет зарегистрироваться в системе еще раз, чтобы иметь возможность следить за выполнением двух программ одновременно. Он может сделать это, не покидая *текстового режима*: достаточно нажать комбинацию клавиш Alt+F2, и на экране появится новое приглашение к *регистрации в системе*:

Welcome to Some Linux / tty2

localhost login: methody

Password:

[methody@localhost methody]$

Пример 1.7. Вторая виртуальная консоль

Мефодий ввел свой новый *пароль* и получил *приглашение командной строки*, аналогичное тому, которое мы уже видели в предыдущих примерах. Нажав комбинацию клавиш Alt+F1, Мефодий вернется к только что покинутой им *командной строке*, в которой он выполнял команду passwd для смены *пароля*. Приглашение в обоих случаях выглядит одинаково, и это не случайно - обе *командные строки* предоставляют эквивалентный доступ к системе, в любой из них можно выполнять все доступные команды.

Наблюдательный Мефодий обратил внимание, что в последнем примере (1.7) первая строка приглашения оканчивается словом " tty2 ". " tty2 " - это обозначение **второй** *виртуальной консоли*. Можно переключаться между *виртуальными консолями* так, как если бы вы переходили от одного монитора с клавиатурой к другому, подавая время от времени команды и следя за выполняющимися программами. По умолчанию в Linux доступно не менее шести *виртуальных консолей*, переключаться между которыми можно при помощи сочетания клавиши Alt с одной из функциональных клавиш ( F1-F6 ). С каждым сочетанием связана соответствующая по номеру *виртуальная консоль*. *Виртуальные консоли* обозначаются " ttyN ", где " N " - номер *виртуальной консоли*.

*Виртуальные консоли*, **virtual console** - это несколько параллельно выполняемых операционной системой программ, предоставляющих пользователю возможность зарегистрироваться в системе в *текстовом режиме* и получить доступ к *командной строке*.

Во многих дистрибутивах Linux одна из *виртуальных консолей* по умолчанию не может быть использована для регистрации пользователя, однако она не менее, если не более, полезна. Если Мефодий нажмет Alt+F12, он увидит консоль, заполненную множеством сообщений системы о происходящих событиях. В частности, там он может обнаружить две записи о том, что в системе зарегистрирован пользователь " methody ". На эту консоль выводятся сообщения обо всех важных событиях в системе: регистрации пользователей, выполнении действий от имени *администратора* ( root ), подключении устройств и подгрузке драйверов к ним и многое другое.

Пример двенадцатой *виртуальной консоли* показывает, что *виртуальные консоли* - довольно гибкий механизм, поддерживаемый Linux, при помощи которого можно решать разные задачи, а не только обеспечивать организацию одновременного доступа к системе. Для того чтобы на *виртуальной консоли* появилось приглашение login: после загрузки системы, для каждой такой консоли должна быть запущена программа getty. Попробуйте нажать Alt+F10 - вероятнее всего, вы увидите просто черный экран. Десятая *виртуальная консоль* поддерживается системой, однако черный экран означает, что для этой консоли не запущена никакая программа, поэтому воспользоваться ею не удастся. Для каких именно консолей будет запущена программа getty - определяется настройкой конкретной системы. Впоследствии эта настройка может быть изменена пользователем. О том, как это можно сделать, речь пойдет в лекции 9.

**Графические консоли**

Впрочем, как ни широки возможности *текстового режима*, Linux ими не ограничена. Подробно работа в графическом режиме будет разбираться в последующих лекциях (см. лекцию 16). Сейчас важно заметить, что если при загрузке системы приглашение " login:" было представлено в виде графического окна, можно вернуться к этому приглашению, нажав комбинацию клавиш Ctrl+Alt+F7. Процедура регистрации здесь будет аналогична регистрации в *текстовом режиме*. С той лишь разницей, что после *идентификации пользователя* (правильно введенного *имени пользователя* и *пароля* ) на экране появится не *приглашение командной строки*, а графическая рабочая среда. Как именно она будет выглядеть - зависит от того, какая система используется и как она настроена.

Кроме того, что несколько пользователей (или несколько "копий" одного и того же пользователя) могут работать параллельно на разных *виртуальных консолях*, они могут параллельно зарегистрироваться и работать в разных графических средах. Обычно в стандартно настроенной Linux-системе можно организовать не менее трех графических консолей, работающих одновременно. Переключаться между ними можно при помощи сочетаний клавиш Ctrl+Alt+F7 - Ctrl+Alt+F9.

Чтобы переключиться из графического режима в одну из текстовых *виртуальных консолей*, достаточно нажать комбинацию клавиш Ctrl+Alt+FN, где " N " - номер необходимой *виртуальной консоли*.

**Простейшие команды**

Работа в Linux при помощи *командной строки* напоминает **диалог** с системой: *пользователь* вводит команды (реплики), получая от системы ответные реплики, содержащие сведения о произведенных операциях, дополнительные вопросы к пользователю, сообщения об ошибках или просто согласие выполнить следующую команду7.

Простейшая *команда* в Linux состоит из одного "слова" - названия программы, которую необходимо выполнить. Одну такую команду ( passwd ) Мефодий уже использовал для того, чтобы изменить свой *пароль*. Теперь Мефодий решил вернуться на одну из *виртуальных консолей*, на которой он зарегистрировался, и попробовать выполнить несколько простых команд:

[methody@localhost methody]$ whoami

methody

[methody@localhost methody]$

Пример 1.8. Команда whoami

Название этой команды происходит от английского выражения "Who am I?" ("Кто я?"). В ответ на эту команду система вывела только одно *слово*: "methody" и завершила свою работу, о чем свидетельствует вновь появившееся *приглашение командной строки*. *Программа* whoami возвращает название *учетной записи* того пользователя, от имени которого она была выполнена. Эта *команда* полезна в системах, в которых работает много разных пользователей, чтобы никто из них не мог по ошибке воспользоваться чужой *учетной записью*. Однако в *приглашении командной строки* зачастую указывается *имя пользователя* (как и в наших примерах), поэтому без команды whoami можно обойтись. Следующий пример демонстрирует программу, которая выдаст Мефодию уже больше полезной информации: who ("Кто"):

[methody@localhost methody]$ who

methody tty1 Sep 23 16:31 (localhost)

methody tty2 Sep 23 17:12 (localhost)

[methody@localhost methody]$

[methody@localhost methody]$ who am i

methody tty2 Sep 23 17:12 (localhost)

[methody@localhost methody]$

Пример 1.9. Команда who

*Команда* who выводит *список* пользователей, которые в настоящий момент зарегистрированы в системе (вошли в систему). Данная *программа* выводит по одной строке на каждого зарегистрированного пользователя: в первой колонке указывается *имя пользователя*, во второй - "точка входа" в систему, далее следует дата и время регистрации и *имя хоста*. Из выведенной who информации можно заключить, что в системе дважды зарегистрирован *пользователь* methody, который сначала зарегистрировался на первой *виртуальной консоли* ( tty1 ), а примерно через сорок минут - на второй ( tty2 ). Конечно, Мефодий и так это знает, однако *администратору* больших систем, когда пользователи могут зарегистрироваться со многих компьютеров и даже по сети, *программа* who может быть очень полезна. Могло создаться впечатление, что who - очень интеллектуальная *программа*, понимающая английский, но это не так. Из всех английских слов она понимает только сочетание "am i" - таким способом Мефодий узнал, за какой консолью он сейчас работает.

Еще одна *программа*, выдающая информацию о пользователях, работавших в системе в последнее время - last 8. Выводимые этой программой строки напоминают *вывод* программы who, с той разницей, что здесь перечислены и те пользователи, которые уже завершили работу:

[methody@localhost methody]$ last

methody tty2 localhost Thu Sep 23 17:12 still logged in

methody tty1 localhost Thu Sep 23 16:31 still logged in

cacheman ??? localhost Thu Sep 23 16:15 - 16:17 (00:01)

cacheman ??? localhost Thu Sep 23 16:08 - 16:08 (00:00)

cyrus ??? localhost Thu Sep 23 16:08 - 16:08 (00:00)

cyrus ??? localhost Thu Sep 23 16:08 - 16:08 (00:00)

reboot system boot 2.4.26-std-up-al Thu Sep 23 16:03 (04:13)

reboot system boot 2.4.26-std-up-al Thu Sep 23 16:03 (04:13)

Пример 1.10. Команда last

В этом примере Мефодий неожиданно обнаружил, кроме себя самого, неизвестных ему пользователей cacheman и cyrus - он точно знает, что не создавал *учетных записей* с такими именами. Это *псевдопользователи* (или системные пользователи) - специальные *учетные записи*, которые используются некоторыми программами. Поскольку эти "пользователи" регистрируются в системе без помощи монитора и клавиатуры, их "точка входа" в систему не определена (во второй колонке записано " ???"). В выводе программы last появляется даже *пользователь* reboot (перезагрузка). В действительности такой учетной записи нет, *программа* last таким способом выводит информацию о том, когда была загружена система.

**Выход из системы**

В строках, выведенных программой last, указан не только момент регистрации пользователя в системе, но и момент завершения работы. Можно представить Linux как закрытое помещение: чтобы начать работу, нужно сначала **войти** в систему (зарегистрироваться, пройти процедуру *идентификации* ), а когда работа закончена, следует из системы **выйти**. В том случае, если в систему вошло несколько пользователей, каждый из них должен выйти, завершив работу, причем не имеет значения, разные это пользователи или "копии" одного и того же.

Вход пользователя в систему означает, что нужно принимать и выполнять его команды и возвращать ему отчеты о выполненных действиях, например, предоставив ему *интерфейс* *командной строки*. *Выход* означает, что работа от имени данного пользователя завершена и более не следует принимать от него команды. Весь процесс взаимодействия пользователя с системой от момента регистрации до выхода называется ***сеансом работы*** . Причем если *пользователь* входит в систему несколько раз под одним и тем же именем, ему будут доступны несколько **разных** сеансов работы, не связанных между собой.

В наших примерах Мефодий зарегистрирован в системе дважды: на первой и второй *виртуальных консолях*. Чтобы завершить работу на любой из них, ему достаточно в соответствующей *командной строкe* набрать команду logout:

[methody@localhost methody]$ logout

Welcome to Some Linux / tty1

localhost login:

Пример 1.11. Команда logout

В ответ на эту команду вместо очередного приглашения командной строки возобновляется приглашение к *регистрации в системе*. На данной *виртуальной консоли* работа с Мефодием завершена, и теперь здесь снова может зарегистрироваться любой *пользователь*.

Есть и другой, еще более "немногословный" способ сообщить системе, что *пользователь* хочет завершить текущий *сеанс работы*. Нажав Alt+F2, Мефодий попадет на вторую *виртуальную консоль*, где все еще открыт *сеанс* для пользователя " methody ", и нажмет сочетание клавиш Ctrl+D, чтобы прекратить и этот *сеанс*. Нажатие комбинации клавиш Ctrl+D приводит не к передаче компьютеру очередного символа, а к закрытию текущего входного потока данных. В сущности, *командная оболочка* вводит команды пользователя с консоли, как если бы она читала их построчно из файла. Нажатие Ctrl+D сигнализирует ей о том, что этот "*файл*" закончился, и теперь ей неоткуда больше считывать команды. Такой способ завершения аналогичен явному завершению командной оболочки командой logout.

**Лекция 2:**

**Терминал и командная строка**

**Терминал**

Как было показано в предыдущей лекции (1), *основное средство* общения с Linux - системная клавиатура и экран монитора, работающий в *текстовом режиме*. Вводимый пользователем текст немедленно отображается на мониторе соответствующими знаками, однако может и не отображаться, как в случае ввода пароля. Для **управления** вводом используются некоторые **нетекстовые** клавиши на клавиатуре: **Backspace** (он же "*Забой*") - для удаления последнего введенного символа или **Enter** - для передачи команды системе. Нажатие на эти клавиши не приводит к отображению символа, вместо этого вводимый текст обрабатывается системой тем или иным способом:

[methody@localhost methody]$ data

-bash: data: command not found

[methody@localhost methody]$ date

Вск Сен 12 13:59:36 MSD 2004

2.1. Сообщение об ошибке

Вначале Мефодий ошибся, и вместо команды date написал data. В ответ он получил **сообщение об ошибке**, поскольку такой команды система не понимает. Затем (этого не видно в примере, но случилось именно так!) он **снова** набрал data, но вовремя одумался и, нажав клавишу **Backspace**, удалил последнее "a", вместо которого ввел "e", превратив data в date. Такая *команда* в системе есть, и на экране возникла текущая дата.

Диалог машины и пользователя неспроста выглядит как обмен текстами. Именно письменную речь используют люди для постановки и описания решения задач в заранее определенном, формализованном виде. Поэтому и задача управления системой может целиком быть представлена и решена в виде формализованного текста - программы. При этом машине отводится роль аккуратного исполнителя программы, а человеку - роль автора. Кроме того, человек анализирует текст, получаемый от системы: запрошенную им информацию и текст **сообщения** - текст, описывающий *состояние системы* в процессе решения задачи (например, *сообщение об ошибке* "*command* *not* found").

Текстовый принцип работы с машиной позволяет отвлечься от конкретных частей компьютера, вроде системной клавиатуры и видеокарты с монитором, рассматривая единое **оконечное устройство**, посредством которого пользователь **вводит** текст и передает его системе, а система **выводит** необходимые пользователю данные и сообщения. Такое устройство называется ***терминалом*** . В общем случае ***терминал*** - это **точка входа** пользователя в систему, обладающая способностью передавать текстовую информацию. *Терминалом* может быть отдельное *внешнее устройство*, подключаемое к компьютеру через *порт* последовательной передачи данных (в персональном компьютере он называется "*COM* port"). В роли *терминала* может работать (с некоторой поддержкой со стороны системы) и *программа* (например, xterm или ssh ). Наконец, *виртуальные консоли* Linux - тоже *терминалы*, только организованные программно с помощью подходящих устройств современного компьютера.

***Терминал*** - устройство последовательного ввода и вывода символьной информации, способное воспринимать часть символов как *управляющие* для редактирования ввода, *сигналов* и т. п. Используется для взаимодействия пользователя и системы.

Для приема и передачи текста *терминалу* достаточно уметь принимать и передавать **символы**, из которых этот текст состоит. Более того, **желательно**, чтобы единицей обмена с компьютером был именно один *байт* (один ascii-символ). Тогда каждая буква, набранная на клавиатуре, может быть передана системе для обработки, если понадобится. С другой стороны, типичный способ управления системой в Linux - работа в *командной строке* - требует **построчного** режима работы, когда набранный текст передается компьютеру только после нажатия клавиши **Enter** (что соответствует символу конца строки). Размер такой строки в байтах предугадать, конечно, нельзя, поэтому, работающий в построчном режиме *терминал*, ничем, *по* сути, не отличается от *терминала*, работающего в посимвольном режиме - за исключением того, что *активность* системы *по* обработке приходящих с этого *терминала*данных падает в несколько раз (обмен ведется не одиночными байтами, а целыми строками).

Свойство *терминала* передавать только символьную информацию приводит к тому, что некоторые из передаваемых символов должны восприниматься не как текстовые, а как *управляющие* (например, символы, возвращаемые клавишами **Backspace** и **Enter** ). На самом деле *управляющих символов* больше: часть из них предназначена для экстренной передачи команд системе, часть - для редактирования вводимого текста. Многие из этих символов не имеют **специальной** клавиши на клавиатуре, поэтому их необходимо извлекать с помощью ***клавиатурного модификатора*** **Ctrl** .

Команды, подаваемые с клавиатуры с помощью **Ctrl**, как и символы, передаваемые при этом системе, принято обозначать знаком " ^ ", после которого следует имя клавиши, нажимаемой вместе с **Ctrl**: например, одновременное нажатие **Ctrl** и " А " обозначается " ^A ".

Так, для завершения работы программы cat, которая построчно считывает данные с клавиатуры и выводит их на *терминал*, можно воспользоваться командой " ^C " или " ^D ":

[methody@localhost methody]$ cat

Any Text

Any Text

^C

[methody@localhost methody]$ cat

Any Text agaim^[[Dn

Any Text again

^D

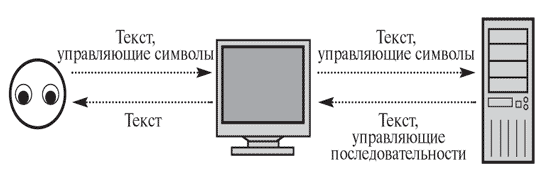
[methody@localhost methody]$

2.2. Как завершить работу cat?

Одну строчку вида "Any Tex..." Мефодий вводит с клавиатуры (что отображается на экране), и после того, как Мефодий нажмет **Enter**, она немедленно выводится программой cat (что тоже отображается на экране). С каждой последующей строкой *программа* cat поступила бы аналогично, но в примере Мефодий оба раза завершил работу программы, в первом случае нажав ", ^C ", а во втором - " ^D ". Команды оказали одинаковый эффект, но работают они *по*-разному: " ^C " посылает программе, которая считывает с клавиатуры *сигнал* аварийного прекращения работы, а " ^D " сообщает ей, что ввод данных с клавиатуры закончен и можно продолжать работу (поскольку *программа* cat больше ничего не делает, она завершается самостоятельно, естественным путем). Можно считать, что " ^C " - это сокращение от "*Cancel*", а " ^D " - от "Done".

В примере не показано, как, набирая первый cat, Мефодий вновь ошибся и написал ccat вместо cat. Чтобы исправить положение, он воспользовался клавишами со стрелочками: с помощью клавиши " Стрелка влево " подвел *курсор* к одному из " c " и нажал **Backspace**, а затем **Enter**. В режиме ввода **команды** это ему удалось, а при передаче данных программе cat клавиша " Стрелка влево " не сдвинула *курсор*, а передала целую последовательность символов: " ^[ ", " [ " и " D ". Дело в том, что на клавиатуре может быть так много разных нетекстовых клавиш, что на них не хватает ограниченного количества **разных***управляющих символов*. Поэтому большинство нетекстовых клавиш возвращают так называемую *управляющую последовательность*, которая начинается *управляющим символом* (как правило - Escape, т. е. " ^[ "), за которым следует строго определенное число обычных символов (для клавиши Стрелка влево - " [ " и " D ").

То же самое можно сказать и о **выводе***управляющих последовательностей* на *терминал*. Современный *терминал* имеет довольно много возможностей помимо простого вывода текста: перемещать *курсор* *по* всему экрану (чтобы вывести текст), удалять и вставлять строки на экране, использовать цвет и т. п. Всем этим заведуют *управляющие последовательности*, которые при выводе на экран *терминала* не отображаются как текст, а выполняются заранее заданным способом. В некоторых случаях *управляющие последовательности*, возвращаемые клавишами, совпадают с теми, что управляют поведением *терминала*. Поэтому-то Мефодий и не увидел " Any Text agaim^[[Dn " в выдаче cat: " ^[[D " при выводе на *терминал* перемещает *курсор* на одну позицию влево, так что было выведено " Any Text agaim ", затем *курсор* встал прямо над " m " и поверх него было выведено " n ". Если бы *терминал* имел вместо дисплея *печатающее устройство*, в этом месте обнаружилось бы нечто, состоящее из начертаний " m " и " n " 1.



**Рис. 2.1.**Интерфейс командной строки. Взаимодействие пользователя с компьютером посредством терминала.

Требования к *терминалу* как к точке входа пользователя в систему весьма невысоки. Формально говоря, *терминал* должен удовлетворять трем обязательным требованиям и одному необязательному. *Терминал* должен уметь:

1. передавать текстовые данные от пользователя системе;
2. передавать от пользователя системе немногочисленные управляющие команды;
3. передавать текстовые данные от системы пользователю;
4. (необязательно) интерпретировать некоторые данные, передаваемые от системы пользователю, как *управляющие последовательности* и соответственно обрабатывать их.

Ограничения на *интерфейс* напрямую не сказываются на эффективности работы пользователя в системе. Однако чем меньше требований к интерфейсу, тем важнее разумно его организовать. Любое взаимодействие может быть описано с трех точек зрения: во-первых, какую задачу решает *пользователь* ( **что** он хочет от системы); во-вторых, **как** он формулирует задачу в доступном пониманию системы виде; в-третьих, какими средствами он пользуется при взаимодействии с системой. В частности, текстовый *интерфейс* удобно рассматривать с точки зрения предоставляемого им **языка** общения с машиной: во-первых, описанием этого языка задается *диапазон* решаемых с его помощью задач, а во-вторых, слова этого компьютерного языка (называемые в программировании **операторами** ) предоставляют способ решения пользовательских задач (в виде небольших программ-сценариев). Команды, помогающие пользователю быстро и эффективно обмениваться с машиной предложениями на этом языке, и будут третьей составляющей интерфейса *командной строки*.

**Командная строка**

Основная среда взаимодействия с Linux - *командная строка*. Суть ее в том, что каждая строка, передаваемая пользователем системе, - это *команда*, которую та должна выполнить. Пока не нажата кнопка **Enter**, строку можно редактировать, затем она отсылается системе:

[methody@localhost methody]$ cal

Сентября 2004

Вс Пн Вт Ср Чт Пт Сб

1 2 3 4

5 6 7 8 9 10 11

12 13 14 15 16 17 18

19 20 21 22 23 24 25

26 27 28 29 30

[methody@localhost methody]$ echo Hello, world!

Hello, world!

2.3. Команды echo и cal

*Команда* cal выводит календарь на текущий месяц, а *команда* echo просто выводит на *терминал* все, что следовало в *командной строке* после нее. Получается, что одну и ту же команду можно использовать с разными параметрами (или аргументами ), причем параметры эти изменяют поведение команды. Здесь Мефодий захотел посмотреть календарь за март 2005 года, для чего и передал команде cal два параметра - 3 и 2005:

[methody@localhost methody]$ cal 3 2005

Марта 2005

Вс Пн Вт Ср Чт Пт Сб

1 2 3 4 5

6 7 8 9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26

27 28 29 30 31

2.4. Команда cal с параметрами

В большинстве случаев при разборе *командной строки* первое *слово* считается именем команды, а остальные - ее параметрами. Более подробно о разборе *командной строки* и работе с ней рассказано в разделе " *Интерпретатор командной строки* (*shell*)" и в лекции 7.

**Подсистема помощи**

Пока же Мефодий решил, что узнал о *командной строке* достаточно для того, чтобы воспользоваться **главными** командами Linux (*по* частоте их употребления при изучении системы) - man и info.

Работать с Linux, не заглядывая в документацию, практически невозможно. На это способны только седые аксакалы, у которых все нужные знания не то что в голове - в кончиках пальцев, и новички. Всем прочим **настоятельно** рекомендуем, приступая к работе, а тем более - к изучению Linux, пользоваться всеми доступными *руководствами*.

Все *утилиты*, все *демоны* Linux, все функции *ядра* и *библиотек*, структура большинства *конфигурационных файлов*, наконец, многие умозрительные, но важные понятия системы описаны либо в *руководствах*, либо в *info*-страницах, либо, на худой конец, в несистематизированной сопроводительной документации. Поэтому от пользователя системы не требуется **заучивать** все возможные варианты взаимодействия с ней. Достаточно **понимать** основные принципы ее устройства и уметь находить справочную информацию. Эйнштейн говорил так: "Зачем запоминать то, что всегда можно посмотреть в справочнике?"

**Страницы руководства (man)**

Больше всего полезной информации содержится в страницах ***руководства (manpages)***, для краткости мы будем называть их просто " *руководство* ". Каждая страница посвящена какому-нибудь одному объекту системы. Для того чтобы посмотреть страницу *руководства*, нужно дать команду системе man **объект**:

[methody@localhost methody]$ man cal

CAL(1) BSD General Commands Manual CAL(1)

NAME

cal - displays a calendar

SYNOPSIS

cal [-smjy13] [[month] year]

DESCRIPTION

Cal displays a simple calendar. If arguments

are not specified, the current month is

displayed.

The options are as follows:

. . .

2.5. Просмотр страницы руководства

Страница *руководства* занимает, как правило, больше одной страницы **экрана**. Для того чтобы читать было удобнее, man запускает программу постраничного просмотра текстов - less. Управлять программой less просто: страницы перелистываются пробелом, а когда читать надоест, надо нажать " q " (Quit). Перелистывать страницы можно и клавишами Page Up/Page Down, для сдвига на одну строку вперед можно применять Enter или стрелку вниз, а на одну строку назад - стрелку вверх. Переход на начало и конец текста выполняется по командам " g " и " G " соответственно (Go). Полный список того, что можно делать с текстом в less, выводится по команде " H " (Help).

Страница *руководства* состоит из **полей** - стандартных разделов, с разных сторон описывающих заинтересовавший Мефодия объект - команду cal. В поле NAME содержится краткое описание объекта (такое, чтобы его назначение было понятно с первого взгляда). В поле *SYNOPSIS* дается формализованное описание способов использования объекта (в данном случае - того, как и с какими параметрами запускать команду cal ). Как правило, в квадратные скобки в этом поле заключены **необязательные** параметры команды, которые можно ей передать, а можно и опустить. Например, строка " [[month] year] " означает, что в этом месте *командной строки* параметров у команды может не быть вообще, может быть указан год или пара - месяц и год. Наконец, текст в поле DESCRIPTION - это развернутое описание объекта, достаточное для того, чтобы им воспользоваться.

Одно из самых важных полей *руководства* находится в конце текста. Если в процессе чтения NAME или DESCRIPTION пользователь понимает, что не нашел в *руководстве* того, что искал, он может захотеть посмотреть, а есть ли **другие***руководства* или иные источники информации по той же теме. Список таких источников содержится в поле SEE ALSO:

[methody@localhost methody]$ man man

. . .

SEE ALSO

apropos(1), whatis(1), less(1), groff(1), man.conf(5).

. . .

2.6. Поле SEE ALSO руководства

До этого поля Мефодий добрался с помощью уже известной команды " G ". Не то чтобы ему неинтересно было читать *руководство* по man, скорее наоборот: им двигала любознательность. В поле SEE ALSO обнаружились ссылки на *руководства* по less, groff (программе форматирования страницы *руководства* ), структуре *конфигурационного файла* для man, а также по двум сопутствующим командам с такими говорящими названиями "Apropos" и "whatis"2, что Мефодий немедленно применяет одну команду к имени другой, даже не заглядывая в документацию. Так **ни в коем случае** не следует делать! А что если запущенная программа начнет с того, что сотрет все файлы в Вашем каталоге?

[methody@localhost methody]$ whatis apropos

apropos (1) - search the whatis database for strings

[methody@localhost methody]$ man apropos

apropos(1) apropos(1)

NAME

apropos - search the whatis database for strings

. . .

2.7. Вызов whatis

На этот раз Мефодию повезло: команда *whatis* не делает ничего разрушительного. Как и команда *apropos*, *whatis* ищет подстроку в некоторой базе данных, состоящей из полей NAME всех страниц помощи в системе. Различие между ними в том, что *whatis* - только среди имен объектов (в левых частях полей NAME ), а *apropos* - по всей базе. В результате у *whatis* получается список кратких описаний объектов с **именами**, включающими искомое слово, а у *apropos* - список, в котором это слово упоминается. Для того, чтобы это узнать, все равно пришлось один раз прочесть документацию.

В системе может встретиться несколько объектов разного типа, но с одинаковым названием. Часто совпадают, например, имена *системных вызовов* (функций *ядра* ) и программ, которые пользуются этими функциями из *командной строки* (т. н. *утилит* ):

[methody@localhost methody]$ whatis passwd

passwd (1) - update a user's authentication tokens(s)

passwd (5) - password file

passwd (8) - manual page for passwd wrapper version 1.0.5

2.8. Руководства с одинаковыми именами

Описания объектов, выводимые *whatis*, отличаются числом в скобках - номером **раздела**. В системе *руководств* Linux - девять разделов, каждый из которых содержит страницы *руководства* к объектам определенного типа. Все разделы содержат по одному *руководству* с именем "intro", в котором в общем виде и на примерах рассказано, какие объекты имеют отношение к данному разделу:

george@localhost:~> whatis intro

intro (1) - Introduction to user commands

intro (2) - Introduction to system calls

intro (3) - Introduction to library functions

intro (4) - Introduction to special files

intro (5) - Introduction to file formats

intro (6) - Introduction to games

intro (7) - Introduction to conventions and

miscellany section

intro (8) - Introduction to administration and

privileged commands

intro (9) - Introduction to kernel interface

2.9. Руководства intro

Вот названия разделов в переводе на русский:

1. Команды пользователя.
2. *Системные вызовы* (пользовательские функции *ядра* Linux; *руководства* рассчитаны на программиста, знающего язык Си).
3. Библиотечные функции (функции, принадлежащие всевозможным *библиотекам* подпрограмм; *руководства* рассчитаны на программиста, знающего язык Си).
4. *Внешние устройства* и работа с ними (в Linux они называются **специальными файлами**, см. лекцию 10).
5. Форматы различных стандартных файлов системы (например, *конфигурационных* ).
6. Игры, безделушки и прочие вещи, не имеющие системной ценности.
7. Теоретические положения, договоренности и все, что не может быть классифицировано.
8. Инструменты администратора (часто недоступные обычному пользователю).
9. Интерфейс *ядра* ( **внутренние** функции и структуры данных *ядра* Linux, необходимые только системному программисту, исправляющему или дополняющему *ядро* ).

В частности, пример с passwd показывает, что в системе " Some Linux ", которую использует Мефодий, есть программа passwd (именно с ее помощью Мефодий поменял себе пароль в предыдущей лекции), файл passwd, содержащий информацию о пользователях, и **администраторская** программа passwd, обладающая более широкими возможностями. По умолчанию man просматривает все разделы и показывает **первое найденное***руководство* с заданным именем. Чтобы посмотреть *руководство* по объекту из определенного раздела, необходимо в качестве первого параметра команды man указать номер раздела:

[methody@localhost methody]$ man 8 passwd

PASSWD(8) System Administration Utilities PASSWD(8)

. . .

[methody@localhost methody]$ man -a passwd

PASSWD(1) Some Linux PASSWD(1) . . .

PASSWD(8) System Administration Utilities PASSWD(8) . . .

PASSWD(5) Linux Programmer's Manual PASSWD(5) . . .

2.10. Выбор среди страниц руководства с одинаковым именем

Если в качестве первого параметра man использовать " -a ", будут последовательно выданы **все***руководства* с заданным именем. **Внутри** страниц *руководства* принято непосредственно после имени объекта ставить в круглых скобках номер раздела, в котором содержится *руководство* по этому объекту: *man (1)*, less(1), passwd(5) и т. д.

**Info**

Другой источник информации о Linux и составляющих ее программах - справочная подсистема info. Страница *руководства*, несмотря на обилие ссылок **различного** типа, остается "линейным" текстом, структурированным только логически. Документ info структурирован прежде всего **топологически** - это настоящий гипертекст, в котором множество небольших страниц объединены в дерево. В каждом разделе документа info всегда есть оглавление, из которого можно перейти сразу к нужному подразделу, откуда всегда можно вернуться обратно. Кроме того, info-документ можно читать и как **непрерывный** текст, поэтому в каждом подразделе есть ссылки на предыдущий и последующий подразделы:

[methody@localhost methody]$ info info

File: info.info, Node: Top, Next: Getting Started, Up: (dir)

Info: An Introduction

. . .

\* Menu:

\* Getting Started:: Getting started using an Info reader.

\* Expert Info:: Info commands for experts.

\* Creating an Info File:: How to make your own Info file.

\* Index:: An index of topics, commands, and variables.

. . .

--zz-Info: (info.info.bz2)Top, строк: 24 --All------------

Welcome to Info version 4.6.

Type ? for help, m for menu item.

2.11. Просмотр info-документа

Программа **info** использует весь экран: на большей его части она показывает текст документа, а первая и две последних строки служат для ориентации в его структуре.

Одна или несколько страниц, которые можно перелистывать клавишей **Пробел** или **Page Up/Page Down** - это *узел (node)*. ***Узел*** содержит обычный текст и *меню (menu)* - список ссылок на другие *узлы*, лежащие в дереве на более низком уровне. Ссылки внутри документа имеют вид **"\* имя\_узла::"** и перемещать по ним курсор можно клавишей **Tab**, а переходить к просмотру выбранного *узла* - клавишей **Enter**. Вернуться к предыдущему просмотренному *узлу* можно клавишей " l " (oт "Last"). И, главное, выйти из программы info можно, нажав " q " ("Quit"). Более подробную справку об управлении программой info можно в любой момент получить у самой info, нажав " ?".

*Узлы*, составляющие документ info, можно просматривать и подряд, один за другим (с помощью команд " n ", Next, и " p ", Previous), однако это используется нечасто. В верхней строке экрана info показывает имя текущего *узла*, имя следующего *узла* и имя родительского (или верхнего) *узла*, в котором находится ссылка на текущий. Показанные Мефодию имя *узла*Top и имя верхнего *узла*( dir ) означают, что просматривается **корневой** *узел* документа, выше которого - только каталог со списком всех info-деревьев. В нижней части экрана расположена строка с информацией о текущем *узле*, а за ней - строка для ввода длинных команд (например, для поиска текста с помощью команды " / ").

Команде info можно указывать в параметрах всю цепочку *узлов*, приводящую к тому или иному разделу документации, однако это бывает нужно довольно редко:

[methody@localhost methody]$ info info "Getting Started" Help-Q

File: info.info, Node: Help-Q, Prev: Help-Int,

Up: Getting Started

Quitting Info

. . .

2.12. Просмотр определенного узла info-документа

Сам ли Мефодий это придумал, или подсказал кто, но совершенно правильно было заключить в кавычки имя *узла* "Getting Started" - в этом случае info искала *узел* по "адресу" " info -> Getting Started -> Help-Q ". Если бы команда имела вид info info Getting Started Help-Q, то "адрес" получился бы неправильный: " info -> Getting -> Started -> Help-Q ". Ничего таинственного в этом нет, и уже к концу лекции станет понятно, в чем здесь дело (см. раздел "Слова и *разделители* ").

**RTFM**

Оказывается, использование кавычек Мефодий придумал не сам: спросил у товарища, опытного пользователя Linux *по* фамилии Гуревич. Гуревич охотно показал, где ставить кавычки, а вот объяснять, что они делают, отказался: "Там отличное *руководство*! Читай!" Документация в Linux играет важнейшую роль. Решение **любой** задачи должно начинаться с изучения *руководств*. Не стоит жалеть на это времени. Даже если рядом есть опытный *пользователь* Linux, который, возможно, **знает** ответ, не стоит беспокоить его **сразу же**. Возможно, даже зная, **что** нужно сделать, он не помнит **как** именно - и поэтому (а также потому, что он - опытный *пользователь*) начнет с изучения *руководства*. Это - закон, у которого даже собственное название: ***RTFM***, что означает "Read The Fine Manual".

***RTFM, Read The Fine Manual*** - правило, согласно которому решение любой задачи надо начинать с изучения документации.

Слова Гуревича - практически дословный перевод этой фразы, так что ее смысл и происхождение очевидны. Linux рассчитан в основном на тех, кто хочет знать, как им пользоваться.

*Руководство* - это не учебник, а скорее справочник. В нем содержится *информация*, **достаточная** для освоения описываемого объекта, но никаких **обучающих** приемов, никаких определений, повторений и выделения главного в нем обычно нет. Тем более не допускается **усечение***руководства* с целью представить небольшие *по* объему, но наиболее важные сведения. Так принято в учебниках, причем сведения раскрываются и объясняются очень подробно, а остальные присутствуют в виде ссылки на документацию для профессионалов. Страницы *руководств* - и есть эта самая документация для профессионалов.

*Руководство* чаще всего читает человек, который уже знает, о чем оно.

Это не значит, что из *руководства* нельзя понять, как, например, пользоваться командой в простейших случаях. Напротив, часто встречается *поле* EXAMPLES, которое как раз и содержит **примеры** использования команды в разных условиях. Однако все это призвано не **научить**, а раскрыть смысл, пояснить сказанное в других полях. Мефодий нашел описание работы двойных кавычек в *руководстве* *по* sh, однако понял из него далеко не все - главным образом, потому, что встретил слишком много незнакомых терминов.

Система info может содержать больше, чем man, поэтому в нее часто включают и учебники (принято называть учебник термином "*tutorial*"), и "howto" (примеры постановки и решения типовых задач), и даже статьи *по* теме. Таким образом, *info*-документ может стать, в отличие от страницы *руководства*, **полным сводом** сведений. Разработка такого документа - дело трудоемкое, поэтому далеко не все объекты системы им сопровождаются. Кроме того, и прочесть большой *info*-документ **целиком** зачастую невозможно. Поэтому имеет смысл начинать именно с *руководства*, а если его недостаточно - изучать info.

Если некоторый *объект* системы не имеет документации ни в формате man, ни в формате info, это нехорошо. В этом случае можно надеяться, что при нем есть **сопроводительная документация**, не имеющая, увы, ни стандартного формата, ни тем более ссылок на *руководства* *по* другим объектам системы. Такая документация (равно как и примеры использования объекта) обычно помещается в каталог **/usr/share/doc/имя\_объекта**.

Документация в подавляющем большинстве случаев пишется на простом английском языке. Если английский - не родной язык для автора документации, она будет только проще. Традиция писать *по*-английски идет от немалого вклада США в развитие компьютерной науки вообще и Linux в частности. Кроме того, английский становится языком международного общения во всех областях, не только в компьютерной. Необходимость писать на языке, который будет более или менее понятен большинству пользователей, объясняется постоянным развитием Linux. Дело не в том, что страницу *руководства* нельзя перевести, а в том, что ее придется переводить **всякий раз**, когда изменится описываемый ею *объект*! Например, *выход* новой версии программного продукта сопровождается изменением его возможностей и особенностей работы, а следовательно, и новой версией документации. Тогда **перевод** этой документации превращается в "moving *target*", сизифов труд.

**Ключи**

Работая в системе и изучая *руководства*, Мефодий заметил, что параметры команд можно отнести к двум различным категориям. Некоторые параметры имеют **собственный** смысл: это имена файлов, названия разделов и объектов в man и info, числа и т. п. Другие параметры собственного смысла не имеют, их *значение* можно истолковать, лишь зная, к какой команде они относятся. Например, *параметр* " -a " можно передать не только команде man, но и команде who, и команде last, при этом значить для них он будет разное. Такого рода параметры называются **модификаторами выполнения** или *ключами* (options):

[methody@localhost methody]$ date

Вск Сен 19 23:01:17 MSD 2004

[methody@localhost methody]$ date -u

Вск Сен 19 19:01:19 UTC 2004

2.13. Команда date с ключом

Для решения разных задач одни и те же действия необходимо выполнять слегка *по*-разному. Например, для синхронизации *работ* в разных точках земного шара лучше использовать единое для всех время (*по* Гринвичу), а для организации собственного рабочего дня - местное время (с учетом сдвига *по* часовому поясу и разницы зимнего и летнего времени). И то, и другое время показывает *команда* date, только для работы *по* Гринвичу ей нужен дополнительный *параметр*- *ключ* " -u " (он же " --universal ").

**Однобуквенные ключи**

Для формата *ключей* нет жесткого стандарта, однако существуют договоренности, нарушать которые в наше время уже неприлично. Во-первых, если параметр начинается на " - ", это - ***однобуквенный ключ*** . За " - ", как правило, следует один символ, чаще всего - буква, обозначающая действие или свойство, которое этот *ключ* придает команде. Так проще отличать *ключи* от других параметров - и пользователю при наборе *командной строки*, и программисту, автору команды:

[methody@localhost methody]$ who -m

methody tty1 Sep 20 13:56 (localhost)

[methody@localhost methody]$ cal -m

Сентября 2004

Пн Вт Ср Чт Пт Сб Вс

1 2 3 4 5

6 7 8 9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26

27 28 29 30

2.14. Использование ключа "-m" в разных командах

Во-вторых, для who *ключ* " -m " означает " Me ", то есть "Я", и в результате who работает похоже на whoami 3. А для cal *ключ* " -m " - это команда выдать календарь, считая первым днем понедельник ("Monday"), как это принято в России.

Свойство *ключа* должно быть, с одной стороны, предельно коротким, а с другой стороны - информативным, называется ***аббревиативностью*** . Не только *ключи*, но и имена наиболее распространенных команд Linux обладают этим свойством.

В-третьих, иногда *ключ* изменяет поведение команды таким образом, что меняется и толкование параметра, следующего в *командной строке* за этим *ключом*. Выглядит это так, будто *ключ* **сам** получает параметр, поэтому *ключи* такого вида называются ***параметрическими*** . Как правило, их параметры - имена файлов различного применения, числовые характеристики и прочие **значения**, которые нужно передать команде:

[methody@localhost methody]$ info info "Expert info" Cross-refs Help-Cross -o text

info: Запись ноды (info.info.bz2)Help-Cross...

info: Завершено.

[methody@localhost methody]$ cat text -n

1 File: info.info, Node: Help-Cross, Up: Cross-refs

2

3 The node reached by the cross reference in Info

4 ------------------------------------------------

. . .

2.15. Использование info -o

Здесь info запустилась не в качестве **интерактивной** программы, а как обработчик info-документа. Результат работы - текст *узла* info -> Expert info -> Cross-refs -> Help-Cross4, программа поместила в файл text. А программа cat вывела содержимое этого файла на *терминал*, пронумеровав все строки (по просьбе *ключа* " **-n** ", "number").

Теперь стало более или менее понятно, что означают неудобочитаемые строки в поле *SYNOPSIS* *руководства*. Например [ -smjy13 ] из *руководства* по cal (5) говорит о том, что команду можно запускать с необязательными *ключами* " -s ", " -m ", " -j ", " -y ", " -1 " и " -3 ".

В-четвертых, есть некоторые менее жесткие, но популярные договоренности о **значении***ключей*. *Ключ* " -h " ("Help") обычно (но, увы, не всегда) заставляет команды выдать **краткую справку** (нечто похожее на *SYNOPSIS*, иногда с короткими пояснениями). Если указать " - " вместо имени **выходного** файла в соответствующем *параметрическом ключе* (нередко это *ключ* " -o "), вывод будет производиться на терминал5. Наконец, бывает необходимо передать команде параметр, а не *ключ*, начинающийся с " - ". Для этого нужно использовать *ключ* " -- ":

[methody@localhost methody]$ info -o -filename-with-

info: Запись ноды (dir)Top...

info: Завершено.

[methody@localhost methody]$ head -1 -filename-with-

head: invalid option -- f

Попробуйте `head --help' для получения более

подробного описания.

[methody@localhost methody]$ head -1 -- -filename-with-

File: dir Node: Top This is the top of the INFO tree

2.16. Параметр-не ключ, начинающийся на "-"

Здесь Мефодий сначала создал файл -filename-with-, а потом пытался посмотреть его первую строку (команда head **-количество\_строк имя\_файла** выводит первые **количество\_строк** из указанного файла). *Ключ* " -- " (первый " - " - признак *ключа*, второй - сам *ключ* ) обычно запрещает команде интерпретировать все последующие *параметры командной строки* как *ключи*, независимо от того, начинаются они на " - " или нет. Только после " -- " head согласилась с тем, что -filename-with- - это имя файла.

**Полнословные ключи**

*Аббревиативность* *ключей* трудно соблюсти, когда их у команды слишком много. Некоторые буквы латинского алфавита (например, " s " или " o ") используются очень часто, и могли бы служить сокращением сразу нескольких команд, а некоторые (например, " z ") - редко, под них и название-то осмысленное трудно придумать. На такой случай существует другой, **полнословный** формат: *ключ* начинается на **два** знака " - ", за которыми следует полное имя обозначаемой им сущности. Таков, например, *ключ* " --help " (аналог " -h "):

[methody@localhost methody]$ head --help

Использование: head [КЛЮЧ]... [ФАЙЛ]...

Print the first 10 lines of each FILE to standard output.

With more than one FILE, precede each with a header

giving the file name.

With no FILE, or when FILE is -, read standard input.

Аргументы, обязательные для длинных ключей, обязательны

и для коротких.

-c, --bytes=[-]N print the first N bytes of each file;

with the leading '-', print all but

the last N bytes of each file

-n, --lines=[-]N print the first N lines instead of

the first 10;

the leading '-', print all but

the last N lines of each file

-q, --quiet, --silent не печатать заголовки с

именами файлов

-v, --verbose всегда печатать заголовки с

именами файлов

--help показать эту справку и выйти

--version показать информацию о версии и выйти

N may have a multiplier suffix: b 512, k 1024, m 1024\*1024.

Об ошибках сообщайте по адресу.

2.17. Ключ -help

Мефодий сделал то, о чем просила его *утилита* head. Обращает на себя внимание то, что некоторые *ключи* head имеют и *однобуквенный*, и *полнословный* формат, а некоторые - только *полнословный*. Так обычно и бывает: часто используемые *ключи* имеют аббревиатуру, а редкие - нет. **Значения***параметрических* *полнословных ключей* принято передавать не следующим параметром *командной строки*, а с помощью конструкции " **=значение** " непосредственно после *ключа*.

**Интерпретатор командной строки (shell)**

В Linux нет отдельного объекта под именем "система". Система - она на то и система, чтобы состоять из многочисленных компонентов, взаимодействующих друг с другом. Главный из системных компонентов - *пользователь*. Это он командует машиной, а та его команды выполняет. В *руководствах* второго и третьего разделов описаны *системные вызовы* (функции *ядра* ) и библиотечные функции. Они-то и есть непосредственные команды системе. Правда, воспользоваться ими можно только написав программу (чаще всего - на языке *Си*), нередко - довольно сложную. Дело в том, что функции *ядра* реализуют низкоуровневые *операции*, и для решения даже самой простой задачи пользователя необходимо выполнить несколько таких операций, преобразуя результат работы одной для нужд другой. Возникает необходимость выдумать для пользователя другой - более высокоуровневый и более удобный- язык управления системой. Все команды, которые использовал Мефодий в работе, были частью именно этого языка.

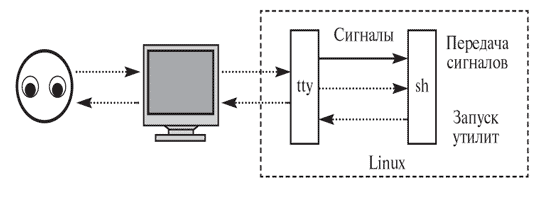
Из этого несложно заключить, что **обрабатывать** эти команды, превращать их в последовательность *системных* и библиотечных вызовов должна тоже какая-нибудь специальная *программа*, и именно с ней непрерывно ведет диалог *пользователь* сразу после входа в систему. Так оно и оказалось - *программа* эта называется *интерпретатор командной строки* или *командная оболочка* ("*shell*"). "Оболочкой" она названа как раз потому, что все *управление системой* идет как бы "изнутри" нее: *пользователь* общается с нею на удобном ему языке (с помощью текстовой *командной строки* ), а она общается с другими частями системы на удобном им языке (вызывая запрограммированные функции).

Таким образом, упомянутые выше правила разбора *командной строки* - это правила, действующие именно в *командном интерпретаторе*: *пользователь* вводит с *терминала* строку, *shell* считывает ее, иногда - преобразует *по* определенным правилам, получившуюся строку разбивает на команду и параметры, а затем выполняет команду, передавая ей эти параметры. *Команда*, в свою *очередь*, анализирует параметры, выделяет среди них *ключи* и делает то, о чем ее попросили, попутно выводя на *терминал* данные для пользователя, после чего завершается. *По* завершении команды возобновляется работа "отступившего на задний план" *командного интерпретатора* - он снова считывает *командную строку*, разбирает ее, вызывает команду... Так продолжается до тех пор, пока *пользователь* не скомандует оболочке **завершиться** самой (с помощью logout или *управляющего символа* " ^D ", который для *shell* значит то же, что и для других программ: больше с *терминала* ввода не будет).

Конечно, *командных интерпретаторов* в Linux несколько. Самый простой из них, появившийся в ранних версиях *UNIX*, назывался sh, или "Bourne *Shell*" - *по* имени автора, Стивена Борна (Stephen Bourne). Со временем его везде, где только можно, заменили на более мощный, bash, "Bourne Again *Shell*" 6. bash превосходит sh во всем, особенно в возможностях редактирования *командной строки*. Помимо sh и bash в системе может быть установлен "The Z *Shell*", zsh, **самый** мощный на сегодня *командный интерпретатор* (шутка ли, 22 тысячи строк документации!), или tcsh, обновленная и тоже очень мощная версия старой оболочки "C *Shell*", *синтаксис* команд которой похож на *язык программирования* *Си*.

Когда Гуревич добавлял учетную *запись* Мефодия в систему, он не стал спрашивать, какой *командный интерпретатор* ему нужен, потому что знал: для новичка имя *командного интерпретатора* - пустой звук. Тем не менее имя оболочки, запускаемой для пользователя сразу после входа в систему - так называемый *стартовый командный интерпретатор* ( login shell ), - это часть пользовательской учетной записи, которую *пользователь* может изменить командой chsh (*change* *shell*).

Какая бы задача, связанная с управлением системой, ни стояла перед пользователем Linux, она должна иметь решение в терминах *командного интерпретатора*. Фактически, решение пользовательской задачи - это описание ее на языке *shell*. Язык общения пользователя и *командного интерпретатора* - это высокоуровневый *язык программирования*, дополненный, с одной стороны, средствами организации взаимодействия команд и системы, а с другой стороны - средствами взаимодействия с пользователем, облегчающими и ускоряющими работу с *командной строкой*.



**Рис. 2.2.**Интерфейс командной строки. Издание второе, переработанное и дополненное. Взаимодействие пользователя с компьютером посредством терминала и оболочки.

**Команды и утилиты**

[methody@localhost methody]$ apropos s

. . . (четыре с половиной тысячи строк!)

2.18. Бессмысленная команда

Одного неудачного запуска *apropos* Мефодию было достаточно для того, чтобы понять: команд в Linux **очень** много. Ему пришло в голову, что никакая программа - пусть даже и оболочка - не может **самостоятельно** разбираться во всех задокументированных командах. Кроме того, Гуревич называл большинство команд *утилитами*, то есть полезными программами. Стало быть, *командный интерпретатор* не обязан уметь **выполнять** все, что вводит пользователь. Ему достаточно разобрать *командную строку*, выделить из нее команду и параметры, а затем запустить *утилиту* - программу, имя которой совпадает с именем команды.

В действительности **собственных** команд в *командном интерпретаторе* немного. В основном это операторы языка программирования и прочие средства управления самим *интерпретатором*. Все знакомые Мефодию команды, даже echo, существуют в Linux в виде отдельных *утилит*. shell занимается только тем, что подготавливает набор параметров в *командной строке* (например, раскрывая *шаблоны* ), запускает программы и обрабатывает результаты их работы:

[methody@localhost methody]$ type info

info is /usr/bin/info

[methody@localhost methody]$ type echo

echo is a shell builtin

[methody@localhost methody]$ type -a echo

echo is a shell builtin

echo is /bin/echo

[methody@localhost methody]$ type -a -t echo

builtin

file

[methody@localhost methody]$ type -a -t date

file

[methody@localhost methody]$ type -at cat

file

2.19. Определение типа команды

В bash тип команды можно определить с помощью команды type. Собственные команды bash называются ***builtin (встроенная команда)*** , а для *утилит* выводится *путь*, содержащий название каталога, в котором лежит файл с соответствующей программой, и имя этой программы. Некоторые - самые нужные - команды встроены в bash, даже несмотря на то, что они имеются в виде *утилит* (например, echo ). Работает *встроенная команда* так же, но так как времени на ее выполнение уходит существенно меньше, *командный интерпретатор* выберет именно ее, если будет такая возможность. *Ключ* " -a " ("all", конечно), заставляет type вывести все возможные варианты интерпретации команды, а *ключ* " -t " - вывести тип команды вместо *пути*.

По совету Гуревича Мефодий сгруппировал *ключи*, написав " -at " вместо " -a -t ". Многие *утилиты* позволяют уменьшать длину *командной строки* подобным образом. Если встречается *параметрический ключ*, он должен быть последним в группе, а его значение - следовать, как и полагается, после. Группировать можно только *однобуквенные ключи*.

**Слова и разделители**

При разборе *командной строки* shell использует понятие *разделитель* (delimiter). ***Разделитель*** - это символ, разделяющий слова; таким образом, ***командная строка*** - это последовательность **слов** (которые имеют значение) и *разделителей* (которые значения не имеют). Для shell *разделителями* являются символ пробела, символ табуляции и символ перевода строки (который все-таки может попасть **между** словами способом, описанным в лекциях 6 и 7). Количество *разделителей* между двумя соседними словами значения не имеет.

Первое слово в тройке передается команде как первый параметр, второе - как второй и т. д. Для того чтобы *разделитель* попал **внутрь** слова (и получившаяся строка с *разделителем* передалась как **один** параметр), всю нужную подстроку надо окружить одинарными или двойными кавычками:

[methody@localhost methody]$ echo One Two Three

One Two Three

[methody@localhost methody]$ echo One "Two Three"

One Two Three

[methody@localhost methody]$ echo 'One

>

> Ой. И что дальше?

> А, кавычки забыл!'

One

Ой. И что дальше?

А, кавычки забыл!

[methody@localhost methody]$

2.20. Закавычивание в командной строке

В первом случае команде echo было передано **три** параметра - " One ", " Two " и " Three ". Она их и вывела, разделяя пробелом. Во втором случае параметров было **два**: " One " и " Two Three ". В результате эти два параметра были также выведены через пробел. В третьем случае параметр был всего **один** - от открывающего апострофа " 'One '" до закрывающего ...забыл!' ". Все время ввода bash услужливо выдавал Мефодию подсказку " > " - в знак того, что набор *командной строки* продолжается, но в режиме ввода содержимого кавычек.

**Лекция 3:**

**Структура файловой системы**

**Организация файловой системы**

**Файл**

*Файл* - это понятие, привычное любому пользователю компьютера. Для пользователя каждый *файл* - это отдельный предмет, у которого есть начало и конец и который отличается от всех остальных *файлов* именем и расположением ("как называется" и "где лежит"). Как и любой предмет, *файл* можно создать, переместить и уничтожить, однако без внешнего вмешательства он будет сохраняться неизменным неопределенно долгое время. *Файл* предназначен для хранения данных любого типа - текстовых, графических, звуковых, исполняемых программ и многого другого. Аналогия *файла* с предметом позволяет пользователю быстро освоиться при работе с данными в операционной системе.

Для операционной системы Linux *файл* - не менее важное понятие, чем для ее пользователя: все данные, хранящиеся на любых носителях, обязательно находятся внутри какого-нибудь *файла*, в противном случае они просто недоступны ни для операционной системы, ни для пользователей. Более того, многие устройства, подключенные к компьютеру (начиная с клавиатуры и заканчивая любыми внешними устройствами, например, принтерами и сканерами), Linux представляет как *файлы* (так называемые *файлы-дырки* ). Конечно, *файл*, содержащий обычные данные, сильно отличается от *файла*, предназначенного для обращения к устройству, поэтому в Linux определено несколько различных типов *файлов*. В основном пользователь имеет дело с *файлами* трех типов: **обычными файлами**, предназначенными для хранения данных, *каталогами* и *файлами-ссылками* (именно о них и пойдет речь в данной лекции, о *файлах* других типов см. лекцию 11).

*Файл* - это oтдельная область данных на одном из носителей информации, у которой есть собственное имя.

**Система файлов: каталоги**

*Файловая система* с точки зрения пользователя - это "пространство", в котором размещаются *файлы*. Наличие *файловой системы* позволяет определить не только "как называется *файл* ", но и "где он находится". Различать *файлы* только по имени было бы нецелесообразно: приходилось бы помнить, как называется каждый *файл* и при этом заботиться о том, чтобы имена никогда не повторялись. Более того, необходим механизм, позволяющий работать с группами тематически связанных между собой *файлов* (например, компонентов одной и той же программы или разных глав диссертации). Иначе говоря, *файлы* нужно **систематизировать**.

*Файловая система* - способ хранения и организации доступа к данным на информационном носителе или его разделе. Классическая *файловая система* имеет иерархическую структуру, в которой *файл* однозначно определяется *полным путем* к нему.

Linux может работать с различными типами *файловых систем*, которые различаются списком поддерживаемых возможностей, производительностью в разных ситуациях, надежностью и другими признаками. Подробнее о работе Linux с разными *файловыми системами* речь пойдет в лекции 11. В этой лекции будут описаны возможности *файловой системы* Ext2/Ext3.

Большинство современных *файловых систем* (но не все!) используют в качестве основного организационного принципа *каталоги*. ***Каталог*** - это список ссылок на *файлы* или другие *каталоги* . Принято говорить, что **каталог содержит файлы** или другие *каталоги*, хотя в действительности он только **ссылается** на них, физическое размещение данных на диске обычно никак не связано с размещением *каталога*. *Каталог*, на который есть ссылка в данном *каталоге*, называется ***подкаталогом*** или *вложенным каталогом* . *Каталог* в *файловой системe* более всего напоминает библиотечный *каталог*, содержащий ссылки на объединенные по каким-то признакам книги и другие разделы *каталога* ( *файлы* и *подкаталоги* ). Ссылка на один и тот же *файл* может содержаться в нескольких *каталогах* одновременно - это делает доступ к *файлу* более удобным. В *файловой системe* Ext2 каждый *каталог* - это отдельный *файл* особого типа (" d ", от англ. "directory"), отличающийся от *обычного файла* с данными: в нем могут содержаться только ссылки на другие *файлы* и *каталоги*.

В *файловой системе* Linux нет папок и документов. Есть *каталоги* и *файлы*, возможности которых куда шире.

Довольно часто вместо термина *"каталог"* употребляется *"папка"* (англ. *folder* ). Этот термин хорошо вписывается в представление о *файлах* как о предметах, которые можно раскладывать по *папкам*, однако часть возможностей *файловой системы*, которая противоречит этому представлению, таким образом затемняется. В частности, с термином *"папка"* плохо согласуется то, что ссылка на *файл* может присутствовать одновременно в нескольких *каталогах*, *файл* может быть ссылкой на другой *файл* и т. д. В Linux эти возможности *файловой системы* весьма важны для эффективной работы, поэтому мы будем использовать более подходящий термин *"каталог"*.

В *файловой системе*, организованной при помощи *каталогов*, на любой *файл* должна быть ссылка как минимум из одного *каталога*, в противном случае *файл* просто не будет доступен внутри этой *файловой системы*, иначе говоря, не будет существовать.

**Имена файлов и каталогов**

**Допустимые имена**

Главные отличительные признаки *файлов* и *каталогов* - их имена. В Linux имена *файлов* и *каталогов* могут быть длиной не более 255 символов, и могут содержать любые символы, кроме " / ". Причина этого ограничения очевидна: данный символ используется как разделитель имен в составе пути, поэтому не должен встречаться в самих именах. Причем Linux всегда различает заглавные и строчные буквы в именах *файлов* и *каталогов*, поэтому " methody ", " Methody " и " METHODY " будут тремя **разными** именами.

Есть несколько символов, допустимых в именах *файлов* и *каталогов*, которые нужно использовать с осторожностью. Это так называемые *спецсимволы* " \* ", " \ ", " & ", " < ", " > ", " ; ", " ( ", " ) ", " | ", а также символы пробела и табуляции. Дело в том, что эти символы имеют особое значение для любой **командной оболочки**, поэтому нужно будет специально позаботиться о том, чтобы командная оболочка воспринимала эти символы как часть имени *файла* или *каталога*. О специальном значении символа " - " для команд Linux уже шла речь в лекции 2, там же обсуждалось, как изменить его интерпретацию1. О том, зачем командной оболочке нужны *спецсимволы*, речь пойдет в лекции 8.

**Кодировки и русские имена**

Как можно было заметить, пока во всех встречавшихся именах *файлов* и *каталогов* употреблялись только символы латинского алфавита и некоторые знаки препинания. Это не случайно и вызвано желанием сделать так, чтобы приводимые примеры выглядели на любых системах одинаково. В Linux в именах *файлов* и *каталогов* допустимо использовать любые символы любого языка, однако такая свобода требует жертв, на которые Мефодий, например, пойти не смог.

Дело в том, что с давних пор каждый символ (буква) каждого языка традиционно представлялся в виде **одного** байта. Такое представление накладывает очень жесткие ограничения на **количество букв** в алфавите: их может быть не больше 256, а за вычетом управляющих символов, цифр, знаков препинания и прочего - и того меньше. Обширные алфавиты (например, иероглифические японский и китайский) пришлось заменять упрощенным их представлением. Вдобавок, первые 128 символов из этих 256 лучше всегда оставлять неизменными, соответствующими стандарту ASCII, включающему латиницу, цифры, знаки препинания и наиболее популярные символы из тех, что встречаются на клавиатуре печатной машинки. Интерпретация остальных 128 символов зависит от того, какая *кодировка* установлена в системе. Например, в русской *кодировке* KOI8-R 228-й символ такой таблицы соответствует букве "Д", а в западноевропейской *кодировке* ISO-8859-1 этот же символ соответствует букве "a" с двумя точками над ней (как у нашей буквы "ё").

Имена *файлов*, **записанные** на диск в одной *кодировке*, выглядят нелепо, если при **просмотре каталога** была установлена другая. Более того, многие *кодировки* заполняют *диапазон символов* с номерами от 128 до 255 **не полностью**, поэтому соответствующего символа может вообще не быть! Это означает, что **ввести** такое искаженное *имя файла* с клавиатуры (например, для того, чтобы его переименовать) напрямую не удастся: придется пускаться на разные ухищрения, описанные в лекции 8. Наконец, многие языки, в том числе и русский, исторически имеют несколько кодировок2. К сожалению, в настоящее время нет стандартного способа указывать *кодировку* прямо в имени *файла*, поэтому в рамках одной *файловой системы* стоит придерживаться единой *кодировки* при именовании *файлов*.

Существует универсальная *кодировка*, включающая символы всех письменностей мира - UNICODE. Стандарт UNICODE в настоящее время получает все большее распространение и претендует на статус общего для всех текстов, хранящихся в электронном виде. Однако пока он не достиг желаемой универсальности, особенно в области *имен файлов*. **Один** символ в UNICODE может занимать **больше** одного байта - и в этом его главный недостаток, так как множество полезных прикладных программ, отлично работающих с **однобайтными кодировками**, необходимо основательно или даже полностью перерабатывать для того, чтобы научить их обращаться с UNICODE. Возможно, причина недостаточной распространенности этой *кодировки* также и в том, что UNICODE - очень громоздкий стандарт, и он может оказаться неэффективным при работе с *файловой системой*, где скорость и надежность обработки - очень существенные качества.

Это не означает, что, называя *файлы*, не следует использовать языки, отличные от английского. Пока точно известно, в какой *кодировке* задано *имя файла* - проблем не возникнет. Однако Мефодий решил, что гарантий в передаче названного по-русски *файла* на какую-нибудь **другую** систему можно добиться, только передавая вместе с ним настройку *кодировки*, даже две: в своей системе и в системе адресата (неизвестно какой!). Другой, гораздо более легкий способ передать *файл* - использовать в его названии **только** символы ASCII.

**Расширения**

Многим пользователям знакомо понятие **расширение** - часть *имени файла* после точки, обычно ограничивающаяся несколькими символами и указывающая на тип содержащихся в *файле* данных. В *файловой системе* Linux нет никаких предписаний по поводу расширения: в *имени файла* может быть любое количество точек (в том числе ни одной), а после последней точки может стоять любое количество символов3. Хотя расширения не обязательны и не навязываются технологией в Linux, они широко используются: расширение позволяет человеку или программе, не открывая *файл*, только по его имени определить, какого типа данные в нем содержатся. Однако нужно учитывать, что расширение - это только набор соглашений о наименовании *файлов* разных типов. Строго говоря, данные в *файле* могут не соответствовать заявленному расширению по той или иной причине, поэтому всецело полагаться на расширение нельзя.

