

# **Введение в ОС Linux**

Г. Курячий К. Маслинский

Москва 2008 г.

## Оглавление

Введение в ОС Linux.....	1
Предисловие.....	9
Структура курса.....	9
Примеры.....	9
Цель.....	10
Предварительные знания.....	10
1. Сеанс работы в Linux.....	11
Пользователи системы.....	11
Многопользовательская модель разграничения доступа.....	11
Учётные записи.....	13
Идентификатор пользователя.....	13
Идентификатор группы.....	14
Полное имя.....	14
Домашний каталог.....	14
Командная оболочка.....	14
Понятие "администратор".....	15
Регистрация в системе.....	15
Идентификация (authentication).....	17
Смена пароля.....	18
Одновременный доступ к системе.....	20
Виртуальные консоли.....	20
Графические консоли.....	21
Простейшие команды.....	22
Выход из системы.....	24
2. Терминал и командная строка.....	26
Терминал.....	26
Командная строка.....	29
Подсистема помощи.....	30
Страницы руководства (man).....	31
Info.....	34
RTFM.....	35
Ключи.....	37
Однобуквенные ключи.....	37
Полнословные ключи.....	39
Интерпретатор командной строки (shell).....	40
Команды и утилиты.....	41
Слова и разделители.....	43
3. Структура файловой системы.....	45
Организация файловой системы.....	45
Файл.....	45
Система файлов: каталоги.....	45
Имена файлов и каталогов.....	47
Допустимые имена.....	47
Кодировки и русские имена.....	47
Расширения.....	48
Дерево каталогов.....	49
Размещение компонентов системы: Стандарт FHS.....	51

4. Работа с файловой системой.....	56
Текущий каталог.....	56
Домашний каталог.....	58
Информация о каталоге.....	58
Перемещение по дереву каталогов.....	60
Создание каталогов.....	62
Копирование и перемещение файлов.....	62
Файл и его имена: ссылки.....	64
Жёсткие ссылки.....	64
Индексные дескрипторы.....	65
Символьные ссылки.....	66
Создание символьных ссылок.....	67
Удаление файлов и каталогов.....	67
5. Доступ процессов к файлам и каталогам.....	70
Процессы.....	70
Запуск дочерних процессов.....	71
Сигналы.....	74
Доступ к файлу и каталогу.....	76
Чтение, запись и использование.....	77
Доступ к файлу.....	77
Сценарий.....	78
Доступ к каталогу.....	80
6. Права доступа.....	83
Права доступа.....	83
Идентификатор пользователя.....	83
Идентификатор группы.....	83
Ярлыки объектов файловой системы.....	84
Разделяемые каталоги.....	86
Суперпользователь.....	88
Подмена идентификатора.....	89
Восьмеричное представление атрибутов.....	90
7. Работа с текстовыми данными.....	92
Ввод и вывод.....	92
Перенаправление ввода и вывода.....	93
Стандартный вывод.....	94
Стандартный ввод.....	95
Стандартный вывод ошибок.....	96
Перенаправление в никуда.....	98
Обработка данных в потоке.....	98
Конвейер.....	98
Фильтры.....	99
Структурные единицы текста.....	100
Примеры задач.....	101
Подсчёт.....	101
Отбрасывание ненужного.....	102
Выбор нужного.....	103
Поиск.....	103
Поиск по регулярному выражению.....	103

Замены.....	105
Упорядочение.....	106
Запуск команд.....	107
8. Доступ процессов к файлам и каталогам.....	109
Редактирование ввода.....	109
Редактирование командной строки.....	111
История команд.....	111
Сокращения.....	112
Достраивание.....	114
Генерация имён файлов.....	114
Шаблоны.....	115
Использование шаблонов.....	116
Окружение.....	118
Работа с переменными в shell.....	119
Переменные окружения, используемые системой и командным интерпретатором.....	120
Язык программирования sh.....	122
Интеграция процессов.....	122
Сценарии.....	124
Настройка командного интерпретатора.....	126
Привязка к клавишам.....	126
Стартовые сценарии.....	127
Пример настроек.....	127
9 Текстовые редакторы.....	130
Задача текстовых редакторов.....	130
Vi и лучше, чем Vi.....	131
Запуск vi.....	132
Режимы.....	133
Работа с файлами.....	135
Перемещение по тексту.....	136
Изменение текста.....	138
Работа с фрагментами текста.....	139
Настройка Vi и Vim.....	141
Лучше, чем Emacs?.....	141
Тексты на разных языках.....	142
Команды Emacs.....	143
Работа с файлами.....	144
Перемещение по тексту.....	145
Изменение текста.....	146
Работа с фрагментами текста.....	146
Настройка Emacs.....	147
Просто текстовые редакторы.....	147
10. Этапы загрузки системы.....	149
Досистемная загрузка.....	149
Загрузчик в ПЗУ.....	149
Загрузочный сектор и первичный загрузчик.....	150
Загрузчик ядра.....	151
Досистемная загрузка Linux.....	153
LILO.....	153

GRUB.....	155
Действия ядра Linux в процессе начальной загрузки.....	156
Загрузка системы.....	158
Стартовый виртуальный диск и модули ядра.....	158
Отец всех процессов.....	161
Запуск системных служб.....	161
Чему служат демоны?.....	161
Стартовый сценарий системной службы.....	162
Схема ". d".....	163
Уровни выполнения.....	164
Загрузка типичной системы на уровень выполнения 5.....	167
Останов системы.....	168
11. Работа с внешними устройствами.....	171
Представление устройства в системе.....	171
Файлы-дырки и другие типы файлов.....	172
Драйверы устройств.....	174
Работа с устройствами.....	174
Права доступа к устройствам.....	175
Разметка диска и именование устройств.....	176
Разметка диска IBM-совместимого компьютера.....	177
Совмещение нескольких схем разметки.....	177
Область подкачки.....	179
Файловая система.....	179
Принципы организации данных на диске.....	180
Работа с файловыми системами.....	181
Монтирование и размонтирование.....	181
Поддерживаемые Linux файловые системы.....	184
Виртуальные и сетевые файловые системы.....	185
Проверка файловой системы.....	187
12. Конфигурационные файлы.....	190
Проектирование свойств системы.....	190
Профиль системы.....	191
Конфигурационный файл.....	193
Изменение конфигурационных файлов.....	197
Системные конфигурационные файлы.....	199
Подсистема учётных записей.....	199
Подсистема идентификации.....	201
Подсистема системных журналов.....	203
Выполнение действий по расписанию.....	204
"Прокручивание" системных журналов.....	205
Конфигурационные файлы в домашнем каталоге.....	206
13. Управление пакетами.....	209
Пакеты.....	209
Архив файлов.....	209
Формат пакета.....	211
Регистрация в системе.....	211
Изменение настроек системы.....	213
Цена удобства.....	214

Зависимости.....	215
Библиотеки.....	216
Цепочки зависимостей.....	216
Конфликты и альтернативы.....	217
Установщики пакетов.....	218
Менеджеры пакетов.....	219
Контроль целостности.....	221
Доставка.....	222
Обновление.....	222
Цена удобства.....	223
14. Сеть TCP/IP в Linux.....	225
Сетевые протоколы. Семейство протоколов TCP/IP.....	225
Аппаратный и интерфейсный уровни.....	228
Сетевой уровень.....	230
Адресация.....	230
Маршрутизация.....	232
Служебный протокол ICMP.....	234
Транспортный уровень.....	235
Прикладной уровень.....	236
Клиент-серверная модель.....	236
Обслуживание прикладного уровня в Linux.....	237
Служба доменных имён.....	239
15. Сетевые и серверные возможности.....	244
Настройка сети.....	244
Настройка вручную.....	244
Настройка при установке или загрузке системы.....	246
Автоматическая настройка.....	248
Настройка соединений "точка--точка".....	250
Межсетевой экран.....	252
Фильтрация.....	254
Подмена адресов.....	255
Сетевые службы.....	257
HTTP.....	258
FTP.....	260
Терминальный доступ.....	262
Пересылка почты.....	263
Доступ к почтовым ящикам.....	265
16. Графический интерфейс (X11).....	267
Графический интерфейс в Linux.....	267
X Window System.....	271
X-сервер и X-клиенты. Протокол X11.....	271
DISPLAY.....	274
Виртуальный сервер.....	276
XFree86 и X.Org.....	277
Конфигурация X-сервера.....	278
Модули и расширения.....	279
X-приложения.....	280
Эмулятор терминала.....	280

Сеанс работы с X11.....	281
Ресурсы X11.....	283
Диспетчер окон.....	284
Задача диспетчера окон.....	284
Работа с окнами.....	285
Удобства.....	286
Настройка диспетчера окон.....	288
Рабочий стол.....	288
17. Прикладные программы.....	293
Рабочий стол.....	294
Эмулятор терминала.....	294
Диспетчеры файлов.....	295
Сеть.....	296
WWW-броузеры.....	296
Почтовые программы.....	297
Обмен сообщениями.....	298
Офисные программы.....	299
Графика.....	300
Векторная графика.....	300
Работа с PostScript и PDF.....	300
Диаграммы.....	301
Плакатная графика.....	302
Растровая графика.....	302
Трёхмерная графика.....	302
Мультимедиа.....	303
Музыкальные шкатулки.....	303
Музыкальные редакторы.....	304
Нотные редакторы.....	304
Редакторы и фильтры оцифрованного звука.....	305
Видеопроеигрыватели.....	306
Видеоредакторы и конвертеры.....	306
Запись CD и DVD.....	307
Издательские системы.....	307
Нельзя объять необъятного.....	308
18. Политика свободного лицензирования. История Linux: от ядра к дистрибутивам.....	310
История возникновения свободного ПО.....	310
Разработка ПО как научное исследование.....	310
ПО как "патентованный" продукт.....	311
Появление свободного ПО.....	312
Определение свободного ПО.....	313
Общественная лицензия GNU.....	314
Сообщество разработчиков и пользователей.....	316
История Linux.....	320
GNU без Linux.....	320
Linux -- ядро.....	321
GNU и Linux.....	322
Возникновение дистрибутивов.....	323
История Linux в России.....	325





# Предисловие

## Структура курса

В курсе даются основные понятия операционной системы Linux и важнейшие навыки работы в ней. Изложение сопровождается большим количеством практических примеров. Данный курс может рассматриваться как учебник для студентов, начинающих обучение по специальностям в области информатики и ещё не знакомых с ОС Linux. Он состоит из двух основных частей:

1. В первой части вводятся основные понятия и навыки, необходимые пользователю для того, чтобы начать грамотно работать в Linux. Здесь рассмотрены: пользователи с точки зрения системы, понятие терминал и работа с командной строкой, устройства файловой системы и работа с ней, права доступа в Linux, возможности командной оболочки, текстовые редакторы.

2. Вторая часть посвящена тем понятиям и навыкам, которые требуются для администрирования ОС Linux. Сюда входит обсуждение этапов загрузки системы, технологий работы с внешними устройствами, файловыми системами и сетью в Linux, администрирование системы посредством конфигурационных файлов, управление пакетами.

В завершающей лекции курса даётся обзор истории возникновения и развития Linux. Здесь же приведён обзор социального контекста, существенного для понимания ОС Linux и работы в ней: сообщество пользователей, лицензирование свободного программного обеспечения, место свободного ПО на современном рынке, дистрибутивы Linux и решения на базе Linux.

## Примеры

Теоретическое изложение материала перемежается практическими примерами: показаны конкретные действия пользователя и их результат. Наиболее эффективный способ освоить материал курса -- по ходу чтения лекций выполнять все примеры самостоятельно. Для этого потребуется доступ к установленному дистрибутиву Linux. Примеры подобраны с таким расчётом, чтобы результат был одинаковым в любом современном дистрибутиве.

В примерах действует один условный пользователь, работающий "в одном и том же месте": все созданные им файлы сохраняются и используются в последующих лекциях. Он совершает типичные ошибки или, наоборот, делает всё правильно.

Примеры набраны моноширинным шрифтом (типа "пишущая машинка") и по возможности точно воспроизводят то, что пользователь видит на экране монитора (иногда с некоторыми купюрами). Команды, которые должен вводить пользователь, в примерах следуют за "приглашением" (оно завершается знаком "\$"): все символы от "\$" до конца строки -- и есть ввод пользователя.

Названия объектов системы (имена файлов, программ и т. п.), встречающиеся в тексте, также набраны моноширинным шрифтом, их можно в неизменном виде вводить в качестве команд и т. п. Однако иногда такие строки для удобства чтения заключены в кавычки -- в этом случае вводить кавычки не нужно.

## **Цель**

Курс адресован студентам, начинающим обучение по специальностям в области информационных технологий, а также всем любознательным пользователям, желающим научиться грамотно и эффективно работать в Linux. Изучение курса не требует специальных знаний в области программирования.

## **Предварительные знания**

Рекомендуемый уровень предварительной подготовки:

1. знакомство с устройством компьютера на уровне пользователя;
2. знакомство с устройством и принципом работы TCP/IP-сетей.

# 1. Сеанс работы в Linux

## Пользователи системы

Между включением питания компьютера и моментом, когда система 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 всё в порядке и не возникло никаких ошибок, загрузка завершится выводом на экран приглашения "login:". Оно может быть оформлено по-разному, в зависимости от настройки системы оно 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 "незнакомому" Linux откажется.

## Многопользовательская модель разграничения доступа

Процедура регистрации в системе *обязательна* для Linux, 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 необязательной, если пользователи компьютера доверяют друг другу, и особенно если у компьютера только один пользователь. Такая ситуация очень распространена на сегодняшний день, 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. Несколько раньше "компьютер" ассоциировался с огромным и дорогостоящим (занимавшем 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 чужих данных (файлов), выделять каждому машинное время (по возможности избежав простаивания), пространство на диске и при этом не допустить узурпирования всех ресурсов одним пользователем и его задачей, а равномерно делить ресурсы между всеми. Для такой системы принципиально важно знать, кому принадлежат задачи и файлы, поэтому и возникла

необходимость выдавать 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 неё Linux -- также многопользовательская система -- унаследовал 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, отвечают за которые *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 изменения этого инструмента и для работы с его помощью используются одни и те же операции: изменение файлов, выполнение программ. Получается, что, если не соблюдать осторожности, побочным результатом работы может стать выход из строя самой системы. 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 выполнение действий, изменяющих саму систему. Таким образом можно избежать повреждения системы в результате ошибки пользователя (нажал не ту кнопку) или ошибки в программе, или даже по злему умыслу (например, вредительской программой-вирусом). Полномочия администратора обычно не ограничены.

Для персонального компьютера, с которым работают несколько человек, довольно важно обеспечить каждому 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., доступ остальных пользователей к этому каталогу ограничивается.

Если компьютер подключён к глобальной или локальной сети, то вполне вероятно, что какую-то часть хранящихся на нем ресурсов имеет смысл сделать публичной и доступной по сети. Напротив, часть данных, скорее всего, делать публичными не следует (например, личную переписку). Ограничив публичный 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 сокращённом варианте это даёт целый ряд названных выше преимуществ.

## Учётные 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 -- **учётная запись**. Английский эквивалент термина **учётная запись** -- **account**, "счёт". Именно с учётными записями, а не с самими пользователями, и работает система. В действительности, соотношение учётных записей и пользователей-людей в Linux обычно не является однозначным: несколько человек могут использовать одну учётную запись -- система не может их различить. И в то же время в Linux имеются учётные записи для **системных пользователей**, от имени которых работают некоторые программы и которые не предназначены для работы людей.

**учётная запись**      Объект системы, при помощи которого Linux ведёт учёт работы пользователя в системе. Учётная запись содержит данные о пользователе, необходимые для регистрации в системе и дальнейшей работы с ней.

Учётные записи могут быть созданы во время установки системы или после установки. Подробно процедура создания учётных записей (добавления пользователей) описана в лекции [Конфигурационные файлы](#).

Главное для человека в учётной записи -- её название, **входное имя пользователя**. Именно о нём спрашивает система, выводя приглашение "log in:". Помимо входного имени в учётной записи содержатся некоторые сведения о пользователе, необходимые системе для работы с ним. Ниже приведён список этих сведений.

**входное имя**      Название учётной записи пользователя, которое нужно вводить при регистрации пользователя в системе.

## Идентификатор пользователя

Linux связывает **входное имя** с **идентификатором пользователя** в системе -- **UID** (User ID). **UID** -- это положительное целое число, по которому система и отслеживает пользователей(2). Обычно это число выбирается автоматически при регистрации учётной записи, однако оно не может быть совершенно произвольным. В Linux есть некоторые соглашения относительно того, каким типам пользователей могут быть выданы идентификаторы из того или иного диапазона. В частности, **UID** от "0" до "100" зарезервированы для **псевдопользователей**(3).

**идентификатор пользователя**      Уникальное число, однозначно идентифицирующее **учётную запись** пользователя в Linux. Таким числом снабжены все **процессы** Linux и все объекты **файловой системы**. Используется для персонального учёта действий пользователя и определения **прав доступа** к другим объектам системы

## Идентификатор группы

Кроме идентификационного номера пользователя с учётной 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 учётной записи пользователя обычно создаётся и группа, имя которой совпадает с **входным именем**(4), именно эта группа будет использоваться как группа по умолчанию для этого пользователя. Пользователь может входить более чем в одну группу, но в учётной записи указывается только номер группы по умолчанию.

## Полное имя

Помимо **входного имени** в учётной записи содержится и **полное имя** (имя и фамилия) использующего данную учётную 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 учётная запись.

## Домашний каталог

Файлы всех пользователей в Linux хранятся отдельно, 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 учётной записи, потому что именно с него начинает работу пользователь, зарегистрировавшийся в системе.

## Командная оболочка

Каждому пользователю нужно предоставить способ взаимодействовать с системой: передавать ей команды и получать её ответы. Для этой цели служит специальная программа -- **командная оболочка** (или **интерпретатор командной строки**), она должна быть запущена для каждого пользователя, зарегистрировавшегося в системе. Поскольку в Linux доступно несколько разных командных оболочек, в учётной записи указано, какую из командных оболочек нужно запустить для данного пользователя. Если специально не указывать командную оболочку при создании учётной записи, она будет назначена по умолчанию, вероятнее всего это будет `bash`.

## интерпретатор командной строки

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 ровно один пользователь, полномочия которого в системе принципиально отличаются от полномочий остальных пользователей -- это пользователь с идентификатором "0". Обычно учётная запись пользователя с UID=0 называется `root` (англ., "корень"). Пользователь `root` -- это "администратор" системы Linux, учётная запись для `root` обязательно присутствует в любой системе Linux, даже если в ней нет никаких других учётных 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, восстановлении после сбоев.

Именно `root` обладает достаточными полномочиями для создания новых учётных записей.

## Регистрация в системе

Вернёмся теперь к нашей загруженной операционной системе Linux, которая по-прежнему ожидает ответа на своё приглашение "login:". Если Ваша система настроена таким образом, что это приглашение оформлено в виде графического окна в центре экрана, нажмите комбинацию клавиш `Ctrl+Alt+F1` -- произойдёт переключение видеорежима и Вы увидите перед собой чёрный экран с примерно следующим текстом:

```
Welcome to Some Linux / tty1
```

```
localhost login:
```

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 очень по-разному(5), более того, графический режим может быть даже недоступен после установки системы без специальной настройки. Текстовый режим, напротив, доступен в любой реализации Linux и всегда выглядит практически одинаково. Именно поэтому все дальнейшие примеры и упражнения мы будем проделывать в текстовом режиме, возможностей которого будет достаточно для освоения всего излагаемого в курсе материала.

Первая строка в примере -- это просто приглашение, она носит информационный характер. Существует очень много различных реализаций Linux (существующие реализации будут обсуждаться в лекции [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)), и в каждом из них принят свой формат первой пригласительной строки, хотя почти наверняка там будет указано, с какой именно версией Linux Вы имеете дело, и, возможно, будут присутствовать ещё 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 подключён к локальной или глобальной сети, если же он ни с кем более не связан, это имя может быть любым. Обычно имя хоста определяется уже при установке системы, однако в нашем случае этого сделано не было, и используется вариант по умолчанию -- "localhost". Заканчивается эта строка собственно приглашением к регистрации в системе -- словом "login:".

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

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

```
Welcome to Some Linux / tty1
localhost login: Methody
```

```
Password:
Login incorrect
```

```
login:
```

#### 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 введённое им имя пользователя, можно заметить, что оно начинается с заглавной буквы, в то время как учётная 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]$
```

Успешная регистрация в системе

В этот раз регистрация прошла успешно, о чём 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 [Возможности командной оболочки](#)), 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 передано для исполнения системе.

## Идентификация (authentication)

Когда система выводит на экран приглашение командной строки после того, как правильно введены имя пользователя и пароль, это означает, что произошла **идентификация пользователя** (authentication, "проверка подлинности"). Пароль может показаться излишней сложностью, но у системы нет другого способа удостовериться, что за монитором находится именно тот человек, который имеет право на использование данной учётной 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 даёт уверенность, что к такой системе не получит доступ случайный человек, не имеющий права использовать её ресурсы и хранящуюся там информацию. Одновременно она даёт определённую гарантию безопасности от злонамеренного вмешательства: даже если навредить попытается пользователь, имеющий учётную запись, его действия будут зарегистрированы в системе (поскольку системе всегда известно, от имени какой учётной записи выполняются те или иные действия), и злоумышленника можно будет найти и остановить.

Для тех пользователей, кому процедура идентификации кажется утомительной и необязательной (например, единственным пользователям персональных компьютеров), существует возможность получить доступ к системе, минуя процедуру идентификации. Для этой цели служит 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 учётную запись `methody` для автоматического входа в систему. Однако у этого подхода есть свои минусы:

- Теряется возможность определить, кто, что и когда делал в системе, потому что все реальные пользователи работают с одной учётной записью, с точки зрения системы все они -- один и тот же пользователь.
- Вся личная информация этого пользователя становится "общественной".
- Пароль легко забывается (пароль всё равно есть у любого пользователя), потому что его не нужно вводить каждый день. При этом `autologin` даёт доступ только человеку, сидящему перед монитором и только к работе с графическим интерфейсом. Если же потребуется, например, скопировать файлы с Вашего компьютера по сети, пароль всё равно нужно будет 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 учётная запись была создана не самим пользователем, а администратором многопользовательской системы (скажем, администратором компьютерного класса), скорее всего был выбран тривиальный пароль с тем расчётом, 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 учётной записью не воспользуется никто, кроме Вас, есть смысл выбирать в качестве пароля неочевидные последовательности символов. В Linux нет серьёзных ограничений на длину пароля или входящие в него символы (в частности, использовать пробел *можно*), но нет смысла делать пароль слишком длинным -- сразу возрастает опасность его забыть. Надёжность паролю придаёт его *непредсказуемость*, а не длина. Например, пароль, представляющий собой Ваше имя или повторяющий название учётной 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:
```

### Смена пароля

Набрав в командной строке "passwd", Мефодий запустил программу passwd, которая предназначена именно для замены информации о пароле в учётной записи пользователя. Она вывела приглашение ввести текущий пароль ("Enter current password"), а затем, в ответ на правильно введённый пароль, предложила подсказку про грамотное составление пароля и даже вариант надёжного 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:
```

### Смена пароля (продолжение)

В данном случае операция не удалась, поскольку с точки зрения passwd пароль, придуманный Мефодием, оказался слишком простым(6). В следующий раз ему придётся ввести более сложный пароль. passwd запрашивает новый пароль дважды, чтобы удостовериться, что в первый раз не было сделано опечатки, если же всё в порядке, она выведет сообщение о том, что операция смены пароля прошла успешно, и завершит работу, вернув Мефодию приглашение командной строки.

```
Enter new password:
```

```
Re-type new password:
```

```
passwd: All authentication tokens updated successfully  
[methody@localhost methody]$
```

### Пароль изменён

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 Ваш пароль (причём не имеет значение, человек это сделал или злонамеренная программа) получит доступ к ресурсам системы ровно в том объёме, в котором он предоставляется Вам, сможет читать и удалять Ваши 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 [Права доступа](#)), в таком случае не менее важно, чтобы и его пароль был надёжным.

Пароль пользователя `root` изначально назначается при установке системы, однако он может быть изменён 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 одновременно нескольким пользователям. 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, что Мефодий хочет зарегистрироваться в системе ещё 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]$
```

Вторая виртуальная консоль

Мефодий ввёл свой новый пароль и получил приглашение командной строки, аналогичное тому, которое мы уже видели в предыдущих примерах. Нажав комбинацию клавиш `Alt+F1`, Мефодий вернётся 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 совершенно

эквивалентный доступ к системе, в любой из них можно выполнять любые доступные команды.

Наблюдательный Мефодий обратил внимание, что в последнем примере ([tty2](#)) 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 там программами. По умолчанию в Linux доступно не менее 6-ти 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 "tty*N*", где "*N*" -- номер виртуальной консоли.

**виртуальная консоль**      Виртуальные консоли -- 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 нажмёт *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 "method". 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 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* -- с большой вероятностью Вы увидите просто чёрный экран. Десятая виртуальная консоль поддерживается системой, однако чёрный экран означает, что для этой консоли не запущена никакая программа, поэтому воспользоваться её существованием не получится. Для каких именно консолей будет запущена программа *getty* -- определяется настройкой конкретной системы. Впоследствии эта настройка может быть изменена пользователем. О том, как это может быть сделано, речь пойдёт в лекции [Этапы загрузки системы](#).

## Графические консоли

Впрочем, как ни широки возможности текстового режима, Linux ими не ограничен. Подробно работа в графическом режиме будет разбираться в последующих лекциях (см. лекцию [Графический интерфейс \(X11\)](#)). 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 совершенно аналогична регистрации в текстовом режиме. С той разницей, что после **идентификации** пользователя (правильно введённого имени пользователя и пароля) Вы увидите не приглашение командной строки, а графическую рабочую среду. Как именно она будет выглядеть -- зависит от того, какую систему Вы используете, 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 параллельно в разных графических средах. Обычно в стандартно настроенной Linux-системе можно организовать не менее трёх 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 произведённых операциях, дополнительные вопросы к пользователю, сообщения об ошибках или просто молчаливое согласие выполнить следующую команду(7).

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]$
```

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 чём свидетельствует вновь появившееся **приглашение командной строки**. Программа `whoami` возвращает название учётной записи того пользователя, от имени которого она была выполнена. Эта команда полезна в системах, в которых работает много разных пользователей, чтобы не воспользоваться по ошибке чужой учётной 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]$
```

### Команда `who`

`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 регистрироваться со многих компьютеров и даже по Сети, программа `who` может быть очень полезна. Могло создаться впечатление, что `who` -- очень умная 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.

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

### Команда `last`

В этом примере Мефодий неожиданно обнаружил кроме себя самого неизвестных ему пользователей `cacheman` и `cyrus` -- он совершенно точно знает, что не создавал учётных записей с такими именами. Это **псевдопользователи** (или **системные пользователи**) -- специальные учётные записи, которые используются для работы 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 учётной 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 представлять Linux как закрытое помещение: чтобы начать работать нужно сначала *войти* в систему (зарегистрироваться, пройти процедуру идентификации), а после того, как работа закончена, нужно из системы *выйти*. В том случае, если в систему вошло несколько пользователей, каждый из них должен выйти, завершив работу, причём совершенно 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 отчёты 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 с момента регистрации до выхода называется **сеансом работы**. Причём 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 строке набрать команду `logout`.

```
[methody@localhost methody]$ logout  
Welcome to Some Linux / tty1  
localhost login:
```

### 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, ещё более "немногословный" способ сообщить системе, что пользователь хочет завершить текущий сеанс работы. Нажав *Alt+F2* Мефодий попадёт на вторую виртуальную консоль, где всё ещё открыт сеанс для пользователя "methody" и нажмёт сочетание клавиш *Ctrl+D*, чтобы прекратить и этот сеанс. Комбинация клавиш *Ctrl+D* приводит не к передаче компьютеру очередного символа, а к закрытию текущего **входного потока данных**. Грубо говоря, 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 совершенно аналогичен явному завершению командной оболочки командой `logout`.



(1) Вместо формального "зарегистрироваться в системе" обычно используют выражение "войти в систему". Операционная система представляется чем-то вроде замкнутого помещения, внутри которого можно оказаться, только успешно проникнув через "дверь" -- пройдя процедуру регистрации.

(2) Это может оказаться важным, например, в такой ситуации: учётную запись пользователя с именем `test` удалили из системы, а потом добавили снова. Однако с точки зрения системы это уже другой пользователь, потому что у него другой UID.

(3) Обычно Linux выдаёт нормальным пользователям UID, начиная с "500" или "1000".

(4) Как правило, численное значение GID в этом случае совпадает со значением UID.

(5) Разнообразие графических интерфейсов Linux гораздо выше, чем, например, в Windows, поэтому составить учебный курс, не ориентируясь специально на ту или иную версию, просто невозможно.

(6) В разных дистрибутивах Linux используется разные версии программы `passwd`, поэтому не всегда она будет столь придирчива, как в дистрибутиве Мефодия.

(7) Реплики в таком диалоге строго чередуются, а собеседники не могут говорить одновременно -- в естественном диалоге так никогда не происходит. Скорее это напоминает диалог в учебнике иностранного языка. Однако и в диалоге с Linux у собеседников есть возможность "перебить" друг друга -- об этом речь пойдёт в последующих лекциях.

(8) В некоторых Linux-системах эта программа может называться `lastlog`.

## 2. Терминал и командная строка

### Терминал

Как стало понятно из предыдущей лекции ([Сеанс работы в Linux](#)), основное средство общения с Linux -- системная клавиатура и экран монитора, работающий в **текстовом режиме**. Вводимый пользователем текст немедленно отображается на мониторе соответствующими буквами, 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 введённого 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
```

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 ввёл "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 определённом, 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 этой программы, а человеку -- роль автора. Кроме того, человек анализирует текст, получаемый от системы: запрошенную им информацию и т. н. **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 передаёт 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, посылки **сигналов** и т. п. Используется для взаимодействия пользователя и системы

Для приёма 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 это понадобится. С другой стороны, типичный способ управления системой в Linux -- работа в **командной строке** -- требует *построчного* режима работы, когда набранный текст передаётся компьютеру только после нажатия клавиши *Enter* (что соответствует символу конца строки). Размер такой строки в байтах предугадать, конечно, нельзя, поэтому терминал, работающий в построчном режиме, 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 ведётся не 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 предназначены 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 "a" обозначается "^A".

---

Так, для завершения работы программы *cat*, которая считывает построчно 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 again^[Dn
  Any Text again
^D
```

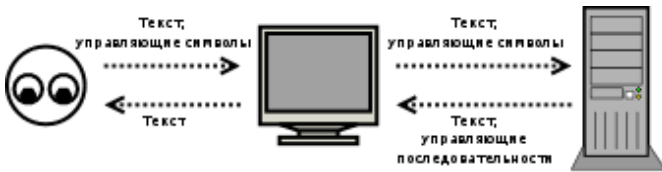
```
[methody@localhost methody]$
```

Как завершить работу `cat`?

Одну строчку вида "Any Tex..." Мефодий вводит с клавиатуры (что отображается на экране), и после того, как Мефодий нажмёт *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`". Эффект команды оказали одинаковый, а 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, можно продолжать работу дальше (поскольку программа `cat` больше ничего не делает, она завершается самостоятельно естественным путём). Можно считать, что "`^C`" -- это сокращение от "Cancel", а "`^D`" -- от "Done".

В пример не попало, как, набирая первый `cat`, Мефодий вновь ошибся и написал `ssat` вместо `cat`. Чтобы исправить положение, он воспользовался клавишами со стрелочками: с помощью клавиши *Стрелка влево* подвёл курсор к одному из "с" 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 терминала может быть так много разных нетекстовых клавиш, что на всех них не хватает ограниченного количества *разных* управляющих символов. Поэтому большинство нетекстовых клавиш возвращают т. н. **управляющую последовательность**, которая начинается управляющим символом (как правило -- `Esc`ре, т. е. "`^[`"), за которым следует строго определённое число обычных (для клавиши *Стрелка влево* -- "`[`" и "`D`").

То же самое можно сказать и о *выводе* управляющих последовательностей на терминал. Современный терминал умеет довольно много помимо простого вывода текста: перемещать курсор по всему экрану (чтобы вывести текст там), 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 again`^[ [Dn`" в выдаче `cat: "^[ [D`" при выводе на терминал перемещает курсор на одну позицию влево, так что было выведено "Any Text again", 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)



Интерфейс командной строки. Взаимодействие пользователя с компьютером посредством терминала.

Требования к терминалу как к точке входа пользователя в систему получаются весьма невысокими. Формально говоря, терминал должен удовлетворять трём 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 трёх точек зрения: во-первых, какую задачу решает пользователь (*что* он хочет от системы); во-вторых, *как* он формулирует задачу в доступном пониманию системы виде; и, 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 задаётся 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 её в том, что каждая строка, передаваемая пользователем системе, -- это **команда**, которую та должна выполнить. Пока не нажат *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!
```

## Команды echo и cal

Команда `cal` выводит календарь на текущий месяц, а команда `echo` просто выводит на терминал всё, что следовало в командной строке *после неё*. Получается, что одну и ту же команду можно использовать с разными **параметрами** (или *аргументами*), причём параметры эти изменяют поведение команды. Здесь Мефодий захотел посмотреть календарь за март 2005-го 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
```

## Команда cal с параметрами

В большинстве случаев при разборе командной строки *первое* слово считается именем команды, а остальные -- её параметрами. Более подробно о разборе командной строки и работе с ней рассказано в разделе [Интерпретатор командной строки \(shell\)](#) и в лекции [Возможности командной оболочки](#).

# 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 предубеждённые новички. Всем прочим *настоятельно* рекомендуется, 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 её устройства и уметь находить справочную информацию.

Эйнштейн говорил на этот счёт так: "Зачем запоминать то, что всегда можно посмотреть в справочнике?".

## Страницы руководства (man)

Больше всего *различной* полезной информации содержится в **страницах руководства (manpages)**. Каждая страница руководства (для краткости -- просто "руководство") 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:
  . . .
```

### 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 вперёд 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` даётся *формализованное* описание способов использования объекта (с данным 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 это 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 развёрнутое 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 нашёл 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).
. . .
```

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 обнаружили 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(2), что Мефодий немедленно применяет одну команду к имени другой, не заглядывая даже в документацию. Так *ни в коем случае* не следует делать! А что если запущенная Вами программа начнёт с того, что сотрёт 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
. . .
```

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` ищет только среди имён объектов (в *левых* частях полей NAME), а `apropos` -- по всей базе. В результате у `whatis` получается список кратких описаний объектов с *именами*, включающими в себя искомое слово, а у `apropos` -- список, в котором это слово *упоминается*. Для того, чтобы это узнать, всё 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 позволяют пользоваться 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
```



## Руководства с одинаковыми именами

Описания объектов, выводимые `what is`, отличаются числом в скобках -- номером раздела. В системе руководств Linux девять разделов, каждый из которых содержит страницы руководства к объектам определённого типа. Все разделы содержат по одному руководству с именем "intro", в котором в общем виде и на примерах рассказано, что за 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
```

## 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 [Работа с внешними устройствами](#)).
5. Форматы различных стандартных файлов системы (например **конфигурационных**).
6. Игры, безделушки и прочие вещи, не имеющие системной ценности.
7. Теоретические положения, договорённости и всё, что не может быть классифицировано.
8. Инструменты администратора (часто недоступные обычному пользователю).
9. Интерфейс ядра (*внутренние* функции и структуры данных ядра Linux, необходимы 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 её помощью Мефодий поменял себе пароль на прошлой 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 определённого 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)
. . .
```

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 его программах -- справочная подсистема `info`. Страница руководства, несмотря на обилие ссылок *различного* типа, остаётся "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.
```

Просмотр `info`-документа

Программа `info` использует весь экран: на большей его части она показывает текст документа, а первая и две последних строки отведены 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" (от "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 бывает нужно 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
```

```
. . .
```

## Просмотр определённого 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 чём здесь дело ([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, и заявил: "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*. Вполне возможно, что, даже зная, *что* нужно сделать, он не помнит *как* именно -- и поэтому (а также потому, что он -- опытный пользователь) начнёт с изучения руководства.

Это -- закон, у которого есть даже собственное название: **RTFM**, что означает "Read That Fine Manual"

**RTFM** Правило, согласно которому решение любой задачи надо начинать с изучения документации.

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

Руководство -- это совсем не учебник, это -- справочник. В нём содержится информация, *достаточная* для освоения описываемого объекта, но никаких *обучающих* приёмов, никаких определений, повторений и выделения главного в нём обычно нет. Тем более не допускается *усечение* руководства с целью представить небольшие по объёму, но наиболее важные сведения. Так принято в учебниках, причём главные 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 чём 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 всё это преследует цель не *научить*, а раскрыть смысл, пояснить сказанное в других полях. Мефодий нашёл описание работы двойных кавычек в руководстве по sh, однако понял из него далеко не всё -- главным образом, потому, что встретил слишком много незнакомых терминов.

Система info может содержать больше, чем man, поэтому в неё часто включают и учебники (принято называть учебник термином "tutorial"), и т. н. "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 трудоёмкое, 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 нём 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 идёт 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 её придётся 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
```

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 учётом 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 жёсткого стандарта, однако существуют договорённости, 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 придаёт команде. Так проще отличать ключи от других параметров -- и пользователю при наборе командной строки, и программисту, автору команды.

Во-вторых, желательно, чтобы имя ключа было *значащим* -- как правило, это первая буква названия пресловутого действия или свойства. Например, ключ `"-a"` в `man` и `who` происходит от слова "All" (всё), и изменяет работу этих команд так, что они начинают показывать информацию, о которой они обычно умалчивают. А в командах `cal` и `who` смысл ключа `"-m"` -- разный:

```
[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
```

Использование ключа "-m" в разных командах

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

Свойство ключа быть, с одной стороны, предельно коротким, а с другой стороны -- информативным, называется **аббревиативностью**. Не только ключи, но и имена наиболее распространённых 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 узлы (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 -----
. . .
```

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-Cross` (4), 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-менее понятно, что означают неудобочитаемые строки в поле SYNOPSIS руководства. Например `[- smjy13]` из руководства по `cal` ([\\$ManCal](#)) говорит о том, что команду можно запускать с необязательными ключами "-s", "-m", "-j", "-y", "-1" и "-3".

В-четвёртых, есть некоторые менее жёсткие, но популярные договорённости 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 терминал(5). 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
```

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 ли 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; with 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 <bug-coreutils@gnu.org>.

Ключ--help

Мефодий сделал то, о чём просила его утилита head. Видно, 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 -- программу 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 в использовании -- язык управления системой. Все команды, которые использовал Мефодий в работе, были частью именно этого языка.

Из чего несложно было заключить, что *обрабатывать* эти команды, превращать их в последовательность системных и библиотечных вызовов должна тоже какая-нибудь специальная программа, и именно с ней непрерывно ведёт 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 всё управление системой идёт как бы "изнутри" неё: 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 её, иногда -- преобразует по определённым 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 что 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 её, вызывает команду... Так

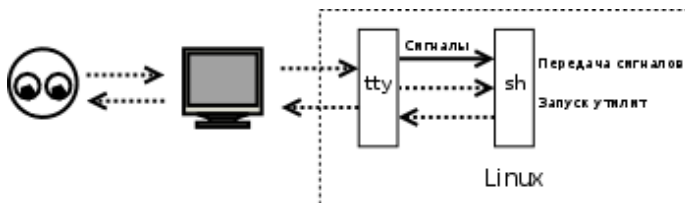


продолжается до тех пор, пока пользователь не скамандует оболочке *завершиться* самой (с помощью `logout` или управляющего символа "`^D`", который для shell значит то же, что и для других программ: больше с терминала ввода не будет).

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 всём, особенно в возможностях *редактирования* командной строки. Помимо `sh` и `bash` в системе может быть установлен "The Z Shell", `zsh`, *самый* мощный на сегодняшний день командный интерпретатор (шутка ли, 22 тысячи строк документации), или `tcsh`, обновлённая и тоже очень мощная версия старой оболочки "C Shell", синтаксис команд которой похож на язык программирования Си.

Когда Гуревич добавлял учётную запись Мефодия в систему, он не стал спрашивать того, 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 -- т. н. **стартовый командный интерпретатор** (login shell), -- это часть пользовательской учётной записи, которую пользователь может изменить командой `chsh` (**ch**ange **sh**ell).

Какая бы задача, связанная с управлением системой, ни встала перед пользователем Linux, она должна иметь решение в терминах командного интерпретатора. Фактически, решение пользовательской задачи -- это описание её 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.



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!)
```

Бесмысленная команда

Одного неудачного запуска `arprows` 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* всё, что вводит пользователь. Ему достаточно разобрать командную строку, выделить из неё 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
```

### 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 её выполнение уходит существенно меньше, командный интерпретатор выберет именно её, 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 так делать, уменьшая длину командной строки. 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 всё-таки может попасть *между* словами способом, описанным в лекциях [Работа с текстовыми данными](#) и [Возможности командной оболочки](#)). Количество разделителей между двумя соседними словами значения не имеет.

Первое слово в тройке передаётся команде как первый параметр, второе -- как второй и т. д. Для того, чтобы разделитель попал *внутри* 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]$
```

### 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! '". Всё время ввода bash услужливо выдавал Мефодию подсказку "> " -- в знак того, что набор командной строки продолжается, но в режиме ввода содержимого кавычек.

---

(1) Некоторые терминалы умеют и так. Следует ещё иметь в виду, что терминалы разных типов имеют *разные* управляющие последовательности.

(2) "Arropos" по-французски означает "кстати", а "whatis" -- по-английски -- "чтотакое".

(3) Кстати, с незапамятных времён who поддерживает один *нестандартный* набор параметров: who am i делает то же, что и who -m.

(4) Странное слово "нода" Мефодий решил оставить на совести неизвестного переводчика сообщений info.

(5) Точнее, на **стандартный вывод**, см. лекцию [Работа с текстовыми данными](#).

(6) Игра слов: "Bourne Again" вслух читается как "born again", т. е. "возрождённый".

## 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 её 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 её пользователей. Более того, все устройства, подключённые к компьютеру (начиная клавиатурой 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 определены несколько различных типов файлов. В основном пользователь имеет дело с файлами трёх типов: **обычными файлами**, предназначенными для хранения данных, **каталогами** и **файлами-ссылками**, именно о них и пойдёт речь в данной лекции, о файлах других типов см. лекцию [Работа с внешними устройствами](#).

**файл** Отдельная 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 слишком неэффективным: про каждый файл приходилось бы помнить, как он называется 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 главы одной 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**

путём к нему.

Linux может работать с различными типами файловых систем, которые различаются списком поддерживаемых возможностей, производительностью в разных ситуациях, надёжностью и другими признаками. Подробнее о работе Linux с разными файловыми системами речь пойдёт в лекции [Работа с внешними устройствами](#). В этой лекции будут описаны возможности файловой системы Ext2/Ext3, на сегодня de facto стандартной файловой системы для Linux.

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* в себе 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 системе более всего напоминает библиотечный каталог, содержащий ссылки на объединённые 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 может сделать доступ к файлу более удобным. В файловой системе Ext2 каждый каталог -- это отдельный файл особого типа ("d", от англ. "directory"), отличающийся от **обычного файла** с данными: в нём 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** можно встретить **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 будем всюду 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 256 символов, и могут содержать любые символы, кроме "/". Причина этого ограничения очевидна: этот символ используется как разделитель имён в составе пути, поэтому не должен встречаться в самих именах. Причём Linux всегда различает прописные и строчные буквы в именах файлов и каталогов, поэтому "method", "Method" и "METHODY" будут тремя *разными* именами.

Есть несколько символов, допустимых в именах файлов и каталогов, которые, при этом, нужно использовать с осторожностью. Это -- так называемые **спецсимволы** "\*", "\", "&", "<", ">", ";", "(", ")", "|", а также пробелы 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 [Терминал и командная строка](#), там же обсуждалось, как изменить его интерпретацию(1). О том, зачем командной оболочке нужны спецсимволы, речь пойдёт в лекции [Возможности командной оболочки](#).

### 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 обеспечить, чтобы приводимые примеры совершенно одинаково выглядели на любых системах. 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 жёсткие 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 упрощённым 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 на ней (как у нашей буквы "ё").

Имена файлов, *записанные* на диск в одной кодировке, выглядят нелепо, если при *просмотре* каталога была установлена другая. Хуже 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 искажённое имя файла с клавиатуры (например, для того, чтобы его переименовать) напрямую не удастся, придётся пускаться на разные ухищрения, описанные в лекции [Возможности командной оболочки](#). Наконец, многие языки, в том числе и русский, исторически имеют *несколько* кодировок(2). 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 всё большее распространение и претендует на статус общего для всех текстов, хранящихся в электронной форме. Однако пока он не достиг желаемой универсальности, особенно в области имён файлов. *Один* символ в UNICODE может занимать *больше* одного байта -- и в этом главный его 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 распространённости этой кодировки также и в том, что UNICODE -- очень громоздкий стандарт, и он может оказаться неэффективным при работе с файловой системой, где скорость и надёжность 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 лёгкий, 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 и ни одной), а после последней точки может быть любое количество символов(3). 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 нём содержатся. Однако нужно учитывать, что расширение -- это только набор соглашений по наименованию файлов разных типов. Строго говоря, данные в файле могут не соответствовать заявленному расширению по той или иной причине, поэтому всецело полагаться на расширение просто 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 данных, содержащихся в файле.

```
[methody@localhost methody]$ file -- -filename-with-  
-filename-with-: ASCII English text  
[methody@localhost methody]$ file /home/methody  
/home/methody: directory
```

### Определение типа данных в файле

Мефодий, забыв, что содержится в файле "`-filename-with-`", который он создал на прошлой 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, в таком файле могут встречаться последовательности, которые случайно совпадут с **управляющими последовательностями** терминала. Поведение терминала после этого может стать непредсказуемым, а неопытный Мефодий 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 не является **обычным файлом**, а, например, каталогом.

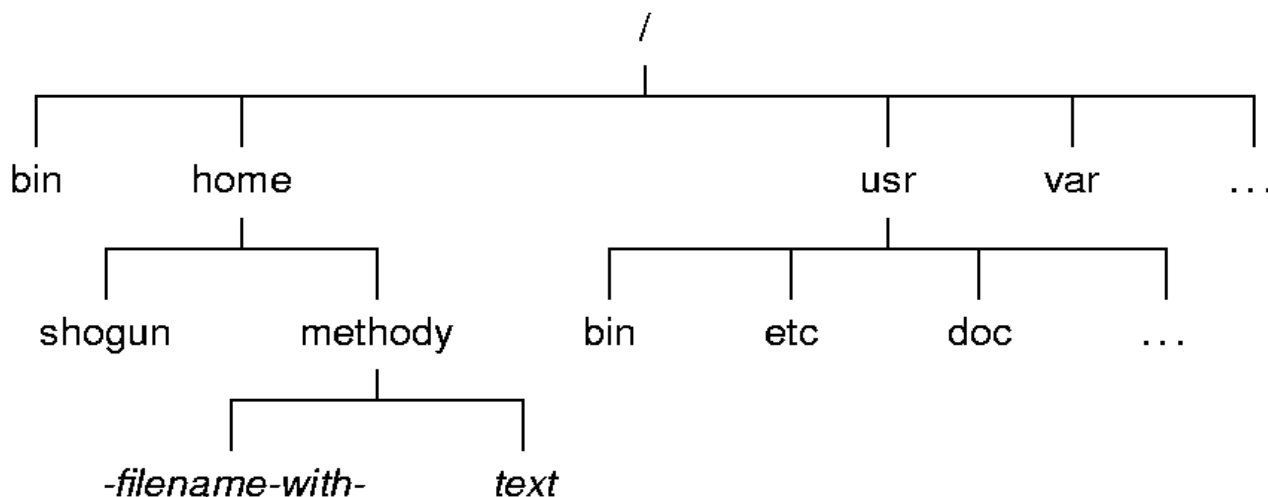
## Дерево каталогов

Понятие каталога позволяет *систематизировать* все объекты, размещённые 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, *всё*, что записано на диске -- файлы, каталоги и специальные файлы -- обязательно "принадлежит" корневому каталогу: либо непосредственно (содержится в нём), 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 нём данные. Если этому принципу *не* следовать, то вскоре окажется гораздо проще складывать *все* файлы в один каталог и искать нужный среди них,

чем проделывать 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 дерева(4), "корнем" которого является корневой каталог, а в вершинах расположены все остальные каталоги. На рис. [dir-tree](#) изображено дерево каталогов, курсивом обозначены имена файлов, прямым начертанием -- имена каталогов.



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 жёстких или сетевых дисках, съёмных дисках, CD-ROM и т. п.(5) Для того, чтобы подключать и отключать файловые системы на разных устройствах в одно общее дерево, используются процедуры **монтирования** и **размонтирования**, о которых речь пойдёт в лекции [Работа с внешними устройствами](#). 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 рёбрам дерева до искомого каталога включительно. Названия соседних вершин разделяются символом "/" ("слэш"). В Linux полный путь, например, до каталога "methody" в файловой системе, приведённой на рис. [dir-tree](#) 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).

Расположение файла в файловой системе аналогичным образом определяется при помощи полного пути, только последним элементом в данном случае будет не название каталога, а название файла. Например, полный путь до созданного Мефодием файла "-filename-with-" будет выглядеть так: "/home/methody/-filename-with-"(7).