Определить тип содержимого *файла* можно и на основании самих данных. Многие форматы предусматривают указание в начале *файла*, как следует интерпретировать дальнейшую информацию: как программу, исходные данные для текстового редактора, страницу HTML, звуковой *файл*, изображение или что-то другое. В распоряжении пользователя Linux всегда есть утилита file, которая предназначена именно для определения типа содержащихся в *файле* данных:

[methody@localhost methody]$ file -- -filename-with-

-filename-with-: ASCII English text

[methody@localhost methody]$ file /home/methody

/home/methody: directory

Пример 3.1. Определение типа данных в файле

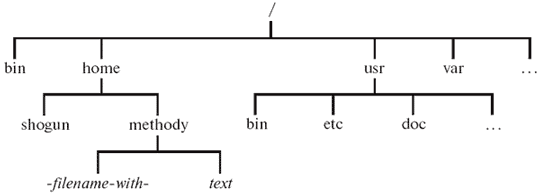
Мефодий, забыв, что содержится в *файле* " -filename-with- ", который он создал в примере, представленном в предыдущей лекции, хотел было уже посмотреть его содержимое при помощи команды cat. Однако его остановил Гуревич, который посоветовал сначала выяснить, что за данные содержатся в этом *файле*. Не исключено, что это двоичный *файл* исполняемой программы, а в таком *файле* могут встречаться последовательности, которые случайно совпадут с **управляющими последовательностями** терминала. Поведение терминала после этого может стать непредсказуемым, и неопытный пользователь вряд ли сможет с ним справиться. Мефодий получил вполне точный ответ от утилиты file: в его *файле* - английский текст в *кодировке* ASCII. file умеет различать очень многие типы данных и почти наверняка выдаст правильную информацию. Эта утилита никогда не доверяет расширению *файла* (если оно присутствует) и анализирует сами данные. file различает не только разные данные, но и разные типы *файлов*, в частности, сообщит, если исследуемый *файл* является не *обычным файлом*, а, например, *каталогом*.

**Дерево каталогов**

Понятие *каталога* позволяет **систематизировать** все объекты, размещенные на носителе данных (например, на диске). В большинстве современных *файловых систем* используется иерархическая модель организации данных: существует один *каталог*, объединяющий все данные в *файловой системе* - это "корень" всей *файловой системы*, ***корневой каталог*** . *Корневой каталог* может содержать любые объекты *файловой системы*, и в частности, *подкаталоги* ( *каталоги* первого уровня вложенности). Те, в свою очередь, также могут содержать любые объекты *файловой системы* и *подкаталоги* (второго уровня вложенности) и т. д. Таким образом, **все**, что записано на диске - *файлы*, *каталоги* и специальные *файлы* - обязательно "принадлежит" *корневому каталогу*: либо непосредственно (содержится в нем), либо на некотором уровне вложенности.

Иерархию вложенных друг в друга *каталогов* можно соотнести с иерархией данных в системе: объединить тематически связанные *файлы* в *каталог*, тематически связанные *каталоги* - в один общий *каталог* и т. д. Если строго следовать иерархическому принципу, то чем глубже будет *уровень вложенности* каталога, тем более частным признаком должны быть объединены содержащиеся в нем данные. Если не следовать этому принципу, то вскоре окажется гораздо проще складывать все *файлы* в один *каталог* и искать среди них нужный, чем выполнять такой поиск по всем *подкаталогам* системы. Однако в этом случае о какой бы то ни было **систематизации файлов** говорить не приходится.

Структуру *файловой системы* можно представить наглядно в виде дерева4, "корнем" которого является *корневой каталог*, а в вершинах расположены все остальные *каталоги*. На рис. 3.1 изображено дерево *каталогов*, курсивом обозначены имена *файлов*, прямым начертанием - имена *каталогов*.



**Рис. 3.1.**Дерево каталогов в Linux

В любой *файловой системе* Linux всегда есть только один *корневой каталог*, который называется " / ". Пользователь Linux всегда работает с единым деревом *каталогов*, даже если разные данные расположены на разных носителях: нескольких жестких или сетевых дисках, съемных дисках, CD-ROM и т. п. 5 Для того чтобы отключать и подключать *файловые системы* на разных устройствах в состав одного общего дерева, используются процедуры *монтирования* и *размонтирования*, о которых речь пойдет в лекции 11. После того, как *файловые системы* на разных носителях подключены к общему дереву, содержащиеся на них данные доступны так, как если бы все они составляли единую *файловую систему*: пользователь может даже не знать, на каком устройстве какие *файлы* хранятся.

Положение любого *каталога* в дереве *каталогов* точно и однозначно описывается при помощи *полного пути*. *Полный путь* всегда начинается от *корневого каталога* и состоит из перечисления всех вершин, встретившихся при движении по ребрам дерева до искомого *каталога* включительно. Названия соседних вершин разделяются символом " / " ("слэш"). В Linux *полный путь*, например, до *каталога* " methody " в *файловой системе*, приведенной на рис. 3.1, записывается следующим образом: сначала символ " / ", обозначающий *корневой каталог*, затем к нему добавляется " home ", затем разделитель " / ", за которым следует название искомого *каталога* " methody ", в результате получается полный путь " /home/methody " 6.

Организация *каталогов* файловой системы в виде дерева не допускает появления циклов: т. е. *каталог* не может содержать в себе *каталог*, в котором содержится сам. Благодаря этому ограничению *полный путь* до любого *каталога* или *файла* в *файловой системе* всегда будет **конечным**.

**Размещение компонентов системы: стандарт FHS**

Попробуем разобраться, как устроено *дерево* *каталогов* Linux, где и что в нем можно найти. Фрагмент дерева *каталогов* типичной *файловой системы* Linux ( Some Linux, которую использует Мефодий) приведен на рис. 3.1. Мефодий решил обследовать свою *файловую систему*, начиная с *корневого каталога*: Гуревич посоветовал использовать для этого команду ls каталог, где каталог - это *полный путь* к *каталогу*: *утилита* ls выведет *список* всего, что в этом *каталоге* содержится:

[methody@localhost methody]$ ls /

bin dev home mnt root tmp var

boot etc lib proc sbin usr

[methody@localhost methody]$

Пример 3.2. Стандартные каталоги в /

*Утилита* ls вывела *список* *подкаталогов* корневого каталога. Этот *список* будет примерно таким же в любом дистрибутиве Linux. В *корневом каталоге* Linux-системы обычно находятся только *подкаталоги* со **стандартными** именами. Более того, не только имена, но и **тип данных**, которые могут попасть в тот или иной *каталог*, также регламентированы этим стандартом. Данный стандарт называется **Filesystem Hierarchy Standard** ("стандартная структура *файловых систем* ").

Опишем кратко, что находится в каждом из *подкаталогов* корневого *каталога*. Мы не будем приводить полные списки *файлов* для каждого описываемого *каталога*, а Мефодий сможет просмотреть их при помощи команды ls имя каталога.

|  |  |
| --- | --- |
| /bin | Название этого *каталога* происходит от слова "binaries" ("двоичные", "исполняемые"). В этом *каталоге* находятся исполняемые *файлы* самых необходимых утилит. Сюда попадают такие программы, которые могут понадобиться системному администратору или другим пользователям для устранения неполадок в системе или при восстановлении после сбоя. |
| /boot | "Boot" - загрузка системы. В этом *каталоге* находятся *файлы*, необходимые для самого первого этапа - загрузки ядра - и, обычно, само *ядро*. Пользователю практически никогда не требуется непосредственно работать с этими *файлами*. |
| /dev | В этом *каталоге* находятся все имеющиеся в системе ***файлы-дырки***: *файлы* особого типа, предназначенные для обращения к различным системным ресурсам и устройствам (англ. "devices" - "устройства", отсюда и сокращенное название *каталога* ). Например, *файлы* /dev/ttyN соответствуют *виртуальным консолям*, где N - номер *виртуальной консоли*. Данные, введенные пользователем на первой *виртуальной консоли*, система считывает из *файла* /dev/tty1; в этот же *файл* записываются данные, которые нужно вывести пользователю на эту консоль. В *файлах-дырках* в действительности не хранятся никакие данные, при их помощи данные **передаются**. Подробнее о работе с *файлами-дырками* речь пойдет в лекции 11. |
| /etc | *Каталог* для системных *конфигурационных файлов*. Здесь хранится информация о специфических настройках данной системы: информация о зарегистрированных пользователях, доступных ресурсах, настройках различных программ. Подробно системные *конфигурационные файлы* будут рассмотрены в лекции 12. |
| /home | Здесь расположены *каталоги*, принадлежащие пользователям системы - ***домашние каталоги***, отсюда и название "home". Отделение всех *файлов*, создаваемых пользователями, от прочих системных *файлов* дает очевидное преимущество: серьезное повреждение системы или необходимость обновления не затронет наиболее ценной информации - пользовательских *файлов*. |
| /lib | Название этого *каталога* - сокращение от "libraries" (англ. "библиотеки"). **Библиотеки** - это собрания стандартных функций, необходимых многим программам: операций ввода/вывода, рисования элементов графического интерфейса и т. д. Чтобы не включать эти функции в текст каждой программы, используются стандартные функции библиотек - это значительно экономит место на диске и упрощает написание программ. В этом *каталоге* содержатся библиотеки, необходимые для работы наиболее важных системных утилит (размещенных в /bin и /sbin ). |
| /mnt | *Каталог* для *монтирования* (от англ. "mount") - временного подключения *файловых систем*, например, на съемных носителях (CD-ROM и др.). Подробно о *монтировании* файловых систем речь пойдет в лекции 11. |
| /proc | В этом *каталоге* все *файлы* "виртуальные" - они располагаются не на диске, а в оперативной памяти. В этих *файлах* содержится информация о программах ( *процессах* ), выполняемых в данный момент в системе. |
| /root | *Домашний каталог* администратора системы - пользователя root. Смысл размещать его отдельно от *домашних каталогов* остальных пользователей состоит в том, что /home может располагаться на отдельном устройстве, которое не всегда доступно (например, на сетевом диске), а *домашний каталог* root должен присутствовать в любой ситуации. |
| /sbin | *Каталог* для важнейших системных утилит (название *каталога* - сокращение от "system binaries"): в дополнение к утилитам /bin здесь находятся программы, необходимые для загрузки, резервного копирования, восстановления системы. Полномочия на исполнение этих программ есть только у системного администратора. |
| /tmp | Этот *каталог* предназначен для *временных файлов*: в таких *файлах* программы хранят необходимые для работы промежуточные данные. После завершения работы программы *временные файлы* теряют смысл и должны быть удалены. Обычно *каталог* /tmp очищается при каждой загрузке системы. |
| /usr | *Каталог* /usr - это "государство в государстве". Здесь можно найти такие же *подкаталоги* bin, etc, lib, sbin, как и в *корневом каталоге*. Однако в *корневой каталог* попадают только утилиты, **необходимые** для загрузки и восстановления системы в аварийной ситуации - **все остальные** программы и данные располагаются в *подкаталогах* /usr. Прикладных программ в современных системах обычно установлено очень много, поэтому этот раздел *файловой системы* может быть очень большим. |
| /var | Название этого *каталога* - сокращение от "variable" ("переменные" данные). Здесь размещаются те данные, которые создаются в *процессе* работы разными программами и предназначены для передачи другим программам и системам (очереди печати, электронной почты и др.) или для сведения системного администратора ( *системные журналы*, содержащие протоколы работы системы). В отличие от *каталога* /tmp сюда попадают те данные, которые могут понадобиться после того, как создавшая их программа завершила работу. |

Стандарт *FHS* регламентирует не только перечисленные *каталоги*, но и их *подкаталоги*, а иногда даже приводит *список* конкретных *файлов*, которые должны присутствовать в определенных каталогах7. Этот стандарт последовательно соблюдается во всех Linux-системах, хотя и не без горячих споров между разработчиками при выходе каждой новой его версии.

Стандартное *размещение* *файлов* позволяет и человеку, и даже программе предсказать, где находится тот или иной *компонент* системы. Для человека это означает, что он сможет быстро сориентироваться в любой системе Linux (где *файловая система* организована в соответствии со стандартом) и найти то, что ему нужно. Для программ стандартное расположение *файлов* - это возможность организации автоматического взаимодействия между разными компонентами системы.

Мефодий уже успел воспользоваться некоторыми преимуществами, которые дает стандартное расположение *файлов*: на предыдущих лекциях он запускал утилиты, не указывая *полный путь* к исполняемому *файлу*, например, cat вместо /bin/cat. *Командная оболочка* "знает", что исполняемые *файлы* располагаются в *каталогах* /bin, /usr/bin и т. д. - именно в этих *каталогах* она ищет исполняемый *файл* cat. Благодаря этому каждая вновь установленная в системе *программа* немедленно оказывается доступна пользователю из командной строки. Для этого не требуется ни перезагружать систему, ни запускать какие-либо процедуры - достаточно просто поместить исполняемый *файл* в один из соответствующих *каталогов*.

Рекомендации стандарта *по* размещению *файлов* и *каталогов* основываются на принципе размещения *файлов*, которые *по*-разному используются в системе, в разных *подкаталогах*. *По* типу использования *файлы* можно разделить на следующие группы:

1. **пользовательские/системные файлы**

Пользовательские *файлы* - это все *файлы*, созданные пользователем и не принадлежащие ни одному из компонентов системы. О пользе разграничения пользовательских и системных *файлов* речь уже шла выше.

1. **изменяющиеся/неизменные файлы**

К неизменным *файлам* относятся все статические компоненты программного обеспечения: библиотеки, исполняемые *файлы* и т. д. - все, что не изменяется само без вмешательства системного администратора. Изменяющиеся *файлы* - это те, которые изменяются без вмешательства человека в *процессе* работы системы: *системные журналы*, очереди печати и пр. Выделение неизменных *файлов* в отдельную структуру (например, /usr ) позволяет использовать соответствующую часть *файловой системы* в режиме "только чтение", что уменьшает вероятность случайного повреждения данных и позволяет применять для хранения этой части *файловой системы* CD-ROM и другие носители, доступные только для чтения.

1. **разделяемые/неразделяемые файлы**

Это разграничение становится полезным, если речь идет о сети, в которой работает несколько компьютеров. Значительная часть информации при этом может храниться на одном из компьютеров и использоваться всеми остальными по сети (к такой информации относятся, например, многие программы и *домашние каталоги* пользователей). Однако часть *файлов* нельзя разделять между системами (например, *файлы* для начальной загрузки системы).

*Полный путь* к *каталогу* формально ничем не отличается от пути к *файлу*, т. е. *по* *полному пути* нельзя сказать наверняка, является его последний элемент *файлом* или *каталогом*. Чтобы отличать *путь* к *каталогу*, иногда используют *запись* с символом " / " в конце пути: " /home/methody/ ".

**Лекция 4:**

**Работа с файловой системой**

**Текущий каталог**

*Файловая система* не только систематизирует данные, но и является основой метафоры "рабочего места" в Linux. Каждая выполняемая программа "работает" в строго определенном *каталоге* файловой системы. Такой *каталог* называется ***текущим каталогом*** . Можно представлять, что *программа* во время работы "находится" именно в этом *каталоге*, это ее "рабочее *место*". В зависимости от *текущего каталога* поведение программы может меняться: зачастую *программа* будет *по* умолчанию работать с файлами, расположенными именно в *текущем каталоге* - до них она "дотянется" в первую *очередь*. *Текущий каталог* есть у любой программы, в том числе и у командной оболочки (*shell*) пользователя. Поскольку взаимодействие пользователя с системой обязательно опосредовано командной оболочкой, можно говорить о том, что *пользователь* "находится" в том *каталоге*, который в данный момент является **текущим каталогом его командной оболочки**.

Все команды, отдаваемые пользователем при помощи *shell*, наследуют *текущий каталог* *shell*, т. е. "работают" в том же *каталоге*. *По* этой причине пользователю важно знать *текущий каталог* *shell*. Для этого служит *утилита* pwd:

[methody@localhost methody]$ pwd

/home/methody

[methody@localhost methody]$

Пример 4.1. Текущий каталог: pwd

*Команда* pwd (*print* *working directory*) возвращает *полный путь* текущего каталога командной оболочки - естественно, именно той командной оболочки, при помощи которой была выполнена *команда* pwd. В данном случае Мефодий узнал, что в этот момент (на данной *виртуальной консоли*) текущим является *каталог* "/home/methody ".

Почти все утилиты, с которыми работал Мефодий в предыдущих лекциях, *по* умолчанию читают и создают файлы в *текущем каталоге*. Так, Мефодий обращался к файлам, не используя никаких путей, просто *по* имени. Например, задействовал утилиту cat, чтобы вывести на экран содержимое файла "text ":

[methody@localhost methody]$ cat text

File: info.info, Node: Help-Cross, Up: Cross-refs

The node reached by the cross reference in Info

. . .

[methody@localhost methody]$ cat /home/methody/text

File: info.info, Node: Help-Cross, Up: Cross-refs

The node reached by the cross reference in Info

. . .

Пример 4.2. Полный и относительный путь к файлу

В действительности, *командная оболочка*, прежде чем передавать *параметр* "text" (*имя файла*) утилите cat, подставляет *значение* *текущего каталога* - получается *полный путь* к этому файлу в *файловой системе*: "/home/methody/text ". Содержимое данного файла *утилита* cat выведет на экран. 1 Набирая только *имя файла* без пути к *текущему каталогу*, Мефодий воспользовался *относительным путем* к этому файлу.

*Относительный путь* (*relative* *path*) - *путь* к объекту *файловой системы*, не начинающийся в *корневом каталоге*. Для каждого *процесса* Linux определен *текущий каталог*, с которого система начинает *относительный путь* при выполнении файловых операций.

*Относительный путь* строится точно так же, как и полный - перечислением через "/" всех названий *каталогов*, встретившихся при движении к искомому *каталогу* или файлу. Между полным и *относительным путем* есть только одно существенное различие: *относительный путь* начинается **от текущего каталога**, в то время как *полный путь* всегда начинается **от корневого каталога**. *Относительный путь* любого файла или *каталога* в *файловой системе* может иметь любую конфигурацию: чтобы добраться до искомого файла, можно двигаться как *по* направлению к *корневому каталогу*, так и от него (см. раздел "Перемещение *по* дереву *каталогов* "). Linux различает полный и *oтносительный пути* очень просто: если *имя объекта* начинается на "/" - это *полный путь*, в любом другом случае - относительный.

Отделить *путь* к файлу от его имени можно с помощью команд dirname и *basename* соответственно:

[methody@localhost methody]$ basename /home/methody/text

text

[methody@localhost methody]$ basename text

text

[methody@localhost methody]$ dirname /home/methody/text

/home/methody

[methody@localhost methody]$ dirname ./text

.

[methody@localhost methody]$ dirname text

.

Пример 4.3. Использование dirname и basename

Мефодий заметил, что для "text" и "./text" dirname выдает одинаковый результат: ".", что понятно: как было сказано выше, эти формы *пути* эквивалентны, а при **автоматической** обработке результатов dirname гораздо лучше получить ".", **чем пустую строку**.

**Домашний каталог**

Мефодий заметил, что в примерах этой и прошлых лекций, заходя с разных *виртуальных консолей* *по* очереди и одновременно, он всегда оказывался в одном и том же *текущем каталоге*: он все время обращался к своим файлам при помощи *относительного пути* и всегда находил нужные. Это не случайно - в Linux у каждого пользователя обязательно есть **собственный каталог**, который и становится текущим сразу после *регистрации в системе* - домашний каталог2. Для Мефодия *домашним каталогом* является "/home/methody ".

***Домашний каталог*** (home directory) - это *каталог*, предназначенный для хранения собственных данных пользователя Linux. Как правило, является текущим непосредственно после регистрации пользователя в системе. *Полный путь* к *домашнему каталогу* хранится в *переменной окружения* HOME.

Поскольку каждый *пользователь* располагает собственным *каталогом* и *по* умолчанию работает в нем, решается задача разделения файлов разных пользователей. Обычно *доступ* других пользователей к чужому *домашнему каталогу* ограничен: наиболее типична ситуация, когда пользователи могут читать содержимое файлов друг друга, но не имеют *права* их изменять или удалять.

**Информация о каталоге**

Чтобы иметь возможность ориентироваться в *файловой системе*, нужно знать, что содержится в каждом *каталоге*. Запомнить всю структуру *файловой системы* невозможно и не нужно: в любой момент можно просмотреть содержимое любого *каталога* при помощи утилиты ls (сокращение от англ. "*list*" - "*список*"):

[methody@localhost methody]$ ls

-filename-with- text

[methody@localhost methody]$

Пример 4.4. Команда ls

Поданная без параметров, *команда* ls выводит *список* файлов и *каталогов*, содержащихся в текущем каталоге3. При помощи этой утилиты Мефодий обнаружил, что в его *домашнем каталоге* (который в данный момент является текущим) содержатся два файла, созданные в примере, приведенном в предыдущей лекции: "-filename-with-" и "text ".

*Утилита* ls принимает один *параметр*: имя *каталога*, содержимое которого нужно вывести. Имя может быть задано любым доступным способом: в виде полного или *относительного пути*. Например, чтобы получить *список* файлов в своем *домашнем каталоге*, Мефодий мог бы использовать команды "ls /home/methody" и "ls" - результат был бы аналогичным.

Кроме параметра, *утилита* ls "понимает" множество ключей, которые нужны главным образом для того, чтобы выводить дополнительную информацию о файлах в *каталоге* или выводить *список* файлов выборочно. Чтобы узнать обо всех возможностях ls, нужно, конечно же, прочесть *руководство* *по* этой утилите (" man ls ").

Почитав *руководство* *по* ls, Мефодий решил изучить содержимое своей *файловой системы* и начал с *корневого каталога*:

[methody@localhost methody]$ ls -F /

bin/ dev/ home/ mnt/ root/ swap/ tmp/ var/

boot/ etc/ lib/ proc/ sbin/ sys/ usr/

[methody@localhost methody]$

Пример 4.5. Команда ls -F

Мефодий использовал *ключ* -F, чтобы отличать файлы от *каталогов*. При наличии этого ключа ls в конце имени каждого *каталога* ставит символ "/ ", чтобы показать, что в нем может содержаться что-то еще. В выведенном списке нет ни одного файла - в *корневом каталоге* содержатся только подкаталоги.

Кроме того, Мефодий решил получить более подробную информацию о содержимом своего *домашнего каталога*:

[methody@localhost methody]$ ls -aF

-filename-with- .bash\_history .bashrc .lpoptions

.rpmmacros Documents/

./ .bash\_logout .emacs .mutt/ .xemacs/ text

../ .bash\_profile .i18n .pinerc .xsession.d/ tmp/

[methody@localhost methody]$

Пример 4.6. Команда ls -aF

Внезапно он обнаружил, что файлов в его *домашнем каталоге* не два, а гораздо больше. Дело в том, что *утилита* ls *по* умолчанию не выводит информацию об объектах, чье имя начинается с "." - в том числе о "." и "..". Для того чтобы посмотреть полный *список* содержимого *каталога*, и используется *ключ* "-a "(all)4. Как правило, с "." начинаются имена *конфигурационных файлов* и *конфигурационных каталогов* (вроде .bashrc, описанного в лекции 8), работа с которыми (т. е. **настройка** окружения, "рабочего места") не пересекается с работой над какой-нибудь прикладной задачей (хотя, конечно, эффективность работы зависит от хорошо настроенного окружения). Кроме того, подобных файлов в *домашнем каталоге* активно работающего пользователя со временем заводится немало (*по* одному на каждую приличную утилиту) и их присутствие в выдаче ls сильно загромождает выводимые данные.

Разберемся подробно в списке файлов в *домашнем каталоге* Мефодия. Начнем с весьма лаконичных имен "." и "..". Мефодий уже знает, что "." - это имя *текущего каталога*. Следующее имя в списке, ".." - это *ссылка* на *родительский каталог*. ***Родительский каталог*** - это тот каталог, в **котором** находится данный *каталог* . *Родительским каталогом* для "/home/methody" будет *каталог* "/home ": он получается просто отбрасыванием последнего имени *каталога* в *полном пути*. Иначе можно сказать, что *родительский каталог* - это один шаг *по* дереву *каталогов* *по* направлению к корню. ".." - это сокращенный способ сослаться на *родительский каталог*: пока *текущим каталогом* является "/home/methody ", *относительный путь* ".." (или, что то же самое, "./..") будет эквивалентен "/home ". С использованием ".." можно строить **сколь угодно** длинные пути, такие как "../../usr/../var/log/../run/../../home "5.

Однако в действительности они применяются только при автоматической подстановке в программах, а во время работы пользователя необходимости в такого рода усложнениях не возникает.

*Родительский каталог* (*parent directory*) - это *каталог*, в котором содержится данный. Для *корневого каталога* родительским является он сам.

Ссылки на текущий и на *родительский каталог* обязательно присутствуют в **каждом каталоге** в Linux. Даже если *каталог* пуст, т. е. не содержит ни одного файла или подкаталога, *команда* "ls -a" выведет *список* из двух имен: "." и "..".

За ссылками на текущий и *родительский каталоги* следуют несколько файлов и каталогов, имена которых начинаются с ".". В них содержатся настройки командной оболочки (файлы, начинающиеся с ".bash ") и других программ. В *домашнем каталоге* каждого пользователя Linux всегда присутствует несколько таких файлов. Использование этих файлов позволяет пользователям независимо друг от друга настраивать поведение командной оболочки и других программ - организовывать свое "рабочее *место*" в системе. Подробнее речь об этом пойдет в лекции 12.

**Перемещение по дереву каталогов**

*Пользователь* может работать с файлами не только в своем *домашнем каталоге*, но и в других *каталогах*. В этом случае будет удобно **сменить текущий каталог**, т. е. "переместиться" в другую точку *файловой системы*. Для смены *текущего каталога* командной оболочки используется *команда* cd (от англ. "*change* *directory*" - "сменить *каталог* "). *Команда* cd принимает один *параметр*: имя *каталога*, в который нужно переместиться - сделать текущим. Как обычно, в качестве имени *каталога* можно использовать полный или *относительный путь*:

[methody@localhost methody]$ cd /home

[methody@localhost home]$ ls

methody shogun

[methody@localhost home]$ cd methody

[methody@localhost methody]$

Пример 4.7. Смена текущего каталога

Сначала Мефодий решил переместиться в *каталог* "/home" и посмотреть, что еще есть в этом *каталоге*, кроме его *домашнего каталога*. Он обнаружил *каталог* "shogun" и догадался, что это *домашний каталог* Гуревича, *входное имя* которого - "shogun ". Кроме того, он заметил, что изменился вид *приглашения командной строки* (подсказки *shell*) - *слово* "methody" заменилось на "home ". В *приглашении командной строки* часто указывается *текущий каталог* *shell* - чтобы пользователю легче было ориентироваться, в каком *каталоге* он "находится" в данный момент.

После этого Мефодий решил вернуться в свой *домашний каталог*, но в этом случае он использовал уже не полный, а *относительный путь* - "cd methody ". Вводя эту команду, Мефодий не стал целиком набирать имя своего *домашнего каталога*, а набрал только первые буквы "me" и нажал клавишу Tab, как ему советовал Гуревич. *Командная оболочка* умеет **достраивать** имена файлов и *каталогов*: пользователю достаточно набрать несколько первых символов имени файла или *каталога* и нажать Tab. Если есть только один вариант завершения имени - *оболочка* закончит его сама, и пользователю не придется набирать оставшиеся символы. *Достраивание* - весьма существенное средство экономии усилий и повышения эффективности при работе с командной строкой. Современные командные оболочки умеют достраивать имена файлов и *каталогов*, а также имена команд. *Достраивание* наиболее развито в командном интерпретаторе zsh.

Те же самые перемещения - в *родительский каталог* и обратно - Мефодий мог бы осуществить, набирая значительно меньше символов. Для перемещения в *родительский каталог* (" /home ") удобно воспользоваться ссылкой "..". Необходимость вернуться в *домашний каталог* из произвольной точки *файловой системы* возникает довольно часто, поэтому *командная оболочка* поддерживает обозначение *домашнего каталога* при помощи символа "~ ". Поэтому чтобы перейти в *домашний каталог* из любого другого, достаточно выполнить команду "cd ~ ". При исполнении команды символ "~" будет заменен командной оболочкой на *полный путь* к *домашнему каталогу* пользователя:

[methody@localhost methody]$ cd ..

[methody@localhost home]$ cd ~

[methody@localhost methody]$ cd ../shogun

[methody@localhost shogun]$ cd

[methody@localhost methody]$

Пример 4.8. Переход в родительский и в домашний каталог

При помощи символа "~" можно ссылаться и на *домашние каталоги* других пользователей: "~имя пользователя ". В примере 4.8 Мефодий перешел в *домашний каталог* Гуревича с помощью команды "cd ../shogun ". *Команда* cd, поданная без параметров, эквивалентна команде "cd ~" и делает *текущим каталогом* *домашний каталог пользователя*.

**Создание каталогов**

*Пользователь*, конечно, не должен хранить все свои файлы в одном *каталоге*. В *домашнем каталоге*, как и в любом другом, можно создавать сколько угодно подкаталогов, в них - свои подкаталоги и т. д. Иными словами, пользователю принадлежит фрагмент (*поддерево*) *файловой системы*, корнем которого является его *домашний каталог*.

Чтобы организовать такое *поддерево*, потребуется создать *каталоги* внутри домашнего. Для этого используется *утилита* mkdir. Она применяется с одним обязательным параметром: именем создаваемого *каталога*. *По* умолчанию *каталог* будет создан в *текущем каталоге*:

[methody@localhost methody]$ mkdir examples

[methody@localhost methody]$ ls -F

-filename-with- Documents/ examples/ text tmp/

[methody@localhost methody]$

Пример 4.9. Создание каталога

Мефодий решил навести порядок в своем *домашнем каталоге* и поместить все файлы с примерами и упражнениями в отдельный *подкаталог* - "examples ". Теперь, создав *каталог*, нужно переместить в него все файлы с примерами.

**Копирование и перемещение файлов**

Для перемещения файлов и *каталогов* предназначена *утилита* mv (от англ. "move" - "перемещать"). У mv два обязательных параметра: первый - перемещаемый *файл* или *каталог*, второй - *файл* или *каталог* назначения. Имена файлов и *каталогов* могут быть заданы в любом допустимом виде: при помощи полного или относительного пути. Кроме того, mv позволяет перемещать не только один *файл* или *каталог*, а сразу несколько. За подробностями о допустимых параметрах и ключах следует обратиться к руководству *по* mv:

[methody@localhost methody]$ mv -- -filename-with- examples/

[methody@localhost methody]$ cd examples

[methody@localhost examples]$ mv ../text .

[methody@localhost examples]$ ls

-filename-with- text

[methody@localhost examples]$

Пример 4.10. Перемещение файлов

Мефодий сначала переместил в *каталог* "examples" *файл* "-filename-with- ", а поскольку имя этого файла начинается с "- ", ему потребовалось предварить его ключом "-- ", чтобы следующее *слово* не было воспринято командной оболочкой как *параметр* (этот прием был описан в лекции 2). Затем он перешел в *каталог* "examples" и переместил из *родительского каталога* ("../") *файл* "text" в *текущий каталог* ("."). Теперь в *каталоге* "examples" находится два файла с примерами.

Перемещение файла внутри одной *файловой системы* в действительности равнозначно его **переименованию**: данные самого файла при этом остаются на тех же секторах диска, а изменяются *каталоги*, в которых произошло перемещение. Перемещение предполагает удаление ссылки на *файл* из того *каталога*, откуда он перемещен, и добавление ссылки на этот самый *файл* в тот *каталог*, куда он перемещен. В результате изменяется *полное имя файла* - *полный путь*, т. е. положение файла в *файловой системе*.

Иногда требуется создать копию файла: для большей сохранности данных, для того, чтобы создать модифицированную версию файла и т. п. В Linux для этого предназначена *утилита* cp (от англ. "*copy*" - "копировать"). *Утилита* cp требует присутствия двух обязательных параметров: первый - копируемый *файл* или *каталог*, второй - *файл* или *каталог* назначения. Как обычно, в именах файлов и *каталогов* можно использовать полные и *относительные пути*. Существует несколько вариантов комбинации файлов и *каталогов* в параметрах cp - о них можно прочесть в руководстве:

[methody@localhost examples]$ cp text text.bak

[methody@localhost examples]$ ls

-filename-with- text text.bak

Пример 4.11. Копирование файлов

Мефодий решил создать резервную копию файла "text ", "text.*bak*" в том же *каталоге*, что и исходный *файл*. Для этой простейшей *операции* копирования достаточно передать cp в качестве двух параметров имя исходного файла и имя копии. *По* умолчанию cp, как и многие другие утилиты, будет работать с файлами в *текущем каталоге*.

Нужно иметь в виду, что в Linux *утилита* cp нередко настроена таким образом, что при попытке скопировать *файл* поверх уже существующего файла никакого предупреждения не выводится. В этом случае *файл* будет просто перезаписан, а данные, которые содержались в старой версии файла, безвозвратно потеряны. Поэтому при использовании cp следует всегда быть внимательным и проверять имена файлов, которые нужно скопировать.

Говоря о копировании, уместно вспомнить широко известное *высказывание*, приписываемое Уильяму Оккаму: "Не следует умножать сущности сверх необходимого". Созданная при помощи cp копия файла связана с оригиналом только в воспоминаниях пользователя, в файловой же системе исходный *файл* и его копия - две совершенно независимые и ничем не связанные единицы. Поэтому при наличии нескольких копий одного и того же файла в рамках **одной файловой системы** повышается *вероятность* запутаться в копиях или забыть о некоторых из них. Если задача состоит в том, чтобы обеспечить *доступ* к одному и тому же файлу из разных точек *файловой системы*, нужно использовать специально предназначенный для этого механизм *файловой системы* Linux - ссылки.

**Файл и его имена: ссылки**

**Жесткие ссылки**

Каждый файл представляет собой область данных на жестком диске компьютера или на другом носителе информации, которую можно найти **по имени**. В *файловой системе* Linux содержимое файла связывается с его именем при помощи *жестких ссылок*. Создание файла с помощью любой программы означает, что будет создана *жесткая ссылка* - имя файла, и открыта новая область данных на диске. Причем количество ссылок на одну и ту же область данных (файл) не ограничено, т. е. у файла может быть несколько имен.

Пользователь Linux может добавить файлу еще одно имя (создать еще одну *жесткую ссылку* на файл) при помощи утилиты ln (от англ. "link" - "соединять, связывать"). Первый параметр - это имя файла, на который нужно создать ссылку, второй - имя новой ссылки. По умолчанию ссылка будет создана в *текущем каталоге*:

[methody@localhost methody]$ ln examples/text text-hardlink

[methody@localhost methody]$ ls -lR

./:

. . .

drwxr-xr-x 3 methody methody 4096 Окт 16 04:45 examples

-rw-r--r-- 2 methody methody 653 Окт 6 10:31 text-hardlink

./examples:

итого 92

-rw-r--r-- 1 methody methody 84718 Окт 6 10:31 -filename-with-

-rw-r--r-- 2 methody methody 653 Окт 6 10:31 text

Пример 4.12. Создание жестких ссылок

Мефодий создал в своем *домашнем каталоге* жесткую ссылку с именем "text-*hardlink*" на файл "text ", который находится в подкаталоге "examples ". Выведя подробный список файлов *текущего каталога* и его подкаталогов (" ls -lR "), Мефодий обратил внимание, что у файлов "text" и "text-*hardlink*" совпадают и размер (" 653 "), и время создания. Это его совершенно не удивило, поскольку он знает, что теперь "/home/methody/text-*hardlink*" и "/home/methody/examples/text" - это два имени одного и того же файла. В подробном описании, выведенном командой "ls -l ", Мефодию остались непонятны только два первых поля. Как объяснил Гуревич, первое "слово", состоящее из знаков "-drwx ", - это обозначение *прав доступа* к файлу, о которых речь пойдет в лекции 6. А следующее за ним число - количество *жестких ссылок* на данный файл или *каталог*. У "text" и "text-*hardlink*" стоит число "2" - у этого файла два имени.

Доступ к одному и тому же файлу при помощи нескольких имен может понадобиться в следующих случаях:

1. Одна и та же программа известна под несколькими именами.
2. Доступ пользователей к некоторым *каталогам* в системе может быть ограничен из соображений безопасности. Однако если все же нужно организовать доступ пользователей к файлу, который находится в таком *каталоге*, можно создать *жесткую ссылку* на этот файл в другом *каталоге*.
3. Современные *файловые системы* даже на домашних персональных компьютерах могут насчитывать до нескольких десятков тысяч файлов и тысячи *каталогов*. Обычно у таких *файловых систем* сложная многоуровневая иерархическая организация - в результате пути ко многим файлам становятся очень длинными. Чтобы организовать более удобный доступ к файлу, который находится очень "глубоко" в иерархии *каталогов*, также можно использовать *жесткую ссылку* в более доступном *каталоге*.
4. Полное имя некоторых программ может быть весьма длинным (например, i586-alt-linux-gcc-3.3 ), к таким программам удобнее обращаться при помощи сокращенного имени (*жесткой ссылки*) - gcc-3.3.

**Индексные дескрипторы**

Поскольку благодаря *жестким ссылкам* у файла может быть несколько имен, понятно, что вся существенная информация о файле в *файловой системе* привязана не к имени. В *файловых системах* Linux вся информация, необходимая для работы с файлом, хранится в *индексном дескрипторе*. Для **каждого** файла существует *индексный дескриптор*: не только для обычных файлов, но и для каталогов6, *файлов-дырок* и т. д. Каждому файлу соответствует **один индексный дескриптор**.

***Индексный дескриптор*** - это описание файла, в котором содержится:

* тип файла (обычный файл, *каталог*, *файл-дырка* и т. д.);
* *права доступа* к файлу;
* информация о том, кому принадлежит файл;
* отметки о времени создания, модификации, последнего доступа к файлу;
* размер файла;
* указатели на физические блоки на диске, принадлежащие этому файлу - в этих блоках хранится "содержимое" файла.

Все *индексные дескрипторы* пронумерованы, поэтому номер *индексного дескриптора* - это *уникальный идентификатор файла* в *файловой системе* - в отличие от **имени** файла (*жесткой ссылки* на него), которых может быть несколько. Узнать номер *индексного дескриптора* любого файла можно при помощи все той же утилиты ls с ключом -i:

[methody@localhost methody]$ ls -i ./text-hardlink

examples/text

127705 examples/text 127705 ./text-hardlink

Пример 4.13. Информация об индексных дескрипторах файлов

Мефодий решил поинтересоваться номерами *индексных дескрипторов* файла "text" и *жесткой ссылки* на него "text-*hardlink*" - он обнаружил, что эти номера совпадают ("127705"), то есть этим двум именам соответствует один *индексный дескриптор*, т. е. один и тот же файл.

Все операции с *файловой системой* - создание, удаление и перемещение файлов - производятся на самом деле над *индексными дескрипторами*, а имена нужны только для того, чтобы пользователь мог легко ориентироваться в *файловой системе*. (Было бы очень неудобно запоминать многозначный номер каждого нужного файла или *каталога*.) Более того, имя (или имена) файла в его **индексном дескрипторе не указаны**. В *файловой системе* Ext2 имена файлов хранятся в *каталогах*: каждый *каталог* представляет собой список имен файлов и номеров их *индексных дескрипторов*. *Жесткую ссылку* (имя файла, хранящееся в *каталоге*) можно представлять как каталожную карточку, на которой указан номер *индексного дескриптора* - идентификатор файла.

***Жесткая ссылка*** (hard link) - запись вида имя файла+номер *индексного дескриптора* в *каталоге* . *Жесткие ссылки* в Linux - основной способ обратиться к файлу по имени.

**Символьные ссылки**

У *жестких ссылок* есть два существенных ограничения:

1. *Жесткая ссылка* может указывать только на файл, но не на *каталог*, потому что в противном случае в *файловой системе* могут возникнуть циклы - бесконечные пути.
2. *Жесткая ссылка* не может указывать на файл в другой *файловой системе*. Например, невозможно создать на жестком диске *жесткую ссылку* на файл, расположенный на дискете7. Чтобы избежать этих ограничений, были разработаны *символьные ссылки*. ***Cимвольная ссылка*** - это просто файл, в котором содержится имя другого файла. *Cимвольные ссылки*, как и жесткие, предоставляют возможность обращаться к одному и тому же файлу по разным именам. Кроме того, *символьные ссылки* могут указывать и на *каталог*, чего не позволяют *жесткие ссылки*. *Символьные ссылки* называются так потому, что содержат **символы** - *путь* к файлу или *каталогу*.

***Cимвольная ссылка*** (*symbolic link*, *файл-ссылка*) - это файл особого типа (" l "), в котором содержится *путь* к другому файлу. Если на пути к файлу встречается *cимвольная ссылка*, система выполняет подстановку: исходный *путь* заменяется тем, что содержится в ссылке.

*Cимвольную ссылку* можно создать при помощи команды ln с ключом "-s" (сокращение от "symbolic"):

[methody@localhost methody]$ ln -s examples/text text-symlink

[methody@localhost methody]$ ls -li

. . .

127699 drwxr-xr-x 2 methody methody 4096 Окт 4 17:12 examples

127705 -rw-r--r-- 2 methody methody 653 Сен 30 10:04 text-hardlink

3621 lrwxrwxrwx 1 methody methody 13 Окт 4 18:05 text-symlink -> examples/text

[methody@localhost methody]$

Пример 4.14. Создание символьных ссылок

Теперь Мефодий решил создать в своем домашнем каталоге *символьную ссылку* на файл text и назвать ее text-symlink. Команда ls -li отобразила этот файл совсем не так, как остальные: стрелочка (" -> ") указывает, куда направлена ссылка. Кроме того, Мефодий обратил внимание, что номер *индексного дескриптора* (первое поле), размер и время создания файла text-symlink отличаются от text-*hardlink*, а также во втором поле (количество жестких ссылок на файл) text-symlink указано "1 ". Все эти признаки недвусмысленно свидетельствуют о том, что text-symlink и text - это **разные** файлы. Однако если выполнить команду cat text-symlink, то на экран будет выведено содержимое файла text.

*Символьная ссылка* вполне может содержать имя несуществующего файла. В этом случае ссылка будет существовать, но не будет "работать": например, если попробовать вывести содержимое такой "битой" ссылки при помощи команды cat, будет выдано сообщение об ошибке.

Узнать, куда указывает *символьная ссылка*, можно при помощи утилиты realpath:

[methody@localhost methody]$ realpath text-symlink

/home/methody/examples/text

Пример 4.15. Раскрытие символьных ссылок

**Удаление файлов и каталогов**

В Linux для удаления файлов предназначена *утилита* rm (сокращение от англ. "*remove*" - "удалять"):

[methody@localhost methody]$ rm examples/text

[methody@localhost methody]$ ls -l text-hardlink

-rw-r--r-- 1 methody methody 653 Сен 30 10:04 text-hardlink

[methody@localhost methody]$ rm text-hardlink

[methody@localhost methody]$ ls -l text-hardlink

ls: text-hardlink: No such file or directory

Пример 4.16. Удаление файла

Разобравшись в ссылках, Мефодий решил удалить *файл* text в *каталоге* examples. После этого *файл* text-*hardlink* в *домашнем каталоге* Мефодия, который является *жесткой ссылкой* на удаленный *файл* text, продолжает благополучно существовать. Единственное отличие, которое заметил Мефодий - количество *жестких ссылок* на этот *файл* теперь уменьшилось с "2" до "1" - действительно, text-*hardlink* - теперь единственное имя этого файла. Получается, что Мефодий удалил только одно из имен этого файла (*жесткую ссылку*) - сам же *файл* остался нетронутым.

Однако если Мефодий удалит и *жесткую ссылку* text-*hardlink*, у этого файла больше не останется ни одного имени, он станет недоступным пользователю *файловой системы* и будет уничтожен.

*Утилита* rm предназначена именно для удаления *жестких ссылок*, а не самих файлов. В Linux, чтобы полностью удалить *файл*, требуется последовательно удалить все *жесткие ссылки* на него. При этом все *жесткие ссылки* на *файл* (его имена) равноправны - среди них нет "главной", с исчезновением которой исчезнет *файл*. Пока есть хоть одна *ссылка*, *файл* продолжает существовать. Впрочем, у большинства файлов в Linux есть только одно имя (одна *жесткая ссылка* на *файл*), поэтому *команда* rm *имя файла* в большинстве случаев успешно удаляет *файл*.

Как уже говорилось, *символьные ссылки* - это отдельные файлы, поэтому после того, как Мефодий удалил *файл* text, text-symlink, который ссылался на этот *файл*, продолжает существовать, однако теперь это - "битая *ссылка*", поэтому его также можно удалить командой rm.

Мефодий решил создать *каталог* для разных упражнений - test, а потом решил обойтись одним *каталогом* examples. Однако *команда* rm не сработала, заявив, что test - это *каталог*:

[methody@localhost methody]$ mkdir test

[methody@localhost methody]$ rm test

rm: невозможно удалить `test': Is a directory

[methody@localhost methody]$ rmdir test

[methody@localhost methody]$

Пример 4.17. Удаление каталога

Для удаления *каталогов* предназначена другая *утилита* - rmdir (от англ. "*remove* *directory*"). Впрочем, rmdir согласится удалить *каталог* только в том случае, если он пуст: в нем нет никаких файлов и подкаталогов. Удалить *каталог* вместе со всем его содержимым можно командой rm с ключом "-r" (*recursive*). *Команда* rm -r *каталог* - очень удобный способ потерять в одночасье **все** файлы: она рекурсивно8 обходит весь *каталог*, удаляя все, что попадется: файлы, подкаталоги, *символьные ссылки*... а *ключ* "-f" (force) делает ее работу еще неотвратимее, так как подавляет запросы вида "удалить защищенный от записи *файл*", так что rm работает безмолвно и безостановочно.

**Помните**: если вы удалили *файл*, значит, он уже не нужен, и не подлежит восстановлению!

В Linux не предусмотрено процедуры восстановления удаленных файлов и *каталогов*. Поэтому стоит быть **очень** внимательным, отдавая команду rm и, тем более, rm -r: нет никакой гарантии, что случайно удаленные данные удастся восстановить. Узнав об этом, Мефодий не огорчился, но подумал, что впредь будет удалять только **действительно** ненужные файлы, а все сомнительное - перемещать с помощью mv в *подкаталог* ~/tmp, где оно не будет мешать, и где можно периодически наводить порядок.

**Лекция 5:**

**Доступ процессов к файлам и каталогам**

**Процессы**

Как уже упоминалось в лекции 1, *загрузка* Linux завершается тем, что на всех *виртуальных консолях* (на самом деле – на всех терминалах системы), предназначенных для работы пользователей, запускается *программа* getty. *Программа* выводит приглашение и ожидает активности пользователя, который может захотеть работать именно на этом терминале. Введенное *входное имя* getty передает программе login, которая вводит *пароль* и определяет, разрешено ли работать в системе с этим входным именем и этим паролем. Если login приходит к выводу, что работать можно, она запускает *стартовый командный интерпретатор*, посредством которого *пользователь* и управляет системой.

Выполняющаяся программа называется в Linux ***процессом*** . Все *процессы* система регистрирует в *таблице процессов*, присваивая каждому уникальный номер – *идентификатор процесса* ( **p** rocess **id** entificator, **PID** ). Манипулируя *процессами*, система имеет дело именно с их *идентификаторами*, другого способа отличить один *процесс* от другого, *по* большому счету, нет. Для просмотра своих *процессов* можно воспользоваться утилитой ps (" **p** rocess **s** tatus"):

[methody@localhost methody]$ ps -f

UID PID PPID C STIME TTY TIME CMD

methody 3590 1850 0 13:58 tty3 00:00:00 -bash

methody 3624 3590 0 14:01 tty3 00:00:00 ps -f

5.1. Просмотр таблицы собственных процессов

Здесь Мефодий вызвал ps с ключом " -f " (" **f** ull"), чтобы добыть побольше информации. Представлены оба принадлежащих ему *процесса*: *стартовый командный интерпретатор*, bash, и выполняющийся ps. Оба *процесса* запущены с терминала tty3 (третьей *системной консоли*) и имеют *идентификаторы* 3590 и 3624 соответственно. В *поле* *PPID* (" **p** arent **p** rocess **id** entificator") указан *идентификатор родительского процесса*, т. е. *процесса*, **породившего** данный. Для ps это – bash, а для bash, очевидно, login, так как именно он запускает стартовый *shell*. В выдаче не оказалось строки для этого login, равно как и для большинства других *процессов* системы, так как они не принадлежат пользователю methody.

*Процесс* - выполняющаяся *программа* в Linux. Каждый *процесс* имеет уникальный *идентификатор процесса*, **PID**. *Процессы* получают *доступ* к ресурсам системы (оперативной памяти, файлам, внешним устройствам и т. п.) и могут изменять их содержимое. *Доступ* регулируется с помощью *идентификатора пользователя* и *идентификатора группы*, которые система присваивает каждому *процессу*.

**Запуск дочерних процессов**

Запуск одного *процесса* **вместо** другого организован в Linux с помощью *системного вызова* exec(). Старый *процесс* из памяти удаляется навсегда, вместо него загружается новый, при этом настройка **окружения** не меняется, даже **PID** остается прежним. **Вернуться** к выполнению старого *процесса* невозможно, разве что запустить его снова с помощью того же exec() (от "execute" – "исполнить"). Кстати, **имя файла** (программы), из которого запускается *процесс*, и **собственное имя** *процесса* (в *таблице процессов* ) могут и не совпадать. Собственное имя *процесса* – это такой же параметр командной строки, как и те, что передаются ему пользователем: для exec() требуется и *путь* к файлу, и **полная** командная строка, **нулевой** (стартовый) элемент которой – как раз название команды1.

Вот откуда " - " в начале имени *стартового командного интерпретатора* ( -bash ): его "подсунула" программа login, чтобы была возможность отличать его от других запущенных тем же пользователем оболочек.

Для работы командного интерпретатора одного exec() недостаточно. В самом деле, shell не просто запускает утилиту, а дожидается ее завершения, обрабатывает результаты ее работы и продолжает диалог с пользователем. Для этого в Linux служит *системный вызов*fork() ("вилка, развилка"), применение которого приводит к возникновению еще одного, ***дочернего***, *процесса* – точной копии породившего его *родительского*. *Дочерний процесс* ничем не отличается от *родительского*: имеет такое же окружение, те же стандартный ввод и *стандартный вывод*, одинаковое содержимое памяти и продолжает работу с той же самой точки (возврат из fork() ). Отличий два: во-первых, эти *процессы* имеют разные **PID**, под которыми они зарегистрированы в *таблице процессов*, а во-вторых, различается *возвращаемое значение*fork(): *родительский процесс* получает в качестве результата fork() *идентификатор процесса-потомка*, а *процесс-потомок* получает " 0 ".

Дальнейшие действия shell при запуске какой-либо программы очевидны. Shell-потомок немедленно вызывает эту программу с помощью exec(), а shell-родитель дожидается завершения работы *процесса-потомка*( **PID** которого ему известен) с помощью еще одного *системного вызова*, wait(). Дождавшись и проанализировав результат команды, shell продолжает работу:

[methody@localhost methody]$ cat > loop

while true; do true; done

^D

[methody@localhost methody]$ sh loop

^C

[methody@localhost methody]$

5.2. Создание бесконечно выполняющегося сценария

По совету Гуревича Мефодий создал *сценарий* для sh (или bash, на таком уровне их команды совпадают), который ничего не делает. Точнее было бы сказать, что этот *сценарий* делает **ничего**, бесконечно повторяя в *цикле команду*, вся работа которой состоит в том, что она завершается без ошибок (в лекции 7 говорится о том, что " > файл " в командной строке просто перенаправляет *стандартный вывод* команды в файл). Запустив этот *сценарий* с помощью команды вида sh имя\_сценария, Мефодий ничего не увидел, но услышал, как загудел вентилятор охлаждения центрального процессора: машина трудилась! Управляющий символ " ^C ", как обычно, привел к завершению *активного процесса*, и командный интерпретатор продолжил работу.

Если бы в описанной выше ситуации *родительский процесс* не ждал, пока *дочерний* завершится, а сразу продолжал работать, получилось бы, что оба *процесса* выполняются параллельно: пока запущенный *процесс* что-то делает, пользователь продолжает командовать оболочкой. Для того чтобы запустить *процесс* параллельно, в shell достаточно добавить " & " в конец командной строки:

[methody@localhost methody]$ sh loop&

[1] 3634

[methody@localhost methody]$ ps -f

UID PID PPID C STIME TTY TIME CMD

methody 3590 1850 0 13:58 tty3 00:00:00 -bash

methody 3634 3590 99 14:03 tty3 00:00:02 sh loop

methody 3635 3590 0 14:03 tty3 00:00:00 ps -f

5.3. Запуск фонового процесса

В результате *стартовый командный интерпретатор* ( **PID** 3590) оказался родителем сразу двух *процессов*: sh, выполняющего *сценарий*loop, и ps.

*Процесс*, запускаемый параллельно, называется ***фоновым*** (background). *Фоновые процессы* не имеют возможности **вводить** данные с того же терминала, что и породивший их shell (только из файла), зато **выводить** данные на этот терминал могут (правда, когда на одном и том же терминале вперемешку появляются сообщения от нескольких *фоновых процессов*, начинается неразбериха). При каждом терминале в каждый момент времени может быть не больше одного *активного* (foreground) *процесса*, которому разрешено вводить данные с этого терминала. На время, пока команда (например, cat ) работает в активном режиме, породивший ее командный интерпретатор "уходит в фон", и там, в фоне, выполняет свой wait().

***активный процесс***, foreground process - *процесс*, предоставляющий возможность вводить данные с терминала..

***Фоновый процесс***, *background process* - *процесс*, не имеющий возможности вводить данные с терминала. Пользователь может запустить любое, но не превосходящее заранее заданное в системе, число *фоновых процессов*.

Стоит заметить, что параллельность работы *процессов* в Linux – **дискретная**. Здесь и сейчас выполняться может столько *процессов*, сколько центральных процессоров есть в компьютере (например, один). Дав этому одному *процессу* немного поработать, система запоминает все, что необходимо ему для работы, приостанавливает его, и запускает **следующий** *процесс*, потом следующий и так далее. Возникает **очередь** *процессов*, ожидающих выполнения. Только что поработавший *процесс* помещается в конец этой очереди, а следующий выбирается из ее начала. Когда очередь вновь доходит до того, первого *процесса*, система вспоминает необходимые для его выполнения данные (они называются *контекстом процесса* ), и он продолжает работать как ни в чем не бывало. Такая схема **разделения времени** между *процессами* называется ***псевдопараллелизмом*** .

В выдаче ps, которую получил Мефодий, можно заметить, что **PID** стартовой оболочки равен 3590, а **PID** запущенных из-под него команд (одной фоновой и одной активной) – 3634 и 3635. Это значит, что за время, прошедшее с момента входа Мефодия в систему до момента запуска sh loop&, в системе было запущено еще 3634-3590=44 *процесса*. Что ж, в Linux могут одновременно работать несколько пользователей, да и самой системе иногда случается запустить какую-нибудь утилиту (например, выполняя действия по расписанию). А вот sh и ps получили **соседние PID**, значит, пока Мефодий нажимал **Enter** и набирал ps -f, никаких других *процессов* не запускалось.

В действительности далеко не всем *процессам*, зарегистрированным в системе, **на самом деле** необходимо давать поработать наравне с другими. Большинству *процессов* работать **прямо сейчас** не нужно: они ожидают какого-нибудь события, которое им необходимо обработать. Чаще всего *процессы* ждут завершения операции ввода-вывода. Чтобы посмотреть, как потребляются ресурсы системы, можно использовать утилиту top. Но сначала Мефодий решил запустить **еще один** бесконечный *сценарий* (ему было интересно, как два *процесса* конкурируют за ресурсы):

[methody@localhost methody]$ bash loop&

[2] 3639

[methody@localhost methody]$ top

14:06:50 up 3:41, 5 users, load average: 1,31, 0,76, 0,42

4 processes: 1 sleeping, 3 running, 0 zombie, 0 stopped

CPU states: 99,4% user, 0,5% system, 0,0% nice, 0,0% iowait, 0,0% idle

Mem: 514604k av, 310620k used, 203984k free, 0k shrd, 47996k buff

117560k active, 148388k inactive

Swap: 1048280k av, 0k used, 1048280k free 184340k cached

PID USER PRI NI SIZE RSS SHARE STAT %CPU %MEM TIME COMMAND

3639 methody 20 0 1260 1260 1044 R 50,3 0,2 0:12 bash

3634 methody 18 0 980 980 844 R 49,1 0,1 3:06 sh

3641 methody 9 0 1060 1060 872 R 0,1 0,2 0:00 top

3590 methody 9 0 1652 1652 1264 S 0,0 0,3 0:00 bash

5.4. Разделение времени между процессами

Оказалось, что дерутся даже не два *процесса*, а три: sh (первый из запущенных интерпретаторов loop ), bash (второй) и сам top. Правда, по сведениям из поля %CPU, львиную долю процессорного времени отобрали sh и bash (они без устали вычисляют!), а top довольствуется десятой долей процента (а то и меньшей: ошибки округления). Стартовый bash вообще не хочет работать, он **спит** (значение " S ", **S** leep, поля STAT, status): ждет завершения *активного процесса*, top.

Увидев такое разнообразие информации, Мефодий кинулся читать руководство по top, однако скоро понял, что без знания **архитектуры** Linux большая его часть не имеет смысла. Впрочем, некоторая часть все же понятна: объем оперативной памяти (всей, используемой и свободной), время работы машины, объем памяти, занимаемой *процессами* и т. п.

Последний *процесс*, запущенный из оболочки в фоне, можно из **этой** оболочки сделать *активным* при помощи команды fg (" **f** ore **g** round" – "передний план"):

[methody@localhost methody]$ fg

bash loop

^C

5.5. Перевод фонового процесса в активное состояние с помощью команды fg (foreground)

Услужливый bash даже написал командную строку, которой был запущен этот *процесс*: " bash loop ". Мефодий решил "убить" его с помощью управляющего символа " ^C ". Теперь **последним** запущенным в фоне *процессом* стал sh, выполняющий *сценарий* loop.

**Сигналы**

Чтобы завершить работу *фонового процесса* с помощью " ^C ", Мефодию пришлось сначала сделать его *активным*. Это не всегда возможно и не всегда удобно. На самом деле, " ^C " – это не волшебная кнопка-убийца, а предварительно установленный символ (с ascii-кодом 3), при получении которого с терминала Linux передаст *активному процессу сигнал 2* (по имени INT, от "interrupt" – "прервать").

***Сигнал*** – это способность *процессов* обмениваться стандартными короткими сообщениями непосредственно с помощью системы. *Сообщение-сигнал* не содержит никакой информации, кроме номера *сигнала* (для удобства вместо номера можно использовать предопределенное системой имя). Для того чтобы передать *сигнал*, *процессу* достаточно задействовать *системный вызов* kill(), а для того чтобы **принять** *сигнал*, не нужно ничего. Если *процессу* необходимо как-то по-особенному реагировать на *сигнал*, он может зарегистрировать **обработчик**, а если обработчика нет, за него отреагирует система. Как правило, это приводит к немедленному завершению *процесса*, получившего *сигнал*. Обработчик *сигнала* запускается **асинхронно**, немедленно после получения *сигнала*, что бы *процесс* в это время ни делал.

*Сигнал* - короткое сообщение, посылаемое системой или *процессом* другому *процессу*. Обрабатывается асинхронно специальной подпрограммой-обработчиком. Если *процесс* не обрабатывает *сигнал* самостоятельно, это делает система.

Два *сигнала* – 9 ( KILL ) и 19 ( STOP ) – всегда обрабатывает система. Первый из них нужен для того, чтобы убить *процесс* **наверняка** (отсюда и название). *Сигнал* STOP **приостанавливает** *процесс*: в таком состоянии *процесс* не удаляется из *таблицы процессов*, но и не выполняется до тех пор, пока не получит *сигнал* 18 ( CONT ) – после чего продолжит работу. В Linux *сигнал* STOP можно передать *активному процессу* с помощью управляющего символа " ^Z ":

[methody@localhost methody]$ sh loop

^Z

[1]+ Stopped sh loop

[methody@localhost methody]$ bg

[1]+ sh loop &

[methody@localhost methody]$ fg

sh loop

^C

[methody@localhost methody]$

5.6. Перевод процесса в фон с помощью "^Z" и bg

Мефодий сначала запустил вечный цикл в качестве *активного процесса*, затем передал ему *сигнал* STOP с помощью " ^Z ", после чего дал команду bg ( **b** ack **g** round), запускающую в фоне последний остановленный *процесс*. Затем он снова перевел этот *процесс* в *активный* режим, и, наконец, убил его.

Передавать *сигналы* из командной строки можно любым *процессам* с помощью команды kill -сигнал PID или просто kill PID, которая передает *сигнал* 15 ( TERM ):

[methody@localhost methody]$ sh

sh-2.05b$ sh loop & bash loop &

[1] 3652

[2] 3653

sh-2.05b$ ps -fH

UID PID PPID C STIME TTY TIME CMD

methody 3590 1850 0 13:58 tty3 00:00:00 -bash

methody 3634 3590 87 14:03 tty3 00:14:18 sh loop

methody 3651 3590 0 14:19 tty3 00:00:00 sh

methody 3652 3651 34 14:19 tty3 00:00:01 sh loop

methody 3653 3651 35 14:19 tty3 00:00:01 bash loop

methody 3654 3651 0 14:19 tty3 00:00:00 ps -fH

5.7. Запуск множества фоновых процессов

Мефодий решил запустить несколько *процессов*, а потом выборочно поубивать их. Для этого он, вдобавок к уже висящему в фоне sh loop, запустил в качестве *активного процесса* новый командный интерпретатор, sh (при этом изменилось *приглашение командной строки* ). Из этого sh он запустил в фоне еще один sh loop и новый bash loop. Сделал он это одной командной строкой (при этом команды **разделяются** символом " & ", т. е. "И"; выходит, что запускается **и** та, **и** другая команда). В ps он использовал новый ключ – " -H " (" **H** ierarchy", "иерархия"), который добавляет в выдачу ps отступы, показывающие отношения "родитель–потомок" между *процессами*:

sh-2.05b$ kill 3634

[1]+ Terminated sh loop

sh-2.05b$ ps -fH

UID PID PPID C STIME TTY TIME CMD

methody 3590 1850 0 13:58 tty3 00:00:00 -bash

methody 3651 3590 0 14:19 tty3 00:00:00 sh

methody 3652 3651 34 14:19 tty3 00:01:10 sh loop

methody 3653 3651 34 14:19 tty3 00:01:10 bash loop

methody 3658 3651 0 14:23 tty3 00:00:00 ps -fH

5.8. Принудительное завершение процесса с помощью kill

Мефодий принялся убивать! Для начала он остановил работу давно запущенного sh, выполнявшего *сценарий* с вечным циклом ( **PID** 3634 ). Как видно из предыдущего примера, этот *процесс* за 16 минут работы системы "съел" не менее 14 минут процессорного времени, и конечно, ничего полезного не сделал. *Сигнал* о том, что *процесс-потомок* "умер", дошел до обработчика в стартовом bash ( **PID** 3590 ), и на терминал было выведено сообщение " [1]+ Terminated sh loop ", после чего стартовый bash продолжил ждать завершения *активного процесса* – sh ( **PID** 3651).

sh-2.05b$ exit

[methody@localhost methody]$ ps -fH

UID PID PPID C STIME TTY TIME CMD

methody 3590 1850 0 15:17 tty3 00:00:00 -bash

methody 3663 3590 0 15:23 tty3 00:00:00 ps -fH

methody 3652 1 42 15:22 tty3 00:00:38 bash loop

methody 3653 1 42 15:22 tty3 00:00:40 sh loop

[methody@localhost methody]$ kill -HUP 3652 3653

[methody@localhost methody]$ ps

PID TTY TIME CMD

3590 tty3 00:00:00 bash

3664 tty3 00:00:00 ps

5.9. Завершение процесса естественным путем с помощью сигнала "Hang Up"

Ждать ему оставалось недолго. Этот sh завершился естественным путем, от команды exit, оставив после себя двух "детей-сирот" ( **PID** 3652 и 3653 ), которые тотчас же усыновил "отец всех *процессов* " – init ( **PID** 1 ). Когда Мефодий расправился и с ними – с помощью *сигнала* 1 ( HUP, то есть " **H** ang **UP** ", "повесить"2 ) – некому было даже сообщить об их кончине (если бы *процесс-родитель* был жив, на связанный с ним терминал вывелось бы что-нибудь вроде " [1]+ Hangup sh loop ").