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 более подробно разобраться, как устроено дерево каталогов Linux и где что в нём можно найти. Фрагмент дерева каталогов типичной файловой системы Linux (Some Linux, которую использует Мефодий) приведён на рис. [dir-tree](#). 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]$
```

Стандартные каталоги в/

Утилита `ls` вывела список подкаталогов корневого каталога. Этот список будет таким же или почти 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. Этот 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: 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" -- "устройство", отсюда и сокращённое название каталога). Например, файлы `/dev/ttyN` соответствуют **виртуальным консолям**, где *N* -- номер виртуальной консоли. Данные, введённые 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 принципе работы с **файлами-дырками** речь пойдёт в лекции [Работа с внешними устройствами](#).

#### **/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 [Конфигурационные файлы](#).

#### **/home**

- Здесь расположены каталоги, принадлежащие пользователям системы -- **домашние каталоги**, отсюда и название "home". Отделение всех файлов, создаваемых пользователями, от прочих системных файлов даёт очевидное преимущество: серьёзное повреждение системы или необходимость обновления не затронет наиболее ценной информации -- пользовательских файлов.

#### **/lib**

- Название этого каталога -- сокращение от "libraries" (англ. "библиотеки"). **Библиотеки** -- это собрания наиболее стандартных функций, необходимых многим программам: операций ввода/вывода, рисования элементов графического интерфейса и проч. 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 (размещённых в `/bin` и `/sbin`).

#### **/mnt**

- Каталог для **монтирования** (от англ. "mount") -- временного подключения файловых систем, например, на съёмных носителях (CD-ROM и др.). Подробно о монтировании файловых систем речь пойдёт в лекции [Работа с внешними устройствами](#).

#### **/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 промежуточные данные, необходимые для работы. 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 и 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 определённых каталогах(8). 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 даёт использование стандартного расположения 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 никаких процедур -- достаточно просто поместить исполняемый файл в один из соответствующих каталогов.

Рекомендации стандарта по размещению файлов и каталогов основываются на принципе разносить в разные подкаталоги файлы, которые по-разному используются в системе. По типу использования файлов их можно разделить на следующие группы:

#### **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.

#### **изменяющиеся/неизменные файлы**

- К неизменным файлам относятся все статические компоненты программного обеспечения: библиотеки, исполняемые файлы и др. -- всё, что не изменяется само без вмешательства системного администратора. Изменяющиеся -- 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 использовать для хранения этой части файловой системы CD-ROM и другие носители, доступные только для чтения.

#### **разделяемые/неразделяемые файлы**

- Это разграничение становится полезным, если речь идёт 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).

(1) Символ "-" означает, что следующее слово -- ключ, а пробелы и табуляции разделяют параметры в командной строке.

(2) Мефодий и сам несколько раз получал электронные письма, начинающиеся словами "бНОПНЯ" или "бМХЛЮМХЕ" -- результат представления текста, имеющего кодировку CP-1251, в кодировке KOI8-R.

(3) В отличие от старых файловых систем, организованных по принципу "8+3" (DOS, ISO9660 и т. п.), где в имени файла допустимо не более одной точки и расширение может быть не длиннее 3-х символов. Это ограничение определило вид многих из известных сегодня расширений файлов, например, "txt" для текстового файла.

(4) Здесь имеется в виду дерево в строгом математическом смысле: ориентированный граф без циклов с одной корневой вершиной, в котором в каждую вершину входит ровно одно ребро.

(5) Это отличается от технологии, применяемой в Windows или Amiga, где для каждого устройства, на котором есть файловая система, используется свой корневой каталог, обозначенный литерой, например "a", "c", "d" и т. д.

(6) Весьма похожий способ записи полного пути используется в системах Windows, с той разницей, что корневой раздел обозначается литерой устройства с последующим двоеточием, а в качестве разделителя используется символ "\" ("обратный слэш").

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

(8) Краткое описание стандартной иерархии каталогов Linux можно получить, отдав команду `man hier`. Полный текст и последнюю редакцию стандарта **FHS** можно прочесть по адресу <http://www.pathname.com/fhs/>.

## 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 определённом каталоге файловой системы. Такой каталог называется **текущим каталогом**, можно представлять, что программа во время работы "находится" именно в этом каталоге, это её "рабочее место". В зависимости от текущего каталога может меняться поведение программы: 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]$
```

Текущий каталог: `pwd`

`pwd` (аббревиатура от **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 этой **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, использовал 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

```
. . . .
```

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 именно этого файла утилита cat выведет на экран(1). Набирая только имя файла без пути к текущему каталогу, Мефодий воспользовался **относительным путём** к этому файлу.

**Путь к объекту файловой системы**, не начинающийся в **корневом относительный каталоге**. Для каждого **процесса Linux** определён **текущий каталог**, с **пути** 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 путём и относительным 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 относительный пути очень просто: если имя объекта *начинается* на "/" -- это полный путь, в любом другом случае -- относительный.

Пользователь может обращаться к файлу при помощи полного или относительного пути -- результат будет совершенно тот же. Так, команды `cat text` и `cat /home/methody/text`, отданные Мефодием, дали одинаковый результат, поскольку выводился *один и тот же* файл. Если в относительном пути встречаются символы "/", рассматриваются *подкаталоги* текущего каталога, их подкаталоги и т. д. Короче говоря, **относительный путь** строится по тем же правилам, что и **полный**, с той разницей, что относительный путь начинается *не* с символа "/". Сам **текущий каталог**, каков бы ни был полный путь к нему, всегда имеет ещё одно обозначение, ".", которое можно использовать, если по каким-то причинам *требуется*, чтобы даже в *относительном* пути к файлу, находящемуся в *текущем* каталоге, присутствовал элемент "имя каталога". Так, пути "text" и "./text" тоже приводят к одному и тому же файлу, однако в первом случае в строке пути не содержится ничего, кроме имени файла.

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  
.
```

Использование `dirname` и `basename`

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

## Домашний каталог

Мефодий заметил, что на прошлых лекциях и на этой, заходя с разных виртуальных консолей по очереди и одновременно, он всегда оказывался в одном и том же **текущем каталоге**: он всё время обращался к своим файлам при помощи относительного пути и всегда находил нужные. Это не случайно -- в Linux у каждого пользователя обязательно есть *свой собственный* каталог, который и становится текущим сразу после **регистрации в системе** -- **домашний каталог**(2). Для Мефодия домашним каталогом является `"/home/methody"`.

**домашний каталог** 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 своим собственным каталогом и по умолчанию работает в нём, 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]$
```

Команда `ls`

Поданная без параметров, команда `ls` выводит список файлов и каталогов, содержащихся в *текущем каталоге*(3). При помощи этой утилиты Мефодий обнаружил, что

в его **домашнем каталоге** (который в данный момент является текущим) содержатся два файла, созданные на прошлой 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 в файлов в своём 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 начала -- 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]$
```

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 нём может содержаться что-то ещё. 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  .lptoptions  .rpmmacros  Documents/
./               .bash_logout  .emacs   .mutt/       .xemacs/     text
../             .bash_profile .i18n    .pinerc      .xsession.d/ tmp/
[methody@localhost methody]$
```

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, чьё 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 [Возможности командной оболочки](#)), 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 её.

Разберёмся подробно в списке файлов в домашнем каталоге Мефодия. Начнём с весьма лаконичных имён "." 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. 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 сокращённый 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`

---

Не сразу понятно, что приводит этот путь всё туда же, в `/home`.

---

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

<b>родительский каталог</b>	Каталог, в котором содержится данный. Для <b>корневого каталога</b> 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 имён: "." и "..".

За ссылками на текущий и родительский каталоги следуют несколько файлов и каталогов, имена которых начинаются на ".". 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 своё "рабочее место" в системе. Подробнее речь об этом пойдёт в лекции [Конфигурационные файлы](#).

## Перемещение по дереву каталогов

Пользователь может работать с файлами не только в своём 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]$
```

### Смена текущего каталога

Сначала Мефодий решил переместиться в каталог `"/home"`, и посмотреть, что ещё есть в этом каталоге, кроме его домашнего каталога. Он обнаружил ещё один 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 придётся набирать оставшиеся символы. Достраивание -- весьма серьёзное 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 сделать и 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 заменён 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]$
```

### Переход в родительский и в домашний каталог

При помощи символа "~" можно ссылаться и на домашние каталоги других пользователей: "~имя пользователя". В примере [\\$cd.](#) Мефодий перешёл в домашний каталог Гуревича командой "cd ~shogun". Команда cd, поданная без параметров, эквивалента 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 пользователя, как и в любом другом, можно создавать сколь угодно много подкаталогов, в них -- свои подкаталоги и т. д. Другими словами, пользователю принадлежит фрагмент (поддерево) файловой системы, корнем которого является домашний каталог пользователя.

Чтобы организовать такое поддерево, потребуется создать каталоги внутри домашнего. Для этого используется утилита `mkdir`. Она используется 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]$
```

### Создание каталога

Мефодий решил навести некоторый порядок в своём домашнем каталоге и поместить все файлы с примерами и упражнениями в отдельном подкаталоге -- "examples". Теперь, создав каталог, нужно переместить в него все файлы с примерами.

## Копирование и перемещение файлов

Для перемещения файлов и каталогов предназначена утилита `mv` (сокращение 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]$
```

### Перемещение файлов

Мефодий сначала переместил в каталог "examples" файл "-filename-with-", поскольку имя этого файла начинается с "-", ему потребовалось предварить его ключом

"- -", чтобы следующее слово было воспринято командной оболочкой как параметр (этот приём был описан в лекции [Терминал и командная строка](#)). Затем он перешёл в каталог "examples" и переместил из родительского каталога (".. /") файл "text" в текущий каталог ("."). Теперь в каталоге "examples" 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, изменяются *каталоги*, в которых произошло перемещение. Перемещение предполагает удаление ссылки на файл из того каталога, откуда он перемещён, и добавление ссылки на этот самый файл в тот каталог, куда он перемещён. 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` (сокращение 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. Есть несколько возможностей при 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
```

## 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 не выводится никакого предупреждения. В этом случае файл будет просто перезаписан, а данные, которые содержались в старой версии файла, бесповоротно 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 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

## Жёсткие ссылки

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

Пользователь Linux может добавить файлу ещё одно имя (создать ещё одну жёсткую ссылку на файл) при помощи утилиты `ln` (сокращение 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
```

### Создание жёстких ссылок

Мефодий создал в своём домашнем каталоге жёсткую 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 пойдёт в лекции [Права доступа](#). А следующее за ним число -- количество жёстких 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 имён 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 всё же нужно организовать доступ пользователей к файлу, который находится в таком каталоге, можно создать жёсткую 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 жёсткую 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 сокращённого имени (жёсткой ссылки) -- `gcc-3.3`.

## Индексные дескрипторы

Поскольку благодаря жёстким ссылкам у файла может быть несколько имён, 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 каталогов(5), **файлов-дырок** и т. д. Каждому файлу соответствует ровно *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 (жёсткой ссылки на него), которых может быть несколько. Узнать номер индексного дескриптора любого файла можно при помощи всё той же утилиты `ls` с ключом `-i`:

```
[methody@localhost methody]$ ls -i ./text-hardlink examples/text
127705 examples/text 127705 ./text-hardlink
```

### Информация об индексных дескрипторах файлов

Мефодий решил поинтересоваться номерами индексных дескрипторов файла `"text"` и жёсткой 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, 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 *не указаны* в его индексном дескрипторе. В файловой системе Ext2 имена файлов хранятся *в каталогах*: каждый каталог представляет собой список имён файлов и номеров их индексных дескрипторов. Жёсткую ссылку (имя файла, хранящееся в каталоге) можно представлять как каталожную карточку, на которой указан номер индексного дескриптора -- идентификатор файла.

**жёсткая ссылка**      Запись вида *имя файла+номер индексного дескриптора* в каталоге. Жёсткие ссылки в Linux -- основной способ обратиться к файлу *по имени*.

## Символьные ссылки

У жёстких ссылок есть два существенных ограничения:

1. Жёсткая ссылка может указывать только на файл, но не каталог, потому что в противном случае в файловой системе могут возникнуть циклы -- бесконечные пути.
2. Жёсткая ссылка не может указывать на файл на другой файловой системе.

Например, невозможно создать на жёстком диске жёсткую ссылку на файл, расположенный на дискете(6).

Чтобы избежать этих ограничений, были разработаны **символьные ссылки**.

Символьная ссылка -- это просто файл, в котором содержится имя другого файла.

Символьные ссылки, как и жёсткие, предоставляют возможность обращаться к одному и тому же файлу по разным именам. Кроме того, символьные ссылки могут указывать и на каталог, чего не позволяют жёсткие ссылки. Символьные ссылки называются так потому, что содержат *символы* -- путь к файлу или каталогу.

**символьная ссылка**      Файл особого типа ("l"), в котором содержится путь к другому файлу. Если на пути к файлу встречается символьная ссылка, система выполняет *подстановку*: исходный путь заменяется на тот, что содержится в ссылке.

Символьную 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]$
```

## Создание символических ссылок

Теперь Мефодий решил создать в своём домашнем каталоге и символическую ссылку на файл `text` и назвать её `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 `text-hardlink`, а также во втором поле (количество жёстких 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`.

Символическая 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 символическая ссылка, можно при помощи утилиты `realpath`:

```
[methody@localhost methody]$ realpath text-symlink
/home/methody/examples/text
```

Раскрытие символических 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
```

Удаление файла

Разобравшись в ссылках, Мефодий решил удалить файл `text` в каталоге `examples`. После этого файл `text-hardlink` в домашнем каталоге Мефодия, который является жёсткой ссылкой на удалённый файл `text` продолжает благополучно существовать. Единственное отличие, которое заметил Мефодий -- количество жёстких 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 имён этого файла (**жёсткую ссылку**), сам файл остался нетронутым.

Однако если Мефодий удалит и жёсткую 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 жёстких ссылок, а не самих файлов. В Linux, чтобы полностью удалить файл, требуется последовательно удалить все жёсткие ссылки на него. При этом все жёсткие 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 жёсткая ссылка на файл), поэтому команда `rm имя файла` успешно удалит файл в большинстве случаев.

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]$
```

#### 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 нём нет никаких файлов и подкаталогов. Удалить каталог вместе со всем его содержимым можно командой `rm` с ключом `"-r"` (**r**ecursive). Команда `rm -r каталог` -- очень удобный способ потерять в одночасье *все* файлы: она рекурсивно(7) обходит весь *каталог*, удаляя всё, что попадётся: файлы, подкаталоги, символьные ссылки... а ключ `"-f"` (**f**orce) делает её работу ещё неотвратимее, так как подавляет запросы вида "удалить защищённый 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 удалённых файлов и каталогов. Поэтому стоит быть *очень* внимательным, отдавая команду `rm` и, тем более, `rm -r`: нет никакой гарантии, что удастся восстановить случайно удалённые данные. Узнав об этом, Мефодий не огорчился, но подумал, что впредь будет удалять только *действительно*

ненужные файлы, а всё сомнительное -- перемещать с помощью `mv` в подкаталог `~/tmp`, где оно не будет мозолить глаза, и где можно периодически наводить порядок.

---

(1) Вообще говоря, в нескольких разных каталогах файловой системы могут оказаться файлы с именем `text`, именно поэтому командная оболочка всегда передаёт программам и утилитам "точный адрес" файла в файловой системе -- **полный путь**.

(2) Домашний каталог указывается в **учётной записи** пользователя, см. лекцию [Сеанс работы в Linux](#).

(3) Вот пример утилиты, которая по умолчанию работает с файлами в текущем каталоге.

(4) Такое поведение `ls` напоминает принцип работы файловых менеджеров со скрытыми файлами в системах MS-DOS/Windows. Разница в том, что в MS-DOS/Windows скрытые файлы предусмотрены файловой системой -- файл может иметь *атрибут* "скрытый" и при этом называться как угодно. В Linux скрытые файлы -- это не свойство файловой системы, а только *соглашение по наименованию* файлов.

(5) Каталоги в Linux -- тоже файлы особого типа, см. раздел [Структура файловой системы. Система файлов: каталоги](#)

(6) Причина этого ограничения в том, что номер индексного дескриптора уникален только в рамках одной файловой системы. В разных файловых системах могут оказаться два разных файла с одинаковым номером индексного дескриптора, в результате будет невозможно установить, на какой из них указывает жёсткая ссылка.

(7) "Рекурсивно" по отношению к каталогам обозначает, что действие будет произведено над самим каталогом, его подкаталогами, подкаталогами его подкаталогов и т. д.

## 5. Доступ процессов к файлам и каталогам

### Процессы

Как уже упоминалось в лекции [Сеанс работы в Linux](#), 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. Введённое входное имя `getty` передаёт 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, он запускает **стартовый командный интерпретатор**, посредством которого пользователь и командует системой.

Выполняющаяся программа называется в Linux **процессом**. Все процессы система регистрирует в **таблице процессов**, присваивая каждому уникальный номер -- **идентификатор процесса** (**process identifier**, PID). Манипулируя процессами, система имеет дело именно с их идентификаторами, другого способа отличить один процесс от другого, по большому счёту, нет. Для просмотра *своих* процессов можно воспользоваться утилитой `ps` ("process status"):

```
[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
```

Просмотр таблицы собственных процессов

Здесь Мефодий вызвал `ps` с ключом `-f` ("full"), 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` ("parent process identifier") 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 устроен 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 остаётся прежним. *Вернуться* к выполнению старого процесса невозможно, разве что запустить его по новой 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 команды

---

### Нулевой параметр -- `argv[0]` в терминах языка Си и `$0` в терминах shell

---

. 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 недостаточно одного `exec()`. В самом деле, shell не просто запускает утилиту, а дожидается её завершения, обрабатывает результаты её работы и продолжает диалог с пользователем. Для этого в Linux служит системный вызов `fork()` ("вилка, развилка"), применение которого приводит к возникновению ещё 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()`). Отличия 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 ещё 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]$
```

## 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 [Работа с текстовыми данными](#) будет сказано 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, привёл 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
```

## Запуск фонового процесса

В результате стартовый командный интерпретатор (PID 3590) оказался отцом 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* на это терминал могут (правда, когда на одном и том же терминале попеременно появляются сообщения от нескольких фоновых процессов, начинается сушая неразбериха). При каждом терминале в каждый момент времени может быть не больше одного **активного** (foreground) процесса, которому разрешено с этого терминала вводить. На время, пока команда (например, cat) работает в активном режиме, породивший её командный интерпретатор "уходит в фон", и там, в фоне, выполняет свой wait().

### активный процесс

Процесс, имеющий возможность вводить данные с терминала. В каждый момент у каждого терминала может быть не более одного активного процесса.

### фоновый процесс

Процесс, не имеющий возможность вводить данные с терминала. Пользователь может запустить любое, не превосходящее заранее заданного 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 всё, что тому для работы необходимо, 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 её 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 чём не бывало. Такая схема *разделения времени* между процессами носит названия **псевдопараллелизма**.

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 ещё 3634 - 3590=44 процесса. Что ж, в Linux могут одновременно работать несколько пользователей, да и самой системе иногда приходит в голову 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 нужно обработать. Чаще всего процессы ждут завершения операции ввода-вывода. Чтобы посмотреть, как потребляются ресурсы системы, можно использовать утилиту top. Но сначала Мефодий решил запустить *ещё один* бесконечный сценарий: ему было интересно, как два процесса конкурируют за ресурсы между собой:

```
[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
```

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 усталы вычисляют!), а `top` довольствуется десятой долей процента (а то и меньшей: ошибки округления). Стартовый `bash` вообще не хочет работать, он *stun* (значение "S", Sleep, поля `STAT`, status): ждёт завершения активного процесса, `top`.

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

Последний процесс, запущенный из оболочки в фоне, можно из *этой* оболочки сделать активным при помощи команды `fg` ("foreground" -- "передний план").

```
[methody@localhost methody]$ fg
bash loop
^C
```

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

Услужливый `bash` даже написал командную строку, какой 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 предопределённое системой имя). Для того, чтобы передать сигнал, процессу достаточно задействовать системный вызов `kill()`, а для того, чтобы *принять* сигнал, не нужно ничего. Если процессу нужно 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.

Короткое сообщение, посылаемое системой или процессом другому процессу. **сигнал** 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 *навверняка* (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]$
```

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 (**back ground**), запускающую в фоне последний остановленный процесс. Затем он снова перевёл 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 передаёт 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
```

Запуск множества фоновых процессов

Мефодий решил поназапускать 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 изменилась **приглашение командной строки**). Из этого sh он запустил в фоне ещё один sh loop и новый bash loop. Сделал он это одной командной строкой (при этом команды *разделяются* символом "&", т. е. "И"; выходит так, что запускается **и та, и другая команда**). В ps он использовал новый ключ -- "-H" ("Hierarchy", "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
```

### 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, дошёл до обработчика в стартовом bash (PID 3590, и на терминал вывелось 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
```

### Завершение процесса естественным путём с помощью сигнала "Hang Up"

Ждать ему оставалось недолго. Этот sh завершился естественным путём, от команды exit, оставив после себя двух детей-сирот (PID 3652 и 3653), которые тотчас же усыновил "отец всех процессов" -- init (PID 1). Когда кровожадный Мефодий расправился и с ними -- с помощью сигнала 1 (HUP, то есть "Hang UP", "повесить" (1)) -- 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

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, а доступ на **использование** (execute) -- выполнить операцию, специфичную для данного типа объектов. Доступ к объекту можно изменить командой `chmod` (**change mode**, 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 (read), "-r" -- запрещает. Аналогично "+w", "-w", "+x" и "-x" разрешают и запрещают доступ на запись (write) и использование (execute).

## 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 защищённый от записи обычный файл `tmpfile'? y
```

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

Следует заметить, что Мефодию известна операция **перенаправления вывода** -- ">", с помощью которой он создаёт файлы в своём домашнем каталоге. Добавление "> *файл*" в командную строку приводит к тому, что всё, что вывелось бы на экран терминала(2), попадает в *файл*. Мефодий создаёт 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` будет неуспешной, а чтение снова заработает. Сработает и *удаление* этого файла, хотя `rm`, на всякий случай, предупредит о том, что файл защищён 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 на использование, 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 [Возможности командной оболочки](#)).

## Сценарий

Исполняемые файлы в Linux бывают ровно 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 приведёт ни к чему полезному: на экран полезут 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 передаёт 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 ещё 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](#)).

Исполняемый текстовый файл. 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!
```

### Создание простейшего исполняемого сценария

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

Если системе не намеркнута специально, 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 приведёт к запуску указанной после `"#!"` команды, последним параметром которой добавится 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
```

### Создание sort-сценария

Утилита `sort` сортирует -- расставляет их в алфавитном, обратном алфавитном или другом порядке -- строки передаваемого ей файла. Совершенно то же самое произойдёт, 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) передастся ей в качестве параметра. Оформлять файлы, которые необходимо отсортировать, в виде sort-сценариев довольно бессмысленно, просто Мефодий хотел ещё раз убедиться, что сценарий можно написать для чего угодно.

## Доступ к каталогу

В отличие от файла, новый каталог создаётся (с помощью `mkdir`) доступным и на чтение, и на запись, и на использование. Суть всех трёх 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 на использование -- возможность *воспользоваться* этим содержимым: во-первых, сделать этот каталог **текущим**, а во-вторых, *обратиться* за доступом к содержащемуся в нём 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
```

### Доступ к каталогу на чтение и использование

Мефодий создал каталог `dir` и файл в `smallfile` в нём. При смене текущего каталога `bash` автоматически изменил строку-подсказку: как было сказано в лекции [Работа с файловой системой](#), последнее слово этой подсказки -- имя текущего каталога. Описанная в той же лекции команда `pwd` (**print work directory**) 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
```

### 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 нём файла -- нельзя. Неожиданно команда `ls` выдала ошибку. Проницательный Мефодий заметил, что *имя файла* -- `smallfile`, команда `ls` всё-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 и нельзя сделать, когда нельзя *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 [Возможности командной оболочки](#).

```
[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]$
```

#### Доступ к каталогу на использование без чтения

Если каталог, доступный на чтение, но недоступный на использование, мало когда нужен, то каталог, доступный на использование, но не на чтение, может пригодиться. Как видно из примера, получить список файлов, находящихся в таком каталоге, не удастся, но получить доступ к самим файлам, *зная* из имени -- 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**) всё 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
```

#### Рекурсивное удаление каталога

Потеряв интерес к секретным файлам, Мефодий решил удалить каталог `dir`. Из лекции [Работа с файловой системой](#) он знает, что это можно сделать не с помощью `rmdir`, а с помощью `rm` с ключом `"-r"` (**recursive**). Но сходу 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` попадётся 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 на чтение и запись; и все каталоги -- доступными на использование. Параметр

"X" (большое) устанавливает доступ на использование файла, только если этот доступ уже был разрешён *хоть комунибудь* (хозяину, группе или всем остальным). Он позволяет не делать запускаемыми все файлы подряд... впрочем, кого это тревожит, если следующей командой будет `rm`?

---

(1) Имя этого сигнала происходит не от казни через повешение, а от повешенной телефонной трубки.

(2) Точнее, на **стандартный вывод** программы, такое перенаправление не касается **стандартного вывода ошибок**.

## 6. Права доступа

### Права доступа

Работая с Linux, Мефодий заметил, что некоторые файлы и каталоги недоступны ему ни на чтение, ни на запись, ни на использование. Зачем такие нужны? Оказывается, другие пользователи *могут* обращаться к этим файлам, а у Мефодия просто *не хватает 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 примере ниже 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) группы=400(shogun),4(adm),10(wheel),19(proc)
```

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 [Сеанс работы в Linux](#), пользователь может быть членом нескольких **групп**, равно как и несколько пользователей может 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 [Конфигурационные файлы](#)

будет рассказано, что **GID** пользователя вписан в **учётную запись** и хранится в `/etc/passwd`, а информация о соответствии *имён* групп их идентификаторам, равно как и о том, в какие *ещё* 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 *удаётся* 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 т. н. **атрибутов**, а также некоторую дополнительную информацию.

**Атрибуты** определяют, кто и что с файлом имеет право делать, и 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
```

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

Ключ `"-l"` утилиты `ls` определяет "длинный" (long) формат выдачи (справа налево): имя файла, время последнего изменения файла, размер в байтах, группа, хозяин, количество **жёстких ссылок** и строка **атрибутов**. Первый символ в строке 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 *жёстких ссылок* на каталог невозможно, поле "количество жёстких 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 имён: собственное (например, `tmp`) и имя `"."` в самом этом каталоге (`tmp/.`). Если же в каталоге создать подкаталог, количество жёстких ссылок на этот каталог увеличится на 1 за счёт 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
```

Несколько жёстких ссылок на *каталог* всё-таки бывают!

Здесь Мефодий использовал ключ "- d" (**directory**) для того, чтобы `ls` выводил информацию не о *содержимом* каталога `tmp`, а о самом этом каталоге.

Пример от рута есть смысл приводить только в том случае, если пример на что-то *совершенно* универсальное, что обязательно будет устроено точно так же в любом Линуксе. Иначе есть опасность, что пользователь начнёт мудрить -- и что он там намудрит... 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
```

Ограничение доступа к полной таблице процессов

Оказывается, запущено немало процессов, в том числе один -- пользователем `shogun` (**PID 4558**). Однако, несмотря на ключ "- a" (**all**), `ps` выдала Мефодию только сведения о его процессах: 4529 -- это входной shell, а 4589 -- видимо, сам `ls`.

Другое дело -- Гуревич. Он, как видно из примера **SID**, 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
```

Доступ к полной таблице процессов: группа `proc`

Гуревич, опытный пользователь Linux, предпочитает `bash`-у "The Z Shell", `zsh`. Отсюда и различие приглашения в командной строке. Во всех `shell`-ах, 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 `ср ~/нужный_файл .` или `ср нужный_файл ~` соответственно.

Команда `ps -a` выводит информацию обо *всех* процессах, запущенных "живыми" (а не системными) пользователями(2). Для просмотра *всех* процессов Гуревич пользуется командой `ps -efH`.

## Разделяемые каталоги

Проанализировав систему прав доступа к каталогам в Linux, Мефодий пришёл к выводу, что в ней имеется существенный недочёт. Тот, кто имеет право *изменять* каталог, может *удалить любой файл* оттуда, даже такой, к которому совершенно не имеет доступа. Формально всё правильно: удаление файла из каталога -- всего лишь изменение содержимого каталога. Если у файла было больше одного имени (существовало несколько жёстких ссылок на этот файл), никакого удаления *данных* не произойдёт, 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, совершенно ему недоступный, да к тому же с подозрительным именем, содержащим пробел:

```
[methody@localhost methody]$ ls
4T0-T0 Мep3кое Documents examples loop script tmp to.sort
[methody@localhost methody]$ ls -l 4*
-rw----- 1 root root 0 Сен 22 22:20 4T0-T0 Мep3кое
[methody@localhost methody]$ rm -i 4*
rm: удалить защищённый от записи пустой обычный файл `4T0-T0 Мep3кое'? y
```

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
```

## Работа с файлами в разделяемом каталоге

Убедившись, что *любой* доступ в каталог /tmp открыт *всем*, Мефодий захотел удалить оттуда все файлы. Затея гораздо более хулиганская, чем *заведение* файла: а вдруг они кому-нибудь нужны? Удивительно, но удалить удалось 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 *ещё* 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 *группа*, 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 придёт 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
```

## Что можно делать с чужим файлом в своём 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 всё как было": чтобы получился файл с тем же именем и тем же содержанием, принадлежащий Мефодию, а не root-у. Это несложно: чужой файл можно переименовать (это действие над каталогом, а не над файлом), скопировать переименованный файл в файл с именем старого (доступ по чтению открыт) и, наконец, удалить чужой файл с глаз долой. Ключ "-f" (**force**, "силком") позволяет утилите mv делать своё 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 изрядно 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.

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 учётных записей Linux всегда есть запись по имени `root` ("корень"[\(3\)](#)), соответствующая нулевому идентификатору, поэтому вместо "суперпользователь" часто говорят "root". Множество системных файлов принадлежат `root`-у, множество файлов только ему доступны на чтение или запись. Пароль этой учётной записи -- одна из самых больших драгоценностей системы. Именно с её 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 всё -- в том числе и всё 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 вошёл *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 такой способ не любят.



Второй способ -- это воспользоваться специальной утилитой `su` (shell of user), 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, 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 **учётную 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* учётных записей? Должен существовать механизм, позволяющий пользователю запускать процессы с идентификаторами *другого* пользователя, причём 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(4)**. 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
```

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* учётных записях. Как правило, `SetUID`-ные файлы доступны обычным пользователям только на выполнение, чтобы не провоцировать этих обычных 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 её 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
```

### Не подверженная взлому программа `passwd`, использующая `SetGID`

Каталог `/etc/tcb` в этой системе содержит подкаталоги, соответствующие входным именам всех её 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 её делать что угодно, злоумышленник всего только и сможет, что... отредактировать *собственную* учётную запись!

Оказывается, атрибут **SetGID** можно присваивать *каталогам*. В последнем примере каталог `/etc/tcb/methody` имел `SetGID`, поэтому все создаваемые в нём файлы получают **GID**, равный **GID** самого каталога -- `auth`. Используется это разнообразными процессами идентификации, который, 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 передаётся 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, поделаться он всё 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*", причём число надо записывать в *восьмеричной системе счисления*. Так, атрибуты каталога `/tmp` будут равны `1777`, атрибуты `/bin/su` -- `4711`, атрибуты `/bin/lis` -- `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` приведёт 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 для них полностью недоступны ("`rw-----`" и "`rwx-----`")<sup>(5)</sup>.

---

(1) Здесь есть тонкость. В файле `group` указываются не идентификаторы, а **входные имена** пользователей. Формально говоря, можно создать двух пользователей с *одинаковым* UID, но *разными* GID и списками групп. Обычно так не делают: надобности -- почти никакой, а неразберихи возникнет много.

(2) Более точно -- обо всех процессах, имеющих право выводить на какой-нибудь терминал, а значит, запущенными из терминального сеанса.

(3) Вместо **полного имени** такому пользователю часто пишут "`root of all evil`".

(4) Строго говоря, при этом меняется не собственно идентификатор пользователя, а т. н. **исполнительный идентификатор пользователя, EUID**; это нужно для того, чтобы знать, кто *на самом деле* запустил программу.

(5) Параметр команды `umask` должен обязательно начинаться на `0`, как это принято для восьмеричных чисел в языке Си.

## 7. Работа с текстовыми данными

### Ввод и вывод

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

Каждая программа работает с данными определённого 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 [Терминал и командная строка](#)). Поскольку ввести с терминала и вывести на терминал можно только текстовую информацию, то ввод и вывод программ, связанных с терминалом(1), 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 даёт Linux при работе с данными в текстовой форме, и посвящена данная лекция.

"Текстовость" данных -- всего лишь *договорённость* об их формате. Никто не мешает выводить на экран нетекстовый файл, однако пользы в том 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 её выполнит. Например, если в скомпилированной программе записано число 1528515121, оно представлено в виде четырёх байтов: 27, 91, 49 и 74. Соответствующий им *текст* состоит из четырёх 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 всё-таки желательно *просмотреть* (то есть превратить в *текст*)? Можно воспользоваться программой `hexdump`, которая выдаёт содержимое файла в виде шестнадцатеричных ASCII-кодов, или `strings`, которая показывает только те части файла, что *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.....|
. . .
[methodoy@localhost methodoy]$ strings /bin/cat | less
/lib/ld-linux.so.2
__Jv_RegisterClasses
__gmon_start__
libc.so.6
stdout
. . .
[methodoy@localhost methodoy]$ strings -n3 /bin/cat | less
/lib/ld-linux.so.2
GNU
__Jv_RegisterClasses
__gmon_start__
libc.so.6
stdout
. . .

```

## 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 передаётся `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, придётся 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 использоваться. 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 *поток данных*, а куда именно ведёт 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.

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

Механизм копирования **окружения**, описанный в лекции [Доступ процессов к файлам и каталогам](#), 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 выводиться 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 файл 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 [Доступ процессов к файлам и каталогам](#)).

```
[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
```

## 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 ещё. Вместо того, чтобы самой обеспечивать доставку вывода до конечного адресата (будь то человек или файл), `cat` отправляет все данные на **стандартный вывод** (сокращённо `-- stdout`).

Подмена стандартного вывода -- задача командной оболочки (`shell`). В данном примере `shell` создаёт пустой файл, имя которого указано после знака ">", и дескриптор этого файла передаётся программе `cat` под номером 1 (**стандартный вывод**). Делается это очень просто. В лекции [Доступ процессов к файлам и каталогам](#) 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: её стандартный вывод уже подменён. Эта операция называется *перенаправлением* стандартного вывода. В том случае, если файл уже существует, `shell` запишет его заново, полностью уничтожив всё, что в нём 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]$
```

## Недеструктивное перенаправление стандартного вывода

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

**стандартный вывод** 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** (сокращённо `-- 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]$
```

## Перенаправление стандартного ввода из файла

Результат действия этой команды совершенно аналогичен команде `sort textfile`, разница 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 передаёт ей на стандартный ввод содержимое файла.

Стоит помнить, что операция ">" *деструктивна*: она всегда создаёт 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** 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]$
```

### Стандартный вывод ошибок

Удивлённый Мефодий обнаружил, что вопреки его указанию отправляться в файл две строки, выведенные командой `info`, всё 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** (сокращённо `-- stderr`).

**стандартный** Поток данных, открываемый системой для каждого процесса в момент



## вывод ошибок

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]$
```

## 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 чтобы её 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]$
```

## 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* её 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 четвёртый 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. Всё, 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]$
```

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 распространённый 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
```

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, снабжённой Клапаном для Чтения или Клапаном для Записи, то **канал** -- это оба Клапана, приклеенные друг к другу вообще без Коробки. Для определённости между Клапанами можно представить Трубу, немедленно доставляющую данные от входа к выходу (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 остаётся запустить по команде в этих процессах и стандартный вывод первой попадёт на стандартный ввод второй.

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

## Фильтры

Если программа и вводит данные, и выводит, то её можно рассматривать как трубу, в которую что-то входит, а что-то выходит. Обычно смысл работы таких программ заключается в том, чтобы определённым образом *обработать* поступившие данные. В Linux такие программы называют **фильтрами**: данные проходят через них, причём что-то "застревает" в фильтре и не появляется на выходе, 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 стандартный конструктор, где все детали 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

несколько 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 имён. Операции над файлами: создание, переименование, перемещение, а также сортировка, перекодировка и прочее, означает замену одних *символов* и *строк* на другие 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 [Возможности командной оболочки](#) и [Конфигурационные файлы](#)).

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:

### Строки

- Строка -- основная единица передачи текста в Linux. Терминал передаёт 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**).

### Поля

- В одной строке может упоминаться и больше одного объекта. Если понимать объект как последовательность символов из определённого набора (например, букв), то строку можно рассматривать как состоящую из слов и **разделителей**(2). 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/method"` разделителем является символ `"/"`, то первое поле пусто, второе содержит слово `"home"`, третье -- `"method"`. Некоторые утилиты позволяют выбирать из строк отдельные поля (по номеру) и работать со строками как с таблицей: выбирать и объединять нужные колонки и проч.

## 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 ещё символ возврата каретки ("`^M`"). Это вызвало путаницу: некоторые системы требуют, чтобы в конце текстового файла стояло *оба этих* символа в определённом порядке. Чтобы путаницы избежать, 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 посвящён нескольким примерам использования стандартных утилит для решения разных типичных (и не очень) задач. Эти 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.

## Подсчёт

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 более удобен для такой процедуры. Стандартная утилита для подсчёта строк, слов и символов -- `wc` (от англ. "word count" -- "подсчёт слов"). Однако Мефодий запомнил, что в Linux многое можно представить как слова и строки, и решил с её помощью посчитать свои файлы.

```
[methody@localhost methody]$ find . | wc -l
42
[methody@localhost methody]$
```

Подсчёт файлов при помощи `find` и `wc`

Удовлетворённый Мефодий получил желаемое число -- "42". Для этого ему потребовалась команда `find --` рекомендованный ему Гуревичем инструмент поиска нужных файлов в системе. Мефодий вызвал `find` с одним параметром -- каталогом, с которого 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 её посчитать количество полученных строк "-l". `wc` выдала в ответ искомое число.

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

## Отбрасывание ненужного

Иногда пользователя интересует только часть из тех данных, которые собираются 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 определённая 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 путём, а только именем файла (скорее всего, это требуется для последующей автоматической обработки этого списка).

```
[methody@localhost methody]$ find /etc -maxdepth 1 -group adm 2> /dev/null \
> | cut -d / -f 3
syslog.conf
anacrontab
[methody@localhost methody]$
```

### Извлечение отдельного поля

Если команда получается такой длинной, что её неудобно набирать в одну строку, можно разбить её 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 ещё не передана системе и набор продолжается(3). Для системы безразлично, в сколько строк набрана команда, возможность набирать 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 запрещённом 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`<sup>(4)</sup>.

## 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 определённое слово или комбинация символов. Для этого подходит стандартная утилита `grep`. `grep` может искать строку в файлах, а может работать и 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
```

### 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, то Мефодий получил только процессы, в имени которых есть `"bash"`. Однако Мефодий неожиданно нашёл 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 ещё 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 определённый 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 -on "\*Note[^:]*::" grep.info
324:*Note Grep Programs::
684:*Note Invoking::
[methody@susanin methody]$
```

## 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 -- звёздочка. `*`, поставленная после элемента регулярного выражения обозначает, что могут быть найдены тексты, где этот элемент повторён любое количество раз, в том числе и ни одного, т. е. просто отсутствует. В нашем примере звёздочка встретилась дважды: в первый раз потребовалось включить в регулярное выражение именно символ "звёздочка", для этого потребовалось лишить его специального значения, поставив перед ним `\`.

Вторая звёздочка обозначает, что стоящий перед ней элемент может быть повторён любое количество раз от нуля до бесконечности. В нашем случае звёздочка относится к выражению в квадратных скобках -- `[^:]`, что означает "любой символ, кроме `:`". Всё 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, отлично 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 по использованию, поэтому чтобы научиться экономить время и усилия при помощи регулярных выражений, полезно прочесть соответствующие главы книги [Курячий:2004](#) и книгу [Фридл:2000](#).



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 -- Vi и Emacs, о которых речь пойдёт в следующей лекции ([Текстовые редакторы](#)). 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 на другие предназначена утилита `tr` (сокращение от англ. "translate", "преобразовывать, переводить"), работающая как фильтр. Мефодий решил употребить её прямо по назначению и выполнить при её помощи транслитерацию -- замену латинских символов близкими по звучанию русскими.

```
[methody@localhost methody]$ cat cat.info | tr abcdefghijklmnopqrstuvwxyz
абцдефгхийклмнопкrstуввсиз \
> | tr ABCDEFGHIJKLMNOPQRSTUVWXYZ 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]$
```

### 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 на другие, причём 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@oblomov methody]$
```

## 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 нём разобраться, нужно обязательно прочесть руководство `sed(1)` и знать регулярные выражения.

## Упорядочение

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 её 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]$
```

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 частотных 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]$
```

Поиск всех исполняемых файлов, которые точно являются сценариями

Здесь Мефодий решил определить, какие из исполняемых файлов в каталоге "/bin" являются сценариями. Для этого он нашёл все обычные исполняемые файлы (указывать "type f" -- "обычные файлы" потребовалось, чтобы в результат не попали каталоги, которые все исполняемые), а затем для каждого найденного файла вызвал grep, чтобы поискать в нём 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 сценариев(5).

---

(1) Т. е. в первую очередь командной оболочки и её процессов-потомков, см. лекцию [Доступ процессов к файлам и каталогам](#).

(2) Например, в командной строке разделителями являются символы пробела и табуляции (см. раздел [Терминал и командная строка. Слова и разделители](#)).

(3) Вид этого приглашения определяется значением переменной окружения "PS2", описанной в лекции [Возможности командной оболочки](#).

(4) Как уже указывалось в разделе [Структурные единицы текста](#), первым полем считается текст от начала строки до первого разделителя, в приведённом примере первое поле -- пусто, "etc" -- содержимое второго поля, и т. д.

(5) Возможны сценарии, в которых не указана программа-интерпретатор, но для автоматического обнаружения такого сценария потребуются более сложные инструменты.

## 8. Доступ процессов к файлам и каталогам

### Редактирование ввода

Уже некоторое время поработав в Linux, понабрав команды в командной строке, Мефодий пришёл 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 влез мусор, 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. Система 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, выдаёт 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 = <undef>;
eol2 = <undef>; 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
```

#### Настройки терминальной линии

При виде столь обширных возможностей Мефодий немедленно взялся читать руководство (`man stty`), однако нашёл в нём 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`, причём `^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 нажмём 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
```

### 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 всё, что вводит пользователь. При включённом `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. Ещё 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 [Этапы загрузки системы](#)) и `terminfo`.

---

Если терминал ведёт себя странно, последовательность `"^J stty sane^J"` может его вылечить!

---

## Редактирование командной строки

Даже не изучая специально возможностей командной оболочки, Мефодий активно использовал некоторые из них, не доступные при вводе текста *большинству* утилит (в частности, ни `cat`, ни `hexdump`). Речь идёт 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 [Терминал и командная строка](#) он уже убедился, что эти команды работают в `bash`, но не работают для `cat`. Более того, для простого командного интерпретатора -- `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 *всё* про работу в командной строке принесла некоторую пользу. Во-первых, перемещаться в командной строке можно не только по одному символу вперёд 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 ещё немало других удобств, о которых и пойдёт 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, её можно отредактировать и подать оболочке (подгонять курсор к концу строки при этом не обязательно).

Если необходимо добыть из истории какую-то давнюю команду, проще не гонять список истории стрелками, а *поискать* в ней с помощью команды "^R" (reverse search). При этом выводится подсказка специального вида ("(reverse-i-search)"), подстрока поиска (окружённая 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"
```

### 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 её *bash* с помощью *Enter*.

Чтобы история команд могла сохраняться *между* сеансами работы пользователя, *bash* записывает её 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 500 командами, но это можно и перенастроить).

## Сокращения

Поиск по истории -- удобное средство: длинную командную строку можно не набирать целиком, а выискать и использовать. Однако *давнюю* команду придётся добывать с помощью нескольких "^R" -- а можно и совсем не доискаться, если она уже выбыла оттуда. Для того, чтобы оперативно заменять короткие команды длинными, 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'
```

### Просмотр заранее определённых сокращений

С сокращениями Мефодий уже сталкивался в лекции [Права доступа](#), 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
```

### Использование сокращений и pushd/popd

От надоедливого "- i" Мефодий избавился с помощью команды `unalias`, а заодно ввёл сокращения для полюбившихся ему команд `bash -- pushd` и `popd`. Эти команды(1), 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.

<b>команда-сокращение</b>	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 имён *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 введённому пути больше файлов, чьё 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-
```

### Использование достраивания

Дальше -- больше. Оказывается, и имя команды можно вводить не целиком: оболочка догадается достроить 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 её часть, относительно которой у него нет сомнений. Если дальнейшее достраивание может пойти *несколькими* путями, то однократное нажатие *Tab* приведёт к тому, что *bash* растерянно пискнет(2), 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 пойдёт 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* -- *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 имён 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 [Работа с текстовыми данными](#): для поиска строк определённой 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, заключённых в квадратные скобки "[" и "]" соответствует строка длиной в *один* символ, причём этот символ должен *встречаться* среди заключённых 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), причём первая подстрока будет отвечать первой части шаблона, вторая -- второй части и т. д. Например, шаблону "a\*b?c" будут соответствовать строки ab@c ("\*" соответствует пустая подстрока), a+b=c и aaabbc, но не соответствовать abc ("?" соответствует подстрока c, а для "c" соответствия не находится), @ab@c (нет соответствия для "a") или aaabbbc (из трёх b первое соответствует "b", второе -- "?", а вот третье приходится на "c").

**шаблон** Строка специального формата, используемая в процедурах текстового поиска.

Говорят, что строка **соответствует** шаблону, если можно по определённым

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 на список *файлов*, имена которых ему соответствуют. Команда, которая затем вызывается, получает в качестве параметров список файлов уже безо всяких шаблонов, как если бы этот список пользователь ввёл вручную. Эта способность командного интерпретатора называется **генерацией имён 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*
```

### 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 превращён 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 *команды*(3), которой -- в качестве *параметров* -- были переданы *остальные* элементы списка -- /bin/ed, /bin/egrep и /bin/ex.

Что же касается ls \*a\*, то, по расчётам 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 чёрной магии тут нет. Во-первых, имена файлов вида ". bash\*" хотя и содержат "a", но начинаются на точку, и, стало быть, считаются **скрытыми**. Скрытые файлы попадают в результат генерации имён только если точка в начале указана *явно* (как в первой команде примера). Поэтому по шаблону "\*a\*" в домашнем каталоге Мефодия bash нашёл только подкаталог с именем examples, его-то он и передал в качестве параметра утилите ls. Что вывелось на экран в результате образовавшейся команды ls examples? Конечно, содержимое каталога. Шаблон в последней команде из примера, "\*[ao]\*", был превращён в список файлов, чьи имена содержат "a" или "o" -- Documents examples loop to.sort, а ключ "-d"

потребовал у `ls` показывать информацию о каталогах, а не об их содержимом. В соответствии с ключом `-F`, `ls` расставил `/` после каталогов и `*` после исполняемых файлов.

Ещё одно отличие генерации имён от стандартной обработки шаблона -- в том, что символ `/`, разделяющий элементы пути, *никогда* не ставится в соответствие `*` или диапазону. Происходит это не потому, что искажён алгоритм, а потому, что при генерации имён 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 ещё и каталоги `/bin` и `/sbin` -- то есть подкаталоги *другого* уровня вложенности -- по стандартным правилам сделать нельзя(4).

Если перед любым специальным символом стоит `\`, этот символ лишается специального значения, **экранируется**: пара `\СИМВОЛ` заменяется командным интерпретатором на `СИМВОЛ` и передаётся в командную строку безо 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
```

#### Экранирование специальных символов и обработка "пустых" шаблонов

Мефодий заметил, что шаблон, которому не соответствует *ни одного* имени файла, `bash` раскрывать не стал, как если бы все `*` в нём 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 способ(5).

Лишить специальные символы их специального значения можно и другим способом. В лекции [Терминал и командная строка](#) было рассказано, что разделители (пробелы, символы табуляции и символы перевода строки) перестают восприниматься таковыми, если часть командной строки, их содержащую, окружить двойными или одинарными кавычками. В кавычках перестаёт "работать" и генерация имён (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 [Доступ процессов к файлам и каталогам](#) уже упоминалось, что системный вызов `fork()`, создавая точную копию родительского процесса, копирует также и **окружение**. Необходимость в "окружении" происходит вот из какой задачи. Акт передачи данных от *этого конкретно* родительского процесса дочернему, и, что ещё 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 файл(6). Хорошо бы, чтобы *изменение* данных и их *передача* делались бы *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 нём 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 изменённом 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
```

Утилита `date` пользуется переменной окружения `LC_TIME`

Набор данных, приписанных системой процессу. Процесс может пользоваться информацией из окружения для настройки, изменять и дополнять его.

**окружение** 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 приёмом: 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 после. Переменная окружения LC\_TIME предписывает использовать определённый язык при выводе даты и времени а значение "C" соответствует "стандартному системному" языку (чаще всего -- английскому).

Если рассматривать shell в качестве высокоуровневого языка программирования, 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
```

### Подстановка значений переменных

Как видно из примера, значение *неопределённой* переменной (B) в shell считается пустым и при подстановке не выводится никаких предупреждений. Сама подстановка происходит, как и генерация имён, *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 четвёртая 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, её 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
```

Экспорт переменных shell в окружение

Здесь Мефодий завёл *новую* переменную `Qwe` и изменил значение переменной окружения `LANG`, доставшейся стартовому `bash` от программы `login`. В результате запущенный дочерний процесс `sh` получил *изменённое* 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 определённых переменных используются либо самой командной оболочкой, либо утилитами системы, поэтому их изменение приводит к тому, что оболочка или утилиты начинают работать слегка 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 имён запускаемых файлов во всех каталогах `PATH`, и только в них. Точно так же будут поступать и другие утилиты, использующие **библиотечную функцию** `exec lp()` 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!
```

Использование `PATH`



Пути, указанные в PATH не обязаны 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 этот 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, стоит ожидать подвоха...

Переменных окружения, влияющих на работу *разных* утилит, довольно много. Например, переменные семейства LC\_ (полный их список выдаётся 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 [Терминал и командная строка](#) разные терминалы имеют различные управляющие последовательности, поэтому программы, желающие эти последовательности использовать, обязательно сверяются с переменной TERM(7). Если какая-то утилита требует редактирования файла, этот файл передаётся программе, путь к которой хранится в переменной EDITOR (обычно это /usr/bin/vi, о котором речь пойдёт в лекции [Текстовые редакторы](#)). 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 завершён (например, не закрыты кавычки). С \$PS2 (символом ">") Мефодий уже сталкивался в лекциях [Терминал и командная строка](#) и [Работа с текстовыми данными](#).