**Доступ к файлу и каталогу**

Но довольно насилия. Пора Мефодию задуматься и о другой стороне работы с Linux: о правах и свободах. Для начала – о свободах. *Таблица процессов* содержит *список* важнейших объектов системы – *процессов*. Однако не менее важны и объекты другого класса, те, что доступны в *файловой системе*: файлы, каталоги и специальные файлы (символьные ссылки, устройства и т. п.). *По* отношению к объектам *файловой системы процессы* выступают в роли **действующих субъектов**: именно *процессы* пользуются файлами, создают, удаляют и изменяют их. Факт использования файла *процессом* называется **доступом** к файлу, а **способ** воспользоваться файлом (каталогом, ссылкой и т. д.) – **видом** доступа.

**Чтение, запись и использование**

Видов доступа в *файловой системе* Linux три. Доступ на **чтение** (read) разрешает получать информацию из объекта, доступ на **запись** (write) – изменять информацию в объекте, и доступ на **использование** (execute) – выполнить операцию, специфичную для данного типа объектов. Доступ к объекту можно изменить командой chmod ( **ch** ange **mod** e, сменить режим (доступа)). В простых случаях формат этой команды таков: chmod доступ объект, где объект – это имя файла, каталога и т. п., а доступ описывает вид доступа, который необходимо разрешить или запретить. Значение " +r " разрешает доступ к объекту на чтение ( **r** ead), " -r " – запрещает. Аналогично " +w ", " -w ", " +x " и " -x " разрешают и запрещают доступ на запись ( **w** rite) и использование (e **x** ecute).

**Доступ к файлу**

Доступ к файлу на чтение и запись – довольно очевидные понятия:

[methody@localhost methody]$ date > tmpfile

[methody@localhost methody]$ cat tmpfile

Срд Сен 22 14:52:03 MSD 2004

[methody@localhost methody]$ chmod -r tmpfile

[methody@localhost methody]$ cat tmpfile

cat: tmpfile: Permission denied

[methody@localhost methody]$ date -u > tmpfile

[methody@localhost methody]$ chmod +r tmpfile; chmod -w tmpfile

[methody@localhost methody]$ cal > tmpfile

-bash: tmpfile: Permission denied

[methody@localhost methody]$ cat tmpfile

Срд Сен 22 10:52:35 UTC 2004

[methody@localhost methody]$ rm tmpfile

rm: удалить защищенный от записи обычный файл `tmpfile'? y

5.10. Что можно и что нельзя делать с файлом, доступ к которому ограничен

Следует заметить, что Мефодию известна операция *перенаправления вывода* – " > ", с помощью которой он создает файлы в своем *домашнем каталоге*. Добавление " > файл " в командную строку приводит к тому, что все, что было бы выведено на экран терминала3, попадает в файл. Мефодий создает файл, проверяет, можно ли из него читать, командой cat, запрещает доступ на чтение и снова проверяет: на этот раз cat сообщает об отказе в доступе ("*Permission denied*"). Тем не менее, **записать** в этот файл, перенаправив выдачу date -u оказывается возможным, потому что доступ на запись не закрыт. Если же закрыть доступ на запись, а доступ на чтение открыть (Мефодий сделал это в одной командной строке, разделив команды символом " ; "), невозможным станет **изменение** этого файла: попытка перенаправить вывод программы cal окажется неудачной, а чтение снова заработает. Сработает и **удаление** этого файла, хотя rm на всякий случай предупредит о том, что файл защищен от записи.

Доступ к файлу на **использование** означает возможность запустить этот файл в качестве программы, выполнить его. Например, все файлы из каталога /bin (в том числе /bin/ls, /bin/rm, /bin/cat, /bin/echo и /bin/date ) – *исполняемые*, т. е. доступны для использования, и оттого их можно применять в командной строке в качестве команд. В общем случае необходимо указать *путь* к программе, например, /bin/ls, однако программы, находящиеся в каталогах, перечисленных в *переменной окружения* PATH, можно вызывать просто по имени: ls (подробнее о *переменных окружения* рассказано в лекции 8).

**Сценарий**

*Исполняемые файлы* в Linux бывают двух видов. Первый – это файлы в собственно исполняемом (executable) формате. Как правило, такие файлы – результат **компиляции** программ, написанных на классических языках программирования вроде Си. Попытка прочитать такой файл с помощью, например, cat не приведет ни к чему полезному: на экране начнут появляться разнообразные бессмысленные символы, в том числе управляющие. Это так называемые **машинные коды** – язык, понятный только компьютеру. В Linux используется несколько форматов *исполняемых файлов*, состоящих из машинных кодов и служебной информации, необходимой операционной системе для запуска программы: согласно этой информации, ядро Linux выделяет для запускаемой программы оперативную память, загружает программу из файла и передает ей управление. Большинство утилит Linux – программы именно такого, "двоичного" формата.

Второй вид *исполняемых файлов* – *сценарии*. ***Сценарий*** – это **текстовый** файл, предназначенный для обработки какой-нибудь утилитой. Чаще всего такая утилита – это **интерпретатор** некоторого языка программирования, а содержимое такого файла – программа на этом языке. Мефодий уже написал один *сценарий* для sh: бесконечно выполняющуюся программу loop. Поскольку к тому времени он еще не знал, как пользоваться chmod, ему всякий раз приходилось явно указывать интерпретатор ( sh или bash ), а *сценарий* передавать ему в виде параметра (см. примеры в разделе " *Процессы* ").

*сценарий* Для выполнения *сценария* требуется программа-интерпретатор, *путь* к которой может быть указан в начале *сценария* в виде " #!путь\_к\_интерпретатору ". Если интерпретатор не задан, им считается /bin/sh.

Если же сделать файл *исполняемым*, то ту же самую процедуру – запуск интерпретатора и передачу ему *сценария* в качестве параметра командной строки – будет выполнять система:

[methody@localhost methody]$ cat > script

echo 'Hello, Methody!'

^D

[methody@localhost methody]$ ./script

-bash: ./script: Permission denied

[methody@localhost methody]$ sh script

Hello, Methody!

[methody@localhost methody]$ chmod +x script

[methody@localhost methody]$ ./script

Hello, Methody!

5.11. Создание простейшего исполняемого сценария

С первого раза Мефодию не удалось запустить *сценарий* script, потому что по умолчанию файл создается доступным для записи и чтения, но не для использования. После chmod +x файл стал *исполняемым*. Поскольку *домашний каталог* Мефодия не входил в PATH, пришлось использовать *путь* до созданного *сценария*, благо он оказался недлинным: " текущий\_каталог/имя\_сценария ", т. е. ./script.

Если системе не дать специального указания, то в качестве интерпретатора она запускает стандартный shell – /bin/sh. Однако есть возможность написать *сценарий* для **любой** утилиты, в том числе и написанной самостоятельно. Для этого первыми двумя байтами *сценария* должны быть символы " #!", тогда всю его первую строку, начиная с третьего байта, система воспримет как **команду** обработки. Исполнение такого *сценария* приведет к запуску указанной после " #!" команды, последним параметром которой будет имя самого файла *сценария*:

[methody@localhost methody]$ cat > to.sort

#!/bin/sort

some

unsorted

lines

[methody@localhost methody]$ sort to.sort

#!/bin/sort

lines

some

unsorted

[methody@localhost methody]$ chmod +x to.sort

[methody@localhost methody]$ ./to.sort

#!/bin/sort

lines

some

unsorted

5.12. Создание sort-сценария

Утилита sort сортирует – расставляет в алфавитном, обратном алфавитном или другом порядке – строки передаваемого ей файла. То же самое произойдет, если сделать этот файл *исполняемым*, вписав в начало /bin/sort в качестве интерпретатора, а затем выполнить получившийся *сценарий*: запустится утилита sort, а *сценарий* (файл с неотсортированными строками) будет передан ей в качестве параметра. Оформлять файлы, которые необходимо отсортировать, в виде *sort-сценариев* довольно бессмысленно – Мефодий просто хотел еще раз убедиться, что *сценарий* можно написать для чего угодно.

**Доступ к каталогу**

В отличие от файла, новый каталог создается (с помощью mkdir ) доступным и для чтения, и для записи, и для использования. Суть всех трех видов доступа к каталогу менее очевидна, чем суть доступа к файлу. Вкратце она такова: доступ по чтению – это **возможность просмотреть** содержимое каталога (список файлов), доступ по записи – это возможность **изменить содержимое** каталога, а доступ для использования – возможность **воспользоваться** этим содержимым: во-первых, сделать этот каталог *текущим*, а во-вторых, **обратиться** за доступом к содержащемуся в нем файлу:

[methody@localhost methody]$ mkdir dir

[methody@localhost methody]$ date > dir/smallfile

[methody@localhost methody]$ /bin/ls dir

smallfile

[methody@localhost methody]$ cd dir

[methody@localhost dir]$ pwd

/home/methody/dir

[methody@localhost dir]$ cd

[methody@localhost methody]$ pwd

/home/methody

[methody@localhost methody]$ cat dir/smallfile

Срд Сен 22 13:15:20 MSD 2004

5.13. Доступ к каталогу на чтение и использование

Мефодий создал каталог dir и файл smallfile в нем. При смене *текущего каталога* bash автоматически изменил строку-подсказку: как было сказано в лекции 4, последнее слово этой подсказки – имя *текущего каталога*. Описанная в той же лекции команда pwd ( **p** rint **w** ork **d** irectory) подтверждает это. Команда cd без параметров, как ей это и полагается, делает *текущим домашний каталог* пользователя.

[methody@localhost methody]$ chmod -x dir

[methody@localhost methody]$ ls dir

ls: dir/smallfile: Permission denied

[methody@localhost methody]$ alias ls

alias ls='ls --color=auto'

[methody@localhost methody]$ /bin/ls dir

smallfile

[methody@localhost methody]$ cd dir

-bash: cd: dir: Permission denied

[methody@localhost methody]$ cat dir/smallfile

cat: dir/smallfile: Permission denied

5.14. Доступ к каталогу на чтение без использования

Мефодий запретил доступ на использование каталога, ожидая, что просмотреть его содержимое с помощью ls будет можно, а сделать его *текущим* и добыть содержимое находящегося в нем файла – нельзя. Неожиданно команда ls выдала ошибку. Проницательный Мефодий заметил, что **имя файла** – smallfile – команда ls все-таки добыла, но ей зачем-то понадобился и сам файл. Гуревич посоветовал посмотреть, что выдаст команда *alias ls*. Оказывается, вместо простого ls умный bash подставляет специфичную для Linux команду ls --color=auto, которая раскрашивает имена файлов в разные цвета в зависимости от их типа. Для того чтобы определить тип файла (например, запускаемый ли он) необходимо получить к нему доступ, а этого-то как раз нельзя сделать, когда невозможно **использовать** каталог. Если явно вызывать утилиту ls ( /bin/ls ), поведение каталога становится вполне предсказуемым. Подробнее о том, чем занимается команда *alias* (о **сокращениях** ), рассказывается в лекции 8.

[methody@localhost methody]$ chmod +x dir; chmod -r dir

[methody@localhost methody]$ /bin/ls dir

/bin/ls: dir: Permission denied

[methody@localhost methody]$ cat dir/smallfile

Срд Сен 22 13:15:20 MSD 2004

[methody@localhost methody]$ cd dir

[methody@localhost dir]$ /bin/ls

ls: .: Permission denied

[methody@localhost dir]$ cd

[methody@localhost methody]$

5.15. Доступ к каталогу на использование без чтения

Если каталог, доступный для чтения, но недоступный для использования, требуется редко, то каталог, доступный для использования, но недоступный для чтения, может пригодиться. Как видно из примера, получить список файлов, находящихся в таком каталоге, не удастся, но получить доступ к самим файлам, **зная** их имена – можно. Мефодий тут же захотел положить в созданный каталог какой-нибудь нужный, но очень секретный файл, чтобы имя этого файла никто, кроме близких друзей, не знал. Поразмыслив, Мефодий отказался от этой затеи: во-первых, не без оснований подозревая, что администратор ( *суперпользователь* ) все равно сможет просмотреть содержимое такого каталога, а во-вторых, потому что нужного, но очень секретного файла под рукой не оказалось:

[methody@localhost methody]$ rm -rf dir

rm: невозможно открыть каталог `dir': Permission denied

[methody@localhost methody]$ chmod -R +rwx dir

[methody@localhost methody]$ rm -rf dir

5.16. Рекурсивное удаление каталога

Потеряв интерес к секретным файлам, Мефодий решил удалить каталог dir. Из лекции 4 он знает, что это можно сделать не с помощью rmdir, а с помощью rm с ключом " -r " ( **r** ecursive). Но сразу воспользоваться rm не удалось: чтение-то из каталога невозможно, и потому неизвестно, какие файлы там надо удалять. Поэтому сначала надо разрешить все виды доступа к dir. На всякий случай (а вдруг внутри dir попадется такой же нечитаемый подкаталог?) Мефодий выполняет рекурсивный вариант chmod – с ключом " -R " (" R " здесь большое, а не маленькое, потому что " -r " уже занято: означает запрет чтения). Команда chmod -R +rwx dir делает все файлы и каталоги в dir доступными для чтения, записи и использования; при этом все файлы становятся *исполняемыми*, но кого это тревожит, если следующей командой будет rm?

**Лекция 6:**

**Права доступа**

**рава доступа в файловой системе**

Работая с Linux, Мефодий заметил, что некоторые файлы и каталоги недоступны ему ни для чтения, ни для записи, ни для использования. Зачем такие нужны? Оказывается, другие пользователи могут обращаться к этим файлам, а у Мефодия просто **недостаточно прав**.

**Идентификатор пользователя**

Говоря о *правах доступа* пользователя к файлам, стоит заметить, что в действительности манипулирует файлами не сам пользователь, а запущенный им *процесс* (например, утилита rm или cat ). Поскольку и файл, и *процесс* создаются и управляются системой, ей нетрудно организовать какую угодно политику доступа одних к другим, основываясь на любых свойствах *процессов* как субъектов и файлов как объектов системы.

В Linux, однако, используются не какие угодно свойства, а результат *идентификации* пользователя – его *UID*. Каждый *процесс* системы обязательно **принадлежит** какому-нибудь пользователю, и *идентификатор пользователя* ( *UID* ) – обязательное свойство любого *процесса* Linux. Когда программа login запускает *стартовый командный интерпретатор*, она приписывает ему *UID*, полученный в результате диалога. Обычный запуск программы ( exec() ) или порождение нового *процесса* ( fork() ) не изменяют *UID* *процесса*, поэтому **все** *процессы*, запущенные пользователем во время терминальной сессии, будут иметь его *идентификатор*.

Поскольку *UID* однозначно определяется *входным именем*, оно нередко используется **вместо** *идентификатора* – для наглядности. Например, вместо выражения " *идентификатор пользователя*, соответствующий *входному имени* methody ", говорят " *UID* methody " (в приведенном ниже примере этот *идентификатор* равен **503** ):

[methody@localhost methody]$ id

uid=503(methody) gid=503(methody) группы=503(methody)

[methody@localhost methody]$ id shogun

uid=400(shogun) gid=400(shogun) г

руппы=400(shogun),4(adm),10(wheel),19(proc)

6.1. Как узнать идентификаторы пользователя и членство в группах

Утилита id, которой воспользовался Мефодий, выводит *входное имя* пользователя и соответствующий ему *UID*, а также *группу по умолчанию* и полный список *групп*, членом которых он является.

**Идентификатор группы**

Как было рассказано в лекции 1, пользователь может быть членом нескольких *групп*, равно как и несколько пользователей могут быть членами одной и той же *группы*. Исторически сложилось так, что одна из *групп* – *группа по умолчанию* – является для пользователя основной: когда (не вполне точно) говорят о " *GID* пользователя", имеют в виду именно идентификатор *группы по умолчанию*. В лекции 12 будет рассказано о том, что *GID* пользователя вписан в *учетную запись* и хранится в /etc/passwd, а информация о соответствии **имен** *групп* их идентификаторам, равно как и о том, в какие **еще** *группы* входит пользователь – в файле /etc/group. Из этого следует, что пользователь **не может не быть** членом как минимум одной *группы*, как снаряд не может не попасть в эпицентр взрыва. 1

Часто процедуру **создания** пользователя проектируют так, что имя *группы по умолчанию* совпадает с *входным именем* пользователя, а *GID* пользователя – с его *UID*. Однако это совсем не обязательно: не всегда нужно заводить для пользователя отдельную *группу*, а если заводить, то не всегда **удается** сделать так,чтобы **желаемый** идентификатор *группы* совпадал с **желаемым** *идентификатором пользователя*.

**Ярлыки объектов файловой системы**

При **создании** объектов файловой системы – файлов, каталогов и т. п. – каждому в обязательном порядке приписывается *ярлык*. ***Ярлык*** включает в себя *UID* – *идентификатор пользователя* - *хозяина файла*, *GID* – идентификатор *группы*, которой принадлежит файл, тип объекта и набор так называемых *атрибутов*, а также некоторую дополнительную информацию. *Атрибуты* определяют, кто и что имеет право делать с файлом, они описаны ниже:

[methody@arnor methody]$ ls -l

итого 24

drwx------ 2 methody methody 4096 Сен 12 13:58 Documents

drwxr-xr-x 2 methody methody 4096 Окт 31 15:21 examples

-rw-r--r-- 1 methody methody 26 Сен 22 15:21 loop

-rwxr-xr-x 1 methody methody 23 Сен 27 13:27 script

drwx------ 2 methody methody 4096 Окт 1 15:07 tmp

-rwxr-xr-x 1 methody methody 32 Сен 22 13:26 to.sort

6.2. Права доступа к файлам и каталогам, показанные командой ls -l

Ключ " -l " утилиты ls определяет "длинный" ( **l** ong) формат выдачи (справа налево): имя файла, время последнего изменения файла, размер в байтах, *группа*, *хозяин*, количество *жестких ссылок* и строчка *атрибутов*. Первый символ в строчке *атрибутов* определяет тип файла. Тип " - " отвечает "обычному" файлу, а тип " d " – каталогу ( **d** irectory). Имя пользователя и имя *группы*, которым принадлежит содержимое домашнего каталога Мефодия, – естественно, methody.

Быстрый разумом Мефодий немедленно заинтересовался вот чем: несмотря на то, что создание *жестких ссылок* на каталог невозможно, значение поля "количество *жестких ссылок* " для всех каталогов примера равно **двум**, а не одному. На самом деле этого и следовало ожидать, потому что **любой** каталог файловой системы Linux всегда имеет не менее двух имен: собственное (например, tmp ) и имя " ." в самом этом каталоге ( tmp/.). Если же в каталоге создать подкаталог, количество *жестких ссылок* на этот каталог увеличится на 1 за счет имени " .." в подкаталоге (например, tmp/subdir1/..):

[methody@arnor methody]$ ls -ld tmp

drwx------ 3 methody methody 4096 Окт 1 15:07 tmp

[methody@arnor methody]$ mkdir tmp/subdir2

[methody@arnor methody]$ ls -ld tmp

drwx------ 4 methody methody 4096 Окт 1 15:07 tmp

[methody@arnor methody]$ rmdir tmp/subdir\*

[methody@arnor methody]$ ls -ld tmp

drwx------ 2 methody methody 4096 Окт 1 15:07 tmp

6.3. Несколько жестких ссылок на каталог все-таки бывает!

#### Здесь Мефодий использовал ключ " -d " ( **d** irectory) для того, чтобы ls выводил информацию не о с**Иерархия прав доступа**

Теперь – более подробно о том, чему соответствуют девять символов в строке *атрибутов*, выдаваемой ls. Эти девять символов имеют вид " rwxrwxrwx ", где некоторые " r ", " w " и " x " могут заменяться на " - ". Очевидно, буквы отражают принятые в Linux три вида доступа – чтение, запись и использование – однако в *ярлыке* они присутствуют в трех экземплярах!

Дело в том, что любой пользователь ( *процесс* ) Linux по отношению к любому файлу может выступать в трех **ролях**: как *хозяин* ( **u** ser), как член *группы*, которой принадлежит файл ( **g** roup), и как *посторонний* ( **o** ther), никаких отношений собственности на этот файл не имеющий. Строка *атрибутов* – это три тройки " rwx ", описывающие *права доступа* к файлу *хозяина* этого файла (первая тройка, " u "), *группы*, которой принадлежит файл (вторая тройка, " g ") и *посторонних* (третья тройка, " o "). Если в какой-либо тройке не хватает буквы, а вместо нее стоит " - ", значит, пользователю в соответствующей роли будет в соответствующем виде доступа отказано.

При выяснении отношений между файлом и пользователем, запустившим *процесс*, роль определяется так:

1. Если *UID* файла совпадает с *UID* *процесса*, пользователь – *хозяин файла*
2. Если *GID* файла совпадает с *GID* **любой** *группы*, в которую входит пользователь, он – член *группы*, которой принадлежит файл.
3. Если ни *UID*, ни *GID* файла не пересекаются с *UID* *процесса* и списком *групп*, в которые входит запустивший его пользователь, этот пользователь – *посторонний*.

Именно в роли *хозяина* пользователь ( *процесс* ) может **изменять** *ярлык файла*. Это вполне соответствует обыденным понятиям о собственности (" **мой** файл: захочу – покажу, захочу – спрячу"). Единственное, чего не может делать *хозяин* со своим файлом – менять ему *хозяина*.

Далее следует протокол действий Гуревича, который, пользуясь возможностями *суперпользователя*, создал в каталоге /tmp несколько файлов с различными *правами доступа* к ним. Для того чтобы напомнить человеку, что работа ведется с *правами* *суперпользователя* (это требует гораздо большей ответственности), bash заменил привычный "доллар" в конце *приглашения командной строки* на "решетку". *Входное имя* он тоже заменил, но практика показывает, что решетка эффективнее:

[root@localhost root]# echo "All can read" > /tmp/read.all

[root@localhost root]# echo "Group wheel can read" > /tmp/read.wheel

[root@localhost root]# echo "Group methody can read" > /tmp/read.methody

[root@localhost root]# echo "Methody himself can read" > /tmp/read.Methody

[root@localhost root]# chgrp wheel /tmp/read.wheel; chmod o-r /tmp/read.wheel

[root@localhost root]# chgrp methody /tmp/read.methody; chmod o-r /tmp/read.methody

[root@localhost root]# chown methody /tmp/read.Methody; chmod og-r /tmp/read.Methody

6.4. Создание файлов с различными правами доступа

*Права доступа* изменяются с помощью трех команд: chown ( **ch** ange **own** er, сменить владельца), chgrp ( **ch** ange **gr** ou p, сменить *группу* ) и chmod с расширенным форматом параметра: перед частью, определяющей **доступ** (перед знаком " + " или " - "), могут быть перечислены роли " u ", " g ", " o " и " a " ( **a** ll, что соответствует " ugo "), доступ для которых изменяется. Кроме того, при задании доступа можно вместо " + " и " - " использовать " = ", тогда для заданных ролей указанные способы доступа разрешаются, а **неуказанные** – запрещаются. Вместо пары команд chown *хозяин файл* ; chgrp группа файл можно применять одну: chown хозяин:группа файл, которая изменяет одновременно и *UID*, и *GID* файла (каталога, ссылки и т. п.).

Мефодий хочет посмотреть, кто имеет доступ к файлам, созданным Гуревичем, а вдобавок – к файлу /etc/shadow. Для этого он использует команду ls -l с *шаблоном*, описывающим сразу **все** файлы, которые находятся в /tmp и имя которых **начинается** на " read.".

[methody@localhost methody]$ ls -l /tmp/read.\* /etc/shadow

-r-------- 1 root root 0 Сен 10 02:08 /etc/shadow

-rw-r--r-- 1 root root 13 Сен 22 17:49 /tmp/read.all

-rw-r----- 1 root methody 23 Сен 22 17:49 /tmp/read.methody

-rw------- 1 methody root 25 Сен 22 17:50 /tmp/read.Methody

-rw-r----- 1 root wheel 21 Сен 22 17:49 /tmp/read.wheel

[methody@localhost methody]$ cat /tmp/read.\* /etc/shadow

All can read

Group methody can read

Methody himself can read

cat: /tmp/read.wheel: Permission denied

cat: /etc/shadow: Permission denied

6.5. Чтение файлов с различными правами доступа

Что же получается? Из файла /etc/shadow можно только читать, причем только пользователю root. Изменять файлы /tmp/read.all, /tmp/read.*wheel* и /tmp/read.methody может только root, он же может и читать из них. Также читать из файла /tmp/read.*wheel* могут члены *группы* *wheel*, а из файла /tmp/read.methody – члены *группы* methody (это имя *группы*, а не имя пользователя!). Читать и писать в файл /tmp/read.Methody может только пользователь methody. Наконец, читать из файла /tmp/read.all могут к тому же и члены *группы* root, и вообще любые пользователи.

Попытка вывести содержимое этих файлов на экран (а значит, **прочитать** их) приводит к ожидаемому результату: *процессу* cat ( *UID* methody ) разрешено чтение из трех файлов. Из /tmp/read.all – так как по отношению к нему Мефодий играет роль *постороннего*, а ему открыт доступ на чтение (" r " в начале третьей тройки). Из /tmp/read.methody – так как пользователь methody входит в *группу* methody (см. пример 6.1), членам которой разрешено читать из этого файла (" r " в начале второй тройки). И, конечно, из файла /tmp/read.Methody, которому methody – *хозяин*, имеющий доступ на чтение (" r " в первой тройке).

Если бы Мефодию захотелось **записать** что-нибудь в эти файлы, он бы получил доступ только к одному – /tmp/read.Methody, потому что по отношению к остальным файлам Мефодий играет роль, которой закрыт доступ на запись ( /tmp/read.methody – член *группы*, остальные три – *посторонние* ).

Таким образом, определение *прав доступа* *процесса* к объекту файловой системы (например, файла) происходит так. Используя *UID* *процесса*, список *групп*, в которые входит пользователь, запустивший этот *процесс*, *UID* файла и *GID* файла, система определяет **роль** *процесса* по отношению к файлу, а затем обращается к соответствующей тройке *атрибутов* файла. Стоит заметить, что *процесс* не может выступать сразу в **нескольких ролях**, поэтому, например, файл с *ярлыком* " ---rw-rw- methody methody " **сам** Мефодий просмотреть не сможет (тройка *хозяина* определяет полное отсутствие доступа) 2. одержимом каталога tmp, а о самом этом каталоге.

**Использование прав доступа в Linux**

**Использование групп**

В Linux определено несколько *системных групп*, задача которых – обеспечивать доступ членов этих *групп* к разнообразным ресурсам системы. Часто такие *группы* носят говорящие названия: " disk ", " audio ", " cdwriter " и т. п. Тогда обычным пользователям доступ к некоторому файлу, каталогу или *файлу-дырке* Linux закрыт, но открыт членам *группы*, которой этот объект принадлежит.

Например, в Linux почти всегда используется *виртуальная файловая система* /proc – каталог, в котором в виде подкаталогов и файлов представлена информация из *таблицы процессов*. Имя подкаталога /proc совпадает с *PID* соответствующего *процесса*, а содержимое этого подкаталога отражает свойства *процесса*. *Хозяином* такого подкаталога будет хозяин *процесса* (с правами на чтение и использование), поэтому **любой** пользователь сможет посмотреть информацию о **своих** *процессах*. Именно каталогом /proc пользуется утилита ps:

[methody@arnor methody]$ ls -l /proc

. . .

dr-xr-x--- 3 methody proc 0 Сен 22 18:17 4529

dr-xr-x--- 3 shogun proc 0 Сен 22 18:17 4558

dr-xr-x--- 3 methody proc 0 Сен 22 18:17 4589

. . .

[methody@localhost methody]$ ps -af

UID PID PPID C STIME TTY TIME CMD

methody 4529 4523 0 13:41 tty1 00:00:00 -bash

methody 4590 4529 0 13:42 tty1 00:00:00 ps -af

6.6. Ограничение доступа к полной таблице процессов

Оказывается, запущено немало *процессов*, в том числе один – пользователем shogun ( *PID* 4558 ). Однако, несмотря на ключ " -a " ( **a** ll), ps выдала Мефодию только сведения о его *процессах*: 4529 – это входной shell, а 4590 – видимо, сам ps.

Другое дело – Гуревич. Он, как видно из примера 6.1, входит в *группу* proc, членам которой разрешено читать и использовать каждый подкаталог /proc:

shogun@localhost ~ $ ps -af

UID PID PPID C STIME TTY TIME CMD

methody 4529 4523 0 13:41 tty1 00:00:00 -bash

shogun 4558 1828 0 13:41 tty3 00:00:00 -zsh

shogun 4598 4558 0 13:41 tty3 00:00:00 ps -af

6.7. Доступ к полной таблице процессов: группа proc

Гуревич, опытный пользователь Linux, предпочитает bash "The Z Shell", zsh. Отсюда и различие приглашения в командной строке. Во всех shell, кроме самых старых, символ " ~ " означает домашний каталог. Этим сокращением удобно пользоваться, если текущий каталог – не домашний, а оттуда (или туда) нужно скопировать файл. Получается команда наподобие " cp ~/нужный\_файл." или " cp нужный\_файл ~ " соответственно.

Команда ps -a выводит информацию обо **всех** *процессах*, запущенных "живыми" (а не *системными) пользователями* 3. Для просмотра всех *процессов* Гуревич пользуется командой ps -efH.

**Разделяемые каталоги**

Проанализировав систему *прав доступа* к каталогам в Linux, Мефодий пришел к выводу, что в ней имеется существенный недочет. Тот, кто имеет право **изменять** каталог, может **удалить любой файл** оттуда, даже такой, к которому не имеет доступа. Формально все правильно: удаление файла из каталога – всего лишь изменение содержимого каталога. Если у файла было больше одного имени ( **существовало несколько** *жестких ссылок* **на этот файл** ), **никакого удаления данных не произойдет, а если ссылка была последней – файл в самом деле удалится**. Вот это-то, по мнению Мефодия, и плохо.

Чтобы доказать новичку, что право на удаление любых файлов **полезно**, кто-то создал прямо в домашнем каталоге Мефодия файл, недоступный ему, да к тому же с подозрительным именем, содержащим пробел:

[methody@localhost methody]$ ls

4TO-TO Mep3koe Documents examples loop script tmp to.sort

[methody@localhost methody]$ ls -l 4\*

-rw------- 1 root root 0 Сен 22 22:20 4TO-TO Mep3koe

[methody@localhost methody]$ rm -i 4\*

rm: удалить защищенный от записи пустой обычный файл `4TO-TO Mep3koe'? y

6.8. Удаление чужого файла с неудобным именем

Подозревая, что от хулиганов всего можно ожидать, Мефодий не решился **набрать** имя удаляемого файла с клавиатуры, а воспользовался *шаблоном* и ключом " -i " ( **i** nteractive) команды rm, чтобы та ожидала **подтверждения** перед тем, как удалять очередной файл. Несмотря на отсутствие доступа к самому файлу, удалить его оказалось возможно:

[methody@localhost methody]$ ls -dl /tmp

drwxrwxrwt 4 root root 1024 Сен 22 22:30 /tmp

[methody@localhost methody]$ ls -l /tmp

итого 4

-rw-r--r-- 1 root root 13 Сен 22 17:49 read.all

-rw-r----- 1 root methody 23 Сен 22 17:49 read.methody

-rw------- 1 methody root 25 Сен 22 22:30 read.Methody

-rw-r----- 1 root wheel 21 Сен 22 17:49 read.wheel

[methody@localhost methody]$ rm -f /tmp/read.\*

rm: невозможно удалить '/tmp/read.all': Operation not permitted

rm: невозможно удалить '/tmp/read.methody': Operation not permitted

rm: невозможно удалить '/tmp/read.wheel': Operation not permitted

[methody@localhost methody]$ ls /tmp

read.all read.methody read.wheel

6.9. Работа с файлами в разделяемом каталоге

Убедившись, что **любой** доступ в каталог /tmp открыт **всем**, Мефодий решил удалить оттуда все файлы. Затея гораздо более хулиганская, чем **заведение** файла: а вдруг они кому-нибудь нужны? Удивительно, но удален оказался только файл, принадлежащий самому Мефодию...

Дело в том, что Мефодий проглядел особенность *атрибутов* каталога /tmp: вместо " x " в тройке "для *посторонних* " ls выдал " t ". Это **еще** один *атрибут* каталога, наличие которого как раз и запрещает пользователю удалять оттуда файлы, которым он не *хозяин*. Таким образом, права записи в каталог с *ярлыком* " drwxrwxrwt группа хозяин " и для членов *группы*, и для *посторонних* ограничены их собственными файлами, и только *хозяин* имеет право изменять список файлов в каталоге, как ему вздумается. Такие каталоги называются *разделяемыми*, потому что предназначены они, как правило, для **совместной** работы всех пользователей в системе, обмена информацией и т. п.

При установке *атрибута* " t " доступ на использование для *посторонних* (" t " в строчке *атрибутов* стоит на месте последнего " x ") не отменяется. Просто они так редко используются друг без друга, что ls выводит их в одном и том же месте. Если кому-нибудь придет в голову организовать *разделяемый каталог* без доступа *посторонним* на использование, ls выведет на месте девятого *атрибута* не " t ", а " T ":

[methody@localhost methody]$ ls -l loop

-rw-r--r-- 1 root root 26 Сен 22 22:10 loop

[methody@localhost methody]$ chown methody loop

chown: изменение владельца `loop': Operation not permitted

[methody@localhost methody]$ cp loop loopt

[methody@localhost methody]$ ls -l loop\*

-rw-r--r-- 1 root root 26 Сен 22 22:10 loop

-rw-r--r-- 1 methody methody 26 Сен 22 22:15 loopt

[methody@localhost methody]$ mv -f loopt loop

[methody@localhost methody]$ ls -l loop\*

-rw-r--r-- 1 methody methody 26 Сен 22 22:15 loop

6.10. Что можно делать с чужим файлом в своем каталоге

Оказывается, мелкие пакости продолжаются. Кто-то сменил файлу loop *хозяина*, так что теперь Мефодий может только читать его, но не изменять. Удалить этот файл – проще простого, но хочется "вернуть все как было": чтобы получился файл с тем же именем и тем же содержанием, принадлежащий Мефодию, а не root. Это несложно: чужой файл можно переименовать (это действие над каталогом, а не над файлом), скопировать переименованный файл в файл с именем старого (доступ на чтение открыт) и, наконец, удалить чужой файл с глаз долой. Ключ " -f " ( **f** orce, "силком") позволяет утилите mv делать свое дело, не спрашивая подтверждений. В частности, увидев, что файл с именем, в которое необходимо переименовывать, существует, даже чужой, даже недоступный на запись, mv преспокойно удалит его и выполнит операцию переименования.

**Суперпользователь**

Мефодий возмутился, узнав, что кто-то может проделывать **над ним** всякие штуки, которые сам Мефодий ни над кем проделывать не может. Обоснованное подозрение пало на Гуревича, единственного администратора этой системы, обладающего *правами* *суперпользователя*.

***Суперпользователь*** - единственный пользователь в Linux, на которого не распространяются ограничения *прав доступа*. Имеет нулевой *идентификатор пользователя* .

*Суперпользователь* в Linux – это **выделенный** пользователь системы, на которого **не распространяются** ограничения *прав доступа*. *UID* суперпользовательских *процессов* равен 0: так система отличает их от *процессов* других пользователей. Именно *суперпользователь* имеет возможность произвольно изменять владельца и *группу* файла. Ему открыт доступ на чтение и запись к **любому файлу** системы и доступ на чтение, запись и использование к **любому каталогу**. Наконец, суперпользовательский *процесс* может на время сменить **свой собственный** *UID* с нулевого на любой другой. Именно так и поступает программа login, когда, проведя процедуру *идентификации* пользователя, запускает *стартовый командный интерпретатор*.

Среди *учетных записей* Linux всегда есть запись по имени root ("корень"), соответствующая нулевому *идентификатору*, поэтому вместо " *суперпользователь* " часто говорят " root ". 4 Множество системных файлов принадлежат root, множество файлов только ему доступны на чтение или запись. Пароль этой учетной записи – одна из самых больших драгоценностей системы. Именно с ее помощью системные администраторы выполняют самую ответственную работу. Свойство root иметь доступ ко **всем** ресурсам системы накладывает очень высокие требования на **человека**, знающего пароль root. *Суперпользователь* может все – в том числе и все поломать, поэтому **любую работу** стоит вести с *правами* обычного пользователя, а к *правам* root прибегать только по необходимости.

Существует два различных способа получить *права* *суперпользователя*. Первый – это **зарегистрироваться в системе** под этим именем, ввести пароль и получить *стартовую оболочку*, имеющую нулевой *UID*. Это – самый неправильный способ, пользоваться которым стоит, только если нельзя применить другие. Что в этом случае выдаст команда last? Что тогда-то с такой-то консоли в систему вошел **неизвестно кто** с *правами* *суперпользователя* и что-то там такое **делал**. С точки зрения системного администратора, это очень подозрительное событие, особенно если сам он в это время к указанной консоли не подходил... Сами администраторы такого способа избегают.

Второй способ —воспользоваться специальной утилитой su ( **s** hell of **u** ser), которая позволяет выполнить одну или несколько команд от лица другого пользователя. По умолчанию эта утилита выполняет команду sh от лица пользователя root, то есть запускает командный интерпретатор с нулевым *UID*. Отличие от предыдущего способа – в том, что всегда известно, **кто именно** запускал su, а значит, ясно, с кого спрашивать за последствия. В некоторых случаях удобнее использовать не su, а утилиту sudo, которая позволяет выполнять **только заранее заданные** команды.

**Подмена идентификатора**

Утилиты su и sudo имеют некоторую странность, объяснить которую Мефодий пока не в состоянии. Эта же странность распространяется и на давно известную программу passwd, которая позволяет редактировать собственную *учетную запись*. Запускаемый *процесс* **наследует** *UID* от родительского, поэтому если этот *UID* – не нулевой, он не в состоянии **поменять** его. Тогда как же su запускает для обычного пользователя **суперпользовательский** shell? Как passwd получает доступ к хранилищу всех *учетных записей*? Должен существовать механизм, позволяющий пользователю запускать *процессы* с *идентификаторами* **другого** пользователя, причем механизм строго контролируемый, иначе с его помощью можно натворить немало бед.

В Linux этот механизм называется *подменой идентификатора* и устроен очень просто. *Процесс* может сменить свой *UID*, если запустит вместо себя при помощи exec() **другую** программу из файла, имеющего специальный *атрибут* *SetUID* 5. В этом случае *UID* *процесса* становится равным *UID* **файла**, из которого программа была запущена:

[foreigner@somewhere foreigner]$ ls -l /usr/bin/passwd /bin/su

-rws--x--x 1 root root 19400 Фев 9 2004 /bin/su

-rws--x--x 1 root root 5704 Янв 18 2004 /usr/bin/passwd

[foreigner@somewhere foreigner]$ ls -l /etc/shadow

-r-------- 1 root root 5665 Сен 10 02:08 /etc/shadow

6.11. Обычная программа passwd, использующая SetUID

Как и в случае с t- *атрибутом*, ls выводит букву " s " вместо буквы " x " в тройке "для *хозяина* ". Точно так же, если соответствующего x- *атрибута* нет (что бывает редко), ls выведет " S " вместо " s ". Во многих дистрибутивах Linux и /bin/su, и /usr/bin/passwd имеют установленный *SetUID* и принадлежат пользователю root, что и позволяет su запускать *процессы* с правами этого пользователя (а значит, и любого другого), а passwd – модифицировать файл /etc/shadow, содержащий в таких системах сведения обо всех *учетных записях*. Как правило, файлы с *атрибутом* *SetUID* доступны обычным пользователям только на выполнение, чтобы не провоцировать пользователей рассматривать содержимое этих файлов и исследовать их **недокументированные возможности**. Ведь если обнаружится способ заставить, допустим, программу passwd **выполнить** любую другую программу, то все проблемы с защитой системы от взлома будут разом решены – нет защиты, нет и проблемы.

Однако Мефодий работает с такой системой, где /usr/bin/passwd вообще не имеет *атрибута* *SetUID*. Зато эта программа принадлежит *группе* shadow и имеет другой *атрибут*, *SetGID*, так что при ее запуске *процесс* получает идентификатор *группы* shadow. Утилита ls выводит *SetGID* в виде " s " вместо " x " во второй тройке *атрибутов* ("для *группы* "). Замечания касательно " s ", " S " и " x " действительны для *SetGID* так же, как и для *SetUID*:

[root@localhost root]# ls -l /usr/bin/passwd

-rwx--s--x 1 root shadow 5704 Jan 18 2004 /usr/bin/passwd

[root@localhost root]# ls -al /etc/tcb/methody

total 3

drwx--s--- 2 methody auth 1024 Sep 22 12:58 .

drwx--x--- 55 root shadow 1024 Sep 22 18:41 ..

-rw-r----- 1 methody auth 81 Sep 22 12:58 shadow

-rw------- 1 methody auth 0 Sep 12 13:58 shadow-

-rw------- 1 methody auth 0 Sep 12 13:58 shadow.lock

6.12. Не подверженная взлому программа passwd, использующая SetGID

Каталог /etc/*tcb* в этой системе содержит подкаталоги, соответствующие *входным именам* всех ее пользователей. В каждом подкаталоге хранится, в числе прочего, собственный файл shadow соответствующего пользователя. Доступ к каталогу /etc/*tcb* на **использование** (а, следовательно, и ко всем его подкаталогам) имеют, кроме root, только члены *группы* shadow. Доступ на **запись** к каталогу /etc/*tcb*/methody и к файлу /etc/*tcb*/methody/shadow имеет только пользователь methody. Значит, чтобы изменить что-либо в этом файле, *процесс* **обязан** иметь *UID* methody и *GID* shadow (или нулевой *UID*, конечно). Именно такой *процесс* запускается из /usr/bin/passwd: он наследует *UID* у командного интерпретатора и получает *GID* shadow из-за *атрибута* *SetGID*. Выходит, что даже найдя в программе passwd ошибки и заставив ее делать что угодно, злоумышленник всего только и сможет, что... отредактировать **собственную** *учетную запись*!

Оказывается, *атрибут* *SetGID* можно присваивать каталогам. В последнем примере каталог /etc/*tcb*/methody имел *SetGID*, поэтому все создаваемые в нем файлы получают *GID*, равный *GID* самого каталога – auth. Используется это разнообразными *процессами* *идентификации*, которые, не будучи суперпользовательскими, но входя в *группу* auth и shadow, могут прочесть информацию из файлов /etc/*tcb*/входное\_имя/shadow.

Вполне очевидно, что *подмена идентификатора* распространяется на **программы**, но не работает для **сценариев**. В самом деле, при исполнении сценария *процесс* запускается не из него, а из программы-интерпретатора. Файл со сценарием передается всего лишь как параметр командной строки, и подавляющее большинство интерпретаторов просто не обращают внимания на *атрибуты* этого файла. Интерпретатор наследует *UID* у родительского *процесса*, так что если он и обнаружит *SetUID* у сценария, поделать он все равно ничего не сможет.

**Восьмеричное представление атрибутов**

Все двенадцать *атрибутов* можно представить в виде битов двоичного числа, равных 1, если *атрибут* установлен, и 0, если нет. Порядок битов в числе следующий: sU|sG|t|rU|wU|xU|rG|wG||xG|rO|wO|xO, где sU – это *SetUID*, sG – это *SetGID*, t – это t- *атрибут*, после чего следуют три тройки *атрибутов* доступа. Этим форматом можно пользоваться в команде chmod вместо конструкции " роли=виды\_доступа ", причем число надо записывать в **восьмеричной системе счисления**. Так, *атрибуты* каталога /tmp будут равны 1777, *атрибуты* /bin/su – 4711, *атрибуты* /bin/ls – 755 и т. д.

Тем же побитовым представлением *атрибутов* регулируются и *права доступа* по умолчанию при **создании** файлов и каталогов. Делается это с помощью команды umask. Единственный параметр umask – восьмеричное число, задающее *атрибуты*, которые **не надо** устанавливать новому файлу или каталогу. Так, umask 0 приведет к тому, что файлы будут создаваться с *атрибутами* " rw-rw-rw- ", а каталоги – " rwxrwxrwx ". Команда umask 022 убирает из *атрибутов* по умолчанию *права доступа* на запись для всех, кроме *хозяина* (получается " rw-r--r-- " и " rwxr-xr-x " соответственно), а с umask 077 новые файлы и каталоги становятся полностью недоступны (" rw------- " и " rwx------ ") всем, кроме их *хозяев*. 6

**Лекция 7:**

**Работа с текстовыми данными**

**Ввод и вывод**

Любая *программа* - это *автомат*, предназначенный для обработки **данных**: получая на входе одну информацию, они в результате работы выдают другую. Хотя входящая и/или выходящая *информация* может быть и нулевой, т. е. попросту отсутствовать. Те данные, которые передаются программе для обработки - это ее *ввод*, то, что она выдает в результате работы - *вывод*. Организация *ввода* и *вывода* для каждой программы - это задача операционной системы.

Каждая *программа* работает с данными определенного типа: текстовыми, графическими, звуковыми и т. п. Как, наверное, уже стало понятно из предыдущих лекций, основной *интерфейс* управления системой в Linux - это *терминал*, который предназначен для передачи текстовой информации от пользователя системе и обратно (см. лекцию 2). Поскольку ввести с *терминала* и вывести на *терминал* можно только текстовую информацию, то *ввод* и *вывод* программ, связанных с терминалом1, тоже должен быть текстовым. Однако необходимость оперировать с текстовыми данными не ограничивает возможности управления системой, а, наоборот, расширяет их. Человек может **прочитать вывод** любой программы и разобраться, что происходит в системе, а разные программы оказываются совместимыми между собой, поскольку используют один и тот же вид представления данных - текстовый. Возможностям, которые дает Linux при работе с данными в текстовой форме, и посвящена данная лекция.

"Текстовость" данных - всего лишь **договоренность** об их формате. Никто не мешает выводить на экран нетекстовый *файл*, однако пользы в этом будет мало. Во-первых, раз уж *файл* содержит не текст, то не предполагается, что человек сможет что-либо понять из его содержимого. Во-вторых, если в нетекстовых данных, выводимых на *терминал*, **случайно** встретится *управляющая последовательность*, *терминал* ее выполнит. Например, если в скомпилированной программе записано число 1528515121, оно представлено в виде четырех байтов: 27, 91, 49 и 74. Соответствующий им **текст** состоит из четырех *символов* *ASCII*: " esc ", " [ ", " 1 " и " J ", и при *выводе* файла на виртуальную *консоль* Linux в этом месте выполнится очистка экрана, так как " ^[[1J " - именно такая *управляющая последовательность* для *виртуальной консоли*. Не все *управляющие последовательности* столь безобидны, поэтому использовать нетекстовые данные в качестве текстов не рекомендуется.

Что же делать, если содержимое нетекстового файла все-таки желательно **просмотреть** (то есть превратить в **текст** )? Можно воспользоваться программой hexdump, которая выдает содержимое файла в виде шестнадцатеричных *ASCII*-кодов, или strings, показывающей только те части файла, которые **могут** быть представлены в виде текста:

[methody@localhost methody]$ hexdump -C /bin/cat | less

00000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|

00000010 02 00 03 00 01 00 00 00 90 8b 04 08 34 00 00 00 |............4...|

00000020 e0 3a 00 00 00 00 00 00 34 00 20 00 07 00 28 00 |Ю:......4. ...(.|

. . .

00000100 00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00 |................|

00000110 04 00 00 00 2f 6c 69 62 2f 6c 64 2d 6c 69 6e 75 |..../lib/ld-linu|

00000120 78 2e 73 6f 2e 32 00 00 04 00 00 00 10 00 00 00 |x.so.2..........|

00000130 01 00 00 00 47 4e 55 00 00 00 00 00 02 00 00 00 |....GNU.........|

. . .

[methody@localhost methody]$ strings /bin/cat | less

/lib/ld-linux.so.2

\_Jv\_RegisterClasses

\_\_gmon\_start\_\_

libc.so.6

stdout

. . .

[methody@localhost methody]$ strings -n3 /bin/cat | less

/lib/ld-linux.so.2

GNU

\_Jv\_RegisterClasses

\_\_gmon\_start\_\_

libc.so.6

stdout

. . .

Пример 7.1. Использование hexdump и strings

В примере Мефодий, зная заранее, что текста будет выдано много, воспользовался *конвейером* " | less ", описанным в разделе " *Конвейер* ". С ключом " -C " *утилита* hexdump выводит в правой стороне экрана текстовое *представление* данных, заменяя непечатные *символы* точками (чтобы среди выводимого текста не встретилось *управляющей последовательности* ). Мефодий заметил, что strings "не нашла" в файле /bin/cat явно текстовых подстрок " *ELF* " и " GNU ": первая из них - вообще не текст (перед ней стоит непечатный *символ* с кодом 7f, а после - *символ* с кодом 1 ), а вторая - слишком короткая, хоть и ограничена *символами* с кодом 0, как это и полагается строке в скомпилированной программе. Наименьшая *длина* строки передается strings ключом " -n ".

**Перенаправление ввода и вывода**

Для того чтобы записать данные в **файл** или прочитать их оттуда, процессу необходимо сначала **открыть** этот *файл* (при открытии на *запись*, возможно, придется предварительно создать его). При этом процесс получает *дескриптор* (описатель) открытого файла - уникальное для этого процесса число, которое он и будет использовать во всех операциях записи. Первый открытый *файл* получит *дескриптор* 0, второй - 1 и так далее. Закончив работу с файлом, процесс **закрывает** его, при этом *дескриптор* освобождается и может быть использован повторно. Если процесс завершается, не закрыв файлы, за него это делает система. Строго говоря, только в *операции* открытия дескриптора указывается, какой именно *файл* будет задействован. В качестве "файла" используются и обычные файлы, и *файлы-дырки* (чаще всего - *терминалы* ), и *каналы*, описанные в разделе " *Конвейер* ". Дальнейшие *операции* - чтение, *запись* и закрытие - работают с дескриптором, как **с потоком данных**, а куда именно ведет этот *поток*, неважно.

Каждый процесс Linux получает при старте **три** "файла", открытых для него системой. Первый из них (*дескриптор* 0 ) открыт на **чтение**, это стандартный *ввод* процесса. Именно со стандартным *вводом* работают все *операции* чтения, если в них не указан *дескриптор* файла. Второй (*дескриптор* 1 ) - открыт на **запись**, это *стандартный вывод* процесса. С ним работают все *операции* записи, если *дескриптор* файла не указан в них явно. Наконец, третий *поток* данных (*дескриптор* 2 ) предназначается для *вывода* *диагностических сообщений*, он называется ***стандартный вывод ошибок***. Поскольку эти три дескриптора уже открыты к моменту запуска процесса, первый *файл*, открытый **самим** процессом, будет, скорее всего, иметь *дескриптор* 3.

**Дескриптор** - это описатель потока данных, открытого процессом. Дескрипторы нумеруются, начиная с 0. При открытии нового потока данных его *дескриптор* получает наименьший из неиспользуемых в этот момент номеров. Три заранее открытых дескриптора - стандартный *ввод* ( 0 ), *стандартный вывод* ( 1 ) и *стандартный вывод ошибок* ( 2 ) - выдаются при запуске.

Механизм копирования *окружения*, описанный в лекции 5, подразумевает, в числе прочего, *копирование* всех открытых дескрипторов родительского процесса дочернему. В результате и родительский, и *дочерний процесс* имеют под одинаковыми дескрипторами одни и те же *потоки данных*. Когда запускается *стартовый командный интерпретатор*, все три заранее открытых дескриптора связаны у него с *терминалом* (точнее, с соответствующим *файлом-дыркой* типа tty ): *пользователь* вводит команды с клавиатуры и видит сообщения на экране. Следовательно, любая *команда*, запускаемая из командной оболочки, будет выводить на тот же *терминал*, а любая *команда*, запущенная **интерактивно** (не в фоне) - вводить оттуда.

**Стандартный вывод**

Мефодий уже сталкивался с тем, что некоторые программы умеют выводить не только на *терминал*, но и в файл. Например, info при указании *параметрического ключа* " -o " с именем файла выведет текст руководства в файл, вместо того, чтобы отображать его на мониторе. Даже если разработчиками программы не предусмотрен такой ключ, Мефодию известен и другой способ сохранить *вывод* программы в файле вместо того, чтобы выводить его на монитор: поставить знак " > " и указать после него имя файла. Таким образом Мефодий уже создавал короткие текстовые файлы ( *сценарии* ) при помощи утилиты cat (см. лекцию 5):

[methody@localhost methody]$ cat > textfile

Это файл для примеров.

^D

[methody@localhost methody]$ ls -l textfile

-rw-r--r-- 1 methody methody 23 Ноя 15 16:06 textfile

Пример 7.2. Перенаправление стандартного вывода в файл

От использования *символа* " > " возможности самой утилиты cat, конечно, не расширились. Более того, cat в этом примере не получила от **командной оболочки** никаких параметров: ни знака " > ", ни последующего имени файла. В этом случае cat работала как обычно, не зная (и даже не интересуясь!), куда попадут выведенные данные: на экран монитора, в файл или куда-нибудь еще. Вместо того, чтобы самой обеспечивать доставку *вывода* до конечного адресата (будь то человек или файл), cat отправляет все данные на *стандартный вывод* (сокращенно - stdout ).

Подмена *стандартного вывода* - задача командной оболочки (shell). В данном примере shell создает пустой файл, имя которого указано после знака " > ", и дескриптор этого файла передается программе cat под номером 1 ( *стандартный вывод* ). Делается это очень просто. В лекции 5 было рассказано о том, как запускаются команды из оболочки. В частности, после выполнения fork() появляется два одинаковых процесса, один из которых - дочерний - должен запустить вместо себя команду (выполнить exec() ). Так вот, перед этим он **закрывает стандартный вывод** (дескриптор 1 освобождается) и **открывает** файл (с ним связывается первый свободный дескриптор, т. е. 1 ), а запускаемой команде ничего знать и не надо: ее *стандартный вывод* уже подменен. Эта операция называется **перенаправлением стандартного вывода**. В том случае, если файл уже существует, shell запишет его заново, полностью уничтожив все, что в нем содержалось до этого. Поэтому Мефодию, чтобы продолжить записывать данные в textfile, потребуется другая операция - " >> ":

[methody@localhost methody]$ cat >> textfile

Пример 1.

^D

[methody@localhost methody]$ cat textfile

Это файл для примеров.

Пример 1.

[methody@localhost methody]$

Пример 7.3. Недеструктивное перенаправление стандартного вывода

Мефодий получил именно тот результат, который ему требовался: добавил в конец уже существующего файла данные со *стандартного вывода* очередной команды.

***Cтандартный вывод*** (*standard output*, stdout) - это поток данных, открываемый системой для каждого процесса в момент его запуска и предназначенный для данных, выводимых процессом.

**Стандартный ввод**

Аналогичным образом для передачи данных на вход программе может быть использован *стандартный ввод* (сокращенно - stdin). При работе с командной строкой *стандартный ввод* - это *символы*, вводимые пользователем с клавиатуры. *Стандартный ввод* можно **перенаправить** при помощи командной оболочки, подав на него данные из некоторого файла. *Символ* " < " служит для перенаправления содержимого файла на *стандартный ввод* программе. Например, если вызвать утилиту sort без параметра, она будет читать строки со *стандартного ввода*. Команда " sort < имя\_файла " подаст на *ввод* sort данные из файла:

[methody@localhost methody]$ sort < textfile

Пример 1.

Это файл для примеров.

[methody@localhost methody]$

Пример 7.4. Перенаправление стандартного ввода из файла

Результат действия этой команды аналогичен команде sort textfile - разница лишь в том, что когда используется " < ", sort получает данные со *стандартного ввода* ничего не зная о файле " textfile ", откуда они поступают. Механизм работы shell в данном случае тот же, что и при перенаправлении *вывода*: shell читает данные из файла " textfile ", запускает утилиту sort и передает ей на *стандартный ввод* содержимое файла.

Необходимо помнить, что операция " > " **деструктивна**: она всегда создает файл нулевой длины. Поэтому для, допустим, сортировки данных **в файле** надо применять последовательно sort < файл > новый\_файл и mv новый\_файл файл. Команда вида команда < файл > тот\_же\_файл просто урежет его до нулевой длины!

***Стандартный ввод*** (*standard input*, stdin) - поток данных, открываемый системой для каждого процесса в момент его запуска и предназначенный для *ввода* данных.

**Стандартный вывод ошибок**

В качестве первого примера и упражнения на перенаправление Мефодий решил записать руководство по cat в свой файл cat.info:

[methody@localhost methody]$ info cat > cat.info

info: Запись ноды (coreutils.info.bz2)cat invocation...

info: Завершено.

[methody@localhost methody]$ head -1 cat.info

File: coreutils.info, Node: cat invocation, Next: tac invocation, Up: Output of entire files

[methody@localhost methody]$

Пример 7.5. Стандартный вывод ошибок

Удивленный Мефодий обнаружил, что вопреки его указанию отправляться в файл, две строки, выведенные командой info, все равно проникли на *терминал*. Очевидно, эти строки не попали на *стандартный вывод* потому, что не относятся непосредственно к руководству, которое должна вывести программа, они информируют пользователя о **ходе выполнения** работы: записи руководства в файл. Для такого рода *диагностических сообщений*, а также для сообщений об ошибках, возникших в ходе выполнения программы, в Linux предусмотрен *стандартный вывод ошибок* (сокращенно - stderr ).

***Стандартный вывод ошибок*** (standard error, stderr) - поток данных, открываемый системой для каждого процесса в момент его запуска и предназначенный для *диагностических сообщений*, выводимых процессом.

Использование *стандартного вывода ошибок* наряду со *стандартным выводом* позволяет отделить собственно результат работы программы от разнообразной сопровождающей информации, например, направив их в разные файлы. *Стандартный вывод ошибок* может быть перенаправлен так же, как и *стандартный ввод* / *вывод*, для этого используется комбинация *символов* " 2> ":

[methody@localhost methody]$ info cat > cat.info 2> cat.stderr

[methody@localhost methody]$ cat cat.stderr

info: Запись ноды (coreutils.info.bz2)cat invocation...

info: Завершено.

[methody@localhost methody]$

Пример 7.6. Перенаправление стандартного вывода ошибок

В этот раз на *терминал* уже ничего не попало, *стандартный вывод* отправился в файл cat.info, *стандартный вывод ошибок* - в cat.stderr. Вместо " > " и " 2> " Мефодий мог бы написать " 1> " и " 2> ". Цифры в данном случае обозначают номера дескрипторов **открываемых** файлов. Если некая утилита ожидает получить **открытый** дескриптор с номером, допустим, 4, то, для того чтобы ее запустить, **обязательно** потребуется использовать сочетание " 4> ".

Иногда, однако, требуется объединить *стандартный вывод* и *стандартный вывод ошибок* в одном файле, а не разделять их. В командной оболочке bash для этого имеется специальная последовательность " 2>&1 ". Это означает "направить *стандартный вывод ошибок* туда же, куда и *стандартный вывод* ":

[methody@localhost methody]$ info cat > cat.info 2>&1

[methody@localhost methody]$ head -3 cat.info

info: Запись ноды (coreutils.info.bz2)cat invocation...

info: Завершено.

File: coreutils.info, Node: cat invocation, Next: tac invocation, Up: Output of entire files

[methody@localhost methody]$

Пример 7.7. Объединение стандартного вывода и стандартного вывода ошибок

В этом примере важен порядок перенаправлений: в командной строке Мефодий сначала указал, куда перенаправить *стандартный вывод* (" > cat.info ") и только потом велел направить туда же *стандартный вывод* ошибок. Сделай он наоборот (" 2>&1 > cat.info "), результат получился бы неожиданный: в файл попал бы только *стандартный вывод*, а *диагностические сообщения* появились бы на *терминале*. Однако логика здесь железная: на момент выполнения операции " 2>&1 " *стандартный вывод* был связан с *терминалом*, значит, **после** ее выполнения *стандартный вывод ошибок* тоже будет связан с *терминалом*. А последующее перенаправление *стандартного вывода* в файл, конечно, никак не отразится на *стандартном выводе ошибок*. Номер в конструкции " &номер " - это номер **открытого** дескриптора. Если бы упомянутая выше утилита, записывающая в четвертый дескриптор, была написана на shell, в ней бы использовались перенаправления вида " >&4 ". Чтобы не набирать громоздкую конструкцию " > файл 2>&1 " в bash используются сокращения: " &> файл " или, что то же самое, " >& файл ".

**Перенаправление в никуда**

Иногда заведомо известно, что какие-то данные, выведенные программой, не понадобятся. Например, предупреждения со *стандартного вывода ошибок*. В этом случае можно перенаправить *стандартный вывод ошибок* в *файл-дырку*, специально предназначенный для уничтожения данных - /dev/null. Все, что записывается в этот файл, просто будет выброшено и **нигде не сохранится**:

[methody@localhost methody]$ info cat > cat.info 2> /dev/null

[methody@localhost methody]$

Пример 7.8. Перенаправление в /dev/null

Точно таким же образом можно избавиться и от *стандартного вывода*, отправив его в /dev/null.

**Обработка данных в потоке**

**Конвейер**

Нередко возникают ситуации, когда нужно обработать *вывод* одной программы какой-то другой программой. Пользуясь перенаправлением *ввода* - *вывода*, можно сохранить *вывод* одной программы в файле, а потом направить этот файл на *ввод* другой программе. Однако то же самое можно сделать и более эффективно: перенаправлять *вывод* можно не только в файл, но и **непосредственно** на *стандартный ввод* другой программе. В этом случае вместо двух команд потребуется только одна - программы передают друг другу данные "из рук в руки". В Linux такой способ передачи данных называется ***конвейер*** .

В bash для перенаправления *стандартного вывода* на *стандартный ввод* другой программе служит *символ* " | ". Самый простой и наиболее распространенный случай, когда требуется использовать *конвейер*, возникает, если *вывод* программы не умещается на экране монитора и очень быстро "пролетает" перед глазами, так что человек не успевает его прочитать. В этом случае можно направить *вывод* в программу просмотра ( less ), которая позволит не торопясь пролистать весь текст, вернуться к началу и т. п.:

[methody@localhost methody]$ cat cat.info | less

Пример 7.9. Простейший конвейер

Можно последовательно обработать данные несколькими разными программами, перенаправляя *вывод* на *ввод* следующей программе и организуя сколь угодно длинный *конвейер* для обработки данных. В результате получаются очень длинные командные строки вида " cmd1 | cmd2 | ... | cmdN ", которые могут показаться громоздкими и неудобными, но оказываются очень полезными и эффективными при обработке большого количества информации, как мы увидим далее в этой лекции.

Организация *конвейера* устроена в shell по той же схеме, что и перенаправление в файл, но с использованием особого объекта системы - *канала*. Если файл можно представить в виде коробки с данными, снабженной клапаном для чтения или клапаном для записи, то *канал* - это оба клапана, приклеенные друг к другу вообще без коробки. Для определенности между клапанами можно представить трубу, немедленно доставляющую данные от выхода к входу (английский термин - "pipe" - основан как раз на этом представлении, а в роли трубы выступает, конечно же, сам Linux). *Каналом* пользуются сразу два процесса: один пишет туда, другой читает. Связывая две команды *конвейером*, shell открывает *канал* (заводится **два** дескриптора - входной и выходной), подменяет по уже описанному алгоритму *стандартный вывод* первого процесса на входной дескриптор *канала*, а *стандартный ввод* второго процесса - на выходной дескриптор *канала*. После чего остается запустить по команде в этих процессах, и *стандартный вывод* первой попадет на *стандартный ввод* второй.

***Канал*** (pipe) - неделимая пара дескрипторов (входной и выходной), связанных друг с другом таким образом, что данные, записанные во входной дескриптор, будут немедленно доступны на чтение с выходного дескриптора.

**Фильтры**

Если программа и вводит данные, и выводит, то ее можно рассматривать как трубу, в которую что-то входит и из которой что-то выходит. Обычно смысл работы таких программ заключается в том, чтобы определенным образом **обработать** поступившие данные. В Linux такие программы называют ***фильтрами***: данные проходят через них, причем что-то "застревает" в *фильтре* и не появляется на выходе, а что-то изменяется, что-то проходит сквозь *фильтр* неизменным. *Фильтры* в Linux обычно по умолчанию читают данные со *стандартного ввода*, а выводят на *стандартный вывод*. Простейшим *фильтром* Мефодий уже пользовался много раз - это программа cat: собственно, никакой "фильтрации" данных она не производит, она просто копирует *стандартный ввод* на *стандартный вывод*.

Данные, проходящие через *фильтр*, представляют собой текст: в стандартных потоках *ввода* - *вывода* все данные передаются в виде *символов*, строка за строкой, как и в *терминале*. Поэтому могут быть состыкованы при помощи *конвейера* *ввод* и *вывод* любых двух программ, поддерживающих стандартные потоки *ввода* - *вывода*. Это напоминает конструктор, все детали которого совмещаются между собой.

В любом дистрибутиве Linux присутствует набор стандартных утилит, предназначенных для работы с файловой системой и обработки текстовых данных. Многими из них Мефодий уже успел воспользоваться: это who, cat, ls, pwd, cp, chmod, id, sort и др. Мефодий уже успел заметить, что каждая из этих утилит предназначена для исполнения какой-то **одной** операции над файлами или текстом: *вывод* списка файлов в каталоге, копирование, сортировка строк, хотя каждая утилита может выполнять свою функцию по-разному, в зависимости от переданных ей ключей и параметров. При этом все они работают со стандартными потоками *ввода* / *вывода*, поэтому хорошо приспособлены для построения *конвейеров*: последовательно выполняя простые операции над потоком данных, можно решать довольно нетривиальные задачи.

Принцип комбинирования элементарных операций для выполнения сложных задач унаследован Linux от операционной системы UNIX (как и многие другие принципы). Подавляющее большинство утилит UNIX не потеряли своего значения и в Linux. Все они ориентированы на работу с данными в текстовой форме, многие являются *фильтрами*, все не имеют графического интерфейса и вызываются из командной строки. Этот пакет утилит называется coreutils.

**Структурные единицы текста**

Работу в системе Linux почти всегда можно представить как работу с текстами. Поиск файлов и других объектов системы - это получение от *системы текста* **особой** структуры - списка имен. Операции над файлами - создание, переименование, перемещение, а также сортировка, перекодировка и прочее - замену одних *символов* и строк другими либо в каталогах, либо в самих файлах. Настройка системы в Linux сводится непосредственно к работе с текстами - редактированию *конфигурационных файлов* и написанию *сценариев* (подробнее об этом см. лекции 8 и 12).

Работая с текстом в Linux, нужно принимать во внимание, что текстовые данные, передаваемые в системе, структурированы. Большинство утилит обрабатывает не непрерывный поток текста, а последовательность **единиц**. В текстовых данных в Linux выделяются следующие структурные единицы:

1. **Строки**

Строка - основная единица передачи текста в Linux. *Терминал* передает данные от пользователя системе строками (командная строка), множество утилит вводят и выводят данные построчно, при работе многих утилит одной строке соответствует один объект системы (имя файла, путь и т. п.), sort сортирует строки. Строки разделяются *символом* конца строки " \n " (newline).

1. **Поля**

В одной строке может упоминаться и больше одного объекта. Если понимать объект как последовательность *символов* из определенного набора (например, букв), то строку можно рассматривать как состоящую из слов и разделителей2. В этом случае текст от начала строки до первого *разделителя* - это первое *поле*, от первого *разделителя* до второго - второе *поле* и т. д. В качестве *разделителя* можно рассматривать любой *символ*, который не может использоваться в объекте. Например, если в пути " /home/methody " *разделителем* является *символ* " / ", то первое *поле* пусто, второе содержит слово " home ", третье - " methody ". Некоторые утилиты позволяют выбирать из строк отдельные *поля* (по номеру) и работать со строками как с таблицей.

1. **Символы**

Минимальная единица текста - *символ*. ***Символ*** - это одна буква или другой письменный знак. Стандартные утилиты Linux позволяют заменять одни *символы* другими (производить транслитерацию), искать и заменять в строках *символы* и комбинации *символов*.

*Символ* конца строки в кодировке ASCII совпадает с *управляющей последовательностью* " ^J " - "перевод строки", однако в других кодировках он может быть иным. Кроме того, на большинстве *терминалов* - но не на всех! - вслед за переводом строки необходимо выводить еще *символ* возврата каретки (" ^M "). Это вызвало путаницу: некоторые системы требуют, чтобы в конце текстового файла стояли **оба этих символа** в определенном порядке. Чтобы избежать путаницы, в UNIX (и, как следствие, в Linux) было принято единственно верное решение: содержимое файла соответствует кодировке, а при *выводе* на *терминал* концы строки преобразуются в *управляющие последовательности* согласно настройке *терминала*.

В распоряжении пользователя Linux есть ряд утилит, выполняющих элементарные операции с единицами текста: поиск, замену, разделение и объединение строк, *полей*, *символов*. Эти утилиты, как правило, имеют **одинаковое** представление о том, как определяются единицы текста: что такое строка, какие *символы* являются *разделителями* и т. п. Во многих случаях их представления можно изменять при помощи настроек. Поэтому такие утилиты легко взаимодействуют друг с другом. Комбинируя их, можно автоматизировать довольно сложные операции по обработке текста.

**Примеры задач**

Этот раздел посвящен нескольким примерам использования стандартных утилит для решения разных типичных (и не очень) задач. Примеры не следует воспринимать как исчерпывающий *список* возможностей, они приведены просто для демонстрации того, как можно организовать обработку данных при помощи *конвейера*. Чтобы освоить их, нужно читать руководства и экспериментировать.

**Подсчет**

В европейской культуре очень большим авторитетом пользуются точные числа и количественные оценки. Поэтому пользователю часто бывает любопытно и даже необходимо точно посчитать что-нибудь многочисленное. Компьютер как нельзя лучше подходит для такой процедуры. Стандартная утилита для подсчета строк, слов и *символов* - wc (от англ. "word count" - "подсчет слов"). Мефодий запомнил, что в Linux многое можно представить как слова и строки, и решил таким образом посчитать свои файлы:

[methody@localhost methody]$ find . | wc -l

42

[methody@localhost methody]$

Пример 7.10. Подсчет файлов при помощи find и wc

Мефодий получил желаемое число - " 42 ". Для этого ему потребовалась команда find - рекомендованный Гуревичем инструмент поиска нужных файлов в системе. Мефодий вызвал find с одним параметром - каталогом, с которого надо начинать поиск. find выводит список найденных файлов по одному на строку, а поскольку критерии поиска в данном случае не уточнялись, то find просто вывела список всех файлов во всех подкаталогах текущего каталога (домашнего каталога Мефодия). Этот список Мефодий передал утилите wc, попросив ее посчитать количество полученных строк " -l ". В ответ wc выдала искомое число.

Задав find критерии поиска, можно посчитать и что-нибудь менее тривиальное, например, файлы, которые создавались или были изменены в определенный промежуток времени, файлы с определенным режимом доступа, с определенным именем и т. п. Узнать обо всех возможностях поиска при помощи find и подсчета при помощи wc можно из руководств по этим программам.

**Отбрасывание ненужного**

Иногда пользователя интересует только часть из тех данных, которые собирается выводить программа. Мефодий уже пользовался утилитой head, которая нужна, чтобы вывести только первые несколько строк файла. Не менее полезна утилита tail (англ. "хвост"), выводящая только последние строки файла. Если же пользователя интересует только определенная часть каждой строки файла - поможет утилита cut.

Допустим, Мефодию потребовалось получить список всех файлов и подкаталогов в " /etc ", которые принадлежат группе " adm ". И при этом ему почему-то нужно, чтобы найденные файлы в списке были представлены не полным путем, а только именем файла (скорее всего, это требуется для последующей автоматической обработки полученного списка):

[methody@localhost methody]$ find /etc -maxdepth 1 -group adm 2> /dev/null \

> | cut -d / -f 3

syslog.conf

anacrontab

[methody@localhost methody]$

Пример 7.11. Извлечение отдельного поля

Если команда получается такой длинной, что ее неудобно набирать в одну строку, можно разбить ее на несколько строк, не передавая системе: для этого в том месте, где нужно продолжить набор со следующей строки, достаточно поставить *символ* " \ " и нажать Enter. При этом в начале строки bash выведет *символ* " > ", означающий, что команда еще не передана системе и набор продолжается3. Для системы безразлично, в сколько строк набрана команда, возможность набора в несколько строк нужна только для удобства пользователя.

Мефодий получил нужный результат, задав параметры find - каталог, где нужно искать и параметр поиска - наибольшую допустимую глубину вложенности и группу, которой должны принадлежать найденные файлы. Ненужные *диагностические сообщения* о запрещенном доступе он отправил в /dev/null, а потом указал утилите cut, что в полученных со *стандартного ввода* строках нужно считать *разделителем* *символ* " / " и вывести только третье *поле*. Таким образом, от строк вида " /etc/filename " осталось только " filename " 4.

**Выбор нужного**

**Поиск**

Зачастую пользователю нужно найти только упоминания чего-то конкретного среди данных, выводимых утилитой. Обычно эта задача сводится к поиску строк, в которых встречается определенное слово или комбинация *символов*. Для этого подходит стандартная утилита grep. grep может искать строку в файлах, а может работать как *фильтр*: получив строки со *стандартного ввода*, она выведет на *стандартный вывод* только те строки, где встретилось искомое сочетание *символов*. Мефодий решил поинтересоваться процессами bash, которые выполняются в системе:

[methody@susanin methody]$ ps aux | grep bash

methody 3459 0.0 3.0 2524 1636 tty2 S 14:30 0:00 -bash

methody 3734 0.0 1.1 1644 612 tty2 S 14:50 0:00 grep bash

Пример 7.12. Поиск строки в выводе программы

Первый аргумент команды grep - та строка, которую нужно искать в *стандартном вводе*, в данном случае это " bash ", а поскольку ps выводит сведения по строке на каждый процесс, Мефодий получил только процессы, в имени которых есть " bash ". Однако Мефодий неожиданно нашел больше, чем искал: в списке выполняющихся процессов присутствовали две строки, в которых встретилось слово " bash ", т. е. два процесса: один - искомый - командный интерпретатор bash, а другой - процесс поиска строки " grep bash ", запущенный Мефодием **после** ps. Это произошло потому, что после разбора командной строки bash запустил **оба** дочерних процесса, чтобы организовать *конвейер*, и на момент выполнения команды ps процесс grep bash уже был запущен и тоже попал в *вывод* ps. Чтобы в этом примере получить правильный результат, Мефодию следовало бы добавить в *конвейер* еще одно звено: | grep -v grep, эта команда **исключит** из конечного *вывода* все строки, в которых встречается " grep ".

**Поиск по регулярному выражению**

Очень часто точно не известно, какую именно комбинацию *символов* нужно будет найти. Точнее, известно только то, как примерно должно выглядеть искомое слово, что в него должно входить и в каком порядке. Так обычно бывает, если некоторые фрагменты текста имеют строго определенный формат. Например, в руководствах, выводимых программой info, принят такой формат ссылок: " \*Note название\_узла::". В этом случае нужно искать не конкретное сочетание *символов*, а "Строку " \*Note ", за которой следует название узла (одно или несколько слов и пробелов), оканчивающееся *символами* " ::"". Компьютер вполне способен выполнить такой запрос, если его сформулировать на строгом и понятном ему языке, например, на языке *регулярных выражений*. ***Регулярное выражение*** - это способ одной формулой задать все последовательности *символов*, подходящие пользователю:

[methody@susanin methody]$ info grep > grep.info 2> /dev/null

[methody@susanin methody]$ grep -no "\\*Note[^:]\*::" grep.info

324:\*Note Grep Programs::

684:\*Note Invoking::

[methody@susanin methody]$

Пример 7.13. Поиск ссылок в файле info

Первый параметр grep, который взят в кавычки - это и есть *регулярное выражение* для поиска ссылок в формате info, второй параметр - имя файла, в котором нужно искать. Ключ " -o " заставляет grep выводить строку не целиком, а только ту часть, которая совпала с *регулярным выражением* (шаблоном поиска), а " -n " - выводить номер строки, в которой встретилось данное совпадение.

В *регулярном выражении* большинство *символов* обозначают сами себя, как если бы мы искали обыкновенную текстовую строку, например, " Note " и " ::" в *регулярном выражении* соответствуют строкам " Note " и " ::" в тексте. Однако некоторые *символы* обладают специальным значением, самый главный из таких *символов* - звездочка (" \* "), поставленная после элемента *регулярного выражения*, обозначает, что могут быть найдены тексты, где этот элемент повторен любое количество раз, в том числе и ни одного, т. е. просто отсутствует. В нашем примере звездочка встретилась дважды: в первый раз нужно было включить в *регулярное выражение* именно *символ* "звездочка", для этого потребовалось лишить его специального значения, поставив перед ним " \ ".

Вторая звездочка обозначает, что стоящий перед ней элемент может быть повторен любое количество раз от нуля до бесконечности. В нашем случае звездочка относится к выражению в квадратных скобках - " [^:] ", что означает "любой *символ*, кроме " :"". Целиком *регулярное выражение* можно прочесть так: "Строка " \*Note ", за которой следует ноль или больше любых *символов*, кроме " :", за которыми следует строка " ::"". Особенность работы " \* " состоит в том, что она пытается выбрать совпадение максимальной длины. Именно поэтому элемент, к которому относилась " \* ", был задан как "не " :"". Выражение "ноль или более **любых символов** " (оно записывается как " .\* ") в случае, когда, например, в одной строке встречается **две** ссылки, вбирает подстроку от конца **первого** " \*Note " до начала **последнего** " ::" ( *символы* " :", поместившиеся внутри этой подстроки, распознаются как "любые").

На языке *регулярных выражений* можно также обозначить "любой *символ* " (" ."), "одно или более совпадений" (" + "), начало и конец строки (" ^ " и " $ " соответственно) и т. д. Благодаря *регулярным выражениям* можно автоматизировать очень многие задачи, которые в противном случае потребовали бы огромной и кропотливой работы человека. Более подробные сведения о возможностях языка *регулярных выражений* можно получить из руководства regex(7). Однако руководство - это не учебник, поэтому чтобы научиться экономить время и усилия при помощи *регулярных выражений*, полезно прочесть соответствующие главы книг [4], [10].

*Регулярные выражения* в Linux используются не только для поиска программой grep. Очень многие программы, так или иначе работающие с текстом, в первую очередь текстовые редакторы, поддерживают *регулярные выражения*. К таким программам относятся два "главных" текстовых редактора Linux - *Vim* и Emacs, о которых речь пойдет в следующей лекции. Однако нужно учитывать, что в разных программах используются разные диалекты языка *регулярных выражений*, где одни и те же понятия имеют разные обозначения, поэтому всегда нужно обращаться к руководству по конкретной программе.

В заключение можно сказать, что *регулярные выражения* позволяют резко повысить эффективность работы, хорошо интегрированы в рабочую среду в системе Linux, и есть смысл потратить время на их изучение.

**Замены**

Удобство работы с потоком не в последнюю очередь состоит в том, что можно не только выборочно передавать результаты работы программ, но и автоматически заменять один текст другим прямо в потоке.

Для замены одних *символов* другими предназначена утилита tr (сокращение от англ. "translate" - "преобразовывать, переводить"), работающая как *фильтр*. Мефодий решил применить ее прямо по назначению и выполнить при ее помощи транслитерацию - замену латинских *символов* близкими по звучанию русскими:

[methody@localhost methody]$ cat cat.info | tr

abcdefghijklmnopqrstuvwxyz абцдефгхийклмнопкрстуввсиз \

> | tr ABCDEFGHIJKLMNOPRSTUVWXYZ АБЦДЕФГХИЙКЛМНОПКРСТУВВСИЗ | head -4

Филе: цореутилс.инфо, Ноде: цат инвоцатион, Нест: тац инвоцатион,

Тп: Оутпут оф ентире филес

'цат': Цонцатенате анд врите филес

===================================

[methody@localhost methody]$

Пример 7.14. Замена символов (транслитерация)

Мефодий потрудился, составляя два параметра для утилиты tr: соответствия латинских букв кириллическим. Первый *символ* из первого параметра tr заменяет первым *символом* второго, второй - вторым и т. д. Мефодий обработал поток *фильтром* tr дважды: сначала чтобы заменить строчные буквы, а затем - прописные. Он мог бы сделать это и за один проход (просто добавив к параметрам прописные после строчных), но не захотел выписывать столь длинные строки. Полученному на выходе тексту вряд ли можно найти практическое применение, однако транслитерацию можно употребить и с пользой. Если не указать tr второго параметра, то все *символы*, перечисленные в первом, будут заменены на "ничто", т. е. попросту удалены из потока. При помощи tr можно также удалить дублирующиеся *символы* (например, лишние пробелы или переводы строки), заменить пробелы переводами строк и т. п.

Помимо простой замены отдельных *символов*, возможна замена последовательностей (слов). Специально для этого предназначен *потоковый редактор* sed (сокращение от англ. "*stream editor*"). Он работает как *фильтр* и выполняет редактирование поступающих строк: замену одних последовательностей *символов* другими, причем можно заменять и *регулярные выражения*.

Например, Мефодий с помощью sed может сделать более понятным для непривычного читателя список файлов, выводимый ls:

[methody@localhost methody]$ ls -l | sed s/^-[-rwx]\*/Файл:/ | sed s/^d[-rwx]\*/Каталог:/

итого 124

Файл: 1 methody methody 2693 Ноя 15 16:09 cat.info

Файл: 1 methody methody 69 Ноя 15 16:08 cat.stderr

Каталог: 2 methody methody 4096 Ноя 15 12:56 Documents

Каталог: 3 methody methody 4096 Ноя 15 13:08 examples

Файл: 1 methody methody 83459 Ноя 15 16:11 grep.info

Файл: 1 methody methody 26 Ноя 15 13:08 loop

Файл: 1 methody methody 23 Ноя 15 13:08 script

Файл: 1 methody methody 33 Ноя 15 16:07 textfile

Каталог: 2 methody methody 4096 Ноя 15 12:56 tmp

Файл: 1 methody methody 32 Ноя 15 13:08 to.sort

[methody@localhost methody]$

Пример 7.15. Замена по регулярному выражению

У sed очень широкие возможности, но довольно непривычный синтаксис, например, замена выполняется командой " s/что\_заменять/на\_что\_заменять/ ". Чтобы в нем разобраться, нужно обязательно прочесть руководство sed(1) и знать *регулярные выражения*.

**Упорядочивание**

Для того чтобы разобраться в данных, нередко требуется их упорядочить: по алфавиту, по номеру, по количеству употреблений. Основной инструмент для упорядочивания - утилита sort - уже знакома Мефодию. Однако теперь он решил использовать ее в сочетании с несколькими другими утилитами:

[methody@localhost methody]$ cat grep.info | tr "[:upper:]" "[:lower:]" | tr "[:space:][:punct:]" "\n" \

> | sort | uniq -c | sort -nr | head -8

15233

720 the

342 of

251 to

244 a

213 and

180 or

180 is

[methody@localhost methody]$

Пример 7.16. Получение упорядоченного по частотности списка словоупотреблений

Мефодий (вернее, компьютер по плану Мефодия) подсчитал, сколько раз какие слова были употреблены в файле " grep.info " и вывел несколько самых частоупотребляемых с указанием количества употреблений в файле. Для этого потребовалось сначала заменить все большие буквы маленькими, чтобы не было разных способов написания одного слова, затем заменить все пробелы и знаки препинания концом строки ( *символ* " \n "), чтобы в каждой строке было ровно по одному слову (Мефодий всюду взял параметры tr в кавычки, чтобы bash не понял их неправильно). Потом список был отсортирован, все повторяющиеся слова заменены одним словом с указанием количества повторений (" uniq -c "), затем строки снова отсортированы по убыванию чисел в начале строки (" sort -nr ") и выведены первые 8 строк (" head -8 ").

**Запуск команд**

Полученные в *конвейере* данные можно превратить в руководство к действию для компьютера. Например, для каждой полученной со *стандартного ввода* строки можно запустить какую-нибудь команду, передав ей эту строку в качестве параметра. Для этой цели служит утилита xargs:

[methody@localhost methody]$ find /bin -type f -perm +a=x \

> | xargs grep -l -e '^#!/' 2> /dev/null

/bin/egrep

/bin/fgrep

/bin/unicode\_start

/bin/bootanim

[methody@localhost methody]$

Пример 7.17. Поиск всех исполняемых файлов, которые точно являются сценариями

Здесь Мефодий решил определить, какие из исполняемых файлов в каталоге " /bin " являются *сценариями*. Для этого он нашел все обычные исполняемые файлы (указывать " -type f " - "обычные файлы" потребовалось, чтобы в результат не попали каталоги, которые обычно являются исполняемыми), а затем для каждого найденного файла вызвал grep, чтобы поискать в нем сочетание *символов* " #!/ " в начале строки. Ключ " -l " велел grep выводить не обнаруженную строку, а имя файла, в котором найдено совпадение. Так Мефодий получил список исполняемых файлов, в которых есть строка с указанием интерпретатора - несомненных сценариев5.

**Лекция 8:**

**Возможности командной оболочки**

**Редактирование ввода**

Некоторое время поработав в Linux и понабирав команды в командной строке, Мефодий пришел к выводу, что в "общении" с оболочкой не помешают кое-какие удобства. Одно из таких удобств - возможность редактировать вводимую строку с помощью клавиши Backspace (удаление последнего символа), " ^W " (удаление слова) и " ^U " (удаление всей строки) - предоставляет сам *терминал* Linux. Эти команды работают для **любого** построчного ввода: например, если запустить программу cat без параметров, чтобы та немедленно отображала вводимые с терминала строки. Если по каким-то причинам в строчку на экране попало что-то лишнее, можно нажать " ^R " (redraw) - система выведет в новой строке содержимое входного буфера.

Мефодий не забыл, что cat без параметров следует завершать командой " ^D " (конец ввода). Эту команду, как и предыдущие, интерпретирует при вводе с терминала система. И она же превращает некоторые другие *управляющие* символы (например, " ^C " или " ^Z ") в *сигналы*. В действительности все *управляющие* символы, интерпретируемые системой, можно перенастроить с помощью команды stty. Полный *список* того, что можно настраивать, выдает *команда* stty -a:

[methody@localhost methody]$ stty -a

localhost 38400 baud; rows 30; columns 80; line = 0;

intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = ;

eol2 = ; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;

lnext = ^V; flush = ^O; min = 1; time = 0;

-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts

-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr

icrnl ixon -ixoff

-iuclc -ixany -imaxbel -iutf8

opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0

tab0 bs0 vt0 ff0

isig icanon iexten echo echoe echok -echonl -noflsh -xcase

-tostop -echoprt

echoctl echoke

Пример 8.1. Настройки терминальной линии

При виде столь обширных возможностей Мефодий немедленно взялся читать руководство ( man stty ), однако нашел в нем не так уж много для себя полезного. Из управляющих символов (строки со второй по четвертую) интересны " ^S " и " ^Q ", с помощью которых можно, соответственно, приостановить и возобновить выдачу на *терминал* (если текста вывелось уже много, а прочесть его не успеваешь). Можно заметить, что настройка erase (удаление одного символа) соответствует управляющему символу, который возвращается клавишей Backspace именно *виртуальной консоли* Linux - " ^?". На многих терминалах клавиша Backspace возвращает **другой** символ - " ^H ". Если необходимо **переопределить** настройку erase, можно воспользоваться командой " stty erase ^H ", причем " ^H " (для удобства) разрешено вводить и как два символа: " ^ " и " H ".

Наконец, чтобы **лишить** передаваемый символ его управляющих функций (если, например, требуется передать программе на ввод **символ** с кодом 3, т. е. " ^C "), непосредственно перед вводом этого символа нужно подать команду " ^V " ( lnext ):

[methody@localhost methody]$ cat | hexdump -C

Сейчас нажмем Ctrl+C

[methody@localhost methody]$ cat | hexdump -C

Теперь Ctrl+V, Ctrl+C, enter и Ctrl+D^C

00000000 f4 c5 d0 c5 d2 d8 20 43 74 72 6c 2b 56 2c 20 43 |Теперь Ctrl+V, C|

00000010 74 72 6c 2b 43 2c 20 45 6e 74 65 72 20 c9 20 43 |trl+C, enter и C|

00000020 74 72 6c 2b 44 03 0a |trl+D..|

00000027

Пример 8.2. Экранирование управляющих символов

Здесь Мефодий прервал, как и собирался, работу первого из cat. При этом до hexdump, *фильтра*, переводящего *входной* *поток* в шестнадцатеричное *представление*, дело даже не дошло, потому что cat не успел обработать ни одной строки. Во втором случае " ^C " после " ^V " потеряло управляющий смысл и отобразилось при вводе. С ключом " -C " hexdump выводит также и текстовое *представление* входного потока, заменяя непечатные символы точками. Так, на точки были заменены и " ^C " (ASCII-код 03 ), и возвращаемый Enter *символ конца строки* (ASCII-код 0a, в десятичном виде - 10 ). Ни " ^V ", ни " ^D " на вход hexdump, конечно, не попали: их, как *управляющие*, обработала система.

Прочие настройки stty относятся к обработке текста при выводе на *терминал* и вводе с него. Они интересны только в том смысле, что при их изменении работать с командной оболочкой становится неудобно. Например, настройка echo определяет, будет ли **система** отображать на экране все, что вводит *пользователь*. При включенном echo нажатие любой алфавитно-цифровой клавиши (ввод символа) приводит к тому, что система (устройство типа tty ) **выведет** этот символ на *терминал*. Настройка отключается, когда с клавиатуры вводится *пароль*. При этом трудно отделаться от ощущения, что ввода с клавиатуры не происходит. Еще хуже обстоит дело с настройками, состоящими из кусков вида " i ", " o ", " cr " и " nl ". Эти настройки управляют преобразованием при вводе и выводе исторически сложившегося обозначения конца строки **двумя** символами в **один**, принятый в Linux. Может случиться так, что клавиша Enter терминала возвращает как раз неправильный *символ конца строки*, а преобразование отключено. Тогда вместо Enter следует использовать " ^J " - символ, на **самом деле** соответствующий концу строки.

Во всех случаях, когда *терминал* находится в непонятном состоянии - не реагирует на Enter, не показывает ввода, не удаляет символов, выводит текст "ступеньками" и т. п., рекомендуется "лечить" настройки терминала с помощью stty sane - специальной формы stty, сбрасывающей настройки терминала в некоторое пригодное к работе состояние. Если непонятное состояние терминала возникло однократно, например, после аварийного завершения экранной программы (редактора *vim* или оболочки mc ), то можно воспользоваться командой reset. Она заново настраивает *терминал* в полном соответствии с системной конфигурацией (указанной в файле /etc/*inittab*, см. лекцию 10) и terminfo 1.

**Редактирование командной строки**

Даже не изучая специально возможностей командной оболочки, Мефодий активно использовал некоторые из них, недоступные при вводе текста **большинству** утилит (в частности, ни cat, ни hexdump ). Речь идет о клавишах Стрелка влево и Стрелка вправо, с помощью которых можно перемещать курсор по командной строке, и клавише Del, удаляющей символ **под** курсором, а не позади него. В лекции 2 он уже убедился, что эти команды работают в bash, но не работают для cat. Более того, в простом командном интерпретаторе - sh - они тоже не работают.

Следовательно, возможности редактора командной строки специфичны для разных командных оболочек. Однако самые необходимые команды редактирования поддерживаются во всех разновидностях shell сходным образом. По словам Гуревича "во всех видах Linux обязательно есть bash, а если ты достаточно опытен, чтобы устанавливать и настраивать пакеты, можешь установить zsh, у него возможностей больше, чем может понадобиться одному человеку". Поэтому Мефодий занялся изучением документации по bash, что оказалось делом непростым, ибо в bash.info он насчитал более восьми с половиной тысяч строк. Даже про *редактирование командной строки* написано столько, что за один раз прочесть трудно.

Попытка "наскоком" узнать **все** про работу в командной строке принесла некоторую пользу. Во-первых, перемещаться в командной строке можно не только по одному символу вперед и назад, но и по словам: команды escF/escB или Alt+F/Alt+B соответственно (от forward и backward), работают также клавиши home и end, или, что то же самое, " ^A " и " ^E ". А во-вторых, помимо работы с **одной** командной строкой, существует еще немало других удобств, о которых и пойдет речь в этой лекции.

**История команд**

Двумя другими клавишами со стрелками - вверх и вниз - Мефодий тоже активно пользовался, не подозревая, что задействует этим весьма мощный механизм bash - работу с *историей команд*. Все команды, набранные пользователем, bash запоминает и позволяет обращаться к ним впоследствии. По стрелке вверх (можно использовать и " ^P ", previous), список поданных команд "прокручивается" от последней к первой, а по стрелке вниз (" ^N ", next) - обратно. Соответствующая команда отображается в командной строке как только что набранная, ее можно отредактировать и подать оболочке (подгонять курсор к концу строки при этом необязательно).

Если необходимо добыть из истории какую-то давнюю команду, проще не гонять список истории стрелками, а выполнить поиск с помощью команды " ^R " (reverse search). При этом выводится подсказка специального вида ("reverse-i-search"), подстрока поиска (окруженная символами ` и ' ) и последняя из команд в истории, в которой эта подстрока присутствует:

[methody@localhost methody]$

^R | (reverse-i-search)`':

i | (reverse-i-search)`i': ls i

n | (reverse-i-search)`in': info

f | (reverse-i-search)`inf': info

o | (reverse-i-search)`info': info

^R | (reverse-i-search)`info': man info

^R | (reverse-i-search)`info': info "(bash.info.bz2)Commands For History"

Пример 8.3. Поиск по истории команд

Пример представляет символы, вводимые Мефодием (в левой части до " | "), и содержимое последней строки терминала. Это "кадры" работы с одной и той же строкой, показывающие, как она меняется при наборе. Набрав "info", Мефодий продолжил поиск этой подстроки, повторяя " ^R " до тех пор, пока не наткнулся на нужную ему команду, содержащую подстроку " info ". Осталось только передать ее bash с помощью Enter.

Чтобы *история команд* могла сохраняться **между** сеансами работы пользователя, bash записывает ее в файл .bash\_history, находящийся в домашнем каталоге пользователя. Делается это в момент **завершения** оболочки: накопленная за время работы история дописывается в конец этого файла. При следующем запуске bash считывает .bash\_history целиком. История хранится не вечно, количество запоминаемых команд в .bash\_history ограничено (обычно сохраняется 500 команд, но это можно и перенастроить).

**Сокращения**

Поиск по истории - удобное средство: длинную командную строку можно не набирать целиком, а отыскать и использовать. Однако **давнюю** команду придется добывать с помощью нескольких " ^R " - а можно и совсем не доискаться, если она уже выбыла оттуда. Для того чтобы оперативно заменять длинные команды короткими, стоит воспользоваться **сокращениями** (aliases). В *конфигурационных файлах* командного интерпретатора пользователя обычно уже определено несколько сокращений, список которых можно посмотреть с помощью команды alias без параметров:

[methody@localhost methody]$ alias

alias cd..='cd ..'

alias cp='cp -i'

alias l='ls -lapt'

alias ll='ls -laptc'

alias ls='ls --color=auto'

alias md='mkdir'

alias mv='mv -i'

alias rd='rmdir'

alias rm='rm -i'

Пример 8.4. Просмотр заранее определенных сокращений

С сокращениями Мефодий уже сталкивался в примерах, приведенных в лекции 6, где команда ls отказалась работать в согласии с теорией. Выяснилось, что по команде ls вместо **утилиты** /bin/ls bash запускает собственную *команду-сокращение*, превращающееся в команду ls --color=auto. **Повторно** появившуюся в команде подстроку " ls " интерпретатор уже не обрабатывает, во избежание вечного цикла. Например, команда ls -al превращается в результате в ls --color=auto -al. Точно так же любая команда, начинающаяся с rm, превращается в rm -i (interactive), что Мефодия крайне раздражает, потому что ни одно удаление не обходится без вопросов в стиле " rm: удалить обычный файл `файл'?":

[methody@localhost methody]$ unalias cp rm mv

[methody@localhost methody]$ alias pd=pushd

[methody@localhost methody]$ alias pp=popd

[methody@localhost methody]$ pd /bin

/bin ~

[methody@localhost bin]$ pd /usr/share/doc

/usr/share/doc /bin ~

[methody@localhost doc]$ cd /var/tmp

[methody@localhost tmp]$ dirs

/var/tmp /bin ~

[methody@localhost tmp]$ pp

/bin ~

[methody@localhost bin]$ pp

~

[methody@localhost methody]$ pp

-bash: popd: directory stack empty

Пример 8.5. Использование сокращений и pushd/popd

От надоедливого " -i " Мефодий избавился с помощью команды unalias, а заодно ввел сокращения для полюбившихся ему команд bash - pushd и popd. Эти команды2, подобно cd, меняют текущий каталог. Разница состоит в том, что pushd все каталоги, которые пользователь делает текущими, запоминает в особом списке (стеке). Команда popd удаляет последний элемент этого стека и делает текущим каталогом предпоследний. Обе команды вдобавок выводят содержимое стека каталогов (то же самое делает и команда dirs ). Команда cd в bash также работает со стеком каталогов: она **заменяет** его последний элемент новым.

***Команда-сокращение*** (alias) - *внутренняя команда* shell, задаваемая пользователем. Обычно заменяет одну более длинную команду, которая часто используется при работе в командной строке. Сокращения не наследуются с *окружением* .

**Достраивание**

Сокращения позволяют быстро набирать **команды**, однако никак не затрагивают имен **файлов**, которые чаще всего и оказываются параметрами этих команд. Бывает, что набранной строки - пути к файлу и нескольких первых букв его имени - достаточно для **однозначного** указания на этот файл, потому что по введенному пути других файлов, чье имя начинается на эти буквы, просто нет. Чтобы не дописывать оставшиеся буквы (а имена файлов в Linux могут быть весьма длинными), Гуревич посоветовал Мефодию нажать клавишу Tab. И - о чудо! - bash **сам** достроил начало имени файла до целого (снова воспользуемся методом "кадров"):

[methody@localhost methody]$ ls -al /bin/base

Tab | [methody@localhost methody]$ ls -al /bin/basename

-rwxr-xr-x 1 root root 12520 Июн 3 18:29 /bin/basename

[methody@localhost methody]$ base

Tab | [methody@localhost methody]$ basename

Tab | [methody@localhost methody]$ basename ex

Tab | [methody@localhost methody]$ basename examples/

Tab | [methody@localhost methody]$ basename

examples/-filename-with-

-filename-with-

Пример 8.6. Использование достраивания

Дальше - больше. Оказывается, и имя команды можно вводить не целиком: оболочка достроит набираемое слово именно до команды, раз уж это слово стоит в начале командной строки. Таким образом, команду *basename* examples/-filename-with- Мефодий набрал за **восемь** нажатий на клавиатуру (" base " и четыре Tab )! Ему не пришлось вводить начало имени файла в каталоге examples, потому что файл там был всего один.

Выполняя *достраивание* (completion), bash может вывести не всю строку, а только ту ее часть, относительно которой у него нет сомнений. Если дальнейшее *достраивание* может пойти **несколькими** путями, то однократное нажатие Tab приведет к тому, что bash растерянно пискнет3, а повторное - к выводу **под** командной строкой списка всех возможных вариантов. В этом случае надо подсказать командной оболочке продолжение: дописать несколько символов, определяющих, по какому пути пойдет *достраивание*, и снова нажать Tab.

Поиск ключевого слова "completion" по документации bash выдал так много информации, что Мефодий обратился к Гуревичу за помощью. Однако тот ответил, что не использует bash, и поэтому не в состоянии объяснять тонкости его настройки. Если в bash предусмотрено **несколько** типов *достраивания* (по именам файлов, по именам команд и т. п.), то в zsh их **сколько угодно**: существует способ запрограммировать любой алгоритм *достраивания* и задать *шаблон* командной строки, в которой именно этот способ будет применяться.

**Генерация имен файлов**

*Достраивание* очень удобно, когда цель пользователя - задать **один** конкретный *файл* в командной строке. Если же нужно работать сразу с **несколькими** файлами - например, для перемещения их в другой каталог с помощью mv, *достраивание* не помогает. Необходим способ задать одно "общее" имя для группы файлов, с которыми будет работать *команда*. В подавляющем большинстве случаев это можно сделать при помощи *шаблона*.

**Шаблоны**

*Шаблон* в командном интерпретаторе используется примерно в тех же целях, что и *регулярное выражение*, упомянутое в лекции 7: для поиска строк определенной структуры среди множества разнообразных строк. В отличие от *регулярного выражения*, *шаблон* всегда примеряется к строке целиком, кроме того, он устроен значительно проще (а значит, и беднее).

Символы в *шаблоне* разделяются на обычные и **специальные**. Обычные символы соответствуют таким же символам в строке, а специальные - обрабатываются особым образом:

* *Шаблону*, состоящему только из обычных символов, соответствует **единственная** строка, состоящая из тех же символов в том же порядке. Например, *шаблону* " abc " соответствует строка abc, но не aBc или ABC, потому что большие и маленькие буквы различаются.
* *Шаблону*, состоящему из единственного *спецсимвола* " \* ", соответствует любая строка любой длины (в том числе и пустая).
* *Шаблону*, состоящему из единственного *спецсимвола* " ?", соответствует **любая** строка длиной в один символ, например, a, + или @, но не ab или 8888.
* *Шаблону*, состоящему из любых символов, заключенных в квадратные скобки " [ " и " ] " соответствует строка длиной в **один** символ, причем этот символ должен **встречаться** среди заключенных в скобки. Например, *шаблону* " [bar] " соответствуют только строки a, b и r, но не c, B, bar или ab. Символы внутри скобок можно не перечислять полностью, а задавать **диапазон**, в начале которого стоит символ с наименьшим ASCII-кодом, затем следует " - ", а затем - символ с наибольшим ASCII-кодом. Например, *шаблону* " [0-9a-fA-F] " соответствует одна шестнадцатеричная цифра (скажем, 5, e или C ). Если после " [ " в *шаблоне* следует " !", то ему соответствует строка из одного символа, **не** перечисленного между скобками.
* *Шаблону*, состоящему из **нескольких** частей, соответствует строка, которую можно разбить на столько же подстрок (возможно, пустых), причем первая подстрока будет отвечать первой части *шаблона*, вторая - второй и т. д. Например, *шаблону* " a\*b?c " будут соответствовать строки ab@c (" \* " соответствует пустая подстрока), a+b=c и aaabbc, но не будут соответствовать abc (" ?" соответствует подстрока c, а для " c " соответствия не находится), @ab@c (нет соответствия для " a ") или aaabbbc (из трех b первое соответствует " b ", второе - " ?", а вот третье приходится на " c ").

***Шаблон*** (pattern) - строка специального формата, используемая в процедурах текстового поиска. Говорят, что строка **соответствует** *шаблону*, если можно по определенным правилам каждому символу строки поставить в соответствие символ *шаблона*. В Linux наиболее популярны *шаблоны* в формате командного интерпретатора и *регулярные выражения* .

**Использование шаблонов**

*Шаблоны* используются в нескольких конструкциях shell. Главное место их применения - командная строка. Если оболочка "видит" в командной строке *шаблон*, она немедленно заменяет его списком **файлов**, имена которых ему соответствуют. Команда, которая затем вызывается, получает в качестве параметров список файлов уже без всяких *шаблонов*, как если бы этот список пользователь ввел вручную. Эта способность командного интерпретатора называется ***генерацией имен файлов*** :

[methody@localhost methody]$ ls .bash\*

.bash\_history .bash\_logout .bash\_profile .bashrc

[methody@localhost methody]$ /bin/e\*

/bin/ed /bin/egrep /bin/ex

[methody@localhost methody]$ ls \*a\*

-filename-with-

[methody@localhost methody]$ ls -dF \*[ao]\*

Documents/ examples/ loop to.sort\*

Пример 8.7. Использование шаблона в командной строке

Мефодий, как это случается с новичками, немедленно натолкнулся на несколько "подводных камней", неизбежных в ситуации, когда конечный результат неизвестен. Только первая команда сработала **не** вопреки его ожиданиям: *шаблон* " .bash\* " был превращен командной оболочкой в список файлов, начинающихся на .bash, этот список получила в качестве параметров командной строки ls, после чего честно его вывела. С " /bin/e\* " Мефодию повезло - этот *шаблон* превратился в список файлов из каталога /bin, начинающихся на " e ", и **первым** файлом в списке оказалась безобидная утилита /bin/echo. Поскольку в командной строке ничего, кроме *шаблона*, не было, именно строка /bin/echo была воспринята оболочкой в качестве команды4, которой (в качестве **параметров** ) были переданы остальные элементы списка - /bin/ed, /bin/egrep и /bin/ex.

Что же касается ls \*a\*, то, по расчетам Мефодия, эта команда должна была выдать список файлов в текущем каталоге, имя которых **содержит** " a ". Вместо этого на экран вывелось имя файла из подкаталога examples... Впрочем, никакой черной магии тут нет. Во-первых, имена файлов вида " .bash\* " хотя и содержат " a ", но начинаются с точки, и, стало быть, считаются **скрытыми**. *Скрытые файлы* попадают в результат генерации имен только если точка в начале указана **явно** (как в первой команде примера). Поэтому по *шаблону* " \*a\* " в домашнем каталоге Мефодия bash нашел только подкаталог с именем examples - его-то он и передал в качестве параметра утилите ls. Что было выведено на экран в результате образовавшейся команды ls examples? Конечно, содержимое каталога. *Шаблон* в последней команде из примера, " \*[ao]\* " был превращен в список файлов, чьи имена содержат " a " или " o " - Documents examples loop to.sort, а ключ " -d " потребовал от ls показывать информацию о каталогах, а не об их содержимом. В соответствии с ключом " -F ", ls расставила " / " после каталогов и " \* " после исполняемых файлов.

Еще одно отличие генерации имен от стандартной обработки *шаблона* - в том, что символ " / ", разделяющий элементы пути, **никогда** не ставится в соответствие " \* " или диапазону. Происходит это не потому, что искажен алгоритм, а потому, что при генерации имен *шаблон* применяется именно к элементу пути, внутри которого уже нет " / ". Например, получить список файлов, которые находятся в каталогах /usr/bin и /usr/sbin и содержат подстроку " ppp " в имени, можно с помощью *шаблона* " /usr/\*bin/\*ppp\* ". Однако **одного шаблона**, который бы включал в этот список еще и каталоги /bin и /sbin - то есть подкаталоги **другого** уровня вложенности - по стандартным правилам сделать нельзя5.

Если перед любым специальным символом стоит " \ ", этот символ лишается специального значения, **экранируется**: пара " \символ " заменяется командным интерпретатором на " символ " и передается в командную строку без всякой дальнейшей обработки:

[methody@localhost methody]$ echo \*o\*

Documents loop to.sort

[methody@localhost methody]$ echo \\*o\\*

\*o\*

[methody@localhost methody]$ echo "\*o\*"

\*o\*

[methody@localhost methody]$ echo \*y\*

\*y\*

[methody@localhost methody]$ ls \*y\*

ls: \*y\*: No such file or directory

Пример 8.8. Экранирование специальных символов и обработка "пустых" шаблонов

Мефодий заметил, что *шаблон*, которому не соответствует **ни одно** имя файла, bash раскрывать не стал, как если бы все " \* " в нем были экранированы. В самом деле, какое из двух зол меньше: изменять **интерпретацию спецсимволов** в зависимости от содержимого каталога, или сохранять логику интерпретации с риском превратить команду с параметрами в команду **без параметров**? Если бы, допустим, *шаблон*, которому не нашлось соответствия, попросту удалялся, то команда ls \*y\* превратилась бы в ls и неожиданно выдала бы содержимое всего каталога. Авторы bash (как и Стивен Борн, автор самой первой командной оболочки - sh ) выбрали более непоследовательный, но и более безопасный первый способ6.

Лишить специальные символы их специального значения можно и другим способом. В лекции 2 было рассказано, что *разделители* (пробелы, символы табуляции и символы перевода строки) перестают восприниматься как таковые, если часть командной строки, их содержащую, окружить двойными или одинарными кавычками. В кавычках перестает "работать" и генерация имен (как это видно из примера), и интерпретация других специальных символов. Двойные кавычки, однако, допускают выполнение подстановок *переменной окружения* и результата работы команды, описанных в двух следующих разделах.

**Окружение**

В лекции 5 уже упоминалось, что *системный вызов* fork(), создавая точную копию родительского процесса, копирует также и *окружение*. Необходимость в " *окружении* " обусловлена вот какой задачей. Акт передачи данных от родительского процесса дочернему, и, что еще важнее, системе, должен обладать свойством **атомарности**. Если использовать для этой цели *файл* (например, *конфигурационный файл* запускаемой программы), всегда сохраняется *вероятность*, что за время между изменением файла и последующим чтением запущенной программой кто-то - например, другой процесс того же пользователя - снова изменит этот файл7. Хорошо бы, чтобы **изменение** данных и их передача выполнялись **одной** операцией. Например, завести для каждого процесса такой "*файл*", содержимое которого, во-первых, мог бы изменить **только** этот процесс, и, во-вторых, оно автоматически копировалось бы в аналогичный "*файл*" *дочернего процесса* при его порождении.

Эти свойства и реализованы в понятии " *окружение* ". Каждый запускаемый процесс система снабжает неким информационным пространством, которое этот процесс вправе изменять как ему заблагорассудится. Правила пользования этим пространством просты: в нем можно задавать именованные хранилища данных ( *переменные окружения* ), в которые записывать какую угодно информацию (присваивать *значение* *переменной окружения* ), а впоследствии эту информацию считывать (подставлять *значение* переменной). *дочерний процесс* - точная копия родительского, поэтому его *окружение* - также точная копия родительского. Если про *дочерний процесс* известно, что он использует значения некоторых переменных из числа передаваемых ему с *окружением*, родительский может заранее указать, каким из копируемых в *окружении* переменных нужно изменить *значение*. При этом, с одной стороны, никто (кроме системы, конечно) не сможет вмешаться в процесс передачи данных, а с другой стороны, одна и та же *утилита* может быть использована **одним** и **тем же** способом, но в измененном *окружении* - и выдавать различные результаты:

[methody@localhost methody]$ date

Птн Ноя 5 16:20:16 MSK 2004

[methody@localhost methody]$ LC\_TIME=C date

Fri Nov 5 16:20:23 MSK 2004

Пример 8.9. Утилита date пользуется переменной окружения LC\_TIME

***Окружение*** (environment) - набор данных, приписанных системой процессу. Процесс может пользоваться информацией из *окружения* для настройки, изменять и дополнять его. *Окружение* представлено в виде *переменных окружения* и их значений. При порождении процесса *окружение* родительского процесса **наследуется** дочерним (копируется).

**Работа с переменными в shell**

В последнем примере Мефодий воспользовался подсмотренным у Гуревича приемом: присвоил некоторое значение *переменной окружения* в командной строке **перед** именем команды. Командный интерпретатор, увидев " = " внутри первого слова командной строки, приходит к выводу, что это - операция присваивания, а не имя команды, и запоминает, как надо изменить *окружение* команды, которая последует далее. *Переменная окружения* LC\_TIME предписывает использовать определенный язык при выводе даты и времени, а значение " C " соответствует стандартному системному языку (чаще всего - английскому).

Если рассматривать shell в качестве высокоуровневого языка программирования, то его переменные - самые обычные **строковые** переменные. Записать значение в переменную можно с помощью операции присваивания, а прочесть его оттуда - с помощью операции *подстановки* вида $переменная:

[methody@localhost methody]$ A=dit

[methody@localhost methody]$ C=dah

[methody@localhost methody]$ echo $A $B $C

dit dah

[methody@localhost methody]$ B=" "

[methody@localhost methody]$ echo $A $B $C

dit dah

[methody@localhost methody]$ echo "$A $B $C"

dit dah

[methody@localhost methody]$ echo '$A $B $C'

$A $B $C

Пример 8.10. Подстановка значений переменных

Как видно из примера, значение **неопределенной** переменной ( B ) в shell считается пустым и при *подстановке* не выводится никаких предупреждений. Сама *подстановка* происходит, как и генерация имен, **перед** разбором командной строки, набранной пользователем. Поэтому вторая команда echo в примере получила, как и первая, **два** параметра (" *dit* " и " dah "), несмотря на то, что переменная B была к тому времени определена и содержала *разделитель* -пробел. А вот третья и четвертая команды echo получили по **одному** параметру. Здесь сказалось различие между одинарными и двойными кавычками в shell: внутри двойных кавычек действует *подстановка* значений переменных.

Переменные, которые командный интерпретатор bash определяет **после** запуска, не принадлежат *окружению*, и, стало быть, не наследуются *дочерними процессами*. Чтобы переменная bash попала в *окружение*, ее надо **проэкспортировать** командой export:

[methody@localhost methody]$ echo "$Qwe -- $LANG"

-- ru\_RU.KOI8-R

[methody@localhost methody]$ Qwe="Rty" LANG=C

[methody@localhost methody]$ echo "$Qwe -- $LANG"

Rty -- C

[methody@localhost methody]$ sh

sh-2.05b$ echo "$Qwe -- $LANG"

-- C

sh-2.05b$ exit

[methody@localhost methody]$ echo "$Qwe -- $LANG"

Rty -- C

[methody@localhost methody]$ export Qwe

[methody@localhost methody]$ sh

sh-2.05b$ echo "$Qwe -- $LANG"

Rty -- C

sh-2.05b$ exit

Пример 8.11. Экспорт переменных shell в окружение

Здесь Мефодий завел **новую** переменную Qwe и изменил значение *переменной окружения* LANG, доставшейся стартовому bash от программы login. В результате запущенный *дочерний процесс* sh получил **измененное** значение LANG и никакой переменной Qwe в *окружении*. После export Qwe эта переменная была добавлена в *окружение* и, соответственно, передалась sh.

**Переменные окружения, используемые системой и командным интерпретатором**

Во время сеанса работы пользователя *стартовый командный интерпретатор* получает от login довольно богатое *окружение*, к которому добавляет и собственные настройки. Просмотреть *окружение* в bash можно с помощью команды set. Большинство заранее определенных переменных используются либо самой командной оболочкой, либо утилитами системы, поэтому их изменение приводит к тому, что оболочка или утилиты начинают работать несколько иначе.

Весьма примечательна *переменная окружения* PATH. В ней содержится список каталогов, элементы которого разделяются двоеточиями. Если команда в командной строке - не собственная команда shell (вроде cd ) и не представлена в виде пути к запускаемому файлу (как /bin/ls или ./script ), то shell будет искать эту команду среди имен запускаемых файлов во всех каталогах PATH, и только в них. Точно так же будут поступать и другие утилиты, использующие *библиотечную функцию* *execlp*() или *execvp*() (запуск программы).

По этой причине исполняемые файлы невозможно запускать просто по имени, если они лежат в текущем каталоге, и текущий каталог не входит в PATH. Мефодий в таких случаях пользовался кратчайшим из возможных путей, " ./ " (например, вызывая *сценарий* ./script ):

[methody@localhost methody]$ echo $PATH

/home/methody/bin:/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/usr/games

[methody@localhost methody]$ mkdir /home/methody/bin

[methody@localhost methody]$ mv to.sort loop script bin/

[methody@localhost methody]$ script

Hello, Methody!

Пример 8.12. Использование PATH

Пути, указанные в PATH, могут и не существовать на самом деле. Обнаружив в $PATH элемент /home/methody/bin (подкаталог bin домашнего каталога), Мефодий решил, что гораздо правильнее будет складывать исполняемые файлы не куда попало, а именно в этот каталог: во-первых, это упорядочит структуру домашнего каталога, а во-вторых, позволит запускать эти файлы по имени. Но для начала пришлось такой каталог **создать**. Порядок, при котором собственные программы лежат в специальном каталоге, куда **безопаснее** беспорядка, при котором поиск запускаемого файла по имени начинается с **текущего** каталога. Если в текущем каталоге окажется программа ls и этот каталог - не /bin, а, допустим, /home/shogun/dontrunme, тогда следует ожидать подвоха...

*Переменных окружения*, влияющих на работу **разных** утилит, довольно много. Например, переменные семейства LC\_ (полный их список выдается командой locale ), определяющие язык, на котором выводятся диагностические сообщения, стандарты на формат даты, денежных единиц, чисел, способы преобразования строк и т. п. Очень важна переменная TERM, определяющая **тип** терминала: как известно из лекции 2, разные терминалы имеют различные управляющие последовательности, поэтому программы, желающие эти последовательности использовать, обязательно сверяются с переменной TERM 8. Если какая-то утилита требует редактирования файла, этот файл передается программе, путь к которой хранится в переменной EDITOR (обычно это /usr/bin/vi, о котором речь пойдет в лекции 9). Наконец, некоторые переменные вроде UID, USER или PWD просто содержат полезную информацию, которую можно было бы добыть и другими способами.

Некоторые *переменные окружения* предназначены специально для bash: они задают его свойства и особенности поведения. Таковы переменные семейства PS (Prompt String). В этих переменных хранится строка-подсказка, которую командный интерпретатор выводит в разных состояниях. В частности, содержимое PS1 - это подсказка, которую shell показывает, когда вводит командную строку, а PS2 - когда пользователь нажимает Enter, а интерпретатор по какой-то причине считает, что ввод командной строки не завершен (например, не закрыты кавычки). С $PS2 (символом " > ") Мефодий уже сталкивался в лекциях 2 и 7:

[methody@localhost methody]$ cd examples/

[methody@localhost examples]$ echo $PS1

[\u@\h \W]\$

[methody@localhost examples]$ PS1="--> "

-->

-->

PS1="\t \w "

22:11:47 ~

22:11:48 ~

22:11:48 ~ PS1="\u@\h:\w \$ "

methody@localhost:~/examples $

methody@localhost:~/examples $

methody@localhost:~/examples $ cd

methody@localhost:~ $

methody@localhost:~ $

Пример 8.13. Использование переменной окружения PS1

Мефодий экспериментировал с PS1, параллельно читая документацию по bash (" (bash.info)Printing a Prompt "). Оказывается, в этом командном интерпретаторе содержимое PS1 не просто подставляется при выводе - оно еще и преобразуется, позволяя выводить всякую полезную информацию: имя пользователя (соответствует подстроке " \u ", user ), имя компьютера (" \h ", host ), время (" \t ", time ), путь к текущему каталогу (" \w ", *work directory*) и т. п. В примере Мефодий заменил " \W " (показывающую последний элемент пути, то есть собственное имя текущего каталога) на " \w ", **полный** путь, потому что " \w " обладает свойством выделять в полном пути домашний каталог и заменять его на " ~ ". Такое преобразование значений переменных семейства PS1 выполняется, только когда их использует bash в качестве подсказки, а при обычной *подстановке* этого не происходит.

**Язык программирования sh**

Некогда Мефодий выучил несколько языков программирования, и уже собрался было написать кое-какие нужные программы на *Си* или *Python*, однако Гуревич его остановил. Большая часть того, что нужно начинающему пользователю Linux, делается с помощью одной правильной команды, или вызовом нескольких команд в конвейере. От пользователя только требуется оформить решение задачи в виде *сценария* на *shell*. На самом же деле уже самый первый из командных интерпретаторов, sh, был настоящим высокоуровневым языком программирования - если, конечно, считать все утилиты системы его операторами. При таком подходе от sh требуется совсем немного: возможность вызывать утилиты, возможность свободно манипулировать результатом их работы и несколько алгоритмических конструкций (условия и циклы).

К сожалению, **программирование** на *shell*, а также других, более мощных интерпретируемых языках в Linux, остается за рамками нашего курса. Так что, пока Мефодий читает документацию по bash и упражняется в написании *сценариев*, нам остается только поверхностно рассмотреть свойства *shell* как языка программирования и **интегратора** команд. Заметим попутно, что писать *сценарии* для bash - непрактично, так как исполняться они смогут лишь при помощи bash. Если же ограничить себя рамками sh, совместимость с которым объявлена и в bash, и в zsh, и в ash (наиболее близком по возможностям к sh ), и в других командных интерпретаторах, выполняться эти *сценарии* смогут любым из sh- подобных интерпретаторов, и не только в Linux.

**Интеграция процессов**

Каждый процесс Linux при завершении передает родительскому *код возврата* (exit status), который равен нулю, если процесс считает, что его работа была успешной, или **номеру ошибки** - в противном случае. Командный интерпретатор хранит *код возврата* последней команды в специальной переменной " ?". Что более важно, *код возврата* используется в *условных операторах*: если он равен нулю, условие считается выполненным, а если нет - невыполненным:

[methody@localhost methody]$ grep Methody bin/script

echo 'Hello, Methody!'

[methody@localhost methody]$ grep -q Methody bin/script ; echo $?

0

[methody@localhost methody]$ grep -q Shogun bin/script ; echo $?

1

[methody@localhost methody]$ if grep -q Shogun bin/script ; then echo "Yes"; fi

[methody@localhost methody]$ if grep -q Methody bin/script ; then echo "Yes"; fi

Yes

Пример 8.14. Оператор if использует код возврата программы grep

*Условный оператор* if запускает команду- **условие**, grep -q, которая ничего не выводит на экран, зато возвращает 0, если *шаблон* найден, и 1, если нет. В зависимости от *кода возврата* этой команды, if выполняет или не выполняет **тело**: список команд, заключенный между then и fi. Точка с запятой разделяет операторы в sh ; либо она, либо перевод строки необходимы перед then и fi, иначе все, что идет после grep, интерпретатор посчитает параметрами этой утилиты.

Множеством функций обладает команда test: она умеет сравнивать числа и строки, проверять ярлык объекта файловой системы и наличие самого этого объекта. У " test " есть второе имя: " [ " (как правило, /usr/bin/[ - символьная или даже *жесткая ссылка* на /usr/bin/test), позволяющее оформлять оператор if более привычным образом:

[methody@localhost methody]$ if test -f examples ; then ls -ld examples ; fi

[methody@localhost methody]$ if [ -d examples ] ; then ls -ld examples ; fi

drwxr-xr-x 2 methody methody 4096 Окт 31 15:26 examples

[methody@localhost methody]$ A=8; B=6; if [ $A -lt $B ] ; then echo "$A<$B" ; fi

[methody@localhost methody]$ A=5; B=6; if [ $A -lt $B ] ; then echo "$A<$B" ; fi

5<6

Пример 8.15. Оператор test

Команда test -f проверяет, не является ли ее аргумент файлом; поскольку examples - это каталог, результат будет неудачным. Команда [ -d - то же самое, что и test -d (не каталог ли первый параметр), только **последним** параметром этой команды - исключительно для красоты - должен быть символ " ] ". Результат - успешный, и выполняется команда ls -ld. Команда test параметр1 -lt параметр3 проверяет, является ли параметр1 числом, меньшим, чем (less than) параметр3. В более сложных случаях оператор if удобнее записывать "лесенкой", выделяя переводами строки окончание условия и команды внутри тела (их может быть много).

Второй тип *подстановки*, которую shell делает внутри двойных кавычек - это *подстановки вывода* команды. ***Подстановка вывода*** имеет вид " `команда` " (другой вариант - " $(команда) "). Как и *подстановка* значения переменной, она происходит перед тем, как начнется разбор командной строки: выполнив команду и получив от нее какой-то текст, shell примется разбирать его, как если бы этот текст пользователь набрал вручную. Это очень удобное средство, если то, что выводит команда, необходимо передать самому интерпретатору:

[methody@localhost methody]$ A=8; B=6

[methody@localhost methody]$ expr $A + $B

14

[methody@localhost methody]$ echo "$A + $B = `expr $A + $B`"

8 + 6 = 14

[methody@localhost methody]$ A=3.1415; B=2.718

[methody@localhost methody]$ echo "$A + $B = `expr $A + $B`"

expr: нечисловой аргумент

3.1415 + 2.718 =

[methody@localhost methody]$ echo "$A + $B" | bc

5.8595

[methody@localhost methody]$ C=`echo "$A + $B" | bc`

[methody@localhost methody]$ echo "$A + $B = $C"

3.1415 + 2.718 = 5.8595

Пример 8.16. Подстановка вывода команды

Сначала для арифметических вычислений Мефодий хотел воспользоваться командой expr, которая работает с параметрами командной строки. С целыми числами expr работает неплохо, и ее результат можно подставить прямо в аргумент команды echo. С действительными числами умеет работать утилита- *фильтр* bc ; арифметическое выражение пришлось сформировать с помощью echo и передать по конвейеру, а результат поместить в переменную C. Во многих современных командных оболочках есть **встроенная** целочисленная арифметика вида " $(( выражение )) ".

**Сценарии**

В языке sh много внимания было уделено удобству написания *сценариев*. В частности, параметры командной строки, переданные *сценарию*, доступны в нем в виде переменных, имена которых совпадают с порядковым номером параметра:

[methody@localhost methody]$ cat > bin/two

#!/bin/sh

echo "Running $0: $\*"

$1 $3

$2 $3

[methody@localhost methody]$ chmod +x bin/two

[methody@localhost methody]$ bin/two file "ls -ld" examples

Running bin/two: file ls -ld examples

examples: directory

drwxr-xr-x 2 methody methody 4096 Окт 31 15:26 examples

[methody@localhost methody]$ two "ls -s" wc "bin/two bin/loop" junk

Running /home/methody/bin/two: ls -s wc bin/two bin/loop junk

4 bin/loop 4 bin/two

4 9 44 bin/two

1 5 26 bin/loop

5 14 70 итого

Пример 8.17. Использование позиционных параметров в сценарии

Как видно из примера, форма " $номер\_параметра " позволяет обратиться и к **нулевому** параметру - команде, а вся строка параметров хранится в переменной " \* ". Кроме того, свойство *подстановки* выполняться до разбора командной строки позволило Мефодию передать в качестве **одного** параметра " ls -ld " или " bin/two bin/loop ", а интерпретатору - разбить эти параметры на имя команды и ключи и два имени файла соответственно.