```
[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:~ $
```

## 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 ещё 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 в конце концов "\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 происходит 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 и других, более мощных интерпретируемых языках в Linux, не помещается в рамки нашего курса. Так что, пока Мефодий читает документацию по bash и самозабвенно упражняется в написании сценариев, нам остаётся 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 к sh по возможностям), 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 передаёт 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
```

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, заключённый между `then` и `fi`. Точка с запятой *разделяет* операторы в `sh`; либо она, либо перевод строки необходимы перед `then` и `fi`, иначе всё, что идёт 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 жёсткая 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
```

Оператор `test`

Команда `test -f` проверяет, является ли её аргумент файлом; поскольку `examples` -- это каталог, результат -- неуспешный. 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 then*) *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 **подстановка вывода** команды. Подстановка вывода имеет вид `"` команда `"` (другой вариант -- `"$( команда) "`). Как и подстановка значения переменной, она происходит *перед* тем, как начнётся разбор командной строки: выполнив команду и получив от неё 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
```

### 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 её 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 нём 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
```

### 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 цикла 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 "<$F>"; done; echo
<Сбт><Ноя><6><12:08:38><MSK><2004>
[methody@localhost methody]$ 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
```

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, куда-то пропадают после окончания его работы. 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 и делается с помощью специальной команды ".". Если вдруг в передаваемом сценарии обнаружится команда exit, ехес 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 непрерывных 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
```

#### 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!). Этим 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` ещё одну пару команд. Правда, Гуревич советовал 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**) выполняется с помощью команды "." сценарий из файла со специальным именем -- /etc/profile. Это -- т. н. **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 берётся 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 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 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 файла `.bash_logout`

Заметив, что команда `alias` выдаёт список сокращений в том же формате, в котором они и задаются, Мефодий придумал, как обойтись без редактирования файла `~/ .alias`. Отныне все сокращения, определённые к моменту завершения сеанса работы, будут записываться обратно в `.alias`. Туда попадут и те сокращения, что прочлись 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 файла `.alias`

Последняя запись в файле `.alias` относится к инструменту, которым Мефодий создавал все эти файлы: текстовому редактору `vim`. О текстовых редакторах речь пойдёт в следующей лекции.

---

(1) Они названы по аналогии с операциями работы со стеком -- `push` и `pop`.

(2) Все терминалы должны уметь выдавать звуковой сигнал при *выводе* управляющего символа `^G`. Для этого не нужно запускать никаких дополнительных программ: "настоящие" терминалы имеют встроенный динамик, а виртуальные консоли обычно пользуются системным ("пищалкой"). В крайнем случае разрешается привлекать внимание пользователя другими способами: например, эмулятор терминала `screen` пишет в служебной строке `"wuff-wuff"` ("гав-гав").

(3) Можно вспомнить про *нулевой* параметр командной строки, обсуждавшийся в лекции [Доступ процессов к файлам и каталогам](#).

(4) Генерация имён файлов в `zsh` предусматривает специальный шаблон `"**"`, которому соответствуют подстроки с *любым* количеством `"/"`. Пользоваться им следует *крайне осторожно*, понимая, что при генерации имён по такому шаблону выполняется операция, аналогичная не `ls`, а `ls -R` или `find`. Так, использование `"/**"` в начале шаблона вызовет просмотр *всей* файловой системы!



(5) Авторы `zsh` пошли по другому пути: в этой версии `shell` использование шаблона, которому не соответствует ни одно имя файла, приводит к *ошибке*, и соответствующая команда не выполняется.

(6) Эта ситуация называется "race condition" ("состояние гонки"), и часто встречается в плохо спроектированных системах, где есть хотя бы *два* параллельных процесса.

(7) В действительности такие программы обычно используют библиотеку `curses`, оперируя *не зависящими* от типа терминала понятиями (вроде "очистка экрана" или "позиционирование курсора", а процедуры из `curses` преобразуют их в управляющие последовательности конкретного терминала, сверившись сначала с `$TERM`, а затем -- с содержимым соответствующего раздела *базы данных по терминалам*, которая называется `terminfo`.

## 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 если имел 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 [Возможности командной оболочки](#) и [Конфигурационные файлы](#)). Но и собственные задачи пользователя могут потребовать редактирования текстовых файлов: например, сценарии и программы, электронные письма, а также заметки для себя, которые пишет Мефодий -- это всё 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 свои 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, т. к. 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, -- много очень, требуется способ вводить любые команды ограниченными средствами терминала. Это условие, равно как и требование удобной работы 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 самой минимальной, 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 Vim(2). Однако верно и обратное: если Вы работаете в незнакомой системе Linux или произошёл сбой, в результате которого доступна только очень небольшая часть системы, нельзя быть уверенным, что найдётся хоть какой-нибудь другой текстовый редактор кроме Vim. Поэтому каждому пользователю Linux нужны хотя бы основные навыки работы в Vim. При *первом* знакомстве с Vim работа обычно не ладится: очень уж он непривычен, его нельзя с удобством использовать, запомнив только две-три простейшие команды редактирования. Стоит понять основные принципы работы в Vim и потратить некоторое время на его освоение, тогда в нём откроется мощный инструмент, позволяющий очень эффективно работать с текстом.

Под именем Vim, на самом деле, могут скрываться несколько разных программ: с момента появления Vim в операционной системе UNIX (а это произошло около 30-ти лет назад), этот редактор стал чем-то вроде стандарта. К настоящему времени существует ряд программ либо в точности повторяющих вид и поведение "классического" Vim (например, nv i), либо очень похожих на него, но со значительно расширенными возможностями (Vi, e l v i s). Наибольшей популярностью пользуется Vi, возможности которого огромны, для их описания потребовалось почти сто тысяч строк документации. Когда пользователь Linux набирает в командной строке Vim, скорее всего будет запущена "облегчённая" 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 Vim мы будем понимать именно Vi в режиме совместимости, все возможности, недоступные в этом режиме, будут отдельно оговариваться. Чтобы вызвать Vi в полнофункциональном режиме, достаточно набрать команду Vi.

Популярность Vi не случайна: этот текстовый редактор позволяет не только производить простые операции редактирования текстовых файлов, но и хорошо приспособлен 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 Vi есть специальные средства для организации цикла отладки программы: компиляция--правка исходного текста--компиляция... Подробнее об этих и прочих возможностях Vi можно узнать из руководств по Vi. Объем данной лекции позволяет описать только самые начала работы с Vi, более подробное введение в этот текстовый редактор можно найти в книге [Курячий:2004](#).

## Запуск vi

Чтобы начать сеанс редактирования в Vim, достаточно выполнить команду Vim 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 возникнуть -- это зависит от версии Vim, установленной в системе.

Для отображения текста и работы с ним Vim 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:

```
#  
~  
~  
~  
~  
~
```

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 Vim 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 Vim пользователю всегда нужно обращать внимание на то, в каком режиме находится редактор.

**режимы**  
**vi**

Состояния редактора Vim, в которых он по-разному обрабатывает нажатия клавиш. Различают три режима Vim: *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*).

Vim 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 Vi, даже в режиме Vim-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*.

---

Не нужно пытаться вводить текст в командном режиме: поскольку у каждой буквы есть командное значение, результат может быть самым неожиданным!

---

Во-вторых, у Vim есть своя командная строка: чтобы её вызвать, нужно ввести ":" в командном режиме. В результате в начале последней строки экрана появится двоеточие -- это приглашение командной строки. Здесь вводятся более сложные команды Vim, которые включают в себя целые слова (например, имена файлов), причём текст набираемой команды, конечно, отображается. Команды передаются Vim клавишей *Enter*. В современных версиях Vim с командной строкой можно работать так же, как и в shell: редактировать её, достраивать команды клавишей *Tab*, пользоваться историей команд.

Главная команда командной строки Vi -- вызов подсистемы помощи ":helpEnter". Двоеточие переводит Vi в режим командной строки, "he lp" -- собственно команда, *Enter* передаёт команду. he lp можно вызывать с аргументом: названием команды или настройки Vi. Vim очень хорошо документирован, поэтому по команде ":help *объект*" можно получить информацию о *любом* свойстве Vi, 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 уместяющиеся 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 `"insert"` -- "вставка"). В этот момент в последней строке появится информационное сообщение о том, что редактор находится в режиме вставки: `" - - INSERT - -"` или `" - - ВСТАВКА - -"`<sup>(4)</sup>, в зависимости от установленного языка системных сообщений.

В режиме вставки можно вводить текст, завершая строку нажатием на *Enter*. Однако нужно помнить, что в некоторых (ортодоксальных) версиях Vim в режиме ввода не работают *никакие* команды перемещения по тексту, здесь можно только набирать. Если Вы заметили, что ошиблись в наборе -- не нужно сразу пытаться переместить курсор и пытаться исправлять ошибку: гораздо удобнее будет вносить 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*.

---

Если Vim пришёл в непонятное для Вас состояние, нажмите *ESC*, чтобы вернуться в командный режим (иногда требуется нажать *ESC* дважды).

---

Мефодий начал упражняться с Vim на файле примеров:

```
methody@oblomov:~ $ vi textfile
Это файл для примеров.
Пример 1.
~
~
~
~
~
"textfile" 2L, 33C          1,1  Весь
```

Vim в командном режиме

Vim начал работу -- как и положено -- в командном режиме. В последней строке возникли некоторые полезные сведения об открытом файле: его имя, общее количество строк ("`2L`"), символов ("`33C`"), позиция курсора ("`1, 1`" -- *номер\_строки, номер\_символа*).

"Весь" обозначает, что всё содержимое файла уместилось на экране терминала. Теперь

Мефодий нажмёт `"i"` и введёт немного текста:

```
Это файл для примеров.
Пример 1.
Моя первая строка в vi!
~
~
```

```

~
~
-- ВСТАВКА --                               3,24  Весь

```

### Vim в режиме вставки

Теперь Vim 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 начнётся ввод символов. Например, по командам "O" и "o" ("open") можно вводить текст с новой строки (до или после текущей), по команде "I" -- с начала строки, команды "a" и "A" ("append") ведут *добавлением* символов (после курсора или в конец строки) и т. п.

## Работа с файлами

Редактируя текст в Vim, пользователь работает не непосредственно с файлом, а с **буфером**. Если открывается уже существующий файл, Vim копирует его содержимое в буфер и отображает буфер на экране. Все изменения, которые делает пользователь, происходят именно в содержимом буфера -- открытый файл пока остаётся неизменным. Если же Vim вызван без параметра, то создаётся пустой буфер, который пока не связан ни с каким файлом(5).

Чтобы записать сделанные изменения в файл, используется команда ":wEnter" (чтобы её отдать, нужно сначала перейти в *командный* режим). О том, что "w" -- это сокращение от англ. "write", "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

```

### 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 ещё не записан ни в каком файле.

Наиболее важна для новичка команда *выхода* из Vim -- ":qEnter" (сокращение от "quit"). 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" Vim обрабатывает по-своему(6). И ":w" и ":q" -- команды режима *командной строки*, в этот режим Vim переводится из *командного режима* с помощью ":" в их начале.

Однако если в буфере есть изменения, которые ещё не записаны ни в каком файле, то Vim откажется выполнять команду ":q", предложив сначала сохранить эти изменения. Если Вы не хотите сохранять изменения, нужно настоять на своём желании выйти из Vim, добавив к команде выхода восклицательный знак ":q!". В этом случае все несохранённые изменения будут выброшены. Восклицательный знак можно добавить в конце любой файловой команды в командной строке Vim, в этом случае Vim будет выполнять команды, не предъявляя пользователю никаких возражений.

В Vim предпринято множество усилий для экономии сил и времени пользователя, командующего редактором. Поэтому вполне можно одним разом записать текст и выйти из редактора: командой ":wq" или аналогичной командой ":x" или просто нажав "ZZ" в командном режиме.

## Перемещение по тексту

При редактировании в тексте всегда есть точка, в которой вы "находитесь" 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 переместиться 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 объёмистого 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, заключённого в скобки, целого текста. Особенно это заметно, если требуется редактировать *структурированный* текст: программу (например, сценарий), конфигурационный файл и т. п. В Vim для каждого такого перемещения предусмотрены специальные команды, обычно состоящие в нажатии *одной* клавиши в командном режиме. Используя их, можно не только одной кнопкой переместиться на любое



расстояние в тексте, но и двигаться по структурным элементам, переходя к предыдущему/следующему слову, предложению, абзацу, скобке и т. д.

---

Прежде чем начинать экспериментировать с перемещением, нужно перейти в командный режим.

---

Итак, передвинуть курсор на начало текущей строки можно командой "0", на первый непробельный символ в строке -- "^", в конец строки -- "\$" (7). Абзацами Vim считает фрагменты текста, разделённые пустой строкой, к началу предыдущего/следующего абзаца можно попасть командами "{" и "}" соответственно. Очень распространённая необходимость -- попасть в самый конец файла: для этого служит команда "G" ("Go"), в самое начало -- "gg" ("go-go").

Передвинуть курсор вперёд на *начало* следующего слова можно командой "w" (от "word", слово), на начало предыдущего -- "b" (от "backward", назад). К началу предыдущего/следующего предложений можно переместиться командами "(" и ")" соответственно. Нужно учитывать, что границы слов и предложений Vim находит по формальным признакам (руководствуясь специально определёнными **регулярными выражениями**), поэтому решение Vim может иногда не совпадать с представлениями пользователя о границах слов и предложений. Однако пользователь всегда властен изменить соответствующие регулярные выражения, подробности -- в документации по Vi.

В Vim никогда не следует вручную повторять по несколько раз одну и ту же команду: если нужно перейти на три слова вперёд, не следует трижды нажимать "w", для повторения команды используется **множитель**. Множитель, это любое число, набранное перед командой Vim: команда будет повторена такое число раз. Например, "3w" -- означает "трижды переместить курсор на слово вперёд", иначе говоря, переместить курсор на три слова вперёд. Обратите внимание, что множители могут применяться не только с командами перемещения, а с *любыми* командами Vim. Аналогично можно переместить курсор на 10 абзацев вперёд 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 возникнет символ "/". Далее следует ввести шаблон для поиска, он будет отображаться в этой строке, его можно редактировать. Обычно Vim настроен таким образом, что шаблон для поиска интерпретируется как регулярное выражение, где ряд символов имеет специальное значение, эту настройку можно выключить (": set nomagic"). После того, как введён 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 Vim -- "`", место в тексте, где было сделано последнее изменение. Подробно об этих и других командах перемещения можно прочесть в руководстве по Vi 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 "delete", удалить, а "dd" -- характерный приём Vim: удвоение команды означает, что её нужно применить к текущей строке.

Из командной строки Vim можно выполнить операцию поиска и замены: для простой строки или для регулярного выражения. Причём синтаксис команды поиска и замены полностью воспроизводит синтаксис потокового редактора sed, о котором уже шла речь в лекции [Работа с текстовыми данными](#).

Это файл для примеров.

Пример 1.

Моя первая строка в vi...

~

~

~

```
:s/.$/.../
```

### Замена по шаблону в Vim

Мефодий хотел заменить в своём файле точки в концах строки многоточиями. Для этого в командном режиме он нажал ":" (вызов командной строки Vim), 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 учёл он следующего: по умолчанию команды 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 расставление символов конца строки, пробелов и табуляций таким образом, чтобы текст хорошо смотрелся на экране терминала. Делать

форматирование вручную крайне неэффективно. В Vi автоматическое форматирование текста (если редактируется программа на каком-либо языке программирования, то с учётом 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 есть 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 именно для *ввода*, и больше ни для чего. Однако в Vi (но не в Vim!) 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" ("previous") -- Vi попытается сам завершить его. Если Vi не угадал и предложил не то слово, можно продолжить перебирать варианты. Команда "^N" ("next") подставляет слова, которые встречаются *ниже* по тексту. Подробнее об этой функции можно узнать из руководства по команде ":help ins-completion".

Иногда, изменив текст, тут же хочется вернуть всё обратно, для этого в Vim предусмотрена команда отмены последнего изменения: "u" в командном режиме (от "undo", отменить). Сколь бы сложным, масштабным (и разрушительным) ни было изменение, совершённое последней командой, "u" вернёт текст в исходное состояние. Впрочем, саму отмену тоже можно отменить. В классическом Vim доступна отмена только последней выполненной команды, в Vi отменить можно сколько угодно последних команд, а также повторить их командой "^R".

## Работа с фрагментами текста

Любая команда перемещения определяет две точки в тексте: та, в которой был курсор до перемещения, и та, в которую он переместился в результате данной команды. Расположенный между этими двумя точками отрезок текста однозначно задаётся командой перемещения. Например, команда ")" захватывает текст от текущего положения курсора до начала следующего предложения. Vim позволяет применить к этому фрагменту любую команду -- так устроены **гнездовые команды**. Гнездовая команда состоит из *действия* и следующего за ним перемещения. Перемещение задаёт 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" ("delete"), "c" ("change"), ">" и "<" (сдвинуть), "u" (запомнить) и "gq" (отформатировать).

**гнездовая команда** Команда редактора Vim, позволяющая применить указанное *действие* к указанному *отрезку текста*. Отрезок задаётся 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. Vim предоставляет возможность упростить эту операцию, сведя два действия к одному: заменить. Гнездовая команда "с" предназначена именно для этого. Например, команда "сw" (буквально: "cange word") заменит текст от курсора до начала следующего слова (так можно заменить одно слово), "с)" 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 своём файле на другие: для этого он в командном режиме подогнал курсор в начало слова "первая" и набрал "с3w" ("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

#### Команда замены в Vim

В примере знак "#" обозначает позицию курсора: как видно, Vim удалил три слова, попавшие в зону действия команды и сразу перешёл 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 Vim не понадобится специальных средств, потому что любая команда удаления ("d", "с", "х" и др.) сохраняет удалённый текст в специальном **регистре**. Для вставки последнего удалённого фрагмента служит команда "р" в командном режиме (от "put", положить). При помощи этой команды Мефодий может вставить только что удалённые им командой "с3w" три слова:

Это файл для примеров.

Пример 1.

Моя новая строка для vi...

первая строка в

~

~

4,1    Весь

#### Вставка удалённого фрагмента

Для того, чтобы скопировать фрагмент текста, служит команда "у" (от "yank", забрать, сдёрнуть): она не удаляет текст, а просто сохраняет его в том же регистре, что и команды удаления. Команду "у" можно использовать в гнездовых командах, например, "у5w" сохранит в регистре фрагмент текста от курсора до начала пятого слова. Вставить скопированный фрагмент можно всё той же командой "р". Однако таким способом можно

вставлять только текст, удалённый или скопированный *последним*, для хранения нескольких разных фрагментов текста следует использовать именованные регистры, см. подробнее в руководстве по Vi.

Для того, чтобы применить команду к нескольким *строкам* текста, не обязательно подгонять к ним курсор. В командной строке Vim любой команде может предшествовать указание **диапазона** в тексте, к которому следует эту команду применить. Команды с указанием диапазона выглядят так: "*: начало , конецкоманда*", 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 "*u*") доступны также и в командной строке. В качестве адреса можно использовать номера строк в файле (команда "*: 1 , 5u*" 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

Удаление диапазона по шаблону. Выполнена команда "*: /Пример 1/ , \$dEnter*".

## Настройка Vi и Vim

Вид и поведение Vim и Vi можно довольно существенно изменить при помощи настроек, приспособив редактор именно к своим вкусам и привычкам. Прямо во время работы редактора можно менять настройки из командной строки Vim при помощи команды "*: set имя\_настройки*". Кроме того, можно сделать настройку постоянной, вписав все нужные значения в конфигурационный файл `.vimrc` (или `.exrc` -- для Vim) в домашнем каталоге пользователя. При каждом запуске Vim/Vi читает этот файл и исполняет все содержащиеся в нём команды. Объём данной 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 Vi (но не в Vim!) 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), в любом случае необходим текстовый редактор. Выбор очень многих пользователей падает на один из двух наиболее популярных и, как следствие, наиболее универсальных (реализованных и работающих *езде*) текстовых редактора: Vim (или одну из его улучшенных версий, чаще всего Vi) и Emacs. Оба эти редактора появились около тридцати лет назад, но их почтенный возраст только пошёл им на пользу: огромное количество пользователей 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 ещё и настолько нетребовательный к интерфейсу: и Vi, и Emacs будут работать на *любом* терминале. Однако ограниченные возможности интерфейса терминала требуют от программ надёжного способа отделения команд редактору от вводимого текста. В Vi и Emacs эта задача решена по-разному -- отсюда большая часть различий в стиле работы с этими редакторами, отсюда и традиционный спор приверженцев этих редакторов о том, который из них лучше.

## Тексты на разных языках

Главное свойство, которое сделало Emacs таким популярным и многофункциональным редактором -- это заложенная в нём 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 любые 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 [Работа с текстовыми данными](#) 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**.

**режим  
emacs**

Комплекс команд и настроек Emacs, предназначенных для работы с текстом определённой 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 определённого вида. Каждый основной режим по-своему переопределяет некоторые управляющие символы, так что наиболее доступными становятся команды, чаще всего нужные 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.

---

Обратите внимание, что в Emacs понятие "режим" имеет совершенно иной смысл, чем в Vim!

---

Не менее хорошо в Emacs развиты средства работы с текстами на самых 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 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**, в которой 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 *Vi* -- 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 её, нажать клавишу с одним из печатных символов (например, "х"). Кроме того, в Emacs используется управляющая клавиша *Meta*, на тех терминалах, где она отсутствует, её 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* неё -- печатного символа (того же "х") эквивалентно "*Meta* х".

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, или просто полное имя команды. Последовательность символов, достаточная для вызова команды, называется **законченным ключом**, а если введённых символов недостаточно для однозначного определения команды, это -- **префиксный ключ**.

Общее правило здесь таково: чем чаще нужна команда, тем короче вызывающий её ключ, и наоборот. Для лаконичной записи длинных клавиатурных комбинаций в сообществе пользователей Emacs сложилась особая традиция сокращённых обозначений. Клавишу *Ctrl* обозначают заглавной буквой "С", *Meta* -- "М"[\(8\)](#). Сочетания с командной клавишей обозначаются дефисом, например, запись *C-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 Vi: 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 ещё один буфер, чтобы отредактировать один из своих сценариев, он запомнил точное название нужного файла и, набрав "*C-x C-f bin/*", нажал *Enter*. При 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
```

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 определённой схеме -- 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**, это 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 пр.

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*. Нарращиваемый 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 такому сочетанию символов после курсора. Поиск в обратном направлении (к началу буфера) осуществляется командой *C-r*. Нарращиваемый 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 доступно множество команд, экономящих усилия при редактировании текста. Если пользователь осознаёт, что набрал что-то неправильно, то можно удалить разом последнее слово (*M-Delete*), предложение (*C-x Delete*). Можно уничтожить и вперёд: до конца слова (*M-d*) и предложения (*M-k*). Emacs хранит не только последний удалённый 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*", для замены регулярного выражения -- 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 её 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 произвести 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 удалённый или скопированный фрагмент можно командой *C-u*.

Есть группа команд, позволяющих работать с меткой более эффективно: установить метку после конца следующего слова (*M-@*), пометить текущий абзац (*M-h*), 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 регистр "x", "*C-x r i x*" -- вставляет текст из этого регистра). В регистрах также можно хранить числа, имена файлов, конфигурацию окон. Подробности как всегда доступны в руководстве.

## Настройка Emacs

Коротко говоря, в Emacs можно настроить всё: 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 браться делать что-то вручную, пользователь обращается к руководству и находит в нём способ выполнить свою задачу максимально быстро и с минимальными затратами ручного труда. Однако если пользователя не устраивает такой принцип работы (когда нужно часто и подолгу читать документацию и думать, 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`, `rioso` (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 Vi и Emacs есть графические варианты (`GVim` и `Emacs-X11` или `XEmacs`), в которых доступны дополнительные возможности графического пользовательского интерфейса: меню, иконки и пр.

---

(1) Эта кажущаяся сегодня тривиальной возможность сколько угодно редактировать текст, не оставляя при этом никаких следов, была радикальнейшим достижением прогресса по сравнению с пишущей машинкой.

(2) Vi, как и большинство основных утилит и принципов, унаследован Linux от UNIX. Название Vim происходит от `visual editor`, "визуальный", экранный редактор, поскольку Vim -- первый редактор для UNIX, использующий весь экран для отображения текста и позволяющий работать с текстом не построчно, а перемещаясь по нему в любых направлениях, как по странице.

(3) Обычно современные терминалы и программы-эмуляторы терминалов поддерживают вывод текста по крайней мере несколькими разными цветами.

(4) Если используется не Vi, или настройка `showmode` по умолчанию запрещена, сообщения может и не быть.

(5) В действительности содержимое буфера хранится в специальном временном файле (`swap file`). Если сеанс работы в редакторе по какой-то причине прервался, то этот временный файл не будет удалён, и при следующей попытке продолжить незаконченную работу с файлом Vim предложит провести процедуру восстановления -- на случай, если во временном файле остались несохранённые изменения.

(6) Управляющая последовательность `^C` приводит к аварийному завершению текущей команды `vi`, а не самого редактора.

(7) Символами `^` и `$` в Linux повсеместно обозначаются начало и конец строки, в частности, в регулярных выражениях.

(8) За свои длинные команды из разных комбинаций управляющих клавиш название Emacs получило шуточную расшифровку: `Escape-Meta-Alt-Control-Shift`.

## 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 совершенно разными людьми, от разработчиков аппаратной составляющей до системного администратора. Предъявляемые к системе требования гибкости, возможности изменять её 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) назовём **досистемной загрузкой**. Начиная с определённого этапа загрузка компьютера уже управляется самим Linux, используются утилиты, сценарии и т. п. Эту стадию (завершающую) назовём **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 виде(1). В *специализированных* компьютерах -- например, в дешёвых игровых приставках -- всё, 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 **энерго-независимую память** (иногда её называют CMOS, иногда -- NVRAM). Объём энерго-независимой памяти очень невелик, а данные из неё сохраняются после выключения компьютера за счёт 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 (жёсткие 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**).

Сокращение от "**Basic Input-Output System**", набор подпрограмм в ПЗУ, **BIOS** предназначенных для простейшего низкоуровневого доступа ко 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 его. Например, выяснить, что устройство для загрузки -- жёсткий 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 передаётся 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 всё 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 определённое место диска, или, на худой конец, положить в заранее определённое место **карты размещения**, описывающую, где именно искать его части (размер вторичного загрузчика ограничен, поэтому и возможно построить такую карту).

**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 приходится на *программную* область. Загрузочный сектор IBM PC, называемый **MBR (master boot record)**, содержит также **таблицу разбиения диска**, структура которой описана в лекции [Работа с внешними устройствами](#). Понятно, что программа такого размера не может похвастаться разнообразием функций. Стандартный для многих систем загрузочный сектор может только считать таблицу разбиения диска, определить т. н. **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 всё 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, позволяющие пользователю самостоятельно *выбрать*, с какого из разделов выполнять загрузку(2). 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 передаётся 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 определённым 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 придёт 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 определённое место, а загрузчик заставить собирать ядро из кусков по карте. Использование карты размещения имеет два существенных недостатка: её *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 ещё раз, *вручную* указав имя файла (или каталога) со старым ядром, под чьим управлением всё 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 отключёнными 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 определённой 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 её порвать: можно, например, загрузить по цепочке MS-DOS, удалить с его помощью раздел Linux, содержащий вторичный загрузчик, переразмечить 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 неё, большинство систем предоставляют *собственные* средства по её организации. В Linux наиболее популярны подсистемы загрузки L I L O (**L**inux **L**Oader) и GRUB (**G**Rand **U**nified **B**ootloader). Обе эти подсистемы имеют текстовый и графический варианты интерфейса, предоставляющего пользователю возможность выбрать определённый заранее настроенный тип загрузки.

### L I L O

Подсистема загрузки L I L O использует и для первичного, и для вторичного загрузчика схему с **картой размещения**. Это делает работу с L I L O 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 L I L O (он называется L I) 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
```

Простейшая настройка L I L O: пример файла `lilo.conf`

Такая настройка L I L O определяет ровно один вариант загрузки: **первичный загрузчик** записывается в начало первого жёсткого диска (строка `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
```

### Настройка LILO на двухсистемной машине

Здесь Linux был установлен на *пятый* раздел диска (о нумерации разделов в IBM-совместимых компьютерах рассказано в лекции [Работа с внешними устройствами](#)), а на первом находится MS-DOS. Кроме загрузки MS-DOS предусмотрено два варианта загрузки Linux и ещё один -- любой операционной системы с дискеты. Каждый вариант загрузки помечен строкой `label=вариант`. При старте LILO выводит простейшее(3) 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, по истечении тайм-аута выбирается метка, указанная в поле `default`.

Ещё 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 посвящён раздел [Стартовый виртуальный диск и модули ядра](#) этой лекции), а внушающая некоторые опасения надпись "unsafe" (для метки floppy) означает всего лишь, что дискета -- съёмное устройство, поэтому бессмысленно во время запуска lilo проверять правильность её загрузочного сектора и составлять карту.

Наконец, записи вида `other=устройство` говорят о том, что LILLO неизвестен тип операционной системы, находящейся на этом устройстве, а значит, загрузить ядро невозможно. Зато ожидается, что в первом секторе *устройства* будет обнаружен *ещё один* первичный загрузчик, LILLO загрузит его и передаст управление по цепочке. Так и загружается MS-DOS на этой машине: первичный загрузчик берётся (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**, располагаемый в файловой системе. Однако **карта размещения** в GRUB обычно используется только для т. н. "полуторного" загрузчика ("stage 1.5") -- по сути дела, драйвера одной определённой 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 подряд в нескольких секторах(4), или в ином специально отведённом 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 LILLO. Таким образом, обновление и перенастройка установленного GRUB *не требует* пересчёта 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
```

## Настройка GRUB на двухсистемной машине

Разница между `lilo.conf` только в синтаксисе, да ещё в том, что жёсткие диски и разделы на них GRUB именуется по-своему, в виде (`hdномер_диска, номер_раздела`), причём 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 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 ещё не загружена. Мало того, можно по-своему перенастроить загрузчик и *записать* результаты настройки. Так и не успев сполна насладиться неожиданной свободой, Мефодий в один прекрасный день обнаружил, что выход в командную строку защищён 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 передаёт ему управление.

2. **Первичный загрузчик** определяет (а чаще всего -- знает), где находится вторичный загрузчик -- большая и довольно умная 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 передаёт ему управление.

3. **Вторичный загрузчик** достаточно умён, чтобы знать, где находится **ядро** системы (возможно, не одно), предлагать пользователю несколько вариантов загрузки на выбор, и даже, в случае GRUB, разрешает задавать собственные варианты загрузки. Его задача -- загрузить в память ядро и всё необходимое для старта системы (иногда -- модули, иногда -- стартовый виртуальный диск), настроить всё 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, т. н. "режиме супервизора", позволяющем ему иметь доступ сразу ко всей оперативной памяти и аппаратной таблице задач. Процессы запускаются в "режиме пользователя": каждый жёстко 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.

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").

Работа ядра после того, как ему передано управление, и до того, как оно начнёт работать в штатном режиме, выполняя системные вызовы, сводится 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 который оно угодило, 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 ещё 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, этот режим позволит, не усугубляя положение, *прочитать* и запустить утилиту `fsck` (`file system check`). 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, очень старое изобретение, он чуть ли не старше самой истории (истории UNIX, конечно), и с давних пор его идентификатор равен 1.

## Загрузка системы

С запуска `init` начинается загрузка самой системы. В давние времена, во времена молодости Linux и ранее, в *этом* месте никаких подводных камней не наблюдалось. Если ядро содержало подпрограммы для работы со *всеми* необходимыми устройствами (т. н. "драйверы"), оно загружалось и запускало `init`. Если ядру недоставало каких-то важных драйверов (например, поддержки дискового массива, с которого и шла загрузка) -- 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. Загрузивши 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 нужны для работы, например, звуковые), и компилировал из исходных текстов новое, **профильное ядро**.

## Стартовый виртуальный диск и модули ядра

Пересборка ядра нынче требуется очень редко. Во-первых, в Linux поддерживается *несметное* число 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 годится. Во-вторых, выяснением того, какой именно драйвер необходим найденному устройству, занимаются сейчас специальные программы, в распоряжении которых есть целые базы данных -- ядру такую работу делать 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 того, кто его пересобирает. Этот процесс нельзя ни автоматизировать, ни упростить. Утилита `linuxconf`, устроенная именно для этого на основе окон и меню, даёт на выходе работоспособное ядро в трёх случаях: (1) в руках профессионала, (2) при чётком следовании *полной* инструкции и (3) по случайности.

---

Не надо вручную пересобирать ядро, даже если учебник по Linux это советует!

---

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 определённым устройством, которые *добавляются* к возможностям ядра(5). 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]
```

### 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 определяются уже самой системой, тогда можно будет и распознать экзотическую аппаратуру, и модуль для неё подгрузить. Однако полностью перевести драйверы всех внешних устройств в модули мешает вот какое соображение: что, если загрузка системы происходит *именно* с того устройства, чей модуль ещё *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 рассказано в лекции [Работа с внешними устройствами](#)). Осталось только обучить подсистемы загрузки GRUB и LILO считывать не одну, а две области данных -- ядро и *образ* файловой системы. Ядру при этом передаётся 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` (**initial ram disk**, где "ram" -- это не "баран", а **random access memory**, 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, причём 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
```

Размеры и наименование файлов с ядром и стартовым виртуальным диском

Как видно из примера, ядро в четыре раза превосходит размером стартовый виртуальный диск. Стоит заметить, что и ядро, и образ диска *упакованы* с помощью утилиты `gzip` (причём 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 символическими 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` (**uniprocessor**, 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 сказано иного, 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 не он сам, то 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` 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*.

Наконец-то Мефодий до конца понял, отчего `getty` ведёт 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 определённым PID, а `init` бездействует до тех пор, пока существует процесс с этим PID `getty`, **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 обеспечения 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 всё время работы системы находится в таблице процессов, однако большей частью бездействует, ожидая, когда её 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 даёт совершать плохое и потворствует хорошему, такую программу стали называть "daemon". Не знакомые с творчеством Платона программисты-любители частенько переименовывали её в "demon" (**демон**); к сожалению, именно в такой, слегка inferнальной форме, daemon и вошёл в русскоязычную терминологию. Выходит, что в Linux услуги пользователям предоставляют... демоны!

Запускаемая в фоне программа, длительное время пребывающая в таблице **демон** процессов. Обычно демон активизируется по запросу пользовательской программы, по сетевому запросу или по наступлению какого-либо системного события.

В ранних версиях UNIX *всё*, 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` ждёт, 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 создаётся **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, загрузки 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 определённом 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, легко 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 символическая 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 его(8).

```
[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
```

### 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` полностью одинаковы, а от `nosound` оба отличаются тем, что со второй строки по пятую содержат названия тех самых модулей ядра (среди которых затесался `gameport`, отвечающий за джойстик).

### Схема ".d"

Итак, имеется 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 ещё не решается. Первая из возникающих задач такова: чаще всего нужно загружать *не все* из размещённых 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 под именем "схема. d". Суть её в следующем. Пусть некоторый процесс управляется конфигурационным файлом, содержимое которого зависит от наличия и активации других служб системы (таким процессом может быть демон централизованной журнализации `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/xinetd.conf` заводится каталог `/etc/xinetd.d`). Каждый файл в таком каталоге соответствует настройке одной службы: при добавлении её в систему файл появляется, при удалении -- исчезает. Остаётся только обучить тот же `xinetd` читать настройки не из одного `xinetd.conf`, но и из *всех* файлов в `xinetd.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` не годится на роль ". d"-каталога. Впрочем, достаточно организовать ещё 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 алфавитно, `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 их ". d"-каталога. Необходимость выбора возникает также, 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 **уровни 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 *отключённой 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 работают 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 **останову и перезагрузке** системы. В сущности, это удобные упрощения для действий, обратных загрузке на уровень: все службы останавливаются, диски размонтируются. В случае останова даже электропитание можно отключать программно, если аппаратура позволяет, а в случае перезагрузки система идёт 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 происходит очень просто: по команде `init номер_уровня`. На какой уровень загружаться при старте системы, написано в `inittab` (в поле *действие* 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
```

## Задание и просмотр уровня выполнения

**уровень выполнения** Сохранённый профиль загрузки системы. В Linux реализован выполнением всех сценариев остановки и запуска служб из подкаталога `rc.уровеньd` каталога `/etc` или `/etc/rc.d`

Схема ".d" легко учитывает уровни выполнения. В каталоге `/etc/rc.d` 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
```

## 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 приведённом примере при переходе на уровень 2 останавливаются несколько служб, в том числе сетевой метадемон (`K50xinetd`) и монтирование по сети удалённых файловых систем (`K75netfs`). Если при переходе с уровня на уровень какой 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 символичные ссылки в каталоги `rc*.d` приходилось вручную. Однако одно из другого не следует: можно предусмотреть и более лёгкий способ наполнения этих каталогов ссылками. Один из способов такой: поставить в стартовый сценарий комментарий особого вида, в котором и 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
```

## 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 `SVKЛnetfs`, а во всех остальных -- `KBЫKЛnetfs`. Эта же утилита позволяет добавлять и удалять службу на каждом уровне в отдельности или запрещать её 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:
```

### 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 всё, без чего дальнейшая полноценная загрузка системы невозможна. Далее из строчки с "initdefault" `init` узнаёт, 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` ждёт, 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
```

### Профиль системы на уровне выполнения 5

Мефодий заметил, сценарий `K20nfs` (с параметром "stop") не выполнялся: 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 выполнялся.

Из служб, запускаемых именно на пятом уровне, примечателен шрифтовый сервер, под номером 44, (**the X font server**, `xfs`) -- программа, у которой графическая подсистема получает шрифты (нередко по сети, тогда такой сервер может быть один на несколько рабочих станций), и **экранный диспетчер**(10), под номером 45, (**the X display manager**, `xdm`) -- программа, регистрирующая пользователя на манер `login`, с той разницей, что регистрация и запуск графических приложений могут происходить по сети с удалённого компьютера. Тут разрешилась ещё одна загадка: вместо обычной виртуальной консоли и `login`-а, 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 [Графический интерфейс \(X11\)](#)) 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 счёт дополнительных свойств файловой системы (журнализация, сводящая вероятность порчи к минимуму, логически упорядоченная запись и т. п.), с другой -- за счёт некоторых предположений, которые делает сама утилита `fsck`. Однако надеяться на неё нельзя: очень редко, но бывают неразрешимые противоречия в лишённой 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 всё время взаимодействия с администратором система будет работать в **однопользовательском режиме**, причём администратору предстоит разбираться с тем, что произошло, 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 продолжают нормально работать.

В экстренных случаях (например, когда при сбое электропитания демон, обслуживающий устройство бесперебойного снабжения, сообщает, что ресурсы на исходе), безопаснее всё-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 внезапной потере 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 в системе *программ*, что займёт минуту-две. А вот с *пользователями*, работающими в системе, всё 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 в работе системы, после чего уже запускает `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
. . .
```

## Использование shutdown

Остаётся заметить, что у `shutdown` есть обязательный параметр -- время начала останова, в примере он равен "+3", то есть "через три минуты", и необязательный -- "- r" (`reboot`, перезагрузка) или "- h" (`halt`, останов). Без необязательных параметров выполняется переход на *первый* уровень выполнения, причём 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`.

---

(1) Современные компьютеры используют *программируемые* ПЗУ, содержимое которых *можно* изменять, однако такое изменение всегда считается ситуацией нештатной: например, запись новой версии содержимого ПЗУ, в которой исправлены ошибки (`upgrade`).

(2) Например, `BOOTACTV` из пакета `pfdisk` или стандартный для FreeBSD предзагрузчик `boot0`, которые, в силу их *досистемности*, можно применять где угодно.

(3) Если установлен графический вариант интерфейса, то -- сколь угодно изукрашенное.

(4) Т. е. на нулевой дорожке нулевого цилиндра, начиная с сектора 2. Эта область диска часто не используется под файловые системы (см. лекцию [Работа с внешними устройствами](#)).

(5) Этим он скорее похож на динамическую библиотеку.

(6) `mingetty` -- упрощённый аналог `getty`, работающий только на виртуальных консолях.

(7) Понятно, что `Ctrl+Alt+Del` -- это не `reset`, а обычное сочетание клавиш. Для удобства пользователя его специально распознаёт клавиатурный драйвер, а ядро сообщает об этом `init`-у.

(8) В некоторых дистрибутивах Linux такая команда может называться `invoke-rc.d`, а команда, аналогичная описанному ниже `chkconfig -- update-rc.d`.

(9) В некоторых дистрибутивах -- в каталоге `/etc/`

(10) Не путать с **диспетчером окон**, описанным в лекции [Графический интерфейс](#)  
(X11).

# 11. Работа с внешними устройствами

## Представление устройства в системе

В лекции [Этапы загрузки системы](#) 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 найдётся 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 и обратное неверно: если производитель создаёт новое устройство, управлять которым нужно по-новому, а распознаётся 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 перестаёт распознаваться до тех пор, пока автор соответствующего модуля Linux не заметит это 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
```

Обращение к несуществующему устройству

Здесь Мефодий попытался прочитать что-либо из устройства `/dev/sdg14`, что соответствует четырнадцатому разделу SCSI-диска под номером семь. Такого диска в этой машине, конечно, нет, а файл-дырка для него заведён 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 её объёма, индексных дескрипторов должен быть изрядный запас.

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 иерархизированную 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, съёмных жёстких дисков, "на лету"). Однако он требует соблюдать дополнительную логику "привязки" найденного устройства к имени, иногда весьма запутанную из-за той же нечёткой 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 -- символичные 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
```

### Идентификация внешних устройства в /dev/

Файл-дырка, как и всякая дырка, не имеет никакого размера: сколько в неё не записывай, в файл на диске ничего не попадёт. Вместо этого ядро передаёт всё 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 неё 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, кто сам всё выдумывает: это и /dev/null, чёрная дыра, в которую что угодно можно записать, и оно пропадёт безвозвратно, и /dev/zero, из которого можно считать сколько угодно нулей (на запись оно ведёт 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 про символическую ссылку, а про файл, 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, грубо 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/random 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* (block) устройства, обмен данными с которыми возможен только порциями (блоками) определённого размера, и *символьные* (character) 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 ещё два типа специальных файлов -- **каналы** и **сокет**. **Канал-файл** (или **fifo**) называют ещё **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, причём 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
```

#### Использование именованного канала

Здесь важно, что утилита `head` показывает начало не "файла" `hole` а именно *последней записываемой* порции данных, как и подобает трубе(1).

Объект Linux, используемый в межпроцессном взаимодействии. Доступен в виде двух **дескрипторов**: один открыт на запись, другой -- на чтение. Все данные, **канал** 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 нём не работают. Процесс, открывший сокет, считается *сервером*:

он постоянно "слушает", нет ли в нём 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 (т. н. unix domain socket), а сетевой адрес и **порт** удалённого компьютера (например internet socket, если подключаться с помощью сети Internet).

## Драйверы устройств

Как уже говорилось в лекции [Этапы загрузки системы](#), 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 всё, 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 -- малопознавательного) 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 представлено отдельной операцией, оно выполняется автоматически при перенаправлении ввода (открытие 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 всё равно передаётся 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 жёсткие 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 объём 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()` (`input-output control`): 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 определённом 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, 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
```

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 в качестве рабочей станции именно этого пользователя, а все последующие доступа к этим устройствам не получают(2). 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
```

Кому принадлежат устройства?

При этом для того, чтобы обеспечить и другим -- псевдо- или настоящим -- пользователям, такие устройства также принадлежат определённым 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, сможет до этого устройства добраться (а можно и просто включить опытного пользователя в такую группу).

## Разметка диска и именование устройств

В начале лекции говорилось о том, что **младший номер устройства**, соответствующего жёсткому диску, обычно указывает на определённый раздел этого диска. Поначалу Мефодию казалось, что смысла "пилить" диск на несколько разделов нет никакого: известно, что один большой раздел файловой системы Linux(3) вмещает чуть больше данных, чем несколько маленьких того же объёма. Кроме того, разбивая диск на разделы, можно не предугадать подходящие размеры этих разделов, и тогда размещение на них файловой системы Linux окажется делом нелёгким, если вообще возможным, так как структура дерева каталогов Linux строго определена стандартом FHS (см. лекцию [Структура файловой системы](#)).

**раздел диска** Часть жёсткого диска, используемая под определённые 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 надёжность хранения данных в них важнее быстродействия и насколько ситуация **переполнения файловой системы** опасна и может помешать работе. Поэтому стоит каталог /tmp, требующий очень частой записи, но не требующий надёжного хранения данных после перезагрузки, держать *не* на том же разделе, что и корневую файловую систему, запись в которую происходит редко (в каталог /etc), но требует повышенной надёжности. В отдельный раздел можно поместить весь каталог /usr, так как он *вообще* не требует операций записи. Наконец, такие каталоги, как /var или /home, суммарный объём файлов в которых с трудом поддаётся 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 этой и предыдущей лекции Мефодий работает именно за такой машиной: помимо Linux, на ней установлен FreeDOS для запуска одной-единственной программы.

## Разметка диска IBM-совместимого компьютера

**таблица разделов** Небольшая часть жёсткого диска, описывающая геометрию и тип его разделов. Стандартная таблица разделов диска IBM-совместимого компьютера может содержать не более четырёх 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 (**hard disk partition table**) -- располагается в конце самого первого сектора диска, после **предзагрузчика** (**master boot record**, MBR) и состоит из четырёх 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 (**hard disk "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 жёсткий диск -- первый в системе (hda), а лазерный накопитель -- *третий* (hdc), второго же вовсе нет. Устройства, соответствующие основным разделам диска, называются /dev/hd*букваномер*, 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 четвёртый:

```
[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
```

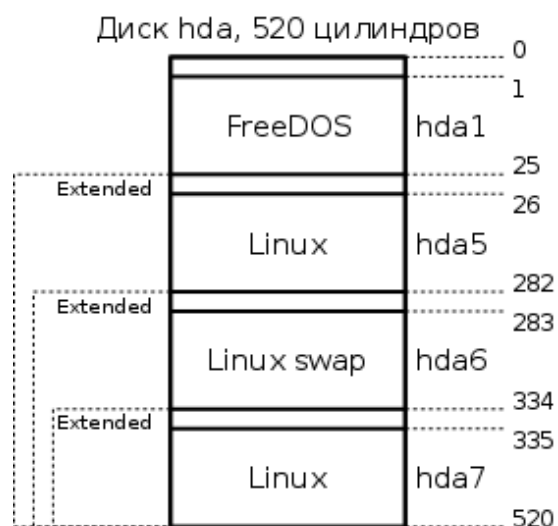
Просмотр таблицы разделов жёсткого диска

Дело в том, что четырёх разделов редко когда бывает достаточно. Куда же помещать дополнительные поля таблицы разбиения? Создатели IBM PC предложили универсальный

способ: один из четырёх основных разделов объявляется **расширенным** (extended partition); он, как правило, занимает *всё* 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, причём один из подразделов может быть, опять-таки, расширенным, со своими подразделами и т. д.

Осторожнее с программой `fdisk`! Она предназначена для создания, изменения и *удаления* разделов диска.

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 четырёх 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 так:



Разбиение диска двухсистемного компьютера

И разделы, и таблицы разбиения принято размещать с начала *цилиндра* (термин, имеющий отношение к внутреннему устройству жёсткого диска), так что при заведении каждого расширенного раздела на этом компьютере тратилось впустую по четыре мегабайта (таков, по словам `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 всё, 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 -- т. н. **область подкачки** (swap space), пространство на диске, используемое системой для организации **виртуальной памяти**. Оказывается, областям оперативной памяти, которые процессы запрашивают у ядра, не всегда соответствуют части физической оперативной памяти. Если процесс долгое время не использует заказанную оперативную память, её 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* её 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 её есть в действительности, и преспокойно работают, потому что не используют *всё* заказанное пространство *сразу*, позволяя системе *откачивать* неиспользуемые области. К тому же многие процессы (особенно демоны) не работают *постоянно*, а ждут наступления определённого события, и чем дольше они ждут, тем дольше не используют оперативную память, и тем выше вероятность, что ядро откачает её.

Стоит отдавать себе отчёт, что если эти самые процессы *вдруг* захотят работать одновременно и со всеми областями памяти, ядру придётся туго. Большую часть времени система будет проводить откачивая и подкачивая, потому что дисковые операции чтения и записи работают в тысячи раз медленнее, чем запись и чтение из оперативной памяти(4). Чтобы хоть как-то облегчить ему жизнь, 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 [Структура файловой системы](#) Мефодий узнал, как *пользоваться* файловой системой и какую структуру она имеет с точки зрения программы, работающей с файлами в ней. Про то, как организована файловая система *изнутри*, что именно находится от начала устройства hda5 до его конца, существует немало больших статей, защищено немало кандидатских (и более одной докторской) диссертаций. Разработка файловой системы -- сложный и интересный процесс, требующий одновременно владения высшей математикой, статистикой, умения безошибочно программировать и полного знания того, как работает то или иное дисковое устройство. Поэтому файловых систем не так много, и каждая из них

устроена по-особому, в соответствии с тем, как решал тот или иной творческий коллектив задачу быстрого и надёжного доступа к файлам.

## Принципы организации данных на диске

Во всех файловых системах есть и 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**, в 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, жёсткие диски). Чтение (и запись) данных на устройствах последовательного доступа идёт 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, причём она тем длиннее, чем дальше отстоит нужный блок от текущего: лента перематывается. Работа с устройствами прямого доступа легче: каков бы ни был текущий прочитанный блок, время, за которое будет прочитан *любой другой* примерно одинаково.

Файлы на магнитной ленте удобнее хранить целиком, каждый файл -- одним длинным куском. У такого способа есть один существенный недостаток: если на ленту объёмом в 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 *трёх* 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 нём в **области данных по частям**, а карта размещения этих частей будет записана в **системную область**. Если, не мудрствуя особо, 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 всё же системная область занимает примерно такую же (от пяти до десяти процентов) долю общего дискового пространства.

**индексный дескриптор**    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 жёстком диске блоки, расположенные *подряд*, считываются (и записываются) быстрее, чем блоки, расположенные как попало. Эффект связан с механическим устройством жёстких дисков, пояснить 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 дойдёт до диска, система вернёт 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.

Ещё эффективней кеш на *запись*: операции записи накапливаются в памяти, а до диска добираются не сразу, и в том порядке, в каком быстрее пройдёт запись, а не в том, в каком были выполнены. Если запись шла во временный файл, который, в конце концов, удалили, обращений к диску может и вообще не случиться. Однако с кешированием операций записи следует обращаться бережно: а вдруг сбой в электроснабжении произойдёт именно тогда, когда часть данных уже записана, а часть -- ещё нет? А если не полностью, кусочками, обновилась системная область, состояние файловой системы после того, как питание опять включают, может оказаться совсем плачевным -- настолько, что даже умная утилита восстановления `fsck` может оказаться бессильной. Поэтому системные области либо вообще не кешируются на запись, либо исключительно с помощью будущих кандидатов и докторов наук, что 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

Из лекции [Структура файловой системы](#) Мефодий знает, что файловые системы на различных разделах "прививаются" в виде ветвей общего дерева каталогов, растущего из `/`. Делается это при помощи команды `mount` -  
*о 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, размещённой 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
```

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 каталогов(5). 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 её каталог в качестве текущего.