В sh есть и *оператор* while, формат которого аналогичен if, и более удобный именно в *сценариях* оператор обхода списка for (список делится на слова так же, как и командная строка - с помощью *разделителей* ):

[methody@localhost methody]$ for Var in Wuff-Wuff

Miaou-Miaou; do echo $Var; done

Wuff-Wuff

Miaou-Miaou

[methody@localhost methody]$ for F in $(date); do echo -n "<$F>"; done; echo

<Сбт><Ноя><6><12:08:38><2004>

[methody@localhost methody]$ cat > /tmp/setvar

QWERTY="$1"

[methody@localhost methody]$ sh /tmp/setvar 1 2 3; echo $QWERTY

[methody@localhost methody]$ . /tmp/setvar 1 2 3; echo $QWERTY

1

Пример 8.18. Использование for и операции "."

Во втором for Мефодий воспользовался *подстановкой вывода* команды date, каждое слово которой вывел с помощью echo -n в одну строку, а в конце команды пришлось вывести один перевод строки вручную.

Вторая половина примера иллюстрирует ситуацию, с которой Мефодий столкнулся во время своих экспериментов: все переменные, определяемые в *сценарии*, после окончания его работы куда-то пропадают. Оно и понятно: для обработки *сценария* всякий раз запускается **новый** интерпретатор ( *дочерний процесс*!), и **все** его переменные принадлежат именно ему и с завершением процесса уничтожаются. Таким образом достигается отсутствие **побочных эффектов**: запуская программу, пользователь может быть уверен, что та не изменит *окружения* командной оболочки. Однако в **некоторых** случаях требуется обратное: запустить *сценарий*, который нужным образом настроит *окружение*. Единственный выход - отдавать такой *сценарий* на обработку **текущему**, а не новому, интерпретатору (т. е. тому, что разбирает команды пользователя). Это делается с помощью специальной команды " .". Если вдруг в передаваемом *сценарии* обнаружится команда exit, exec или какая-нибудь другая, приводящая к завершению работы интерпретатора, завершится именно **текущая** командная оболочка, чем сеанс работы пользователя в системе может и закончиться.

**Настройка командного интерпретатора**

Научившись (главным образом в результате чтения документации и непрестанных экспериментов) создавать работающие *сценарии*, Мефодий решил приступить к настройке командной оболочки, поскольку, как он слышал, для этого используются именно *сценарии*.

**Привязка к клавишам**

Оказалось, что настройка **управляющих клавиш** в bash не выглядит как *сценарий*, и даже имеет отношение не только к bash, а ко **всем** программам, использующим *библиотеку* терминального ввода readline. *Конфигурационный файл* readline называется .inputrc и состоит, в основном, из команд вида " управляющая\_последовательность ": функция, где управляющая\_последовательность - это символы, при получении которых readline выполнит функцию работы с вводимой строкой. Список всех функций readline можно узнать у bash по команде bind -l, а список всех **привязок** этих функций к клавиатурным последовательностям - по команде bind -p. Мефодий вписал в .inputrc такие две строки:

"\e[5~": backward-word

"\e[6~": forward-word

Пример 8.19. Настройка .inputrc

Упомянутые в примере функции позволяют перемещать курсор в командной строке по **словам**, а esc-последовательности возвращаются, соответственно, клавишами Page Up и Page Down *виртуальной консоли* Linux (сочетание " \e " означает в .inputrc клавишу esc, то есть " ^[ ", символ с ASCII-кодом 27 ).

К одной и той же функции readline можно привязать **сколько угодно** управляющих последовательностей: например, клавиша **home** делает то же, что и " ^A ", Стрелка вверх - то же, что и " ^P ", а Del - то же, что и " ^D " (только не в **пустой** строке!). Таким образом отчасти решается проблема **несовместимости** управляющих последовательностей терминалов: если в каком-нибудь терминале **другого** типа Page Up или Page Down будут возвращать другие последовательности, Мефодий просто добавит в .inputrc еще одну пару команд. Правда, Гуревич советовал ему вовсе отказаться от редактирования .inputrc, а воспользоваться утилитой tput, которая обращается к переменной TERM и базе данных по терминалам terminfo и готова выдать верную для любого данного терминала информацию по kpp (key previous page) и knp (key next page). Выдачу tput можно "скормить" той же bind, и получить команду, которая работает на любом терминале: bind "\"`tput kpp`\": backward-word" (кавычки, экранированные обратной косой чертой, \", передадутся bind в неизменном виде)

**Стартовые сценарии**

Настройка оболочки - это в первую очередь настройка *окружения*. В начале сеанса работы (при запуске *стартового командного интерпретатора* ) с помощью команды " ." выполняется *сценарий* из файла со специальным именем - /etc/profile. Это - так называемый *общесистемный профиль*, стартовый *сценарий*, выполняющийся при входе в систему **любого**, кто использует командную оболочку, подобную sh. Следом выполняется ***персональный профиль*** (или просто **профиль** ) пользователя - *сценарий*, находящийся в домашнем каталоге, и называющийся .profile. Этот *сценарий* пользователь может видоизменять, как ему заблагорассудится.

Что касается bash, то структура его стартовых файлов сложнее. Прежде всего, ~/.profile выполняется, только если в домашнем каталоге нет файла .bash\_profile или .bash\_login, иначе стартовый *сценарий* берется оттуда. В эти файлы можно помещать команды, несовместимые с другими версиями shell, например, управление **сокращениями** или привязку функций к клавишам. Кроме того, каждый **интерактивный** (взаимодействующий с пользователем), но не стартовый bash выполняет системный и персональный конфигурационные *сценарии* /etc/bashrc и ~/.bashrc. Чтобы стартовый bash также выполнял ~/.bashrc, соответствующую команду необходимо вписать в ~/.bash\_profile. Далее, каждый **неинтерактивный** (запущенный для выполнения *сценария* ) bash сверяется с *переменной окружения* BASH\_ENV и, если в этой переменной записано имя существующего файла, выполняет команды оттуда. Наконец, при завершении стартового bash выполняются команды из файла ~/.bash\_logout.

**Пример настроек**

Ниже приведены примеры *конфигурационных файлов*, которые Мефодий, сам или с помощью Гуревича, разместил в домашнем каталоге.

PS1="\u@\h:\w \$ "

EDITOR="/usr/bin/vim"

export PS1 EDITOR

# Get the aliases and functions

if [ -f ~/.bashrc ]; then

. ~/.bashrc

fi

Пример 8.20. Пример файла .bash\_profile

В этом файле вызывается ~/.bashrc (если он существует).

# User specific aliases and functions

if [ -r ~/.alias ]; then

. ~/.alias

fi

# Source global definitions

if [ -r /etc/bashrc ]; then

. /etc/bashrc

fi

Пример 8.21. Пример файла .bashrc

Мефодий решил, что сокращения удобнее будет хранить в отдельном файле - ~/.alias. Кроме того, вызывается *сценарий* bashrc, который Мефодий обнаружил в каталоге /etc. Этот файл не входит в число автоматически выполняемых bash, поэтому его выполнение надо задавать явно:

alias > ~/.alias

Пример 8.22. Пример файла .bash\_logout

Заметив, что команда alias выдает список сокращений в том же формате, в котором они и задаются, Мефодий придумал, как обойтись без редактирования файла ~/.alias. Отныне все сокращения, определенные к моменту завершения сеанса работы, будут записываться обратно в .alias. Туда попадут и те сокращения, что были считаны во время выполнения .bashrc, и те, что впоследствии были определены вручную:

alias l='ls -FAC'

alias ls='ls --color=auto'

alias pd='pushd'

alias pp='popd'

alias v='ls -ali'

alias vi='/usr/bin/vim'

Пример 8.23. Пример файла .alias

Последняя запись в файле .alias относится к инструменту, с помощью которого Мефодий создавал все эти файлы: текстовому редактору *vim*. .

**Лекция 9:**

**Текстовые редакторы**

**Задача текстовых редакторов**

После основных утилит для работы с файлами и текстом первая *программа*, которая понадобится любому пользователю Linux – это *текстовый редактор* (для краткости – просто редактор). Из предыдущих лекций и собственных экспериментов Мефодию уже стало понятно, какое значительное *место* занимают в системе Linux данные в *текстовом формате*, т. е. состоящие из символов, которые могут быть отображены на экране терминала и которые может прочесть человек. Однако пока Мефодий мог работать с текстом только последовательно, строка за строкой; даже имея дело с файлом, он не мог вернуться и отредактировать уже переданные системе строки. Именно для того, чтобы работать с текстовым файлом как со страницей, *по* которой можно перемещаться и редактировать текст в любой *точке*, и нужны *текстовые редакторы*. 1

*Текстовый редактор* потребуется пользователю Linux в первую *очередь* для того, чтобы изменить настройки системы или своего окружения, например, *shell* – при этом нужно будет редактировать *конфигурационные файлы*, которые всегда представлены в *текстовом формате* (см. лекции 8 и 12). Но и собственные задачи пользователя могут потребовать редактирования текстовых файлов: например, сценарии и программы, электронные письма, а также заметки для себя, которые пишет Мефодий —все это данные в *текстовом формате*. Текстовые данные, полученные при помощи стандартных утилит, тоже бывает удобно сохранять в файлах и редактировать.

Не стоит путать *текстовые редакторы* и *текстовые процессоры*. *Текстовые процессоры*, например OpenOffice Writer или Microsoft *Word*, предназначены для создания **документов**, в которых, помимо собственно текста, содержится и различная **метаинформация** (*информация* об оформлении): *размещение* текста на странице, *шрифт* и т. п. Поскольку в *текстовом формате* не предусмотрено средств для сохранения информации об оформлении (там есть только символы и строки), *текстовые процессоры* используют собственные форматы для хранения данных. Текст, в котором нет никакой метаинформации об оформлении, называют "plain *text*" (только текст, "плоский", простой текст).

Однако при помощи *текстовых редакторов* можно работать не только с форматом plain *text*. Различная метаинформация (об оформлении, способе использования текста, например, в качестве ссылки и пр.) может быть записана и в виде обычных символов (т. е. в *текстовом формате* ), но со специальным соглашением, что эти символы нужно интерпретировать особым образом: как инструкции *по* обработке текста, а не как текст. Такие инструкции называются *разметкой*. Таким образом устроен, например, формат *HTML*. Для того чтобы обработать *разметку* *HTML* и в соответствии с ней **отобразить** текст, нужна специальная *программа* – *браузер*, но **редактировать** файлы *HTML* и прочие форматы *разметки* можно и при помощи *текстового редактора*. Кроме того, программы на любых языках программирования и сценарии (программы на *shell*) тоже представляют собой *текстовые файлы*. Многие *текстовые редакторы* ориентированы на работу не только с "плоским" текстом, но и с текстом в различных форматах. Для этого придумана масса усовершенствований, уменьшающих количество символов, которые нужно вводить вручную: специальные команды, клавиатурные сокращения и автодополнение ключевых слов и конструкций.

Важнейшее условие для *текстового редактора* в Linux – возможность работать в терминале, так как это основной способ управления системой. Поэтому и ввод данных, и редактирование должны полностью осуществляться средствами терминала, т. е. алфавитно-цифровыми и некоторыми функциональными клавишами. Поскольку функциональных клавиш, на которые можно рассчитывать на любом терминале, совсем немного, а команд, которые нужно отдавать редактору, – очень много, требуется способ вводить любые команды ограниченными средствами терминала. Это условие, равно как и требование удобства при работе с разнообразными структурированными текстами, выполнено в двух "главных" *текстовых редакторах* Linux – *Vi* и Emacs, о которых в основном и будет идти речь в этой лекции.

**Vi и лучше, чем Vi**

В любой системе Linux, даже при самой минимальной конфигурации, всегда присутствует *текстовый редактор*, поскольку в любой – даже самой катастрофической – ситуации у пользователя должна быть возможность отредактировать *конфигурационные файлы*, чтобы привести систему в рабочее состояние. *По* сложившейся традиции *текстовым редактором*, который обязательно запустится из любой командной строки Linux, является Vi2. Однако верно и обратное: если вы работаете в незнакомой системе Linux или произошел сбой, в результате которого доступна только очень небольшая часть системы, нельзя быть уверенным, что найдется хоть какой-нибудь другой *текстовый редактор*, кроме *Vi*. Поэтому каждому пользователю Linux нужны хотя бы основные навыки работы в *Vi*. При первом знакомстве c *Vi* работа обычно не ладится: очень уж он непривычен, его нельзя с удобством использовать, запомнив только две-три простейшие команды редактирования. Cтоит понять основные принципы работы в *Vi* и потратить некоторое время на его освоение, тогда в нем откроется мощный инструмент, позволяющий очень эффективно работать с текстом.

Под именем *Vi*, на самом деле, может скрываться несколько разных программ: с момента появления *Vim* в операционной системе *UNIX* (а это произошло около 30 лет назад) этот редактор стал чем-то вроде стандарта. К настоящему времени существует ряд программ, либо в точности повторяющих вид и поведение "классического" *Vi* (например, nvi ), либо очень похожих на него, но со значительно расширенными возможностями (*Vim*, elvis ). Наибольшей популярностью пользуется *Vim*, возможности которого огромны – для их описания потребовалось почти сто тысяч строк документации. Когда *пользователь* Linux набирает в командной строке *Vim*, скорее всего, будет запущена "облегченная" версия *Vi*, которая настроена таким образом, чтобы максимально воспроизводить поведение классического редактора *Vim*. Естественно, в таком режиме часть возможностей *Vi* недоступна. Все свойства, которыми *Vi* отличается от *Vim*, обязательно снабжены в руководстве *по* *Vi* указанием "*not* in *vi*". В дальнейшем изложении под *Vi* мы будем понимать именно *Vim* в режиме совместимости, все возможности, недоступные в этом режиме, будут оговариваться. Чтобы вызвать *Vim* в полнофункциональном режиме, достаточно набрать команду *Vim*.

Популярность *Vi* не случайна: этот *текстовый редактор* позволяет не только производить простые *операции* редактирования текстовых файлов – он хорошо приспособлен и для максимально быстрого и эффективного решения ряда смежных с редактированием задач. Среди самых важных его возможностей – инструменты для работы с текстами на различных языках программирования и в различных форматах *разметки*. *Vim* умеет подсвечивать разными цветами синтаксические конструкции языка программирования или *разметки* 3, автоматически выставлять отступы, что облегчает восприятие структуры документа. Кроме того, в *Vim* есть специальные средства для организации *цикла* отладки программы: компиляция–правка исходного текста–компиляция... Подробнее об этих и прочих возможностях *Vim* можно узнать из руководств *по* *Vim*. Объем данной лекции позволяет описать только самое начало работы с *Vi*.

**Запуск Vi**

Чтобы начать сеанс редактирования в Vi, достаточно выполнить команду Vi на любом терминале. Чтобы открыть для редактирования уже существующий файл, *путь* к этому файлу нужно указать в качестве параметра: " vi путь\_к\_файлу ". Как и всякая уважающая себя программа UNIX, *Vim* может быть запущен с множеством модифицирующих его поведение ключей, которые подробно описаны в руководстве. Вызванный без параметра, редактор откроет пустой *буфер* – чистый лист для создания нового текста. В центре экрана при этом может появиться краткое приветственное сообщение, где указаны версия программы и команды для получения помощи и выхода из редактора (что нетривиально). Однако такое сообщение может и не появиться – это зависит от версии Vi, установленной в системе.

Для отображения текста и работы с ним Vi использует весь экран терминала – только последняя строка предназначена для диалога с пользователем: вывода информационных сообщений и ввода команд. Пока *буфер* не заполнен текстом, в начале каждой строки экрана отображается символ " ~ ", обозначающий, что в этом месте *буфера* нет ничего, даже пустой строки. Общий вид экрана в начале работы будет примерно такой:

#

~

~

~

~

~

Пример 9.1. Начало работы с Vi

Символ " # " обозначает курсор. На экране терминала умещается больше строк, но в примерах мы будем для компактности изображать только необходимый минимум.

**Режимы**

В Vi проблема разделения команд редактору и вводимого текста решена при помощи режимов: в **командном** режиме нажатие на любую клавишу – это команда редактору, в режиме **вставки** нажатие на клавишу приводит к вставке соответствующего символа в редактируемый текст. Поэтому при работе с Vi пользователю всегда нужно обращать внимание на то, в каком режиме находится редактор.

**Режимы Vi**. Состояния редактора Vi, в которых он по-разному обрабатывает нажатия клавиш. Различают три режима Vi: **командный** (нажатие любой клавиши считается командой и немедленно исполняется), **вставки** (нажатие клавиши-печатного символа приводит к вставке этого символа в текст) и **командной строки** (для ввода длинных команд, отображаемых на экране; ввод завершается **Enter** ).

Редактор Vi всегда начинает работу в **командном** режиме. В этом режиме есть два способа отдавать команды редактору. Во-первых, нажатие практически на любую клавишу редактор воспринимает как команду. В *Vim*, даже в режиме Vi-совместимости, командное значение определено для всех латинских букв (в верхнем и нижнем *регистрах* ), цифр, знаков препинания и большинства других печатных символов. При нажатии на эти клавиши вводимые команды нигде не отображаются – они просто **исполняются**. 4

Во-вторых, у Vi есть своя командная строка: чтобы ее вызвать, нужно ввести в командном режиме " :". В результате в начале последней строки экрана появится двоеточие – это *приглашение командной строки*. Здесь вводятся более сложные команды Vi, которые включают в себя целые слова (например, имена файлов), причем текст набираемой команды, конечно, отображается. Команды передаются Vi клавишей Enter. В современных версиях *Vim* с командной строкой можно работать так же, как и в shell: редактировать ее, достраивать команды клавишей Tab, пользоваться *историей команд*.

Главная команда командной строки *Vim* – вызов подсистемы помощи " :help Enter ". Двоеточие переводит *Vim* в *режим командной строки*, " help " – собственно команда, **Enter** передает команду. help можно вызывать с аргументом: названием команды или настройки *Vim*. *Vim* очень хорошо документирован, поэтому по команде " :help объект " можно получить информацию о **любом** свойстве *Vim*, например, команда " :help i " выведет сведения о значении клавиши " i " в командном режиме Vi.

Команда " :set имя\_настройки " позволяет настраивать *Vim* прямо в процессе работы с ним. Например, отдав команду " :set wrap " пользователь тем самым включает настройку " wrap ", что заставляет редактор переносить слишком длинные строки, не умещающиеся в ширину терминала. Выключить эту настройку можно командой " :set nowrap ", так что концы длинных строк исчезнут за правым краем экрана.

Наконец, чтобы ввести текст, нужно перейти из командного режима в *режим вставки*, нажав клавишу " i " (от " **i** nsert" – "вставка"). В этот момент в последней строке появится сообщение о том, что редактор находится в режиме вставки: " --INSERT-- " или " --ВСТАВКА-- " 5, в зависимости от установленного языка системных сообщений.

В режиме вставки можно вводить текст, завершая строку нажатием Enter. Однако нужно помнить, что в некоторых (ортодоксальных) версиях Vi в режиме ввода не работают **никакие** *команды перемещения* по тексту – здесь можно только набирать. Если вы заметили, что ошиблись в наборе – не нужно сразу перемещать курсор и пытаться исправить ошибку: гораздо удобнее будет внести все исправления потом, в командном режиме, где доступно множество специальных команд быстрого перемещения и замены текста. Чтобы перейти из режима вставки обратно в *командный режим*, нужно нажать ESC. 6

Мефодий начал упражняться с *Vim* на файле примеров:

methody@oblomov:~ $ vi textfile

Это файл для примеров.

Пример 1.

~

~

~

~

~

"textfile" 2L, 33C 1,1 Весь

Пример 9.2. Vim в командном режиме

*Vim* начал работу, как и положено – в командном режиме. В последней строке содержатся некоторые полезные сведения об открытом файле: его имя, общее количество строк (" 2L "), символов (" 33С "), позиция курсора (" 1,1 " – номер\_строки,номер\_символа). " Весь " обозначает, что все содержимое файла уместилось на экране терминала. Теперь Мефодий нажмет " i " и введет немного текста:

Это файл для примеров.

Пример 1.

Моя первая строка в vi!

~

~

~

~

-- ВСТАВКА -- 3,24 Весь

Пример 9.3. Vim в режиме вставки

Теперь *Vim* работает в режиме вставки: в последней строке появилось информационное сообщение об этом. Набрав текст, Мефодий может вернуться в *командный режим*, нажав ESC (подсказка " --ВСТАВКА-- " при этом исчезнет из последней строки).

На самом деле, из командного режима можно перейти в режим ввода несколькими командами. Разница между ними заключается в том, в какой *точке* начнется ввод символов. Например, по командам " O " и " o " (" **o** pen") можно вводить текст с новой строки (до или после текущей), по команде " I " – с начала строки, команды " a " и " A " (" **a** ppend") ведают **добавлением** символов (после курсора или в конец строки) и т. п.

**Работа с файлами**

Редактируя текст в Vi, пользователь работает не непосредственно с файлом, а с *буфером*. Если открывается уже существующий файл, Vi копирует его содержимое в *буфер* и отображает *буфер* на экране. Все изменения, которые производит пользователь, происходят именно в содержимом *буфера* – открытый файл пока остается неизменным. Если же Vi вызван без параметра, то создается пустой *буфер*, который пока не связан ни с каким файлом 7.

Чтобы записать сделанные изменения в файл, используется команда " :wEnter " (чтобы ее отдать, нужно сначала перейти в *командный режим*). О том, что " w " – это сокращение от англ. " **w** rite", "записать", можно узнать, нажав Tab после " :w " – и *Vim* дополнит эту команду до " write ". Подобным образом можно поступить с большинством команд в командной строке *Vim* – в этом редакторе очень последовательно соблюдается принцип *аббревиативности*. Мефодий выполнил " write " и получил такое информационное сообщение:

Это файл для примеров.

Пример 1.

Моя первая строка в vi!

~

~

~

~

"textfile" 3L, 57C записано 3,24 Весь

Пример 9.4. Запись файла

Мефодий не указал, куда именно записать содержимое *буфера*, и по умолчанию оно было записано в тот файл, который он и открывал для редактирования: " textfile ". Однако команде " write " можно указать любое имя файла в качестве параметра – и тогда содержимое *буфера* будет записано в этот файл, а если такого файла нет, то он будет создан. Параметр " write " обязательно потребуется, если текст в *буфере* еще не записан ни в каком файле.

Наиболее важна для новичка команда **выхода** из Vi – " :qEnter " (сокращение от " **q** uit"). Пользователь, запустивший редактор в первый раз, нередко сталкивается с тем, что никак не может его закрыть: не работает ни один из привычных способов завершения программы, даже " ^C " Vi обрабатывает по-своему\*. И " :w ", и " :q " – команды **режима командной** строки; в этот режим Vi переводится из **командного режима** с помощью " :", набираемого в начале команды.

Однако если в *буфере* есть изменения, которые еще не записаны ни в каком файле, то Vi откажется выполнять команду " :q ", предложив сначала сохранить эти изменения. Если вы не хотите сохранять изменения, нужно настоять на своем желании выйти из Vi, добавив к команде выхода восклицательный знак: " :q!". В этом случае все несохраненные изменения будут выброшены. Восклицательный знак можно добавить в конце любой файловой команды в командной строке Vi – в этом случае Vi будет без возражений выполнять команды.

В Vi предпринято множество усилий для экономии сил и времени пользователя, командующего редактором. Поэтому можно одним разом записать текст и выйти из редактора: командой " :wq " или аналогичной командой " :x ", просто нажав " ZZ " в командном режиме.

**Перемещение по тексту**

При редактировании в тексте всегда есть *точка*, в которой пользователь "находится" в данный момент: вводимые с клавиатуры символы появятся именно здесь, удаляться символы будут тоже отсюда, к этой *точке* будут применяться команды редактора и т. п. Обычно эта *точка* обозначается курсором, а в последней (информационной) строке экрана *Vim* указывает номера текущей строки и колонки (номер символа в строке), в которых находится курсор.

Для того чтобы выполнять редактирование текста, по нему необходимо перемещаться, т. е. перемещать курсор. Самый очевидный способ это делать – воспользоваться клавишами со стрелками. Нажатие на одну из этих клавиш обычно заставляет курсор перемещаться на один символ влево/вправо или на одну строку вверх/вниз. Трудно придумать более неэффективный и медленный способ перемещения, если нужно попасть на другой конец объемного текста, и даже простое перемещение к началу или концу строки может занять несколько секунд.

Заметим, что в процессе редактирования текста обычно возникает необходимость перемещаться не в произвольную *точку*, а в некоторые ключевые: **начало** и **конец** строки, слова, предложения, абзаца, выражения, заключенного в скобки, целого текста. Особенно это заметно, если требуется редактировать **структурированный** текст: программу (например, сценарий), *конфигурационный файл* и т. п. В Vi для каждого такого перемещения предусмотрены специальные команды, обычно состоящие в нажатии **одной** клавиши в командном режиме. Используя их, можно не только нажав одну кнопку переместиться на любое расстояние в тексте, но и двигаться по структурным элементам, переходя к предыдущему/следующему слову, предложению, абзацу, скобке и т. д. 8

Итак, передвинуть курсор на начало текущей строки можно командой " 0 ", на первый непробельный символ в строке – " ^ ", в конец строки – " $ ". 9 Абзацами Vi считает фрагменты текста, разделенные пустой строкой, к началу предыдущего/следующего абзаца можно попасть командами " { " и " } " соответственно. Очень распространенная задача – необходимость попасть в самый конец файла: для этого служит команда " G " (" **G** o"), в самое начало – " gg ".

Передвинуть курсор вперед на начало следующего слова можно командой " w " (от "word", слово), на начало предыдущего – " b " (от " **b** ackward", назад). К началу предыдущего/следующего предложений можно переместиться командами " ( " и " ) " соответственно. Нужно учитывать, что границы слов и предложений Vi находит по формальным признакам (руководствуясь специально определенными *регулярными выражениями* ), поэтому решение Vi может иногда не совпадать с представлениями пользователя о границах слов и предложений. Однако пользователь всегда властен изменить соответствующие *регулярные выражения*, подробности – в документации по *Vim*.

В *Vim* никогда не следует вручную повторять одну и ту же команду: если нужно перейти на три слова вперед, не следует трижды нажимать " w " – для повторения команды используется *множитель.* ***Множитель*** – это любое число, набранное перед командой Vi: команда будет повторена соответствующее число раз. Например, " 3w " – означает "трижды переместить курсор на слово вперед", иначе говоря, переместить курсор на три слова вперед. Обратите внимание, что *множители* могут применяться не только с *командами перемещения*, но и с **любыми** командами Vi. Аналогично можно переместить курсор на 10 абзацев вперед командой " 10} ".

**Множитель**. Число, предшествующее команде *Vim* и означающее, что данную команду следует выполнить указанное число раз.

Не сразу очевидно, что поиск шаблона в тексте (строки или *регулярного выражения* ) – это тоже *команда перемещения*. Как и любое перемещение, поиск осуществляется в командном режиме: прежде всего, нужно нажать " / ": в последней строке появится символ " / ". Далее следует ввести шаблон для поиска – он будет отображаться в этой строке, его можно редактировать. Обычно Vi настроен таким образом, что шаблон для поиска интерпретируется как *регулярное выражение*, где ряд символов имеет специальное значение, эту настройку можно отключить (" :set nomagic "). После того, как введен шаблон, следует нажать Enter – курсор переместится к ближайшему (далее по тексту) совпадению с шаблоном. Поиск в обратном направлении (к предыдущему совпадению) следует начинать с команды " ?".

Совсем просто перейти к следующему употреблению в тексте того слова, на котором стоит курсор: для этого нужно просто нажать " \* " в командном режиме. Аналогичная команда поиска слова в обратном направлении – " # ". Можно специально отметить в тексте *точку* и затем в любой момент возвращаться к ней, как к закладке. Одну закладку определяет сам Vi – " `` ", место в тексте, где было сделано последнее изменение. Подробно об этих и других командах перемещения можно прочесть в руководстве по *Vim* по команде " :help usr\_03.txt ".

**Изменение текста**

В командном режиме нельзя вводить символы в текст с клавиатуры, но изменить текст при этом можно, например, удаляя символы. Чтобы удалить отдельный символ (тот, на котором стоит курсор), достаточно нажать " x " в командном режиме, а чтобы удалить сразу целую строку (естественно, текущую, то есть ту, в которой находится курсор) – " dd ". " d " – это сокращение от " **d** elete", удалить, а " dd " – характерный прием Vi: удвоение команды означает, что ее нужно применить к текущей строке.

Из командной строки Vi можно выполнить операцию поиска и замены: для простой строки или для *регулярного выражения*. Причем синтаксис команды поиска и замены полностью воспроизводит синтаксис потокового редактора sed, о котором уже шла речь в лекции 7:

Это файл для примеров.

Пример 1.

Моя первая строка в vi...

~

~

~

:s/.$/.../

Пример 9.5. Замена по шаблону в Vi

Мефодий хотел заменить в своем файле *точки* в концах строки многоточиями. Для этого в командном режиме он нажал " :" (вызов командной строки Vi), где набрал команду " s " (сокращение от "substitute", заменить), за которой следует уже знакомое по "sed" выражение " /что\_заменить/на\_что\_заменить/ ". Только результат получился совсем не тот, какого ожидал Мефодий: заменился на многоточие восклицательный знак последней строки. Не учел он следующего: по умолчанию шаблоны для поиска и замены – это *регулярные выражения*, то есть " ." означает вовсе не *точку*, а "любой символ". Имея в виду *точку*, следовало написать " \.". " $ ", как и ожидал Мефодий, это конец строки. В момент выполнения команды поиска курсор находился в последней строке, в первом совпадении после курсора и была произведена замена.

Форматирование текста – это расстановка символов конца строки, пробелов и табуляций таким образом, чтобы текст хорошо смотрелся на экране терминала. Форматирование вручную крайне неэффективно. В *Vim* автоматическое форматирование текста (если редактируется программа на каком-либо языке программирования, то с учетом правил этого языка) может происходить прямо в режиме вставки, в режиме вставки же можно изменять отступ текущей строки (по командам " ^D " и " ^T "). Для выравнивания текста по центру, правому или левому краю команды " :center ", " :left " и " :right " соответственно. Эти команды, как и большинство команд командной строки, можно применять к *диапазону строк* способом, описанным ниже.

Режим ввода не богат специальными командами изменения текста, что и понятно: он предназначен для **ввода**. Однако в *Vim* (но не в Vi!) есть некоторые удобства, упрощающие и сам процесс набора. Если слово, которое нужно ввести, уже встречалось в тексте, можно набрать только первые буквы и нажать " ^P " (" **p** revious") – *Vim* попробует сам завершить его. Если *Vim* не угадал и предложил не то слово, можно продолжить перебирать варианты. Команда " ^N " (" **n** ext") подставляет слова, которые встречаются ниже по тексту. Подробнее об этой функции можно узнать из руководства по команде " :help ins-completion ".

Иногда пользователь, изменив текст, тут же хочет вернуть все обратно. Для этого в Vi предусмотрена команда отмены последнего изменения: " u " в командном режиме (от " **u** ndo", отменить). Каким бы сложным, масштабным (и разрушительным) ни было изменение, совершенное последней командой, " u " вернет текст в исходное состояние. Впрочем, саму отмену тоже можно отменить. В классическом Vi доступна отмена только последней выполненной команды, а в *Vim* отменить можно сколько угодно последних команд, а также повторить их командой " ^R ".

**Работа с фрагментами текста**

Любая *команда перемещения* определяет две *точки* в тексте: ту, в которой был курсор до перемещения, и ту, в которую он переместился в результате данной команды. Расположенный между этими двумя *точками* отрезок текста однозначно задается командой перемещения. Например, команда " ) " захватывает текст от текущего положения курсора до начала следующего предложения. Vi позволяет применить к этому фрагменту любую команду редактирования – так устроены *гнездовые команды*. ***Гнездовая команда*** состоит из **действия** и следующего за ним **перемещения**. Перемещение задает фрагмент текста, а действие определяет, что с этим фрагментом сделать. Например, команда " d) " удалит весь текст от текущей позиции курсора до начала следующего предложения. Наиболее полезные действия – " d " (" **d** elete"), " c " (" **c** *hange*"), " > " и " < " (сдвинуть), " y " (запомнить) и " gq " (отформатировать).

**Гнездовая команда**. Команда редактора Vi, позволяющая применить указанное **действие** к указанному **отрезку текста**. Отрезок задается стандартной командой перемещения по тексту.

Очень часто возникает необходимость заменить фрагмент в тексте: слово, предложение, строку и т. д. Это можно сделать в два действия: сначала удалить часть текста, затем перейти в *режим вставки* и вставить замену. Vi предоставляет возможность упростить эту операцию, сведя два действия к одному: заменить. *Гнездовая команда* " c " предназначена именно для этого. Например, команда " cw " (буквально: " **c** *hange* **w** ord") заменит текст от курсора до начала следующего слова (так можно заменить одно слово), " c) " заменит текст от курсора до начала следующего предложения.

Мефодий не забыл, что *команды перемещения* можно использовать с *множителями*, и попробовал заменить сразу три слова в своем файле на другие: для этого он в командном режиме подогнал курсор в начало слова "первая" и набрал " c3w " ("заменить фрагмент отсюда до начала третьего слова", буквально: "change 3 words"). Результат этой команды выглядел так:

Это файл для примеров.

Пример 1.

Моя #vi...

~

~

~

-- ВСТАВКА -- 3,5 Весь

Пример 9.6. Команда замены в Vi

В примере знак " # " обозначает позицию курсора: как видно, Vi удалил три слова, попавшие в зону действия команды и сразу перешел в режим вставки. Мефодию осталось только набрать что-то взамен.

Перестановка частей – типичная задача, возникающая в процессе редактирования. Для перестановки требуется удалить фрагмент в одном месте текста и вставить его же в другом. Для решения первой части задачи в Vi нет специальных средств, потому что любая команда удаления (" d ", " c ", " x " и др.) сохраняет удаленный текст в специальном *регистре*. Для вставки последнего удаленного фрагмента служит команда " p " в командном режиме (от " **p** ut", положить). При помощи этой команды Мефодий может вставить только что удаленные им командой " c3w " три слова:

Это файл для примеров.

Пример 1.

Моя новая строка для vi...

первая строка в

~

~

4,1 Весь

Пример 9.7. Вставка удаленного фрагмента

Для того чтобы скопировать фрагмент текста, служит команда " y " (от " **y** ank" – забрать, сдернуть): она не удаляет текст, а просто сохраняет его в том же *регистре*, что и команды удаления. Команду " y " можно использовать в *гнездовых командах*, например, " y5w " сохранит в *регистре* фрагмент текста от курсора до начала пятого слова. Вставить скопированный фрагмент можно все той же командой " p ". Однако таким способом можно вставлять только текст, удаленный или скопированный **последним**, для хранения нескольких разных фрагментов текста следует использовать именованные *регистры* (см. подробнее в руководстве по *Vim*).

Для того чтобы применить команду к нескольким **строкам** текста, не обязательно подгонять к ним курсор. В командной строке Vi любой команде может предшествовать указание *диапазона* в тексте, к которому следует эту команду применить. Команды с указанием *диапазона* выглядят так: " :начало,конец}команда ", где начало и конец – это адреса начальной и конечной строк *диапазона* (т. е. фрагмента текста), а команда – это команда режима **командной строки**, такая как " :w " или " :r ". Многие команды **командного** режима (в частности, " d " и " y ") доступны также и в командной строке. В качестве адреса можно использовать номера строк в файле (команда " :1,5y " будет означать "скопировать в *регистр* строки с первой по пятую"), есть специальные обозначения для текущей строки (" ."), последней строки (" $ ") и всего файла (" % "). Указать границу *диапазона* можно и при помощи шаблона: граничной будет считаться та строка, в которой обнаружится шаблон.

Последним свойством Мефодий воспользовался, чтобы удалить плоды своих экспериментов: он выполнил команду " :/Пример 1/,$d " (от строки " Пример 1 " до конца файла – удалить):

Это файл для примеров.

~

~

~

3 fewer lines

Пример 9.8. Удаление диапазона по шаблону. Выполнена команда ":/Пример 1/,$dEnter".

**Настройка Vi и Vim**

Вид и поведение Vi и *Vim* можно существенно изменить при помощи настроек, приспособив редактор именно к своим вкусам и привычкам. Прямо во время работы редактора можно менять настройки из командной строки Vi при помощи команды " :set имя\_настройки ". Кроме того, можно сделать настройку постоянной, вписав все нужные значения в *конфигурационный файл* .vimrc (или .exrc – для Vi) в домашнем каталоге пользователя. При каждом запуске *Vim*/Vi читает этот файл и исполняет все содержащиеся в нем команды. Объем лекции не позволяет подробно остановиться на настройке Vi, читатель может заняться изучением этого вопроса сам: все необходимые сведения есть в руководствах. Чтобы оценить возможности настройки, можно выполнить в *Vim* (но не в Vi!) команду " :options ", по которой будет выведен список всех доступных опций с кратким описанием их смысла.

**Лучше, чем Emacs?**

Заголовок этого раздела сформулирован таким провокационным образом не случайно: любому пользователю похожей на *UNIX* операционной системы (к таким системам относится и Linux) необходим *текстовый редактор*. Выбор очень многих пользователей падает на один из двух наиболее популярных и, как следствие, наиболее универсальных (реализованных и работающих везде) *текстовых редактора*: *Vi* (или одну из его версий, чаще всего *Vim*) и Emacs. Оба эти редактора появились около тридцати лет назад, но почтенный возраст им на пользу: огромное количество разработчиков *по* всему миру все эти годы занималось их отладкой, локализацией и расширением.

*Vim* и Emacs образуют альтернативную пару не только *по* исторической случайности: оба редактора претендуют на роль универсального средства для работы с текстом на любых естественных и искусственных языках. И действительно, трудно назвать сравнимый с ними *по* возможностям *текстовый редактор*, да еще и настолько нетребовательный к интерфейсу: и *Vim*, и Emacs будут работать на любом терминале. Однако ограниченные возможности интерфейса терминала требуют от программ надежного способа отделения команд редактора от вводимого текста. В *Vi* и Emacs эта задача решена *по*-разному – отсюда и большая часть различий в стиле работы с этими редакторами, и традиционный спор приверженцев этих редакторов о том, какой из них лучше.

**Тексты на разных языках**

Главное свойство, которое сделало Emacs таким популярным и многофункциональным редактором – это заложенная в нем с самого начала принципиальная **расширяемость**. Emacs почти полностью написан на специально созданном для него языке программирования Emacs Lisp, и у любого пользователя есть возможность запрограммировать необходимые **именно ему** функции и подключить их в качестве модуля к Emacs. При этом сам Emacs никак изменять не требуется. Сообщество программистов не замедлило воспользоваться расширяемостью Emacs, и к настоящему времени важнейшее достоинство этого редактора состоит именно в свободно распространяемых пакетах расширений, содержащих инструменты для решения самых разнообразных задач, связанных с редактированием текста.

Современный Emacs – это не просто *текстовый редактор*, а интегрированная среда **для работы в системе**. Основная идея сообщества разработчиков и пользователей Emacs состоит в том, что Emacs позволяет работать с любыми данными, которые могут быть представлены как текст (в лекции 7 обсуждалось, что таким образом можно представить очень многое в системе). Естественно, список содержащихся в каталоге файлов, программа на каком-либо языке программирования или электронное письмо – это тексты, которые сильно различаются по структуре и по тому, что от них нужно пользователю. В Emacs для работы с текстами разного типа используются режимы. 10

**Режим Emacs**. Комплекс команд и настроек Emacs, предназначенных для работы с текстом определенной структуры, например содержимым каталога, программой на Си и. т. п.

Каждый *буфер* в Emacs находится в одном из *основных режимов*. ***Основной режим*** – это набор функций и настроек Emacs, приспособленных для редактирования текста определенного вида. Каждый *основной режим* по-своему переопределяет некоторые *управляющие символы*, так что наиболее доступными становятся команды, чаще всего именно для работы с текстом данного типа. Команды, специфичные для текущего *основного режима*, обычно начинаются с *управляющего символа* C-c. Некоторое представление о возможностях Emacs может дать весьма неполный список тех текстов, для которых существуют *основные режимы*:

* список файлов в каталоге;
* программы на самых разных языках программирования, от Си до самых экзотических;
* тексты в различных форматах *разметки*: XML, HTML, TeX;
* словари;
* электронная почта (режим позволяет не только читать и писать письма, но и отправлять и получать их);
* календарь;
* дневник и личное расписание;
* многое другое.

Когда требуется много и быстро работать с текстом на каком-либо искусственном языке (языке программирования, *разметки* и пр.), возможно, Emacs – это лучший выбор.

Не хуже в Emacs развиты и средства работы с текстами на разных естественных языках с самыми экзотическими письменностями. Просто для оценки возможностей Emacs в этой *области* можно выполнить команду " C-h h ", по которой будет выведен файл, изображающий приветствие на разных языках.

**Команды Emacs**

Если в вашей системе установлен Emacs, то вы можете его запустить, набрав emacs в командной строке любого терминала. Как и *Vim*, Emacs использует весь экран терминала, хотя интерфейс у него более богатый: вверху экрана находится строка с пунктами меню, под ней – окно для отображения и редактирования текста, которое заканчивается строкой *режима*, отображаются сведения о происходящем в окне. В самом низу экрана – строка *минибуфера*, используемая для диалога с пользователем, в частности для отображения и редактирования вводимых команд.

Vi и вслед за ним *Vim* – это **многорежимные** редакторы, когда команды вводятся в одном режиме, а текст – в другом, что позволяет использовать в качестве командных любые клавиши. В Emacs нет специального командного режима, но использован тот факт, что с клавиатуры можно вводить не только печатные, но и некоторые *управляющие символы*. Для этого используются несколько управляющих клавиш терминала (прежде всего Ctrl и Alt ), нажатые в сочетании с различными текстовыми символами. Чтобы ввести такой символ, нужно нажать управляющую клавишу (например, Ctrl ) и, удерживая ее, нажать клавишу с одним из печатных символов (например, " x "). Кроме того, в Emacs используется *управляющая клавиша* Meta. На тех терминалах, где она отсутствует, ее функции обычно передаются клавише Alt. На "настоящих" терминалах обычно не бывает ни Meta, ни Alt ; из *клавиатурных модификаторов* присутствуют только Ctrl и Shift. Тогда на помощь приходит старая добрая ESC: нажатие ESC, а **после** нее – печатного символа (того же " x ") эквивалентно " Meta x ".

Команд в редакторе Emacs чрезвычайно много, доступных *управляющих символов* на всех не хватает, поэтому чтобы вызвать команду Emacs, обычно требуется ввести ключ, начинающийся с *управляющего символа*, за которым следует комбинация из *управляющих* или обычных символов, просто полное имя команды. Последовательность символов, достаточная для вызова команды, называется ***законченным ключом***, а если введенных символов недостаточно для однозначного определения команды, это – ***префиксный ключ*** .

Общее правило здесь таково: чем чаще встречается команда, тем короче вызывающий ее ключ, и наоборот. Для лаконичной записи длинных клавиатурных комбинаций в сообществе пользователей Emacs сложилась особая традиция сокращенных обозначений. Клавишу Ctrl обозначают заглавной буквой " C ", Meta – " M " 11. Сочетания с командной клавишей обозначаются дефисом, например, запись С-h обозначает, что нужно, удерживая Ctrl, нажать " h ". C-h – это *префиксный ключ* для команд справочной системы Emacs. Начинающему пользователю стоит выполнить команды " C-h?" (набрать C-h и затем нажать " ?") – справка по командам помощи, " C-h t " – интерактивный учебник для начинающих пользователей Emacs, и " C-h i " – полное руководство по Emacs (в формате info ). С ключа C-x начинаются основные команды Emacs, в частности, для работы с файлами и *буферами*. Чтобы завершить работу Emacs, нужно ввести " C-x C-c ".

У любой команды Emacs есть собственное имя. По этому имени можно вызвать команду, даже если она не привязана ни к какому клавиатурному ключу. Для вызова команд по имени используется *префиксный ключ* M-x. Например, посмотреть справку о помощи в Emacs можно командой " M-x help-for-help ".

**Работа с файлами**

В Emacs, как и в *Vim*, пользователь редактирует текст не в самом файле, а в *буфере*. Отличие Emacs в том, что нельзя написать "безымянный" текст и затем сохранить его в файле. При запуске Emacs без параметров открывается специальный *буфер* – " \**scratch*\* ". Он предназначен для временных заметок – его содержимое будет выброшено при закрытии Emacs. Если нужно создать новый файл – его следует открыть командой " C-x C-f ", точно так же открывается для редактирования уже существующий файл.

После того как Мефодий нажал " C-x C-f ", в *минибуфере* возникло приглашение: " Find file: ~/ ". Теперь нужно ввести *путь* к файлу, начиная с текущего каталога (Emacs любезно подсказал его Мефодию). С текстом в *минибуфере* можно обращаться почти так же, как с командной строкой shell или *Vim*: редактировать, использовать автодополнение (клавишей Tab ), перемещаться по истории стрелочками вверх/вниз. Мефодий не замедлил воспользоваться этой возможностью и, набрав " te ", нажав Tab и Enter, приступил к редактированию того же " textfile ". Сохранить сделанные изменения можно командой " C-x C-s ".

Когда Мефодий захотел открыть еще один *буфер*, чтобы выполнить один из своих сценариев, он забыл точное название нужного файла и, набрав " C-x C-f bin/ ", нажал Enter. В окне возник список файлов в подкаталоге " ~/bin ", похожий на вывод ls -l:

File Edit Options Buffers Tools Operate Mark Regexp Immediate Subdir Help

/home/methody/bin:

итого 24

drwxr-xr-x 2 methody methody 4096 Дек 2 15:21 .

drwx------ 10 methody methody 4096 Дек 2 15:21 ..

-rwxr-xr-x 1 methody methody 26 Ноя 9 21:34 loop

-rwxr-xr-x 1 methody methody 23 Ноя 9 21:34 script

-rwxr-xr-x 1 methody methody 32 Ноя 9 21:34 to.sort

-rwxr-xr-x 1 methody methody 44 Ноя 9 21:34 two

-RRR:%%-F1 bin (Dired by name)--L5--C51--All--Reading directory /home/methody/bin/...done

Пример 9.9. Emacs. Режим dired

Как указано в *строке режима*, это Dired, редактор каталогов, *режим Emacs*, предназначенный для просмотра и изменения каталогов прямо из редактора. В Dired можно выбирать отдельные файлы или группы файлов и производить над ними различные действия: открыть для редактирования, удалить, скопировать, переместить, переименовать по определенной схеме и т. д. Одним словом, Dired – довольно мощное средство для наглядной работы с файловой системой, особенно он удобен для работы с группой файлов. Подробности о командах, доступных в этом *режиме*, можно найти в руководстве по Emacs.

**Перемещение по тексту**

В Emacs, как и в *Vim*, есть понятие " *точка* " – то место в *буфере*, где будет происходить вставка или удаление данных. Перемещение по тексту – это перемещение *точки*. *Команды перемещения* по структурным элементам текста развиты не менее, чем в *Vim* – помимо обычных стрелок, действуют *команды перемещения* в начало и конец строки ( C-a и C-e ), *буфера* ( M-< и M-> ), предложения ( M-a и M-e ), к предыдущему и следующему слову ( M-f и M-b ), абзацу ( M-{ и M-} ). Различные *основные режимы* предоставляют специализированные команды для перемещения по структурным элементам текстов на разных языках программирования, *разметки* и др.

В Emacs несколько видов поиска: существуют отдельные команды для поиска строки и поиска по *регулярному выражению*. Если требуется найти ближайшее употребление конкретного слова, удобнее всего воспользоваться *наращиваемым* поиском по команде C-s. *Наращиваемый поиск* уже встречался Мефодию: так был устроен поиск по *истории команд* в bash. По мере набора первых символов искомой строки Emacs переносит *точку* к ближайшему подобному сочетанию символов после курсора. Поиск в обратном направлении (к началу *буфера* ) осуществляется командой C-r. *Наращиваемый поиск* можно выполнять по *регулярному выражению* ( C-M-s ). Все виды *наращиваемого поиска* в Emacs не различают прописные и строчные буквы.

**Изменение текста**

В Emacs есть множество команд, позволяющих пользователю выполнять меньше действий при редактировании текста. Если пользователь осознает, что набрал что-то неправильно, он может разом удалить последнее слово ( M-Del ) или предложение ( C-x Del ). Можно уничтожать и вперед: до конца слова ( M-d ) и предложения ( M-k ). Emacs хранит не только последний удаленный фрагмент, но и все предыдущие, формируя *список удалений*. Только что уничтоженный текст можно вставить командой C-y. После этого его можно заменить **предыдущим** уничтоженным фрагментом – M-y. Можно двигаться и далее назад по списку удалений, повторяя M-y.

Хорошо продуманы команды для перестановки частей текста вокруг *точки*: двух знаков ( C-t ), слов ( M-t ), строк ( C-x C-t ). Команда M-t не перемещает знаки препинания между словами, поэтому " потеха, дело " превратится в " дело, потеха ".

Прямо из Emacs можно вызвать программу проверки орфографии (" M-x ispell-buffer ") или даже включить проверку "на лету", когда неправильно написанные слова выделяются другим цветом (" M-x flyspell-mode "). Можно проверить написание отдельного слова, в котором находится *точка* (" M-x $ ") или завершить недописанное слово, основываясь на орфографическом словаре (" M-x Tab ").

В Emacs так много специальных команд для изменения текста, что команды поиска и замены бывают нужны не так часто. Произвести замену строки всюду в *буфере* можно по команде " M-x replace-string что заменить Enter на\_что\_заменить Enter ", а для замены *регулярного выражения* существует аналогичная команда " M-x replace-*regexp* ".

Если нужно заменить строку только в некоторых случаях, пригодится команда M-%, запрашивающая подтверждение о замене при каждой найденной строке. Аналогичная команда для *регулярных выражений* – C-M-%.

Любые изменения в тексте можно отменить командой C-\_ (нужно нажать Ctrl, Shift и " - ").

**Работа с фрагментами текста**

Многие команды Emacs работают с произвольным фрагментом текущего *буфера*. Такие команды всегда применяются к текущей *области*. *Область* – это отрезок текста между *точкой* (где находится курсор) и *меткой*. *Метка* в любой момент присутствует в любом *буфере*, пользователь может установить ее в любом месте текста явно – командой M-Пробел. *Метка* может перемещаться и без вмешательства пользователя: команды перемещения и редактирования могут изменять положение *метки*. Таким образом, чтобы выделить в *буфере* фрагмент текста, можно проделать следующие операции:

* переместить *точку* (курсор) на один конец нужного фрагмента;
* нажать M-Пробел (установить *метку* );
* переместить *точку* к другому концу нужного фрагмента.

Теперь можно выполнять команду редактирования – она будет применена именно к выделенной *области*. Например, C-w удалит текст *области*, а M-w скопирует его. Вставить удаленный или скопированный фрагмент можно командой C-y.

Есть группа команд, позволяющих работать с *меткой* более эффективно: установить *метку* после конца следующего слова ( M-@ ), пометить текущий абзац ( M-h ) или весь *буфер* (" C-x h "). Различные *основные режимы* предоставляют команды для пометки структурных элементов текста, например, разделов документа, определения функции (в тексте программы) и т. п. Все положения *метки* хранятся в *списке пометок*, перенести *точку* в любое из предшествующих положений *метки* можно, нужное количество раз повторив команду " C-u C-@ ".

**Область**. Непрерывный отрезок текста, ограниченный *точкой* с одной стороны и *меткой* с другой.

Как и в *Vim*, в Emacs можно использовать для хранения информации *регистры*. В *регистре Emacs* можно сохранить позицию в *буфере* и затем перейти к этой позиции (" C-x r Пробел x " записывает позицию *точки* в *регистр* " x ", а " C-x r j x " переходит в эту позицию). В *регистре* можно сохранить текст из *области* (" C-x r s x " сохраняет *область* в *регистре* " x ", " C-x r i x " – вставляет текст из этого *регистра* ). В *регистрах* также можно хранить числа, имена файлов, конфигурацию окон. Подробности, как всегда, в руководстве.

**Настройка Emacs**

Коротко говоря, в Emacs можно настроить все: связи между ключами и командами редактора, определить макрокоманды, написать собственные расширения. Есть возможность изменять настройки Emacs как в процессе работы, так и при помощи *конфигурационного файла* .emacsrc.

**Просто текстовые редакторы**

И в *Vim*, и в Emacs интегрировано множество средств для автоматизации процесса редактирования. Эти редакторы становятся удобны в том случае, если прежде чем делать что-то вручную, *пользователь* обращается к руководству и находит в нем способ выполнить свою задачу максимально быстро и с минимальными затратами ручного труда. Однако если пользователя не устраивает такой принцип работы (когда нужно часто читать документацию и думать, как организовать так, чтобы "ручную" работу выполнял *компьютер*), *Vim* и Emacs будут не самым лучшим выбором. Для обычного редактирования текста вручную лучше выбрать один из *текстовых редакторов* с простым и привычным интерфейсом: в дистрибутивах Linux можно найти огромное количество таких *текстовых редакторов* с большими или меньшими возможностями: mcedit, joe, pico (часть почтовой программы pine ) – всех не перечислить. Есть редакторы, которые предназначены для работы не в терминале, а в графической среде (например, nedit ), у тех же *Vim* и Emacs есть графические варианты (GVim и Emacs-X11 или XEmacs), в которых доступны дополнительные возможности графического пользовательского интерфейса: *меню*, иконки и пр.

**Лекция 10:**

**Этапы загрузки системы**

**Досистемная загрузка**

*Программа* login, регистрирующая пользователей в системе, запускается только тогда, когда сама система уже приведена в полную готовность и работает в обычном режиме. Происходит это далеко не сразу после включения компьютера: Linux - довольно сложная система, объекты которой попадают в оперативную *память* не сами собой, а в процессе загрузки. Сама *загрузка* - процесс ступенчатый: поведение компьютера на различных этапах загрузки определяется разными людьми - от разработчиков аппаратной составляющей до системного администратора. Предъявляемые к системе требования гибкости, возможности изменять ее настройку в зависимости от аппаратной составляющей, необходимость решать **разные** задачи с помощью одного и того же компьютера тоже делают процесс загрузки ступенчатым: сначала определяется **профиль** будущей системы, а затем этот профиль **реализуется**.

Начальный этап вообще не зависит от того, какая *операционная система* установлена на компьютере, для некоторых этапов в каждой *операционной системе* предлагаются свои решения - по большей части, взаимозаменяемые. Эту стадию (начальную) назовем ***досистемной загрузкой*** . Начиная с определенного этапа, загрузка компьютера уже управляется самой Linux, используются утилиты, сценарии и т.п. Эту стадию (завершающую) назовем ***системной загрузкой*** .

**Загрузчик в ПЗУ**

Сразу после включения оперативная память компьютера классической архитектуры девственно чиста. Для того чтобы начать работать, процессору необходима хоть какая-то программа. Эта программа автоматически загружается в память из *постоянного запоминающего устройства*, *ПЗУ* (или ROM, read-only memory), в которое она вписана раз и навсегда в неизменном виде1. В **специализированных** компьютерах (например, в дешевых игровых приставках) все, что нужно пользователю, записывается именно на *ПЗУ* (часто сменное), и запуском программы оттуда загрузка заканчивается.

Обычно в компьютерах общего назначения программа из *ПЗУ* пользователю ничем полезна не бывает: она невелика, да и делает всегда одно и то же. Слегка изменить поведение программы из *ПЗУ* можно, оперируя данными, записанными в *энергонезависимую память* (иногда ее называют CMOS, иногда - *NVRAM*). Объем *энергонезависимой памяти* очень невелик, а данные из нее сохраняются после выключения компьютера за счет автономного электропитания (как правило, от батарейки вроде часовой).

Что должна уметь эта начальная программа? Распознавать основные устройства, на которых может быть записана другая - нужная пользователю - программа, уметь загружать эту программу в память и передавать ей выполнение, а также поддерживать интерфейс, позволяющий менять настройки в *NVRAM*. Собственно, это даже не одна программа, а множество **подпрограмм**, занимающихся взаимодействием с разнообразными устройствами ввода-вывода - как с теми, на которых могут храниться программы (жесткие и гибкие диски, магнитные ленты и даже сетевые карты), так и теми, посредством которых можно общаться с пользователем (последовательные порты передачи данных - если есть возможность подключить консольный терминал, системная клавиатура и видеокарта - для простых персональных рабочих станций). Этот набор подпрограмм в *ПЗУ* обычно называется ***BIOS*** (basic input-output system).

**BIOS**. Сокращение от "Basic Input-Output System", набор подпрограмм в *ПЗУ*, предназначенных для простейшего низкоуровневого доступа к внешним устройствам компьютера. В современных ОС используется только в процессе начальной загрузки.

Этот этап загрузки системы можно назвать **нулевым**, так как ни от какой системы он не зависит. Его задача - определить (возможно, с помощью пользователя), с какого устройства будет идти загрузка, загрузить оттуда специальную программу- *загрузчик* и запустить ее. Например, выяснить, что устройство для загрузки - жесткий диск, считать самый **первый** сектор этого диска и передать управление программе, которая находится в считанной области.

**Загрузочный сектор и первичный загрузчик**

Чаще всего размер *первичного* дискового *загрузчика* - программы, которой передается управление после нулевого этапа, - весьма невелик. Это связано с требованиями **универсальности** подобного рода программ. Считывать данные с диска можно **секторами**, размер которых различается для разных типов дисковых устройств (от половины килобайта до восьми или даже больше). Кроме того, если считать один, первый, сектор диска можно всегда одним и тем же способом, то команды чтения нескольких секторов на разных устройствах могут выглядеть по-разному. Поэтому-то *первичный загрузчик* занимает обычно не более одного сектора в самом начале диска, в его *загрузочном секторе*.

Если бы *первичный загрузчик* был побольше, он, наверное, и сам мог бы разобраться, где находится *ядро* *операционной системы*, и смог бы самостоятельно считать его, разместить в памяти, настроить и передать ему управление. Однако *ядро* *операционной системы* имеет довольно сложную структуру - а значит, и непростой способ загрузки; оно может быть довольно большим, и, что неприятнее всего, может располагаться неизвестно где на диске, подчиняясь законам *файловой системы* (например, состоять из нескольких частей, разбросанных по диску). Учесть все это *первичный загрузчик* не в силах. Его задача скромнее: определить, где на диске находится "большой" *вторичный загрузчик*, загрузить и запустить его. *Вторичный загрузчик* прост, и его можно положить в заранее определенное место диска, или, на худой конец, положить в заранее определенное место *карту размещения*, описывающую, где именно искать его части (размер *вторичного загрузчика* ограничен, поэтому построить такую *карту* возможно).

**Карта размещения**. Представление области с необходимыми данными (например, *вторичным загрузчиком* или *ядром* системы) в виде списка секторов диска, которые она занимает.

В случае IBM-совместимого компьютера размер *загрузочного сектора* составляет всего **512 байтов**, из которых далеко не все приходятся на **программную** область. *Загрузочный сектор* IBM PC, называемый ***MBR*** (*master boot record*), содержит также *таблицу разбиения диска* , структура которой описана в лекции 11. Понятно, что программа такого размера не может похвастаться разнообразием функций. Стандартный для многих систем *загрузочный сектор* может только считать *таблицу разбиения диска*, определить так называемый *загрузочный раздел* (*active partition*) и загрузить программу, расположенную в начале этого *раздела*. Для каждого типа диска может быть своя программная часть *MBR*, что позволяет считывать данные из любого места диска, сообразуясь с его типом и геометрией. Однако считывать можно все же не более одного сектора: неизвестно, для чего используются установленной на этом разделе *операционной системой* **второй** и последующие сектора. Выходит, что стандартная программная часть *MBR* - это некий **предзагрузчик**, который считывает и запускает настоящий *первичный загрузчик* из первого сектора *загрузочного раздела*.

Существуют версии *предзагрузчика*, предоставляющие пользователю возможность самостоятельно **выбрать**, с какого из разделов выполнять загрузку2. Это позволяет для каждой из установленных *операционных систем* хранить **собственный** *первичный загрузчик* в начале раздела и свободно выбирать среди них. В стандартной схеме загрузки Linux используется иной подход: простой *первичный загрузчик* записывается прямо в *MBR*, а функция выбора передается *вторичному загрузчику*.

**Первичный загрузчик**. Первая стадия загрузки компьютера: программа, размер и возможности которой зависят от аппаратных требований и функций *BIOS*. Основная задача - загрузить *вторичный загрузчик* .

**Загрузчик ядра**

В задачу *вторичного загрузчика* входит загрузка и начальная настройка *ядра* *операционной системы*. Как правило, *ядро* системы записывается в файл с определенным именем. Но как *вторичному загрузчику* прочитать файл с *ядром*, если в Linux эта операция и есть **функция** *ядра*? Эта задача может быть решена тремя способами.

Во-первых, *ядро* может и не быть файлом на диске. Если загрузка происходит по сети, достаточно попросить у сервера "файл с таким-то именем", и в ответ придет цельная последовательность данных, содержащая запрошенное *ядро*. Все файловые операции выполнит сервер, на котором система уже загружена и работает. В других случаях *ядро* "загоняют" в специально выделенный под это раздел, где оно лежит уже не в виде файла, а таким же непрерывным куском, размер и местоположение которого известны. Однако в Linux так поступать не принято, так как места для специального раздела на диске, скажем, IBM-совместимого компьютера может и не найтись.

Во-вторых, можно воспользоваться описанной выше *картой размещения*: представить *ядро* в виде набора секторов на диске, записать этот набор в заранее определенное место, а *загрузчик* заставить собирать *ядро* из кусков по *карте*. Использование *карты размещения* имеет два существенных недостатка: ее **создание** возможно только под управлением **уже загруженной** системы, а **изменение** *ядра* должно обязательно сопровождаться изменением *карты*. Если по какой-то причине система не загружается ни в одной из заранее спланированных конфигураций, единственная возможность поправить дело - загрузиться с внешнего носителя (например, с лазерного диска). А система может не загружаться именно потому, что администратор **забыл** после изменения *ядра* пересобрать *карту*: в *карте* указан список секторов, соответствовавших **старому** файлу с *ядром*, и после удаления старого файла в этих секторах может содержаться какой угодно "мусор".

В-третьих, можно научить *вторичный загрузчик* распознавать структуру *файловых систем* и находить там файлы по имени. Это заметно увеличит его размер и потребует "удвоения функций" - ведь точно такое же, даже более мощное, распознавание будет и в самом *ядре*. Зато описанной выше тупиковой ситуации можно избежать, если, скажем, не удалять старое *ядро* при установке нового, а переименовывать его. Тогда, если загрузка системы с новым *ядром* не удалась, можно загрузиться еще раз, **вручную** указав имя файла (или каталога) со старым *ядром*, под управлением которого все работало исправно.

*Вторичный загрузчик* может не только **загружать** *ядро*, но и настраивать его. Чаще всего используется механизм настройки *ядра*, похожий на командную строку shell: в роли команды выступает *ядро*, а в роли параметров - настройки *ядра*. Настройки *ядра* нужны для временного изменения его функциональности: например, чтобы выбрать другой графический режим *виртуальных консолей*, чтобы отключить поддержку дополнительных возможностей внешних устройств (если аппаратура их не поддерживает), чтобы передать самому *ядру* указания, как загружать систему и т.п.

Очень часто конфигурация *вторичного загрузчика* предусматривает несколько вариантов загрузки, начиная от нескольких вариантов загрузки одного и того же *ядра* с разными настройками (например, стандартный профиль и профиль с отключенными расширенными возможностями) и заканчивая вариантами загрузки разных *ядер* и даже разных *операционных систем*. Это требует от самого *загрузчика* некоторого разнообразия интерфейсных средств. С одной стороны, он должен уметь работать в непритязательном окружении, например обмениваться с пользователем данными через последовательный порт, к которому подключена *системная консоль*. С другой стороны, если есть стандартные графические устройства ввода/вывода, хотелось бы, чтобы *загрузчик* использовал и их. Поэтому все *загрузчики* имеют универсальный текстовый интерфейс (зачастую с довольно богатыми возможностями) и разнообразный графический (чаще в виде меню).

Особенная ситуация возникает в случае, когда на компьютере установлено несколько *операционных систем* (например, если персональный компьютер используется также и для компьютерных игр, строго привязанных к определенной системе). В этом случае не стоит надеяться на "универсальность" *вторичного загрузчика*: даже если он способен различать **множество** *файловых систем* и **несколько** форматов загрузки *ядер*, невозможно знать их все. Однако если в *загрузочном секторе* **раздела** *операционной системы* записан *первичный загрузчик*, можно просто загрузить его, как если бы это произошло непосредственно после работы *MBR*. Таким образом, *вторичный загрузчик* может выступать в роли **предзагрузчика**, передавая управление "по цепочке" (chainloading). К сожалению, чем длиннее цепочка, тем выше вероятность ее порвать: можно, например, загрузить по цепочке MS-DOS, удалить с его помощью раздел Linux, содержавший *вторичный загрузчик*, а затем переразметить этот раздел, чем и привести компьютер в неработоспособное состояние.

**Вторичный загрузчик**. Вторая стадия загрузки компьютера: программа, размер и возможности которой практически не зависят от аппаратных требований. Основная задача - полностью подготовить и запустить загрузку *операционной системы* .

**Досистемная загрузка Linux**

Несмотря на то, что *досистемная загрузка* не зависит от типа *операционной системы*, которая начинает работу после, большинство систем предоставляют **собственные** средства по ее организации. В Linux наиболее популярны подсистемы загрузки *LILO* ( **LI** nux **LO** ader) и *GRUB* ( **GR** and **U** nified **B** ootloader). Обе эти подсистемы имеют текстовый и графический варианты интерфейса, предоставляющего пользователю возможность выбрать определенный заранее настроенный тип загрузки.

**LILO**

Подсистема загрузки *LILO* использует и для *первичного*, и для *вторичного загрузчика* схему с *картой размещения*. Это делает работу с *LILO* занятием, требующем повышенной аккуратности, так как изменение процедуры загрузки **не атомарно**: сначала пользователь изменяет *ядро* или его модули, потом - редактирует файл /etc/*lilo*.conf, в котором содержатся сведения обо всех вариантах загрузки компьютера, а затем - запускает команду *lilo*, которая собирает таблицы размещения для всех указанных *ядер* и *вторичного загрузчика* и записывает *первичный* и *вторичный загрузчик* вместе с *картами* в указанное место диска. *Первичный загрузчик* *LILO* (он называется LI ) можно записывать и в *MBR*, и в начало **раздела** Linux.

Простейший файл *lilo*.conf может выглядеть так:

boot=/dev/hda

map=/boot/map

image=/boot/vmlinuz-up

root=/dev/hda1

Пример 10.1. Простейшая настройка LILO: пример файла lilo.conf

Такая настройка *LILO* определяет только один вариант загрузки: *первичный загрузчик* записывается в начало первого жесткого диска (строчка boot=/dev/*hda* ), *карту размещения* утилита *lilo* записывает в файл /boot/map, *ядро* добывается из файла /boot/vmlinuz-up, а запись root=/dev/hda1 указывает *ядру*, что *корневая файловая система* находится на первом разделе первого диска.

Одна из машин, за которыми случалось работать Мефодию, использовалась иногда для запуска единственной программы, написанной для MS-DOS. Исходные тексты этой программы давно потерялись, автор - тоже, поэтому на машине пришлось устанавливать и MS-DOS и Linux. В результате *lilo*.conf оказался таким:

[root@localhost root]# cat /etc/lilo.conf

boot=/dev/hda

map=/boot/map

default=linux-up

prompt

timeout=50

image=/boot/vmlinuz-up

label=linux-up

root=/dev/hda5

initrd=/boot/initrd-up.img

read-only

image=/boot/vmlinuz-up

label=failsafe

root=/dev/hda5

initrd=/boot/initrd-up.img

vga=normal

append=" failsafe noapic nolapic acpi=off"

read-only

other=/dev/hda1

label=dos

other=/dev/fd0

label=floppy

unsafe

Пример 10.2. Настройка LILO на двухсистемной машине

Здесь Linux была установлена на пятый раздел диска (о нумерации разделов в IBM-совместимых компьютерах будет рассказано в лекции 11), а на первом находится MS-DOS. Кроме загрузки MS-DOS предусмотрено два варианта загрузки Linux и еще один - любой *операционной системы* с дискеты. Каждый вариант загрузки помечен строкой label=вариант. При старте *LILO* выводит простейшее3 окошко, в котором перечислены все метки (в данном случае - " linux-up ", " failsafe ", " dos " и " floppy "). Пользователь с помощью "стрелочек" выбирает нужный ему вариант и нажимает Enter. При необходимости пользователь может вручную дописать несколько **параметров**, они передадутся *ядру* системы. Если пользователь ничего не трогает, то по истечении тайм-аута выбирается метка, указанная в поле default.

Еще несколько пояснений. Метки linux-up и failsafe в примере используют одно и то же *ядро* ( vmlinuz-up ), но во втором случае перенастраивается режим *графической карты* и добавляются параметры, отключающие поддержку необязательных для загрузки аппаратных расширений (многопроцессорность, автоматическое управление электропитанием и т.п.). Строчку, стоящую после append=, пользователь мог бы ввести и самостоятельно, это и есть параметры *ядра*. Поле initrd= указывает, в каком файле *находится стартовый виртуальный диск* (ему посвящен раздел "*Стартовый виртуальный диск* и модули" этой лекции), а внушающая некоторые опасения надпись " *unsafe* " (для метки floppy ) означает всего лишь, что дискета - съемное устройство, поэтому бессмысленно во время запуска *lilo* проверять правильность ее *загрузочного сектора* и составлять *карту*.

Наконец, записи вида other=устройство говорят о том, что *LILO* неизвестен тип *операционной системы*, находящейся на этом устройстве, а значит, загрузить *ядро* невозможно. Зато ожидается, что в первом секторе устройства будет обнаружен **еще один** *первичный загрузчик*, *LILO* загрузит его и передаст управление по цепочке. Так и загружается MS-DOS на этой машине: *первичный загрузчик* берется (по метке dos ) из начала первого раздела первого диска.

**GRUB**

Подсистема загрузки *GRUB* устроена более сложно. Она также имеет *первичный загрузчик*, который записывается в первый сектор диска или раздела, и *вторичный загрузчик*, располагающийся в *файловой системе*. Однако *карта размещения* в *GRUB* обычно используется только для так называемого "полуторного" *загрузчика* ("stage 1.5") - по сути дела, драйвера одной определенной *файловой системы*. Процедура загрузки при этом выглядит так. *Первичный загрузчик* загружает полуторный по записанной в него *карте размещения*. Эта *карта* может быть очень простой, так как обычно полуторный *загрузчик* размещается непосредственно после *первичного* в нескольких секторах4 подряд, или в ином специально отведенном месте **вне** *файловой системы*. Полуторный *загрузчик* умеет распознавать **одну** *файловую систему* и находить там *вторичный* уже по **имени** (обычно /boot/*grub*/stage2 ). Наконец, *вторичный загрузчик*, пользуясь возможностями полуторного, читает из файла /boot/*grub*/menu.lst меню, в котором пользователь может выбирать варианты загрузки так же, как и в *LILO*. Таким образом, обновление и перенастройка установленного *GRUB* **не требует** пересчета *карт размещения* и изменения чего-то, кроме файлов в каталоге /boot/*grub*.

По требованию Мефодия Гуревич установил на двухсистемную машину *GRUB*. При этом файл /boot/*grub*/menu.lst получился таким:

[root@localhost root]# cat /boot/grub/menu.lst

default 0

timeout 50

title linux-up

kernel (hd0,4)/boot/vmlinuz-up root=/dev/hda5

initrd (hd0,4)/boot/initrd-up.img

title failsafe

kernel (hd0,4)/boot/vmlinuz-up root=/dev/hda5 failsafe noapic nolapic

acpi=off

initrd (hd0,4)/boot/initrd-up.img

title floppy

root (fd0)

chainloader +1

title dos

root (hd0,0)

chainloader +1

Пример 10.3. Настройка GRUB на двухсистемной машине

Разница между *lilo*.conf только в синтаксисе, да еще в том, что жесткие диски и разделы на них *GRUB* именует по-своему, в виде ( hdномер\_диска, номер\_раздела ), причем нумеровать начинает с нуля. Метки (" title ") тоже нумеруются с нуля, так что запись default 0 означает, что по истечении тайм-аута будет загружена самая первая конфигурация (по имени " linux-up ").

Изучая руководство по *GRUB*, Мефодий обнаружил гораздо более важное отличие от *LILO*. Оказывается, в *GRUB* не только параметры, но и сами файлы ( *ядро*, *стартовый виртуальный диск* и т.п.) распознаются и **загружаются** в процессе работы. Вместо пунктов меню можно выбрать *режим командной строки*, подозрительно похожий на bash, в котором можно заставить *GRUB* загрузить какое-нибудь другое, не предписанное конфигурацией, *ядро*, посмотреть содержимое каталогов *файловой системы*, распознаваемой полуторным *загрузчиком*, и даже содержимое этих файлов, невзирая ни на какие права доступа: система-то еще не загружена. Мало того, можно по-своему перенастроить *загрузчик* и **записать** результаты настройки. Так и не успев насладиться неожиданной свободой, Мефодий в один прекрасный день обнаружил, что выход в командную строку защищен паролем.

**Действия ядра Linux в процессе начальной загрузки**

Итак, *досистемная загрузка* проходит в три этапа.

1. *Загрузчик* из *ПЗУ* определяет, с каких устройств можно грузиться и, возможно, предлагает пользователю выбрать одно из них. Он загружает с выбранного устройства *первичный загрузчик* и передает ему управление.
2. *Первичный загрузчик* определяет (а чаще всего - знает), где находится *вторичный загрузчик* - большая и довольно интеллектуальная программа. Ему это сделать проще, чем программе из *ПЗУ*: во-первых, потому что для каждого устройства *первичный загрузчик* свой, а во-вторых, потому что его можно легко изменять при изменении настроек загружаемой системы. В схеме, предлагаемой *LILO* и *GRUB*, *первичный загрузчик* не вступает в разговоры с пользователем, а немедленно загружает *вторичный* и передает ему управление.
3. *Вторичный загрузчик* достаточно умен, чтобы знать, где находится *ядро* системы (возможно, не одно), предложить пользователю несколько вариантов загрузки на выбор, и даже, в случае *GRUB*, разрешает задавать собственные варианты загрузки. Его задача - загрузить в память *ядро* и все необходимое для старта системы (иногда - модули, иногда - *стартовый виртуальный диск* ), настроить все это и передать управление *ядру*.

*Ядро* - это и мозг, и сердце Linux. Все действия, которые нельзя доверить отдельной подзадаче (процессу) системы, выполняются *ядром*. Доступом к оперативной памяти, сети, дисковым и прочим внешним устройствам заведует *ядро*. *Ядро* запускает и регистрирует процессы, управляет разделением времени между ними. *Ядро* реализует разграничение прав и вообще определяет политику безопасности, обойти которую, не обращаясь к нему, нельзя просто потому, что в Linux больше никто не предоставляет подобных услуг.

*Ядро* работает в специальном режиме, так называемом "режиме супервизора", позволяющем ему иметь доступ сразу ко всей оперативной памяти и аппаратной таблице задач. Процессы запускаются в "режиме пользователя": каждый жестко привязан *ядром* к одной записи таблицы задач, в которой, в числе прочих данных, указано, к какой именно части оперативной памяти этот процесс имеет доступ. *Ядро* постоянно находится в памяти, выполняя ***системные вызовы*** - запросы от процессов на выполнение этих подпрограмм.

**Ядро**. Набор подпрограмм, используемых для организации доступа к ресурсам компьютера, для обеспечения запуска и взаимодействия процессов, для проведения политики безопасности системы и для других действий, которые могут выполняться только в режиме полного доступа (т.н. "режиме супервизора").

Функции *ядра* после того, как ему передано управление, и до того, как оно начнет работать в штатном режиме, выполняя *системные вызовы*, сводятся к следующему.

Сначала *ядро* определяет аппаратное окружение. Одно и то же *ядро* может быть успешно загружено и работать на разных компьютерах одинаковой архитектуры, но с разным набором внешних устройств. Задача *ядра* - определить список внешних устройств, составляющих компьютер, на котором оно оказалось, классифицировать их (определить диски, терминалы, сетевые устройства и т.п.) и, если надо, настроить. При этом на *системную консоль* (обычно первая *виртуальная консоль* Linux) выводятся диагностические сообщения (впоследствии их можно просмотреть утилитой dmesg ).

Затем *ядро* запускает несколько *процессов ядра*. ***Процесс ядра*** - это часть *ядра* Linux, зарегистрированная в *таблице процессов*. Такому процессу можно послать сигнал и вообще пользоваться *средствами межпроцессного взаимодействия*, на него распространяется политика *планировщика задач*, однако никакой задаче в режиме пользователя он не соответствует - это просто еще одна ипостась *ядра* . Команда ps -ef показывает *процессы ядра* в квадратных скобках, кроме того, в Linux принято (но не обязательно), чтобы имена таких процессов начинались на "k": [kswapd], [keventd] и т.п.

Далее *ядро* подключает ( **монтирует** ) *корневую файловую систему* в соответствии с переданными параметрами (в наших примерах - root=/dev/hda5 ). Подключение это происходит в режиме "только для чтения" (read-only): если целостность *файловой системы* нарушена, данный режим позволит, не усугубляя положения, **прочитать** и запустить утилиту fsck ( **f** ile **s** ystem **c** *hec* **k** ). Позже, в процессе загрузки, *корневая файловая система* подключится на запись.

### Наконец, *ядро* запускает из файла /sbin/init первый **настоящий** процесс. Идентификатор процесса ( *PID* ) у него равен единице, он - первый в *таблице процессов*, даже несмотря на то, что **до** него там были зарегистрированы *процессы ядра*. Процесс init - очень старое изобретение, он чуть ли не старше самой **Загрузка системы**

С запуска init начинается *загрузка* самой системы. Во времена молодости Linux и ранее в **этом** месте никаких подводных камней не наблюдалось. Если *ядро* содержало подпрограммы для работы со **всеми** необходимыми устройствами (так называемые "драйверы"), оно загружалось и запускало init. Если *ядру* недоставало каких-то важных драйверов (например, поддержки дискового массива, с которого и шла *загрузка*) - оно не загружалось и не запускало. Из положения выходили просто: в *ядро* старались включить как можно больше драйверов. Такое *ядро* называлось *базовым* (*generic*) и имело довольно внушительный размер. Загрузив систему с *базовым ядром*, *администратор* обычно **пересобирал** его: выбрасывал из специального файла-профиля драйверы всех отсутствующих в системе устройств, быть может, добавлял новые (те, что не нужны для загрузки, но необходимы для работы, например, звуковые) и компилировал из исходных текстов новое, *профильное ядро*.

**Стартовый виртуальный диск и модули ядра**

Пересборка *ядра* в наше время требуется очень редко. Во-первых, в Linux поддерживается **несметное** количество различных внешних устройств, драйверы которых (особенно похожих, но разных) вполне могут помешать друг другу работать, если их использовать одновременно. Пришлось бы собирать множество разных *ядер*, без возможности указать **пользователю**, какое из них подходит для его компьютера. Во-вторых, выяснением того, какой именно драйвер необходим найденному устройству, занимаются сейчас специальные программы, в распоряжении которых есть целые базы данных - *ядру* такую работу выполнять неудобно, да и незачем. Это делает процедуру пересборки *ядра* почти что обязательной (пока не загружено *базовое ядро*, непонятно, какие драйверы добавлять в *профильное* ). А в-третьих, пересборка *ядра* требует весьма высокой квалификации. Этот процесс нельзя ни автоматизировать, ни упростить. Утилита linuxconf, устроенная именно для этого на основе окон и меню, дает на выходе работоспособное *ядро* в трех случаях: (1) в руках профессионала, (2) при четком следовании **полной** инструкции и (3) по случайности5.

Совсем другие времена настали, когда изобрели и активно внедрили в Linux **загружаемые** *модули ядра*. *Модуль ядра* - это часть *ядра* Linux, которую можно добавлять и удалять во время работы системы. *Модуль ядра* - не процесс, он работает в режиме супервизора и в *таблице процессов* не регистрируется: это набор подпрограмм для работы с определенным устройством, которые **добавляются** к возможностям ядра6. При загрузке в память *модуль* компонуется с ядром, образуя с ним одно целое. Просмотреть список загруженных модулей можно командой lsmod, а подгрузить модуль в память, добавив его к *ядру*, и удалить его оттуда - командами insmod и rmmod соответственно.

# lsmod

Module Size Used by Not tainted

usb-uhci 21676 0 (unused)

usbcore 58464 1 [usb-uhci]

af\_packet 12392 1 (autoclean)

pcnet32 15140 1 (autoclean)

mii 2544 0 (autoclean) [pcnet32]

crc32 2880 0 (autoclean) [pcnet32]

floppy 48568 0 (autoclean)

subfs 4296 4 (autoclean)

ac 1792 0

rtc 6236 0 (autoclean)

ext3 62288 2

jbd 37852 2 [ext3]

Пример 10.4. Получение списка загруженных модулей

Изменилось и *базовое ядро*: теперь оно включает в себя только устройства, **необходимые** для загрузки системы: главным образом диски и графическую консоль. Остальные устройства определятся уже самой системой - тогда можно будет и распознать экзотическую аппаратуру, и модуль для нее подгрузить. Однако полностью перевести драйверы всех внешних устройств в модули мешает следующее соображение: что, если загрузка системы происходит **именно** с того устройства, чей модуль еще **не** загружен в *ядро*, например, с дискового массива (RAID)? *Вторичный загрузчик* и *ядро* можно, недолго думая, разместить на **другом** носителе (например, на лазерном диске) или добыть с дискового массива средствами *BIOS* ( *карты размещения* позволяют не обращать внимания на логическую структуру RAID). Но как добыть **модуль** работы с RAID, тот самый, что распознает эту логическую структуру?

**Модуль ядра**. Необязательная часть *ядра*, расширяющая его функциональность. Модуль можно загрузить в память или удалить оттуда в процессе работы системы.

Подсистема загрузки *GRUB* умеет разбираться в *файловых системах* и даже подключать модули к *ядру*, однако для того, чтобы сделать процесс загрузки более или менее универсальным, пришлось бы обучить *GRUB* всем видам логики RAID и всем способам *подключения модулей*. И то, и другое постоянно изменяется, и успевать за этими изменениями означает поддерживать собственную, параллельную Linux, дисковую подсистему.

Вдумаемся. Для того чтобы средствами Linux подключить *модуль ядра*, работающий с дисковым устройством, необходимо загрузить Linux с этого же устройства. Так ли это невозможно? Ведь если можно прочесть оттуда " *ядро* ", то, наверное, можно прочесть и "Linux"? Более точно, вдобавок к **одной** области данных, соответствующей *ядру*, надо прочитать **вторую**, соответствующую некоторой уменьшенной до предела установке Linux, в которой содержатся только нужные программы и модули, загрузить оттуда "маленький Linux", который настроит и подключит злополучный RAID и запустит процесс загрузки полноценной системы оттуда.

Предельно сжатый вариант Linux есть - это проект busybox, используемый во встроенных системах, где дорог каждый байт. Разместить *файловую систему* в памяти тоже легко - этим, например, занимается модуль tmpfs, который можно включить в *базовое ядро* (подробнее о типах *файловых систем* будет рассказано в лекции 11). Осталось только обучить подсистемы загрузки *GRUB* и *LILO* считывать не одну, а две области данных - *ядро* и **образ** *файловой системы*. *Ядру* при этом передается параметр "пользуйся виртуальным диском", чтобы оно подключило загруженный образ в качестве временной *корневой файловой системы*. Можно также потребовать, чтобы память, занимаемая временной *файловой системой*, освобождалась в процессе дальнейшей загрузки.

Такой механизм называется initrd ( **init** ial **r** am **d** isk, где "ram" - это не "баран", а **r** andom **a** ccess **m** emory, то есть оперативная память) или *стартовым виртуальным диском*. *Стартовый виртуальный диск* собирается по команде mkinitrd в соответствии с профилем компьютера и записывается на диск по тем же правилам, что и *ядро*. В примере двухсистемной машины, за которой работал Мефодий, также был *стартовый виртуальный диск*, причем довольно маленький:

[root@localhost root]# ls -lg /boot

drwxr-xr-x 2 root 4096 Nov 20 21:08 grub

-rw------- 1 root 205374 Nov 9 01:33 initrd-2.4.26-std-up.img

lrwxrwxrwx 1 root 29 Nov 9 01:33 initrd-up.img ->

initrd-2.4.26-std-up.img

-rw------- 1 root 45056 Nov 20 19:07 map

-rw-r--r-- 1 root 935892 Aug 3 21:59 vmlinuz-2.4.26-std-up

lrwxrwxrwx 1 root 26 Nov 9 01:33 vmlinuz-up -> vmlinuz-2.4.26-std-up

Пример 10.5. Размеры и наименование файлов с ядром и стартовым виртуальным диском

Как видно из примера, *ядро* в четыре раза превосходит по размеру *стартовый виртуальный диск*. Стоит заметить, что и *ядро*, и образ диска **упакованы** с помощью утилиты gzip (причем *ядро* умеет распаковываться в памяти самостоятельно), поэтому их действительный размер - больше. В файле map хранится *карта размещения* *LILO*, а упомянутые в *lilo*.conf и menu.lst файлы vmlinuz-up и initrd-up.img оказались символьными ссылками на файлы с более "говорящими" именами. Никаких требований к названиям *ядер* в Linux нет, это дело авторов дистрибутива. В этом случае в имени *ядра* и образа диска встречается **версия** *ядра* (2.4.26), тип сборки std (по-видимому, "standard") и тип архитектуры up ( **u** ni **p** rocessor, т.е. однопроцессорная).

**Стартовый виртуальный диск**. Минимальный набор программ и модулей Linux, необходимый для обеспечения загрузки системы. Представляет собой *виртуальную файловую систему* в оперативной памяти. Загружается *вторичным загрузчиком* вместе с *ядром* .

**Отец всех процессов**

Если в параметрах не указано иное, *ядро* считает, что init называется /sbin/init. В *стартовом виртуальном диске* это обычно некоторый простейший сценарий, а в полноценной системе у init другая задача: он запускает все процессы. Если процессы запускает не он сам, то это делают его потомки, так что все процессы Linux, кроме *ядерных*, происходят от init, как весь род людской - от Адама.

Первым делом init разбирает собственный конфигурационный файл - /etc/*inittab*. Файл этот имеет довольно простую структуру: каждая строка (если она не комментарий) имеет вид " id:уровни:действие:процесс ", где id - это некоторая двух- или однобуквенная метка, уровни - это слово, каждая буква которого соответствует *уровню выполнения* (об *уровнях выполнения* будет рассказано далее), действие - это способ запуска процесса. Например, запись 4:2345:respawn:/sbin/mingetty tty4 означает, что меткой " 4 " помечен запуск /sbin/mingetty tty4 7 на *уровнях выполнения* 2, 3, 4 и 5 по алгоритму " respawn " (запустить в фоне, а когда процесс завершится, запустить заново). Помимо " respawn ", существуют методы " once " (запустить в фоне однократно), " wait " (запустить интерактивно, при этом никаких других действий не выполняется, пока процесс не завершится) и множество других, включая даже " ctrlaltdel " - процесс, запускаемый, когда пользователь нажимает на консоли Ctrl+Alt+Del 8.

Наконец-то Мефодий до конца понял, отчего getty ведет себя так непохоже на остальные процессы: не просто запускает из-под себя login, а дожидается окончания его работы, **отсутствуя** при этом в *таблице процессов*. На самом деле дожидается не getty, а init, используя метод " respawn ": порождается (в фоне) процесс getty с определенным *PID*, а init бездействует до тех пор, пока существует процесс с этим *PID*: getty, login, *стартовый командный интерпретатор* или программа, запущенная из него с помощью exec() ; когда же процесс, наконец, умирает, порождается новый getty.

**Запуск системных служб**

Полноценно загруженная Linux-система - не только login на виртуальной консоли. Системе есть чем заняться и помимо идентификации пользователей. Даже если компьютер не работает WWW-, FTP- или почтовым сервером для "внешнего мира", себе самой и своим пользователям система предоставляет множество услуг: отсылка заданий на печать и обеспечение их очереди, запуск заданий по расписанию, проверка целостности и т.п. Набор утилит и системных программ, предназначенных для предоставления таких услуг, принято называть подсистемами или **службами**.

**Чему служат демоны?**

Как правило, *системная служба* организована так. Во время начальной загрузки запускается в фоновом режиме программа, которая на протяжении работы системы находится в *таблице процессов*, однако большей частью бездействует, ожидая, когда ее о чем-нибудь попросят. Для того чтобы попросить эту программу об услуге, которую она предоставляет, используются утилиты, взаимодействующие с ней по специальному протоколу. По аналогии с сократовским "даймонионом", который незримо присутствует, по своей инициативе не делает ничего, не дает совершать плохое и способствует хорошему, такую программу стали называть " *daemon* ". Не знакомые с творчеством Платона программисты-любители частенько переименовывали ее в " *demon* " ( ***демон*** ); к сожалению, именно в такой, слегка инфернальной форме, *daemon* и вошел в русскоязычную терминологию. Выходит, что в Linux услуги пользователям предоставляют... *демоны*!

**Демон**. Запускаемая в фоне программа, длительное время пребывающая в *таблице процессов*. Обычно *демон* активизируется по запросу пользовательской программы, по сетевому запросу или по наступлению какого-либо системного события.

В ранних версиях UNIX **все**, что нужно было запускать при старте системы, вписывалось в *inittab*. Было довольно удобно в одном файле указывать, какие именно *демоны* должны работать в системе, и в каком порядке их запускать. Само поведение *демона* при запуске явно рассчитано на использование в *inittab* по методу " wait ": классический *демон* запускается **интерактивно**, проверяя правильность конфигурационных файлов и прочие условия работы, а затем **самостоятельно** уходит в **фон** (попросту делая fork() и завершая родительский процесс). Таким образом, init ждет, пока *демон* работает интерактивно, а когда служба, возглавляемая этим *демоном*, готова к работе, переходит к следующей строке *inittab*. Однако часто бывает так, что автор *демона* знать не знает, какие именно условия разработчики той или иной версии Linux сочтут пригодными для запуска. Для этого ими создается *стартовый сценарий*, в котором запрограммирована **логика** запуска и *останова* службы. Кроме того, если на каждом из *уровней выполнения* запускается много различных служб, попытка записывать их запуск в *inittab* делает его громоздким и совсем неочевидным.

Гуревич посоветовал Мефодию отложить на время изучение загрузки **системы**, а сначала посмотреть, как управлять *стартовыми сценариями*.

**Стартовый сценарий системной службы**

***Стартовый сценарий*** - программа (обычно написанная на shell), управляющая включением или выключением какого-нибудь свойства системы. Это может быть запуск и остановка HTTP-сервера, активизация и деактивизация сетевых настроек, загрузка модулей и настройка звуковой подсистемы и т.п. Простейший *стартовый сценарий* обязан принимать один параметр, значение которого может быть словом "start" для запуска (включения) и "stop" для остановки (выключения). Если в определенном дистрибутиве Linux принято решение, что *стартовые сценарии* должны понимать и другие параметры, например "restart" (обычно "stop"+"start", но не всегда) и "status" (для опроса состояния), это требование распространяется на **все** *стартовые сценарии*. Единообразие позволяет, например, без труда запускать и останавливать *демоны*, не выясняя, каков *PID* останавливаемого процесса и какой **именно** сигнал ему следует послать. Достаточно **запуск** написать так, чтобы *PID* процесса откладывался в специальный файл (обычно /var/run/имя\_службы ), а в **остановку** вписать что-то вроде kill -правильный\_сигнал `cat /var/run/имя\_службы`.

Все *стартовые сценарии* служб, которыми может воспользоваться система, принято хранить в каталоге /etc/rc.d/init.d (в некоторых дистрибутивах, для совместимости со старыми версиями UNIX, используется /etc/init.d, иногда это просто символьная ссылка на /etc/rc.d/init.d ). Запустить или остановить службу можно, просто вызвав соответствующий сценарий с параметром "start" или "stop". Часто ту же самую задачу выполняет и специальная команда service, которая проверяет, есть ли указанный *стартовый сценарий*, и запускает его9:

[root@localhost root]# lsmod > old

[root@localhost root]# /etc/rc.d/init.d/sound stop

Saving OSS mixer settings: [ DONE ]

Unloading sound module (es1371): [ DONE ]

[root@localhost root]# lsmod > nosound

[root@localhost root]# service sound start

Loading sound module (es1371): [ DONE ]

Loading OSS mixer settings: [ DONE ]

[root@localhost root]# lsmod > new

[root@localhost root]# diff3 old new nosound

====3

1:2,5c

2:2,5c

es1371 25608 0

ac97\_codec 11880 0 [es1371]

soundcore 3652 4 [es1371]

gameport 1628 0 [es1371]

3:1a

Пример 10.6. Перезапуск звуковой подсистемы

Здесь Мефодий сначала остановил, а потом снова активизировал звуковую подсистему. Остановка привела к выгрузке звуковых модулей, а повторный запуск - к загрузке, полностью аналогичной исходной. В этом Мефодий убедился, сравнив с помощью утилиты diff3 **три** списка модулей: old (до остановки звуковой подсистемы), new (после повторного запуска) и nosound (между остановкой и повторным запуском). Файлы old и new одинаковы, а от nosound оба отличаются тем, что со второй строки по пятую содержат названия тех самых *модулей ядра* (в том числе gameport, отвечающий за джойстик).

**Схема ".d"**

Итак, существует способ единообразно и гибко управлять запуском и остановкой **каждой** *системной службы* в отдельности (или включением и выключением **одного** свойства системы). Однако задача целиком организовать загрузку системы, от запуска init до полноценной работы, этим еще не исчерпывается. Первая из возникающих задач такова: чаще всего нужно загружать не **все** из размещенных в /etc/rc.d/init.d сценариев, потому что некоторые из установленных в системе служб администратор решил не использовать. Удалять оттуда не используемые при **загрузке** сценарии - значит, лишать администратора возможности запускать эти сценарии **вручную**.

Создателям ранних версий UNIX пришла в голову простая мысль: написать один большой сценарий по имени /etc/rc, в который и заносить только нужные для запуска команды вида /etc/init.d/сценарий start. Можно даже занести туда все имеющиеся *стартовые сценарии*, но строки, запускающие те, что не используются, закомментировать. Такая (монолитная) схема имела существенный недостаток: добавление и удаление службы в систему (например, добавление и удаление *пакета*, содержащего исполняемые файлы службы) требовало редактирования этого файла. Если учесть, что **порядок** запуска служб весьма важен (например, бессмысленно запускать сетевые *демоны* до активизации сетевых настроек), становится ясно, что **автоматическое** изменение такого файла не может гарантировать нормальную загрузку системы, а значит, недопустимо.

На помощь пришла тактика, известная как "схема .d ". Суть ее в следующем. Пусть некоторый процесс управляется конфигурационным файлом, содержимое которого зависит от наличия и активации других служб системы (таким процессом может быть *демон* централизованного журналирования syslogd, ведущий журнал всех событий системы, или сетевой *метадемон* inetd, принимающий сетевые запросы и превращающий сетевой поток данных в обыкновенный посимвольный ввод-вывод). Тогда, чтобы избежать постоянного редактирования конфигурационного файла, его превращают в **каталог** (например, вдобавок к файлу /etc/syslog.conf заводится каталог /etc/syslog.d ). Каждый файл в таком каталоге соответствует настройке одной службы: при добавлении ее в систему файл появляется, при удалении - исчезает. Остается только обучить тот же syslog читать настройки не только из одного syslog.conf, но и из всех файлов в syslog.d, и задача решена.

На случай **запускаемых сценариев** схема " .d " распространяется с двумя дополнениями. Во-первых, как уже было сказано, *стартовые сценарии* можно запускать, а можно и не запускать. Поэтому сам init.d на роль " .d "-каталога не годится. Впрочем, достаточно организовать еще один каталог, скажем, rc.d, и создать там **ссылки** (для наглядности лучше символьные) на те сценарии из init.d, которые планируется запускать при старте системы. Общий *стартовый сценарий* rc как раз и будет заниматься запуском *стартовых сценариев* из rc.d.

Во-вторых, необходимо обеспечить строгий порядок запуска этих сценариев. Теоретически это совсем просто: отсортировать их по алфавиту, как это делает ls, и запускать подряд. Практически же такое требование накладывает ограничение на имя ссылки в " .d "-каталоге, поэтому принято, чтобы в **начале** имени стояло **двузначное число**. Тогда, запуская подряд все сценарии, отсортированные по алфавиту, rc будет в первую очередь руководствоваться этим номером, а уж потом - названием службы, которое после него стоит.

**Уровни выполнения**

В Linux схема **начальной загрузки** слегка сложнее, чем обычная " .d ". Связано это с тем, что одну и ту же систему в разных случаях бывает необходимо загружать с разным набором служб. Если, скажем, использование сети нежелательно, удобнее сказать что-то вроде "Система! Загружайся без сети!", чем вручную удалять *стартовые сценарии* всех предположительно сетевых служб из " .d "-каталога. Необходимость выбора также возникает, если компьютер используется в качестве рабочей станции с запуском графической среды и всего с нею связанного или в качестве стоечного сервера, управлять которым лучше с *системной консоли*.

Поэтому в Linux предусмотрено несколько вариантов начальной загрузки, называемых *уровнями выполнения* (run levels). *Уровни выполнения* нумеруются с 0 до 9:

* *Уровень* 1 соответствует *однопользовательскому режиму* загрузки системы. При загрузке на *уровень* 1 не запускается никаких служб, и даже *системная консоль*, как правило, бывает доступна только одна, так что в системе может работать не более одного пользователя. В *однопользовательском режиме* изредка работает администратор - исправляет неполадки системы, изменяет ключевые настройки, обслуживает *файловые системы*.
* *Уровень* 2 соответствует *многопользовательскому режиму* загрузки системы с **отключенной сетью**. В этом режиме не запускаются никакие сетевые службы, что, с одной стороны, соответствует строгим требованиям безопасности, а с другой стороны, позволяет запускать службы и настраивать сеть вручную.
* *Уровень* 3 соответствует *многопользовательскому сетевому режиму* загрузки системы. Сеть при загрузке на этот *уровень* настроена, и все необходимые сетевые службы запущены. На этом *уровне* обычно работают компьютеры-серверы.
* *Уровень* 5 соответствует *многопользовательскому графическому режиму* загрузки системы. На этом *уровне* обычно функционируют рабочие станции, предоставляя пользователям возможность работать с *графической подсистемой* X11. Сеть на этом *уровне* настроена, а вот список запущенных сетевых служб может быть меньше, так как рабочая станция не всегда выполняет серверные функции (хотя, безусловно, может).
* *Уровни* 0 и 6 - специальные. Они соответствуют *остановке* и *перезагрузке* системы. В сущности, это удобные упрощения для действий, обратных загрузке на *уровень*: все службы останавливаются, диски размонтируются. В случае *остановки* даже электропитание можно отключать программно, если аппаратура позволяет, а в случае *перезагрузки* система идет на повторную загрузку.

Остальные *уровни* никак специально в Linux не описаны, однако администратор может использовать и их, определяя особый профиль работы системы. Переход с *уровня* на *уровень* выполняется очень просто: по команде init номер\_уровня. На какой *уровень* загружаться при старте системы, написано в *inittab* (в поле действия должно быть написано " initdefault ", а в поле *уровни* - только одна цифра). Узнать текущий *уровень выполнения* можно с помощью команды *runlevel*:

[root@localhost root]# grep initdefault /etc/inittab

id:3:initdefault:

[root@localhost root]# runlevel

N 3

Пример 10.7. Задание и просмотр уровня выполнения

**Уровень выполнения**. Сохраненный профиль загрузки системы. В Linux реализован выполнением всех сценариев остановки и запуска служб из подкаталога rc.d каталога /etc или /etc/rc.d

Схема " .d " легко учитывает *уровни выполнения*. В каталоге /etc/rc.d 10 заводится несколько " .d "-подкаталогов, соответствующих каждому *уровню выполнения*: /etc/rc.d/rcуровень.d. Именно оттуда их запускает *стартовый сценарий* /etc/rc.d/rc:

[root@localhost root]# ls -F /etc/rc.d

init.d/ rc.powerfail\* rc0.d/ rc2.d/ rc4.d/ rc6.d/

rc\* rc.sysinit\* rc1.d/ rc3.d/ rc5.d/ scripts/

[root@localhost root]# ls /etc/rc2.d

K10power K75netfs S15random S31klogd S37gpm S54sshd

K44rawdevices K95kudzu S30sound S32hotplug S40crond S98splash

K50xinetd S10network S30syslogd S35keytable S41anacron S99local

[root@localhost root]# ls -l /etc/rc2.d/ K75netfs

lrwxrwxrwx 1 root root 15 Nov 9 01:16 /etc/rc2.d/K75netfs ->

../init.d/netfs

Пример 10.8. Содержимое каталогов /etc/rc.d и /etc/rc.d/rc2.d

Переход с *уровня* на *уровень* должен сопровождаться не только **запуском**, но и **остановкой** служб. Это касается не только *уровней* 0 и 6, но и любых других. Например, при переходе с *уровня* 3 на *уровень* 2 необходимо остановить все сетевые службы. Поэтому схема " .d " была расширена: сначала с параметром " stop " запускаются сценарии, имена которых начинаются на " K " ( **K** ill), а затем, с параметром " start " - те, имена которых начинаются на " S " ( **S** tart). В приведенном примере при переходе на *уровень* 2 останавливается несколько служб, в том числе сетевой *метадемон* ( K50xinetd ) и *монтирование* по сети удаленных *файловых систем* ( K75netfs ). Если при переходе с *уровня* на *уровень* некой службе не требуется менять своего состояния, сценарий не запускается вовсе. Так, при переходе с *уровня* 3 на *уровень* 2 сетевые **настройки** остаются активными, поэтому соответствующий сценарий ( S10network ), скорее всего, запущен не будет.

Долгое время считалось, что определение порядка загрузки - дело системного администратора, поэтому расставлять символьные ссылки в каталогах rc\*.d приходилось вручную. Однако одно из другого не следует: можно предусмотреть и более простой способ наполнения этих каталогов ссылками. Один из способов такой: поставить в *стартовый сценарий* комментарий особого вида, в котором описать, на каких *уровнях* служба должна быть активизирована, и какой по порядку должна быть запущена и остановлена:

[root@localhost root]# grep chkconfig /etc/init.d/netfs

# chkconfig: 345 25 75

[root@localhost root]# chkconfig --list netfs

netfs 0:off 1:off 2:off 3:on 4:on 5:on 6:off

[root@localhost root]# ls /etc/rc.d/rc\*.d/\*netfs

/etc/rc.d/rc0.d/K75netfs /etc/rc.d/rc3.d/S25netfs

/etc/rc.d/rc6.d/K75netfs

/etc/rc.d/rc1.d/K75netfs /etc/rc.d/rc4.d/S25netfs

/etc/rc.d/rc2.d/K75netfs /etc/rc.d/rc5.d/S25netfs

Пример 10.9. Управление порядком выполнения стартовых сценариев

Здесь Мефодий использовал утилиту chkconfig, которая ищет в *стартовом сценарии* комментарий вида chkconfig: уровни вкл выкл, и самостоятельно проставляет ссылки в соответствии с этими полями: во всех каталогах, упомянутых в уровнях соответствующий netfs сценарий имеет вид Sвклnetfs, а во всех остальных - Kвыклnetfs. Эта же утилита позволяет добавлять и удалять службу на каждом *уровне* в отдельности или запрещать ее вовсе.

**Загрузка типичной системы на уровень выполнения 5**

Итак, что же происходит после запуска init?

[root@localhost root]# grep rc /etc/inittab

si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0

l1:1:wait:/etc/rc.d/rc 1

l2:2:wait:/etc/rc.d/rc 2

l3:3:wait:/etc/rc.d/rc 3

l4:4:wait:/etc/rc.d/rc 4

l5:5:wait:/etc/rc.d/rc 5

l6:6:wait:/etc/rc.d/rc 6

[root@localhost root]# grep initdefault /etc/inittab

id:5:initdefault:

Пример 10.10. Стартовые сценарии в /etc/inittab

Метод " sysinit " в *inittab* означает, что процесс запускается во время начальной загрузки системы, **до** перехода на какой-нибудь *уровень выполнения*. Следовательно, первым запускается сценарий /etc/rc.d/rc.sysinit. Он настраивает аппаратуру дисковых массивов, проверяет и монтирует дисковые *файловые системы*, инициализирует область подкачки, межсетевой экран - словом, делает все, без чего дальнейшая полноценная загрузка системы невозможна. Далее из строчки с " initdefault " init узнает, что *уровень выполнения* по умолчанию - пятый ( *многопользовательский графический* ), и выполняет все строки из *inittab*, в поле уровни которых есть 5. В частности, запускается сценарий rc с параметром 5 (l5:5:wait:/etc/rc.d/rc 5), который и выполняет необходимые действия из " .d "-каталога /etc/rc.d/*rc5*.d. Метод запуска rc - " wait ", так что init ждет, пока не выполнятся все *стартовые сценарии*, а потом продолжает разбор *inittab*:

[root@localhost root]# ls /etc/rc.d/rc5.d/

K10acpid S10network S30syslogd S37gpm S50xinetd

K20nfs S13portmap S31klogd S40crond S54sshd

K65apmd S15random S32hotplug S41anacron S56rawdevices

K86nfslock S25netfs S35keytable S44xfs S64power

S05kudzu S30sound S36update\_wms S45dm S98splash

Пример 10.11. Профиль системы на уровне выполнения 5

Мефодий заметил, что сценарий K20nfs (с параметром " stop ") не выполнился: соответствующего сообщения на *системной консоли* не появилось. Беглый просмотр /etc/rc.d/init.d/nfs показал, что этот сценарий предназначен для запуска и остановки **сервера** сетевой *файловой системы* (NFS). Сервер используется на *уровне* 3, а на *уровне* 5 - нет, поэтому при переходе с 3 на 5 его следует останавливать. Поскольку во время начальной загрузки останавливать нечего, сценарий не выполнился.

Из служб, запускаемых именно на пятом *уровне*, примечателен шрифтовый сервер, под номером 44 (the **X** **f** ont **s** erver, xfs ) - программа, у которой графическая подсистема получает шрифты (нередко по сети; тогда такой сервер может быть один на несколько рабочих станций), и *экранный диспетчер* 11, под номером 45 (the **X** **d** isplay **m** anager, xdm ) - программа, регистрирующая пользователя на манер login, с той разницей, что регистрация и запуск графических приложений могут происходить по сети с удаленного компьютера. Тут разрешилась еще одна загадка: вместо обычной виртуальной консоли и login, Мефодий нередко наблюдал окно графической подсистемы с надписью " Login:" и " Password:", а кое-где даже " Логин:", " Пароль:" и портрет самого пользователя! Оказывается, это были различные версии xdm. Дабы не забивать себе голову разрозненными сведениями, Мефодий решил до поры (до лекции 15) не использовать графическую среду и нажал Ctrl+Alt+F1, переключившись в текстовую консоль.

Текстовая консоль на пятом уровне доступна: записи вида 1:2345:respawn:/sbin/mingetty tty1 обычно включают 5 в поле уровни.

**Останов системы**

Как уже говорилось, операция, обратная загрузке системы - *останов* - устроена в Linux как специальный *уровень выполнения*: 0 - если требуется выключить систему, и 6 - если требуется *перезагрузка*. Соответствующие каталоги rc0.d и *rc6*.d будут состоять почти сплошь из ссылок вида K\*, но как *минимум* один *сценарий*, killall, будет запущен с параметром " start ". Этот *сценарий* остановит все процессы, которые не были остановлены K-сценариями: программы пользователей, *демоны*, запущенные администратором вручную, и т.п.

Нечего и говорить, что **отключение** электропитания в разгар работы системы - операция очень рискованная. Даже в самом удачном случае при повторной загрузке rc.sysinit увидит, что *файловые системы* не были **размонтированы**, и станет проверять их *целостность*. В не самом удачном случае эта *целостность* будет нарушена: некоторые открытые на *запись* и не закрытые файлы окажутся в странном, недописанном состоянии, появятся индексные дескрипторы, не связанные ни с каким каталогом и т.п. Как правило, такие ошибки исправляются программой восстановления *файловых систем* fsck: с одной стороны, за счет дополнительных свойств *файловой системы* (*журналирование*, сводящая *вероятность* порчи к минимуму, логически упорядоченная *запись* и т.п.), с другой - за счет некоторых предположений, которые делает сама *утилита* fsck. Однако надеяться на нее нельзя: очень редко, но бывают неразрешимые противоречия в лишенной цельности *файловой системе*, и тогда fsck обращается за помощью к администратору, требуя подтверждения действий (например, для удаления испорченного файла, который **точно** раньше был), или выполняя эти рискованные действия автоматически. В первом случае все время взаимодействия с администратором система будет работать в *однопользовательском режиме*, причем администратору предстоит разбираться с тем, что получилось; а во втором есть нешуточная *вероятность* того, что система испортится, а замечено это будет слишком поздно.

*Останов системы* может занимать **больше** времени, чем *загрузка*: например, процессы, выполняющие *системный вызов* (скажем, чтения с дискеты), не завершаются по сигналу TERM **сразу**, а получив его, могут некоторое время заниматься обработкой (дописыванием в *файл* и т.п.). Остановка службы, особенно сетевой, тоже может длиться долго: например, когда требуется сообщить о закрытии сервиса каждому клиенту. Однако только в этом случае можно быть уверенным, что все процессы завершились нормально, и что после перезагрузки они продолжат нормально работать.

В экстренных случаях (например, когда при сбое электропитания *демон*, обслуживающий устройство бесперебойного снабжения, сообщает, что ресурсы на исходе) безопаснее все-таки **быстро** поостанавливать процессы, чем дожидаться отключения питания на работающей системе. Для этого можно послать всем процессам сначала TERM, а короткое время спустя - KILL. Для обработки таких ситуаций в *inittab* есть методы, начинающиеся со слова " power ", а в /etc/rc.d - специальный *сценарий* rc.powerfail. На самый крайний случай существуют команды halt и reboot с ключом -f, однако их почти мгновенное действие практически эквивалентно внезапному отключению питания, и использовать их не рекомендуется.

Для *останова* или *перезагрузки системы* можно выполнять команды init 0 и init 6. Они вполне справятся с оповещением и остановкой активных **программ**, что займет минуту-две. А вот с **пользователями**, работающими в системе, все сложнее. Как правило, для завершения работы требуется хотя бы минут пять, а лучше - десять. Поэтому вежливые администраторы пользуются утилитой shutdown, которая запускается за несколько минут до времени *перезагрузки*, каждую минуту предупреждая пользователей о грядущем событии, после чего уже запускают init:

[root@localhost root]# shutdown -r +3 "Sorry, we need to reboot"

Broadcast message from root (ttyS0) (Sun Nov 28 14:05:41 2004):

Sorry, we need to reboot

The system is going DOWN to maintenance mode in 3 minutes!

. . .

Broadcast message from root (ttyS0) (Sun Nov 28 14:06:41 2004):

Sorry, we need to reboot

The system is going DOWN to maintenance mode in 2 minutes!

. . .

Broadcast message from root (ttyS0) (Sun Nov 28 14:07:41 2004):

Sorry, we need to reboot

The system is going DOWN to maintenance mode in 1 minute!

. . .

Broadcast message from root (ttyS0) (Sun Nov 28 14:08:41 2004):

Sorry, we need to reboot

The system is going down to maintenance mode NOW!

INIT: Switching to runlevel: 6

. . .

Пример 10.12. Использование shutdown

Остается заметить, что у shutdown есть обязательный *параметр* - время начала *останова* (в примере он равен " +3 ", то есть "через три минуты"), и необязательный - " -r " ( **r** eboot, *перезагрузка* ) или " -h " ( **h** *alt*, *останов* ). Без необязательных параметров выполняется переход на первый *уровень выполнения*, причем запускается *стартовый командный интерпретатор* суперпользователя, а после его завершения система вновь переходит на *уровень выполнения* по умолчанию (используется, например, для профилактических действий в системе). Нажатие Ctrl+Alt+Del или кнопки выключения питания (в системах, где эта кнопка ничего не выключает, а лишь посылает соответствующий аппаратный сигнал) приводит к запуску именно shutdown -r или shutdown -h.

**Лекция 11:**

**Работа с внешними устройствами**

**Представление устройства в системе**

В лекции 10 говорилось о том, что аппаратный профиль компьютера определяется *ядром* на ранних этапах загрузки системы или в процессе подключения *модуля*. Это не означает, что устройство, не распознанное *ядром*, задействовать невозможно. Если неизвестным *ядру* устройством можно управлять по какому-нибудь стандартному протоколу, вполне возможно, что среди пакетов Linux найдется *утилита* или служба, способная с этим устройством работать. Например, *программа* записи на лазерный *диск* cdrecord знает великое множество разнообразных устройств, отвечающих стандарту *SCSI*, в то время как *ядро*, как правило, только позволяет работать с таким устройством как с обычным лазерным приводом (на чтение) и передавать ему различные *SCSI*-команды.

К сожалению, иногда обратное неверно: если производитель создает новое устройство, управлять которым нужно по-новому, а распознается оно как одно из старых, ошибки неизбежны. Многие стандарты внешних устройств предусматривают строгую идентификацию модели, однако хорошего мало и тут: незначительно изменив схемотехнику, производитель меняет и *идентификатор*, и устройство перестает распознаваться до тех пор, пока *автор* соответствующего *модуля* Linux не заметит этого и не добавит новый *идентификатор* в *список* поддерживаемых.

Большинству распознанных устройств, если они должны поддерживать *операции* чтения/записи или хотя бы **управления** ( ioctl(), описанный ниже), соответствует *файл-дырка* в каталоге /dev или одном из его подкаталогов. В зависимости от того, выбрана ли в системе **статическая** или **динамическая** схема именования устройств, *файлов-дырок* в /dev может быть и очень много, и относительно мало. При статической схеме именования то, что *ядро* распознало *внешнее устройство*, никак не соотносится с тем, что в /dev имеется для этого устройства *файл-дырка*:

[root@localhost root]# cat /dev/sdg14

cat: /dev/sdg14: No such device or address

Пример 11.1. Обращение к несуществующему устройству

Здесь Мефодий попытался прочитать что-либо из устройства /dev/sdg14, что соответствует четырнадцатому *разделу SCSI-диска* под номером 7. Такого диска в этой машине, конечно, нет, а *файл-дырка* для него заведен на всякий случай: вдруг появится? Поскольку появиться может **любое** из поддерживаемых Linux устройств, таких файлов "на всякий случай" в системе бывает и десять тысяч, и двадцать. *Файл-дырка* не занимает места на диске, однако использует *индексный дескриптор*, поэтому в *корневой файловой системе*, независимо от ее объема, должен быть изрядный запас *индексных дескрипторов*.

При динамической схеме именования применяется специальная *виртуальная файловая система*, которая либо полностью подменяет каталог /dev, либо располагается в другом каталоге (например, /sys ), имеющем непохожую на /dev иерархическую структуру; в этом случае *файлы-дырки* в /dev заводит специальная служба. Этот способ гораздо удобнее и для человека, который запустил команду ls /dev, и для компьютера (в случае подключения внешних устройств, например, съемных жестких дисков, "на лету"). Однако он требует соблюдать дополнительную логику "привязки" найденного устройства к имени, иногда весьма запутанную из-за той же нечеткой идентификации. Поскольку происходить это должно в самый ответственный момент, при загрузке системы, динамическую схему именования используют с осторожностью.

**Виртуальная файловая система**. Механизм отображения в виде *файловой системы* любых иерархически организованных данных. Существенно упрощает доступ к таким данным, так как позволяет применять обычные операции ввода-вывода: открытие и закрытие файла, чтение и запись и т.п.

**Файлы-дырки и другие типы файлов**

Кое-какие идеи динамического именования устройств присутствуют и в статической схеме. Так, файлы /dev/mouse или /dev/cdrom на самом деле представляют собой символьные ссылки на соответствующие *файлы-дырки*. Если тип мыши или лазерного привода изменится, достаточно изменить эти ссылки и перезапустить соответствующие службы:

[root@localhost root]# ls -l /dev/cdrom /dev/mouse

lrwxrwxrwx 1 root root 8 Nov 20 23:23 /dev/cdrom -> /dev/hdc

lrwxrwxrwx 1 root root 5 Nov 9 01:16 /dev/mouse -> psaux

[root@localhost root]# ls -lL /dev/cdrom /dev/mouse /dev/hda1 /dev/ur\*

/dev/ze\*

brw-r----- 1 root cdrom 22, 0 Jul 26 16:59 /dev/cdrom

brw-rw---- 1 root disk 3, 1 Jul 26 16:59 /dev/hda1

crw------- 1 root root 10, 1 Dec 2 11:58 /dev/mouse

crw-r--r-- 1 root root 1, 9 Nov 28 14:10 /dev/urandom

crw-rw-rw- 1 root root 1, 5 Jul 26 16:59 /dev/zero

Пример 11.2. Идентификация внешних устройств в /dev/

*Файл-дырка* не имеет размера: сколько в него ни записывай, в файл на диске ничего не попадет. Вместо этого *ядро* передает все записанное драйверу, отвечающему за *файл-дырку*, а тот по-своему обрабатывает эти данные. Точно так же работает и чтение из *файла-дырки*: все запрашиваемые данные в нее подсовывает драйвер. Большинство драйверов - дисковые, звуковые, последовательных и параллельных *портов* и т.п. - обращаются за данными к какому-нибудь внешнему устройству или передают их ему. Но есть и такие, которые сами все "выдумывают": это и /dev/null, черная дыра, в которую что угодно можно записать, и все пропадет безвозвратно, и /dev/zero, из которого можно считать сколько угодно нулей (на запись оно ведет себя как /dev/null ), и /dev/urandom, из которого можно считать сколько угодно относительно случайных байтов.

Изучив выдачу команды ls -lL (ключ " -L " заставляет ls выводить информацию не о символьной ссылке, а о файле, на который она указывает), Мефодий обнаружил, что та вместо **размера** *файла-дырки* (который равен нулю) выводит **два** числа. Первое из этих чисел называется *старшим номером устройства* (major *device number*), оно, упрощенно говоря, соответствует **драйверу**, отвечающему за устройство. Второе называется *младшим номером устройства* (minor *device number*), оно соответствует **способу** работы с устройством, а для дисковых носителей - *разделу*. В частности, из примера видно, что устройствами /dev/zero и /dev/urandom занимается один и тот же драйвер со старшим номером 1. При этом часть устройств (по преимуществу - дисковые) имеет тип " b ", а другая часть - " c " (этот тип имеют, например, терминалы). Тип указан в *атрибутах файла* первым символом. Это **блочные** ( **b** lock) устройства, обмен данными с которыми возможен только порциями (блоками) определенного размера, и **символьные** ( **c** haracter) устройства, запись и чтение с которых происходит побайтно. *Блочные устройства*, вдобавок, могут поддерживать команды **прямого доступа** вида "прочитать блок номер такой-то " или "записать данные на диск, начиная с такого-то блока".

Блочные и *символьные устройства* - полноправные объекты *файловой системы*, такие же, как файлы, каталоги и символьные ссылки. Есть еще два типа специальных файлов - *каналы* и *сокеты*. *Канал* -файл (или ***fifo*** ) называют еще ***именованным каналом*** (named pipe): это такой же объект системы, как и тот, что используется командной оболочкой для организации *конвейера* (его называют ***неименованным каналом*** ), разница между ними в том, что у *fifo* есть имя, он зарегистрирован в *файловой системе* . Это - типичный *файл-дырка*, причем дырка двухсторонняя: любая программа может записать в *канал* (если позволяют права доступа) и любая программа может оттуда прочитать. Создать *именованный канал* можно с помощью команды *mkfifo*:

methody@localhost:~ $ mkfifo hole

methody@localhost:~ $ ( date >> hole & head -1 < hole ) 2> /dev/null

Птн Дек 3 15:11:05 MSK 2004

methody@localhost:~ $ ( cal >> hole & head -1 < hole ) 2> /dev/null

Декабря 2004

methody@localhost:~ $ rm hole

Пример 11.3. Использование именованного канала

Здесь важно, что утилита head показывает начало не "файла" *hole*, а именно последней **записываемой** порции данных, как и подобает трубе1.

**Канал**. Объект Linux, используемый в межпроцессном взаимодействии. Доступен в виде двух *дескрипторов*: один открыт на запись, другой - на чтение. Все данные, записываемые в первый *дескриптор*, немедленно можно прочитать из второго. Различают *неименованный канал*, уничтожаемый с закрытием обоих *дескрипторов*, и *именованный канал* ( *FIFO* ) - *файл-дырку*, создаваемый в *файловой системе* .

Что же касается *сокетов*, то это - более сложные объекты, предназначенные для **связи** двух процессов и передачи информации в **обе** стороны. *Сокет* можно представить в виде двух *каналов* (один "туда", другой "обратно"), однако стандартные файловые операции открытия/чтения/записи на нем не работают. Процесс, открывший *сокет*, считается **сервером**: он постоянно "слушает", нет ли в нем новых данных, а когда те появляются, считывает их, обрабатывает, и, возможно, записывает в *сокет* ответ. Процесс- **клиент** может **подключиться** к *сокету*, обменяться информацией с процессом-сервером и отключиться. Точно так же можно передавать данные и по сети - в этом случае указывается не *путь* к *сокету* на *файловой системе* (так называемый unix domain socket), а сетевой адрес и *порт* удаленного компьютера (например internet socket, если подключаться с помощью сети Internet).

**Драйверы устройств**

Как уже говорилось в лекции 10, часть системы, отвечающая за взаимодействие с каким-нибудь внешним устройством и называемая "драйвер", в Linux либо входит в *ядро*, либо оформляется в виде *модуля ядра* и подгружается по необходимости. Следовательно, *файл-дырка*, обращение к которому приводило к "no such device or address", вполне может и заработать (в этом одна из причин огромного количества устройств в /dev ). Гуревич наотрез отказался объяснять Мефодию, как добавить новый драйвер, до тех пор, пока тот не будет лучше разбираться в архитектуре компьютеров вообще и в аппаратной части IBM-совместимых компьютеров в частности. Поэтому все, что смог понять Мефодий, не имея таких знаний, сводилось к следующему. Во-первых, если существуют различия между тем, как по умолчанию загружает *модули* система и тем, как **на самом** деле это необходимо делать, различия должны быть описаны в файле /etc/modules.conf. Во-вторых, после изменения этого файла, добавления нового устройства, обновления самих *модулей* и т.п. следует запускать утилиту depmod, которая заново выстраивает непротиворечивую последовательность загрузки *модулей*. В-третьих, интересно (но в отсутствие знаний - малопознавательно) запускать утилиту lspci, которая показывает список устройств (распознаваемых по стандарту PCI), найденных на компьютере.

**Работа с устройствами**

Все *файлы-дырки* подчиняются одним и тем же правилам работы с файлами: их можно открывать для записи или чтения, записывать данные или считывать их стандартными средствами, а по окончании работы - закрывать. Открытие и закрытие файла ( *системные вызовы* open() и close() ) в командном интерпретаторе не представлены отдельной операцией, а выполняются автоматически при перенаправлении ввода (открытия на чтение) или вывода (на запись). Это позволяет работать и с устройствами, и с *каналами*, и с файлами совершенно одинаково, что активно используется в Linux программами- *фильтрами*. Каждый тип файлов имеет свою специфику, например, при записи на блочное устройство данные накапливаются *ядром* в специальном буфере размером в один блок, и только после заполнения буфера записываются. Если при **закрытии** файла буфер неполон, он все равно передается целиком: часть - данные, записанные пользователем, часть - данные, оставшиеся от **предыдущей** операции записи). Это, конечно, не означает, что из **файла**, находящегося на блочном устройстве, легко по ошибке прочитать такой "мусор": длина файла известна, и *ядро* само следит за тем, чтобы программа не прочла лишнего.

Даже такие, казалось бы, простые устройства, как жесткие диски, поддерживают гораздо больше различных операций, чем просто чтение или запись. Пользователю, как минимум, может потребоваться узнать размер блока (для разных типов дисков он разный) или объем всего диска в блоках. Для многих устройств собственно передача данных - лишь итог замысловатого общения с управляющей программой или *ядром*. Скажем, для вывода оцифрованного звука на звуковую карту сначала необходимо настроить параметры звукогенератора: частоту, размер шаблона, количество *каналов*, формат передаваемых данных и многое другое. Для **управления** устройствами существует *системный вызов* ioctl() ( **i** nput- **o** utput **c** on **t** ro **l** ): устройство надо открыть, как файл, а затем использовать эту функцию. У каждого устройства - свой набор команд управления, поэтому в виде отдельной утилиты ioctl() не встречается, а используется неявно другими утилитами, специализирующимися на определенном типе устройств.

**Права доступа к устройствам**

Некоторые устройства просто обязаны быть доступны пользователю на запись и чтение. Например, *виртуальная консоль*, за которой работает Мефодий, доступна пользователю methody на запись и на чтение, именно поэтому командный интерпретатор Мефодия может посылать туда символы и считывать их оттуда. В то же время терминал, за которым работает Гуревич, другому пользователю недоступен, а терминалы, за которыми не работает никто, доступны только суперпользователю:

methody@localhost ~ $ who

methody tty1 Dec 3 16:02 (localhost)

shogun ttyS0 Dec 3 16:03 (localhost)

methody@localhost ~ $ ls -l /dev/tty1 /dev/tty2 /dev/ttyS0

crw--w---- 1 methody tty 4, 1 Дек 3 16:02 /dev/tty1

crw------- 1 root root 4, 2 Дек 3 15:51 /dev/tty2

crw--w---- 1 shogun tty 4, 64 Дек 3 16:03 /dev/ttyS0

methody@localhost:~ $ ls -l /usr/bin/write

-rwx--s--x 1 root tty 8708 Июн 25 14:00 /usr/bin/write

Пример 11.4. Кому принадлежат терминалы?

Права на владение терминалом передаются с помощью chown пользователю программой login после успешной регистрации в системе. Она же выставляет право записи на **терминал** членам группы tty. "Настоящих" пользователей в этой группе может и не быть, зато есть *setGID*-программы, например, write, которая умеет выводить сообщения сразу на все активные терминалы.

Множество устройств в системе, используемой как рабочая станция, также отдаются во владение - на этот раз **первому** пользователю, зарегистрировавшемуся в системе. Предполагается, что компьютер служит рабочей станцией именно этого пользователя, а все последующие доступа к этим устройствам не получат2. Как правило, так поступают с устройствами, которые могут понадобиться только одному человеку, сидящему за монитором: звуковыми и видеокартами, лазерными приводами, дисководом и т.п.:

shogun@localhost ~ $ ls -l /dev | grep methody | wc 665 6649 41459

shogun@localhost ~ $ ls -lL /dev/{audio,cdrom,fd0,hda,kmem}

crw-rw---- 1 methody audio 14, 4 Июл 26 16:59 /dev/audio

brw-r----- 1 methody cdrom 22, 0 Июл 26 16:59 /dev/cdrom

brw-rw---- 1 methody floppy 2, 0 Июл 26 16:59 /dev/fd0

brw-rw---- 1 root disk 3, 0 Июл 26 16:59 /dev/hda

crw-r----- 1 root kmem 1, 2 Июл 26 16:59 /dev/kmem

Пример 11.5. Кому принадлежат устройства?