Для того, чтобы файловые системы монтировались при загрузке системы, 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
```

Содержимое `/etc/fstab`

Первое поле каждой строки этого файла -- устройство или название **виртуальной файловой системы**, второе -- точка монтирования, третье -- *тип* файловой системы, четвёртое -- 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 *усечённый* 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 загрузке системы какое-либо устройство не монтировалось, а усечённым `mount` его можно было смонтировать вручную, в поле "настройки монтирования" добавляется ключевое слово `noauto`.

Две последних строки относятся к монтированию *съёмных* (removable) носителей: лазерного и гибкого дисков. Съёмные носители приходится монтировать гораздо чаще несъёмных, не во время загрузки системы, а всякий раз, когда носитель сменился, и содержимое нового необходимо пользователю. Мало того, надо разрешить выполнять операцию `mount` *пользователю*, который принёс дискету и желает поработать с ней. С другой стороны, нельзя всем и каждому давать право запускать `mount` и особенно `umount` с *любыми* параметрами! Есть четыре способа разрешить возникающее противоречие:

1. Воспользоваться усечённым 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 усечённым вариантом `mount` и настройкой `owner` в `fstab`, которая позволяет выполнять операцию монтирования *хозяину* устройства; при этом `/dev/hdc` отдаётся 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 монтирует 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 с тем, поменялся ли носитель, разбирается самостоятельно. Этот способ лучше предыдущего тем, что пользователю не приходится ждать "долгое время", пока система не соизволит размонтировать и "отдать" лазерный диск. Кроме того, `subfs` может снабжать пользователя данными из кеша, даже если диск давно уже вынут (речь идёт, 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 нём 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 "защёлку", не позволяющую извлечь диск, пока он используется, а дисководы или устройства USB Flash -- нет (хотя, казалось бы, она нужнее там, где происходит запись). Единственная надежда -- на то, что пользователь не будет выдёргивать дискету из дисковода, пока он занимается записью, и на нём горит зелёная 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 жёстких 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** ведёт постоянный учёт 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 определённое 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 ещё не успели завершиться, записаны в журнале, так что стоит после включения компьютера "проиграть" их ещё раз, и файловая система войдёт 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 всё 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 неё используется своя 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
```

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" всё 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 святой святым системы. В Linux `ps` может быть переписана чуть ли что 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
OME=/TERM=linux

```

## Виртуальная файловая система PROCFS

В частности, подкаталоги `/proc` с числовыми именами содержат информацию о процессах с соответствующими PID. Файл `exe` такого подкаталога -- символическая 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 симитировать 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 ещё 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 ещё куда-нибудь, например, в сеть. Так работают **удалённые 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 удалёнными файловыми системами Windows, причём как на уровне модулей ядра, с помощью `mount -t smbfs`), так и без монтирования, с помощью утилиты `smbclient`. Linux и сам может служить сервером, предоставляющим удалённый доступ к файлам, причём 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)
```

Монтирование содержимого файла при помощи `mount -o loop`

Как заметил Мефодий, `mount` создаёт для такого способа монтирования специальное устройство -- `/dev/loop0`, и уже с его помощью работает с файлом.

Обширное поле для экспериментов -- т. н. *пользовательская файловая система* (**linux userland file system, 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**heck). 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 [Этапы загрузки системы](#), `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 всё в порядке, её **журнал** не испорчен, и `fsck` "проигрывает" его, после чего всё 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
```

## Использование `fsck`

Со второго раза `fsck`[\(6\)](#) работать тоже не захотела, ссылаясь на то, что файловая система и так находится в штатном состоянии (её 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**, её 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, остаётся пробовать загрузку с других носителей (например, с установочного CD), и разбираться.

Если какая-то порча файловой системы всё-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 (т. н. *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 приведёт 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 такие запросы администратор заранее ответил `"yes"`.

Когда-то такие вот 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-трёх мгновенных воспроизведений журнала и жестокого нагоняя от Гуревича.

---

(1) Если стандартный вывод ошибок всего конвейера перенаправлен в `/dev/null`, то командный интерпретатор не выводит сообщений о запуске и остановке фонового процесса.

(2) Что называется, "кто первым встал -- того и тапки".

(3) Для некоторых других файловых систем, например, для `vfat`, это неверно.

(4) Такая ситуация называется "дребезг" (*trashing*) и свидетельствует о том, что для текущих задач компьютеру требуется больше физической памяти.

(5) Мефодий заметил, что `/tmp` и `/var` не смонтированы никуда, и, следовательно, корневая файловая система, вопреки рекомендациям FHS, слишком часто используется на запись.

(6) Мефодий заметил, что для файловой системы Ext3 запустилась специализированная версия `e2fsck`, подходящая также и для Ext2.

## 12. Конфигурационные файлы

### Проектирование свойств системы

Операционная система, позволяющая задействовать все возможности компьютера, резко отличается от специализированного программного обеспечения огромным числом т. н. "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 её 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: где? на какие кнопки нажимать? А кнопок-то и нет...

Можно попытаться описать операционную систему как большой и сложный универсальный инструмент решения любых задач. Предполагается, что пользователь, прочтя документацию, в которой описывается *как* работает система и *как* применять её в различных ситуациях, сможет решать и свои задачи. Правда, для этого ему придётся 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 займёт 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 и как 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. лекцию [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)). Вывод: пользователь -- не разработчик, ему всё-таки важнее быстро и качественно решить задачу, чем долго и качественно улучшать инструмент решения.

Можно пойти обратным путём: 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 обычно 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 неопределённой форме ("хочу, чтобы был текст", "хочу, чтобы играла музыка"), этот способ работает на ура: та же игровая приставка -- это отличное решение крайне неопределённой 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 то из готовых "решений", которое нанесёт меньше вреда.

Что же нужно идеальному -- достаточно подготовленному, чтобы действовать самостоятельно, и достаточно занятому, чтобы не переделывать системы -- 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 её 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**. Всё, что не потребует вмешательства пользователя, необходимо запрограммировать и использовать 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, являясь как бы строительным материалом, деталями и инструментами сборки "больших" решений.

Всё, чего может коснуться рука человека, из реализации переносится в **профиль** системы. Профиль задаёт *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 её заранее известную, но достаточно общую функциональность.

Таким образом система *полностью* описывается в виде набора необходимых компонентов реализации, активизированных (запущенных) с определёнными 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 задаёт её 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 нём нельзя.

**профиль** Изменяемая часть системы, задающая её поведение во время работы.

Как проще всего *создать* профиль, если не всей системы, то хотя бы её компонента (программы)? Один из вариантов такой: снабдить программу функцией "сохранить настройки", тогда можно будет эту программу запустить, *любым способом* добиться её работоспособности, а после зафиксировать достигнутое состояние с помощью этой функции. При этом по началу 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, неведь как приводит систему в работоспособное состояние. Такая подсистема и называется-то "wizard", причём в русском переводе её 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 <configfile-name>
      (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<Info>: Speed 115200; init "ATQ0 V1 E1 S0=0 &C1 &D2 +FCLASS=0"
```

Кудесник `wvdialconf`

Даже ни о каких наводящих вопросах 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` создаёт именно профиль) лежит теперь в файле `.wvdialrc`, так что программа `wvdial` начнёт 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 определённой 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`, причём четыре из них запускаются с изменённым 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 создаётся 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 то, что в Linux есть высокоразвитая система хранения и *переработки* (как ручной, так и автоматической) данных в текстовом виде, изобретать какой-то новый формат не лучше, чем изобретать новый велосипед. Тем более, что именно текст, разделённый на строки и слова, лучше всего подходит тогда, когда есть чёткое 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 <F2> :wall!^M
map! <F2> ^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
```

### Настройки редактора vim

Вот как выглядит конфигурационный файл для Vi, написанный Мефодием на основе 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 [Текстовые редакторы](#)). 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
```

### 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 всё преимущество текстового формата сойдёт на нет. Например, стандартный конфигурационный файл утилиты `ls` (точнее, только её 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 безо 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. Во-первых, уже известный по лекции [Этапы загрузки системы](#) механизм ".d": файл разделяется на несколько *независимых* друг от друга 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 объёма 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. Дополнительная выгода 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 съёмного 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 съёмными устройствами системы. Если администратору нужно как-то изменить поведение updfstab в отношении определённого устройства, он копирует соответствующую группу настроек из updfstab.conf.default в updfstab.conf *после* строки 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 -- это *секционирование* профиля, когда все настройки разбиваются на группы, каждой группе даётся 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 разбиение на файлы неудобно (например, если группы не полностью независимы, поэтому может понадобиться редактировать их сразу и несколько). Конфигурационный файл утилиты дозвона wvdial, например, секционируется по адресату (провайдеру) плюс отдельная секция "по умолчанию". Сами секции отделяются друг от друга заголовками, заключёнными 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 = gimmethemun  
Force Address=10.0.0.120
```

### 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, затем запустить на *сервере* сетевой демон `pppd`, и только затем 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 всём этом и о многом другом `wvdial` умеет догадываться, как `wvdialconf` умел определять, какое же из устройств -- модем.

Однако на любой искусственный интеллект найдётся 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, всё ждут, пока кто-нибудь не проявит себя... Помогает настройка `TOnline`, которая заставляет `wvdial` немедленно использовать протокол `ppp`, на что сервер, подумавши "ах, `ppp!`", с облегчением запускает `pppd`. Остаётся вопросом: почему эта полезная настройка *никак* не отражена в документации (её нашёл в *исходных текстах* программы Гуревич)? Не потому ли, что пара `wvdialconf-wvdial` не по Linux-овски стремится всё делать за пользователя, а стало быть, пользовательская документация для разработчиков этой программы -- не главное?..

Идею чтения настроек по умолчанию можно развить чуть дальше. 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 из себя представляет, при этом отпадает необходимость сначала находить строчку в файле, а потом искать её же в руководстве. Такой способ оформления называется **самодокументированием** профиля и часто используется. 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 xyzy" will show all man pages for xyzy.
# When CMP is defined man will try to avoid showing the same
# text twice. (But compressed pages compare unequal.)
#
CMP                /usr/bin/cmp -s
. . .
```

### 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, предварительно они сравниваются, и показываются только несовпадающие.

Если пойти ещё 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 `Vi` 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 определённого настроечного файла может происходить неявно: например, строка `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-браузера` `lynx` не просто хорошо документирован, но и размечен теми же средствами, какие используются в самом браузере для представления `HTML`).

## Изменение конфигурационных файлов

Как правило, конфигурационный файл считывается программой при запуске, отражая, таким образом, её 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 есть 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 [Этапы загрузки системы](#). Чего *не* стоит искать в /etc, так это разнообразных *примеров* настройки той или иной службы. Считается, что пример -- это часть документации, и их стоит 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 т. н. "песочниц", расположенных обычно в `/var/lib`.

Смысл термина "песочница" вот в чём. 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* кажутся *команде* каталогами первого уровня вложенности, и т. д. Если необходимо во что бы то ни стало *ограничить* область действия некоторой утилиты (например, по причине её небезопасности), можно запускать её 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 утилита "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 создаётся 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.

# Системные конфигурационные файлы

## Подсистема учётных записей

Несколько конфигурационных файлов и способы работы с ними заслуживают отдельного рассмотрения. В первую очередь Мефодий заинтересовался системой учётных записей, о которой упоминалось сразу в нескольких предыдущих лекциях. Итак, существует два файла, доступных для чтения всем пользователям: `/etc/passwd`, хранящий учётные 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:
```

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 раньше, 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 введённого им пароля изготавливается ещё один ключ. Если он совпадает с тем, что хранится в учётной 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 втихомолку на каком-нибудь суперкомпьютере. В конце концов, ключ не нужен никому, кроме подсистемы идентификации, поэтому его, вместе с другими полями учётной записи, о которых знать *всем* не обязательно, из `/etc/passwd` перенесли в "теневой" файл учётных 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 четвёртое поля `passwd` -- **идентификатор пользователя** и **идентификатор группы по умолчанию**. Как уже говорилось в лекции [Права доступа](#) именно идентификатор пользователя, а не его входное имя, *однозначно* определяет пользователя в системе и его права. Вполне возможно создать несколько учётных 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 одинаковый 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`, содержащего *допустимые* командные интерпретаторы, неизбежны трудности при идентификации пользователя.

Строки файла `/etc/group` состоят из четырёх полей, причём второе -- ключ группового пароля -- обычно не используется. Первое поле -- это имя группы, третье -- это **идентификатор группы**, а четвёртое -- это список **входных имён** 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 учётной записи: нижняя и верхняя граница времени жизни пароля, самой учётной записи, дата последнего изменения и т. п. Ключ пароля (второе поле) указывается в одном из поддерживаемых форматов, а если формат не опознан, вся учётная запись считается недействительной. Поэтому один из способов запретить регистрацию пользователя в системе -- добавить один символ (например, "!") прямо в поле ключа, отчего всё 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 учётную 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 `vi` или `vi` `gr` (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
```

### 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 учётной 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 всё равно невозможно.

## Подсистема идентификации

Подсистемой учётных записей пользуется подсистема **идентификации**, которая в Linux имеет модульную структуру и называется PAM (**P**luggable **A**uthentication **M**odules, т. е. Подключаемые Модули Идентификации). Идея PAM -- в том, чтобы унифицировать и, вместе с тем, сделать более гибкой *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 её 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
cron              login     sshd       user-group-mod
groupadd          newusers  su         useradd
```

### Подключаемые Модули Идентификации (PAM)

В PAM определено четыре случая, требующие идентификации: `auth` -- собственно идентификация, определение, тот ли пользователь, за кого он себя выдаёт, `account` -- определение, всё ли хорошо с учётной записью пользователя, `password` -- *изменение* пароля в учётной 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). Четвёртое 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
```

### Настройка PAM для `login`

Такие настройки `login` обнаружил Мефодий на своём компьютере. Во всех четырёх случаях используется включаемый файл `system-auth` (к нему обращаются и другие службы), с некоторыми дополнениями: 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 неё `pam_console.so` выполняет описанную в лекции [Права доступа](#) "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 "теневого" файл паролей, а схема ТСВ, описанная в лекции [Права доступа](#). (Как уже известно Мефодию, в этой схеме вместо общего для всех `/etc/shadow` задействованы файлы вида `/etc/tcb/входное_имя/shadow`, причём 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 желает сообщить системе о том, что наступило событие, которое стоит запомнить, у неё 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 разберётся, что с этим сообщением делать: записать в файл, вывести на 12-ю консоль или забыть о нём. Второй путь (*централизованная журнализация*) предпочтительнее почти всегда; исключение -- 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, сообщаемые `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 важности: `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
```

### Настройка системных журналов

В первом поле строки указываются профили сообщений, разделённые 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 объёма), либо отводить поток сообщений определённой 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 и 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 `what is` 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
```

### 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, подённого, понедельного и помесечного запуска специального сценария (в

примере -- run-parts). Этот сценарий реализует упрощённую схему ".d", он попросту запускает отсортированные в лексикографическом порядке(4) сценарии из соответствующего каталога (например, из /etc/cron.daily):

```
[root@localhost root]# ls /etc/cron.daily
000anacron logrotate makewhatis osec stmpclean updatedb
```

Сценарии, запускаемые ежедневно

Вот что происходит каждый день на машине Мефодия: запуск `anacron` и "прокручивание" системных журналов (об этом речь пойдёт далее), обновление базы `whatis`, проверка цельности системы с помощью `ossec`, прореживание старых и неиспользуемых файлов в `/tmp` (утилита `stmpclean`) и, наконец, обновление базы `locatedb`.

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 ещё 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* запустить(5). 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

Ещё изучая работу `syslog`, Мефодий не расставался с мыслью, что файл, в котором записывается системный журнал, постоянно растёт. 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, сложности **атомарность** таких составных операций нелегко, а во-вторых, они требуют *удвоенного* 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 ресурсоёмкий алгоритм, позволяющий избежать переполнения `/var`: т. н. "прокручивание" системных журналов.

Суть алгоритма в следующем: когда настает пора укоротить журнал (например, раз в неделю или если файл журнала достиг определённого 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 добавлением 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
```

## 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 определённые 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 [Графический интерфейс \(X11\)](#)). 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 использовать 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/     .lproptions   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 .*~
```

## Конфигурационные файлы в домашнем каталоге

Многие утилиты *создают* конфигурационный файл при запуске, если его в домашнем каталоге пользователя нет, поэтому со временем объём `ls -A` становится всё больше. Файл `.lproptions` задаёт параметры подсистемы печати, `.pinerc` -- это настройки почтового клиента `pine`, `.viminfo` -- файл истории команд редактора `Vi`, а файл `.Xauthority` и каталог `.xsession.d` управляют запуском графической подсистемы `X`, описанной в лекции [Графический интерфейс \(X11\)](#). Из файлов в примере некоторые вообще не являются "стандартными": так, `.aliases` и `.i18n` просто "втягиваются" стартовым командным сценарием `bash`, потому что упомянуты в нём 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. Ещё 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
```

## 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 выделенным а отдельный от руководства по самой утилите документ.

---

(1) Эта команда определяет, файлы какого типа и в каком количестве содержатся в каталоге /etc.

(2) Обозначение "i18n" происходит от слова "internationalization", в котором 20 букв, т. е. "i", "n" и 18 букв между ними.

(3) Программисты имели в виду Хроноса, стихийное божество времени у древних греков. Уже сами греки часто называли так "владыку неба" титана Крона (отца богов-Кронинов и, среди прочих, Зевса, который впоследствии папу и других титанов заключил в Тартар, и стал владыкой неба сам). У древних римлян и Крон и Хронос почитались под именем Сатурна, божества неумолимого времени.

(4) То есть в таком порядке, в котором они были бы расставлены в словаре. Причём цифры предшествуют алфавитным знакам, а между собой сортируются по возрастанию, от 0 до 9. Отсюда "000anacron" -- такое имя обеспечит, чтобы этот сценарий был выполнен самым первым.

(5) Название *этого* демона пародирует cron с намёком на анахронизм, то есть несвоевременность выполнения заданий.



## 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 определёнными *функциональностями* самой системы: например, добавлял в неё 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 нужно 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 одним 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 жёсткого требования, чтобы один пакет содержал только одну программу. В пакет естественно объединять такие ресурсы, с которыми удобно работать как с одним 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
```

### Создание файлового архива при помощи `tar`

Первый параметр `tar` состоит из двух частей: операция, которую следует произвести над архивом, в данном случае "с" (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.

---

С каждым указанным каталогом `tar` работает рекурсивно, т. е. запаковывает все содержащиеся в нём файлы и подкаталоги.

---

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 тут внимание на два обстоятельства. Во-первых, в архиве имена файлов сохранились вместе с путём. Во-вторых, все пути, сохранённые в архиве -- относительные.

При распаковке архива `tar` файлы извлекаются вместе с путём, недостающие подкаталоги создаются по мере необходимости. Все пути `tar` считает относительными 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
```

### Распаковка архива

Ключ "v" велит `tar` быть "разговорчивым" (verbose), т. е. выводить больше диагностических сообщений, благодаря этому `tar` при распаковке выводит имена (с путём) всех записываемых файлов. Если в рабочем каталоге уже есть файл, который `tar` собирается

создать при распаковке, то этот файл будет попросту *заменён* файлом из архива. Так, когда Мефодий распаковал свой архив, подкаталог "bin/" со всем его содержимым *заменился* на подкаталог из архива. В данной ситуации это не страшно, 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 распространённых -- 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 определённые (хотя и не очень жёсткие 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 обеспечить, чтобы в разных пакетах не оказалось файлов с одинаковым именем и путём, чтобы файл, принадлежащий одному пакету, не мог быть заменён 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 удалённых 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
. . .
```

Какие файлы принадлежат пакету?

Мефодий получил довольно длинный список имён файлов, среди которых встретил многие из уже знакомых ему утилит и кое-какие незнакомые. Причём запрашивая список файлов, Мефодий использовал только *имя* пакета, без *версии* -- `rpm` позволяет обращаться так любому из установленных пакетов(2).

Можно выполнить и обратную процедуру -- выяснить про любой файл, какому пакету он принадлежит:

```
methody@localhost:~ $ rpm -qf /etc/passwd
setup-2.2.5-some1
```

Какому пакету принадлежит файл?

Файлы, принадлежащие пакету, могут быть не только удалены, но и изменены. Это может быть сделано сознательно (например, отредактирован конфигурационный файл), в таком случае при обновлении или удалении пакета изменённый файл нужно сохранить, потому что в нём 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 проверить соответствие 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
```

Что изменилось в пакете?

Мефодий получил список изменившихся с момента установки пакета файлов. Видимо, всё 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 придётся внимательнее читать руководство.

Система Linux раскладывается на компоненты без остатка: *каждый* файл в Linux принадлежит какому-нибудь (и только одному!) пакету(3). 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 выполнить в момент 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, причём 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
```

### 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, 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 [Этапы загрузки системы](#).

## Цена удобства

Удобство, которое получает пользователь при работе с пакетами достигается не само собой, а 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(4). Узнать, кто и когда создал пакет, получить краткую справку о программном обеспечении, которое в нём содержится, можно командой `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 <shogun@somelinux.org>
Summary    : Initial set of configuration files
Description:
Initial set of configuration files to be placed into /etc.
```

### Описание пакета

Нужно принимать во внимание, что любой пакет, содержащий программное обеспечение для Linux, не является универсальным. Если у вас есть такой пакет, это ещё 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 пойдёт в лекции [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)). 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 с ориентацией на один дистрибутив, может оказаться несовместимым с другим. Чтобы в вашем дистрибутиве появилась некоторая программа, кто-то должен подготовить и сделать доступным соответствующий пакет.

Ресурсы, необходимые для установки и интеграции в систему некоторого компонента **пакет** (архив файлов, до- и послеустановочные сценарии, информация о пакете и его сопровождающем), объединённые в одном файле.

Несмотря на частные 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. Однако пакет может содержать и исходные тексты программ, такие пакеты называются **исходными**. Доступность исходных кодов -- обязательное условие распространения большей части программного обеспечения для Linux, см. лекцию [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#). Если получилось так, что никто не подготовил пакет с нужной вам программой для вашего дистрибутива, у вас есть возможность установить исходный пакет и скомпилировать программу самостоятельно(5). При успешной компиляции из исходного пакета получается соответствующий двоичный, который уже можно установить в системе.

## Зависимости

Мефодий нашёл в Интернете пакет с заинтересовавшей его программой в подходящем формате rpm и решил попробовать его установить(6).

```
[root@localhost RPMS.local]# rpm -i xsltproc-1.0.32-some1.i586.rpm
ошибка: неудовлетворённые зависимости:
    libxslt = 1.0.32-some1 нужен для xsltproc-1.0.32-some1
[root@localhost RPMS.local]#
```

Пакет не установлен из-за неудовлетворённых 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, причём 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 нём 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.

Ситуация, при которой **пакет** не может быть установлен в систему, если в **зависимость** ней не установлен хотя бы один из некоторого множества пакетов. Аналогично, пакет не может быть удалён 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, сколько бы они не 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 её 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

Однако 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(7). Это могут быть и утилиты, которые запускаются при работе самой программы или во включённых в пакет сценариях, программа-интерпретатор для исполнения этих сценариев, и даже определённые файлы, которые должны присутствовать для правильной работы программы (например, утилита `passwd` предполагает, что существует файл `/etc/passwd`).

Зависимость может быть и безусловной. 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 определённой версии, ни больше, ни меньше. Бывают случаи, когда зависимость имеет обобщённую форму, например, почтовому клиенту (программе для чтения и написания электронной почты) может потребоваться служба доставки



электронной почты. В Linux такую услугу предоставляют несколько разных программ, и любая из них удовлетворит зависимость.

Разобравшись с понятием зависимости, Мефодий набрался твёрдой решимости установить-таки нужный ему пакет, установив всё, что он потребует. Но не тут-то было: взявшись устанавливать библиотеки, Мефодий выяснил, что каждой из них требуются какие-то ещё пакеты, отсутствующие в системе, у каждого из них тоже есть зависимости и т. п. -- один единственный пакет повлёт 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 распространённый 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 разрешён очень просто: следует удалить один из конфликтных пакетов, после чего свободно устанавливать другой.

Каждый пакет помимо имени обозначен и номером версии, указывающим степень обновлённости 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 определённый 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, Vim (`Vi` и `nvim`). Пакеты `Vi` и `nvim` не конфликтуют друг с другом, однако оба могут с равным правом быть вызваны по команде `v i`. Чтобы определить, какой именно из них запускать как `v i`, во многих дистрибутивах Linux (в частности, в том, который использует Мефодий) используется 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` может быть символьной ссылкой либо на `/usr/bin/vim`, либо на `/usr/bin/nvim` (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 её 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 системах 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 [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)), 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 метainформация 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 распространённым форматом: авторы программ для Linux обычно выкладывают свои программы в Интернет 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` смогли усмотреть много недостатков и недочетов, поэтому распространено множество улучшенных и дополненных версий `rpm`, и, соответственно, пакетов, ориентированных на какую-то из этих версий, но носящих всё то же расширение. На практике это означает, что разные версии `rpm` не полностью совместимы между собой, поэтому даже если в вашей системе используется `rpm`, это совершенно не означает, что вы сможете установить любой найденный в Интернете пакет в этом формате.

Случай `rpm` -- только самая яркая демонстрация 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 -- делать их самостоятельно.

Другая проблема установщиков пакетов 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, и 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 может 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". Однако 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, которая вычисляет весь комплекс операций над отдельными пакетами, который нужно произвести для установки/удаления нового компонента (пакета), и сама запускает **установщик пакетов** сколько нужно раз с нужными 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 удалённых репозиториях, вычисление зависимостей и установку требуемых по ним пакетов, удаление замещаемых пакетов и т. п.).

Наиболее известный и популярный менеджер пакетов называется АРТ (**A**dvanced **P**ackage **T**ool). 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 АРТ.

Чтобы установить пакет, прежде всего нужно узнать о его существовании. Пакетов для каждого дистрибутива Linux доступны тысячи и даже десятки тысяч, ориентироваться в них непросто. АРТ предоставляет возможность поиска нужного среди доступных пакетов, для этого используется утилита `apt-cache`. В каждом пакете обязательно содержится краткая аннотация (в одну строку) и небольшое описание содержащихся в пакете ресурсов (не длиннее нескольких абзацев). По команде "`apt-cache search подстрока`" АРТ найдёт и выведет список из имён 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
```

### Поиск пакетов в АРТ

Для установки и удаления пакетов предназначена утилита `apt-get`, а команда установки выглядит совсем просто: "`apt-get install имя_пакета`", причём не нужно указывать никаких сведений о версии и местонахождении пакета: АРТ сам найдёт 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.

## Установка пакета с помощью АРТ

Процедуру установки АРТ 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 АРТ нашёл 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 АРТ, есть ещё несколько менеджеров пакетов. Большинство из них специфичны для определённого дистрибутива, как, например, `emerge` для Gentoo или `yast` для SUSE. Их задачи и возможности примерно совпадают с АРТ.

## 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 неудовлетворёнными 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 её 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 ещё один -- `python-strict`:

```
[root@localhost shogun]# rpm -ql python
(не содержит файлов)
[root@localhost shogun]# rpm -ql python-strict
(не содержит файлов)
```

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 АРТ мог работать с пакетами, они должны содержаться в организованном по специальным правилам хранилище -- **репозитории**. Список доступных АРТ репозиторий 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
```

Файл `sources.list`

После каждого изменения файла `/etc/apt/sources.list` нужно обновлять кеш АРТ, 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.

АРТ позволяет и просто доставить пакет в систему, не устанавливая его. Так, например, всегда происходит с **исходными пакетами**, которые просто копируются из репозитория в определённый 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 своём **репозитории** (репозитории, отражающие наиболее современное состояние программного обеспечения, доступны через Интернет). Пользователю имеет смысл не отставать от обновлений программного обеспечения, потому что новые версии программ -- это и большая надёжность работы системы, и новые возможности.

Менеджеры пакетов позволяют делать **комплексные обновления** всей системы. В АРТ эту процедуру можно выполнить одной командой: "`apt-get dist-upgrade`". Эта процедура сначала исследует содержимое всех доступных репозиторий и находит там все пакеты более поздних версий, чем соответствующие пакеты, установленные в системе. После этого вычисляется объём 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 ручное

вмешательство пользователя. Тот род обновлений системы, который нужно делать регулярно и *обязательно* -- это обновления, связанные с безопасностью (security updates). Когда в программе обнаруживают и исправляют серьёзные ошибки, угрожающие безопасности всей системы, разработчики дистрибутивов обычно заботятся о том, чтобы соответствующие обновления достигли 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 всё-таки нет в официальном репозитории дистрибутива, обычно находятся "частные" репозитории, доступные по сети Интернет, включающие не вошедшие в официальный репозиторий пакеты.

Если всё-таки нужный вам пакет нигде не найти собранным именно для вашего дистрибутива, можно установить и сторонний пакет, но это может быть выполнено только при помощи установщика пакетов, менеджер пакетов в этой ситуации будет бесполезен. Можно установить программу и самостоятельно скомпилировав её из исходных кодов, однако здесь стоит иметь в виду следующее.

Автор *программы* совершенно не обязан учитывать в ней все тонкости 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 ожидает, что *все остальные* программы тоже находятся в этом каталоге). Это значит, что придётся самостоятельно разобраться и с тем, как и с какими параметрами компилировать программу, как устанавливать её 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 ещё кому-нибудь, программу, которой пока нет в дистрибутиве, то самое правильное -- сделать из неё *пакет*, 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 ещё раз (на другом компьютере или обновляя версию самой программы), и, главное, *всему сообществу пользователей* вашего дистрибутива!

Наконец, во многие современные дистрибутивы включаются средства, помогающие 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. Короче говоря, собрав пакет с помощью такого средства, вы можете серьёзно претендовать 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 роста и улучшения собственной операционной системы.

---

(1) Tar появился намного раньше Linux и изначально служил для создания файловых архивов на магнитной ленте. Отсюда и его название -- `tape archiver`, "архиватор для (магнитных) лент". В настоящее время `tar` присутствует в любой UNIX-подобной системе и позволяет работать с файловыми архивами на любых носителях.

(2) Что логично, поскольку в системе может быть установлена только *одна* версия данного пакета. См. подробнее раздел [Конфликты и альтернативы](#).

(3) Естественно, кроме тех файлов, которые созданы пользователями.

(4) Функции по созданию пакета в формате `rpm` также выполняет программа `rpm`.

(5) Слухи о том, что для сборки программы из исходных текстов не обязательно даже знать, что такое "компилятор", далеки от действительности.

(6) Для установки и удаления пакетов нужны права администратора -- это серьёзные изменения в системе.

(7) Имеет смысл исключать из понятия зависимости использование наиболее стандартных ресурсов, без которых немислима система Linux как таковая. К таким ресурсам можно отнести **системные вызовы** и некоторые стандартные файлы, вроде `/dev/null`.



## 14. Сеть TCP/IP в Linux

### Сетевые протоколы. Семейство протоколов TCP/IP

Так случилось, что Мефодий мало что знал о компьютерных сетях до знакомства с Linux. Если пользоваться только www-браузером и почтовой программой, сведений, вроде "у каждого компьютера Internet есть имя, на компьютерах бывает почта и WWW", бывает вполне достаточно. Строго говоря, если сеть настроена, почтовые клиенты или браузеры Linux не требуют большего объёма знаний. Однако Linux хорош именно тем, что позволяет проследить работу сети от процедур самого низкого уровня, вроде поведения сетевых карт, до приложений высокого уровня и их протоколов.

В разговоре о сетях передачи данных понятие "уровень" возникает неспроста. Дело в том, что передача данных между компьютерами -- сложный процесс, в котором решается сразу несколько разноплановых задач. Если представить себе весь процесс организации сети "на пустом месте", как если бы никаких сетевых разработок доньше не было, все эти задачи встают одна за другой, по мере решения предыдущей.

Итак, если бы Мефодий получил задание "придумать Internet" на пару с Гуревичем, какие бы вопросы перед ними встали?

1. Среда передачи данных. Посредством чего передавать данные? Как именно представляется передаваемая информация?
2. Устройство передачи данных. (Раз уж известно, как передаются данные). Как подключаться к среде? Как отличить данные от не-данных (т. е. определить *идёт ли* передача)? Как определить очерёдность работы *нескольких* устройств, подключённых к одной среде передачи данных? Как определить, *кому* предназначаются данные, передаваемые в общей среде?
3. Топология неоднородной сети. (Раз уж известно, как подключить компьютер к одной или нескольким средам передачи данных). Если в сеть объединены *несколько* сред передачи данных, как определить адресата (и отправителя тоже)? Как обеспечить пересылку данных из одной среды в другую? Как выстроить *непрерывный* маршрут пересылок от отправителя к адресату.
4. Доставка данных. (Раз уж есть механизм передачи данных от любого абонента сети к любому). Как обеспечить целостность и надёжность передачи данных (и нужно ли)? Как управлять самим каналом передачи данных (например, чтобы не отправлять данных больше, чем принимающая сторона в состоянии принять)? Как разделять *несколько* каналов передачи данных (например, когда от одного компьютера к другому одновременно передаются *два* файла)?

5. Интерпретация данных. (Раз уж возможна надёжная и без искажений доставка). Что делать с полученными данными? Какие части операционной системы отвечают за их обработку, и откуда про это знает абонент с другой стороны соединения?

Ответы на эти вопросы в формализованном виде носят название **протоколов**: в них пунктуально описывается, как именно предлагается решать ту или иную задачу, какая для этого потребна дополнительная информация, какова должна быть логика поведения передающей и принимающей стороны и т. п.

В приведённом делении на этапы (уровни) примечательна их относительная независимость: если группа задач, связанная с некоторым уровнем, решена, на следующем уровне можно забыть, как именно решались эти задачи. Так, устройство передачи данных типа "Ethernet" с точки зрения компьютера всегда одно и то же, какой бы носитель при этом не использовался: коаксиальный кабель или кабель типа "витая пара", хотя с физической и даже топологической точки зрения эти среды сильно различаются(1). Точно так же обстоят дела при переходе со второго уровня на третий: во время получения данных уже совершенно неважно, какие среды передачи были при этом задействованы (ethernet, три провода, голубиная почта(2)...). Переход с третьего уровня на четвёртый и с четвёртого на пятый тоже обладает этим свойством.

По всей видимости, именно с этими задачами сталкивались и разработчики из института ARPA (Advanced Research Projects Agency, "Агентство Перспективных Исследовательских Проектов"; в процессе работы оно было переименовано в DARPA, где "D" означало Defence). По крайней мере, предложенное ими в середине семидесятых **семейство протоколов** TCP/IP также подразделялось на пять уровней: аппаратный, интерфейсный, сетевой, транспортный и прикладной. Впоследствии аппаратный уровень стали смешивать с интерфейсным, так как с точки зрения операционной системы они неразличимы(3). Именно разделение на независимые друг от друга уровни позволило со временем объединить большинство разнородных локальных сетей в единое сетевое пространство -- глобальную сеть Internet.

В TCP/IP вопрос о том, как обеспечить нескольким абонентам сети возможность передавать данные, не мешая друг другу, решён с помощью **разделения пакетов** данных. Разделение пакетов предполагает, что данные передаются не единым блоком, а по частям, **пакетами**. Алгоритмы, определяющие, когда абоненту разрешено посылать *следующий* пакет, могут быть разными, но результат всегда один: в сети передаются попеременно фрагменты *всех* сеансов передачи данных. В сильно загруженном состоянии такая сеть может просто *не принять* очередной пакет от абонента-отправителя, и тому придётся ждать удобного случая, чтобы всё-таки "пропихнуть" его в переполненную другими пакетами среду. Таким образом, обеспечить *гарантированное* время передачи одного пакета в сетях с разделением пакетов бывает довольно сложно, хотя есть алгоритмы, позволяющие это сделать.

Противоположность метода разделения пакетов -- метод **разделения каналов**, который предполагает, что в сети имеется определённое число каналов передачи данных, которые

абоненты сети арендуют на всё время передачи. По такому принципу построены, например, телефонные линии: дозвонившись, мы арендуем канал связи между двумя телефонными аппаратами, и до тех пор, пока этот канал занят, невозможно ни воспользоваться им кому-то другому, ни организовать параллельно передачу данных откуда-нибудь ещё. Главное достоинство сетей с разделением каналов -- постоянная (за вычетом помех на линии) скорость передачи данных. Главный недостаток -- ограниченное количество каналов передачи. Проектировать среду передачи так, чтобы каждый абонент был связан с каждым отдельным каналом, имеет смысл только когда абонентов очень мало: количество каналов будет пропорционально *квадрату* количества абонентов. Количество каналов в большой сети будет существенно меньшим, и ровно столько сеансов передачи данных можно будет в этой сети установить. Попытка соединиться с абонентом, когда все каналы уже заняты, окончится неудачей. Мефодий припомнил своего знакомого, дозвониться которому тяжело, хотя телефон тот занимает нечасто и не подолгу. По всей видимости, каналов между какими-то двумя АТС, через которые Мефодий связывается с приятелем, хронически недостаёт (так бывает, когда отдалённый район быстро застраивается и наполняется телефонами).

Если вернуться к сети с разделением пакетов, то можно заметить, что на каждом уровне, под **пакетом** понимается разное. С точки зрения интерфейсного уровня пакет -- это ограниченный возможностями среды передачи данных фрагмент, в котором необходимо дополнительно указать, какое устройство из числа подключённых к среде передачи данных его отправило, и какому устройству он предназначен. С точки зрения сетевого уровня размер пакета определяется удобством его обработки, а дополнительно в нём надо указать уникальные для всей сети адреса отправителя и получателя (а также тип протокола и многое другое). С точки зрения уровня транспортного размер пакета определяется качеством связи (чем меньше пакет, тем ниже вероятность порчи, но тем больше теряется на дополнительной информации: идентификатор сеанса, тип, специальные поля, описывающие логику связи и т.п.). Наконец, если на прикладном уровне определено понятие "пакет", то размер его и содержимое определяются протоколом прикладного уровня.

Таким образом, процесс передачи данных выглядит так: порция данных прикладного уровня нарезается на части, соответствующие размеру пакета транспортного уровня (фрагментируется), к каждому фрагменту приписывается транспортная служебная информация, получаются пакеты транспортного уровня. Каждый пакет транспортного уровня может быть опять-таки фрагментирован для передачи по сети, к каждому получившемуся фрагменту добавляется служебная информация сетевого уровня, что даёт последовательность сетевых пакетов. Каждый сетевой пакет тоже может быть фрагментирован до размера, "пролезающего" через конкретное сетевое устройство, из него формируются пакеты интерфейсного уровня (фреймы). Наконец, к каждому фрейму само устройство (по крайней мере, так это сделано в ethernet) приписывает некоторый ключ, по которому принимающее устройство распознаёт *начало* фрейма. В таком виде данные передадутся по проводам. Процесс "заворачивания" пакетов более высокого уровня в пакеты более низкого уровня называется **инкапсуляцией**.

Компьютер, получивший фрейм, выполняет процедуры, обратные инкапсуляции и фрагментации: пакеты низкого уровня освобождаются от служебной информации и накапливаются до тех пор, пока не сформируется пакет более высокого уровня. Затем этот пакет отсылается на уровень выше и всё повторяется до тех пор, пока освобождённые от всей дополнительной информации и заново собранные воедино данные не попадут к пользователю (или к программе, которая их обрабатывает).

Единица передачи информации в компьютерной сети. Помимо передаваемых данных содержит служебную информацию, в частности, идентификаторы **сетевой** отправителя и адресата, контрольную сумму, поля используемого протокола. **пакет** Наибольший размер пакета определяется чаще всего не объёмом передаваемых данных, а требованиями протокола и необходимостью разделять сеть передачи данных между несколькими абонентами.

## Аппаратный и интерфейсный уровни

Итак, на аппаратном уровне возможна какая угодно среда передачи данных, с точки зрения Linux сеть начинается в месте *подключения* к этой среде, то есть на **сетевом интерфейсе**. Список сетевых интерфейсов и их настроек в системе можно посмотреть с помощью команды `ip link show`:

```
methody@localhost:~ $ ip link show
bash: ip: команда не найдена
methody@localhost:~ $ /sbin/ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 00:0C:29:56:C1:36 brd ff:ff:ff:ff:ff:ff
```

Запуск `ip link`

Утилитой `ip` пользуется, в основном, сама система или администратор. Эта утилита -- "универсальная": первый её параметр указывает группу настроек сетевой подсистемы, а второй -- действие, производимое с этими настройками. В примере утилита `ip` показывает ("show") настройки сети на интерфейсном уровне ("link"). Названия групп и команды в утилите `ip` можно сокращать до нескольких первых букв, наиболее часто употребляемые -- и до одной; команду `show` можно вовсе опускать. Так что `ip link show` и `ip l` означают одно и то же. Ошибка в начале примера вызвана тем, что в дистрибутиве, которым пользуется Мефодий, каталог `/sbin`, не входит по умолчанию в значение переменной `PATH` у обычного пользователя.

Название сетевого интерфейса состоит из его типа и порядкового номера (каким по счёту его когда-то распознано ядро). Все сетевые интерфейсы ethernet в Linux обычно называются *ethномер*, начиная с `eth0`. Параметр `mtu` (**m**aximum **t**ransfer **u**nit) определяет наибольший размер фрейма, а большинство прочих параметров описывают куда более тонкие настройки сетевого интерфейса.

**сетевой интерфейс** Точка взаимодействия утилит Linux с реализацией TCP/IP в ядре системы. Как правило, имеет уникальный сетевой адрес. Интерфейсу может соответствовать некоторое сетевое оборудование (например, карта ethernet), в этом случае определён также и интерфейсный его адрес.

Значение параметра `link/ether` -- уникальный внутри среды передачи данных идентификатор сетевого устройства (или аппаратный адрес), в данном случае -- отвечающий протоколу Ethernet. В Ethernet аппаратный адрес называется MAC-address (от **Media Access Control**, управление доступом к среде), и состоит из шести байтов, которые принято записывать в шестнадцатеричной системе счисления и разделять двоеточиями. Каждая ethernet-карта имеет собственный уникальный MAC-address (в примере -- `00:0C:29:56:C1:36`), поэтому его легко использовать для определения отправителя и получателя в рамках одной ethernet-среды. Если идентификатор получателя неизвестен, используется аппаратный **широковещательный адрес**, `FF:FF:FF:FF:FF:FF`. Сетевая карта, получив широковещательный фрейм или фрейм, MAC-адрес получателя в котором совпадает с её MAC-адресом, обязана отправить его на обработку системе.

Термин "Media Access Control" имеет отношение к алгоритму, с помощью которого решается задача очерёдности передачи. Алгоритм базируется на трёх принципах:

1. Прослушивание среды. Каждое устройство умеет определять, идёт ли в данное время передача данных по среде. Если среда свободна, значит устройство имеет право само передавать данные.
2. Обнаружение коллизий. Если решение о начале передаче данных *одновременно* приняли несколько устройств, в среде возникнет **коллизия**, и распознать, где чьи были данные, становится невозможно. Зато устройства всегда замечают произошедшую коллизию, и передают данные повторно.
3. Случайное время ожидания перед повтором. Если бы после коллизии все устройства начали одновременно повторять передачу данных, случилась бы новая коллизия. Поэтому каждое устройство выжидает некоторое случайное время, и только после этого повторяет передачу. Если повторная коллизия всё-таки возникает, устройство ждёт вдвое дольше(4), и т. д. до тех пор, пока не превысится допустимое время ожидания, после чего системе сообщается об ошибке.

Приведённый алгоритм обладает двумя недостатками. Во-первых, уже на интерфейсном уровне время передачи одного пакета может быть любым, так как неопределённое промедление с передачей предусмотрено протоколом. Во-вторых, сеть ethernet считается хорошо загруженной, если на протяжении некоторого промежутка времени в среднем треть этого времени была потрачена на передачу данных, а две трети времени среда была свободна. Сеть ethernet, нагруженная *наполовину*, работает очень медленно и с большим числом коллизий, а сеть, нагруженная на две трети, считается неработающей. Это -- плата за отсутствие синхронизации работы всех устройств в сети.

## Сетевой уровень

Создатели первых сетей, объединяющих несколько сред передачи данных, для идентификации абонента таких сетей пытались использовать те же аппаратные адреса. Это оказалось делом неблагодарным: если в ethernet аппаратный адрес уникален всегда, то в других сетях аппаратные адреса могут быть уникальны только в рамках *одной* среды (например, все устройства нумеруются, начиная с 0) или даже выдаваться динамически, да и форматы аппаратных адресов в разных средах различны. Возникла необходимость присвоить каждому сетевому интерфейсу некоторый единственный на всю глобальную сеть **адрес**, который бы не зависел от среды передачи данных и всегда имел один и тот же формат.

### Адресация

Адрес, определяемый протоколом IP (Internet Protocol), состоит из четырёх байтов, записываемых традиционно в десятичной системе счисления и разделяемых точкой. Адреса, присвоенные сетевым интерфейсам, можно просмотреть командой `ip address show` (сокращается до `ip a`). Чтобы не писать каждый раз полный путь утилиты `ip`, Мефодий просто добавил `/sbin` к содержимому переменной `PATH`:

```
methody@localhost:~ $ export PATH="$PATH:/sbin"
methody@localhost:~ $ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:0c:29:56:c1:36 brd ff:ff:ff:ff:ff:ff
    inet 192.168.102.125/24 brd 192.168.102.255 scope global eth0
```

#### Запуск `ip addr`

Адрес сетевого интерфейса `eth0` из примера -- `192.168.102.125`. Первый сетевой интерфейс из примера, `lo`, -- так называемая **заглушка** (`loopback`), которая используется для организации сетевых взаимодействий компьютера с самим собой: любой посланный в заглушку пакет немедленно обрабатывается как принятый оттуда. Заглушка обычно имеет адрес `127.0.0.1`.

Отдельная среда передачи данных (локальная сеть) также имеет собственный адрес. Если представить IP-адрес в виде линейки из 32 битов, она строго разделяется на две части: столько-то битов слева отводится под **адрес сети**, а оставшиеся -- под **адрес абонента** в этой сети. Для того, чтобы определить размер адреса сети, используется **сетевая маска** -- линейка из 32 битов, в которой на месте адреса сети стоят единицы, а на месте адреса компьютера -- нули. При наложении маски на IP-адрес все единицы в нём, которым соответствуют нули в маске, превращаются в нули(5). Таким образом вычисляется IP-адрес сети. В примере сетевая маска интерфейса `eth0` состоит из 24 единиц и 8 нулей, о чём свидетельствует окончание `/24`, приписанное к адресу сетевого интерфейса. Мефодий заметил, что если сетевая маска выравнена по границе байта, производить двоичные операции вообще не надо:

так, в примере можно было просто сказать, что адрес сети занимает первые три байта, 192.168.102.0, а адрес абонента -- оставшийся один, 0.0.0.125.

Сетевую маску нередко записывают непосредственно в числовом виде. Так, маска /24 выглядит как 255.255.255.0, /8 -- как 255.0.0.0, а допустим, /27 -- как 255.255.255.224. При этом стоит отдельно обращать внимание на то, какие нули в адресе сети -- значащие, а какие -- часть адреса абонента. Например, в адресе 10.0.0.1 при сетевой маске 255.255.0.0 адрес сети (10.0.0.0) имеет два значащих байта, из которых второй -- полностью нулевой. Запись вида 10.0.0.0/16 в данном случае более наглядна.

IP-адрес, составленный из адреса сети, за которым следуют *все единицы* (в примере -- 192.168.102.255), называется **широковещательным адресом**: любой принадлежащий сети 192.168.102.0 компьютер, получивший IP-пакет с адресом получателя 192.168.102.255, должен обработать его, как если бы в поле "получатель" стоял его собственный IP-адрес.

В более ранних дистрибутивах Linux для работы с сетевыми интерфейсами использовалась утилита `ifconfig`. Эта утилита входит в состав современных дистрибутивов и вполне работоспособна, однако `ip link` и `ip address` обладают существенно большими возможностями и потому пользоваться `ip` предпочтительнее, чем `ifconfig`.

Когда компьютер с некоторым IP-адресом решает отправить пакет другому компьютеру, он выясняет, принадлежит ли адресат той же локальной сети, что и отправитель (т. е. подключены ли они к одной среде передачи данных). Делается это так: на IP-адрес получателя накладывается сетевая маска, и таким образом вычисляется адрес сети, которой принадлежит получатель. Если этот адрес совпадает с адресом сети отправителя, значит, оба находятся в одной локальной сети. Это, в свою очередь, означает, что *аппаратный* адрес (MAC) получателя должен быть отправителю известен.

MAC-адреса компьютеров локальной сети хранятся в специальной таблице ядра, называемой "таблицей соседей" или, как это принято в протоколе IP, **ARP-таблицей/APR-таблица**. Просмотреть содержимое этой таблицы можно с помощью команды `ip neighbour show` (сокращается до `ip n`):

```
[root@localhost ~]# ip neighbour show
192.168.102.1 dev eth0 lladdr 00:50:56:C0:00:01 STALE
192.168.102.7 dev eth0 lladdr 00:50:56:C3:11:a2 REACHABLE
[root@localhost ~]# sleep 60
[root@localhost ~]# ip n
192.168.102.7 dev eth0 lladdr 00:50:56:C3:11:a2 DELAY
[root@localhost ~]# ping -c1 192.168.102.1
PING 192.168.102.1 (192.168.102.1) 56(84) bytes of data.
64 bytes from 192.168.102.1: icmp_seq=1 ttl=64 time=0.217 ms

--- 192.168.102.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.217/0.217/0.217/0.000 ms
[root@localhost ~]# ip n
192.168.102.1 dev eth0 lladdr 00:50:56:C0:00:01 REACHABLE
```

## Просмотр таблицы ARP

Более точно, ARP-таблица отражает *соответствие* между IP- и MAC-адресами. Таблица эта динамическая: устаревшие соответствия из неё удаляются, так как компьютеру может быть назначен *другой* IP-адрес, интерфейс можно отключить от сети, заменить и т. д. Если вновь понадобится связаться с компьютером, чей MAC-адрес устарел, соответствие IP и MAC придётся устанавливать по новой. В примере была использована команда `ping`, посылающая на указанный IP-адрес пакеты служебного протокола ICMP, на который адресат обязан ответить. Если ответа нет, значит, связь по каким-то причинам невозможна. Соответствие между MAC-адресом `00:50:56:C0:00:01` и IP-адресом `192.168.102.1` к моменту первого вызова `ip neighbour` уже было довольно устаревшим (пометка STALE), а через 60 секунд и вовсе просрочилось, но после обмена ICMP-пакетами свежее соответствие снова появилось в таблице.

Устанавливать соответствие между адресами сетевого и интерфейсного уровня -- дело протокола ARP (Address Resolution Protocol, "протокол преобразования адресов"). В случае преобразования IP в MAC он работает так: отправляется широковещательный ethernet-фрейм типа "ARP-запрос", внутри которого -- IP-адрес, что означает "Эй! У кого такой IP?". Каждый работающий компьютер обрабатывает этот фрейм и тот, чей IP-адрес совпадает с запрошенным, возвращает отправителю *пустой* фрейм типа "ARP-ответ", в поле "отправитель" которого указан искомый MAC-адрес. Это означает "У меня. А что?". Тут ARP-таблица заполняется и первый компьютер готов к инкапсуляции IP-пакета.

В более ранних версиях Linux для работы с ARP-таблицей используется команда `arp`. Как и в случае с `ifconfig`, эта команда никуда не делась, просто её возможности уже возможностей `ip`.

## Маршрутизация

Более сложный вопрос встаёт, если IP-адрес компьютера-адресата *не* входит в локальную сеть компьютера-отправителя. Ведь и в этом случае пакет необходимо отослать какому-то абоненту локальной сети, с тем, чтобы тот перенаправил его дальше. Этот абонент, **маршрутизатор**, подключён к нескольким сетям, и ему вменяется в обязанность пересылать пакеты между ними по определённым правилам. В самом простом случае таких сетей две: "внутренняя", к которой подключены компьютеры, и "внешняя", соединяющая маршрутизатор со всей глобальной сетью. Таблицу, управляющую маршрутизацией пакетов, можно просмотреть с помощью команды `ip route show` (сокращается до `ip r`):

```
[root@localhost ~]# ip route show
192.168.102.0/24 dev eth0 proto kernel scope link src 192.168.102.102
default via 192.168.102.1 dev eth0
```

### Простая таблица маршрутизации

На машине Мефодия в таблице маршрутизации всего две записи: одна -- про сеть `192.168.102.0/24`, доступную по интерфейсу `eth0`, вторая -- про сеть `default`,



доступную через маршрутизатор с адресом 192.168.102.1. Сеть default -- это и есть "весь интернет", потому что ей принадлежат *любые* IP-адреса: в этом случае ни одного бита на сетевую маску не отводится (получается /0). Такая запись в таблице называется **маршрут по умолчанию**. Если маршрут по умолчанию не задан, попытка связаться с удалённым компьютером может закончиться ошибкой "No route to host": система не сможет определить, кому пересылать пакет.

На **маршрутизаторе** таблица выглядит чуть сложнее:

```
[root@fuji ~]# ip r
192.168.102/24 dev eth1 proto kernel scope link src 192.168.102.1
10.13.0.0/16 dev eth0 proto kernel scope link src 10.13.102.1
default via 10.13.0.1 dev eth0
[root@fuji ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 08:00:27:50:7a:b1 brd ff:ff:ff:ff:ff:ff
   inet 10.13.102.1/16 brd 10.13.255.255 scope global eth0
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 08:00:27:7f:75:e2 brd ff:ff:ff:ff:ff:ff
   inet 192.168.102.1/24 brd 192.168.102.255 scope global eth1
```

### Сложная таблица маршрутизации

Помимо сетевого интерфейса eth1 с адресом 192.168.102.1, который "смотрит" в ту же локальную сеть, что и машина Мефодия, имеется интерфейс eth0 с адресом 10.13.102.1. Локальная сеть, подключённая к этому интерфейсу, имеет маску /16. Скорее всего, к этой сети не подключено ни 65534 компьютера, ни даже в сто раз меньше: для сети ethernet и 500 компьютеров -- слишком много. Мефодий заметил, что в IP-адресе *обоих* сетевых интерфейсов третий байт равен 102; сетевые администраторы часто так настраивают свои серверы, дабы не перегружать память избыточными данными.

По умолчанию система обрабатывает таблицу маршрутизации так. Получив IP-пакет, сначала проверяет, не предназначен ли он самому маршрутизатору. В этом случае пакет следует "поднять" на транспортный уровень. Иначе система начинает "примерять" его поочерёдно ко всем записям таблицы маршрутизации, отсортированным в порядке убывания размера сетевой маски (в том же порядке выдаёт их команда `ip route show`). Если сеть адресата совпадает с сетью из таблицы, можно принимать решение, куда пересылать пакет. Параметр `scope link` означает, что получатель находится в одной локальной сети с отправителем, и пакет пора "спускать" на интерфейсный уровень (например, выяснять MAC-адрес получателя и заниматься пересылкой ethernet-фреймов). В примере три локальных сети: на "заглушке" lo -- 127.0.0.0/8, на интерфейсе eth0 -- 10.13.0.0/16, и на интерфейсе eth1 -- 192.168.102.0/24.

Параметр `via IP-адрес` говорит о том, что в доступных локальных сетях абонента нет, зато есть маршрутизатор, отвечающий за пересылку пакетов с соответствующим адресом сети. Таблица маршрутизации просматривается заново: процесс *должен* закончиться

выяснением, куда передавать данные на интерфейсном уровне. В противном случае выдаётся сообщение об ошибке. В примере все пакеты, не предназначенные сетям 192.168.102.0/24, 10.13.0.0/16 и 127.0.0.0/8, отправляются на **маршрутизатор по умолчанию** с адресом 10.13.0.1.

Относительно IP-адресов на маршрутизаторе Гуревич как-то заметил, что ни один из них не "настоящий". Он имел в виду стандарт RFC1918, описывающий, какие диапазоны IP-адресов можно использовать в *любой* внутренней сети. Задача системного администратора -- сделать так, чтобы при работе с сетью Internet ни в одном пакете не стояло такого внутреннего адреса отправителя: например, подменять внутренние адреса на единственный внешний ("настоящий"). Задача эта решается с помощью межсетевого экрана (firewall), который в Linux называется iptables, но когда Мефодий попросил Гуревича рассказать поподробнее, тот только рукой махнул: надо хорошо знать TCP/IP.

## Служебный протокол ICMP

Есть такие протоколы уровня IP, действие которых этим уровнем и ограничиваются. Например, служебный протокол ICMP (**I**nternet **C**ontrol **M**essage **P**rotocol), предназначенный для передачи служебных сообщений. С одним примером применения ICMP Мефодий уже знаком: это утилита ping. Другое применение ICMP -- сообщать отправителю, почему его пакет невозможно доставить адресату, или передавать информацию об изменении маршрута, о возможности фрагментации и т. п. Протоколом ICMP пользуется утилита traceroute, позволяющая приблизительно определять маршрут следования пакета (ключ "-n" означает, что преобразовывать IP-адреса в доменные имена не надо):

```
[root@localhost ~]# traceroute www.ru -n
traceroute to www.ru (194.87.0.50), 30 hops max, 38 byte packets
 1 192.168.102.1  0.223 ms  0.089 ms  0.105 ms
 2 10.13.0.1  25.599 ms  21.390 ms  21.812 ms
 3 195.34.53.53  24.111 ms  21.213 ms  25.778 ms
 4 195.34.53.53  23.614 ms  33.172 ms  22.238 ms
 5 195.34.53.10  43.552 ms  48.731 ms  44.402 ms
 6 195.34.53.81  26.805 ms  21.307 ms  22.138 ms
 7 213.248.67.93  41.737 ms  41.565 ms  42.265 ms
 8 213.248.66.9  50.239 ms  47.081 ms  64.781 ms
 9 213.248.65.42  99.002 ms  81.968 ms  62.771 ms
10 213.248.78.170  62.768 ms  63.751 ms  78.959 ms
11 194.87.0.66  101.865 ms  88.289 ms  66.340 ms
12 194.87.0.50  70.881 ms  67.340 ms  63.791 ms
```

### Определения маршрута пакета

Утилита traceroute показывает список абонентов, через которых проходит пакет по пути к адресату, и потраченное на это время. Однако список этот *приблизительный*. Дело в том, что первому пакету (точнее, первым трём, так как по умолчанию traceroute шлёт пакеты по три) в специальное поле TTL (**T**ime **T**o **L**ive, время жизни) выставляется значение "1". Каждый маршрутизатор должен уменьшать это значение на 1, и если оно обнулилось, передавать отправителю ICMP-пакет о том, что время жизни закончилось, а адресат так и не

найден. Так что на первую серию пакетов отреагирует первый же маршрутизатор, и `traceroute` выдаст первую строку маршрута. Второй пакет посылается с TTL=2, и, если за две пересылки адресат не достигнут, об этом рапортует *второй* маршрутизатор. Процесс продолжается до тех пор, пока очередной пакет не "доживёт" до места назначения. Строго говоря, неизвестно, каким маршрутом шла очередная группа пакетов, потому что с тех пор, как посылалась предыдущая группа, какой-нибудь из промежуточных маршрутизаторов мог передумать и послать новые пакеты другим путём.

## Транспортный уровень

Транспортных протоколов в TCP/IP два -- это TCP (Transmission Control Protocol, протокол управления соединением) и UDP (User Datagram Protocol). UDP устроен просто. Пользовательские данные помещаются в единственный транспортный пакет-датаграмму, которой приписываются обычные для транспортного уровня данные: адреса и **порты** отправителя и получателя, после чего пакет уходит в сеть искать адресата. Проверять, был ли адресат способен этот пакет принять, дошёл ли пакет до него и не испортился ли по дороге, предоставляется следующему -- прикладному -- уровню.

Иное дело -- TCP. Этот протокол очень заботится о том, чтобы передаваемые данные дошли до адресата в целостности и сохранности. Для этого предпринимаются следующие действия:

### Устанавливается соединение

- Перед тем, как начать передавать данные, TCP проверяет, способен ли адресат их принимать. Если адресат отвечает согласием на открытие соединения, устанавливается *двусторонняя* связь между ним и отправителем. Помимо адресов отправителя и адресата и номеров **порта** на отправителе и адресате, в TCP-соединении участвуют два **номера последовательности** (SEQuential Number, SEQN), с помощью которых каждая сторона проверяет, не потерялись ли пакеты по пути, не перепутались ли.

### Обрабатываются подтверждения

- Двусторонняя связь нужна ещё и потому, что на каждый TCP-пакет с любой стороны требуется *подтверждение* того, что этот пакет принят. Упрощённо можно представить дело так, что отправитель и адресат *по очереди* обмениваются пакетами, каждый из которых содержит подтверждение только что принятого, и, возможно, полезные данные. Если происходит какая-то ошибка, она возвращается вместо подтверждения и отправитель обрабатывает её (например, посылает пакет ещё раз).

### Отслеживаются состояния абонентов

- С первым же подтверждением каждый из абонентов передаёт размер т. н. **скользящего окна** (sliding window). Этот размер показывает, сколько ещё данных готов принять адресат. Отправитель посылает сразу несколько пакетов суммарным размером с это окно, а после ждёт подтверждения об их принятии. Когда приходит подтверждение первого из пакетов в окне, окно "скользит" вперёд: теперь оно

начинается со второго пакета, и в него попадает один или несколько ещё не посланных пакетов. Если адресат может принять больше данных, он сообщает о большем размере окна, а если данные перерабатываться не успевают -- о меньшем.

Кажется, что TCP -- протокол по всем статьям удобнее UDP. Однако в случаях, когда пользовательские данные всегда помещаются в один пакет, зато самих пакетов идёт очень много, посылать всего одну датаграмму намного выгоднее, чем всякий раз устанавливать соединение, пересылать данные и закрывать соединение (что как минимум требует по три пакета в каждую сторону). Очень трудно использовать TCP для *широковещательных* передач, когда число абонентов-адресатов весьма велико или вовсе неизвестно. Посмотреть параметры всех передаваемых через сетевой интерфейс пакетов можно с помощью команды `tcpdump -ri интерфейс`, хотя Мефодию не хватило поверхностного знания TCP/IP для того, чтобы понять выдачу этой команды.

## Прикладной уровень

Как бы не был надёжен протокол TCP, он не имеет никакого понятия о том, что же, собственно, за данные с его помощью передаются. Да и не должен: принцип разделения уровней не позволяет заглядывать "внутрь" передаваемого пакета, и способов наверняка распознать используемый в нём прикладной протокол нет. Прикладной уровень, в отличие от транспортного, предусматривает *сколько угодно* протоколов передачи данных.

Интерпретация данных, в конце концов, дело уже не ядра, а какой-нибудь программы ("приложения", как правило, демона). Для того, чтобы можно было предположить, какой протокол используется при передаче данных и чтобы система могла передать эти данные соответствующей программе, ещё на транспортном уровне было введено понятие **порт**.

## Клиент-серверная модель

С точки зрения прикладного уровня, **порт** -- это идентификатор *сервиса*, предоставляемого системой. В самом деле, практически любой акт передачи данных выглядит, как если бы некий **клиент**, которому эти данные нужны, запрашивал их у **сервера**, который может их предоставить(6). Отношения между программами, которые связываются по сети друг с другом, почти всегда асимметричны: одной что-то надо, у другой это что-то есть. При установлении соединения и приложение (программа-клиент), и служба (программа-сервер) используют механизм **сокетов**, описанный в лекции [Работа с внешними устройствами](#), однако ведут себя по-разному.

Служба, запускаясь на сервере, создаёт сетевой сокет и *прикрепляет* его к определённому порту сервера с помощью системного вызова `bind()`. Затем она регистрируется в качестве **обработчика запросов** (listener), приходящих на этот порт. Служба ждёт запросов, и когда они поступают, предпринимает какие-нибудь действия, например, считывает пришедшие данные и анализирует их в соответствии со своим протоколом, отсылает какие-то данные абоненту, пославшему запрос и т. п.

Приложение, запускаясь на клиенте, также создаёт сокет и *присоединяется* с его помощью к тому же порту на *сервере*, где запущена служба, используя системный вызов

connect(). Затем оно, как и служба, посылает и получает данные. Разницы между *обменом* данными по сетевому сокету и по сокету в файловой системе нет. Очередность обмена данными определяется прикладным протоколом.

Как приложение узнаёт, к какому именно порту необходимо подключиться? За большинством прикладных протоколов закреплён постоянный номер порта. Постоянные номера портов и названия соответствующих протоколов хранятся в файле /etc/services:

```
[root@localhost ~]# wc /etc/services
 553 2794 19869 /etc/services
[root@localhost ~]# egrep "^(ftp|http|smtp|ssh).*tcp" /etc/services
ftp          21/tcp          # File Transfer [Control]
ssh          22/tcp          # SSH Remote Login Protocol
smtp         25/tcp          mail            # Simple Mail Transfer Protocol
http         80/tcp          www www-http    # World Wide Web HTTP
```

### Постоянные номера портов для некоторых протоколов

Этот файл -- не догма, а руководство к действию: каждый может организовать, допустим, сервис HTTP по 25-му порту. Только как об этом узнают другие клиенты, и что подумают почтовые программы, ожидая по этому порту встретить сервис SMTP (пересылка почты)? Вывести разнообразную информацию о сетевых соединениях можно командой netstat. В примере используются ключи -a (выводить как установленные соединения, так и службы, готовые принять соединение), -n (не преобразовывать числовую информацию в символьную), -t и -u (показывать информацию только о TCP и UDP):

```
[root@localhost ~]# netstat -antu
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp      0      0 0.0.0.0:22             0.0.0.0:*               LISTEN
tcp      0      0 192.168.102.125:22     192.168.102.1:33208    ESTABLISHED
udp      0      0 0.0.0.0:111             0.0.0.0:*
```

### Просмотр установленных соединений и служб

Здесь видно, что на компьютере зарегистрировано два TCP-обработчика (на портах 111 и 22), один UDP-обработчик по 11-му порту (понятие Listener, то есть обработчик *соединения* для UDP не имеет смысла), а также установлено одно соединение с компьютера 192.168.102.1, исходящий порт 33208, к 22-му порту (это порт службы Secure Shell, предоставляющей удалённый терминальный доступ... видимо, Гуревич работает?). В более сложных случаях, когда номер порта заранее неизвестен, а известно только название и версия сервиса, используется служба portmap, которая раздаёт незанятые порты службам и сообщает приложениям, к какому из них надо обратиться. Порт 111 соответствует именно этой службе.

## Обслуживание прикладного уровня в Linux

Самый простой способ проверить, предоставляет ли некий сервер услуги по некоему TCP-порту -- это подключиться к нему. Если под рукой нет приложения, работающего по

требуемому протоколу, не беда: подойдёт утилита `telnet`. В качестве первого параметра следует указать адрес компьютера, к которому нужно подключиться, а в качестве второго (необязательного) -- номер порта. Когда-то эта утилита использовалась в качестве клиента к терминальной службе, однако от неё пришлось отказаться: пароль пользователя передавался по сети нешифрованным. Но в качестве клиента других служб, многие из которых используют текстовые протоколы, `telnet` используется и поныне. Если даже протокол и не текстовый -- не беда: можно выйти в командный режим `telnet`, нажав "`^[`", и подать команду `close`, которая закроет соединение:

```
[root@localhost ~]# telnet 192.168.102.1 112
Trying 192.168.102.1...
telnet: connect to address 192.168.102.1: Connection refused
[root@localhost ~]# telnet 192.168.102.1 111
Trying 192.168.102.1...
Connected to 192.168.102.1.
Escape character is '^['.
^[
telnet> close
Connection closed.
```

## Использование `telnet`

В сценариях, а также в случае взаимодействия по **нетекстовым** протоколам вместо интерактивной утилиты `telnet` стоит использовать `netcat`, которая работает как `cat` в указанный сокет или из него.

Как уже говорилось, интерпретацией прикладных протоколов занимаются разнообразные программы. Прикладной протокол можно представить как обмен сообщениями, часто текстовыми, между клиентом и сервером. Было бы естественно оформлять такие программы в виде **фильтров**, чтобы пользоваться простейшими функциями ввода-вывода. Однако механизм сокетов предусматривает асинхронную передачу данных, для чего используются *другие* функции. Программа, желающая обслуживать сетевые соединения по определённому порту, должна удовлетворять четырём требованиям:

1. Быть **демоном**, то есть постоянно находиться в памяти;
2. Создавать сокет, прикреплять его к порту;
3. Регистрироваться как обработчик по этому сокету и принимать соединения (возможно, придётся обрабатывать несколько соединений одновременно);
4. Анализировать прикладной протокол и действовать по результатам анализа.

Нетрудно заметить, что первые три свойства -- общие для большинства сервисов. В Linux есть метадемон `inetd`, который берёт на себя всю общую сетевую часть работы, а программам предоставляет разбираться в прикладном протоколе. Сделать свой сетевой сервис с помощью `inetd` становится очень просто: пользователь программирует **фильтр**, задача которого -- обмениваться командами прикладного протокола с помощью стандартного ввода и стандартного вывода. Этот фильтр регистрируется в настройках `inetd` с указанием порта, с которого будут приниматься запросы. После чего сам `inetd` становится обработчиком запросов по *всем* указанным портам, сам открывает соединение, запуская

соответствующий фильтр, а данные из сокета пересылает туда и обратно по двум **каналам**. При этом фильтр-обработчик даже не должен быть демоном: это обычная программа, которая завершается, когда это предусмотрено прикладным протоколом, или когда закрывается входной поток.

Мефодий минут за пять написал службу, которая в ответ на подключение передаёт календарь на текущий месяц. В его системе используется модернизированная версия `inetd` -- `xinetd`, обученная чтению конфигурационных файлов по схеме ". d":

```
[root@localhost ~]# grep quake /etc/services
quake          26000/tcp
quake          26000/udp
[root@localhost ~]# cat /etc/xinetd.d/calendar
service quake
{
    socket_type      = stream
    protocol        = tcp
    wait            = no
    user            = nobody
    server          = /usr/bin/cal
    disable         = no
}
```

#### Настройка `cal` в качестве сетевой службы

Вместо номера порта можно использовать название протокола из `/etc/services`. Мефодий воспользовался портом `26000` (чем мог создать некоторые трудности поклонникам одной компьютерной игры). Осталось только перезагрузить `xinetd`, чтобы он нашёл новый конфигурационный файл, и подключиться к `26000` порту:

```
[root@localhost ~]# service xinetd restart
Stopping xinetd service: [ DONE ]
Starting xinetd service: [ DONE ]
[root@localhost ~]# telnet localhost quake
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
    December 2004
Su Mo Tu We Th Fr Sa
      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
```

Подключение к самодельной службе "календарь"

## Служба доменных имён

В предыдущих примерах Мефодий использовал ключ "-n" многих сетевых утилит, чтобы избежать путаницы между IP-адресами и **доменными именами** компьютеров. С другой стороны, доменные имена -- несколько слов (часто осмысленных)--запоминать гораздо удобнее, чем адреса (четыре каких-то числа).

Когда-то имена *всех* компьютеров в сети, соответствующие IP-адресам, хранились в файле `/etc/hosts`. Пока абоненты Internet были наперечёт, поддерживать правильность его содержимого не составляло труда. Как только сеть начала расширяться, неувязок стало больше. Трудность была не только в том, что содержимое `hosts` быстро менялось, но и в том, что за соответствие имён адресам в различных сетях отвечали разные люди и разные организации. появилась необходимость структурировать глобальную сеть не только топологически (с помощью IP и сетевых масок), но и *административно*, с указанием, за какие группы адресов кто отвечает.

Проще всего было структурировать сами имена компьютеров. Вся сеть была поделена на **домены** -- зоны ответственности отдельных государств ("ru", "de", "us", "uk", "it" и т. п.) или независимые зоны ответственности ("com", "org", "net", "edu", "info" и т. п.). Для каждого из таких **доменов первого уровня** должно присутствовать подразделение, выдающее всем желающим абонентам имена, заканчивающиеся на ". *ДОМЕН*". Подразделение обязано организовать и поддерживать *службу*, заменяющую файл `hosts`: любой желающий имеет право узнать, какой IP-адрес соответствует имени компьютера в этом домене или какому доменному имени соответствует определённый IP-адрес.

Такая служба называется DNS (**D**omain **N**ame **S**ervice, служба доменных имён). Она имеет иерархическую структуру. Если за какую-то группу абонентов домена отвечают не хозяева домена, а кто-то другой, ему выделяется **поддомен** (или домен *второго уровня*), и он сам распоряжается именами вида "*имя\_компьютера. поддомен. домен*". Например, за компьютеры в домене ". ru", принадлежащие корпорации "Dried Bugs" отвечают сотрудники соответствующего подразделения этой корпорации. Корпорация владеет поддоменом `dbugs.ru`. В домене `ru` не обязаны знать, какие именно адреса есть в поддомене, но обязаны предоставить информацию о том, как обратиться к серверу имён поддомена `dbugs.ru`. Сами сотрудники "Dried Bugs" тоже отвечают только за несколько собственных компьютеров, а всю ответственность за компьютеры в подразделениях перекалывают на сетевых администраторов этих подразделений, выделив им поддомены *третьего* уровня `pr.dbugs.ru`, `cook.dbugs.ru` и `warehouse.dbugs.ru`. Таким образом получается нечто вроде распределённой сетевой базы данных, хранящей короткие записи о соответствии доменных имён IP-адресам.

**доменное  
имя**

Имя абонента Internet, имеющее текстовый формат и используемое вместо IP-адреса. Состоит из собственного имени абонента в домене и имени домена, определяющего административную принадлежность абонента. В отличие от IP-адреса, доменное имя не задаётся самим абонентом сети, а устанавливается службой доменных имён.

Все программы, работающие с доменными именами, оказываются клиентами какого-нибудь сервера доменных имён (DNS-сервера). Для этого они обращаются к функциям из библиотеки `libresolv` (или им подобным), а те уже определяют, как превратить доменное имя в адрес. Функции используют файл `/etc/host.conf`, описывающий, какими



способами они должны выполнять преобразование доменных имён в адреса и обратно, а также формат выдачи данных в различных случаях. Обычно сначала проверяется файл `/etc/hosts`, а если там соответствий не найдено -- `/etc/resolv.conf`. В `resolv.conf` указан домен по умолчанию (он приписывается в конец имени, если другим способом имя никак не удаётся преобразовать в адрес) и один-два адреса DNS-серверов, к которым и обращаются функции. Такими клиентами выступили утилиты `ping` и `traceroute` в предыдущих примерах, преобразуя имя `www.ru` в адрес `194.87.0.50`. Если утилите `traceroute` не указывать ключ `"-n"`, она подаст несколько DNS-запросов, по одному на каждый маршрутизатор, на *обратное* преобразование -- из IP-адреса в доменное имя.

```
[root@localhost ~]# cat /etc/host.conf
order hosts,bind
multi on
[root@localhost ~]# cat /etc/resolv.conf
domain nipponman.ru
nameserver 192.168.102.1
[root@localhost ~]# traceroute -q1 www.ru
traceroute to www.ru (194.87.0.50), 30 hops max, 38 byte packets
 1 fuji.nipponman.ru (192.168.102.1)  1.378 ms
 2 gateway.nipponman.ru (10.13.0.1)  41.155 ms
 3 195.34.53.53 (195.34.53.53)  48.503 ms
 4 195.34.53.53 (195.34.53.53)  24.033 ms
 5 M9-cr01-A197-cr01.core.mtu.ru (195.34.53.10)  33.414 ms
 6 M9-gw2-M9-cr01.core.mtu.ru (195.34.53.81)  26.259 ms
 7 s-b3-pos0-0.telia.net (213.248.67.93)  59.791 ms
 8 s-bb1-pos5-0-0.telia.net (213.248.66.1)  67.011 ms
 9 mow-b1-pos1-0.telia.net (213.248.101.10)  76.138 ms
10 demos-101566-mow-okt-i1.c.telia.net (213.248.78.170)  78.591 ms
11 m9-3-GE4-0-0-vl10.Demos.net (194.87.0.66)  69.813 ms
12 www.ru (194.87.0.50)  70.583 ms
```

### Работа DNS-клиента, встроенного в `traceroute`

Как видно из примера, обратное преобразование в современной сети работает не всегда. Отсутствие обратной зоны не поощряется сообществом, но и не считается преступлением. Мефодий заметил, что компьютер, не имеющий обратного преобразования адреса, вообще какой-то странный: один раз он передал пакет *сам себе* (здесь Мефодий не совсем прав: на самом деле этот маршрутизатор отчего-то уменьшает TTL пакета на 2, поэтому-то и на третьем, и на четвёртом шаге именно он возвращает ICMP-сообщение). Кстати сказать, именно по причине того, что DNS-запрос невелик, зато даже один абонент сети может выдать их множество, основным транспортным протоколом для DNS выбран UDP, а не TCP (это не касается протоколов обмена целыми зонами между DNS-серверами, там, конечно, господствует TCP).

Если вся задача пользователя -- это послать DNS-запрос, то лучше воспользоваться утилитой `host`, специально для этого предназначенной:

```
methody@localhost:~ $ host www.ru
www.ru has address 194.87.0.50
methody@localhost:~ $ host 194.87.0.51
```

```
51.0.87.194.in-addr.arpa domain name pointer www.demos-internet.ru.
methody@localhost:~ $ host -t ns www.ru
www.ru name server ns.demos.su.
www.ru name server ns1.demos.net.
methody@localhost:~ $ host -t mx www.ru
www.ru mail is handled by 5 hq.demos.ru.
```

## Утилита host

Довольно необычен формат, в котором хранятся таблицы обратного преобразования адресов. Оказывается, IP-адрес представлен в такой таблице как *имя* в домене *in-addr.arpa*, причём это имя совпадает с адресом, записанным задом наперёд. Такой формат идёт от иерархической структуры DNS. Если некоторая организация получает во владение сеть, допустим, *194.0.0.0/8*, она должна обслуживать DNS-запросы к домену *194.in-addr.arpa*. Если при этом сеть *194.87.0.0/16* передана другой организации, ей же передаётся обязанность обслуживать DNS-запросы к *поддомену* этого домена -- *87.194.in-addr.arpa*, и так вплоть до собственно IP-адресов. Вместо *host* можно использовать утилиту *dig*, которая выводит больше информации о том, как проходил сам запрос.

Помимо записей типа "адрес" (A, прямое преобразование) и "указатель ни имя" (PTR, обратное преобразование) в системе DNS может храниться и другая информация. Таблица (зона) некоторого домена должна содержать адреса доменных серверов всех его поддоменов (записи типа NS). Кроме того в домене должно быть определено имя **почтового пересыльщика** (запись типа MX). Если почтовый пересыльщик домена не указан, электронная почта направляется почтовому пересыльщику родительского домена.

В зоне DNS есть даже запись типа "просто текст", TXT: при желании можно самому определить интерпретацию полей этой записи и организовать поверх DNS предназначенную для каких угодно целей базу данных по IP-адресам или доменным именам(7). В отличие от *dig*, которой для запроса к обратной зоне требуется ключ "-x", утилита *host* различает запросы к прямой и обратной зонам по тому, задан ли в качестве параметра адрес или доменное имя. Для указания конкретного типа искомой записи обеим утилитам требуется этот тип передать явно. Тип *any* означает поиск всех записей с указанным именем:

```
methody@localhost:~ $ dig www.us any
```

```
; <<>> DiG 9.2.4rc5 <<>> www.us any
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6451
;; flags: qr rd ra; QUERY: 1, ANSWER: 10, AUTHORITY: 0, ADDITIONAL: 4

;; QUESTION SECTION:
;www.us.                IN      ANY

;; ANSWER SECTION:
www.us.                 1766   IN      A       209.173.57.26
www.us.                 1766   IN      A       209.173.53.26
www.us.                 1767   IN      NS      pine.neustar.com.
```

```
www.us.          1767    IN      NS      willow.neustar.com.
www.us.          1767    IN      NS      cypress.neustar.com.
www.us.          1767    IN      NS      oak.neustar.com.
www.us.          1771    IN      MX      20 pine.neustar.com.
www.us.          1771    IN      MX      5 oak.neustar.com.
www.us.          1771    IN      MX      5 willow.neustar.com.
www.us.          1771    IN      MX      10 cypress.neustar.com.

;; ADDITIONAL SECTION:
pine.neustar.com. 135024  IN      A       209.173.57.70
willow.neustar.com. 135024  IN      A       209.173.53.84
cypress.neustar.com. 135024  IN      A       209.173.57.84
oak.neustar.com. 135024  IN      A       209.173.53.70

;; Query time: 932 msec
;; SERVER: 192.168.102.1#53(192.168.102.1)
;; WHEN: Wed Dec 22 22:01:24 2004
;; MSG SIZE rcvd: 281
```

## Утилита dig

---

(1) Ethernet с коаксиальным кабелем имеет топологию "общая шина": все абоненты подключаются к единому кабелю, "врезая" в него Т-образный отводок; ethernet с витой парой имеет топологию "звезда": от каждого абонента идёт *собственный* кабель к центральному устройству-концентратору.

(2) Организация ТСП/Р с помощью почтовых голубей описана в RFC1149.

(3) Поэтому, если в книге написано, что ТРС/Р имеет *четыре* уровня, это тоже будет правдой -- с учётом двойственности самого нижнего.

(4) Время ожидания не *удваивается* а выбирается, опять-таки случайно, но из другого временного диапазона.

(5) Применяется побитовая операция "И".

(6) Обратная ситуация, когда клиент хочет *передать* что-то серверу, сути дела не меняет: сервер предоставляет услугу клиенту, на этот раз -- по приёму данных.

(7) Так поступают, например, при создании "чёрных списков" абонентов сети, от которых почтовый сервер не принимает писем.

# 15. Сетевые и серверные возможности

## Настройка сети

Итак, с работой сети в Linux Мефодий немного познакомился, однако то, как эту сеть *использовать* для личных нужд, понятнее не стало. Прежде всего: как приучить имеющийся компьютер пользоваться имеющейся локальной сетью?

### Настройка вручную

Первая мысль -- настроить сетевые интерфейсы вручную. Это довольно просто, если знать полагающиеся при настройке данные: IP-адрес самого компьютера, IP-адрес маршрутизатора по умолчанию и адрес сервера доменных имён.

Задать IP-адреса интерфейсам `eth0` и `lo` можно уже известной командой `ip`:

```
[root@sakura ~]# ip address show
1: lo: <LOOPBACK> mtu 16436 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 08:00:27:44:a3:3a brd ff:ff:ff:ff:ff:ff
[root@sakura ~]# ip address add 127.0.0.1/8 dev lo
[root@sakura ~]# ip address show dev lo
1: lo: <LOOPBACK> mtu 16436 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
[root@sakura ~]# ip link set lo up
[root@sakura ~]# ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 08:00:27:44:a3:3a brd ff:ff:ff:ff:ff:ff
```

### Настройка интерфейса-заглушки

Сетевой интерфейс необходимо "поднять" (активизировать) командой

`ip link set интерфейс up` и присвоить ему хотя бы один IP-адрес командой `ip address add адрес/маска dev интерфейс`.

```
[root@sakura ~]# ip address a 192.168.102.125/24 broadcast 192.168.102.255 dev eth0
[root@sakura ~]# ip link set eth0 up
[root@sakura ~]# ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 08:00:27:44:a3:3a brd ff:ff:ff:ff:ff:ff
   inet 192.168.102.125/24 brd 192.168.102.255 scope global eth0
[user@sakura ~]$ ping -c1 192.168.102.1
PING 192.168.102.1 (192.168.102.1) 56(84) bytes of data.
64 bytes from 192.168.102.1: icmp_seq=1 ttl=64 time=3.82 ms

--- 192.168.102.1 ping statistics ---
```

```
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.821/3.821/3.821/0.000 ms
```

## Настройка сетевого интерфейса

Здесь по совету Гуревича Мефодий указал вдобавок широковещательный адрес сети.

Теперь пакеты доходят до любого абонента локальной сети, но не дальше, поскольку не задан ни один маршрутизатор. Добавить маршрутизатор можно командой `ip route add`:

```
[root@sakura ~]# ip route show
192.168.102.0/24 dev eth0 proto kernel scope link src 192.168.102.125
[root@sakura ~]# ping -c1 80.68.240.144
connect: Network is unreachable
[root@sakura ~]# ip route add default via 192.168.102.1
[root@sakura ~]# ping -c1 80.68.240.144
PING 80.68.240.144 (80.68.240.144) 56(84) bytes of data.
64 bytes from 80.68.240.144: icmp_seq=1 ttl=53 time=7.36 ms

--- 80.68.240.144 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.369/7.369/7.369/0.000 ms
[root@sakura ~]# ip r
192.168.102.0/24 dev eth0 proto kernel scope link src 192.168.102.125
default via 192.168.102.1 dev eth0
```

## Добавление маршрутизатора по умолчанию

Мефодий заметил, что запись в таблице маршрутизации относительно локальной сети `192.168.102.0/24` уже была до выполнения команды `ip route add` -- она появилась там в результате настройки сетевого интерфейса `eth0`. Не хватало только явного указания маршрутизатора. Параметры команды `ip route add` соответствуют формату таблицы маршрутизации, описанному в предыдущей лекции. Под конец Мефодий даже вспомнил о краткой форме команды: "route" сокращается до "r", а "show" можно вообще не писать.

Тем не менее служба доменных имён пока не работает: необходимо заполнить файл `/etc/resolv.conf`:

```
[root@sakura ~]# ping www.ru
ping: unknown host www.ru
[root@sakura ~]# cat /etc/resolv.conf
[root@sakura ~]# cat > /etc/resolv.conf
domain nipponman.ru
nameserver 192.168.102.1
[root@sakura ~]# ping www.ru
PING www.ru (194.87.0.50) 56(84) bytes of data.
64 bytes from www.ru (194.87.0.50): icmp_seq=1 ttl=55 time=84.3 ms
. . .
[root@sakura ~]# update_chrooted conf
```

## Определение домена и DNS-сервера

Последнюю команду присоветовал Гуревич. Дело в том, что подсистему, работающую с DNS, нередко "сажают в песочницу", то есть переносят в отдельный каталог, в котором

выполняется `chroot`. Как сказано в лекции [Конфигурационные файлы](#), в такой "песочнице" должны быть все нужные для работы файлы -- и уж конечно процедурам DNS нужен *свой* файл-копия `/etc/resolv.conf`. Информация о том, кому что нужно копировать при изменении профиля системы, обычно хранится централизованно и управляется несложными сценариями. В данном дистрибутиве команда `update_chrooted conf` как раз и копирует все изменившиеся конфигурационные файлы по песочницам.

## Настройка при установке или загрузке системы

Мефодий очень обрадовался заработавшей сети и немедленно принялся сочинять простейший стартовый сценарий, который выполнял бы все нужные команды автоматически. Выяснилось, что такой сценарий уже есть в любом дистрибутиве Linux, хотя называться он может по-разному, как правило, `/etc/init.d/network` или `networking`. Как и полагается стартовому сценарию, с параметром `start` он настраивает сеть, а с параметром `stop` -- "выключает" сетевые настройки.

Безусловно, ни список сетевых интерфейсов, ни параметры их настройки не указаны в самом стартовом сценарии, как то хотел сделать Мефодий. Всевозможные сетевые настройки хранятся в `/etc` отдельно, как правило, в специальном подкаталоге. В разных дистрибутивах Linux применяются различные схемы размещения настроек. Система, установленная на компьютере Мефодия, использует подсистему сетевой настройки `etcnet`, которая устанавливается каталог `/etc/net`.

```
[root@sakura ~]# cat /etc/sysconfig/network
When set to no, this may cause most daemons' initscripts skip starting.
NETWORKING=yes

# Used by hotplug/pcmcia/ifplugd scripts to detect current network config
# subsystem.
CONFMETHOD=etcnet

# Used by rc.sysinit to setup system hostname at boot.
HOSTNAME=sakura.nipponman.ru
DOMAINNAME=nipponman.ru

# This is used by ALTlinux ppp-common to decide if we want to install
# nameserver lines into /etc/resolv.conf or not.
RESOLV_MODS=yes

[root@sakura ~]# ls -F /etc/net
ifaces/  options.d/  scripts/  sysctl.conf
[root@sakura ~]# ls -F /etc/net/ifaces/
default/  eth0/  lo/  unknown/
```

### Каталоги и файлы с сетевыми настройками

"Головной" файл `network` с сетевыми настройками (которые могут понадобиться не только сетевой подсистеме) лежит в подкаталоге `/etc/sysconfig/`. Это обычный сценарий на `sh`, в котором некоторым переменным присваиваются значения. Другие сценарии (в частности, `start-stop`ные из `/etc/init.d`) "втягивают" этот файл командой

". " и используют значения переменных. Здесь указано, среди прочего, доменное имя компьютера и тип подсистемы сетевой настройки. Все остальные файлы -- вспомогательные сценарии, настройки параметров ядра, настройки по умолчанию для всех интерфейсов и для каждого в отдельности и т. п. можно найти в `/etc/net`.

В частности, настройки единственного на компьютере Мефодия сетевого интерфейса `eth0` находятся в каталоге `/etc/net/ifaces/eth0`:

```
[root@sakura ~]# ls /etc/net/ifaces/eth0
  ipv4address  ipv4route  options
[root@sakura ~]# cat /etc/net/ifaces/eth0/ipv4address
192.168.102.125/24
[root@sakura ~]# cat /etc/net/ifaces/eth0/ipv4route
default via 192.168.102.1
[root@sakura ~]# cat /etc/net/ifaces/eth0/options
DISABLED=no
BOOTPROTO=static
```

### Настройка сети по умолчанию

Файлы `ipv4address` и `ipv4route` содержат параметры команд `ip address add` и `ip route add` соответственно, а файл `options` -- это снова сценарий на shell.

Эти настройки используются подсистемой `etcnet`, которая запускается на этом компьютере в процессе системной загрузки под именем службы `network`. Таким образом, стоит только подать команду `service network stop`, как все сетевые интерфейсы "пропадут" (деактивируются и потеряют IP-адреса), а после `service network start` -- снова появятся.

Как правило, пользователю вообще не обязательно редактировать эти файлы. С каждым дистрибутивом поставляется программа-конфигуратор, которая позволяет "настроить сеть", не вспоминая, какие данные, в каком формате и куда нужно записывать. Обычно такая программа оформляется в стиле **мастера**, "кудесника", задающего только вопросы по существу, с её помощью и формируются более или менее подходящие конфигурационные файлы. Результатов работы мастера в большинстве случаев бывает достаточно, а в тех случаях, когда его искусственный интеллект пасует, администратор применяет свой естественный интеллект и редактор Vim. С другой стороны, изменить несколько значений в трёх конфигурационных файлах не так уж сложно. Когда настройщик действительно необходим -- это во время *установки* системы на компьютер или непосредственно после неё. Настраивать приходится сразу всё в системе, так что любая экономия времени при этом существенна.

В некоторых дистрибутивах используется подсистема `ifupdown`, основанная на уже известной схеме ". d":

```
debian!shogun$ ls -F /etc/network
if-down.d/      if-pre-up.d/   ifstate.hotplug  interfaces
if-post-down.d/ ifstate        if-up.d/         options
```

### Настройка сети с применением схемы ". d"

В `ifupdown` настройка сетевых интерфейсов и маршрутизатора по умолчанию хранится одном файле (считается, что редактировать его автоматически -- просто). Тонкая настройка сети -- в файле `options`. Каталоги `if-pre-up.d`, `if-up.d`, `if-down.d` и `if-post-down.d` предназначены для служб, которые хотят производить какие-то действия, соответственно, перед тем, как сетевой интерфейс будет активизирован ("поднят"), после успешной активизации интерфейса, перед тем как сетевой интерфейс будет деактивизирован ("опущен") и после этого.

## Автоматическая настройка

Программа-настройщик регулярно предлагала Мефодию "настроить сеть автоматически". В режиме автоматической настройки практически не запрашивается данных у пользователя. Это значит, что данная система должна брать откуда-то ещё, видимо, со специального сервера в локальной сети.

Запрашивать сетевые настройки с сервера вместо того, чтобы хранить их на каждом компьютере, довольно удобно. В самом деле: один сервер, один администратор, один файл с общими настройками. Более того, можно вообще не хранить персональных настроек для каждого компьютера в сети, а ограничиться настройками *групповыми*, лишь бы IP-адреса внутри группы различались. Одним из первых был разработан протокол RARP (reverse ARP), который, как следует из названия, занимается преобразованием, обратным ARP: по интерфейсному адресу компьютер узнаёт у сервера сетевой. В ethernet-сетях для этого посылается широковещательный ethernet-фрейм типа "RARP-запрос", который означает "Вот мой MAC-адрес. Кто-нибудь, дайте мне IP!". На что специальная программа-сервер отвечает RARP-ответом "Вот тебе IP!" -- фреймом, содержащим IP-адрес, который сервер нашёл в своей таблице. Если в сети нет ни одного RARP-сервера или ни в одном из них не зарегистрирован интерфейсный адрес компьютера-клиента, тот останется без IP. Похожую схему использует и протокол BOOTP, применяющийся для сетевой загрузки компьютеров. Предполагается, что, получив IP-адрес, клиент заберёт с сервера (по протоколу TFTP, trivial FTP) некий файл, загрузит его в память и передаст управление. Поэтому в BOOTP передаётся не только IP-адрес клиента, но и IP-адреса TFTP-сервера и маршрутизатора по умолчанию и имя файла-загрузчика.

В современных сетях чаще всего используется протокол DHCP (Dynamic Host Configure Protocol, протокол динамической настройки сетевых абонентов). Он имеет очень широкие возможности: с сервера можно получить IP-адрес, сетевую маску и широковещательный адрес, имя домена, адреса маршрутизатора и серверов доменных имён, а также великое множество других параметров, вплоть до не предусмотренных в DHCP явно, так что их тип задаётся обычным числом, а интерпретация значения целиком определяется клиентом. Урезанную часть DHCP поддерживают "умные" сетевые устройства (те, что снабжены BootROM, т. е. ПЗУ с загрузочной программой). Но полностью обрабатывать все поля DHCP умеет специальный демон-клиент. В Linux этот демон называется `dhcpcd` (DHCP client



daemon). В его ведение, как минимум, входит настройка сетевого интерфейса, маршрута по умолчанию и `resolv.conf`.

Так что всё, что Мефодий делал вручную или вписывал в настроечный файл, можно получить "за просто так", если в сети работает DHCP-сервер:

```
[root@sakura ~]# ifdown eth0
[root@sakura ~]# ip address show eth0
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:44:a3:3a brd ff:ff:ff:ff:ff:ff
[root@sakura ~]# dhcpcd -h sakura eth0
[root@sakura ~]# ip address show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 08:00:27:44:a3:3a brd ff:ff:ff:ff:ff:ff
    inet 192.168.102.125/24 brd 192.168.102.255 scope global eth0
[root@sakura ~]# ps -ef | grep dhcpcd
root      9408      1   0 17:22 ?          00:00:00 dhcpcd -h sakura eth0
[root@sakura ~]# cat /etc/resolv.conf
nameserver 192.168.102.1
search nipponman.ru
```

## Использование dhcpcd

Протокол DHCP даже позволяет передавать серверу *желаемое* имя и адрес компьютера. Впрочем, выдача IP-адреса привязана, как правило, к MAC-адресу. Здесь есть особая хитрость. DHCP может неплохо работать, когда IP-адресов не хватает на всех: компьютеров с разными MAC-адресами в сети больше, чем выделенных IP, но эти компьютеры никогда не включаются все одновременно. Компьютер, определяемый в DHCP по MAC-адресу, не "присваивает" выданный IP навсегда: адрес сдаётся в "аренду" (lease) на некоторое время. Если до истечения срока аренды бывший владелец не подтвердил желание пользоваться адресом и дальше (не послал повторный DHCP-запрос), адрес считается незанятым. Но когда компьютер подключается к сети после долгого перерыва, сервер DHCP сначала просматривает "арендную историю" на предмет того, какой IP этому абоненту уже выдавался. Если этот IP не занят, то будет выдан именно он. И только когда к сети подключится совсем новый абонент, а все адреса уже когда-нибудь кому-то выдавались, среди них будет выбран и отдан в аренду новичку тот, что дольше всех оставался невостребованным.

Наконец, чтобы избавиться и от этой ручной работы, можно перенастроить `/etc/net/ifaces/eth0/` на использование DHCP:

```
[root@sakura ~]# ls /etc/net/ifaces/eth0/
options
[root@sakura ~]# cat /etc/net/ifaces/eth0/options
DISABLED=no
BOOTPROTO=dhcp
```

## Настройка интерфейса на DHCP по умолчанию

## Настройка соединений "точка--точка"

Если компьютер стоит дома, далеко не всегда есть возможность подключиться к локальной сети, непосредственно граничащей с Internet. Для передачи небольших сообщений чаще всего используется временное подключение посредством телефонной линии. На обеих сторонах линии устанавливается **модем** -- устройство, преобразующее один формат сигнала в другой. На российских телефонных линиях обычно используются аналоговые модемы, способные обмениваться данными по довольно низкокачественным линиям с большой долей помех и относительно неискажённой передачей сигнала только в диапазоне слышимых звуковых частот. За низкое качество канала приходится расплачиваться низкой скоростью передачи данных: на таких модемах она до сих пор не превышает (после отбрасывания служебной информации, ошибок и прочего) трёх-четырёх килобайтов в секунду, а в действительности бывает раза в два меньше.

Соединение между *двумя* устройствами, вообще говоря, *не* сетевыми, а только способными передавать данные, описывается несколькими протоколами. Самый распространённый из них -- PPP (**P**oint-to-**P**oint **P**rotocol, протокол "точка--точка") -- решает задачи, возникающие в силу особенностей соединений "точка--точка".

Во-первых, само участвующее в соединении устройство почти никогда не представлено в виде сетевого интерфейса, потому что не обладает нужными для организации сети свойствами. Это значит, что какая-то часть системы (скорее всего, демон) будет разговаривать с устройством на понятном ему языке, а с пользовательскими утилитами взаимодействовать посредством специально организованного виртуального сетевого интерфейса.

Во-вторых, нет необходимости поддерживать часть *интерфейсного* протокола: абонента на среде передачи данных два, никакой идентификации не требуется, потому что каждый может отличить себя от не-себя. Так, виртуальный сетевой интерфейс `ppp0`, соответствующий устройству, обмениваемому данными по протоколу PPP, не обладает MAC-адресом.

В-третьих, оттого, что соединение не постоянное, а среда за время, пока абоненты не были связаны, могла измениться до неузнаваемости, обеим сторонам приходится при каждом дозвоне идентифицировать себя. Обычно идентифицируется только сторона, которой предоставляется доступ в сеть, но проверять, до правильного ли места мы дозвонились, тоже не мешает. При установлении соединения "точка--точка" процедуры идентификации проходят после того, как появляется возможность передавать данные, но *до* всякой сетевой настройки. Мало того, данные по настройке сети (аналогичные тем, что используются в DHCP) также передаются на этом этапе взаимодействия по протоколу PPP.

В силу того, что дозвон нужен пользователям самого разного уровня знаний, для PPP написано множество программ-"звонилки", использующих графическую подсистему, со звуками и прочими не относящимися к делу украшениями. Пример такой программы -- `krppr`, утилита модемного доступа для рабочего стола KDE. Всё, что требуется от пользователя -- это указать модем, список телефонов, куда звонить и тип идентификации. Впрочем, и здесь пользователь не освобождается от "тяжёлой" мыслительной работы:

некоторые провайдеры (организации, предоставляющие выход в Internet) после дозвола требуют идентификацию не по протоколу PPP, а открытым текстом, на манер "login--password" в Linux. Иногда это и есть самый настоящий login: пользователю предоставляется терминальный доступ, а дальше пускай делает, что хочет. В этом случае приходится писать **сценарий-диалог** (chat script) в стиле "Дождаться строки "login:" -- ввести входное имя. Дождаться строки "Password:" -- ввести пароль. Дождаться подсказки командного интерпретатора -- ввести "pppd" с параметрами".

Что совсем уже просто для пользователя, так это утилита `wvdial`, описанная в лекции [Конфигурационные файлы](#). Она и модем сама определяет, и тип идентификации, и `pppd` настраивает и запускает тоже сама. В действительности же сетью занимается демон `pppd`, чьи конфигурационные файлы находятся в каталоге `/etc/ppp`:

```
[root@sakura ~]# ls /etc/ppp
callback-client  chap-secrets  ip-up         options.dialin  peers
callback-server  ip-down       ip-up.d       options.dialout
callback-users   ip-down.d     options       pap-secrets
[root@sakura ~]# ls -l /etc/ppp/*secrets
-rw-----  1 root    root    78 Jun 23  2008 /etc/ppp/chap-secrets
-rw-----  1 root    root    77 Jun 23  2008 /etc/ppp/pap-secrets
```

#### Каталог с настройками PPP

Большинство из этих файлов по умолчанию не используются. Следует помнить, что идентификационная информация, используемая в PPP (в зависимости от особенностей соединения и пристрастий провайдера могут применяться протоколы идентификации PAP или CHAP), хранится в файлах `pap-secrets` и `chap-secrets` в текстовом виде. Хранить не пароль, а ключ, как это сделано в `/etc/shadow` нельзя, так что остаётся либо надеяться на права доступа к этим файлам, либо запускать `pppd` вручную (или с помощью тех же `wvdial` и `kppp`) и каждый раз этот пароль вводить.

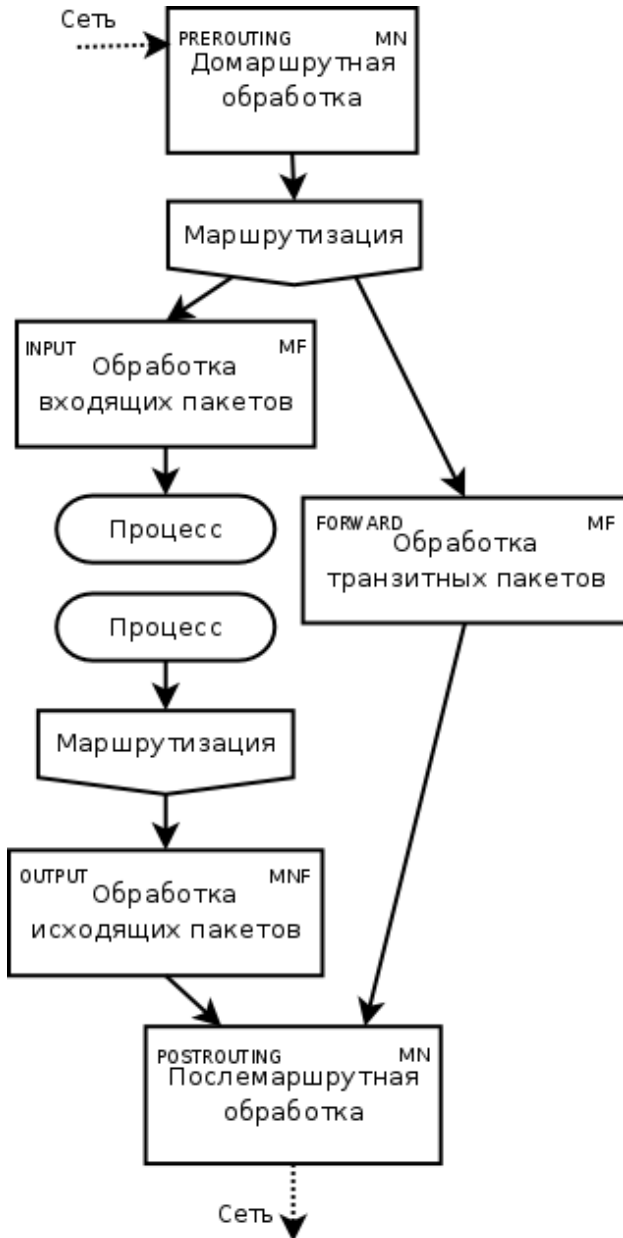
Установить PPP-соединение можно поверх *любой* среды передачи данных, в том числе и поверх локальной сети. В этом случае `pppd` обменивается данными (посредством псевдотерминальной пары `pty -- tty` или `ptmx -- pts/`, описанной в лекции [Работа с внешними устройствами](#)) с демоном `pppoe`, выполняющим роль "модема". При этом передаваемые данные можно сжимать или шифровать, а главное, связь "точка--точка" устанавливается с конкретным пользователем, который ввёл одному ему известный пароль, поэтому часто выход в Internet, требующий *строгого* учёта трафика, делают именно с помощью пары `pppd/pppoe`. Наконец, в сети может быть настоящий модем, преобразующий эти данные в формат, пригодный для передачи по *цифровым* телефонным линиям (DSL). Кстати сказать, такой модем легко собирается из маленького компьютера с низким энергопотреблением, ядра Linux и доработанного **стартового виртуального диска**. Некоторые современные DSL-модемы устроены именно так, причём ядро и `initrd` записываются в перепрограммируемое ПЗУ.

## Межсетевой экран

В Linux существует мощный механизм анализа сетевых и транспортных пакетов, позволяющий избавляться от нежелательной сетевой активности, манипулировать потоками данных и даже преобразовывать служебную информацию в них. Обычно такие средства носят название "firewall" ("fire wall" -- противопожарная стена, брандмауэр), общепринятый русский термин -- **межсетевой экран**. В более старых версиях межсетевого экрана Linux использовался вариант межсетевого экрана `ipchains`, который впоследствии был заменён на более мощный, `iptables`.

Суть `iptables` в следующем. Обработка сетевого пакета системой представляется в виде *конвейера*. Пакет нужно получить из сетевого интерфейса или от системного процесса, затем следует выяснить предполагаемый маршрут этого пакета, затем отослать через сетевой интерфейс либо отдать какому-нибудь процессу, если пакет предназначался нашему компьютеру. Налицо три конвейера обработки пакетов: "получить -- маршрутизировать -- отослать" (действие маршрутизатора), "получить -- маршрутизировать -- отдать" (действие при получении пакета процессом) и "взять -- маршрутизировать -- отослать" (действие при отсылке пакета процессом).

Между каждыми из этих действий системы помещается модуль межсетевого экрана, именуемый **цепочкой**. Цепочка обрабатывает пакет, исследуя, изменяя и даже, возможно, уничтожая его. Если пакет уцелел, она передаёт его дальше по конвейеру. В этой стройной схеме есть два исключения. Во-первых, ядро Linux даёт доступ к *исходящему* пакету только после принятия решения о его маршрутизации, поэтому связка "взять -- маршрутизировать" остаётся необработанной, а цепочка, обрабатывающая исходящие пакеты (она называется OUTPUT) вставляется *после* маршрутизации. Во-вторых, ограничения на "чужие" пакеты, исходящие не от нас и не нам предназначенные, существенно отличаются от ограничений на пакеты "свои", поэтому после маршрутизации транзитные пакеты обрабатываются ещё одной цепочкой (она называется FORWARD). Цепочка, обслуживающая связку "получить -- маршрутизировать", называется PREROUTING, цепочка, обслуживающая связку "маршрутизировать -- отдать" -- INPUT, а цепочка, стоящая непосредственно перед отсылкой пакета -- POSTROUTING:



### Обработка пакетов цепочками iptables

В варианте `ipchains` каждая цепочка представляла собой список **правил**. В правиле задаются свойства пакета и действие, которое нужно выполнять со всеми пакетами, обладающими указанными свойствами. Когда пакет попадает в таблицу, к нему начинают последовательно применяться правила. Если пакет не имеет свойств, требуемых первым правилом, к нему применяется второе, если второе также не подходит -- третье, и так вплоть до последнего, правила по умолчанию, которое применяется к любому пакету. Если свойства пакета удовлетворяют правилу, над ним совершается действие. Действие `DROP` уничтожает пакет, а действие `ACCEPT` немедленно выпускает его из таблицы, после чего пакет движется дальше по конвейеру. Некоторые действия, например `LOG`, никак не влияют на судьбу пакета,

после их выполнения он остаётся в таблице: к нему применяется следующее правило, и т. д. до ACCEPT или DROP.

Из сказанного выше следует, что действия ACCEPT или DROP в каждой таблице могут примениться к пакету лишь однократно. Для большей гибкости цепочки iptables состоят из *нескольких* (двух-трёх) таблиц. Выходя живым из одной таблицы, пакет попадает в следующую, а уж последняя передаёт его, если завершается действием ACCEPT, на конвейер. Хотя таблицы функционально одинаковы, принято использовать их по разному назначению. Таблицу mangle используют для внесения исправлений в служебную информацию пакета, таблицу filter -- для определения того, не стоит ли пакет уничтожить, а таблицу nat -- для подмены сетевых адресов. В приведённой выше диаграмме буквами M, N и F отмечено, какие именно таблицы есть в цепочках и в каком порядке их проходит пакет.

Для просмотра правил во всех таблицах всех цепочек iptables можно воспользоваться командой iptables-save:

```
[root@sakura ~]# iptables-save
# Generated by iptables-save v1.2.11 on Fri Dec 24 21:06:12 2007
*nat
:PREROUTING ACCEPT [1:261]
:POSTROUTING ACCEPT [3:220]
:OUTPUT ACCEPT [3:220]
COMMIT
# Completed on Fri Dec 24 21:06:12 2007
# Generated by iptables-save v1.2.11 on Fri Dec 24 21:06:12 2007
*filter
:INPUT ACCEPT [7:1077]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [5:355]
COMMIT
# Completed on Fri Dec 24 21:06:12 2007
# Generated by iptables-save v1.2.11 on Fri Dec 24 21:06:12 2007
*mangle
:PREROUTING ACCEPT [7:1077]
:INPUT ACCEPT [7:1077]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [5:355]
:POSTROUTING ACCEPT [5:355]
COMMIT
# Completed on Fri Dec 24 21:06:12 2007
```

## Пустые цепочки iptables

Команда группирует одинаковые таблицы каждой цепочки. В пустой таблице присутствует только правило по умолчанию (policy), в этом примере все умолчания равны ACCEPT (пропускать пакет).

## Фильтрация

Мефодий озаботился судьбой "календарного сервера", изобретённого им на прошлой лекции. Как уже было замечено, он запустил этот сервис на порту 26000, используемом популярной сетевой компьютерной игрой, что может помешать игрокам. Поэтому Мефодий решил запретить обращение к 26000-му порту отовсюду, кроме самой машины, то есть со

всех интерфейсов, кроме `lo`. С этой задачей справлялась настройка `only_from = 127.0.0.1` метадемона, но её пришлось выключить, так как она автоматически запрещала доступ из сети ко всем сервисам компьютера, обслуживаемым `inetd`:

```
[root@sakura ~]# iptables --append INPUT --in-interface lo --protocol tcp
--destination-port quake --jump ACCEPT
[root@sakura ~]# iptables --append INPUT --protocol tcp --destination-port quake
--jump REJECT
[root@sakura ~]# iptables-save
.
.
.
*filter
:INPUT ACCEPT [1030:72984]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [730:69581]
-A INPUT -i lo -p tcp -m tcp --dport 26000 -j ACCEPT
-A INPUT -p tcp -m tcp --dport 26000 -j REJECT --reject-with icmp-port-
unreachable
COMMIT
.
.
.
[root@sakura ~]# service iptables save
saving current rules to /etc/sysconfig/iptables: [ DONE ]
```

### Фильтрация TCP-запросов из сети

Команды `iptables` получаются довольно длинными: нужно аккуратно описать свойства каждого фильтруемого пакета. В примере Мефодий добавил два правила в таблицу `filter` цепочки `INPUT` (таблица `filter` выбирается по умолчанию, если не оказан ключ `-t таблица`). Первое правило разрешает принимать TCP-пакеты, адресуемые на порт 26000, если они пришли из интерфейса `lo`. Проверка таких пакетов не дойдёт до второго правила: по действию `ACCEPT` они немедленно отправятся дальше по конвейеру. Второе правило не просто уничтожает пакет, пришедший на 26000 порт, но и, как то предписывает действие `REJECT`, посылает отправителю ICMP-пакет, сообщающий о том, что такие запросы не обслуживаются (при обычном действии `DROP` клиент некоторое время дожидается подтверждения). Из примера видно, что полнословные ключи `iptables` можно заменять на однобуквенные. Кроме того, пример показывает, что стартовый сценарий `iptables` (в этом дистрибутиве) обрабатывает параметр `save`, который делает изменения, внесённые вручную, постоянными. Сценарий `service iptables start`, выполняемый при загрузке системы (если он включён с помощью `chkconfig iptables on`), использует `/etc/sysconfig/iptables`.

### Подмена адресов

Если в некоторой сети используются адреса из описанного стандартом RFC1918 внутреннего диапазона (например, из сети `10.0.0.0/8`), без дополнительных действий абоненты этой сети доступа к Internet иметь не будут. Пакеты с такими адресами запрещено передавать в Internet, а даже если они туда просочатся через маршрутизатор, соединяющий "внутреннюю" и "внешнюю" сети, следующий же маршрутизатор откажется их пересылать. Простая, казалось бы, мысль научить межсетевой экран, установленный на маршрутизаторе,

подменять IP-адреса в пакетах, приходящих из внутренней сети, своим *внешним* IP-адресом наталкивается на серьёзное препятствие.

Допустим, абоненты с адресом 10.0.0.3 и 10.0.0.7 устанавливают TCP-соединение с адресом в Internet (скажем, 209.173.53.26). Специально обученный маршрутизатор подменяет 10.0.0.3 и 10.0.0.7 на адрес своего сетевого интерфейса, подключённого к внешней сети (допустим, 194.87.0.50). Пакеты уходят адресату, как если бы и тот, и другой были отправлены самим маршрутизатором. 209.173.53.26 отвечает на два запроса двумя пакетами, оба -- на адрес 194.87.0.50. Что делать дальше? Как отличить пакет, предназначенный для 10.0.0.3 от такового же для 10.0.0.7?

На помощь приходит знание TCP! Как известно, TCP-соединение идентифицируется шестью параметрами: IP-адресами отправителя и получателя, портами на отправителе и получателе и **номера́ми последовательности (SEQN)** входящего и исходящего потока данных. В нашей схеме межсетевой экран обязательно заменяет IP-адреса одним, поэтому у двух принятых пакетов они совпадают. А вот с четырьмя оставшимися параметрами он волен поступать, как заблагорассудится: в любом случае каждому из сеансов должны соответствовать *разные SEQN* и *разные* номера исходящих портов. Осталось только держать в памяти таблицу соответствия TCP-соединений из внутренней сети TCP-соединениям во внешнюю сеть.

Этот механизм носит название "преобразование сетевых адресов" (Network Address Translation, NAT). Следует помнить, что чем больше транспортных соединений отслеживается межсетевым экраном, тем больше требуется оперативной памяти ядру Linux и тем медленнее работает процедура сопоставления проходящих пакетов таблице. Впрочем, мощности современных компьютеров позволяют без каких-либо затруднений обслуживать преобразование адресов для сети с пропускной способностью 100Мбит/с и даже выше. В `iptables` есть специальный модуль для NAT и соответствующие действия в правиле. Такие правила принято помещать в таблицы типа `nat`:

```
[root@fuji root]# ip route show
192.168.102/24 dev eth1 proto kernel scope link src 192.168.102.1
10.13.0.0/16 dev eth0 proto kernel scope link src 10.13.102.1
default via 10.13.0.1 dev eth0
[root@fuji ~]# iptables-save
# Generated by iptables-save v1.2.11 on Sat Dec 25 14:02:44 2007
*nat
:PREROUTING ACCEPT [216:12356]
:POSTROUTING ACCEPT [242:27148]
:OUTPUT ACCEPT [1428:91596]
-A POSTROUTING -o eth0 -j MASQUERADE
COMMIT
. . .
```

## Использование простейшего преобразования адресов

На том самом маршрутизаторе, где, по словам Гуревича, "ни одного настоящего IP-адреса", как раз и настроено преобразование адресов. Делается это всего одним правилом, добавленным в таблицу `nat` цепочки `POSTROUTING`. Действие `MASQUERADE` отличается от



другого подобного действия -- SNAT (Source NAT) -- только тем, что может быть использовано на интерфейсах с *изменяемым* IP-адресом, таких как ppp или настраиваемый по DHCP eth. Если в процессе работы IP-адрес интерфейса поменяется, правило SNAT продолжит подменять адреса на старый, заданный во время настройки iptables. Зато MASQUERADE вынуждено спрашивать у системы IP-адрес указанного интерфейса для *каждого* преобразуемого пакета, что может быть неудобно на очень медленных или очень загруженных разнообразными службами компьютерах.

Преобразование адресов работает не только для TCP-соединений, но и для многих других протоколов, где возможно отследить либо настоящего адресата на внутренней сети, либо идентификатор сеанса связи. Например, ICMP-пакет команды ping содержит уникальный идентификатор, который используется в *ответе* на него, поэтому не составляет труда отследить, на чей именно ping пришёл ICMP-ответ. Таблица отслеживаемых соединений отражается в файле net/ip\_conntrack виртуальной файловой системы /proc:

```
[root@fuji ~]# cat /proc/net/ip_conntrack
. . .
icmp      1 30 src=192.168.102.125 dst=209.173.53.26 type=8 code=0 id=50179
[UNREPLIED] src=209.173.53.26 dst=10.13.102.1 type=0 code=0 id=50179 use=1
tcp       6 431981 ESTABLISHED src=192.168.102.125 dst=194.87.0.50 sport=1027
dport=80 src=194.87.0.50 dst=10.13.102.1 sport=80 dport=1027 [ASSURED] use=1
```

### Просмотр таблицы подменяемых адресов

Так и есть: во-первых, Мефодий (скорее всего) что-то рассматривает на сайте www.ru (он же 194.87.0.50), а во-вторых, он зачем-то запустил ping www.us: команда cat подана как раз между ICMP-запросом по адресу 209.173.53.26 и ответом на него. Левая часть таблицы описывает соединение до подмены адресов, а правая -- после. Точно так же можно "ловить" и UDP-запросы к службе доменных имён, и многое другое. Кстати, адрес 10.13.102.1 -- тоже "внутренний", так что где-то(1) устроено как минимум *ещё одно* преобразование адресов, ибо в "большой Интернет" допускаются только "настоящие" адреса.

Конечно, фильтрацией и маскарардом функции iptables не исчерпываются. Можно, например, задать *статическую* таблицу подмены адресов, тогда появится возможность *принимать* из Internet соединения, предназначенные для внутренних серверов. Того же можно добиться, используя подмену адреса только для соединений по определённым портам и т. д.

## Сетевые службы

Эта часть лекции -- обзорная. Поскольку, с одной стороны, толковое и последовательное объяснение устройства многочисленных сетевых служб Linux требует, по крайней мере, отдельного курса лекций, а с другой стороны -- хорошей теоретической и практической подготовки слушателя. Так что придётся ограничиться поверхностным описанием наиболее востребованных для личного или домашнего пользования сервисов.

Стоит заранее отметить, что описываемые задачи, как правило, могут решаться *несколькими* путями с помощью различных демонов или утилит, по-разному выполняющих одну и ту же работу. У администратора Linux всегда есть свобода выбора!

## HTTP

Мефодий, конечно, знает, что Internet, как глобальная сеть *компьютеров* служит хранилищем глобальной сети *документов* под общим названием WWW (**World Wide Web**, "Всемирная Паутина"). Связь между документами в Паутине обеспечивается за счёт особого -- гипертекстового -- формата этих документов. Большинство из них написаны на специальном языке гипертекстовой разметки, HTML (**HyperText Markup Language**) или его диалектах и расширениях. Гипертекст может содержать **ссылки** на любые другие документы в Паутине. Формат такой ссылки описывается стандартом URL (**Universal Resource Locator**, всеобщий указатель ресурсов). Всемирность сети HTML-документов образовалась за счёт удобства доступа к ним: огромное число абонентов Internet предоставляют возможность просмотра этих документов по специальному протоколу HTTP (**HyperText Transfer Protocol**), и ещё большее число (фактически, каждый компьютер) запускают клиентские программы-навигаторы (или "броузеры", от англ. "browse", "просматривать"), позволяющие легко "переходить по ссылке", т. е. начинать просмотр документа, на который в выбранном месте ссылался исходный. Сами документы при этом принято называть WWW-страницами, или просто страницами.

Apache -- HTTP-сервер, обладающий самым большим набором возможностей. Учитывая организованный в нём механизм **подключаемых модулей** (plug-ins), создавать которые может любой грамотный программист, описать умения Apache полностью, видимо, невозможно. Документация по одним только стандартным его возможностям занимает более 50 тысяч строк. Конфигурационные файлы Apache хранятся в /etc/httpd/conf (или, в зависимости от дистрибутива, /etc/apache). Главный конфигурационный файл -- httpd.conf. Этот файл неплохо самодокументирован: вместе с комментариями в нём много больше тысячи строк, что позволяет не изучать руководство, если администратор запомнил синтаксис той или иной настройки, но, конечно, не позволяет вовсе не изучать руководства.

```
DirectoryIndex index.html index.htm index.shtml index.cgi
```

```
AccessFileName .htaccess
```

```
DocumentRoot "/var/www/html"
```

```
<Directory "/var/www/html">
    Options Indexes Includes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

```
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
<Directory "/var/www/cgi-bin">
    AllowOverride None
```

```
Options ExecCGI
Order deny,allow
Deny from all
Allow from 127.0.0.1 localhost
</Directory>
```

### Отрывок конфигурационного файла apache

Пользователь, набравший в браузере "`^http://доменное_имя_сервера`", увидит содержимое каталога, указанного настройкой `DocumentRoot` (в примере -- `/var/www/html`). Каждый каталог, содержащий WWW-страницы, должен быть описан группой настроек `<Directory>`, включающей права доступа к страницам этого каталога, настройки особенностей просмотра этих страниц и т. п. В частности, настройка `DirectoryIndex` описывает, какие файлы в каталоге считаются индексными -- если такой файл есть в каталоге, он будет показан вместо содержимого этого каталога. Если в настройке каталога `Options` не указано значение `Indexes`, просмотр каталога будет вообще невозможен. Значение `Includes` этой настройки позволяет WWW-страницам включать в себя текст из других файлов, `FollowSymLinks` позволяет серверу работать с каталогами и файлами, на которые указывают символьные ссылки (это может вывести точку доступа за пределы `DocumentRoot`!), а `MultiViews` позволяет серверу для разных запросов на одну и ту же страницу выдавать содержимое *различных* файлов -- сообразно языку, указанному в запросе.

Если настройку каталога `AllowOverride` установить в `All`, в любом его подкаталоге можно создать дополнительный конфигурационный файл, изменяющий общие свойства каталога. Имя этого файла задаёт настройка `AccessFileName` (в примере -- `".htaccess"`). Доступом к страницам в каталоге управляют настройки `Order`, `Deny` и `Allow`. Так, доступ к каталогу `/var/www/html` разрешён отовсюду, а к каталогу `/var/www/cgi-bin` -- только с самого сервера.

Важное свойство WWW-серверов -- поддержка т. н. динамических WWW-страниц. Динамическая WWW-страница не хранится на диске в том виде, в котором её получает пользователь. Она создаётся -- возможно, на основании какого-то шаблона -- непосредственно после запроса со стороны браузера. Никаких особенных расширений протокола HTTP при этом можно не вводить, просто сервер получает запрос на WWW-страницу, которой не соответствует ни один файл. Зато HTTP-адрес (точнее говоря, URL) этой страницы распознаётся сервером как динамический и передаётся на обработку выделенной для этого программе. Программа генерирует текст в формате HTML, который и передаётся пользователю в качестве запрошенной страницы. В примере для этого используется каталог `/var/www/cgi-bin`, группа настроек которого имеет единственный параметр `Options -- ExecCGI`. Для того, чтобы этот каталог, в действительности не входящий в `/var/www/html`, был доступен браузеру как подкаталог `/cgi-bin` всего дерева, необходимо прикрепить его к `DocumentRoot` с помощью `ScriptAlias`; эта

настройка говорит также и о том, что файлы в этом каталоге -- сценарии, и их надо запускать, а не показывать(2).

Как и многие другие прикладные протоколы, HTTP был и остаётся текстовым. При желании можно выучить команды HTTP и получать странички с серверов с помощью `telnet` вручную. Это означает, что любая система **идентификации**, основанная на одном только HTTP, будет небезопасна: если передавать учётные данные по HTTP непосредственно, любой абонент сети на протяжении всего маршрута пакета от броузера к серверу сможет подглядеть внутрь этого пакета и увидеть там пароль. Более или менее безопасно можно чувствовать себя, только зашифровав весь канал передачи данных. Для этого неплохо подходит алгоритм шифрования с открытым ключом, описанный в разделе [Терминальный доступ](#). Универсальный способ шифрования большинства текстовых прикладных протоколов называется SSL, **Secure Socket Layer** (уровень надёжных сокетов). Идея этого способа -- в том, что шифрованием данных занимается не само приложение, а специальная библиотека работы с **сокетами**, то есть шифрование происходит на стыке прикладного и транспортного уровня. Конечно, и клиент, и сервер должны включать в себя поддержку SSL, поэтому для служб, защищённых таким способом, обычно отводятся *другие* номера портов (например, 80 -- для HTTP, и 443 -- для HTTPS). В Apache этим занимается специальный модуль -- `mod_ssl`, его настройки и способ организации защищённой службы можно найти в документации.

Несмотря на то, что Apache решает практически любые задачи, связанные с организацией WWW-страниц, есть, конечно, и области, где его применение нежелательно или невозможно. Если, например, задача WWW-сервера -- отдавать десятков-другой статически оформленных страниц небольшому числу клиентов, запускать для этого Apache -- как стрелять из пушки по воробьям. Лучше воспользоваться сервером `thttpd`, специально для таких задач предназначенным: он займёт намного меньше ресурсов системы. Особая ситуация возникает, когда самый важный параметр сервера -- его быстродействие. Например сервер, раздающий т. н. "баннеры" (`banners`, небольшие картинки рекламного характера), приносит тем больше дохода, чем быстрее работает, что означает: может обслуживать больше клиентов. Способ радикально уменьшить время отклика такого сервера на HTTP-запрос -- оформить его не в виде **демона**, а в виде **модуля ядра**. Такой сервер существует в виде дополнений к ядру Linux под общим названием `tux`.

## FTP

В Linux существует несколько вариантов службы, предоставляющей доступ к файлам по протоколу FTP (**F**ile **T**ransfer **P**rotocol). Как правило, они отличаются друг от друга сложностью настроек, ориентированных на разные категории абонентов, подключающихся к серверу. Если выбор предоставляемых в открытый доступ данных велик, будет велик и наплыв желающих эти данные получить ("скачать"), так что возникает естественное желание этот наплыв ограничить. При этом, допустим, компьютеры из локальной сети могут неограниченно пользоваться файловыми ресурсами сервера, соединений из того же города или в пределах страны должно быть не более двух десятков одновременно, а соединений из-

за границы -- не более пяти. Такие ухищрения бывают нужны нечасто, но и они поддерживаются большинством FTP-демонов, вроде `vsftpd`, `proftpd`, `pure-ftpd` или `wu-ftpd`.

FTP -- по-своему очень удобный протокол: он разделяет поток *команд* и поток собственно *данных*. Дело в том, что команды FTP обычно очень короткие, и серверу выгоднее обрабатывать их как можно быстрее, чтобы подолгу не держать ради них (часто -- ради одной команды) открытое TCP-соединение. А вот файлы, передаваемые с помощью FTP обычно большие, поэтому задержка при передаче в несколько долей секунды, и даже в пару секунд, не так существенна, зато играет роль пропускная способность канала. Было бы удобно, если бы для передачи команд использовался "быстрый, но тонкий" канал (т. е. среда передачи данных с малым временем отклика и небольшой пропускной способностью, например, наземное оптоволокно), а для передачи данных -- "медленный, но толстый" (например, спутниковая магистраль).

Для того, чтобы эти каналы было проще различить, *данные* в FTP пересылаются по инициативе *сервера*, что означает, что именно сервер подключается к клиенту, а не наоборот. Делается это так: клиент подключается к 20-му порту сервера (*управляющий* порт FTP) и передаёт ему команду: "Хочу такой-то *файл*. Буду ждать твоего ответа на таком-то (временно выделенном) *порту*". Сервер подключается со своего 21-го порта (порт *данных* FTP) к *порту* на клиенте, указанном в команде, и пересылает содержимое запрошенного *файла*, после чего связь по данным разрывается и клиент перестаёт обрабатывать подключения к *порту*.

Трудности начинаются, когда FTP-клиент находится за **межсетевым экраном**. Далеко не каждый администратор согласится открывать доступ из *любого* места Internet к *любому* абоненту внутренней сети по *любому* порту (пускай даже и с порта 21)! Да это и не всегда просто, учитывая подмену адресов. Для того, чтобы FTP работал через межсетевой экран, придумали протокол Passive FTP. В нём оба сеанса связи -- и по командам, и по данным -- устанавливает клиент. При этом происходит такой диалог. Клиент: "Хочу такой-то *файл*. Но пассивно". Сервер: "А. Тогда забирай его у меня с такого-то *порта*". Клиент подключается к *порту* сервера и получает оттуда содержимое *файла*. Если со стороны сервера тоже находится бдительный администратор, он может не разрешить подключаться *любому* абоненту Internet к *любому* порту *сервера*. Тогда не будет работать как раз Passive FTP. Если и со стороны клиента, и со стороны сервера есть по бдительному системному администратору, никакой вариант FTP не поможет.

Ещё один недостаток протокола FTP: в силу двухканальной природы его трудно "затолкать" в SSL-соединение. Поэтому идентификация пользователя, если таковой имеется, в большинстве случаев идёт открытым текстом. Мефодий тут же вспомнил про своего приятеля, который сначала завёл себе где-то в Сети бесплатную WWW-страницу, файлы на которую надо было передавать по FTP, а потом отказался от этой затеи: с утомительным постоянством кто-то под завязку набивал эту страничку сомнительного вида архивами и программами. Несмотря на то, что для FTP подходит другой механизм шифрования,

называемый TLS, далеко не все FTP-клиенты его поддерживают, и он оказывается не востребован на серверах. Поэтому рекомендуется использовать службу FTP только для организации архивов *публичного* доступа.

## Терминальный доступ

Текстовый интерфейс позволяет пользователю Linux работать на компьютере *удалённо* с помощью терминального клиента. Весьма удобно, находясь далеко от компьютера, управлять им самым естественным способом, с помощью командной строки. Препятствий этому немного: объём передаваемых по сети данных крайне невелик, ко времени отклика в полсекунды вполне можно привыкнуть, а если оно меньше десятой доли секунды, то задержка и вовсе не мешает. Что необходимо соблюдать строго, так это шифрование учётных записей при подключении к удалённому компьютеру, а на самом деле, и самого сеанса терминальной связи, так как в нём вполне *может* "засветиться" пароль: например, пользователь заходит на удалённый компьютер и выполняет команду `passwd`.

Как уже говорилось в этих лекциях, сначала для терминального доступа использовался протокол TELNET и соответствующая пара клиент-сервер `telnet - telnetd`. От них пришлось отказаться в силу их вопиющей небезопасности. На смену TELNET пришла служба Secure Shell ("Надёжная Оболочка"), также состоящая из пары клиент-сервер -- `ssh` и `sshd`. Шифрование данных в Secure Shell организовано по принципу "асимметричных ключей", который позволяет более гибко шифровать или идентифицировать данные, чем более распространённый и привычный принцип "симметричных ключей". Симметричный ключ -- это пароль, которым данные можно зашифровать, и им же потом расшифровать.

Упрощённо метод "асимметричных ключей" можно описать так. Используется алгоритм, при котором специальным образом выбранный пароль для шифрования данных *не совпадает* с соответствующим ему паролем для их **расшифровки**. Более того, зная любой один из этих паролей, никак нельзя предугадать другой. Некто, желая, чтобы *передаваемые* ему данные нельзя было подсмотреть, раздаёт всем желающим *шифрующий* пароль со словами: "будете писать мне -- шифруйте *этим*", т. е. делает этот пароль **открытым**. *Дешифрующий* пароль он хранит в строгой тайне, никому не открывая. В результате зашифровать данные его паролем может любой, а расшифровать (что и значит -- подглядеть) -- может только он. Цель достигнута.

Тот же принцип используется для создания т. н. **электронной подписи**, помогающей идентифицировать данные, то есть определить их автора. На этот раз открытым делается *дешифрующий* ключ (конечно, не тот, о котором только что шла речь, а ещё один). Тогда любой, получив письмо и расшифровав его, может быть уверен, что это письмо написал *этот* автор -- потому что *шифрующим* ключом не владеет больше никто.

Здесь уместно сделать два замечания. Во-первых, алгоритмы шифрования с асимметричными ключами весьма ресурсоёмки, поэтому в Secure Shell (и упомянутом ранее SSL) они используются только на начальном этапе установления соединения. Обмен данными шифруется симметричным ключом, но, поскольку сам этот ключ был защищён асимметричным, соединение считается надёжным. Во-вторых этот кажущийся надёжным

алгоритм имеет серьёзный изъян, с которым, впрочем, нетрудно справиться. Опасность таится в самом начале: как, получив от *товарища открытый ключ*, удостовериться, что этот ключ *действительно* ему принадлежит? А вдруг на пути от *товарища* ключ был перехвачен *злоумышленником*, и до нас дошёл уже его, *злоумышленника*, открытый ключ? Мы легкомысленно шифруем наши данные этим ключом, *злоумышленник* перехватывает их на обратном пути, просматривает их (только он и может это сделать, так как подсунул нам *свой* шифрующий пароль), и обманывает таким же манером *товарища*, притворяясь на этот раз нами.

Такая уязвимость называется "man-in-the-middle" (дословно "человек-посередине"), своего рода "испорченный телефон". Есть два способа борьбы с ней. Первый: не доверять никаким открытым ключам, кроме тех, которые получил *лично* от *товарища*. Если с товарищем вы незнакомы, можете попросить у него удостоверение личности: а вдруг он всё-таки *злоумышленник*? Второй способ: получить от *товарища* открытый ключ несколькими *независимыми* путями. В этом случае подойдёт т. н. "отпечаток пальца" (fingerprint) -- получаемая из ключа контрольная сумма такого размера, что подделать её ещё невозможно, но уже нетрудно сравнить с этой же контрольной суммой, размещённой на WWW-странице или присланной по электронной почте.

## Пересылка почты

Ещё один немаловажный сервис, отлично поддерживаемый в Linux, -- пересылка электронной почты. Протокол SMTP (Simple Mail Transfer Protocol), задающий порядок пересылки почты, впервые был описан и помещён в **RFC** в самом начале 80-х годов. С тех пор он неоднократно модифицировался, однако в основе своей остался прежним: SMTP -- это протокол передачи *текстовых* сообщений, снабжённых вспомогательными **заголовками**, часть из которых предназначена для почтового *сервера*, передающего сообщения, а часть -- для почтового *клиента*, с помощью которого пользователь просматривает эти сообщения.

Постоянно пополняемое собрание рабочих материалов (технических отчётов, проектов **RCF** и описаний стандартов протоколов), используемых разработчиками и пользователями Internet.

В качестве адреса в электронном письме обычно используется сочетание *пользователь@доменное\_имя*. Изначально поле *пользователь* совпадало с **входным именем** пользователя в UNIX-системе, а *доменное\_имя* -- с именем компьютера. Пользователь -- удалённо или через "настоящий" терминал -- подключался к системе и просматривал почту, скажем, утилитой `mail`. Когда компьютеров в сети стало больше, выяснилось, что, имея учётные записи на многих машинах, почту всё-таки удобнее хранить и читать на одной, только для почты предназначенной, а значит, *доменное\_имя* в почтовом адресе может не совпадать с доменным именем персонального компьютера адресата. Для удобства решили ввести один уровень косвенности: прежде, чем соединиться с *компьютером "доменное\_имя"*, почтовый сервер проверяет, нет ли в DNS записи вида *доменное\_имя . . . MX . . . сервер*. Эта запись означает, что почту, адресованную на

*пользователь@доменное\_имя* необходимо посылать компьютеру "*сервер*", а уж тот разберётся, что делать дальше.

Иногда необходимо, чтобы сервер принимал письма, *не* предназначенные для зарегистрированных на нём пользователей. Эти письма немедленно помещаются в очередь на отправку, и пересылаются дальше. Такой режим работы сервера называется "relay" (пересыльщик). Когда-то все почтовые сервера работали в режиме "open relay", т. е. соглашались пересылать почту откуда угодно куда угодно. К сожалению, этим немедленно стали пользоваться желающие подзаработать массовой рассылкой рекламы (т. е. "спамом"). Поэтому открытые пересыльщики сегодня запрещены **RFC**. Однако как минимум в трёх случаях сервер имеет право работать пересыльщиком: если он пересылает почту от абонента обслуживаемой сети, если письмо адресовано в обслуживаемый домен или его поддомен, и если письмо исходит непосредственно от почтового *клиента* пользователя, который предварительно каким-нибудь способом идентифицировался в системе (например, с помощью разработанного для этого расширения SMTPAUTH). Во всех трёх случаях ответственность за возможные злоупотребления почтовым сервисом возлагается на администратора сервера, так как он имеет возможность отыскать провинившегося пользователя.

В Linux существует несколько различных почтовых серверов. Во-первых, Sendmail, корифей почтового дела, возникший вместе с SMTP. Возможности этого сервера весьма велики, однако воспользоваться ими в полной мере можно только после того, как научишься понимать и *исправлять* содержимое конфигурационного файла `sendmail.cf`, который уж более двадцати лет служит примером самого непонятного и заумного способа настройки. Впрочем, на сегодняшний день вокруг `sendmail.cf` на языке препроцессора m4 написано несметное, на все случаи жизни, число макросов, так что `sendmail.cf` редактировать не приходится. Вместо него из этих макросов составляется файл `sendmail.mc`, небольшой и вполне читаемый, а он, с помощью утилиты m4, транслируется в `sendmail.cf`, который не читает никто, кроме самого Sendmail. К сожалению, упростить *исходный текст* этой программы невозможно, поэтому специалисты по компьютерной безопасности не любят давать относительно неё гарантии: редко, но находится в `sendmail` какая-нибудь замысловатая уязвимость.

Другой вариант почтового сервера, Postfix, весьма гибок в настройке, прекрасно подходит для почтовых серверов размера предприятия, и, в отличие от Sendmail, более прозрачно спроектирован и написан. Он поддерживает все хитрости, необходимые современной почтовой службе: виртуальных пользователей, виртуальные домены, подключаемые антивирусы, средства борьбы со спамом и т. п. Настройка его хорошо документирована, в том числе и с помощью комментариев в конфигурационном файле (как правило, `/etc/postfix/main.cf`, и с помощью файлов-примеров.

Стоит упомянуть ещё как минимум три почтовых службы: QMail -- по мнению многих, этот демон наиболее защищён и от атак со стороны, и от возможных и несуществующих ошибок в собственных исходных текстах; Exim -- как наиболее гибкий в настройках (в том



числе и пока не реализованных); и ZMailer, предназначенный для работы на больших и очень больших серверах, выполняющих, в-основном, работу по пересылке.

## Доступ к почтовым ящикам

Электронная почта нужна далеко не только тем, кто имеет терминальный доступ Linux-машине. Доступ к **почтовому ящику** на сервере не должен зависеть от того, есть ли у данного пользователя право запускать на этом сервере какие-то программы. Для этого необходимо организовать специальную службу, предоставляющую пользователю только возможность манипулировать сообщениями в своём ящике с помощью программы-клиента. Самые популярные протоколы доступа к ящикам -- POP3 (**Post Office Protocol** версии 3) и IMAP4 (**Internet Message Access Protocol** версии 4).

POP3 -- довольно простой протокол, в нём определён единственный почтовый ящик пользователя, где тот может посмотреть список заголовков сообщений, прочитать (скачать) некоторые из них и удалить некоторые из них. Такой протокол удобен, когда пользователь хранит всю переписку на своём компьютере, а удалённый почтовый ящик служит исключительно для приёма входящей почты. Протокол IMAP4 гораздо сложнее: в нём разрешено заводить *несколько* ящиков на сервере, в том числе и вложенных подобно каталогам. Каждый из этих ящиков может обладать особыми свойствами: может быть входящим (тогда пользователь уведомляется о новых поступлениях в этот ящик), мусорной корзиной (сообщения из которой удаляются после того, как устареют), и даже быть *исходящим* (в такой ящик пользователь складывает новые письма, а сервер их через некоторое время отсылает, удаляя оттуда). IMAP4 подходит для ситуации, когда пользователь не имеет возможности хранить свою переписку и/или обрабатывать её с одного и того же компьютера, поэтому хранит её в ящиках на сервере. IMAP4 используют и в качестве "движка" т. н. WEB-почты, этого заменителя почтовых клиентов для торопливых.

В большинстве случаев с помощью одного и того же почтового клиента можно и просматривать электронные письма в ящиках, и создавать новые письма, и отсылать их. Это сделано для удобства пользователя, однако стоит понимать, что общее у этих трёх действий -- только формат сообщения. Строго говоря, при доступе к почтовому ящику совершенно неважно, каким путём там оказалось письмо, а при пересылке почты никак не определяет, каким способом пользователь будет её читать. Что касается *написания* письма -- чьё же это дело, как не текстового редактора?

Как водится, в Linux есть несколько IMAP/POP-серверов. Наиболее мощный из них -- Sjugus (его авторы участвовали в разработке протокола IMAP4), в нём поддерживается больше всего дополнений и расширений IMAP, которые бывает удобно использовать, наиболее простой -- UW-IMAP, разрабатываемый в университете штата Вашингтон. UW-IMAP вообще не имеет конфигурационного файла: пользователи, пароли, входящие почтовые ящики и домашние каталоги для личных почтовых ящиков берутся системные (при этом вместо командного интерпретатора почтовому пользователю можно выдать `/sbin/nologin` или `/usr/bin/passwd`). Сервер Vinc можно рекомендовать на системах, использующих QMail, интеграция с которым других IMAP-серверов имеет некоторые

шероховатости, а сервер Dovecot -- тем, кто, как и его авторы, в первую очередь озабочен надёжностью работы сервиса(3).

Протоколы POP3 и IMAP4, как и многие другие, -- текстовые. Как и в большинстве других протоколов, это порождает проблему передачи пароля в открытом виде. Решается она так же, как и для других протоколов -- "заворачиванием" всего сеанса в SSL (порту 110--POP3 соответствует порт 995--POP3S, а порту 143--IMAP4 -- 993--IMAPS), либо использование внутрисеансового шифрования с помощью TLS. Кроме того, в протоколе POP3 есть и собственное расширение, APOP, решающее ту же задачу. Здесь Мефодий опять вспомнил своего незадачливого приятеля: чтобы не сильно задумываться, тот всегда использовал один и тот же пароль, в том числе и для доступа к почте по протоколу POP3 безо всяких SSL/TLS/APOP... Увы, и эта беззаботность ему даром не прошла: однажды его учётной записью кто-то воспользовался для *отсылки* почты с помощью SMTPAUTH. Конечно же, вся отосланная почта оказалась спамом, и приятель ещё долго расхлёбывал неприятности.

---

(1) скорее всего, как это видно из выдачи `traceroute` в предыдущей лекции, на компьютере с адресом 10.13.0.1

(2) Конечно, сами файлы при этом должны быть исполняемыми.

(3) Точнее, потенциальной **ненадёжностью**, так как все перечисленные службы, конечно, тоже работают исправно.

## 16. 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 для пользователя, -- 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**. На сегодняшний день 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**

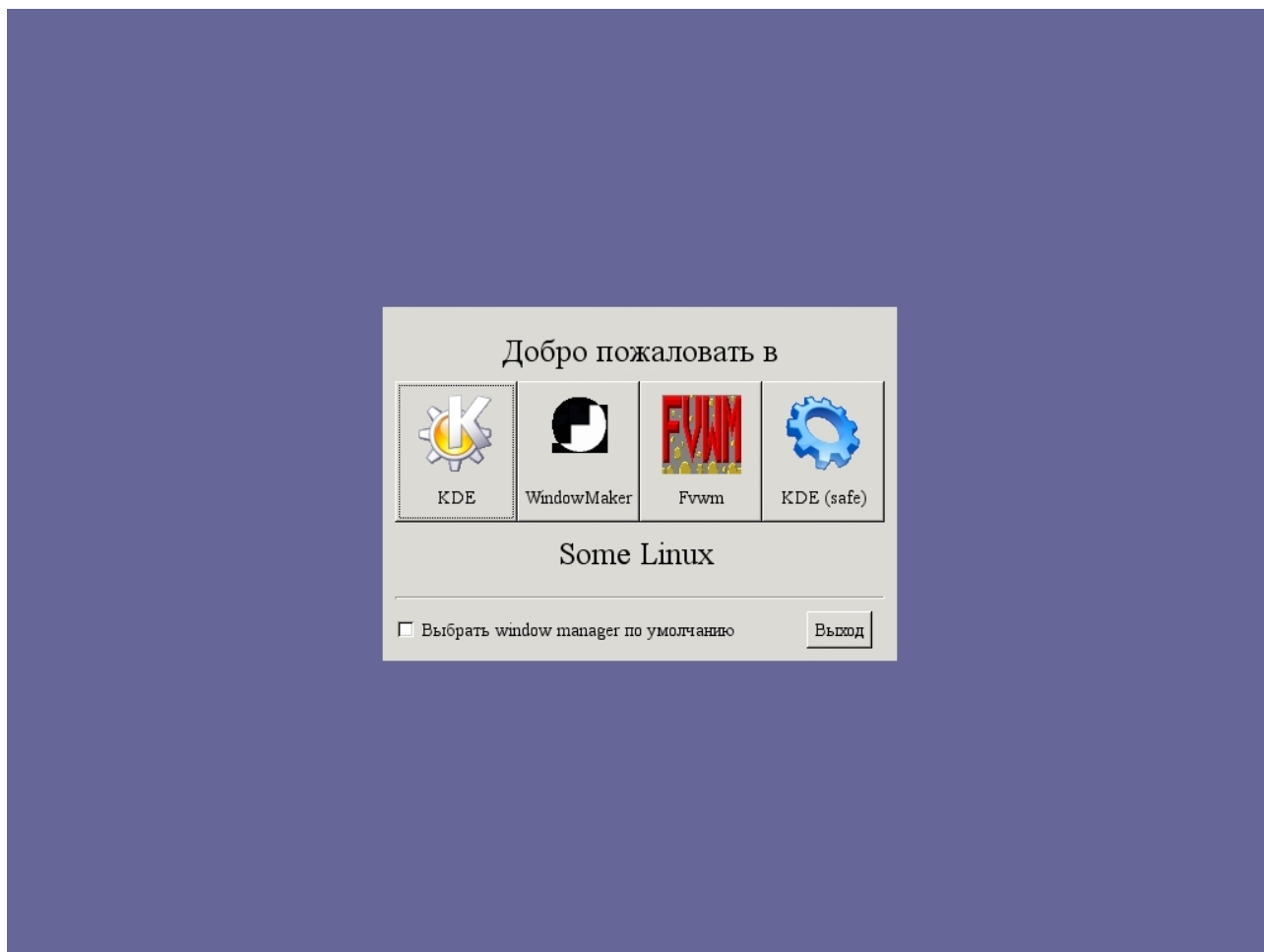
Модель интерфейса, в которой пространством ресурсов является экран -- прямоугольная область, в которой организуется ввод и вывод. Субъектами в оконном интерфейсе выступают задачи, вводящие и выводящие данные в рамках окна -- области в рамках экрана.