### При этом для того чтобы обеспечить доступ и другим - псевдо- или настоящим - пользователям, такие устройства также принадлежат определенным группам с соответствующими правами. Практика "раздачи" устройств группам вообще очень удобна: даже если доступ к устройству имеет только суперпользователь, существует возможность написать *setGID*-программу, которая, не получая суперпользовательских прав, с**Разметка диска и именование устройств**

В начале лекции говорилось о том, что *младший номер устройства*, соответствующего жесткому диску, обычно указывает на определенный *раздел* этого диска. Поначалу Мефодию казалось, что смысла "пилить" *диск* на несколько *разделов* нет: известно, что один большой *раздел* *файловой системы* Linux3 вмещает чуть больше данных, чем несколько маленьких того же объема. Кроме того, разбивая *диск* на *разделы*, можно не предугадать подходящие размеры этих *разделов*, и тогда *размещение* на них *файловой системы* Linux окажется делом нелегким, если вообще возможным, так как структура дерева каталогов Linux строго определена стандартом *FHS* (см. лекцию 3).

**Раздел диска**. Часть жесткого диска, используемая под определенные задачи: *файловую систему* того или иного типа, *область подкачки* и т.п. Изменение содержимого и типа одного *раздела* никак не сказывается на других.

Впрочем, в том же *FHS* весьма наглядно обоснована необходимость **разнесения** всего дерева каталогов по **разным** *разделам*, каждый из которых будет иметь собственную *файловую систему*. Каталоги сильно различаются по тому, как часто приходится в них записывать, насколько *надежность* хранения данных в них важнее быстродействия и насколько ситуация *переполнения файловой системы* опасна и может помешать работе. Поэтому стоит держать каталог /tmp, требующий очень частой записи, но не требующий надежного хранения данных после перезагрузки, не на том же *разделе*, что и *корневую файловую систему*, *запись* в которую происходит редко (в каталог /etc ), но требует повышенной надежности. В отдельный *раздел* можно поместить весь каталог /usr, так как он **вообще** не требует операций записи. Наконец, такие каталоги, как /var или /home, суммарный объем файлов в которых с трудом поддается контролю со стороны системы, тоже не следует размещать на том же *разделе*, что и *корневую файловую систему*, *переполнение* которой может быть болезненно воспринято Linux.

К тому же на компьютере может быть установлено **несколько** операционных систем, и каждой из них понадобится для *корневой файловой системы* отдельный *раздел*. В примерах этой и предыдущей лекций Мефодий работает именно за такой машиной: помимо Linux, на ней установлен FreeDOS для запуска одной-единственной программы.

**Разметка диска IBM-совместимого компьютера**

**Таблица разделов, таблица разбиения диска, HDPT**. Небольшая часть жесткого диска, описывающая геометрию и тип его *разделов*. Стандартная *таблица разделов* диска IBM-совместимого компьютера может содержать не более четырех *разделов* .

Разбиение диска на *разделы* - дело (теоретически) несложное: какая-то часть диска должна быть отведена под *таблицу разделов*, в которой и будет написано, как разбит диск. Стандартная *таблица разделов* для диска IBM-совместимого компьютера - **HDPT** ( **h** ard **d** isk **p** artition **t** able) - располагается в конце самого первого сектора диска, после *предзагрузчика* ( **m** aster **b** *oot* **r** ecord, **MBR** ) и состоит из четырех записей вида " тип начало конец ", описывающих очередной *раздел* диска (если *раздела* нет, поле тип устанавливается в 0 ). *Разделы*, упомянутые в HDPT **диска**, принято называть ***основными*** (*primary partition*). Устройство Linux, соответствующее первому диску компьютера, обычно называется /dev/*hda* ( **h** ard **d** isk " **a** "). Второй диск получает имя *hdb*, третий - *hdc* и так далее. На типичном IBM-совместимом компьютере такое же имя получит и лазерный накопитель. Часто бывает, что жесткий диск - первый в системе ( *hda* ), а лазерный накопитель - **третий** ( *hdc* ); второго же вовсе нет. Устройства, соответствующие *основным разделам* диска, называются /dev/hdбукваномер, для первого диска - от hda1 до hda4. Просмотреть список *разделов* можно с помощью команды fdisk -l.

**Совмещение нескольких схем разметки**

На той самой - двухсистемной - машине fdisk обнаружила **пятый**, **шестой** и **седьмой** *разделы*, однако не показала ни третий, ни четвертый:

[root@localhost root]# fdisk -l

Disk /dev/hda: 2147 MB, 2147483648 bytes

128 heads, 63 sectors/track, 520 cylinders

Units = cylinders of 8064 \* 512 = 4128768 bytes

Device Boot Start End Blocks Id System

/dev/hda1 \* 1 25 100768+ 6 FAT16

/dev/hda2 26 520 1995840 5 Extended

/dev/hda5 26 282 1036192+ 83 Linux

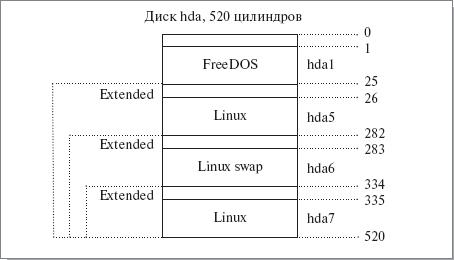
/dev/hda6 283 334 209632+ 82 Linux swap

/dev/hda7 335 520 749920+ 83 Linux

Пример 11.6. Просмотр таблицы разделов жесткого диска

Дело в том, что четырех *разделов* редко когда бывает достаточно. Куда же помещать дополнительные поля *таблицы разбиения*? Создатели IBM PC предложили универсальный способ: один из четырех *основных разделов* объявляется *расширенным* (*extended partition*); он, как правило, занимает все оставшееся пространство диска. *Расширенный раздел* разбивается на подразделы по тем же правилам, что и весь диск: в самом его начале заводится HDPT с четырьмя записями (соответствующие им *разделы* называются *дополнительными* (secondary partition), которые снова можно использовать, причем один из подразделов может быть, опять-таки, *расширенным*, со своими подразделами и т.д. 4

Чтобы не усложнять эту схему, при разметке диска соблюдают два правила: во-первых, *расширенных разделов* в *таблице разбиения* **диска** может быть не более одного, а во-вторых, *таблица разбиения* *расширенного раздела* может содержать либо одну запись - описание *дополнительного раздела*, либо две - описание *дополнительного раздела* и описание вложенного *расширенного раздела*. Соблюдение этого правила позволяет в Linux нумеровать *разделы* **линейно**: после четырех основных номер 5 получает *дополнительный раздел* в первом *расширенном*, 6 - *раздел* во втором *расширенном*, вложенным в первый, и т. п. Сами вложенные *расширенные разделы* при этом не нумеруются и никакому устройству в /dev/ не соответствуют. В действительности разбиение диска двухсистемной машины Мефодия выглядит как на рис.11.1.



**Рис. 11.1.**Разбиение диска двухсистемного компьютера

И *разделы*, и *таблицы разбиения* принято размещать с начала **цилиндра** (термин, имеющий отношение к внутреннему устройству жесткого диска), так что при заведении каждого *расширенного раздела* на этом компьютере тратилось впустую по четыре мегабайта (таков, по сведениям fdisk, размер цилиндра).

Той же тактикой - разбиением не диска, а *раздела* - пользуются, когда *таблица разбиения* нестандартна для IBM PC. Например, UNIX-подобные системы семейства BSD используют собственный универсальный формат разбиения (он старше, чем сама идея об IBM PC!), для чего подобной системе выделяется **один** *раздел*, и она творит с ним все, что заблагорассудится.

**Область подкачки**

Итак, Linux на компьютере из примера использует три *раздела*: hda5, hda6 и hda7. Тип *раздела* hda6, " Linux swap ", отличается от двух других - по словам Гуревича, "это вообще не *файловая система* ". Это - так называемая ***область подкачки*** (*swap space*), пространство на диске, используемое системой для организации *виртуальной памяти* . Оказывается, областям оперативной памяти, которые запрашивают процессы у *ядра*, не всегда соответствуют части физической оперативной памяти. Если процесс долгое время не использует заказанную оперативную память, ее содержимое записывается на диск, в *область подкачки* - тем самым освобождается место в физической памяти для других процессов. Когда же он "вспомнит" об этой области памяти, *ядро* **подкачает** ее с диска, разместит в оперативной памяти (возможно, откачав другие области), и только тогда позволит процессу продолжить работу.

Вполне может сложиться ситуация, когда несколько процессов заказали оперативной памяти больше, чем ее есть в действительности, и преспокойно работают, потому что не используют все заказанное пространство сразу, позволяя системе откачивать неиспользуемые области. К тому же многие процессы (особенно демоны) не работают постоянно, а ждут наступления определенного события, и чем дольше они ждут, тем дольше не используют оперативную память, и тем выше вероятность, что *ядро* откачает ее.

Следует отдавать себе отчет в том, что если эти процессы **вдруг** захотят работать одновременно и со всеми областями памяти, *ядру* придется туго. Большую часть времени система будет проводить, откачивая и подкачивая данные, потому что дисковые операции чтения и записи работают в тысячи раз медленнее, чем запись и чтение из оперативной памяти5. Чтобы хоть как-то облегчить *ядру* задачу, *область подкачки* размещают на отдельном *разделе*, обмен данными с которым работает быстрее, чем чтение и запись в файл, обслуживаемые *файловой системой*.

### **Файловая система**

Из лекции 3 Мефодий узнал, как пользоваться *файловой системой* и какую структуру она имеет с точки зрения программы, работающей с файлами в ней. О том, как огранизована *файловая система* **изнутри**, написан ряд больших статей, защищено немало кандидатских (и несколько докторских) диссертаций. Разработка *файловой системы* - сложный и интересный процесс, требующий одновременно владения высшей математикой, статистикой, умения безошибочно программировать и досконального знания того, как работает то или иное дисковое устройство. Поэтому *файловых систем* не так много, и каждая из них устроена по-своему, в соответствии с тем, как решал тот или иной коллектив разработчиков задачу быстрого и надежного доступа к файлам.

**Принципы организации данных на диске**

Во всех *файловых системах* есть немало общего. Например, в каждой из них решается вопрос **метаданных**, то есть информации, не имеющей прямого отношения к содержимому, допустим, файла, но описывающей, как до этого содержимого добраться. В *файловой системе* обычно различается ***системная область***, в которой записываются метаданные, и *область данных*, где хранятся собственно файлы. *Системная область* может составлять заметную долю общего дискового пространства, и вот почему.

Различают *устройство последовательного доступа* (например, накопители на магнитных лентах) и *устройства прямого доступа* (например, жесткие диски). Чтение (и запись) данных на *устройства последовательного доступа* идет последовательно: если сейчас записан **первый** блок носителя, то следующим будет доступен **второй**, за ним - **третий** и т.д. Если доступен пятый блок, а нужен первый или тысячный, выполняется длительная операция позиционирования, причем она тем длиннее, чем дальше отстоит нужный блок от текущего: лента перематывается. Работать с *устройствами прямого доступа* легче: каков бы ни был текущий прочитанный блок, время, за которое будет прочитан **любой другой**, примерно одинаковое.

Файлы на магнитной ленте удобнее хранить целиком, каждый файл - одним длинным куском. У такого способа есть один существенный недостаток: если на ленту объемом в один гигабайт записать 1024 мегабайтных файла, а потом удалить **каждый второй**, то образуется полгигабайта свободного места, но кусочками по мегабайту каждый. Тогда запись, скажем, двухмегабайтного файла потребует трех операций: сначала надо переписать какой-нибудь мегабайтный файл на свободное место, затем удалить старую его копию, и только затем записать на образовавшееся место большой файл.

На *устройстве прямого доступа* можно избежать этой неприятной ситуации, если постановить, что файл может размещаться на нем в *области данных* **по частям**, а *карта размещения* этих частей будет записана в *системную область*. Если, не мудрствуя, предположить, что в *системную область* записываются номера полукилобайтных секторов, в которых лежит файл (по 32 бита каждый номер), то выходит, что размер *системной области*, который **может** потребоваться, всего в 16 раз меньше файловой. Но в Linux в *системную область* записываются *индексные дескрипторы*, размер которых существенно больше. Количество *индексных дескрипторов* может быть намного меньше количества блоков, но все же *системная область* занимает примерно такую же (от пяти до десяти процентов) долю общего дискового пространства.

**Индексный дескриптор, inode**. Внутренний объект *файловой системы* Linux, однозначно определяющий принадлежащий ей файл. *Индексный дескриптор* содержит *атрибуты файла*, размер, указывает расположение файла на диске и т.п. Каждому *индексному дескриптору* соответствует единственный в данной *файловой системе* идентификатор - целое число.

На самом деле, даже на жестком диске блоки, расположенные подряд, считываются (и записываются) быстрее, чем блоки, расположенные как попало. Эффект связан с механическим устройством жестких дисков, пояснять которое Мефодию Гуревич не стал, ссылаясь на общеизвестность. Суть его в том, что задержки при чтении данных, находящихся на разных цилиндрах диска, растут линейно, как для ленты (чем дальше, тем дольше). Один из остроумных способов оптимизировать работу с диском состоит в том, чтобы разбить все цилиндры на **группы**, а внутри каждой группы выделить свою *системную область* и *область данных*. Тогда сами файлы и их *индексные дескрипторы* будут лежать, если это возможно, на соседних цилиндрах, и доступ ускорится.

Другое, более общее решение - использование *кеширования*, при котором данные с диска частично дублируются в памяти. Если какой-то процесс прочитал данные из файла, эти данные некоторое время находятся в памяти, на случай, если они ему (или кому-нибудь другому) опять понадобятся. Повторное обращение уже не дойдет до диска, система вернет процессу данные из *кеша*, раз уж они ничем не отличаются от тех, что на диске. Если процесс **записал** данные на диск, содержимое *кеша* обновляется, оставаясь актуальным.

Еще эффективней *кеш* на **запись**: операции записи накапливаются в памяти, а до диска добираются не сразу, и в том порядке, в каком быстрее пройдет запись, а не в том, в каком были выполнены. Если запись шла во временный файл, который, в конце концов, удалили, обращений к диску может вообще не случиться. Однако с *кешированием* операций записи следует обращаться бережно: а вдруг сбой в электропитании произойдет именно тогда, когда часть данных уже записана, а часть - еще нет? А если не полностью, кусочками, обновилась *системная область*, состояние *файловой системы* после того, как питание опять включат, может оказаться совсем плачевным - настолько, что даже утилита восстановления fsck может оказаться бессильной. Поэтому *системные области* либо вообще не *кешируются* на запись, либо делают это исключительно с помощью будущих кандидатов и докторов наук, которые рассчитывают безопасные алгоритмы обновления *файловой системы* из *кеша* на запись...

**Работа с файловыми системами**

Итак, Linux свободно работает (и даже предпочитает работать) с несколькими *разделами диска*, содержащими, возможно, разные типы *файловых систем*.

**Монтирование и размонтирование**

В лекции 3 было рассказано о том, что *файловые системы* на различных *разделах* "прививаются" в виде ветвей общего дерева каталогов, растущего из " / ". Делается это при помощи команды mount -o настройки\_монтирования устройство точка\_монтирования, где устройство - это имя блочного *файла-дырки*, точка\_монтирования (mountpoint) - полный *путь* к каталогу, а настройки\_монтирования определяют особые параметры, разные для разных *файловых систем*. После выполнения этой команды содержимое *файловой системы*, размещенной на устройстве (как правило, *дисковом разделе* ), становится доступным в виде дерева подкаталогов точки\_монтирования. Посмотреть список всех смонтированных *файловых систем* можно с помощью команды mount без параметров:

root@localhost root]# mount

/dev/hda5 on / type ext3 (rw)

/dev/hda7 on /home type ext3 (rw)

/dev/fd0 on /mnt/floppy type subfs (rw,nosuid,nodev,sync)

/dev/hdc on /mnt/cdrom type subfs (ro,nosuid,nodev)

proc on /proc type proc (rw,gid=19)

devpts on /dev/pts type devpts (rw,gid=5,mode=0620)

[root@localhost root]# umount /home

[root@localhost root]# ls /home

[root@localhost root]# mount /dev/hda7 /home

[root@localhost root]# ls /home

methody shogun tmpuser

Пример 11.7. Просмотр списка смонтированных файловых систем

Оба Linux- *раздела* смонтированы при старте системы: /dev/hda5 образует *корневую файловую систему*, а /dev/hda7 используется для хранения пользовательских домашних каталогов6. Суперпользователь может размонтировать *файловую систему* вручную с помощью команды umount точка\_монтирования, если на ней не открыто никаких файлов и никто не использует какой-либо ее каталог в качестве текущего.

Для того чтобы *файловые системы* монтировались при старте, их описывают в файле /etc/fstab:

/dev/hda5 / ext3 defaults 1 1

devpts /dev/pts devpts gid=5,mode=0620 0 0

/dev/hda7 /home ext3 defaults 1 2

proc /proc proc gid=19 0 0

/dev/hda6 swap swap defaults 0 0

/dev/fd0 /mnt/floppy subfs fs=floppyfss,sync,nodev,nosuid

/dev/cdrom /mnt/cdrom subfs fs=cdfss,nodev,nosuid

Пример 11.8. Содержимое /etc/fstab

Первое поле каждой строки этого файла - устройство или название *виртуальной файловой системы*, второе - точка монтирования, третье - **тип** *файловой системы*, четвертое - настройки монтирования, а пятое и шестое относятся к организации резервного копирования и процедуре проверки цельности. Содержимое fstab практически повторяет выдачу mount (dev/cdrom на этой машине - ссылка на /dev/*hdc*). Здесь указывается и *область подкачки*, которую *ядро* не монтирует, а использует напрямую. Утилита mount поддерживает **усеченный** вариант командной строки mount точка\_монтирования, при котором она самостоятельно ищет в /etc/fstab, каким способом должна быть смонтирована точка\_монтирования. Для того чтобы при старте системы какое-либо устройство не монтировалось, а усеченным mount его можно было смонтировать вручную, в поле "настройки монтирования" добавляется ключевое слово noauto.

Две последних строки относятся к монтированию **съемных** (removable) носителей: лазерного и гибкого дисков. Съемные носители приходится монтировать гораздо чаще несъемных - не во время старта системы, а всякий раз, когда носитель сменился, и содержимое нового необходимо пользователю. Мало того, надо разрешить выполнять операцию mount **пользователю**, который принес дискету и желает поработать с ней. С другой стороны, нельзя всем и каждому давать право запускать mount и особенно umount с **любыми** параметрами! Существует четыре способа разрешить возникающее противоречие:

1. Воспользоваться усеченным вариантом mount (запись с настройкой noauto в fstab ) и утилитой sudo, при помощи которой позволить пользователю выполнять, скажем, **только** команды mount /cdrom и umount /cdrom.
2. Воспользоваться усеченным вариантом mount и настройкой owner в fstab, которая позволяет выполнять операцию монтирования **хозяину** устройства; при этом /dev/*hdc* отдается во владение первому зарегистрированному пользователю так же, как /dev/audio и прочие устройства персонального использования. Этот способ лучше предыдущего тем, что исключает ситуацию, когда один пользователь смонтирует диск, а другой немедленно его размонтирует.
3. Воспользоваться специальным **демоном** из пакета autofs, который отслеживает обращения пользователей к некоторому каталогу (например, /mnt/cdrom/auto ) и самостоятельно выполняет операцию mount, а если к содержимому носителя долгое время никто не обращался - umount. Этот способ лучше предыдущего тем, что пользователю вообще никаких дополнительных команд подавать не приходится.
4. Воспользоваться специализированным *модулем ядра* (в примере - subfs ), который всегда сообщает программе пользователя, что устройство смонтировано и готово к работе, а поменялся ли носитель, разбирается самостоятельно. Этот способ лучше предыдущего тем, что пользователю не приходится ждать, пока система соизволит размонтировать и "отдать" лазерный диск. Кроме того, subfs может снабжать пользователя данными из *кеша*, даже если диск давно уже вынут (речь идет, разумеется, об операциях чтения).

Во всех случаях, когда диск монтирует **не** системный администратор, стоит предпринять некоторые дополнительные действия. Например, диск должен монтироваться так, чтобы с него не работал запуск с *подменой идентификатора* (*setUID*), и чтобы на нем нельзя было создавать *файлы-дырки* (чтобы не потворствовать хулиганству, вроде запуска *setUID*-оболочки или записи прямо в устройство, соответствующее *hda* ). За это отвечают настройки nosuid и nodev, упомянутые в /etc/fstab.

Кроме того, лазерные приводы имеют "защелку", не позволяющую извлечь диск, пока он используется, а дисководы или устройства USB Flash - нет (хотя, казалось бы, она как раз нужнее там, где происходит запись). Единственная надежда - на то, что пользователь не будет выдергивать дискету из дисковода, пока он занимается записью, и на нем горит зеленая лампочка. Чтобы каждая операция записи немедленно приводила к передаче данных, необходимо полностью отключить *кеш* записи, то есть использовать **синхронный** режим работы *файловой системы*. Это делается при помощи настройки sync.

**Поддерживаемые Linux файловые системы**

Если бы на компьютере из примера использовался способ монтирования лазерного диска 1 или 2, то в поле " тип " fstab было бы написано iso9660. Так называется тип *файловой системы*, обычно используемой на лазерных дисках. Что же касается жестких дисков, то на них может использоваться несколько типов *файловых систем*, даже на одном Linux-компьютере.

Основная *файловая система* в Linux называется "Ext2". Имя происходит от слова "extended" (расширенная) и появилось после того, как самая первая версия *файловой системы* ранних Linux, повторяющая возможности одного из вариантов *файловой системы* UNIX, окончательно устарела. Пришлось переписать соответствующую часть *ядра*, **расширив** уже имеющиеся возможности. Так появилась "ExtFS". Когда и она устарела, возможности снова расширили, и к названию добавилось число "2". Повсеместно используемая в дистрибутивах Linux *файловая система* "Ext3" - "трижды расширенная" - отличается от Ext2 поддержкой **журнализации**.

*Журналируемая файловая система* ведет постоянный учет всех операций записи на диск. Получающийся *журнал обращений* сам, в свою очередь, записывается на диск. Разница между записью *журнала* и записью самих данных в том, что данные следует записывать в строго определенное место, а *журнал* устроен так, чтобы записываться **как можно быстрее**. Выгода от такой двухступенчатой процедуры особенно остро ощущается после сбоя электропитания: все операции, записи, которые еще не успели завершиться, записаны в *журнале*, так что стоит после включения компьютера "проиграть" их еще раз, и *файловая система* войдет в норму! Если часть данных уже была записана на диск, повторная (по требованию *журнала* ) запись **тех же самых** данных на **то же самое** место ничем повредить не может. Наконец, если операция записи не попала даже в *журнал* (что бывает редко), то *файловая система* все равно останется в рабочем состоянии, каким оно было **до** начала этой операции.

Журналирование поддерживается и другими *файловыми системами*, используемыми в Linux - XFS и ReiserFS. ReiserFS вообще похожа скорее на базу данных: внутри нее используется собственная система индексации и быстрого поиска данных, а *индексные дескрипторы* и каталоги в стиле Linux выполнены в виде одной из возможных надстроек над этой системой. Традиционно считается, что ReiserFS отлично подходит для хранения огромного числа маленьких файлов (что было одной из целей проекта), а XFS - для хранения очень больших файлов, в которых постоянно что-нибудь дописывается или изменяется.

В Linux поддерживается, кроме собственных, немало форматов *файловых систем*, используемых другими ОС. Если способ записи на эти *файловые системы* известен и не слишком замысловат, то работает и запись, и чтение, в противном случае - только чтение (чего нередко бывает достаточно). Полностью поддерживаются *файловые системы* FAT12/*FAT16*/FAT32 (тип *vfat* ), используемые в MS-DOS и Windows, ранние версии *UFS*, используемые в системах семейства BSD. Новые версии *UFS* (например, UFS2 из FreeBSD5, обладающая свойствами, которые не входят в стандарт Linux) поддерживаются только на чтение, как и созданная на основе DEC *VMS*, но впоследствии многократно переработанная *файловая система* NTFS из Windows.

Для того чтобы смонтировать *файловую систему*, имеющую заданный тип, команде mount необходимо указать его с помощью ключа " -t ":

[root@localhost root]# fdisk -l

. . .

Device Boot Start End Blocks Id System

/dev/hda1 \* 1 25 100768+ 6 FAT16

/dev/hda2 26 520 1995840 5 Extended

/dev/hda5 26 282 1036192+ 83 Linux

/dev/hda6 283 334 209632+ 82 Linux swap

/dev/hda7 335 520 749920+ 83 Linux

[root@localhost root]# mount -t vfat /dev/hda1 /mnt/disk

[root@localhost root]# ls /mnt/disk

autoexec.bat config.sys fdconfig.sys freedos.bss

command.com fdconfig.old fdos kernel.sys

Пример 11.9. Монтирование файловой системы FAT16

**Виртуальные и сетевые файловые системы**

В /etc/fstab Мефодий сразу заметил две строки, начинающиеся не с имени устройства, а с названия *виртуальной файловой системы*, содержимое которой доступно в соответствующей точке монтирования. *Виртуальная файловая система* обычно не обращается ни к какому внешнему устройству, а "придумывает" все дерево каталогов и находящиеся в них файлы сама. Такова, например, *файловая система* в памяти (ROMFS, или аналогичная ей TMPFS, поддерживающая операции записи), используемая в *стартовом виртуальном диске*. Как правило, *виртуальные файловые системы* используются для того, чтобы предоставить доступ по чтению/записи к некоторой иерархической структуре данных.

Во многих версиях UNIX программа ps работает непосредственно с устройством /dev/kmem (памятью *ядра* ), чтобы добыть оттуда информацию о таблицах процессов; это сложная и даже опасная программа, имеющая доступ к святая святых системы. В Linux ps может быть переписана чуть ли не на shell, потому что таблица процессов и масса другой информации о системе доступны в виде дерева подкаталогов /proc:

[root@localhost root]# ls -F /proc

1/ 585/ 793/ 882/ es1371 irq/ modules stat

1041/ 598/ 794/ acpi/ execdomains kcore mounts@ swaps

16/ 6/ 795/ bus/ fb kmsg mtrr sys/

2/ 681/ 796/ cmdline filesystems ksyms net/ sysrq-trigger

3/ 697/ 797/ cpufreq fs/ loadavg partitions sysvipc/

4/ 7/ 798/ cpuinfo ide/ locks pci tty/

492/ 725/ 8/ devices interrupts mdstat scsi/ uptime

5/ 751/ 840/ dma iomem meminfo self@ version

572/ 784/ 844/ driver/ ioports misc slabinfo

[root@localhost root]# ls -l /proc/1

total 0

-r--r--r-- 1 root proc 0 Dec 4 16:15 cmdline

lrwxrwxrwx 1 root proc 0 Dec 4 16:15 cwd -> /

-r-------- 1 root proc 0 Dec 4 16:15 environ

lrwxrwxrwx 1 root proc 0 Dec 4 16:15 exe -> /sbin/init

dr-x------ 2 root proc 0 Dec 4 16:15 fd

-r--r--r-- 1 root proc 0 Dec 4 16:15 maps

-rw------- 1 root proc 0 Dec 4 16:15 mem

-r--r--r-- 1 root proc 0 Dec 4 16:15 mounts

lrwxrwxrwx 1 root proc 0 Dec 4 16:15 root -> /

-r--r--r-- 1 root proc 0 Dec 4 16:15 stat

-r--r--r-- 1 root proc 0 Dec 4 16:15 statm

-r--r--r-- 1 root proc 0 Dec 4 16:15 status

[root@localhost root]# cat /proc/1/environ ; echo HOME=/TERM=linux

Пример 11.10. Виртуальная файловая система PROCFS

В частности, подкаталоги /proc с числовыми именами содержат информацию о процессах с соответствующими PID. Файл exe такого подкаталога - символьная ссылка на запущенную программу, файл cmdline содержит командную строку, а environ - *окружение* процесса. Мефодий углубился в чтение man proc, руководства по PROCFS, и, как всегда, убедился, что для полного понимания **всего**, что есть в /proc, ему пока не хватает знаний.

*Файловая система* devpts - шаг навстречу динамическому именованию устройств. Она предназначена для *эмуляторов терминала*, таких как sshd, xterm или screen. Задача *эмулятора терминала* - предоставить пользователю полноценный интерфейс командной строки (с запуском командного интерпретатора, с распознаванием и передачей сигналов и т.п.) **в отсутствие** терминального оборудования - по сети или из *графической подсистемы*, или при необходимости сымитировать несколько терминалов. Раньше для этого использовались пары устройств /dev/pty## - /dev/tty##, где ## - двухсимвольный идентификатор. Программа- *эмулятор* начинала обмениваться данными (от пользователя или из сети) с первым свободным устройством (скажем, ptya2, которое, в свою очередь, было привязано к соответствующему терминальному устройству ( ttya2 )). Именно с этим устройством и взаимодействовал командный интерпретатор и прочие процессы Linux, находясь в полной уверенности, что это полноценный терминал.

Выходило, что пар tty## - pty## при статическом именовании устройств могло не хватить, даже если создать их очень много (достаточно запустить еще больше *эмуляторов терминала* ). Поэтому придумали завести **одно** устройство типа pty - /dev/ptmx и *виртуальную файловую систему* /dev/pts для терминальных *файл-дырок*. Каждая программа, открывающая ptmx, получает свой *дескриптор* ), а в pts/ заводится очередное *терминальное устройство*, имя которого совпадает с порядковым номером. Когда *дескриптор* закрывается, *терминальное устройство* исчезает.

Среди *файловых систем* есть и такие, что не выдумывают содержимое сами, а обращаются за ним еще куда-нибудь, например, в сеть. Так работают *удаленные файловые системы*, например, **NFS** ( **n** etwork **f** ile **s** ystem), стандартная для всех UNIX-подобных ОС. Вместо поля "устройство" обычно указывается сетевой адрес компьютера-сервера и имя ресурса (название каталога), который необходимо смонтировать по сети. Поддерживается и работа с *удаленными файловыми системами* Windows, причем как на уровне *модулей ядра*, с помощью mount -t smbfs, так и без монтирования, с помощью утилиты smbclient. Linux и сама может служить сервером, предоставляющим удаленный доступ к файлам, причем служба samba, занимающаяся *файловыми системами* для Windows под управлением Linux, работает зачастую быстрее, чем Windows, запущенный на том же компьютере для тех же целей.

Возможности *файловых систем* этим не исчерпываются. Например, можно смонтировать *образ устройства* из файла, если вызвать команду mount с ключом -o loop. ***Образ устройства*** - это файл, содержимое которого в точности повторяет содержимое устройства; его можно, например, получить с помощью команды cat устройство образ . Именно *образами устройств* манипулируют программы записи на лазерные носители. Смонтировать *образ* бывает нужно для проверки или изменения содержимого перед записью, или для хранения содержимого нескольких дисков в исходном виде:

[root@localhost root]# ls -l floppy.flp

-rw-r--r-- 1 root root 1474560 Dec 4 16:53 floppy.flp

[root@localhost root]# mount -t vfat -o loop floppy.flp /mnt/disk/

[root@localhost root]# ls /mnt/disk/

command.com kernel.sys

[root@localhost root]# mount | grep disk

/root/floppy.flp on /mnt/disk type vfat (rw,loop=/dev/loop0)

Пример 11.11. Монтирование содержимого файла при помощи mount -o loop

Как заметил Мефодий, mount создает для такого способа монтирования специальное устройство - /dev/loop0, и уже с его помощью работает с файлом.

Обширное поле для экспериментов - так называемая **пользовательская** *файловая система* ( **l** inux **u** serland **f** ile **s** ystem, **LUFS** ). Это *модуль ядра* и набор библиотек, позволяющий организовать *файловую систему*, обращающуюся за информацией к обычному процессу Linux. Так организованы разнообразные сетевые "эмуляторы" *файловых систем* с использованием протокола FTP или Secure Shell. Так работает и доступ **на запись** к *файловой системе* NTFS: некоторая программа обращается к **устройству**, содержащему *файловую систему*, задействует драйвер NTFS, взятый из лицензионной копии самой Windows (это можно сделать с помощью библиотек wine, подсистемы, распознающей исполняемые форматы Windows), и обменивается данными с *модулем* LUFS, который и предоставляет обычный файловый доступ процессам.

**Проверка файловой системы**

Если доступная на запись *файловая система* не была размонтирована перед выключением компьютера, после включения она окажется в нештатном состоянии, независимо от того, испортилось на ней что-либо или нет. Проверкой цельности *файловой системы* занимается утилита fsck ( **f** ile **s** ystem **c** *hec* **k** ). На самом деле таких утилит **несколько** - по одной для каждого из основных типов *файловых систем* (есть fsck даже для *VFAT*). Как уже говорилось в лекции 10, fsck запускается при старте Linux, если *файловая система* находится в нештатном состоянии, или для профилактики, если *файловую систему* просто давно не проверяли.

В самом лучшем случае fsck не находит ничего подозрительного, и система продолжает загрузку. Чаще всего, даже если в *файловой системе* не все в порядке, ее *журнал* не испорчен, и fsck "проигрывает" его, после чего все опять приходит в норму. Запустить fsck можно и вручную, в виде fsck устройство или fsck точка\_монтирования, однако прежде следует размонтировать *файловую систему*:

[root@localhost root]# fsck -fy /home

fsck 1.35 (28-Feb-2004)

/dev/hda7 is mounted.

WARNING!!! Running e2fsck on a mounted filesystem may cause

SEVERE filesystem damage.

Do you really want to continue (y/n)? no

check aborted.

[root@localhost root]# umount /home

[root@localhost root]# fsck /home

fsck 1.35 (28-Feb-2004)

e2fsck 1.35 (28-Feb-2004)

/dev/hda7: clean, 168/93888 files, 7269/187480 blocks

[root@localhost root]# fsck -f /home

fsck 1.35 (28-Feb-2004)

Pass 1: Checking inodes, blocks, and sizes

Pass 2: Checking directory structure

Pass 3: Checking directory connectivity

Pass 4: Checking reference counts

Pass 5: Checking group summary information

/dev/hda7: 168/93888 files (0.6% non-contiguous), 7269/187480 blocks

Пример 11.12. Использование fsck

Со второго раза fsck 7 работать тоже отказалась, ссылаясь на то, что *файловая система* и так находится в штатном состоянии (ее аккуратно размонтировали). Пришлось применить ключ " -f " ( **f** orce), который **заставляет** fsck работать - конечно же, никаких ошибок найдено не было. Сама процедура проверки довольно сложна - она состоит из пяти этапов, каждый из которых отнюдь не тривиален и в этой лекции не описывается. Кстати сказать, для того чтобы проверить *корневую файловую систему*, ее приходится сначала монтировать только на чтение, находить там /sbin/fsck, проверять и только после этого монтировать на чтение-запись. Если *корневая файловая система* испорчена настолько, что /sbin/fsck в ней найти невозможно, остается пробовать загрузку с других носителей (например, с установочного CD) и разбираться.

Если какая-то порча *файловой системы* все-таки обнаружилась, fsck может поступить двояко. Во-первых, все ошибки, которые не приводят к **изменению** данных на диске, можно попробовать исправить автоматически. Например, *индексные дескрипторы*, на которые не ссылается ни одно имя (так называемые **потерянные файлы**, unref files), помещаются в специальный каталог /lost+found под именами, соответствующими номерам этих *индексных дескрипторов*. Впоследствии администратор может посмотреть в эти файлы и решить, нужны они или нет. Во-вторых, когда fsck встречается с ошибкой, исправление которой приведет к изменению данных на диске, загрузка Linux останавливается, и система переходит в *однопользовательский режим*. Предполагается, что администратор сам запустит fsck: либо интерактивно (тогда каждому изменению в *файловой системе* будет требоваться подтверждение с клавиатуры), либо пакетно, с ключом " -y " (тогда считается, что на все запросы администратор заранее ответил "yes").

Когда-то такие запуски fsck -y производили катастрофические разрушения по вине неумелых администраторов, а нынче Мефодий, как ни нажимал "Reset" на многострадальной двухсистемной машине, не смог добиться ничего, кроме двух-трех мгновенных воспроизведений *журнала* и жестокого нагоняя от Гуревича.

**Лекция 12:**

**Конфигурационные файлы**

**Проектирование свойств системы**

*Операционная система*, позволяющая задействовать все возможности компьютера, резко отличается от специализированного программного обеспечения огромным числом так называемых "вариантов использования" (use cases) и обширнейшими возможностями тонкой настройки для решения задач конкретного пользователя наилучшим способом. Достаточно сравнить какую-нибудь игровую приставку (например, PlayStation2) под управлением собственной операционной системы и ее же под управлением Linux. Вычислительная и мультимедийная *мощность* такого компьютера весьма высока (известно, что именно компьютерные игры определяют сейчас ресурсопотребление персонального компьютера). Однако способы управления одной и другой системами настолько различны, что неподготовленный человек просто теряется при виде возможностей Linux: на какие кнопки нажимать? А кнопок-то и нет...

Можно попытаться описать операционную систему как большой и сложный универсальный инструмент для решения любых задач. Предполагается, что *пользователь*, прочтя документацию, в которой описывается, как работает система и как применять ее в различных ситуациях, сможет решать и свои задачи. Правда, для этого ему придется прочесть большую часть документации по системе (в том числе и технической) и перепрограммировать некоторые части системы сообразно своим нуждам. На такой подвиг способны немногие, времени это займет немало, да и *вероятность* ошибки (которая тем выше, чем сложнее средства управления системой) при таком подходе недопустимо велика. Сами утилиты или службы Linux, каждую из которых можно "окинуть взором" и понять, что она умеет и чего в ней не хватает, **разрабатываются** именно теми из пользователей, у которых хватает времени, знаний и навыков на такое полное освоение (см. лекции 17 и 18). *Вывод*: *пользователь* – не разработчик, ему все-таки важнее быстро и качественно решить задачу, чем долго совершенствовать инструмент решения.

Можно пойти обратным путем: попытаться предусмотреть все **основные** способы использования операционной системы на всех основных пользовательских задачах, и на каждый такой способ создать (запрограммировать) отдельную часть, управляемую "кнопкой" или утилитой. Эту часть обычно называют "решением" (solution), и в документации пишут, **что** должно быть "на входе" системы, и что получается "на выходе" после применения решения. Если *пользователь* не умеет сам поставить задачу, или делает это в неопределенной форме ("хочу, чтобы был текст", "хочу, чтобы играла музыка"), этот способ работает превосходно: та же игровая приставка – это отличное решение крайне неопределенной задачи "хочу без толку потратить время". Однако стоит пользователю захотеть чего-то конкретного, начинаются трудности. Трудности могут быстро стать непреодолимыми, как только для этого "конкретного" не окажется **готового** решения: внутренняя структура систем, ориентированных на "решения", столь сложна и столь плохо документирована, что сделать что-либо вручную, скорее всего, не удастся. *Вывод*: *пользователь*, понимающий суть собственных задач, – не "клиент", он должен иметь возможность быстро и качественно решать задачи **самостоятельно**, а не выбирать из готовых "решений" то, которое нанесет меньше вреда.

Что же нужно идеальному – достаточно подготовленному, чтобы действовать самостоятельно, и достаточно занятому, чтобы не переделывать систему – пользователю? По-видимому, механизм, с помощью которого можно сформулировать и придать операционной системе все требуемые свойства, имея возможность описывать решение задачи, по крайней мере, на том же уровне конкретности, на котором было поставлено ее условие. Большая часть **других**, не нужных для решения собственных задач пользователя, свойств должны быть "стандартными" и не требовать его вмешательства.

**Профиль системы**

Так возникает идея разделить систему на два подмножества: *профиль* и *реализацию*. Все, что не потребует вмешательства пользователя, необходимо запрограммировать и применять в готовом виде в качестве составных частей *реализации*. В Linux этому соответствуют программы и подпрограммы: ядро, модули, демоны, утилиты; используемые ими библиотеки и прочие разделяемые файлы и т. п. ***Реализация*** – это монолитная, неизменяемая часть системы, устроенная по типу "решений" основных задач, только задачи эти, как правило, не совпадают с задачами пользователей, а только **помогают** решать их, то есть служат как бы строительным материалом, деталями и инструментами сборки "больших" решений.

Все, чего может коснуться рука человека, из *реализации* переносится в *профиль* системы. *Профиль* задает поведение *реализации* на данных пользователя и должен быть устроен так, чтобы пользователь мог его беспрепятственно изменять, если понадобится. С одной стороны, это может быть вариант "высокоуровневого программирования", когда пользователь описывает алгоритм решения и структуру используемых данных на некотором высокоуровневом языке (специализированном или общем, например, на shell). С другой стороны, задание свойств может превращаться в указание **модификаторов поведения**, когда пользователь просто перечисляет необходимые параметры работы программы, которые изменяют ее заранее известную, но достаточно общую функциональность.

Таким образом система полностью описывается в виде набора необходимых компонентов *реализации*, активизированных (запущенных) с определенными *профилями*, вкупе с текущим состоянием каждого компонента. Поскольку компонент *реализации* не может изменяться, а его текущее состояние, наоборот, меняется постоянно и не управляется пользователем, можно считать, что систему задает ее *профиль*. Это означает, что для того, чтобы продублировать работу системы на другом компьютере, достаточно установить там стандартную *реализацию* и перенести *профиль* (обычно занимающий несравненно меньше места) и *пользовательское наполнение*. Наполнение (файлы пользователей, содержимое www-страниц и т. п.) может занимать много места, но оно входит в понятие "задача пользователя", поэтому забывать о нем нельзя.

**Профиль**. Изменяемая часть системы, определяющая ее поведение во время работы.

Как проще всего **создать** *профиль* если не всей системы, то хотя бы ее компонента (программы)? Один из вариантов такой: снабдить программу функцией "сохранить настройки", тогда можно будет эту программу запустить, **любым способом** добиться ее работоспособности, а после зафиксировать достигнутое состояние с помощью этой функции. При этом поначалу совершенно неважно, как **выглядят** эти настройки: программа-то заработала, значит, цель достигнута (проницательный Мефодий немедленно заметил, что название функции "сохранить настройки" как-то подозрительно похоже на название **кнопки** ).

Зачастую для того, чтобы собрать более или менее отвечающий требованиям пользователя *профиль*, задействуется больше ресурсов, чем для работы самой программы. Утилита автоматической настройки выглядит эдаким шаманом, или кудесником, который, задав всего несколько вопросов человеку, непонятным образом приводит систему в работоспособное состояние. Такая подсистема и называется "wizard", причем в русском переводе ее отчего-то стесняются называть "шаманом", а величают уважительно "мастером".

Вот пример поведения обычного шамана-настройщика (пакет wvdial, заведующий модемным подключением к Internet):

[root@localhost root]# wvdialconf

Usage: wvdialconf

(create/update a wvdial.conf file automatically)

[root@localhost root]# wvdialconf .wvdialrc

Scanning your serial ports for a modem.

Port Scan<\*1>: Scanning ttyS4 first, /dev/modem is a link to it.

. . .

ttyS4<\*1>: Modem Identifier: ATI -- Xircom CardBus 10/100+Modem 56 (Revision 2.40)

. . .

ttyS4<\*1>: ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0 -- OK

ircomm0<\*1>: ATQ0 V1 E1 -- failed at 9600 and 19200 baud.

. . .

ircomm9<\*1>: ATQ0 V1 E1 -- failed at 9600 and 19200 baud.

Port Scan<\*1>: LT0

. . .

ttyS0<\*1>: ATQ0 V1 E1 -- and failed too at 115200, giving up.

. . .

ttyS1<\*1>: ATQ0 V1 E1 -- and failed too at 115200, giving up.

Port Scan<\*1>: S2 S3 S5 S6 S7 S8 S10

. . .

Port Scan<\*1>: USB11 USB12 USB13 USB14 USB15

Found a modem on /dev/ttyS4, using link /dev/modem in config.

Modem configuration written to .wvdialrc.

ttyS4: Speed 115200; init "ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0"

Пример 12.1. Кудесник wvdialconf

Ни о каких наводящих вопросах даже речи не зашло! Программа проверила более полусотни устройств, не модемы ли они, но нашла всего одно – /dev/ttyS4. Его настройки определились автоматически (и хорошо, потому что Мефодий не знает, что такое " ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0 "). *Профиль* (а wvdialrc создает именно *профиль* ) лежит теперь в файле .wvdialrc, так что программа wvdial начнет работать с модемом, нуждаясь только в **пользовательских** настройках ( *входное имя*, пароль и т. п.).

Яркий пример того, как элементы *реализации* связываются *профилем* в единую подсистему для решения определенной задачи – командная строка и сценарии командного интерпретатора. Здесь утилиты играют роль элементов *реализации*, их параметры – роль "настроечной" части *профиля*, а способ их объединения в сценарий – "программируемой" части *профиля*. Скажем, команда find /etc -type f 2> /dev/null | xargs -n1 file | cut -d: -f2 | sort | uniq -c задействует шесть утилит системы: командную оболочку, find, xargs, cut, sort и uniq, причем четыре из них запускаются с измененным *профилем* 1. Командная оболочка дополнительно программируется для создания конвейера между командами.

**Конфигурационный файл**

Задание *профиля* с помощью командной строки – метод далеко не всегда удобный. Даже при работе с самой командной строкой используется *окружение* для сохранения настроек, чтобы не задавать их всякий раз и для всякой команды. Что уж говорить о сложных системных службах, свойства которых должны сохраняться не от сеанса к сеансу, а постоянно (в том числе при перезагрузке системы). Вывод прост: *профиль* необходимо держать в файле, вроде того, что создается по команде "сохранить настройки".

Однако сам подход к хранению *профиля* в файле, при котором пользователь не может изменять этот файл напрямую, а пользуется "умными" конфигураторами, удобен только в случаях, когда настроек много, а цена ошибки невелика (например, при настройке внешнего вида рабочего стола). В общем случае довольно сложно задать **поведение** системы на основании описания (часто неявного) свойств того, что эта система должна получать **в результате**. Иными словами, из описания того, что должно получиться, далеко не всегда можно автоматически сделать вывод, как оно должно получаться.

**Конфигурационный файл**. Текстовый файл, содержащий настройки какой-нибудь части системы (утилиты, демона и т. п.). Как правило, считывается ею при запуске. Типичный для Linux способ организации *профиля* .

Одним словом, если есть *конфигурационный файл*, то должны быть и средства редактирования этого файла. Учитывая, что в Linux реализована высокоразвитая система хранения и **переработки** (как ручной, так и автоматической) данных в текстовом виде, изобретать какой-то новый формат – все равно что изобретать велосипед. Тем более, что именно текст, разделенный на строки и слова, лучше всего подходит тогда, когда есть четкое деление *профиля* на объекты управления и их свойства (например, настройки какого-нибудь демона и значения этих настроек). Вдобавок, именно со структурированными текстами отменно управляются текстовые редакторы Linux: Vi, Emacs и др:

methody@localhost:~ $ cat .vimrc

so $VIMRUNTIME/vimrc\_example.vim

" Some mappings

map :wall!^M

map! ^O:wall!^M

" Tune up

set shiftwidth=2 tabstop=8 history=200 viminfo='50

set showmode showmatch showcmd ruler modeline

set autoindent ignorecase smartcase

set nohlsearch noincsearch

set dir=/var/tmp

set wildmode=list:longest,full

set wildmenu

" Colouring

syntax on

colorscheme desert

Пример 12.2. Настройки редактора vim

Вот как выглядит *конфигурационный файл* для *Vim*, написанный Мефодием на основе файла, взятого у Гуревича. Легко заметить, что файл состоит из команд режима командной строки Vi с комментариями (в отличие от большинства утилит Linux, в Vi комментарии начинаются на """). Символы " ^O " и " ^M " – это именно соответствующие управляющие символы (вставленные в текстовый файл с помощью " ^V ", см. лекцию 9). Такой *конфигурационный файл* легко понимать и изменять.

Как уже было замечено, набор переменных *окружения* составляет особенный *профиль*, к которому чувствительны все запускаемые программы – в этом его достоинство. Задаются переменные *окружения* обычно в командном сценарии, который тоже можно рассматривать как *конфигурационный файл* ). Например, во многих дистрибутивах используется *конфигурационный файл* .i18n для настройки языковых особенностей клавиатуры, языка вывода сообщений и т. п. 2:

methody@localhost:~ $ cat .i18n

LANG=ru\_RU.KOI8-R

LANGUAGE=ru\_RU.KOI8-R

SYSFONTACM=koi8-r

SYSFONT=UniCyr\_8x16

DICTIONARY=russian

MPAGE="-CKOI8-R"

export DICTIONARY MPAGE

Пример 12.3. Файл настройки языковых особенностей

Однако хранить настройки специфической программы (не нужные всем остальным) в *окружении* – не самое удачное решение: синтаксис, задающий переменную *окружения*, слишком прост ( имя\_переменной=значение ), а самих переменных становится слишком много, поэтому при просмотре трудно выделить, какая из них к какой группе настроек относится. Если пытаться упаковать все настройки в значение **одной** переменной, это значение окажется трудночитаемым, и все преимущество текстового формата сойдет на нет. Например, стандартный *конфигурационный файл* утилиты ls (точнее, только ее цветовых предпочтений) – /etc/DIR\_COLORS (его можно подменить личным файлом ~/.dir\_colors ) занимает около ста строк вместе с комментариями. Команда ls использует не этот файл, а создаваемую утилитой dircolors переменную LS\_COLORS, значение которой – 600-символьная строка без всяких комментариев.

Если *профиль* слишком велик, держать его в одном *конфигурационном файле* – значит, доставлять пользователю сомнительное удовольствие разбирать этот файл целиком даже при необходимости внести минимальное изменение. Методов борьбы с неудобочитаемостью несколько. В частности, уже известный по лекции 10 механизм " .d ": файл разделяется на несколько **независимых** друг от друга файлов так, что редактировать приходится только один из файлов, а программа во время самонастройки считывает все.

Другой способ опирается на то, что **изменения**, которые пользователь вносит в *профиль*, как правило, существенно меньше объема всего *профиля*. Поэтому может быть выгодно хранить все настройки по умолчанию в каком-нибудь файле, менять который вообще не надо, а файл пользовательских настроек использовать как бы "поверх", изменяя *профиль* в соответствии с ними **после** того, как выставлен *профиль* по умолчанию. Дополнительное преимущество такого способа – в том, что пользователь всегда может подглядеть в "большой" файл, чтобы узнать, как оформляется та или иная настройка. Например, утилита updfstab, которая изменяет содержимое /etc/fstab при появлении или удалении съемного дискового носителя (например, лазерного диска), считывает данные из *конфигурационного файла* /etc/updfstab.conf. Сам этот файл состоит из единственной строки: include /etc/updfstab.conf.default, что приводит к чтению файла с настройками по умолчанию, где задан способ работы со многими съемными устройствами системы. Если администратору нужно как-то изменить поведение updfstab в отношении определенного устройства, он копирует соответствующую группу настроек из updfstab.conf.default в updfstab.conf **после** строчки include... и исправляет их. То, что эти группы настроек читаются дважды, не играет особой роли: чтение коротких файлов выполняется быстро.

Наконец, третий способ сделать *конфигурационный файл* удобочитаемым — **секционирование** *профиля*, когда все настройки разбиваются на группы, каждой группе дается собственное имя, и синтаксис *конфигурационного файла* проектируется так, чтобы границы групп хорошо различались при просмотре. В сущности, этот способ – предок схемы " .d ", где группе соответствует отдельный файл, однако нередки ситуации, когда разбивать на файлы неудобно (например, если группы не полностью независимы, поэтому может понадобиться редактировать их сразу несколько). *Конфигурационный файл* *утилиты дозвона* wvdial, например, секционируется по адресату (провайдеру) плюс отдельная секция "по умолчанию". Сами секции отделяются друг от друга заголовками, заключенными в квадратные скобки:

root@localhost:~> cat .wvdialrc

[Dialer Defaults]

Modem = /dev/modem

Baud = 115200

Init1 = ATZ

Init2 = ATQ0 L0 M4 V1 E1 S0=0 &C1 &D2 +FCLASS=0

Auto DNS = on

Modem Type = Analog Modem

[Dialer hotspace]

Phone = 0123456

Username = fireman

Password = Fire!Fire!

TOnline = true

[Dialer warlock]

Phone = 0246813

Username = cop-120

Password = gimmethegun

Force Address=10.0.0.120

Пример 12.4. Секционированный конфигурационный файл

Утилита wvdial обладает высокоразвитым искусственным интеллектом: она самостоятельно догадывается, какой именно тип *идентификации* используется на сервере. Например, "с той стороны" может оказаться *терминал* Linux, которому требуется сначала ввести обыкновенное *входное имя* и пароль, затем надо получить командную строку, запустить на **сервере** сетевой демон *pppd*, и только после этого запустить *pppd* на собственной машине. Другой вариант: *pppd* на сервере уже запущен, а настройки "Username" и "Password" означают идентификационную информацию протокола *CHAP*, используемого *pppd*. Обо всем этом и о многом другом wvdial способна догадаться,так же как wvdialconf умел определять, какое же из устройств является модемом.

Однако на любой искусственный интеллект найдется непостижимая ему жизненная ситуация. На одном из серверов (секция "Dialer hotspace") **тоже** стоит программа с зачатками искусственного интеллекта, которая **тоже** пытается определить, каким способом хочет идентифицироваться позвонивший. Оттого эти два кудесника, созвонившись, все ждут, пока кто-нибудь не проявит себя... Помогает настройка TOnline, которая заставляет wvdial немедленно задействовать протокол ppp, на что сервер, подумавши "ах, ppp!", с облегчением запускает *pppd*. Остается вопрос: почему эта полезная настройка никак не отражена в документации (ее нашел в исходных текстах программы Гуревич)? Не потому ли, что пара wvdialconf-wvdial не по-Linux-овски стремится все делать за пользователя, а стало быть, пользовательская документация для разработчиков этой программы – не главное?

Идею чтения настроек по умолчанию можно развить. Оказывается, бывает удобно, когда описание настройки помещено не в руководство, а непосредственно в *конфигурационный файл* в виде комментариев. Тогда при изменении этой настройки пользователь сразу видит, что она собой представляет, при этом отпадает необходимость сначала находить строчку в файле, а потом искать ее же в руководстве. Такой распространенный способ оформления называется *самодокументированием* *профиля* . Например, файл /etc/man.conf, управляющий работой команды man, оформлен в самодокументированном стиле:

methody@localhost:~ $ cat /etc/man.conf

. . .

# NOCACHE keeps man from creating cache pages ("cat pages")

# (generally one enables/disable cat page creation by

# creating/deleting the directory they would live in – man

# never does mkdir)

#

# NOCACHE

# The command "man -a xyzzy" will show all man pages for xyzzy.

# When CMP is defined man will try to avoid showing the same

# text twice. (But compressed pages compare unequal.)

#

CMP /usr/bin/cmp -s

. . .

Пример 12.5. Самодокументированный конфигурационный файл

Мефодий, может быть, и не понял бы сразу, зачем команде man использовать утилиту cmp, однако в поясняющем комментарии написано: когда нужно показать несколько руководств разом, они предварительно сравниваются, и показываются только несовпадающие.

Если пойти еще дальше, то можно создать несколько различных файлов с примерами настроек, чтобы пользователь мог взять один из них и довести до нужного ему состояния. Именно такую – демонстрационную – настройку Мефодий и включил в качестве настройки по умолчанию в свой .vimrc (в первой строке). Кстати, на самом деле *профиль* *Vim* весьма сложен, но большинство его настроек по умолчанию находятся в различных файлах дерева каталогов /usr/share/*vim* – эдакая "схема .d/.d ", где файлы *профиля*, соответствующие подгруппам настроек, лежат в подкаталогах, соответствующих группам. Включение определенного настроечного файла может происходить неявно: например, строчка colorscheme desert из .vimrc приводит к чтению /usr/share/*vim*/colors/desert.*vim*.

#### Конфигурационные файлы могут иметь довольно замысловатый синтаксис, если соответствуют сложным структурам данных (таковы, например, настройка irc-клиента irssi ) или содержать в себе дополнительные средства самодокументирования (например, файл настройки текстового www-броузера lynx не просто хорошо документирован, но и размечен теми же средствами, какие используются в самом броузере для п**Изменение конфигурационных файлов**

Как правило, *конфигурационный файл* считывается программой при запуске, отражая, таким образом, ее состояние на момент старта. Изменения настроек работающей программы в *конфигурационном файле*, как правило, не отражаются. Тому есть несколько причин: не стоит превращать файл, изредка редактируемый пользователем, в файл, изменение которого происходит постоянно; не стоит держать *конфигурационный файл* всегда открытым; тяжело, изменяя файл автоматически, не испортить структуру комментариев (интерпретируемых не машиной, а пользователем) и т. д. Впрочем, многие утилиты, особенно использующие графическую среду, могут записывать настройки в файл по окончании работы. Большинство *конфигурационных файлов* весьма удобно редактировать вручную, с помощью Vi или Emacs (для файлов, более или менее похожих, используется общая *подсветка синтаксиса*, а для наиболее популярных существуют и собственные варианты подсветки).

В /etc хранятся настройки системных служб, в том числе настройки по умолчанию, настройки по умолчанию пользовательских утилит, *профили* командных интерпретаторов, а также настройки, используемые в процессе загрузки системы (например, modules.conf ). Там же располагаются и *стартовые сценарии*, о которых рассказано в лекции 10. Чего не стоит искать в /etc, так это разнообразных **примеров** настройки той или иной службы. Считается, что пример – это часть документации, и их следует помещать, например, в /usr/share/doc/название\_службы/examples.

Файлы, имеющие отношение к процессу *досистемной загрузки*, обычно лежат не там, а в /boot ; это стоит иметь в виду, так как /boot/*grub*/menu.lst – тоже часть *профиля* системы, хотя и довольно специфическая. В *профиль* системы входит содержимое каталогов etc из так называемых "песочниц", расположенных обычно в /var/lib.

Смысл термина "песочница" вот в чем. В Linux есть замечательный *системный вызов* chroot() и использующая его утилита chroot, формат командной строки которой chroot каталог команда. Эта утилита запускает команду, изменив *окружение* таким образом, что та считает каталог корневым. Соответственно, все подкаталоги каталога представляются команде каталогами первого уровня вложенности, и т. д. Если необходимо во что бы то ни стало **ограничить** область действия некоторой утилиты (например, по причине ее небезопасности), можно запускать ее с помощью chroot. Тогда, даже имея права суперпользователя, эта утилита получит доступ только к каталогу и его подкаталогам, а /etc и прочие важные части системы окажутся в неприкосновенности. Сам каталог как раз и играет роль "песочницы", в которую утилиту "пустили поиграть", позволяя вытворять что угодно. Часто бывает, что в "песочнице" есть и свой каталог etc, содержащий необходимые для запуска утилиты (или системной службы) настройки. Вот этот-то etc из "песочницы" также входит в список каталогов, хранящих *профиль* системы.

В /etc могут находиться не только файлы, но и подкаталоги (особенно в стиле " .d ") и целые поддеревья каталогов. Например, в некоторых дистрибутивах Linux используется подкаталог /etc/sysconfig. Этот каталог создается и заполняется файлами при установке системы или при запуске специального "конфигуратора" – программы-кудесника, задающей наводящие вопросы. Некоторые *стартовые сценарии*, использующие полученные настройки, также лежат в этом каталоге или его подкаталогах. Если в системе есть каталог /etc/sysconfig, там должны оказаться настройки, относящиеся не к самим службам или утилитам, а к способу их запуска при загрузке, а также языковые и сетевые настройки, тип мыши и т. д.

**Подсистема учетных записей**

Несколько *конфигурационных файлов* и способы работы с ними заслуживают отдельного рассмотрения. В первую очередь Мефодий заинтересовался системой учетных записей, о которой упоминалось сразу в нескольких предыдущих лекциях. Итак, существует два файла, доступных для чтения всем пользователям: /etc/passwd, хранящий учетные данные пользователей, и /etc/group, определяющий членство пользователей в группах (во всех, кроме группы по умолчанию):

methody@localhost:~ $ cat /etc/passwd

root:x:0:0:System Administrator:/root:/bin/bash

bin:x:1:1:bin:/:/dev/null

daemon:x:2:2:daemon:/:/dev/null

adm:x:3:4:adm:/var/adm:/dev/null

lp:x:4:7:lp:/var/spool/lpd:/dev/null

. . .

nobody:x:99:99:Nobody:/var/nobody:/dev/null

shogun:x:400:400:Лев Гуревич:/home/shogun:/bin/zsh

methody:x:503:503:Мефодий Кашин:/home/methody:/bin/bash

methody@localhost:~ $ cat /etc/group

root:x:0:root

bin:x:1:root,bin,daemon

daemon:x:2:root,bin,daemon

sys:x:3:root,bin,adm

adm:x:4:root,adm,daemon,shogun

wheel:x:10:root,shogun

. . .

proc:x:19:root,shogun

shogun:x:400:

methody:x:503:

Пример 12.6. Файлы /etc/passwd и /etc/group

Оба файла состоят из строк, поля которых разделяются двоеточиями. В файле passwd – семь полей. Первое из них определяет *входное имя* пользователя – то самое, что вводится в ответ на " login:". Второе поле в ранних версиях UNIX использовалось для хранения *ключа пароля*. В Linux пользовательские пароли не хранятся **нигде** – ни в явном виде, ни в зашифрованном. Вместо этого хранится *ключ (hash) пароля* – комбинация символов, **однозначно** соответствующая паролю, из которой, однако, сам пароль получить нельзя. Иными словами, существует алгоритм превращения пароля в *ключ*, а алгоритма превращения *ключа* в пароль нет. Когда пользователь регистрируется в системе, из введенного им пароля изготавливается еще один *ключ*. Если он совпадает с тем, что хранится в учетной записи, значит, пароль правильный.

Авторы UNIX предполагали, что, раз пароль из *ключа* получить нельзя, *ключ* можно выставлять на всеобщее обозрение, однако выяснилось, что, узнав *ключ*, пароль можно попросту **подобрать** на очень мощной машине или в предположении, что пароль – это английское слово, год рождения, имя любимой кошки и т. п. Если подбор пароля сопровождается неудачными попытками входа в систему, это отражается в *системных журналах* и может быть легко замечено. А завладев *ключом*, злоумышленник сможет заняться подбором пароля втихомолку на каком-нибудь суперкомпьютере. В конце концов, *ключ* не нужен никому, кроме подсистемы *идентификации*, поэтому его вместе с другими полями учетной записи, о которых всем знать не обязательно, из /etc/passwd перенесли в "теневой" файл учетных записей – /etc/shadow. На месте *ключа* в Linux должна быть буква " x "; если там стоит что-то другое, *идентификация* пользователя не сработает, и он не сможет войти в систему.

Третье и четвертое поля passwd – *идентификатор пользователя* и *идентификатор группы* по *умолчанию*. Как уже говорилось в лекции 6, именно *идентификатор пользователя*, а не его *входное имя*, **однозначно** определяет пользователя в системе и его права. Можно создать несколько учетных записей с **одинаковыми** UID, которые различаются другими полями. Тогда при регистрации в системе под именами из этих записей пользователи могут получать разные *домашние каталоги* и командные оболочки, разное членство в группах, но иметь один и тот же UID. Пятое поле отводится под "полное имя" пользователя; это единственное поле passwd, содержимое которого ни на что не влияет. Наконец, шестое и седьмое поля содержат *домашний каталог* и *стартовый командный интерпретатор* пользователя. Если седьмое поле пусто, подразумевается /bin/sh, а если его содержимое не встречается в файле /etc/shells, содержащем **допустимые** командные интерпретаторы, неизбежны трудности при *идентификации* пользователя.