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 может даже вовсе отсутствовать программное обеспечение для организации графического интерфейса(1). 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 жёсткого диска в терминах объектной модели файловой системы. Поэтому весь комплекс программ для работы с графическими устройствами принято называть **графической подсистемой**.

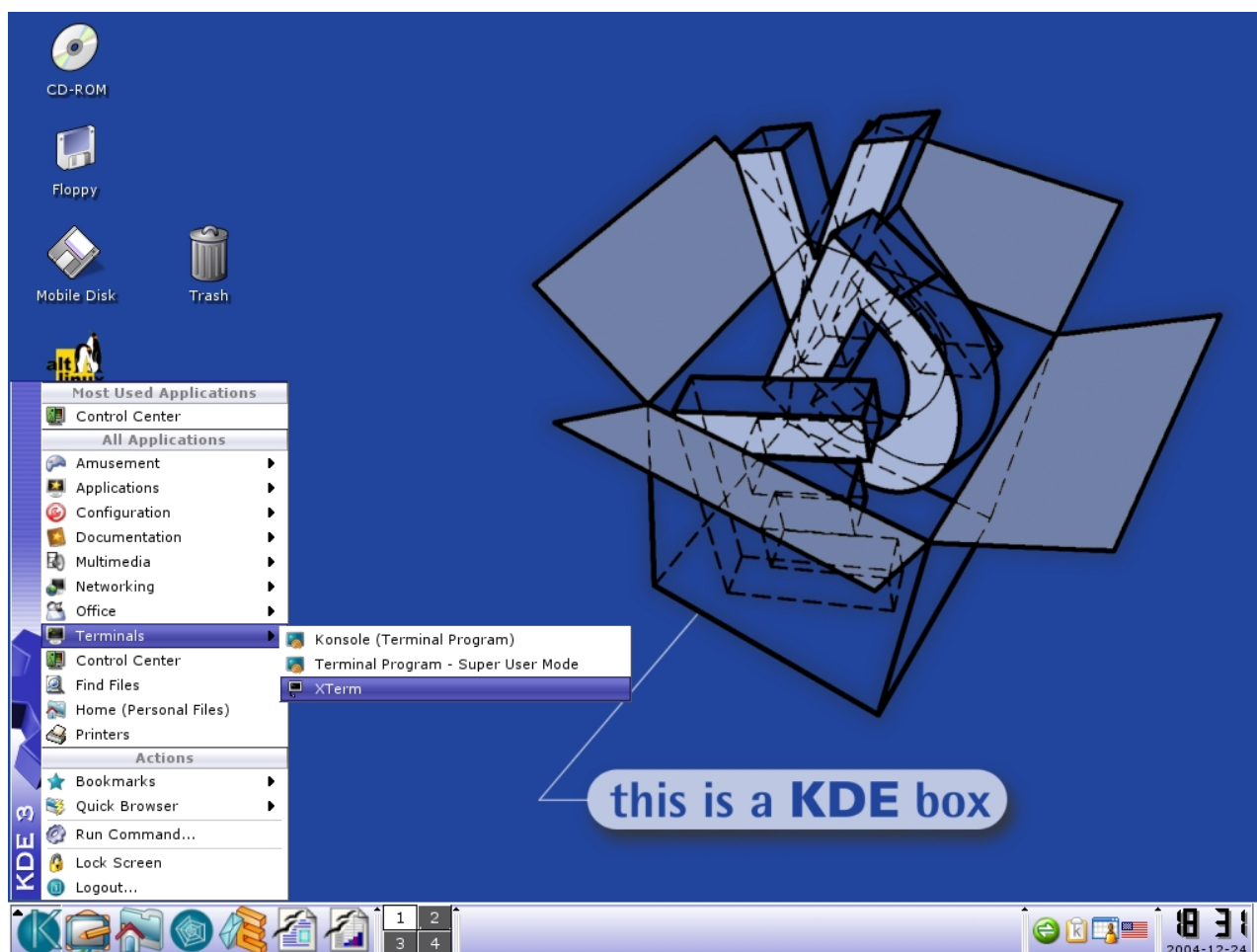
Пользователю домашнего настольного компьютера графический интерфейс почти наверняка понадобится при каждом сеансе работы. Можно настроить систему таким образом, чтобы процесс начальной загрузки завершался запуском графической подсистемы, так что даже регистрация пользователей будет происходить уже в графическом режиме при помощи специальной программы -- **экранный диспетчера** (см. лекцию [Этапы загрузки системы](#)). 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 экранный 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 экраным 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 её из командной строки в любой момент(2), для этого используется команда `startx`, которую Мефодий и исполнил.



Запуск графической подсистемы из командной строки

В некоторое недоумение поставил 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 все по очереди и выбрать ту, которая будет наиболее подходящей для его стиля работы. Не мудрствуя лукаво, Мефодий нажал на первую же кнопку, обозначенную "KDE".



Запуск KDE

После некоторого ожидания на мониторе возникло всё 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 приёмы работы. Далее в лекции объясняется устройство графической подсистемы в Linux. Станет понятно, почему процесс запуска графического интерфейса оказался таким двухступенчатым и почему работа с графическим интерфейсом в Linux может быть организована по-разному.

# X Window System

---

Обратите внимание на то, что все заглавные буквы "X" в этой лекции -- латинские.

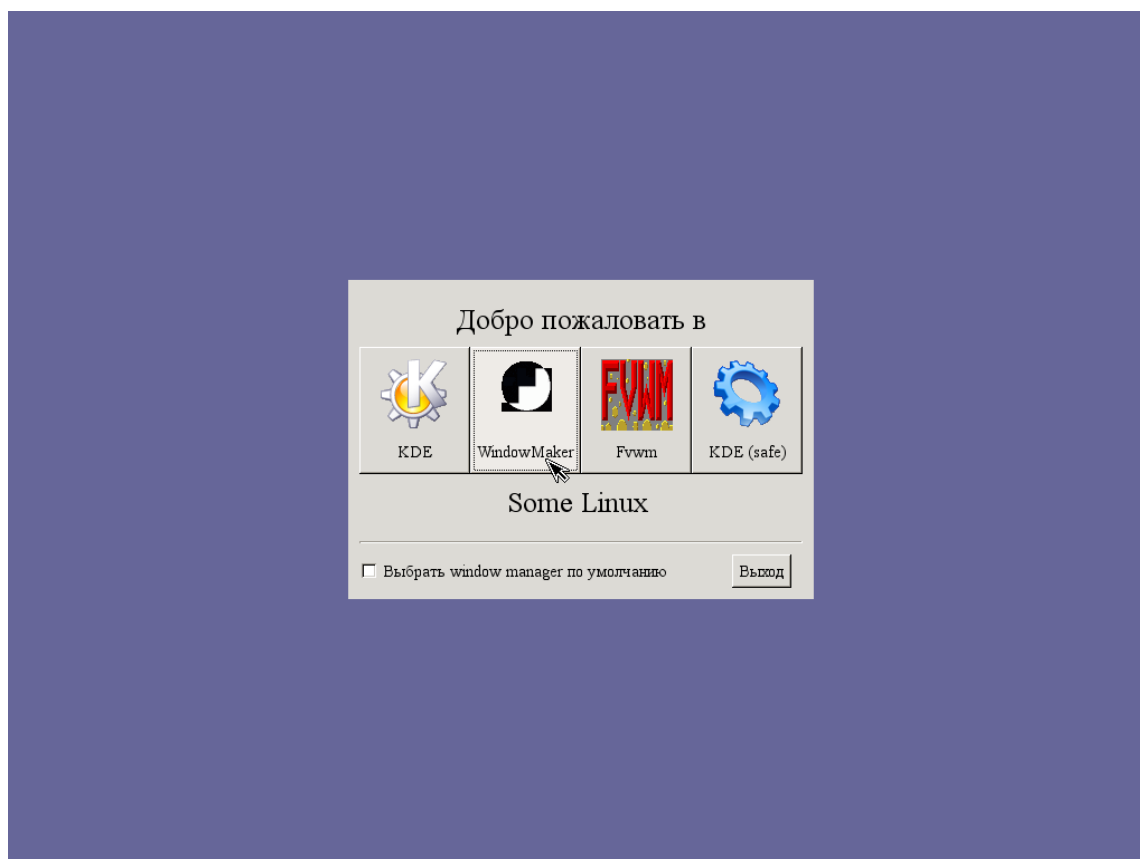
---

На свете существует множество графических устройств, управление которыми на низком уровне (вывод изображений и ввод данных, например, о перемещении мыши) -- задача совсем не для пользователя, тем более, что каждый вид устройства управляется по-своему. Заботы о вводе и выводе на низком уровне берёт на себя **графическая подсистема Linux -- X Window System**, предоставляя пользовательским программам возможность работать в терминах **оконного интерфейса**.

**X Window System** появилась всё в той же UNIX, проект этот был настолько наукоёмок и настолько полно охватывал тогдашнюю область задач, связанную с графикой, что ему так и не возникло никаких серьёзных альтернатив.

## X-сервер и X-клиенты. Протокол X11

X Window System использует традиционную оконную модель, в которой пространством ресурсов является **экран**. Экран -- это прямоугольник, на котором отображаются команды графического *вывода* и организуется обратная связь с устройствами графического *ввода*. Пример обратной связи -- указатель мыши. Сама мышь -- довольно простое устройство ввода, способное передавать информацию о его перемещении и состоянии кнопок. Указатель же отображает мнение подсистемы об абсолютных координатах гипотетической "точки ввода".



#### Расположение точки ввода (фокус)

В примере указатель мыши показывает расположение точки ввода (на кнопке "WindowMaker"). Если сейчас Мефодий нажмёт на левую клавишу мыши, графическая подсистема зафиксирует это событие ввода и передаст его той задаче, которой принадлежит соответствующая кнопка. Именно задачи (а не сидящий за монитором пользователь) и являются субъектами для **X Window System**, между ними разделяются графические ресурсы. Каждой задаче принадлежит одно или несколько **окон**, представленных некоторой (как правило, прямоугольной) частью экрана. Внутри окна выполняются графические операции (вывод) и именно окнам передаётся поток данных от устройств ввода. Какое окно получит события ввода -- определяется с помощью синтетического понятия **фокус**: вводимые данные передаются от графической подсистемы только тому окну, которое "получило фокус", по умолчанию это происходит, когда указатель мыши попадает в часть экрана, занимаемую этим окном.

В более сложном случае окна могут *перекрываться*, частично занимая один и тот же участок экрана. Если дополнительно постановить, что каждое из них лежит на своей *глубине*, то самое "верхнее" будет отображаться полностью, и ему будет доступен для вывода и получения фокуса весь заказанный прямоугольник. *Следующее* за верхним окном может быть им "загорожено", тогда *отображается* только часть этого окна, которую "видно из-под" верхнего. Заметим, что *выводить* это окно может в пределах *всего* заказанного прямоугольника, просто видно может быть не всё, и управление фокусом будет происходить на основании видимой части окна.



Программа, которая отвечает за работу с устройствами графического ввода и вывода и обеспечивает при этом логику оконной системы, называется **X-сервером** (X Server, то есть сервер системы "Икс"). В рамках **X Window System**, X-сервер -- это ядро. Подобно ядру, он выполняет низкоуровневые операции и взаимодействует с аппаратурой, ничего самостоятельно не предпринимая. Подобно ядру, он предоставляет задачам унифицированный интерфейс к этим низкоуровневым функциям, а также занимается разделением доступа (окно и фокус) к графическим ресурсам. X-сервер не волнуется, отчего эти задачи вообще появляются и чем живут. Он только принимает **запросы** на выполнение графических действий и передаёт по назначению вводимые данные. Жизнеобеспечение процессов и даже способ передачи X-запросов -- дело исключительно операционной системы, по отношению к которой и сам X-сервер -- задача.

Задачи, которые обращаются к X-серверу с запросами, называются **X-клиентами**. Обычно X-клиент сначала регистрирует окно (можно несколько), которое и будет ему полем ввода-вывода. Потом он сможет рисовать в этом окне и обрабатывать происходящие с окном **события**: активность устройств ввода и изменение свойств самого окна (размер, перемещение, превращение в иконку, закрытие и т. п.). X-клиент в Linux -- это *процесс*, запускаемый обычно *в фоне* (не связанный по вводу с терминальной линией). В самом деле, зачем процессу читать с терминала, когда для ввода он может использовать X-сервер? Если с X-сервером связаться не удастся, на стандартном выводе ошибок может появиться какое-нибудь сообщение -- его легко перенаправить в файл.

**X-сервер** Программа, принимающая и обрабатывающая X-запросы.

Клиент передаёт серверу X-запросы любым доступным ему способом. В разных версиях Linux, например, могут использоваться различные объекты файловой системы (чаще всего -- т. н. **сокеты**, сходные по функциональности с двунаправленными каналами). Во многих случаях запросы передаются по сети, при этом неважно, какой именно транспортный уровень будет использован для соединения клиента с сервером (в современных системах это, чаще всего, сеть TCP/IP и протокол TCP). Главное, чтобы клиент посылал стандартные запросы, соответствующие определённому **протоколу** обмена данными. Кстати сказать, другое имя **X Window System** -- X11 (или [X11R6](#)) -- это просто номер *версии* X-протокола, стандартизирующего X-запросы, при этом "R6" обозначает номер подверсии (revision) и вполне может увеличиться, если [X11R6](#) устареет настолько, что потребует нового пересмотра (revision).

"Голый" X-сервер, к которому ни присоединён ни один X-клиент, можно запустить из командной строки, для этого достаточно выполнить команду "X" (одна заглавная латинская буква X). Именно так и поступил Мефодий, текстовая консоль сменилась чёрным экраном без всяких окон(3).

На экране есть только крест, который перемещается при перемещении мыши -- курсор. Эта картина означает, что X-сервер запущен корректно, установил необходимую связь с устройствами графического ввода и вывода, и ожидает, когда к нему с **X-запросами**

обратится какой-нибудь X-клиент. Однако для пользователя, пока не запущено ни одного X-клиента, X-сервер совершенно бесполезен: кроме перемещения курсора ничего и невозможно сделать. Мефодий мог бы растеряться, оказавшись перед чёрным экраном X-сервера, если бы не знал о том, что может переключиться обратно на любую виртуальную консоль, нажав сочетание клавиш *Ctrl + Alt + FN*, где *N* -- номер консоли от 1 до 12(4). Переключиться обратно на экран, управляемый X-сервером, он сможет комбинацией клавиш *Ctrl + Alt + F7*.

## DISPLAY

Чтобы начать работу с графической средой, **X-клиенты** должны каким-то образом доставить свой запрос X-серверу, для этого у X-сервера должен быть какой-то точный адрес. Адрес X-сервера, к которому должны обращаться с запросом X-клиенты, хранится в **переменной окружения** DISPLAY. Формат DISPLAY прост:

*способ\_доступа: номер\_сервера. номер\_экрана*. Под способом доступа может подразумеваться сеть (тогда используется сетевой адрес машины с X-сервером) или какой-нибудь ещё механизм, принятый в конкретной системе. Если не написать ничего, будет выбран способ по умолчанию. Номер сервера нужен для различения *разных* X-серверов, запущенных на одном компьютере. В Linux можно запустить несколько X-серверов и переключаться между ними как между виртуальными консолями -- с помощью *Ctrl + Alt + F7*, *Ctrl + Alt + F8* и т. д. В системе может быть несколько *виртуальных* серверов (см. раздел [Виртуальный сервер](#)). Все они должны иметь разные номера. Наконец, один сервер может работать с несколькими экранами -- и физически (есть видеокарты с выходами на несколько мониторов), и виртуально (вот тут уж никаких ограничений нет). Правда, это бывает нечасто, и номер экрана тоже можно не указывать.

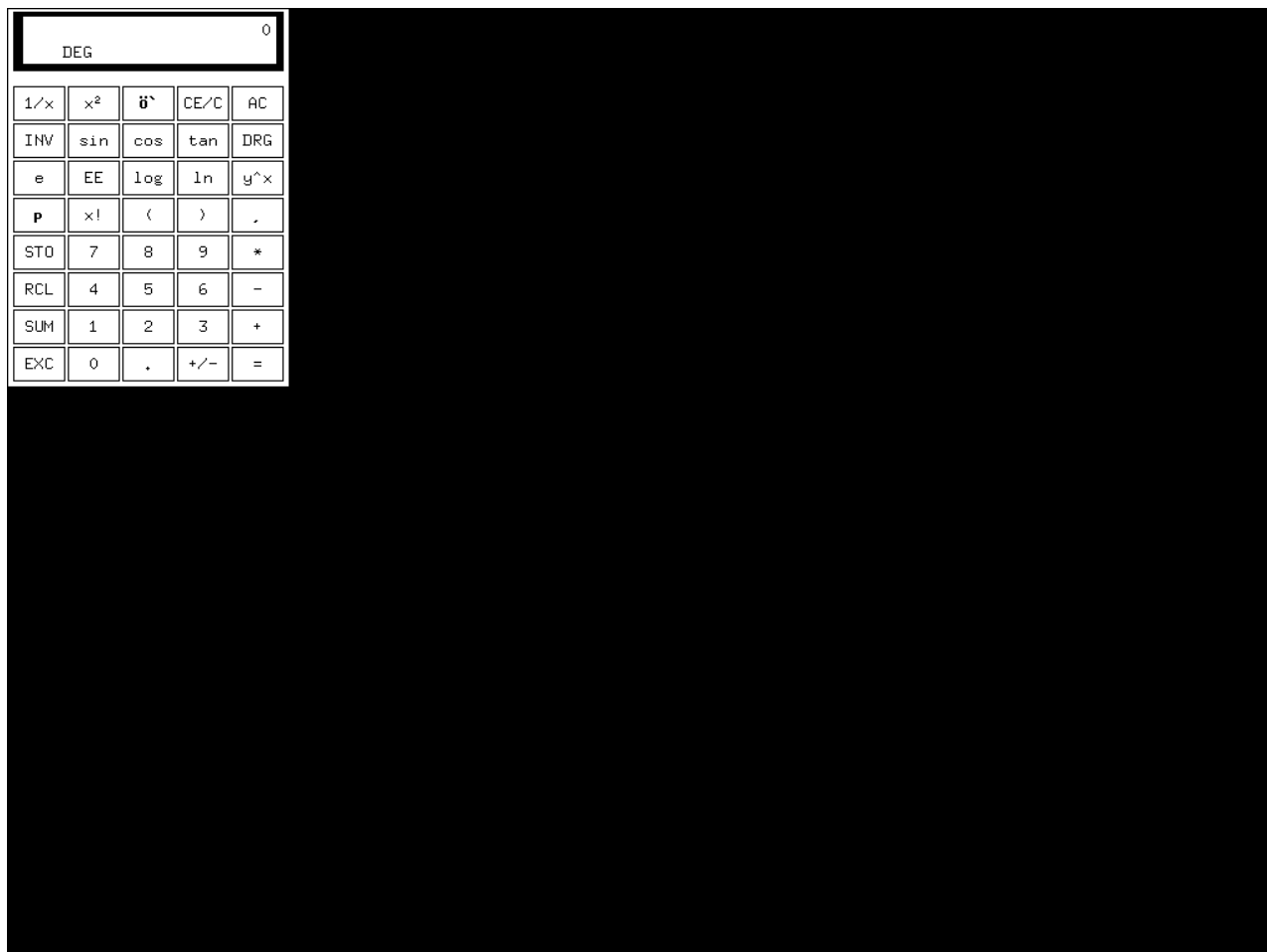
Адрес X-сервера, запущенного Мефодием, будет выглядеть так: `":0"` -- поскольку сервер запущен на той же машине, на которой работает Мефодий, можно использовать способ доступа по умолчанию (поэтому адрес начинается с двоеточия), поскольку сервер единственный, он получил номер "0", а экран можно не указывать. Теперь Мефодий может в любой командной оболочке (shell) указать адрес X-сервера в переменной DISPLAY, так что любой запущенный из этой shell X-клиент унаследует это значение и будет отправлять X-запросы тому серверу, который запустил Мефодий.

```
methody@susanin:~ $ export DISPLAY=:0
methody@susanin:~ $ echo $DISPLAY
:0
methody@susanin:~ $ xcalc &
```

### Запуск X-клиента из виртуальной консоли

В результате этих действий изменился экран X-сервера: в левом верхнем углу открылось окно калькулятора -- `xcalc`. Произошло следующее: при запуске `xcalc` проверил значение переменной DISPLAY и направил **X-запрос** по указанному там адресу; по запросу от X-клиента (`xcalc`) X-сервер выделил ему окно и выполнил необходимые

операции графического вывода, чтобы отрисовать содержимое окна (опять же, по запросам `xcallc`). Теперь при помощи мыши и клавиатуры, переместив точку ввода на это окно, вполне можно что-нибудь вычислить, однако ни переместить окно, ни изменить его размер, ни свернуть -- те операции, к которым так привыкли пользователи оконного интерфейса -- сам X-сервер не выполняет, для этих операций требуется специальная программа -- **диспетчер окон**, о которой речь пойдёт ниже.



Запуск X-клиента

С помощью специальных X-запросов можно изменить вид и поведение самого X-сервера из той же командной оболочки, в которой установлена переменная окружения `DISPLAY`. Например, команда `"xsetroot -solid green &"` изменит цвет фона на экране сервера на зелёный.

Итак, X-сервер запускается на одном компьютере, а X-клиенты вполне могут работать на других (причём на нескольких!), посылая ему запросы. С точки зрения человека, сидящего за (обратите внимание!) *X-сервером*, каждый такой клиент представлен в виде окна. Требования к аппаратуре на машинах, запускающих X-клиенты, будут изрядно отличаться от требований к аппаратуре машины для X-сервера. Типичная машина с X-сервером -- это *рабочее место* (workstation). Она должна быть оборудована качественными устройствами

ввода-вывода -- монитором, видеокартой, клавиатурой и мышью. Что же касается её вычислительных способностей, то их должно быть достаточно для выполнения X-запросов, и только. Такой компьютер не обязан работать под управлением Linux, на нём даже может вообще не быть операционной системы! В восьмидесятые годы выпускались подобные устройства, называемые "X-терминал" (X terminal).

**X-клиент** Программа, осуществляющая ввод и вывод графических данных при помощи X-запросов, обрабатываемых X-сервером.

В отличие от машины с X-сервером, компьютер для запуска X-клиентов может совсем не иметь устройств графического ввода-вывода. Его задача в том, чтобы все X-программы и запустившие их пользователи не мешали друг другу работать. На такой машине нужна хорошо настроенная операционная среда, с достаточным для запуска многих процессов быстродействием и объёмом оперативной памяти. Пара X11--Linux весьма неплохо работает на т. н. *бездисковых комплексах*. Рабочие станции в таких комплексах -- самые настоящие X-терминалы, они не имеют жёстких дисков. Вся работа происходит на центральном компьютере, с которого на рабочую станцию загружается по сети урезанный вариант системы, достаточный для запуска X-сервера, и сам X-сервер. В таких комплексах администрировать нужно одну только центральную машину, они надёжнее компьютерных залов и, что немаловажно, стоят дешевле, причём в качестве X-терминалов можно использовать и довольно маломощные, пожилые компьютеры.

## Виртуальный сервер

Одно из многих достоинств X-протокола -- в том, что X-сервером может служить любая программа, исполняющая X-запросы, а работает ли она на самом деле с каким-нибудь графическим устройством или только притворяется -- неважно. Протоколом X11 пользуется сервер печати `Xrpt`, который выводит на принтер все X-запросы или `Xvnc` -- X-сервер, *управлять* которым по специальному протоколу можно с нескольких машин. С помощью `Xvnc` (другой вариант названия -- `vncserver`) можно устраивать показ работы какого-нибудь X-клиента по сети. При этом *все* пользователи одновременно смогут запустить VNC-клиент (`vncviewer`) и гонять по экрану *один и тот же* указатель мыши (что, конечно, можно и запретить).

Виртуальный X-сервер может вообще никаких действий не выполнять, а только передавать X-запросы куда-нибудь дальше, например, "настоящему" X-серверу. Так поступает демон Secure Shell, `ssh` (программа терминального доступа, о которой шла речь в лекции [Сетевые и серверные возможности](#)), переправляя X-запросы X-серверу в *зашифрованном* виде. Этим свойством `ssh` можно воспользоваться, если сообщение по X-протоколу между двумя компьютерами невозможно (запрещено межсетевым экраном), или вы считаете такое соединение небезопасным.

```
methody@sakura:~ ssh methody@fuji
methody@fuji's password:
Last login: Sat Dec 25 13:26:40 2004 from localhost
methody@fuji:~ $ xcalc
```

```
Error: Can't open display:
methody@fuji:~ $ export DISPLAY=sakura:0
methody@fuji:~ $ xcalc
Error: Can't open display: sakura:0
methody@fuji:~ $ logout
Connection to fuji closed.
methody@sakura:~ ssh -X methody@fuji
methody@fuji's password:
Last login: Sun Dec 26 11:13:08 2004 from sakura.nipponman.ru
methody@fuji:~ $ echo $DISPLAY
localhost:10.0
methody@fuji:~ $ xcalc
# работает :) !
```

## Виртуальный X-сервер ssh

Допустим, Мефодий хочет запустить X-клиент (например, `xcalc`) на другой машине в локальной сети -- `fuji`, где у него есть учётная запись (тоже `methody`). После всех операций, проделанных в примере, на экране X-сервера на *локальной* машине Мефодия (за которой он сидит), появится ещё одно окно `xcalc`, при этом этот `xcalc` в действительности запущен на машине `fuji` и все вычислительные операции выполняются именно там.

Демон SSH заводит виртуальный X-сервер на *удалённой* машине, причём *номер\_сервера* обычно заводится таким, чтобы не пересекаться с X-серверами, которые могут быть запущены на этой машине (в примере *номер\_сервера* равен 10). Виртуальный `sshd-X` сервер принимает все X-запросы с того же компьютера и передаёт их -- в зашифрованном виде -- на компьютер, где запущен `SSH` и неvirtуальный X-сервер. Здесь все X-запросы вынимаются из `SSH`-"водопровода" и передаются местному серверу, как если бы они исходили от местного X-клиента (так оно и сеть: этот клиент -- `ssh`).

## XFree86 и X.Org

Наиболее распространённая версия *реализации* [X11R6](#) до начала 2000-х годов называлась XFree86. Эта графическая подсистема изначально проектировалась как реализация [X11R5](#) для машин архитектуры i386 -- самых распространённых на сегодня персональных компьютеров. Главная особенность этой архитектуры -- бесчисленное многообразие устройств графического вывода (т. н. видеокарт) и непрерывное нарушение их разработчиками всех мыслимых стандартов. Поэтому главной задачей создателей XFree86 было устроить гибкую структуру компоновки и настройки X-сервера в соответствии с подвернувшимся под руку устройством графического вывода, а заодно и ввода, потому что клавиатур, мышей и заменяющих их устройств на свете тоже немало. Сегодня XFree86 существует для многих архитектур и многих операционных систем.

В последние годы параллельно с XFree86 развивается основанная на тех же исходных текстах **X Window System** графическая подсистема X.Org. Поначалу по спектру поддерживаемого оборудования, архитектур и функциональности X.Org мало чем отличалась от XFree86, однако направления *развития* этих двух проектов, состав их разработчиков и

лицензионная политика несхожи. Сегодня X.Org представляется более перспективной разработкой, чем XFree86.

## Конфигурация X-сервера

Чтобы приспособить графическую подсистему (в любой реализации) к имеющемуся оборудованию, требуется организовать соответствующий **профиль**. Профиль графической подсистемы находится в каталоге `/etc/X11`, основной конфигурационный файл X.Org называется `xorg.conf`, именно его считывает при запуске X-сервер. Конфигурационный файл XFree86 называется `XF86Config` (или `XF86Config-4`).

Мы рассмотрим конфигурацию графической подсистемы на примере X.Org. Файл `xorg.conf` структурирован: состоит из нескольких обязательных разделов, которые могут следовать в любом порядке. В раздел объединяется часть профиля, связанная с одной из сторон деятельности X-сервера. Каждый раздел имеет такую структуру:

```
Section "НазваниеРаздела"  
КлючевоеСлово "Параметры"  
.  
.  
.  
EndSection
```

### Структура раздела `xorg.conf`

Внутри раздела содержатся *записи*, каждая из которых занимает обычно одну строку и задаёт значение для одного из параметров **профиля** X.Org. В начале записи стоит *КлючевоеСлово*, за которым следуют *Параметры*, количество и формат которых зависит от ключевого слова. Ниже приводится список обязательных разделов с краткими аннотациями, для чего они служат.

<code>Files</code>	Пути к файлам с ресурсами, необходимыми X-серверу
<code>ServerFlags</code>	Общие параметры X-сервера
<code>Module</code>	Расширения, которые следует загрузить
<code>InputDevice</code>	Описание устройств ввода
<code>Device</code>	Описание устройства вывода (видеокарты)
<code>Monitor</code>	Описание монитора
<code>Modes</code>	Описание видеорежимов
<code>Screen</code>	Описание экрана (связывает монитор и видеокарту)
<code>ServerLayout</code>	Конфигурация сервера

### Разделы `xorg.conf`

Почти каждый из перечисленных разделов может присутствовать в конфигурационном файле в нескольких экземплярах, например, может быть несколько разделов (`InputDevice`), описывающих разные устройства ввода (разные мыши и клавиатуры). Однако эти разделы не равноправны, а образуют иерархическую структуру, самым главным (корневым) элементом которой является конфигурация сервера (`ServerLayout`). В этом разделе указывается, какие *именно* из описанных в файле устройств ввода (разделы `InputDevice`, как минимум два -- для клавиатуры и мыши) и вывода (`Screen`, который связывает в единое устройство вывода монитор и видеокарту, ссылаясь на их описания в

соответствующих разделах) будут использованы при работе X-сервера. В каждом разделе присутствует строка "Identifier *"идентификатор"*", именно эта строка используется для выбора нужного из однотипных устройств в разделе "ServerLayout". Например, на машине, где работает Мефодий, общая конфигурация сервера выглядит так:

```
Section "ServerLayout"
    Identifier "layout1"
    Screen      "screen1"
    InputDevice "Mouse1"  "CorePointer"
    InputDevice "Keyboard1" "CoreKeyboard"
EndSection
```

Раздел ServerLayout конфигурационного файла xorg.conf

Соответственно, при запуске сервера будут использованы тот раздел Screen, в котором содержится запись "Identifier "screen1", мышь "Mouse1" и клавиатура "Keyboard1".

Чтобы разобраться в подробностях каждого раздела, требуются определённые познания в работе и характеристиках устройств ввода и вывода, поэтому здесь мы не будем приводить конкретных примеров. Подробно о формате xorg.conf можно прочитать в соответствующем руководстве xorg.conf(5). Для многих пользователей будет достаточно профиля графической подсистемы, созданного одним из существующих **мастеров**, включаемых разработчиками дистрибутивов Linux в процедуру установки системы. С их помощью можно создать более или менее подходящий профиль, не вникая в тонкости, нередко -- непосредственно при установке системы. Во всяком случае, у пользователя всегда остаётся возможность корректировать профиль вручную, отредактировав конфигурационный файл. Простой конфигурационный файл можно получить, запустив X-сервер с ключом -configure с правами суперпользователя. При этом в текущем каталоге создастся файл xorg.conf.new, в котором X-сервер сохранит результаты автоматического определения внешних устройств.

## Модули и расширения

Требование гибкости привело к тому, что в реализации XFree86 и X.Org графическая подсистема стала совсем уже похожа на операционную систему. Сам X-сервер играет роль *ядра*. Запускаясь, сервер подгружает *драйверы* -- специальные компоненты, работающие с выбранной видеокартой, и *модули* -- компоненты, расширяющие функциональные возможности сервера (в конфигурационном файле xorg.conf необходимые модули перечисляются в разделе Modules). Есть весьма нужные расширения, вроде glx (высокоуровневые функции трёхмерной графики) или freetype (поддержка шрифтов [TrueType](#)), а есть экзотические, которые иногда могут понадобиться, например, RECORD, позволяющее записывать, а после -- "проигрывать" все происходящие с сервером события.

**Расширения** называются так ещё и потому, что их возможности расширяют сам протокол X11. Вместо того, чтобы изменять протокол всякий раз, когда в голову придёт очередная ещё не реализованная в нём возможность, создатели X11 предусмотрели

стандартный способ дополнения этого протокола. При этом X- клиент, желающий воспользоваться определённым расширением, всегда может спросить у X-сервера, поддерживается ли оно, и действовать по обстановке.

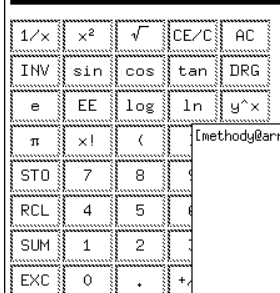
## X-приложения

Программный продукт, использующий X11, называется **приложением** (application, другое значение -- "применение"). Если считать, что сами графические возможности *уже* реализованы X-сервером и библиотеками, то программа и на самом деле окажется *приложением* к системе, и вся её заслуга будет состоять в том, что она *применила* эти возможности для решения своей задачи.

## Эмулятор терминала

С графикой или без, основным интерфейсом управления Linux была и остаётся командная строка. X11, предлагая иной способ взаимодействия с компьютером, не должна лишать пользователя возможности работать с самой системой испытанным и эффективным методом -- через терминал. Поэтому первое совершенно необходимое X-приложение должно предоставлять возможность доступа к терминалу в **X Window System**.

Задача дать пользователю X11 командную строку решается довольно легко. Нужно завести X-приложение, окно которого работает аналогично окну терминала: передаёт символьную информацию от пользователя системе и обратно. Делается это уже описанным в лекции [Mount](#) механизмом псевдотерминалов `tty/pty` (или `pts/ptmx`): X-приложение получает во владение специальное устройство типа `pty`, связанное по вводу и выводу с терминальным устройством типа `tty`, с которым работает shell. Общее название таким программам -- **эмулятор терминала для X11** (`xterm`). Мефодий может запустить `xterm` всё из той же виртуальной консоли, в которой определено значение переменной окружения





## Эмулятор терминала для X11 -- `xterm`

В левом верхнем углу открылось окно `xterm`, которое легло "поверх" открытого ранее `xcalc`. В открывшемся окне `xterm` Мефодий увидел привычное приглашение командной строки `bash`: теперь из этой командной строки можно выполнять любые команды, в том числе и запускать новые X-приложения, например, ещё один `xterm`. Чтобы при этом избежать наложения окон друг на друга, можно запустить `xterm` с параметром "`-geometry +150+150`", что заставит X-сервер выдать ему окно, верхний левый угол которого отстоит на 150 экранных точек (пикселей) от левой границы экрана, и на 150 же -- от верхней. В этом `xterm` значение `DISPLAY` унаследовано от родительского процесса и равно `":0"`, так что окна всех запущенных из него X-приложений откроются на этом же экране.

Не следует путать *программу* `xterm` со *способом* организации рабочей станции (т. н. "X-терминал"): термины эти созвучны, но относятся к разным областям знаний. Нередко бывает, что на экране X-терминала (компьютера) есть окно терминала X11 (программы `xterm`). XTerm передаёт сигналы как настоящий терминал, имеет богатый набор управляющих последовательностей (унаследованный от устройства "DEC VT102/VT220"), а вдобавок позволяет воспользоваться всеми преимуществами графической среды: выбрать шрифт, запомнить текст на экране (даже тот, что уже исчез с экрана) и многое другое.

Кстати сказать, копирование текста при помощи мыши -- свойство совсем не только XTerm. На самом деле любое окно, зарегистрированное в X11 как текстовое, позволяет отметить (при постоянно нажатой первой кнопке или последовательными нажатиями третьей) часть текста. Выделенный текст можно немедленно вставить в любое окно *текстового ввода* нажатием второй (обычно -- средней) кнопки. Различают временное хранилище (т. н. `cut buffer`), содержимое которого обновляется *всякий раз*, когда пользователь выделяет текст, и буфер обмена (т. н. `clipboard`), содержимое которого заполняется, если пользователь вызвал соответствующую команду (например, выбрал пункт меню "Сору"). `Cut buffer` -- удобное средства копирования в два движения (выделить в одном месте, нажать среднюю кнопку в другом). `Clipboard` -- более традиционный, но более трудоёмкий инструмент (выделить, скопировать, выделить вставить).

## Сеанс работы с X11

Как догадался Мефодий, команда `startx` делает несколько больше, чем просто запуск X-сервера, она кроме того запускает несколько **X-приложений**. Для удобной работы в X11 пользователю прямо при запуске графической подсистемы потребуется сразу несколько приложений. Во-первых, нужно запустить приложение, которое позволит управлять окнами (как минимум перемещать их), чего не делает сам X-сервер, такие приложения называются **диспетчеры окон** и будут обсуждаться немного позднее. Кроме того, полезно сразу запустить разные мелкие программки, вроде индикатора загрузки системы `xload` или экранных часов `xclock`. Сам X-сервер не мешает настроить с помощью `xset` (можно поменять курсор, звуковой сигнал, переименовать кнопки мыши). Одним словом,

пользователю, как правило, нужен небольшой *стартовый сценарий*, который запускался бы вместе с X-сервером.

С другой стороны, сервер хорошо бы *останавливать*, когда он больше не используется. Это, конечно, относится к схеме *без диспетчера экрана* (xdm): пользователь работает с терминала, потом запускает X-сервер для выполнения графических программ, выполняет их и останавливает сервер, чтобы графическим устройством мог воспользоваться кто-нибудь другой. Стандартный способ аварийного завершения работы X.Org (*Ctrl + Alt + Backspace*), во-первых, доступен только на X.Org, во-вторых, его можно отключить, а в-третьих, все запущенные приложения получают в этом случае сообщение о внезапной смерти сервера и тоже завершатся аварийно.

Если запускать не сам X-сервер, а некоторую *оболочку* вокруг него, называемую `startx`, то алгоритм работы будет такой. Сначала запустится X-сервер и сформируется значение переменной окружения DISPLAY. Затем запустится сценарий `.xinitrc`, находящийся в домашнем каталоге пользователя, а если такого нет -- системный стартовый сценарий `/etc/X11/xinit/xinitrc`. Предполагается, что X-сервер будет работать до тех пор, пока выполняется `.xinitrc`. Когда все команды из `.xinitrc` отработают, его выполнение завершится, а вместе с ним завершится и сервер. Поэтому рекомендуется все X-приложения из `.xinitrc`, кроме последнего, запускать *в фоне*, чтобы командный интерпретатор не дожидался окончания их работы. В этом случае с завершением последнего приложения из `.xinitrc` завершится и сам X-сервер. Последней программой в стартовом сценарии может быть `xterm`, как это сделано в стандартном `xinitrc`, или **диспетчер окон**. Для завершения `xterm` (а с ним и всего сеанса работы X11) достаточно послать "`^D`" запущенному в нём shell, а оконвод обычно имеет какую-нибудь кнопочку или менюшку "Exit". Программа, с завершением которой завершается сеанс X11, называется **лидером сеанса** (session leader).

Лидер сеанса может быть указан и в качестве параметра `startx` в командной строке, например, по команде `startx /usr/bin/xterm` будет открыт сеанс X11, лидером которого будет `xterm`, его окно откроется при запуске на экране X-сервера.

Диспетчер экрана организует сеанс работы с X11 сходным образом. Единственное отличие -- в том, что ко времени запуска `startx` вручную пользователь уже имеет настроенное **окружение** (этим занимается **стартовый командный интерпретатор**), а после регистрации в диспетчере экрана -- нет. Поэтому стартовый сценарий нужно запускать другой -- `.xsession`. На самом деле и сам X-сервер необходимо перезапускать и при использовании xdm. Несмотря на то, что пользователи взаимодействуют только с X-сервером, не используя виртуальные консоли, было бы неудобно и небезопасно сохранять какие бы то ни было настройки, сделанные *одним* пользователем ко времени работы *другого*. Самое неприятное -- это т. н. "клавиатурный вор" (keyboard grabber), программа, притворяющаяся оконводом -- для того лишь, чтобы запоминать *всё*, что пользователь ввёл с клавиатуры (злоумышленников интересуют, как правило, пароли).

Нарушения безопасности легко избежать, если *не* использовать `xhost` (авторизацию на основе *адреса*) и не доверять X-серверу, запущенному не при вас. Можно доверять автоматически запускаемой в сеансе программе `xauth`, которая связывается с X-сервером и записывает в файл `~/.Xauthority` специальный ключ доступа. Все X-приложения пользуются библиотекой `libX11`, которая обязательно обращается к этому файлу. Таким образом, X-приложение может воспользоваться X-сервером, если оно знает его адрес (указанный в переменной окружения `DISPLAY` или переданный ключом `-display`) и авторизовано сервером (по адресу компьютера или по ключу доступа в `~/.Xauthority`).

## Ресурсы X11

X-приложения создаются с помощью разнообразных готовых инструментариев. Большая их часть разрабатывается независимыми авторами по всему миру (это общее свойство свободного ПО, см. [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)), но библиотека самого низкого уровня, "X Lib", и несколько более высокоуровневых с давних пор включаются в основной дистрибутив X11. Из них наиболее примечательна "X Toolkit" (Xt), организующая из стандартных окон X11 **оконные примитивы** (widgets).

Дело в том, что каждый объект X11 обладает некоторым набором свойств, вроде размера окна, цвета фона или текстового содержимого. Для удобства доступа примитивы Xt, реализованные поверх объектов Xlib, имеют собственные имена (у каждого объекта своё) и фамилии (одинаковые у всех объектов одного класса). Более того, программа, написанная на Xt, имеет *карту* объектов, в которой указано, какой объект внутри какого находится, и приведены имена объектов и их классов. Посмотреть структуру запущенного X-приложения можно с помощью программы `editres` ("Commands/Get Tree"). Например, программа `xlogo` состоит из трёх примитивов: `xlogo` (класс `XLogo`) и вложенными в него `xlogo.xlogo` (класс `Logo`) и `shellext` (неоконный, класс [VendorShellExt](#)). Мало того, свойства этих примитивов можно подменить прямо в работающей программе. Можно, например, на время работы перекрасить фон `xlogo` в красный цвет.

Xlib, со своей стороны, предоставляет универсальный способ хранить такие настройки в файле. Многие приложения хранят свои настройки по умолчанию в `/etc/X11/app-defaults/ИмяКласса` (в некоторых системах этот каталог перенесён, как ему и подобает, в `/etc/X11`). *ИмяКласса* -- это имя класса самого главного окна этого приложения. Пользователь может дополнить настройки ресурсов, которые сервер накапливает при старте, при помощи команды `xrdb -merge файл_настроек` или переписать их заново (ключ `-load`). Приложению остаётся только вызвать специальную функцию Xlib, которая считывает настройку с определённым именем из таблиц сервера. Если `xrdb` не запускалась ни разу (в том числе и в стартовом сценарии), запрос приложения направится не в таблицу сервера, а непосредственно в файл ресурсов `.Xdefaults(5)`, лежащий в домашнем каталоге. В этом случае для изменения настроек не надо запускать `xrdb`.

Xt добавляет в эту процедуру *иерархию* ресурсов. Как это можно наблюдать в `editres`, при задании свойства ресурса используются четыре конструкции: имя ресурса,

имя класса, разделитель ". ", означающий, что один ресурс вложен в другой *непосредственно*, и разделитель "\*", означающий, что ресурс вложен в другой *в конце концов* (возможно, по цепочке). Например, для задания цвета фона в программе `xload` можно использовать именование `xload.paned.load.background` (это всё по именам). Можно попробовать единым махом изменить цвет фона *всех* примитивов этой программы, записав в `.Xdefaults` (или передав `xrdb`) строку вида `"XLoad*background: midnightblue"`.

Если в `.Xdefaults` есть *несколько* строк, удовлетворяющих имени ресурса, то имена собственные имеют преимущество над классами, а ". " -- над "\*". Так что запись вида `"*Text*background: peachpuff"` повлияет только на те примитивы класса `Text` и вложенные в них, для которых нет более строгого указания (например, `"*Text.background"` или `"XConsole*Text*background"`). Обратите внимание, как поэтично называются порой цвета в X11! Посмотреть таблицу цветов можно командой `x1scolors`.

К сожалению, `Xt` -- всё-таки довольно старая библиотека, не во всём следующая новым веяниям в области графического интерфейса. Для быстрой разработки оконных программ нужны разнообразные и мощные инструменты, которые манипулируют понятиями, далеко ушедшими от стандартного "окно--меню--кнопка". Если такие инструментариумы написаны с применением `Xt` (например `Xm`, "open Motif", `Xaw`, "Athena Widget"), их настройки можно поселить в `.Xdefaults`. Незадача ещё и в том, что структуру классов `Xt` неудобно использовать при программировании на языке `C++`, а именно на нём сейчас пишется большинство оконных приложений. Так что самые мощные инструментариумы для X11 -- `Qt` ("The Q Toolkit") и `GTK` ("The GIMP toolkit") -- не используют `App-defaults`.

## Диспетчер окон

### Задача диспетчера окон

`X`-сервер берёт на себя ответственность только за выдачу `X`-приложению области для ввода-вывода и управление **фокусом** -- окна, но не занимается никакими манипуляциями по изменению этого окна: перемещением, изменением размера, сворачиванием и т. п. Мефодий уже наблюдал `X`-сервер, к которому присоединено ровно два клиента, чьи окна перекрываются: два чёрно-белых окна друг на друге, и их ни растащить по углам, ни сжать нельзя. `X`-сервер умеет очень ловко манипулировать окошками, но сам никогда ничего не делает, дожидается команды от пользовательской программы. А какой программе захочется самостоятельно отслеживать перекрытие окон, фокус, заниматься изменением размера, перемещением и тому подобным? Ведь основная задача программы может быть совсем другой...

Ответ очевиден: этим должна заниматься программа, *основная задача* которой состоит в том, чтобы отслеживать перекрытие, изменять размер, двигать, превращать в иконку и так далее. По совместительству эта же программа будет рисовать при окнах всякие красоты: рамочки, заголовки, кнопки и меню управления -- словом, делать всё, что нужно для организации *логики* управления окнами. Для этого придётся также обрабатывать практически все события, передаваемые от устройств ввода, и многочисленные "подсказки" от

приложений относительно того, какими они хотят видеть собственные окна. Это X-приложение называется **диспетчером окон** (window manager)(6).

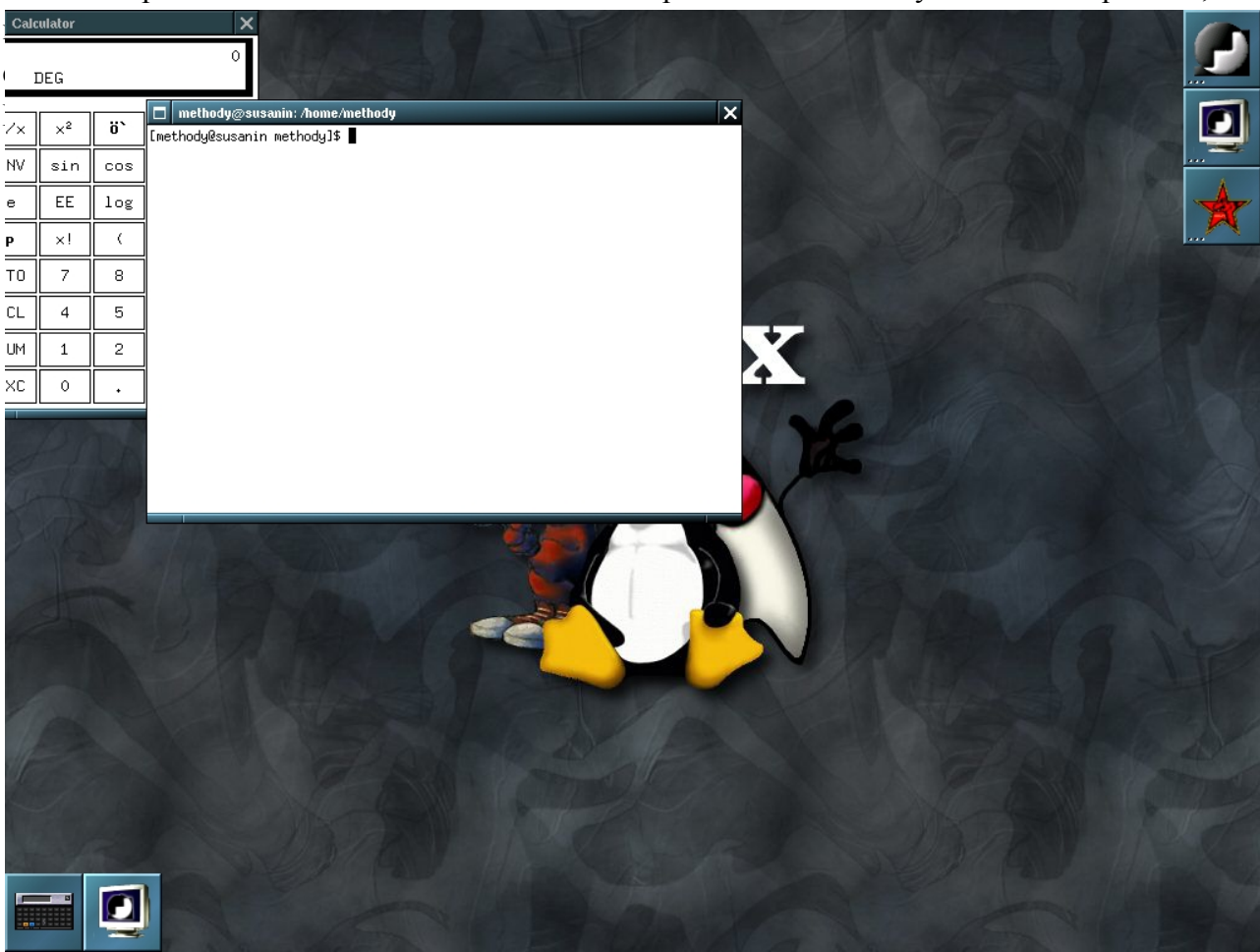
**диспетчер окон** Программа, основная функция которой -- обеспечивать манипуляции с окнами: перемещение, изменение размера, сворачивание и т. п.