Строки файла /etc/group состоят из четырех полей, причем второе – *ключ* группового пароля – обычно не используется. Первое поле – это имя группы, третье – это *идентификатор группы*, а четвертое – это список *входных имен* пользователей, которые в эту группу входят (более точно – для которых эта группа является **дополнительной**, так как *группа по умолчанию* указывается в /etc/passwd, хотя никто не мешает продублировать *группу по умолчанию* и в /etc/group ). Таким образом, определение членства пользователя в группах зависит не от его UID, а от *входного имени*.

Упомянутый выше файл /etc/shadow, доступ к которому имеет только суперпользователь, также состоит из полей, разделяемых двоеточиями. Помимо *входного имени* и *ключа пароля* там указываются различные временные характеристики учетной записи: нижняя и верхняя граница времени жизни пароля, самой учетной записи, дата последнего изменения и т. п. *Ключ пароля* (второе поле) указывается в одном из поддерживаемых форматов, а если формат не опознан, вся учетная запись считается недействительной. Поэтому один из способов запретить регистрацию пользователя в системе – добавить один символ (например, " !") прямо в поле *ключа*, отчего все поле становится синтаксически неправильным. Вновь разрешить пользователю входить в систему можно, удалив из поля лишний символ. Именно так работает (с ключами " -L ", lock, и " -U ", unlock) утилита usermod, изменяющая учетную запись. С помощью этой утилиты можно отредактировать и все остальные поля как passwd, так и shadow.

Добавить и удалить пользователя или группу можно с помощью утилит useradd, userdel, groupadd и groupdel соответственно. Не стоит пользоваться текстовым редактором, так как он не гарантирует *атомарности* операции добавления/удаления, хотя бы потому, что изменению подлежат сразу два файла – passwd и shadow. Даже если необходимо **отредактировать** /etc/passwd или /etc/group (например, для добавления пользователя в группу или удаления его оттуда), стоит запускать не просто редактор, а vipw или vigr (именно их поведение, позволяющее соблюсти *атомарность*, копирует утилита visudo, описанная ранее):

[root@localhost root]# useradd -g users -G proc,cdrom -c "Incognito" incognito

[root@localhost root]# id incognito

uid=504(incognito) gid=100(users) groups=100(users),19(proc),22(cdrom)

[root@localhost root]# userdel -r incognito

[root@localhost root]# id incognito

id: incognito: No such user

Пример 12.7. Добавление и удаление пользователя

Здесь был добавлен пользователь incognito, *группа по умолчанию* – users, член групп proc и cdrom, полное имя – "Incognito". Стоит заметить, что **пароль** для этой учетной записи установлен не был (чтобы создать пароль, стоило запустить команду passwd incognito ), и, даже если бы пользователя тут же не удалили ( userdel -r удаляет также и *домашний каталог*, и почтовый ящик), зарегистрироваться в системе под именем incognito было бы все равно невозможно.

#### **Подсистема идентификации**

Подсистемой учетных записей пользуется подсистема *идентификации*, которая в Linux имеет модульную структуру и называется *PAM* (Pluggable Authentication Modules, т. е. "Подключаемые модули *идентификации* "). Идея *PAM* – в том, чтобы унифицировать и, вместе с тем, сделать более гибкими **любые** процедуры *идентификации* в системе – начиная от команды login и заканчивая доступом к файлам по протоколу, скажем, FTP. Для этого недостаточно просто написать "библиотеку *идентификации* " и заставить все программы ее использовать. В зависимости от того, для чего производится *идентификация*, условия, при которых она будет успешной, могут быть более или менее строгими, а если она прошла успешно, бывает нужно выполнить действия, связанные не с **определением** пользователя, а с настройкой системы.

В большинстве дистрибутивов *PAM* обучен схеме " .d ", и настройки каждой службы, которая использует *идентификацию*, лежат в отдельном файле:

[root@localhost root]# ls /etc/pam.d

chpasswd groupdel other system-auth userdel

chpasswd-newusers groupmod passwd system-auth-use\_first\_pass usermod

crond login sshd user-group-mod

groupadd newusers su useradd

Пример 12.8. Подключаемые модули идентификации (PAM)

В *PAM* определено четыре случая, требующие *идентификации*: auth – собственно *идентификация*, определение, тот ли пользователь, за кого он себя выдает, account – определение, все ли хорошо с учетной записью пользователя, password – **изменение** пароля в учетной записи, и session – дополнительные действия непосредственно перед или непосредственно после того, как пользователь получит доступ к затребованной услуге. Эти значения принимает первое поле любого файла настройки из *pam*.d, а в третьем поле записывается **модуль**, который проверяет какой-нибудь из аспектов *идентификации*. Второе поле определяет, как успех или неуспех проверки одного модуля влияет на общий успех или неуспех *идентификации* данного типа (например, required означает, что в случае неуспеха модуля проверка пройдена не будет). Четвертое и последующие поля отведены под параметры модуля:

[root@localhost root]# cat /etc/pam.d/login

auth include system-auth

auth required pam\_nologin.so

account include system-auth

password include system-auth

session include system-auth

session optional pam\_console.so

[root@localhost root]# cat /etc/pam.d/system-auth

auth required pam\_tcb.so shadow count=8 nullok

account required pam\_tcb.so shadow

password required pam\_tcb.so use\_authtok shadow count=8 write\_to=tcb

session required pam\_tcb.so

Пример 12.9. Настройка PAM для login

Такие настройки login обнаружил Мефодий на своем компьютере. Во всех четырех случаях используется *включаемый файл* system-auth (к нему обращаются и другие службы), с некоторыми дополнениями. Так, во время *идентификации* pam\_nologin.so дополнительно проверяет, не запрещено ли пользователям регистрироваться вообще (как это бывает за несколько минут до перезагрузки системы), а перед входом в систему и после выхода из нее pam\_console.so выполняет описанную в лекции 6 "передачу прав на владение устройствами" (и, соответственно, лишение пользователя этих прав).

Каталог /etc/*pam*.d – замечательный пример того, как *профиль* определяет **поведение** системы. В частности, четыре первых строки из system-auth показывают, что в этом дистрибутиве используется не просто "теневой" файл паролей, а схема *TCB*, описанная в лекции 6. (Как уже известно Мефодию, в этой схеме вместо общего для всех /etc/shadow задействованы файлы вида /etc/*tcb*/входное\_имя/shadow, причем права доступа к ним устроены таким образом, чтобы при выполнении команды passwd можно было обойтись без подмены пользовательского *идентификатора* на суперпользовательский).

**Подсистема системных журналов**

Проста и остроумна в Linux подсистема ведения *системных журналов* – демон syslogd, управляемый *конфигурационным файлом* /etc/syslog.conf и " .d "-каталогом /etc/syslog.d. Если какой-нибудь демон или служба желают сообщить системе о том, что наступило событие, которое стоит запомнить, у нее есть два пути. Во-первых, можно просто добавлять очередную запись в файл, который сам этот демон и открыл; этот файл будет **журналом** его сообщений. Во-вторых, можно воспользоваться *системным вызовом* syslog(), который переадресует текстовое сообщение специальному демону – syslogd – а уж тот разберется, что с этим сообщением делать: записать в файл, вывести на 12-ю консоль или забыть о нем. Второй путь ( **централизованная** *журнализация*) предпочтительнее почти всегда; исключение составляет случай, когда сообщения по какой-то причине не могут быть текстовыми или этих сообщений предполагается посылать так много, что syslogd просто не справится.

Все события, о которых сообщается syslogd, подразделяются горизонтально – по **типу службы** (facility), с которой это событие произошло, и вертикально – по степени его **важности** (priority). Типов событий насчитывается около двадцати (среди них auth, daemon, kern, mail и т. п., а также восемь неименованных, от local0 до local7 ). Степеней важности всего восемь, по возрастанию: debug, info, notice, warning, err, crit, alert и emerg. Таким образом, каждое событие определяется парой значений, например, mail.err означает для syslogd событие, связанное с почтой, притом важности, **не меньшей** err. Из таких пар (с возможной заменой типа или важности на "\*", что означает "любые", или none, что означает "никакие") составляется *конфигурационный файл* /etc/syslog.conf:

[root@localhost root]# cat /etc/syslog.conf

\*.notice;mail.err;authpriv.err /var/log/messages

authpriv.\*;auth.\* /var/log/security.log

\*.emerg \*

\*.\* /dev/tty12

mail.info /var/log/maillog

Пример 12.10. Настройка системных журналов

В первом поле строки указываются *профили* сообщений, разделенные символом " ; ", а во втором – хранилище сообщений (файл, *терминал*, есть способы отдавать их на обработку программе и пересылать по сети). В примере в файл /var/log/messages попадают все сообщения важности не меньшей, чем notice, за исключением сообщений типа mail и authpriv, которые попадают туда, только если имеют важность не ниже err. Сообщения типа authpriv и auth любой важности попадают в файл /var/log/security.log, а типа mail и важности не ниже info – в файл /var/log/maillog. Сообщения типа emerg (наивысшей важности) выводятся на все *терминалы* системы, и, наконец, все сообщения выводятся на 12-ю *виртуальную консоль*.

Во многих системах используется основательно доработанный syslogd, позволяющий фильтровать сообщения не только по типу/важности, но и, например, по отправителю, задавать **точные** (а не "не меньшие") значения priority и т. п., однако такие доработки нужны для того, чтобы либо вести практически нефильтрованную *журнализацию* (получаются *системные журналы* совершенно нечитаемого объема), либо отводить поток сообщений определенной службы в отдельный журнал, опять-таки, не для чтения, а для последующей обработки.

Стоит заметить, что каталог /etc/syslog.d в новых версиях syslogd предназначен для хранения не профильных *конфигурационных файлов* в стиле " .d ", а *сокетов*, из которых демон может получать сообщения так же, как из сети или в результате *системного вызова* syslog().

**Выполнение действий по расписанию**

Другой пример типичной для Linux службы, управляемой *конфигурационным файлом*, – демон cron 3, регулярно выполняющий в заданное время заданные действия. Время от времени в системе необходимо обновлять разнообразные файлы, например, базы данных антивирусов (вирусов в Linux нет, а антивирусы есть!), базу данных *whatis* или список **всех доступных** на чтение файлов системы, locatedb (поискать по этому списку можно командой locate ); нужно собирать статистику по работе системы, анализировать цельность системы (этим занимаются службы OSec, TripWire или AIDE) и производить множество других регулярных действий. Всем этим и занимается демон cron.

*Конфигурационный файл* демона cron называется /etc/crontab.

[root@localhost root]# cat /etc/crontab

#minute (0-59),

#| hour (0-23),

#| | day of the month (1-31),

#| | | month of the year (1-12),

#| | | | day of the week (0-6 with 0=Sunday).

#| | | | | user

#| | | | | | commands

01 \* \* \* \* root run-parts /etc/cron.hourly

02 4 \* \* \* root run-parts /etc/cron.daily

22 4 \* \* 0 root run-parts /etc/cron.weekly

42 4 1 \* \* root run-parts /etc/cron.monthly

Пример 12.11. Настройка cron

Первые пять полей этого файла определяют время запуска команды: минуту, час, число месяца, месяц и день недели. Символ "\*" означает, что соответствующая часть даты не учитывается. Шестое поле – имя пользователя, от лица которого запускается команда, указанная в остальных полях строки. Так, в примере команда run-parts /etc/cron.weekly будет запускаться в 4 часа 22 минуты каждое воскресенье (нулевой день) любого числа любого месяца. Как видно из примера, обычно /etc/crontab невелик: чаще всего он состоит из почасового, подневного, понедельного и помесячного запуска специального сценария (в примере – run-parts ). Этот сценарий реализует упрощенную схему " .d ", он попросту запускает отсортированные в лексикографическом порядке4 сценарии из соответствующего каталога (например, из /etc/cron.daily ):

[root@localhost root]# ls /etc/cron.daily

000anacron logrotate makewhatis osec stmpclean updatedb

Пример 12.12. Сценарии, запускаемые ежедневно

Вот что происходит каждый день на машине Мефодия: запуск anacron и "прокручивание" *системных журналов* (об этом речь пойдет далее), обновление базы *whatis*, проверка цельности системы с помощью osec, прореживание старых и неиспользуемых файлов в /tmp (утилита stmpclean ) и, наконец, обновление базы updatedb.

Пользователям системы можно разрешить иметь собственные расписания, также обрабатываемые демоном cron. Эти расписания имеют тот же синтаксис, что и crontab, только шестое поле ("user") в них отсутствует. Редактировать пользовательские таблицы рекомендуется с помощью команды crontab -e (чтобы не подсунуть демону синтаксически неверный файл). Сами таблицы могут храниться, в зависимости от версии и настроек cron, в /var/spool/cron/crontabs, /var/spool/cron, /var/cron/tabs или еще где-нибудь.

Служба anacron появилась в Linux-системах в то время, когда их начали активно использовать на персональных рабочих станциях. Такие станции, в отличие от серверов, не обязаны работать круглосуточно. Скорее всего, на ночь, на праздники и на время отпуска их выключают. Это значит, что все настройки cron надо менять в соответствии с графиком включений/выключений (иначе cron.daily никогда не выполнится в четыре часа ночи) – или запускать отдельную службу, которая будет выполнять **некоторые** задачи не по расписанию, а потому что их **давно уже пора** запустить5. Дополнительно anacron рассчитывает запуск задач так, чтобы не перегрузить компьютер работой, если их накопилось слишком много. *Конфигурационный файл* anacron называется /etc/anacrontab.

**"Прокручивание" системных журналов**

Еще изучая работу syslog, Мефодий не расставался с мыслью, что файл, в котором записывается *системный журнал*, постоянно растет. Это значит, что каков бы ни был размер файловой системы /var, она в конце концов заполнится журналами под завязку – если как-то их не укорачивать. К сожалению, в Linux укоротить файл **от начала**, отрезав самые старые записи, нельзя, как нельзя и добавлять новые записи **в начало** файла. Эти операции легко реализовать с помощью копирования нужной области в новый файл и последующего переименования, но, во-первых, соблюсти *атомарность* таких составных операций нелегко, а во-вторых, они требуют **вдвое больше** места в файловой системе на время работы (и, стало быть, каких-то аварийных процедур на случай нехватки места).

Поэтому в Linux принят другой, существенно менее ресурсоемкий алгоритм, позволяющий избежать переполнения /var: так называемое "прокручивание" *системных журналов*. Суть алгоритма в следующем: когда настает пора укоротить журнал (например, раз в неделю или если файл журнала достиг определенного размера), этот файл **переименовывают**, и открывают новый пустой файл с тем же именем. Если хранить **несколько** (скажем, семь) переименованных старых файлов, с ними уже можно производить операцию "отбрасывания старого": самый старый – седьмой – файл удаляется, шестой переименовывается в седьмой, пятый – в шестой, и т. д. до первого (моложе которого только текущий журнал), который переименовывается во второй. Только тогда можно переименовать текущий журнал в "первый старый" и открыть новый. Получается очередь устаревающих файлов, пополняемая с одной стороны и усекаемая с другой.

Как правило, имя "первого старого" журнала получается путем добавления к имени журнала суффикса ".1", второго – ".2" и т. д.:

[root@localhost root]# ls -l /var/log/syslog/messages\*

-rw-r----- 1 root adm 292654 Dec 15 14:01 /var/log/syslog/messages

-rw-r----- 1 root adm 34452 Dec 13 01:09 /var/log/syslog/messages.1.bz2

-rw-r----- 1 root adm 35892 Dec 6 09:38 /var/log/syslog/messages.2.bz2

-rw-r----- 1 root adm 60806 Nov 28 10:59 /var/log/syslog/messages.3.bz2

-rw-r----- 1 root adm 61063 Nov 21 10:47 /var/log/syslog/messages.4.bz2

-rw-r----- 1 root adm 60079 Nov 14 21:18 /var/log/syslog/messages.5.bz2

Пример 12.13. Системный журнал messages

Прокручиванием *системных журналов* занимается утилита logrotate, которая тоже управляется и *конфигурационным файлом* /etc/logrotate.conf, и " .d "-каталогом /etc/logrotate.d/. Согласно настройкам, старые файлы можно сжимать упаковщиками bzip2 (как в примере) или gzip, можно задавать им определенные права доступа, можно посылать *сигнал* некоторой службе (чтобы она заметила подмену журнала, если она сама, а не syslogd занимается его пополнением) и т. п.

**Конфигурационные файлы в домашнем каталоге**

Немало *конфигурационных файлов* находится в *домашнем каталоге* пользователя. Этими файлами практически любая утилита может быть перенастроена по сравнению с обычным поведением, или поведением, задаваемым *конфигурационным файлом* из /etc. В Linux принято предоставлять пользователю возможность задавать *профиль* любого используемого им инструмента, начиная от простой утилиты и заканчивая графической подсистемой управления "рабочим столом" (см. об этом лекцию 15). Как правило, имена таких файлов или каталогов начинаются на " .", т. е. считаются скрытыми – для того, чтобы не засорять выдачу ls. Если пользователю нужно работать не со своими файлами, а именно с настройками, он всегда может применить ключ " -a " или " -A ":

methody@localhost:~ $ ls

bin cat.info cat.stderr Documents examples grep.info textfile tmp

methody@localhost:~ $ ls -AF

.alias .bashrc .emacs .inputrc~ textfile .Xauthority

.bash\_history bin/ examples/ .lpoptions tmp/ .xsession.d/

.bash\_logout cat.info grep.info .pinerc .viminfo

.bash\_profile cat.stderr .i18n .pyhistory .vimrc

.bash\_profile~ Documents/ .inputrc .pythonstartup .vimrc~

methody@localhost:~ $ rm .\*~

Пример 12.14. Конфигурационные файлы в домашнем каталоге

Многие утилиты **создают** *конфигурационный файл* при запуске, если его в *домашнем каталоге* пользователя нет, поэтому со временем объем ls -A становится все больше. Файл .lpoptions задает параметры подсистемы печати, .pinerc – это настройки почтового клиента pine, .viminfo – файл *истории команд* редактора *Vim*, а файл .Xauthority и каталог .xsession.d управляют запуском *графической подсистемы* X11, описанной в лекции 15. Из файлов в примере некоторые вообще не являются "стандартными": так, .aliases и .i18n просто "втягиваются" стартовым командным сценарием bash, потому что упомянуты в нем явно; строго говоря, они могли бы называться и по-другому. Все *конфигурационные*, стартовые и прочие **вспомогательные** файлы принято делать скрытыми, даже если никаких требований к их названиям нет.

Файл **.pythonstartup** (настройки интерпретатора языка программирования Python) выполняется потому, что имя этого файла задано в переменной *окружения* PYTHONSTARTUP. Мефодию пришлось дописать строку PYTHONSTARTUP="/home/methody/.pythonstartup"; export PYTHONSTARTUP в ~/.bash\_profile и "C-i": complete в ~/.inputrc, чтобы достраивание заработало и в этом интерпретаторе. Еще один файл, .pyhistory, используется в самом .pythonstartup:

methody@localhost:~ $ cat .pythonstartup

import atexit, os, readline, rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save\_history(historyPath=historyPath):

import readline

readline.write\_history\_file(historyPath)

if os.path.exists(historyPath):

readline.read\_history\_file(historyPath)

atexit.register(save\_history)

del os, atexit, readline, rlcompleter, save\_history, historyPath

Пример 12.15. Стартовый файл интерпретатора Python

Подавляющее большинство *конфигурационных файлов* предназначено для того, чтобы их мог редактировать пользователь. Эти файлы часто имеют самодокументированный формат и/или сопровождаются руководством, нередко вынесенным в отдельный от руководства по самой утилите документ.

**Лекция 13:**

**Управление пакетами**

**Пакеты**

Пригодная для работы пользователя система состоит из *множества* (сотен и тысяч) программ и утилит. В Linux каждый *компонент* системы представлен в виде *пакета*. Все *операции*, связанные с изменением состава системы – установка, удаление, проверка, обновление компонентов, –производятся над пакетами. В целом, *пакет* – это средство сделать так, чтобы *пользователь*-*администратор*, изменяя или обновляя программное наполнение системы, работал не с **файлами** (имена которых ему подчас неизвестны), а с определенными **функциями** самой системы: например,добавлял в нее не "500 файлов", а "WWW-сервер apache ".

**Архив файлов**

На первый взгляд, программа состоит из одного – исполняемого файла: запускаем файл, получаем работающую программу. Однако во время работы даже самая простая программа использует другие файлы, содержащие различные ресурсы: **библиотеки**, конфигурационные файлы, *файлы-дырки* и даже запускает другие программы. Чтобы программа действительно заработала, необходимо помимо главного исполняемого файла обеспечить наличие в системе всех нужных файлов с ресурсами.

Понятно, что при установке или удалении программы следует позаботиться и обо всех используемых ею файлах (которых может быть даже очень много). Это – **первая задача пакетирования**: все файлы, используемые программой, объединяются в один файл – **архив**. Это позволяет не копировать при установке программы все файлы по отдельности, а потом не удалять их таким же способом, а работать со всеми данными программы как с единым целым – устанавливать и удалять один *пакет*.

**Файловый архив**. Дерево каталогов, представленное в виде единого файла, состоящего из содержимого всех файлов в этом дереве и информации об имени и атрибутах каждого файла.

Нет жесткого требования, чтобы один *пакет* содержал только одну программу. В *пакет* естественно объединять такие ресурсы, с которыми удобно работать как с единым целым. Это может быть отдельная *программа* или набор утилит (например, coreutils, основные утилиты, унаследованные Linux от *UNIX*) или *модуль* с дополнительными возможностями программы, или общие для нескольких программ ресурсы. В процессе развития и/или устаревания программного обеспечения выделение некоторых задач в отдельный *пакет* может приобретать или терять смысл, поэтому способ объединения ресурсов в пакеты – это не что-то раз и навсегда выбранное: пакеты могут разделяться и сливаться.

Самый простой и традиционный для Linux способ объединить несколько файлов в один – использовать утилиту tar 1 Мефодий, написав несколько программ и сценариев, решил собрать их в одном файловом архиве, чтобы их удобнее было переносить на все системы, в которых ему случается работать. Для этого Мефодий создал *архив* со всем содержимым своего подкаталога bin/:

methody@localhost:~ $ tar -cf methody.progs.tar bin/

methody@localhost:~ $ tar -tf methody.progs.tar

bin/

bin/loop

bin/script

bin/to.sort

bin/two

Пример 13.1. Создание файлового архива при помощи tar

Первый *параметр* tar состоит из двух частей: операция, которую следует произвести над архивом, в данном случае "c" (*create*, создать), и *ключ* "f", который указывает, что *архив* следует создать в файле, *имя файла* архива – следующий *параметр*. Имя архива может быть любым, но Гуревич посоветовал снабдить его расширением ".tar", чтобы потом не путаться. После имени архива следуют имена файлов и каталогов, которые следует запаковать. 2

Чтобы проверить, какие файлы попали в *архив*, Мефодий просмотрел содержимое получившегося архива командой "tar -tf имя\_архива" ( "t" – просмотреть, "f" использовать *файл*, указанный следующим параметром). Тут Мефодий обратил внимание на два обстоятельства. Во-первых, в архиве имена файлов сохранились вместе с путем. Во-вторых,все пути, сохраненные в архиве – относительные.

При распаковке архива tar файлы извлекаются вместе с путем, недостающие подкаталоги создаются *по* мере необходимости. Все пути tar считает относительными, начиная от своего рабочего каталога. Если теперь Мефодий распакует свой *архив* (командой "tar -xf имя\_архива" ),то в рабочем каталоге будет создан *подкаталог* "bin/" и в него будут записаны все файлы из архива:

methody@localhost:~ $ tar -xvf methody.progs.tar

bin/

bin/loop

bin/script

bin/to.sort

bin/two

Пример 13.2. Распаковка архива

*Ключ* "v" велит tar быть "разговорчивым" (*verbose*), т. е. выводить больше диагностических сообщений, и благодаря этому tar при распаковке выводит имена (с путем) всех записываемых файлов. Если в рабочем каталоге уже есть *файл*, который tar собирается создать при распаковке, то этот *файл* будет попросту **заменен** файлом из архива. Так, когда Мефодий распаковал свой *архив*, *подкаталог* "bin/" со всем его содержимым **заменился** на *подкаталог* из архива. В данной ситуации это нестрашно, поскольку в архиве файлы такие же, но вот если бы Мефодий перед распаковкой изменил какие-то из своих файлов в "bin/", он лишился бы всех изменений.

**Формат пакета**

Помимо хранения архива файлов у *пакета* есть и другие задачи (они обсуждаются в двух следующих разделах), поэтому для *пакета* в Linux не очень подходит обычный *файловый архив*, наподобие .tar, а нужен специальный формат. Таких форматов в Linux бывает несколько (краткое перечисление и характеристика – в разделе "*Установщики пакетов*"), в системе Мефодия используется один из наиболее распространенных – rpm, поэтому все примеры в данной лекции будут с его участием. Для работы с пакетами в специальном формате нужна специальная программа-установщик, которая так же и называется – rpm: она занимается установкой, удалением, обновлением и проверкой пакетов.

*Пакет* в формате rpm – это единый файл со всеми необходимыми данными. Существуют определенные (хотя и не очень жесткие и последовательные) соглашения относительно названий файлов-пакетов. Например, rpm -файл, содержащий комплект стандартных утилит coreutils, в системе Мефодия называется coreutils-5.2.1-some5.rpm: сначала – *имя пакета*, затем через дефис – служебная информация о номерах версий программного обеспечения и самого *пакета*.

**Регистрация в системе**

Итак, *пакет* с компонентом системы – это в первую очередь *файловый архив*, в котором хранятся все необходимые файлы вместе с путями к ним (т. е. каталогами). Когда компонентов много, нужно сделать так, чтобы в разных пакетах не оказалось файлов с одинаковым именем и путем,чтобы файл, принадлежащий одному *пакету*, не мог быть заменен файлом другого *пакета* при установке. Отслеживать такого рода *конфликты пакетов* – **вторая задача пакетирования**.

Чтобы предупреждать конфликты, в системе должна храниться информация обо всех уже установленных пакетах и принадлежащих им файлах. Когда точно известно, какие файлы принадлежат *пакету*, можно полностью удалить *пакет*, не оставив и не удалив при этом ничего лишнего.Такой подход препятствует образованию в системе "мусора" – бесполезных файлов, оставшихся от удаленных программ – и делает операцию установки/удаления *пакета* полностью обратимой.

Rpm хранит информацию обо всех установленных в системе пакетах и принадлежащих им файлах и может выдать эту информацию по запросу пользователя. Почитав руководство по rpm, Мефодий выяснил, что список всех установленных в системе пакетов (скорее всего, очень длинный, в несколько тысяч строк) можно получить командой "rpm -qa",список всех файлов, принадлежащих *пакету*, – командой "rpm -ql *имя\_пакета* ". Он решил проверить, есть ли в его системе уже известный ему по предыдущим лекциям *пакет* coreutils и узнать, какие утилиты ему принадлежат:

methody@localhost:~ $ rpm -qa | grep coreutils

coreutils-5.2.1-some5

methody@localhost:~ $ rpm -ql coreutils | grep bin

/bin/basename

/bin/cat

/bin/chgrp

/bin/chmod

. . .

Пример 13.3. Какие файлы принадлежат пакету?

Мефодий получил довольно длинный список имен файлов, среди которых встретил многие из уже знакомых ему утилит и кое-какие незнакомые. Причем запрашивая список файлов, Мефодий использовал только *имя пакета*, **без версии** – rpm позволяет обращаться так к любому из установленных пакетов 3.

Можно выполнить и обратную процедуру – выяснить про любой файл, какому *пакету* он принадлежит:

methody@localhost:~ $ rpm -qf /etc/passwd

setup-2.2.5-some1

Пример 13.4. Какому пакету принадлежит файл?

Файлы, принадлежащие *пакету*, могут быть не только удалены, но и изменены. Это может быть сделано сознательно (например, отредактирован конфигурационный файл), и в таком случае при обновлении или удалении *пакета* измененный файл нужно сохранить, потому что в нем содержится результат работы, проделанной администратором. Для этого система должна уметь определять, что принадлежащий *пакету* файл изменился. Для этого в *пакете* должна храниться информация обо всех файлах архива: размер, атрибуты, *контрольная сумма* – в этом случае процедура проверки сможет убедиться в соответствии атрибутов файла в *пакете* атрибутам установленного в системе файла. Мефодий может проверить, что изменилось в *пакете* командой "rpm -V *имя\_пакета* ":

methody@localhost:~ $ rpm -V setup

S.5....T c /etc/X11/fs/config

S.5....T c /etc/exports

S.5....T c /etc/fstab

S.5....T c /etc/printcap

..?..... c /etc/securetty

Пример 13.5. Что изменилось в пакете?

Мефодий получил список изменившихся с момента установки *пакета* файлов. Видимо, все это – конфигурационные файлы, отредактированные администратором системы. Мефодий догадался, что комбинация цифр, букв и точек – это список атрибутов, по которым rpm сравнивал установленные файлы с данными *пакета*, однако чтобы разобраться, что именно означает каждая буква, ему придется внимательнее читать руководство.

Система Linux раскладывается на компоненты без остатка: каждый файл в Linux принадлежит какому-нибудь (и только одному!) *пакету* 4. Это позволяет свести любые изменения в составе системы к операциям над пакетами – от начальной установки до сложных *комплексных обновлений*. В этом случае для всех изменений будет использоваться одна и та же программа-установщик, например, rpm.

**Изменение настроек системы**

Для полноценной регистрации *пакета* в системе обычно недостаточно, чтобы система хранила список файлов, принадлежащих *пакету*. При установке, удалении или обновлении *пакета* часто требуется выполнить ряд операций, чтобы обновить сведения о *пакете*, адаптировать **настройки** – как самого *пакета* к уже имеющимся в системе, так и наоборот. К тому же некоторые изменения в системе, например, добавление и удаление *псевдопользователя*, не сводятся к добавлению и удалению файлов, и вдобавок зависят от **текущего состояния системы**. Получается, что *регистрация в системе* – дело не только системы, но и самого *пакета*. Поэтому в каждом *пакете* должны храниться сведения о том, какие действия следует выполнять в ходе различных операций с ним – в этом состоит **третья задача пакетирования**.

Списки необходимых действий представлены в *пакете* в виде **сценариев**. Эти сценарии привязаны к процедурам установки и удаления *пакета*, причем могут быть вызваны по необходимости как до, так и после соответствующей процедуры. В результате процедуры установки и удаления *пакета* выглядят так:

* выполнение предшествующих установке/удалению сценариев;
* копирование файлов из архива в систему или удаление файлов из системы;
* выполнение следующих за установкой/удалением сценариев.

methody@localhost:~ $ rpm -q --scripts coreutils

preinstall scriptlet (through /bin/sh):

# Remove info pages from fileutils, textutils and sh-utils.

for f in /usr/share/info/{fileutils,textutils,sh-utils}.info\*; do

[ -f "$f" ] || continue

RPM\_INSTALL\_ARG1=0 /usr/sbin/uninstall\_info "$f" ||:

done

postinstall scriptlet (through /bin/sh):

/usr/sbin/install\_info coreutils.info

preuninstall scriptlet (through /bin/sh):

/usr/sbin/uninstall\_info coreutils.info

Пример 13.6. Просмотр сценариев в пакете

Мефодий выяснил, что сценарии в *пакете* coreutils запускаются перед началом установки (preinstall), после установки (postinstall) и перед удалением (preuninstall), они выполняются стандартным командным интерпретатором ( /bin/sh ). Все эти сценарии нужны для того, чтобы зарегистрировать в системе info установленную *пакетом* документацию или удалить эту документацию из системы (командами /usr/sbin/install\_info и /usr/sbin/uninstall\_info соответственно). Поскольку info строит общее оглавление всей имеющейся в *системе документации*, простого копирования файлов было бы недостаточно.

В результате подобных операций по интеграции *пакета* в систему могут быть изменены или удалены файлы, не принадлежащие данному *пакету*, и созданы новые файлы. Если программа, содержащаяся в *пакете*,пользуется услугами какой-нибудь уже установленной службы (например, syslogd ), может понадобиться регистрация этой программы в конфигурационных файлах службы. Конечно, изменение "чужих" файлов в процессе установки *пакета* нежелательно: впоследствии, удаляя *пакет*, потребуется вернуть файл в исходное состояние, что не всегда возможно (например, после вдумчивого редактирования администратором). Избежать редактирования конфигурационных файлов позволяет схема ".d",описанная в лекции 10.

**Цена удобства**

Удобство, которое получает пользователь при работе с пакетами, не приходит само по себе, а достигается человеческим трудом: пакеты должен создавать человек, его работа называется "сопровождающий" ("package *maintainer*" или "packager"). В обязанности сопровождающего *пакет* входит подготовка файлового архива, необходимых для установки и удаления сценариев и прочей информации о *пакете* и его содержимом, и объединение их в одном файле-пакете 5. Узнать, кто и когда создал *пакет*, получить краткую справку о программном обеспечении, которое в нем содержится, можно с помощью команды "rpm -qi *имя\_пакета* ":

methody@localhost:~ $ rpm -qi setup

Name : setup Relocations: (not relocateable)

Version : 2.2.5 Vendor: Some Linux Team

Release : some1 Build Date: Чтв 29 Янв 2004 18:08:05

Install date: Пнд 23 Авг 2004 15:08:45 Build Host: shogun.somelinux.org

Group : Система/Настройка/Прочее Source RPM: setup-2.2.5-some1.src.rpm

Size : 39969 License: GPL

Packager : Leon B. Gourievitch

Summary : Initial set of configuration files

Description :

Initial set of configuration files to be placed into /etc.

Пример 13.7. Описание пакета

Нужно принимать во внимание, что любой *пакет*, содержащий программное обеспечение для Linux, не является универсальным. Если у вас есть такой *пакет*, это еще не означает, что его можно установить в вашей системе. Дело в том, что разные дистрибутивы Linux различаются именно тем, каким образом программное обеспечение организовано в **систему** (о дистрибутивах речь пойдет в лекции 18). Дистрибутивы могут различаться размещением файлов и процедурами, предусмотренными для интеграции в систему программного обеспечения, не говоря уже о том, что в разных дистрибутивах используется разный формат пакетов. Это значит, что *пакет*, подготовленный в расчете на один дистрибутив, может оказаться несовместимым с другим. Чтобы в вашем дистрибутиве появилась некоторая программа, кто-то должен подготовить и сделать доступным соответствующий *пакет*.

**Пакет**. Ресурсы, необходимые для установки и интеграции в систему некоторого компонента (архив файлов, до- и послеустановочные сценарии, информация о *пакете* и его сопровождающем), объединенные в одном файле.

Несмотря на некоторые различия, дистрибутивы Linux представляют собой варианты одной и той же системы, поэтому в конечном итоге любую программу, работающую в одном дистрибутиве, можно "приспособить" к любому другому. Только для этого нужно располагать исходными текстами соответствующей программы. До сих пор речь шла только о так называемых **двоичных пакетах**, в которых программы содержатся в виде уже скомпилированных двоичных (исполняемых) файлов, в таком виде программа может зависеть от некоторых свойств системы и работать не везде. Чтобы получить работающую программу в системе, нужно установить именно *двоичный пакет*. *Пакет*, впрочем, может содержать и исходные тексты программ, такие пакеты называются ***исходными*** . Доступность исходных кодов – обязательное условие распространения большей части программного обеспечения для Linux (см. лекцию 17). Если получилось так, что никто не подготовил *пакет* с нужной вам программой для вашего дистрибутива, можно установить *исходный пакет* и скомпилировать программу самостоятельно 6. При успешной компиляции из *исходного пакета* получается соответствующий двоичный, который уже можно установить в системе.

**Зависимости**

Мефодий нашел в *Internet* *пакет* с заинтересовавшей его программой в подходящем формате rpm и решил попробовать его установить 7:

[root@localhost RPMS.local]# rpm -i xsltproc-1.0.32-some1.i586.rpm

ошибка: неудовлетворенные зависимости:

libxslt = 1.0.32-some1 нужен для xsltproc-1.0.32-some1

[root@localhost RPMS.local]#

Пример 13.8. Пакет не установлен из-за неудовлетворенных зависимостей

Однако rpm отказался выполнять установку, ссылаясь на **зависимость** от другого *пакета*. Здесь Мефодий впервые столкнулся с тем, что пакеты – не всегда (точнее, почти никогда) бывают независимы от имеющейся системы. В разделе "*Архив* файлов" уже говорилось о том, что для работы программы нужны различные ресурсы, причем несколько программ могут нуждаться в одном и том же ресурсе. В последнем случае общий *ресурс* может оказаться в отдельном собственном *пакете* (чтобы не включать его сразу в несколько), и этот *пакет* должен быть установлен в системе, чтобы заработали нуждающиеся в нем программы. Потребность *пакета* в ресурсах, находящихся в другом *пакете*, называют **зависимостью** этого *пакета* от другого. В процедуре установки *rpm* проверяет,все ли зависимости устанавливаемого *пакета* **удовлетворены** (т. е. все ли необходимые пакеты уже установлены в системе), и если чего-то не хватает – прекращает установку. Именно с такой ситуацией и столкнулся Мефодий.

**Зависимость пакетов**. Ситуация, при которой *пакет* не может быть установлен в систему, если в ней не установлен хотя бы один из некоторого множества пакетов. Аналогично, *пакет* не может быть удален из системы до тех пор, пока в ней установлен хотя бы один зависящий от него *пакет* .

**Библиотеки**

Мефодию помешала установить *пакет* самая типичная зависимость– на *библиотеку*. Библиотеки возникают оттого, что все программы, как бы они ни отличались друг от друга, нуждаются в выполнении одних и тех же операций: вводе и выводе, получении доступа к ресурсам системы (памяти, процессорному времени, файлам), вычислениях, работе с сетью, рисовании окошек, кнопок, меню и т. п. Для выполнения таких операций используются небольшие подпрограммы – функции. Любые функции, необходимые более чем одной программе, есть смысл не включать в текст каждой программы, а собирать в отдельных библиотеках. Тогда программа сможет использовать не собственную подпрограмму, а готовую *функцию* из *библиотеки*. Поскольку библиотеки нужны нескольким программам, они обычно оформляются в виде отдельного *пакета*. Если *библиотека* не будет установлена, использующая ее программа просто не будет работать.

Библиотеки подвержены тем же изменениям с течением времени, что и все прочие программы: исправлению обнаруженных ошибок, модернизации, оптимизации и пр. Поэтому версии библиотек должны быть согласованы с версией программного обеспечения. Например, программа может отказаться работать даже при наличии *библиотеки*, если эта *библиотека* слишком старая либо слишком новая по сравнению с самой программой.

**Цепочки зависимостей**

Понятие зависимости включает не только зависимость программы от библиотек. Вообще говоря, зависимость возникает там, где программное обеспечение использует любой не поставляемый непосредственно с ним ресурс 8. Это могут быть и утилиты, которые запускаются при работе самой программы или во включенных в *пакет* сценариях, программа-интерпретатор для исполнения этих сценариев и даже определенные файлы, которые должны присутствовать для правильной работы программы (например, утилита passwd предполагает, что существует файл /etc/passwd ).

Зависимость может быть и небезусловной. Например, в некоторых случаях нужно обеспечить наличие ресурса не к моменту запуска программы, а прямо к моменту установки *пакета*. Так, для выполнения доустановочного *сценария* нужна программа-интерпретатор. В некоторых случаях требуется ресурс строго определенной версии – ни больше, ни меньше. Бывают случаи, когда зависимость имеет обобщенную форму, например, почтовому клиенту (программе для чтения и написания электронной почты) может требоваться служба доставки электронной почты.В Linux такую услугу предоставляют несколько разных программ, и любая из них удовлетворит зависимость.

Разобравшись с понятием зависимости, Мефодий набрался решимости установить-таки нужный ему *пакет*, установив все, что он потребует. Но не тут-то было: взявшись устанавливать библиотеки, Мефодий выяснил, что каждой из них требуются какие-то еще пакеты, отсутствующие в системе, у каждого из них тоже есть зависимости и т. п. – один-единственный *пакет* повлек за собой снежный ком других, вытягивая их по цепочкам зависимостей.

**Конфликты и альтернативы**

В противоположность зависимости, когда *пакет* не может быть установлен при отсутствии некоторого другого, *конфликт пакетов* – это ситуация, когда *пакет* не может быть установлен при **наличии** некоторого другого, т. е. они несовместимы в рамках одной системы. Одна из причин возникновения конфликтов уже упоминалась выше – в пакетах есть файлы с совпадающими именами. Самый распространенный источник конфликтов – программы, которые предоставляют разные реализации одной и той же функциональности системы (например, службы доставки электронной почты или печати, программы проверки орфографии, компиляторы и т. п.). Можно было бы, конечно, просто **назвать** конфликтующие файлы по-разному, но и тогда путаница неизбежна: если, допустим, старый компилятор Си называется gcc2.96, а новый – gcc3.3, то **что** запускается по стандартной команде gcc? В каждом *пакете* должна содержаться информация о том, с какими пакетами он конфликтует. *Конфликт пакетов* может быть разрешен очень просто: следует удалить один из конфликтных пакетов, после чего свободно устанавливать другой.

Каждый *пакет*, помимо имени, обозначен и номером версии, указывающим степень обновленности содержащегося в *пакете* программного обеспечения и самого *пакета*. В системе одновременно может быть установлена только одна версия любого *пакета*, со всеми остальными версиями она конфликтует. Такой подход вполне понятен, поскольку файлы в *пакете* имеют строго определенный путь, по которому они должны быть размещены в файловой системе. Поэтому при использовании пакетов не должно (и не может) возникнуть ситуации, когда одна и та же программа установлена в разных местах файловой системы.

Однако не все функции в системе должны эксклюзивно выполняться одной программой. Например, в системе может быть установлено сколько угодно текстовых редакторов и даже несколько разных реализаций одного редактора, например, Vi (*Vim* и NVi). Пакеты *Vim* и NVi не конфликтуют друг с другом, но оба с равным основанием могут быть вызваны по команде vi. Чтобы определить, какой именно из них запускать как vi, во многих дистрибутивах Linux (в частности, в том, который использует Мефодий) применяется механизм *альтернатив*. ***Альтернативы*** – это система символьных ссылок на принадлежащие пакетам файлы. Однотипные файлы из пакетов называются по-разному, а символьная ссылка, к которой обращается пользователь, указывает на один из них. Например, файл /usr/bin/vi будет символьной ссылкой либо на /usr/bin/*vim*, либо на /usr/bin/nvi (то же самое относится и к руководствам по этим редакторам). При установке и удалении любого из пакетов с одной из альтернативных программ символьная ссылка автоматически обновляется. На какую из них будет указывать ссылка, решается на основании веса каждого *пакета*. Вес – это условное число, выбирается та альтернатива из установленных, у которой наибольший вес. Пользователь может вмешаться в выбор *альтернатив* и вручную. Все необходимые утилиты для работы с *альтернативами* предоставляет *пакет* alternatives.

**Установщики пакетов**

Для выполнения всех операций над пакетами требуется специальная *программа* – *установщик пакетов*. В ее задачи входит весь цикл *работ* с *пакетом*: от создания *пакета* (компиляции *исходного пакета* в двоичный), до его установки, удаления, обновления, а также хранение и *вывод* *по* запросу пользователя или системы информации об установленных и неустановленных пакетах, принадлежащих им файлах и т. п.

В Linux формат пакетов не унифицирован, распространено несколько различных форматов, и для каждого из них требуется собственный *установщик пакетов*. Наиболее известны уже описанный rpm, dpkg, используемый в Debian (см. подробнее лекцию 18), а также пакеты в формате tgz (он же tar.gz – *файловый архив* tar, сжатый упаковщиком gzip, *GNU* *Zip*), то есть обычные файловые архивы, где вся необходимая в *пакете* метаинформация упакована в виде файлов наряду с файлами программного обеспечения. *Установщики пакетов* различаются не только форматом пакетов, с которыми они работают, но и кругом возможностей, внутренним форматом хранения информации и т. д.

**Установщик пакетов**. Программа, выполняющая основные операции с пакетами: установку, удаление, проверку, вывод информации о пакетах.

В рамках этой лекции мы ограничимся обсуждением только одного из установщиков пакетов – rpm (*Red* *Hat* *Package* *Manager*). Он первоначально возник в дистрибутиве RedHat, но в настоящее время используется и во многих других дистрибутивах. Пожалуй, сейчас его можно назвать самым распространенным форматом: авторы программ для Linux обычно выкладывают свои программы в *Internet* в виде файловых архивов tgz и пакетов rpm.

Обратной стороной популярности rpm является его нестандартность. Под расширением .rpm довольно редко оказывается канонический формат, разрабатываемый RedHat. В формате rpm усматривают много недостатков и недоделок, поэтому распространено множество усовершенствованных и дополненных версий rpm, и, соответственно, пакетов, ориентированных на какую-нибудь из этих версий, но носящих все то же расширение. На практике это означает, что разные версии rpm не полностью совместимы между собой, поэтому даже если в вашей системе используется rpm, из этого совершенно не следует, что вы сможете установить любой найденный в *Internet* *пакет* в этом формате.

Случай rpm – только самое яркое проявление более общей проблемы: в общем случае ни в одном дистрибутиве нельзя без потерь, помех или ручного вмешательства установить *пакет*, не разработанный **специально** для данного дистрибутива. В следующем разделе ( *"Менеджеры пакетов"* ) изложены некоторые соображения, почему это нежелательно, и почему следует *по* возможности пользоваться именно "родными" пакетами,а если их нет – создавать их самостоятельно.

Другая проблема установщиков пакетов заключается в том, что они годятся только для установки/удаления отдельных пакетов, но не предназначены для доставки пакетов в системы (*пользователь* сам должен найти и скачать нужный *пакет*, а также указать местоположение файла *пакета* установщику в командной строке). Кроме того, установщик работает с каждым *пакетом* *по* отдельности: он может указать, что не удовлетворены некоторые зависимости, или имеют *место* конфликты, но не в состоянии в ходе процедуры установки ни установить все необходимые пакеты *по* цепочке зависимостей, ни удалить конфликтующие – *пользователь* должен делать это вручную. *Установщики пакетов* не предоставляют также никаких средств *по* автоматизации обновления системы.

**Менеджеры пакетов**

*Установщики пакетов* делают атомарными (одношаговыми) *операции* с отдельными пакетами: вместо копирования *множества* файлов и запуска нескольких сценариев *пользователь* вводит одну команду "установить/удалить *пакет* ". Атомарная с точки зрения пользователя операция – добавление в систему **одного** нового компонента может состоять из нескольких (и даже многих) операций над пакетами. Мефодий уже столкнулся с подобным случаем, изучая на собственном опыте понятие "цепочка зависимостей". Здесь *установщики пакетов* никак не могут облегчить работу пользователя. Чтобы сделать процедуру установки, удаления и обновления **компонента системы** атомарной, были разработаны *менеджеры пакетов*. *Менеджер пакетов* – это *программа*, вычисляющая весь комплекс операций над отдельными пакетами, который нужно произвести для установки/удаления нового компонента ( *пакета* ), и сама запускает *установщик пакетов* необходимое количество раз с соответствующими параметрами. Кроме того, *менеджер пакетов* хранит информацию не только о пакетах, уже установленных в системе, но и обо всех, которые доступны для установки с какого-либо носителя или *по* Сети (подробнее об этом в разделе "Доставка").

**Менеджер пакетов**. Программа, выполняющая установку, удаление или обновление любого *пакета* или группы пакетов и автоматически выполняющая все необходимые для этого процедуры (доставку пакетов из удаленных репозиториев, вычисление зависимостей и установку требуемых по ним пакетов, удаление замещаемых пакетов и т. п.).

Наиболее известный и популярный *менеджер пакетов* называется *APT* (*Advanced* *Package* Tool). Первоначально он был разработан в рамках дистрибутива Debian и работал только с *установщиком пакетов* dpkg, впоследствии для других дистрибутивов была разработана версия, работающая с rpm. В дистрибутиве Мефодия также используется *APT*.

Чтобы установить *пакет*, прежде всего нужно узнать о его существовании. Пакетов для каждого дистрибутива Linux доступны тысячи и даже десятки тысяч, и ориентироваться в них непросто. *APT* предоставляет возможность поиска нужного среди доступных пакетов, для этого используется *утилита* *apt*-cache. В каждом *пакете* обязательно имеется краткая аннотация (в одну строку) и небольшое описание содержащихся в *пакете* ресурсов (не длиннее нескольких абзацев). *По* команде " *apt*-cache search **подстрока** " *APT* найдет и выведет *список* из имен и аннотаций пакетов, где в имени, аннотации или описании нашлась указанная *подстрока*:

[root@localhost shogun]# apt-cache search python | wc

146 1158 8994

[root@localhost shogun]# apt-cache search python | grep "programming"

python – An interpreted, interactive object-oriented programming language

Пример 13.9. Поиск пакетов в APT

Для установки и удаления пакетов предназначена *утилита* *apt*-get, а *команда* установки выглядит совсем просто: "*apt*-get install *имя\_пакета* ", причем не нужно указывать никаких сведений о версии и местонахождении *пакета*: *APT* сам найдет и установит самую последнюю из доступных версий:

[root@localhost shogun]# apt-get install python

Чтение списков пакетов... Завершено

Построение дерева зависимостей... Завершено

Следующие дополнительные пакеты будут установлены:

libpython libgdbm libgmp python-base python-modules python-modules-bsddb

python-modules-compiler python-modules-curses python-modules-email

python-modules-encodings python-modules-hotshot python-modules-logging

python-modules-xml python-strict

Следующие НОВЫЕ пакеты будут установлены:

libpython libgdbm libgmp python python-base python-modules

python-modules-bsddb python-modules-compiler python-modules-curses

python-modules-email python-modules-encodings python-modules-hotshot

python-modules-logging python-modules-xml python-strict

0 будет обновлено, 15 новых установлено, 0 пакетов будет удалено и 0 не

будет обновлено.

Необходимо получить 0B/4466kB архивов.

После распаковки потребуется дополнительно 16,9MB дискового пространства.

Продолжить? [Y/n] y

Получено: 1 cdrom://SomeLinux CD RPM/main libpython 2.3.3-some2 [17,4kB]

Получено: 2 cdrom://SomeLinux CD RPM/main libgdbm 1.8.3-some3 [25,6kB]

Получено: 3 cdrom://SomeLinux CD RPM/main libgmp 4.1.2-some3 [153kB]

. . .

Получено: 14 cdrom://SomeLinux CD RPM/main python-base 2.3.3-some12 [782kB]

Получено: 15 cdrom://SomeLinux CD RPM/main python 2.3.3-some12 [11,5kB]

Получено 4466kB за 0s (19,5MB/s).

Совершаем изменения...

Preparing... ######################################### [100%]

1: libpython ######################################### [ 6%]

2: libgdbm ######################################### [ 13%]

3: libgmp ######################################### [ 20%]

4: python-base ######################################### [ 26%]

. . .

13: python-modules-logging ######################################### [ 86%]

Завершено.

Пример 13.10. Установка пакета с помощью APT

Процедуру установки *APT* выполняет в несколько этапов: сначала он ищет запрошенный *пакет* в списках доступных, найдя, рассчитывает, какие пакеты следует установить, чтобы удовлетворить его зависимости, после чего получает файлы всех нужных пакетов (в данном случае *APT* нашел нужные пакеты на диске *CD-ROM*) и запускает *установщик пакетов* последовательно для установки всего необходимого. Аналогично, чтобы удалить *пакет*, достаточно выполнить команду "*apt*-get remove *имя\_пакета* ".

Кроме *APT*, есть еще несколько *менеджеров пакетов*. Большинство из них специфичны для определенного дистрибутива, как, например, emerge для Gentoo или YaST для SuSE. Их задачи и возможности примерно совпадают с *APT*.

**Контроль целостности**

Поскольку *менеджер пакетов* умеет строить цепочки *зависимостей пакетов* друг от друга, с его помощью всегда можно определить, все ли зависимости удовлетворены у пакетов, установленных в системе. Система,в которой нет пакетов с неудовлетворенными зависимостями, называется ***целостной*** . Если *целостность* нарушена, это означает, что часть установленного в системе программного обеспечения попросту неработоспособна или работает некорректно.

*Целостность* системы может нарушиться в момент каких-то изменений в ее составе: при установке, удалении или обновлении части пакетов или всей системы. Если для всех этих операций использовать *менеджер пакетов*, то *целостность* системы не должна нарушиться. Хотя иногда даже *менеджеру пакетов* бывает сложно найти правильное решение, чтобы удовлетворить все зависимости и устранить конфликты. При наличии *менеджера пакетов* механизм зависимостей можно обернуть и на пользу человеку. Так, можно создать *пакет*, в котором есть только зависимости и нет никаких ресурсов – такой *пакет* называется ***виртуальным*** . Это бывает полезно в том случае, когда нужно упростить пользователю установку полной среды для выполнения какой-либо задачи. Необходимые для этого пакеты могут напрямую не зависеть друг от друга, но чтобы установить их все за один шаг, пользователю будет достаточно установить один – *виртуальный – пакет*. Таким *виртуальным пакетом* оказался сам *пакет* python в примере, и еще один – python-strict:

[root@localhost shogun]# rpm -ql python

(не содержит файлов)

[root@localhost shogun]# rpm -ql python-strict

(не содержит файлов)

Пример 13.11. Виртуальные пакеты не содержат файлов

Именно поэтому *apt* "получил" 15 пакетов (включая два виртуальных), а "совершил изменения" только для 13-ти.

**Доставка**

Важная задача, которую не решает *установщик пакетов* – доставка файла *пакета* в систему для последующей установки. Архивы пакетов обычно не хранятся в самой системе: они слишком велики (тысячи пакетов) и должны регулярно обновляться (выход обновлений программ, т. е. новых версий пакетов). Поэтому для установки обычно требуется сначала скопировать необходимые файлы с того носителя, где они хранятся (это либо установочные диски дистрибутива, либо хранилища в сети Интернет).

Чтобы *APT* мог работать с пакетами, они должны содержаться в организованном по специальным правилам хранилище – **репозитории**. Список доступных *APT* репозиториев хранится в файле /etc/*apt*/sources.list, для каждого репозитория указан способ доступа (например, "cdrom:", "ftp:", "file:" и др.) и адрес:

rpm cdrom:[SomeLinux CD]/ RPM contrib main

rpm [sme] ftp://updates.somelinux.com 2.4/i586 updates

Пример 13.12. Файл sources.list

После каждого изменения файла /etc/*apt*/sources.list нужно обновлять кеш *APT*, в котором хранятся сведения о доступных пакетах,командой *apt*-get update. Для того чтобы добавить в кеш информацию о пакетах, доступных на CD, следует использовать команду "*apt*-cdrom add", а не редактировать sources.list вручную.

*APT* позволяет и просто доставить *пакет* в систему, не устанавливая его. Так, например, всегда происходит с *исходными пакетами*, которые просто копируются из репозитория в определенный каталог системы по команде "*apt*-get source *имя\_пакета* ".

**Обновление**

Программное обеспечение в мире Linux (и не только) постоянно обновляется: исправляются ошибки, расширяются возможности. Разработчики каждого дистрибутива по мере выхода новых версий программ готовят новые версии соответствующих пакетов и делают их доступными в своем репозитории (репозитории, отражающие наиболее современное состояние программного обеспечения, доступны через Internet). Пользователю имеет смысл не отставать от обновлений программного обеспечения, потому что новые версии программ – это и большая надежность работы системы, и новые возможности.

*Менеджеры пакетов* позволяют выполнять *комплексные обновления* всей системы. В *APT* эту процедуру можно выполнить одной командой: "*apt*-get dist-upgrade". Эта процедура сначала исследует содержимое всех доступных репозиториев и находит там все пакеты более поздних версий, чем соответствующие пакеты, установленные в системе. После этого вычисляется объем обновления: должна быть удалена связанная область зависящих друг от друга устаревших пакетов и заменена соответствующей областью более новых версий. Сложные ситуации могут возникать в том случае, если изменилось распределение ресурсов по пакетам: пакеты были разделены или объединены – здесь может потребоваться вмешательство пользователя.

Тот род обновлений системы, который нужно выполнять регулярно и **обязательно** – это обновления, связанные с безопасностью (security updates). Когда в программе обнаруживают и исправляют серьезные ошибки, угрожающие безопасности всей системы, разработчики дистрибутивов обычно заботятся о том, чтобы соответствующие обновления дошли до пользователя. Обычно присутствует отдельный репозиторий с обновлениями, существенными для безопасности.

**Цена удобства**

Удобство *менеджеров пакетов* оплачивается тем, что они могут успешно работать только со специальными целостными областями источников ( *репозиториями пакетов* ). Хотя для большинства пользователей это ограничение не так существенно: те дистрибутивы, в которых используются *менеджеры пакетов*, обычно имеют огромные *репозитории пакетов*, где можно найти любое мыслимое и немыслимое программное обеспечение. Если же нужной программы все-таки нет в официальном репозитории дистрибутива, обычно находятся "частные" репозитории, доступные по Internet, включающие не вошедшие в официальный *репозиторий пакетов*.

Если все-таки нужный вам пакет нигде не найти собранным именно для вашего дистрибутива, можно установить и сторонний пакет, но это может быть сделано только при помощи *установщика пакетов* – *менеджер пакетов* в такой ситуации будет бесполезен. Можно установить программу, скомпилировав ее из исходных текстов самостоятельно.

Автор **программы** совершенно не обязан учитывать в ней особенности всех дистрибутивов, поэтому возможны, с одной стороны, прямые конфликты с файлами в системе (которые никто уже не отследит), а с другой стороны – конфликты и противоречия скрытые (например, программа устанавливается в подкаталог каталога /usr/local и предполагает, что все остальные программы тоже находятся в этом каталоге). Это значит, что придется самостоятельно разобраться и с тем, как и с какими параметрами компилировать программу, как устанавливать ее в систему, и как избежать при этом конфликтов. А если так, если вы и в самом деле в состоянии правильно собрать и установить в систему нужную вам, а значит и еще кому-нибудь, программу, которой пока нет в дистрибутиве, то самое правильное – сделать из нее, по крайней мере, *исходный пакет*, а если получится, то и **двоичный**. Это здорово облегчит жизнь и вам, когда вы будете компилировать и устанавливать эту программу еще раз (на другом компьютере или обновляя версию самой программы), и, главное, **всему сообществу пользователей** вашего дистрибутива!

Наконец, во многие современные дистрибутивы включаются средства, помогающие в сборке двоичных пакетов. Такие средства (например, *пакет* hasher из ALT Linux) позволяют не только скомпилировать программу в "универсальной среде", содержащей лишь заданный набор пакетов, но и автоматически выстраивают зависимости, проверяют правильность установки, отслеживают конфликты. Собрав *пакет* с помощью такого средства, вы можете стать серьезным претендентом на роль сопровождающего этот *пакет* в дистрибутиве. Напротив, скомпилировав программу втихомолку, с помощью шаманства и ручной работы, вы проявите себя как лентяй и эгоист, которому нет дела до совершенствования собственной операционной системы.

**Лекция 14:**

**Графический интерфейс (X11)**

**Графический интерфейс в Linux**

На протяжении предыдущих лекций Мефодию ни разу не потребовалось для выполнения своих задач покидать пределы текстового терминала. Что и понятно: в основном он занимался освоением самой **системы** Linux, а главные средства управления ею – *командная строка* и *текстовый редактор*. Тем не менее, для современного пользователя персональный (да и вообще любой) *компьютер* – это не в последнюю *очередь* устройство с широкими графическими возможностями, и часть задач, которые должен выполнять *компьютер*, – непосредственно графической природы, например, показ фильмов или создание изображений. Но такими специфическими задачами использование графического интерфейса не ограничивается.

Графические средства ввода-вывода позволяют организовать *интерфейс*, принципиально отличающийся от терминала – **оконный**. Сегодня любому пользователю компьютера знакома такая модель организации графического интерфейса: *окна*, *меню*, кнопки. *Оконный интерфейс* позволяет использовать *пространство* *экрана* гораздо более эффективно, чем обыкновенный текстовый *терминал* на *виртуальной консоли*: здесь одновременно можно открыть несколько окон, относящихся к разным задачам, и наблюдать за их работой одновременно. Собственно, в рамках *окна* может выполняться любая задача, в частности, текстовый *терминал*. При помощи *оконного интерфейса* *пользователь* Linux может следить за несколькими задачами на разных терминалах одновременно, а не по очереди.

**Оконный интерфейс**. Модель интерфейса, в которой пространством ресурсов является *экран* – прямоугольная область, в которой организуется ввод и вывод. Субъектами в *оконном интерфейсе* выступают задачи, вводящие и выводящие данные в рамках *окна* – области в рамках *экрана* .

Однако все задачи управления системой в Linux решаются посредством текстового терминала, да и очень многие задачи пользователя – как заметил Мефодий даже по своему небольшому опыту – тоже, поэтому никакой системной **необходимости** в графических средствах ввода-вывода в Linux нет. Графический *интерфейс* в Linux – это отдельная задача, наподобие *системной службы* или *демона*, поэтому в некоторых системах *программное обеспечение* для организации графического интерфейса1 может вовсе отсутствовать. Такая задача получает единоличный *доступ* к устройству графического вывода (видеокарта), а программам, использующим графические ресурсы, она предоставляет **объектную модель** графических примитивов (функции рисования линий, прямоугольников, отображения цвета и т. п.), наподобие того, как *ядро* предоставляет *доступ* к ресурсам жесткого диска в терминах объектной модели файловой системы. Поэтому весь комплекс программ для работы с графическими устройствами принято называть *графической подсистемой*.

Пользователю домашнего настольного компьютера графический *интерфейс* почти наверняка понадобится при каждом сеансе работы. Можно настроить систему таким образом, чтобы процесс начальной загрузки завершался запуском *графической подсистемы*, так что даже *регистрация* пользователей будет происходить уже в графическом режиме при помощи специальной программы – *экранного диспетчера* (см. лекцию 10). *Экранный диспетчер* опознать очень просто: он всегда отображает *окно* с приглашением к регистрации login: и password:, которое может быть оформлено и минималистично, и с барочной пышностью. После регистрации в *экранном диспетчере* пользователю предоставляется сразу и *доступ* к системе, и *доступ* к *графической подсистеме*.

Однако ни в одной из систем, в которых работает Мефодий, ему не случалось встречаться с *экранным диспетчером*, и всюду он регистрировался в системе и работал только в текстовом режиме на *виртуальной консоли*. Поскольку *графическая подсистема* – отдельная задача, авторизованный *пользователь* может запустить ее из командной строки в любой момент2: для этого используется *команда* startx, которую Мефодий и исполнил (рис. 16.1).



**Рис. 16.1.**Запуск графической подсистемы из командной строки

В некоторое недоумение ввел Мефодия предложенный ему выбор из нескольких кнопок. Проконсультировавшись у Гуревича, он выяснил, что каждая из кнопок соответствует программе, по-своему организующей графический *интерфейс*, и что он может попробовать их все по очереди и выбрать ту, которая будет наиболее подходящей для его стиля работы. Не мудрствуя лукаво, Мефодий нажал на первую же кнопку, "KDE" (см. рис. 16.2).

После некоторого ожидания на мониторе возникло все то, что Мефодий ожидал увидеть в графическом интерфейсе: иконки, панель с кнопками внизу *экрана*, *меню*. Однако если бы после запуска startx Мефодий выбрал другую кнопку вместо "KDE", графический *интерфейс* предстал бы перед ним совсем в другом виде и предоставлял бы несколько другие возможности и приемы работы. Далее в лекции объясняется устройство *графической подсистемы* в Linux. Из этого объяснения станет понятно, почему процесс запуска графического интерфейса оказался двухступенчатым и почему работа с графическим интерфейсом в Linux может быть организована по-разному.



**Рис. 16.2.**Запуск KDE