Благодаря стандартному протоколу X11 появилось такое множество диспетчеров окон для **X Window System**, что перечислить их все просто невозможно. Все они различаются видом и кругом возможностей для манипулирования окнами: от самых простых (рамочка вокруг окна позволяет двигать его, изменять размер и поднимать из глубины; собственно, всё) до весьма изощрённых (виртуальные экраны, анимированные полупрозрачные меню, панели инструментов, причудливой формы украшения на окнах; сами окна ползают по экранам, кувыркаются, растворяются как утренний туман; всё это лязгает, попискивает и разговаривает приятным женским голосом). Когда Мефодий в первый раз запустил команду `startx`, была запущена утилита `wm-select`, которая предложила ему выбрать, какой из установленных в его системе диспетчеров окон запустить.

Выбор диспетчера окон на свой вкус -- очень непростое и вдумчивое занятие. Мы советуем просто соблюдать меру, сообразуясь с тем, для чего вы используете оконную систему: обилие ярких декораций отвлекает от работы (а если они вдобавок шевелятся?), а слишком строгий минимализм её усложняет. Имейте в виду, что чем причудливее и многообразнее возможности оконвода, тем труднее будет его *полностью* настроить именно под себя. Скорее всего, вы просто согласитесь пользоваться уже настроенными -- общеупотребительными -- возможностями, не доводя их до совершенства.

## Работа с окнами

Совершенно необязательно начинать сеанс работы в X11 с запуска диспетчера окон, его



## Диспетчер окон [WindowMaker](#)

С запуском диспетчера окон экран X-сервера заметно преобразился: одноцветный фон сменился изображением, по углам возникли квадратные кнопки, а вокруг окна `xterm` образовалась рамка с названием окна и кнопками. Эта рамка -- "органы управления" окном, так называемые **оконные примитивы** (их собственное имя -- "widgets" -- сокращение от "window gadgets", "оконные безделушки"). "Ухватившись" мышью за рамку, Мефодий поменял размеры окна, а нажав на правую кнопку мыши на заголовке окна -- увидел меню, состоящее из списка операций, которые можно произвести над этим окном. Все эти возможности предоставлены диспетчером окон WindowMaker.



Меню операций с окном в WindowMaker

### Удобства

Однако помимо своей основной функции -- операций с окнами -- многие диспетчеры окон владеют массой других приёмов по упрощению работы пользователя и повышению наглядности. Самые главные из таких удобств -- кнопки для запуска типичных задач: нажатие на кнопку заменяет собой выполнение команды в эмуляторе терминала. Команда может быть длинной или забываться, а нажатие одно и кнопка всё время на виду -- это удобно для привычных повседневных действий. Например, в правом верхнем углу экрана

WindowMaker присутствует кнопка с изображением монитора -- для запуска `xterm`. Вторая важная возможность -- общее меню, в котором классифицированы доступные в системе приложения, при помощи такого меню пользователь может найти и запустить нужную прикладную задачу.



Общее меню в WindowMaker

Большое удобство, предоставляемое очень многими диспетчерами окон для X11, -- **виртуальные экраны**. Когда пользователь работает сразу с несколькими задачами, ему обычно приходится открывать много окон, так что они уже не умещаются без наложения на физической поверхности экрана. Чтобы при этом переключаться между задачами было удобно, можно использовать некоторые специально для этого предназначенные функции диспетчера окон: сворачивание и разворачивание, перекладывание окон "выше" и "ниже" в стопке, список активных задач и т. п. Но ещё удобнее было бы не перекладывать окна, а разложить их на большей, чем размеры экрана, поверхности -- **виртуальном экране**. Таким образом настоящий экран -- это только небольшое "окошко", в которое можно видеть только часть виртуального, а при необходимости можно это окошко сместить в другой конец "виртуального стола", где лежат окна с другими задачами.

Диспетчер окон организует виртуальный экран сам: X-сервер при запуске диспетчера окон выдаёт ему одно окно размером во весь экран, так что все остальные окна оказываются внутри него, и диспетчер окон уже сам решает, когда и кому передать фокус, как обойтись с окнами и т. п. При этом он вполне может "делать вид", что его экран больше экрана монитора, по определённой команде (переключиться на другой конец виртуального экрана) запоминая и пряча текущее расположение окон и заменяя его другим, до этого хранившимся в памяти. Конфигурация виртуального экрана может быть любой -- она зависит только от логики работы диспетчера окон. Виртуальный экран может состоять из нескольких частей размером в реальный экран, поставленных в ряд, доступных по номерам, организованных в виде прямоугольника, бывают даже трёхмерные виртуальные экраны.

Виртуальные экраны есть и в WindowMaker, переключение между ними осуществляется при помощи специальной кнопки в левом верхнем углу экрана или сочетаниями клавиш  $Alt + N$ , где  $N$  -- номер виртуального экрана. Однако чтобы не забывать, где лежит какое окно, полезна возможность окинуть одним взглядом всё, разложенное на виртуальном экране. Окно, отображающее в уменьшенном масштабе вид виртуального экрана и позволяющее перейти к нужной части, называется **пейджер** и относится к очень распространённым в диспетчерах окон удобствам.

### Настройка диспетчера окон

Удобство графического интерфейса -- понятие очень субъективное, при этом технологически безразлично, рисовать ли кнопки в левом верхнем углу или в правом нижнем, какой цвет и шрифт использовать в меню и т. п. -- всё это составляет **профиль** диспетчера окон. Поэтому в подавляющем большинстве диспетчеров окон существуют более или менее развитые возможности изменить внешний вид и оформление графической среды. В WindowMaker для этого используется специальная утилита с графическим интерфейсом (Мефодий может её вызвать при помощи самой верхней кнопки в левом верхнем углу экрана). В других диспетчерах окон, как, например, в очень развитом и fvwm, профиль может просто храниться в конфигурационном файле ( $\sim / .fvwm2rc$ ), для изменения внешнего вида fvwm следует редактировать этот файл.

Среди диспетчеров окон в тяжёлых весовых категориях выступают и fvwm2 (множество функций), и Enlightenment (настоящая игрушка), и WindowMaker, и некоторые другие. Человеку, проводящему рабочее время за экраном компьютера, важен комфорт на рабочем месте, поэтому всякая мелочь или её отсутствие может иметь решающее значение при выборе того или иного диспетчера окон.

### Рабочий стол

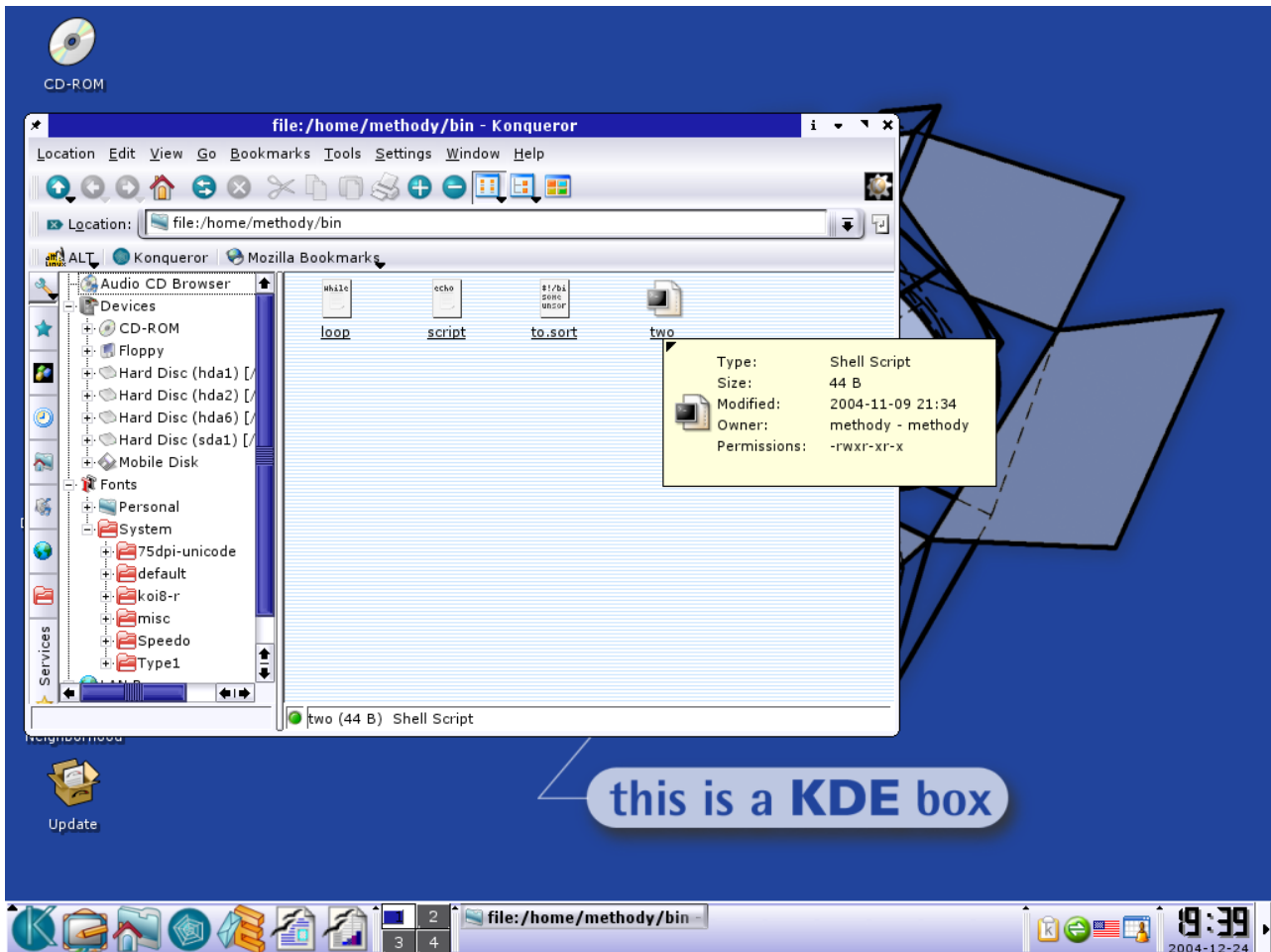
С появлением универсальных высокоуровневых инструментариев стала обретать плоть идея *полного* воплощения метафоры "рабочего стола", впервые реализованная в системах семейства MacOS. Смысл рабочего стола в том, чтобы предложить принципиально иной способ взаимодействия человека и компьютера -- способ, основанный на манипуляции единичными именованными объектами. Каталоги превращаются в "папки с документами", причём каждый тип документов можно "открыть" с помощью специального "документатора".



Программы превращаются в эдакие поместилща абстрактных функциональностей: "Интернет", "Почта", "Видео" и т. п. Рабочий стол особенно потребен человеку, чья область деятельности далека от компьютерного дела, для кого он -- не вычислительная машина, а инструмент решения отдельных -- типовых и, чаще всего, *непрофильных* -- задач.

Компьютер так стремительно вошёл "в каждый дом", что давно уже стал бытовым прибором для игр, чтения электронной почты, просмотра WWW, а в последнее время -- музыкальным центром и видеопроигрывателем. Грешно требовать от садыщейся за клавиатуру домохозяйки или какого-нибудь оболтуса *строого* следования принципам *проективной* системы. Лучше дать им в руки красивую и сравнительно безопасную *игрушку*, которая способна удовлетворять бытовые их нужды. Таких игрушек для X11 несколько. Две мощные среды "офисного" плана -- KDE (основанное на Qt переосмысление коммерческой среды CDE) и Gnome (основанная на GTK) -- содержат всё необходимое для деловой работы (включая *собственные* офисные приложения и средства просмотра WWW). Или, например, среда XFCE (основанная также на GTK) -- крепко сколоченный минималистский вариант CDE, простой и ясный, как выдвигной ящик.

Мефодий выбрал одну из таких игрушек -- KDE, начав с оформления рабочего стола на свой вкус. Когда ему, наконец, наскучило перекрашивать меню и украшать рабочий стол безделушками, он попробовал заняться собственно делом. В примере отображён снимок экрана в один из моментов его работы, когда он попытался воспользоваться для работы с файлами специально разработанной для KDE программой (konqueror, которая служит заодно и браузером) вместо уже привычной ему командной строки. При этом Мефодия насторожило время, которое ему потребовалось на выполнение вполне привычных простейших операций над несколькими файлами и каталогами: для каждого потребовалось отдельное движение мышью, да и в целом всё стало происходить как-то гораздо медленнее, чем обычно, а компьютер то и дело принимался хрустеть жёстким диском.



## Работа в KDE

До сей поры Мефодий считал, что его -- не такой уж новый -- компьютер вполне подходит для работы в Linux, и ничто этому утверждению не противоречило. Для того, чтобы проверить, чем же занимается система, он запустил команду `top`, нажал "M" для того, чтобы посмотреть, какие процессы занимают больше всего памяти и обнаружил довольно неприятную, хотя и вполне терпимую картину:

```
00:55:08 up 13:20, 13 users, load average: 1,71, 1,87, 0,97
29 processes: 28 sleeping, 1 running, 0 zombie, 0 stopped
CPU states: 60,6% user, 14,6% system, 6,2% nice, 0,0% iowait, 18,4% idle
Mem:      54232k av,  53076k used,   1156k free,      0k shrd,   1628k buff
          18032k active,           22620k inactive
Swap:    200772k av, 108288k used,  92484k free                15812k cached
```

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	COMMAND
15410	methody	13	5	9448	8980	8096	S N	0,0	16,5	0:01	kdeinit
15379	methody	9	0	9656	7928	7184	S	0,1	14,6	0:06	kdeinit
15404	methody	9	0	9652	7612	7428	S	0,0	14,0	0:02	kdeinit
15395	methody	9	0	9596	7376	7372	S	0,0	13,6	0:02	kdeinit
15406	methody	9	0	10084	7216	6192	S	0,0	13,3	0:09	konqueror
15397	methody	9	0	8592	7140	6560	S	0,5	13,1	0:03	korgac
15387	methody	9	0	8464	6920	6748	S	0,0	12,7	0:02	kdeinit
15390	methody	9	0	8488	6644	6640	S	0,0	12,2	0:01	kdeinit
15393	methody	9	0	8576	6636	6632	S	0,0	12,2	0:02	kkbswitch

15407	methody	9	0	6864	6164	6064	S	0,0	11,3	0:00	kdeinit
15377	methody	9	0	7592	5844	5440	S	0,0	10,7	0:04	kdeinit
15380	methody	9	0	6564	5736	5624	S	0,0	10,5	0:00	kdeinit
15346	methody	9	0	6652	5028	4744	S	0,1	9,2	0:02	kdeinit
15368	methody	9	0	6864	4972	4972	S	0,0	9,1	0:02	kdeinit
15376	methody	9	0	6164	4504	4504	S	0,0	8,3	0:01	kdeinit
15356	methody	9	0	6608	4436	4432	S	0,0	8,1	0:01	kdeinit
15366	methody	9	0	6008	4436	4436	S	0,0	8,1	0:01	kdeinit
15343	methody	9	0	5248	4388	4156	S	0,1	8,0	0:00	kdeinit
15385	methody	9	0	5412	3984	3876	S	0,0	7,3	0:01	apt-indicator
15341	methody	9	0	4540	3768	3612	S	0,0	6,9	0:00	kdeinit
15338	methody	8	0	2260	1444	1368	S	0,0	2,6	0:00	kdeinit
15411	methody	15	0	1092	1040	844	R	2,3	1,9	0:01	top
15200	methody	9	0	628	520	520	S	0,0	0,9	0:00	xinit
15159	methody	9	0	912	508	508	S	0,0	0,9	0:00	bash
15209	methody	9	0	640	476	472	S	0,0	0,8	0:01	startkde
15185	methody	9	0	564	408	408	S	0,0	0,7	0:00	startx
15364	methody	9	0	304	276	256	S	0,0	0,5	0:00	kwrapper

### Загрузка компьютера во время работы KDE

Первое, что бросилось ему в глаза -- множество процессов, запущенных явно средой KDE (кому же ещё может принадлежать программа `kdeinit`?), Мефодий подсчитал число процессов, начинающихся на "kde" (`ps ax | grep kde | grep -v grep | wc -l`) -- их оказалось 17 штук. Каждый из этих процессов затребовал у системы по три-четыре мегабайта памяти (поле `SIZE`), из которых полтора-два (поле `RSS`) немедленно использовал. Не так уж и много, если бы такая программа запускалась *одна*. Но две дюжины `kdeinit` вполне способны израсходовать *всю* оперативную память компьютера, если объём его физической памяти равен, как на компьютере Мефодия, 64 мегабайта (из них порядка девяти мегабайтов заняло ядро -- эта память не отображается в поле `mem` -- и сколько-то уходит на сам X-сервер и прочие программы, не принадлежащие Мефодию).

Впрочем, даже в таком состоянии Linux продолжает работать довольно-таки бодро. Дело в том, что большинство из этих процессов (по словам `top` -- все, кроме одного, самого `top`, об это говорит строка "1 running") в данный момент неактивны (`sleeping`). Большинство полученных ими ресурсов система давно уже отправила в область подкачки (`swap`) на диске. Затруднения начнутся, если несколько неактивных программ "проснутся": система начнёт поднимать из `swap` их ресурсы, а чтобы для них хватило места в оперативной памяти -- откачивать туда память других программ (отсюда и неожиданная дисковая активность, на которую Мефодий обратил внимание). Хуже всего, если для работы всех активных процессов *одновременно* не хватает места, тогда процесс откачки-закачки будет отнимать большую часть процессорного времени, для полезных задач его просто не останется. Определить такую ситуацию (она называется "дребезг", `thrashing`), можно по высоким значениям полей `system`, а ещё по постоянно ненулевым значениям в полях `Si` и `so` команды `vmstat`.

procs			memory			swap		io		system			cpu	
r	b	w	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id
0	1	0	106092	1352	1168	14	10	265	33	127	407	14	4	82

## Вывод команды `vmstat`

Этот опыт произвёл отрицательное впечатление на Мефодия, и он решил устраивать себе графическое рабочее место на основании какого-нибудь менее громоздкого инструмента. Впрочем, провести границу, где заканчиваются обязанности диспетчера окон и начинаются ухищрения рабочего стола, очень трудно. Видимо, разумно считать, что диспетчер окон делается средой рабочего стола, когда появляются *пользовательские* приложения с использованием его особых свойств и его библиотек. Если главная задача рабочего места -- запускать `xterm`, то достаточен даже очень старый диспетчер окон `twm` (он есть в составе `X.Org`, но редко используется по причине некрасивого, "плоского" оформления интерфейса). Вместо него можно использовать диспетчеры, подобные `IceWM` или `OpenBox`, обладающие большими возможностями, и при этом нетребовательными к ресурсам. Диспетчеры в среде `XFCE`, `WindowMaker` или `fvwm`, предоставляют множество возможностей, дополнительных модулей и мелких удобств. Они гибко настраиваются, не "вытягивая" за собой ресурсоёмких программных структур, используемых в `KDE` или `Gnome`.

Поразмыслив, Мефодий решил остановиться на `WindowMaker`: система меню показалась ему приятной для глаз, а способ быстрого запуска -- удобным. Кроме того, его позабавили "активные иконки", шевелящиеся в момент запуска соответствующего приложения.

(1) Такие системы -- не теоретическая выдумка, они вполне реальны и многочисленны. Например, графический интерфейс совершенно ни к чему на сервере, который занимается только маршрутизацией пакетов в сети.

(2) Каким пользователям разрешено запускать и останавливать графическую систему -- зависит от **профиля** системы.

(3) В некоторых вариантах **X Window System** экран по умолчанию раскрашивается в чёрно-белую крапинку.

(4) Эта функция не будет работать, если в конфигурационном файле X-сервера включён параметр `"DontVTSwitch"`.

(5) Кроме этого файла в домашнем каталоге пользователя можно обнаружить файл `.Xresources`, очень похожий по функции и аналогичный по синтаксису. Разница между этими файлами в использовании: `.Xresources` загружается только в процессе исполнения `xinitrc` при помощи утилиты `xrdb`, а `.Xdefaults` в дополнение к этому читается автоматически средствами `libX11`.

(6) Не путать с **диспетчером экранов**, который занимается совсем другим: подменяет утилиту `login`.

## 17. Прикладные программы

Основная особенность программного обеспечения Linux -- многообразие продуктов, решающих сходные задачи, особенно если дело касается области, в которой существует *несколько* подходов к их решению. Открытая модель разработки программ, описанная в лекции [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#), позволяет любому выбрать самый подходящий для него инструмент и развивать именно его. Поэтому список проектов, так или иначе связанных с Linux, насчитывает десятки (или даже сотни) тысяч наименований.

Конечно же, работа с самой операционной системой не может быть самоцелью. Все усилия Мефодия по изучению операционной системы Linux и основных утилит, нужны были для того, чтобы он впоследствии мог наилучшим образом решать в этой операционной системе любые из своих прикладных задач, разрешимых при помощи компьютера. Как уже уяснил Мефодий, для очень многих задач достаточно стандартных инструментов Linux и текстового редактора, однако есть случаи, в которых всё-таки необходима специальная прикладная программа, именно для этого предназначенная, или в которых специальная программа удобнее комбинации стандартных утилит.

Эта лекция посвящена краткому обзору **прикладных программ** для Linux, специально предназначенных для решения самых разных пользовательских задач. Материал, вошедший в эту лекцию нужно воспринимать только как пример, демонстрацию того, что и как можно делать в Linux, но вовсе не исчерпывающий список. В отличие от основных принципов устройства системы или стандартных утилит, обсуждавшихся в предыдущих лекциях, которые не изменяются (почти) в течение десятилетий, прикладное программное обеспечение -- это область, где всё меняется очень быстро. Технологии, сегодня считающиеся самыми передовыми, уже через несколько месяцев могут устареть. Вместе с ними могут устареть использующие их программы, а другие программы могут, наоборот, перейти в разряд наиболее современных и развитых. Поэтому перечисленные в этой лекции прикладные программы -- это не безусловная рекомендация, а довольно случайная выборка, отражающая текущее состояние дел в разработке приложений для Linux. Самый лучший способ найти и выбрать самые подходящие прикладные программы для своих задач -- посоветоваться с людьми, которые решают подобные задачи в Linux в настоящее время -- и попробовать.

Нужно отдавать себе отчёт в том, что прикладные программы для Linux не являются частью самой Linux, поэтому любой из названных ниже программ может не оказаться в каком-то из конкретных **дистрибутивов** Linux. Но почти наверняка в любом дистрибутиве найдётся не меньше одной или нескольких программ для решения каждой из перечисленных ниже прикладных задач. Чтобы не загромождать изложение, мы остановимся лишь на самых распространённых программных продуктах, входящих во многие дистрибутивы Linux.

## Рабочий стол

Первое, что стоит сделать, начав постоянно использовать Linux -- организовать для себя удобное "рабочее место": подобрать и настроить программы, с которыми приходится работать каждый день. Рабочее место в Linux может выглядеть очень по-разному. Можно вообще обойтись без графического интерфейса, используя только текстовый терминал для управления системой. Такой выбор будет правильным, если рабочее место находится на сервере, подключённом к сети Internet, доступ к которому осуществляется только при помощи `ssh` или аналогичных клиентов удалённого доступа. Впрочем, некоторые пользователи предпочитают работать в текстовом интерфейсе по эстетическим, а не прагматическим соображениям.

Если графический интерфейс используется, то и в этом случае есть огромный выбор, как его организовать, прежде всего, нужно решить, нужно ли организовывать "**рабочий стол**" (для этого подходят GNOME, KDE, XFCE) или можно обойтись возможностями одного из развитых диспетчеров окон (Enlightenment, FVWM2, [WindowMaker](#) и многие другие). Помимо функциональности, в выборе графической среды решающее значение могут сыграть и эстетические критерии. Дальше всех в этом направлении продвинулась среда Enlightenment, работа с которой в некоторых вариантах настройки количеством украшений и эффектов напоминает участие в компьютерной игре (скорее всего, сетевой).

## Эмулятор терминала

Даже для такой на первый взгляд тривиальной функции, как эмуляция терминала для **X Window System**, существует целый круг программ. Самая стандартная из них поставляется вместе с X.Org -- `xterm`. Вариант `xterm`, поддерживающий отображение шрифтов в кодировке UNICODE, вызывается командой `uxterm`. Однако каждое приложение, организующее среду рабочего стола, включает собственный **эмулятор терминала**, внешний вид и поведение которого настраивается централизованно вместе со всеми остальными приложениями рабочего стола. Есть и другие эмуляторы терминала, не связанные с конкретным рабочим столом, к таким относится `9term`, повторяющий возможности "окна" системы Plan9, `rxvt` -- очень нетребовательный к ресурсам эмулятор терминала, или его потомки, наподобие `aterm`.

Большое количество терминальных окон на рабочем столе может образоваться даже если раскладывать их по разным виртуальным экранам. Некоторые версии `xterm` (например, `konsole`), позволяют открывать окна "стопками", переключаясь между ними с помощью "закладок", как в записной книжке. Если основная работа происходит на удалённом компьютере, и каждый `xterm` соответствует сеансу удалённой работы, можно пойти и другим путём. Устанавливается *одно* соединение с удалённым компьютером, а *там* запускается эмулятор терминала `screen`. Эта программа не взаимодействует с графической средой. Она просто открывает требуемое число **псевдотерминалов**, и в каждом из них запускает по командному интерпретатору. Ввод со стандартного ввода попадает на тот псевдотерминал ("экран"), который `screen` считает "видимым", а вывод происходит на всех

экранах независимо. С помощью управляющего символа "^A" этой утилите подаются команды -- переключиться на следующий экран ("^A^N", при этом пользователь увидит то, что на этот экран выводилось), открыть новый ("^A^C") и т. п.

Типичное применение `screen` -- в одном окне запущен текстовый редактор, в другом -- командная строка. Удобство дополняется тем, что от работающего `screen` можно "отсоединиться" ("^A^D"), при этом пользователь может прервать терминальный сеанс и пойти спать, а интерпретатор командной строки и редактор будут работать на удалённом компьютере, как ни в чём не бывало (но, конечно, не будут проявлять никакой активности). Когда пользователь вернётся, он вновь присоединится к удалённой машине и "подключится" к работающему `screen` (команда `screen -R`), и продолжит работать с редактором и командной строкой.

## Диспетчеры файлов

Все изменения в файловой системе Мефодий привык производить с помощью стандартных утилит командной строки и находит это весьма удобным. Однако многие пользователи привыкли оперировать с файлами и каталогами как с наглядными штучными объектами (папками и документами), они могут выбрать для себя программу, которая позволяет наглядно и поштучно работать с объектами файловой системы -- **диспетчер файлов** (file manager). Поскольку представление файлов и каталогов как папок и документов нужно в первую очередь в рамках метафоры рабочего стола, то и диспетчеры файлов для Linux разрабатываются в первую очередь как приложения той или иной среды рабочего стола. В частности, и в KDE, и в GNOME есть свои диспетчеры файлов -- `konqueror` и `nautilus` соответственно, которые по совместительству служат `www`-браузерами. Такое совмещение функций вполне логично, поскольку в среде рабочего стола нужно представлять доступные локальные и удалённые ресурсы как единое пространство, наполненное объектами, которыми можно манипулировать, можно "открывать", т. е. запускать соответствующее приложение для просмотра и/или редактирования.

Для многих пользователей наиболее удобный способ работы с файловой системой -- "классический" двухпанельный диспетчер файлов, работающий в текстовом режиме (в терминале) -- `Midnight Commander` (название утилиты -- `mc`)(1). Его функциональность также шире просто операций с файлами -- он позволяет открывать файлы для просмотра и редактирования, вызывать вспомогательные программы для работы с архивами (и даже "заходить" в архивы, как в каталоги), передавать данные по сети и т. п. `Midnight Commander` имеет также неплохой встроенный текстовый редактор, опять-таки "классического" стиля.

Восторг, охвативший Мефодия при виде старых добрых синих панелек, довольно быстро угас. Далеко не всё, что нужно делать в Linux, в среде `mc` так же удобно, как и в полноценной командной строке. Кроме того, при работе с *графическими* файлами сильно не хватает представления этих файлов в виде миниатюр (thumbnails), чтобы выбирать среди них по содержанию, а не только по имени. Такими возможностями обладают многочисленные графические диспетчеры файлов; помимо тех, что включены в среды KDE и GNOME, есть множество независимых: `dfm` (похожий на диспетчер файлов OS/2), `emelfm2`, `EZFM` и X

Northern Captain (двухпанельные, причём автор последнего -- наш человек из Дубны), gentoo и worker (двухпанельные, в стиле диспетчера файлов [DirectoryOpus](#) из AmigaOS), FSV и XCruiser (трёхмерные! причём последний похож скорее на космический симулятор). Среди них встречаются и ориентированные специально на просмотр изображений, такие как GQView, endeavour, gview, qiv, xzgv и некоторые другие, -- с возможностями слайд-шоу, автоматического изменения размера, показа картинки на полный экран и т. п.

## Сеть

### WWW-браузеры

WWW-браузер -- программа для просмотра гипертекста, доступного через Internet -- на сегодня чуть ли не самое важное приложение для персонального компьютера. Сегодняшний **www-браузер** должен "уметь" гораздо больше, чем просто отображать страницы HTML и переходить по гиперссылкам. Фактически, на него ложится задача работы данными Internet во всём их многообразии, сюда входит и поддержка постоянно развивающихся стандартов, и обеспечение безопасности, и многое другое.

р В Linux есть довольно большой выбор www-браузеров, однако первым действительно современным свободным приложением для работы с Internet во всём его многообразии стал программный продукт Mozilla Suite. Наименование "Mozilla" вскоре отошло компании разработчику Mozilla Corporation и всей линейке программных продуктов, разрабатываемых в ней. Наследник Mozilla Suite -- Mozilla Seamonkey -- до сих пор вполне успешно конкурирует с аналогичными коммерческими программами, несмотря на затрудняющую успешное развитие идеологию "три в одном". Seamonkey -- пакет приложений для работы с Интернетом. Это мощный, насыщенный функциями коммуникационный центр для персонального компьютера. В состав пакета входит браузер, программа для работы с электронной почтой и редактор www-страниц. История Mozilla началась в 1998 году, когда фирма Netscape опубликовала исходные тексты своего браузера Netscape Navigator.

Одно из важных свойств программных продуктов Mozilla -- их принципиальная расширяемость. Программная часть расширений выполняется на языке Java Script, а интерфейсная -- на специально разработанном языке описания интерфейсов XUL. Такое расширение будет работать в любой операционной системе и на любой архитектуре компьютера, лишь бы версия "движка" Mozilla была не слишком старой.

Более современен Firefox, разрабатываемый командой Mozilla на основе исходных кодов, соответствующих только WWW-браузеру. Остальная часть [FireFox](#) написана полностью на XUL, поэтому разработка этой программы идёт существенно бодрее и проще, её настройка считается самой гибкой среди www-браузеров, а главное, любой желающий может написать на высокоуровневых языках программирования XUL/JavaScript и опубликовать свой модуль расширения (т. н. add-on). Ко времени выхода Firefox версии 3.0 таких расширений известно более *пяти тысяч*. По сути, Mozilla [FireFox](#) -- однопользовательская операционная система, со своим хранилищем ПО, правилами установки, политикой безопасности и т. п.



О `www`-броузерах, разработанных специально для той или иной среды рабочего стола уже шла речь выше (они, вдобавок, превосходно справляются с ролью файловых диспетчеров). Важная разновидность `www`-броузеров -- текстовые броузеры, т. е. те, которые могут быть запущены в любом текстовом терминале Linux. Самый старый и известный из них, один из прототипов современных `www`-броузеров -- Lynx. Он не имеет возможности отображать графическую информацию, но отлично поддерживает HTML, формы и таблицы. Современные версии поддерживают также соединения, защищённые при помощи SSL. Links -- это текстовый броузер, на первый взгляд очень похожий на Lynx, но все же несколько отличающийся от него:

- умением работать с таблицами и фреймами;
- отображением цветов, указанных в HTML-странице;
- использованием выпадающих меню (как в Midnight Commander);
- возможностью загрузки файлов в фоновом режиме.

Помимо возможности *просмотра* WWW-страниц часто выпадает необходимость их "скачивания", т. е. записи в файл. Это же относится и к ресурсам, доступным по протоколу FTP. Все описанные выше броузеры способны записывать HTTP- и FTP-ресурс в файл. Имеются несколько расширений для [FireFox](#), добавляющие очень ценные возможности (параллельное скачивание, скачивание по частям, расписание и т. п.). Для работы в командной строке, кроме старой и весьма простой утилиты `ftp`, имеются два её мощных "наследника": `lftp` и `wget`. Обе утилиты поддерживают как FTP, так и HTTP, причём `lftp` может работать, как и `ftp`, в режиме "оболочки". Если при получении файлов с какого-то сервера или группы серверов необходимо описывать множество исключений (чтобы не скачать лишнего), выполнять какие-то действия (например, заполнять формы или выполнять `java`-сценарии), можно воспользоваться более сложными программами `pvuk` или `httrack`.

## Почтовые программы

Подобно тому, как Firefox возрождает WWW-ипостась Mozilla, Thunderbird повторяет -- и расширяет -- почтовую составляющую Mozilla. Большинство сказанного о Thunderbird на сегодня справедливо и для [MozillaMail](#). Так же, как и в Firefox, в Thunderbird используется часть исходного кода Mozilla, которая работает с сетью (на этот раз -- с отсылкой почты и доступом к почтовым ящикам), а интерфейс и архитектура приложения в целом -- переделаны для того, чтобы избавиться от стародавних частей Netscape и облегчить дальнейшую разработку. Thunderbird (как и [MozillaMail](#)) обладает самым мощным на сегодняшний день встроенным антиспам-фильтром. Если непрошенная почта всё-таки попадает в ваш почтовый ящик, просто показывайте её Thunderbird со словами "это -- спам!". Через некоторое время программа сама научится отличать непрошеную почту от полезной. Как и Firefox, Thunderbird легко расширять собственными модулями, написанными на высокоуровневых языках и можно очень гибко настраивать.

Ещё один **почтовый клиент**, несколько уступающий Thunderbird по возможностям, но превосходящий его по быстрдействию, называется Sylpheed. Интерфейс этой программы

весьма похож на стандартную почтовую программу для Windows, Outlook Express, что может помочь избежать лишних хлопот при смене операционной системы. Автор этой программы, Хироюки Ямамото, человек аккуратный и пунктуальный, так что некоторый недостаток возможностей (эта программа умеет столько же, сколько и Outlook Express) компенсируется безотказной работой. Отпочковавшийся от Sylpheed проект Claws Mail имеет больше возможностей и снабжён собственной системой создания дополнений. Дополнения Claws позволяют гибко интегрировать его с другими утилитами системы (антивирусом, антиспам-фильтром и т. п.), встраивать в структуру рабочего стола и изменять внешний вид.

Поскольку управление электронной перепиской -- одна из задач рабочего стола, в каждой среде рабочего стола есть свой собственный почтовый клиент. Почтовый клиент для KDE называется KMail, он поддерживает как локальную доставку почты, так и множество почтовых протоколов (POP3, IMAP, SMTP). Почтовый клиент для GNOME называется Evolution. Эти программы интегрированы с другими приложениями, работающими с личными данными (т. н. PIM): календарём, адресной книгой и т. п.

Электронная переписка сама по себе не требует графического интерфейса, для чтения и написания электронных писем вполне достаточно возможностей терминала и текстового редактора. Среди текстовых почтовых клиентов для Linux наиболее известны Mutt и Pine, оба очень функциональны, поддерживают множество протоколов и форматов почтовых ящиков, хорошо настраиваются. Требовательным пользователям, которые хотят иметь возможность изменять внешний вид и способ работы почтового клиента, дополнять его сценариями и получать от почтовых служб всё, что те могут дать, рекомендуется Mutt. Тем же, кому главное -- просто получать, читать и отправлять почту (со всеми полагающимися удобствами), стоит начать с Pine. Любители Emacs используют встроенный в него модуль GNUS, весьма богатый функциями.

## Обмен сообщениями

Если компьютер подключён к Internet постоянно, бывает удобно пользоваться службами, передающими сообщения в реальном времени (instant messaging service). Таких служб довольно много, самая популярная из них -- ICQ. Множественность объясняется тем, что в большинстве случаев этот сервис предоставляется *централизованно*, какой-нибудь крупной корпорацией. Во многих случаях *серверы* этих служб не доступны под свободной лицензией. Исключение в ряду "собственников" -- служба Jabber, основанная на полностью открытом протоколе XMPP. Jabber позволяет любому сообществу создавать собственные сервера, управляемые собственными администраторами. Сам Jabber-сервер имеет возможность соединять своих клиентов не только с другими Jabber-серверами, но и со службами ICQ, MSN, Yahoo и AIM. В Linux есть несколько клиентских программ для обмена мгновенными сообщениями. Особняком стоят клиенты IRC (**I**nternet **R**elay **C**hat), службы с более долгой историей и сложным протоколом (имеется в виду и сетевой протокол, и протокол работы пользователя в IRC).

Psi -- удобный графический клиент сети быстрого обмена сообщениями Jabber (а значит, по всем протоколам, которые поддерживает выбранный Jabber-сервер). Psi

поддерживает такие возможности Jabber, как одновременная работа с несколькими серверами, конференции, криптозащиту передаваемой информации (через SSL и GnuPG), работу через HTTP (S) прокси-сервер и т. д. SIM -- многопротокольный клиент обмена мгновенными сообщениями. Поддерживаются протоколы ICQ, Jabber, MSN, AIM, YIM, а также [LiveJournal](#). Кроме того, имеется множество модулей, реализующих дополнительные возможности. Есть вариант SIM, ориентированный на среду KDE. "Прицельно" на среду KDE ориентирован и другой мощный клиент, имеющий поддержку также и IRC, -- Kopete. На среду Gnome ориентирован Gaim -- наиболее мощный и наиболее гибко настраиваемый клиент. Имеет модули доступа почти ко всем мыслимым протоколам, позволяет писать сценарии на Perl и TCL. Для IRC есть и специальные клиенты: [ChatZilla](#) (как можно догадаться из названия, он "встроен" в Mozilla, но доступен и как дополнение к Firefox) или X-Chat -- весьма мощная программа, ориентированная на "хитрости" IRC.

---

**Предупреждение!** Обмен информацией и бессмысленными текстами при помощи любой из перечисленных служб, а также телефона, *не заменяет* человеческого общения! Помните, что компьютер передаёт только данные (например, такие: : ) ), но не эмоции.

---

Не обойдён стороной и интерфейс текстовой консоли: CenterICQ, поддерживающий несколько протоколов (среди них Jabber и IRC); licq, обладающий как текстовым, так и графическим интерфейсами (следовательно, им можно пользоваться и находясь за рабочей станцией, и дистанционно); irssi, нацеленный на службы типа IRC (на сегодняшний день поддерживаются IRC, SILC и ICB), и т. д.

## Офисные программы

Важной частью современной рабочей станции являются так называемые офисные средства обработки информации. Под **офисными приложениями** обычно понимают стандартный набор из словарного процессора, средства работы с электронными таблицами, средства создания презентаций, средства для работы с базами данных. Все перечисленные офисные приложения входят в пакет [OpenOffice.org](#) -- свободный набор офисных программ, не уступающий по возможностям несвободному Microsoft Office, а кое в чём даже превосходящий его. Например, частности, которая может иметь очень важное значение: компонент [OpenOffice.org](#) Writer позволяет экспортировать документы непосредственно в формат PDF. Интерфейс [OpenOffice.org](#) устроен принципиально так же, как и у аналогичных продуктов Microsoft, так что пользователю, привыкшему к Microsoft Office, не составит большого труда перейти к работе в [OpenOffice.org](#). Кроме того, [OpenOffice.org](#) позволяет работать со всеми форматами файлов Microsoft Office.

История [OpenOffice.org](#) напоминает историю Mozilla: сначала проект (под именем [StarOffice](#)) развивался закрыто, без доступа мирового программистского сообщества к

исходным текстам. Однако в 2000-м году компания Sun Microsystems открыла исходные тексты программного продукта, образовав [OpenOffice.org](http://OpenOffice.org). Так же, как и в случае Netscape/Mozilla, пара [StarOffice/OpenOffice.org](http://StarOffice/OpenOffice.org) использует двойное лицензирование, дающее право как свободного доступа к исходным текстам, так и использования их в закрытых коммерческих продуктах.

По возможностям [OpenOffice.org](http://OpenOffice.org) остаётся самым развитым и полным офисным пакетом для Linux, однако есть и другие офисные средства. В частности, офисный пакет Koffice, ориентированный на среду KDE, в котором есть примерно тот же набор офисных приложений, что и в [OpenOffice.org](http://OpenOffice.org). Кроме того, есть отдельные офисные приложения, не составляющие пакетов -- словарный процессор Abiword и электронные таблицы GNUmeric.

## Графика

Чем проще пользовательская задача, тем больше программ под Linux её решают. В частности, манипуляция геометрическими фигурами с возможностью изменения их параметров (цвета, размера и т. п.), хранением набора фигур в файле и преобразованием получившегося изображения в растровый формат -- довольно простая задача, требующая аккуратной реализации основных функций какой-нибудь высокоуровневой библиотеки (или двух -- интерфейсной и графической). Неудивительно, что редакторы с подобными возможностями есть и для каждого рабочего стола, и независимо от них. Это утверждение относится и к ещё более простым программам работы с растровой графикой. Ниже описаны только существенно более сложные программы.

## Векторная графика

**Векторной графикой** называется способ работы с изображениями, при котором оно представлено в виде фигур, каждая из которых имеет собственное описание (тип, размеры, кривизну или иные параметры составных частей, их цвета, способ представления и т. п.). Некоторые графические устройства (например, распознающие формат [PostScript](http://PostScript)) умеют сами интерпретировать описания фигур, для других необходимо заранее просчитать и сформировать картинку программным путём.

### Работа с PostScript и PDF

Современная полиграфия уже не мыслится в отрыве от компьютеров, все допечатные материалы обычно существуют в электронной форме, и именно электронные документы подаются на печатающие устройства для вывода. Причём для современной полиграфии de facto стандартом является формат [PostScript](http://PostScript). [PostScript](http://PostScript) -- это язык описания страницы, позволяющий представить любые полиграфические материалы в векторном формате (однако он допускает и включение растровых фрагментов). Файл в формате [PostScript](http://PostScript) фактически представляет собой программу, описывающую, какие действия нужно произвести, чтобы получить требуемый вывод. Профессиональные печатающие устройства умеют непосредственно интерпретировать документы на языке [PostScript](http://PostScript).

PDF (**P**ortable **D**ocument **F**ormat, переносимый формат документов) -- модификация языка [PostScript](http://PostScript), разработанная для того, чтобы обмениваться полиграфическими

документами через Internet. В PDF есть специальные возможности для публикации документов в Сети, в частности, поддержка гиперссылок, а некоторые возможности языка [PostScript](#) оттуда, наоборот, исключены.

Ghostscript -- интерпретатор языка описания страниц [PostScript](#) и файлов в формате PDF (формат переносимых документов). Ghostscript преобразует [PostScript](#) во многие растровые форматы, подходящие для вывода данных на экран или на принтер, не поддерживающий [PostScript](#). Обычно Ghostscript используется для просмотра файлов [PostScript](#) и для печати на принтерах, не поддерживающих язык [PostScript](#), [GhostScript](#) используется множеством приложений для вывода данных на печать. Графический интерфейс для [GhostScript](#) предоставляет программа [GhostView](#) (команда `gv`), она позволяет отображать документы в форматах [PostScript](#) и PDF в графической среде **X Window System**. Для различных манипуляций с файлами в формате [PostScript](#) предназначен пакет утилит командной строки `psutils`, с их помощью можно выбрать, переупорядочивать, масштабировать страницы в [PostScript](#)-файлах, изменять параметры текста и делать многое другое.

Специально для просмотра PDF-файлов предназначена программа `xpdf`, она позволяет переходить по гиперссылкам в документе, просматривать структуру документа, производить поиск и поддерживает сглаживание шрифтов. Для KDE и GNOME имеются свои аналоги: `kpdf` и `evince`. Альтернативой им может быть Acroread -- версия известного приложения Adobe Acrobat Reader для Linux, которое, однако, является несвободным программным продуктом.

## Диаграммы

Отдельно стоит упомянуть редакторы диаграмм, которые часто смешивают с обычными редакторами векторной (плакатной) графики. Между тем задачи у них разные: если для плакатной графики главное -- построение "картинки", соответствующей задумке автора по *внешнему виду*, то в диаграмме автора более беспокоит *логическое* соответствие изображения проекту и его *наглядность*. Поэтому при построении диаграммы много внимания уделяется "стрелочкам" и прочим соединительным линиям, оптимальному размещению объектов на странице, типизации объектов и т. п.

Самая старая из подобных утилит, `xfig`, и по сей день активно используется, формат её диаграмм распознают многие средства работы с векторной графикой. Более мощной является ориентированная на среду Gnome утилита Dia, возможности которой продолжают расти (есть даже проект перевода диаграмм в нотации языка моделирования программных продуктов UML непосредственно в текст на C++). Аналогом Dia для KDE является встроенная в пакет KOffice утилита Kivio.

Отдельно стоит отметить [OpenOffice.org Draw](#) -- программу, сочетающую в себе возможности построения диаграмм, плакатную и т. н. бизнес-графику. Пакет [OpenOffice.org](#), в отличие от своего несвободного аналога, не провоцирует пользователя создавать *все* виды документов (графики, схемы, формулы, раздаточный материал к докладам и прочее) в одном только текстовом процессоре.

## Плакатная графика

Что же касается собственно векторной (плакатной) графики, то и здесь есть из чего выбирать. Например, Inkscape -- программа векторного рисования общего назначения, реализующая стандарт W3C SVG. Применяется новейшее программное ядро для отображения, со сглаживанием, альфа-каналом и векторными шрифтами. Ещё мощнее -- Xara Xtreme, продукт, также сменивший лицензию на свободную; помимо прочего, он умеет импортировать файлы в некоторых несвободных форматах. Подаёт надежды также и проект SK1, под эгидой которого, в частности, разрабатывает универсальный преобразователь векторных файлов в свободных и несвободных форматах uniconvertor.

## Растровая графика

**Растровая графика** означает работу с изображением, представленным в виде матрицы точек ("пикселей"). Это значит, что при сильном увеличении границы любого объекта будут выглядеть "лесенкой" из точек (в отличие от векторного представления, где увеличение повышает качество изображения). С другой стороны, растр -- удобный для компьютерной обработки формат представления фотографий, сделанных от руки рисунков и прочих изображений, которые нельзя расчлнить на отдельные фигуры.

В GNU/Linux есть развитые средства для редактирования растровой графики. Самым мощным из них является GIMP (**GNU Image Manipulation Program**). С её помощью пользователь сможет редактировать изображения, создавать логотипы и другие графические элементы особенно полезные при создании Web-страниц. GIMP включает много инструментов и фильтров, аналогичных тем, которые можно найти в коммерческих графических редакторах, а также несколько возможностей, эксклюзивных для этой программы. GIMP предоставляет возможность работать с цветовыми каналами, уровнями изображения, накладывать эффекты, сглаживать шрифты и конвертировать изображения в разные форматы. В GIMP имеется собственный язык программирования сценариев (на основе Scheme), на котором можно создавать довольно замысловатые дополнения к основной программе.

Очень полезен набор утилит для обработки графики из командной строки -- [ImageMagick](#). В этот набор входят утилиты для отображения (`display`), преобразования (`convert`) изображений, захвата изображений с экрана (`import`) и даже собственный интерпретируемый язык программирования, Magick Scripting Language. Для полуавтоматического перевода из растрового представления в векторное существует несколько специальных утилит, например, `autotrace/autofig` или `potrace`.

## Трёхмерная графика

Для Linux создано несколько программных пакетов, работающих с пространственным представлением объектов.

Исходные тексты одного из самых мощных пакетов трёхмерного моделирования, пересчёта (рендеринга) и анимации -- Blender -- в 2002 году были открыты и весь проект полностью переведён под свободную лицензию. Авторы Blender пришли к выводу, что

открытая разработка *инструмента* более эффективна и прибыльна для тех, кто этим инструментом (а не его продажей) зарабатывает. Для этого пришлось выкупить находящиеся в собственности спонсоров части проекта у хозяев за сумму сто тысяч евро. Искомую сумму предоставило сообщество пользователей Blender, уже тогда немалое: каждый внёс сколько смог, и менее, чем за два месяца, денег на счёте оказалось достаточно. С тех пор круг пользователей и возможности Blender продолжают постоянно расти.

Для выполнения задач, совмещаемых Blender, есть и отдельные программные средства. Например, популярный пакет трассировки лучей (трёхмерного проектирования и сценографии) POV-Ray, с помощью которого создаются проекты удивительной сложности и красоты (например, перевод картины Уильяма Марлоу "Каприччо" в трёхмерное представление -- с тем только, чтобы из определённой точки *повторить* её). Многие графические редакторы имеют встроенные средства анимации, а иные (как, например, [CinePaint](#), называвшийся ранее [FilmGimp](#)) специально разрабатываются для мультипликаторов.

Не стоит забывать, что популярный нынче стандарт OpenGL -- открытый; он разрабатывался для UNIX-подобных систем и используется большим числом программ для Linux (в том числе и Blender). К сожалению, производители *аппаратного* обеспечения (видеокарт), как правило, скрывают не только устройство своих карт, но даже и способ их низкоуровневого использования. Поэтому в открытом доступе оказываются лишь готовые драйверы (без исходных текстов) к *некоторым* версиям ядра Linux и определённым сборкам XFree86. Отображение трёхмерных объектов с пересчётом на программном уровне пока работает существенно медленнее, хотя ничуть не хуже, поэтому используя OpenGL для игр и прочих программ, требующих действительно быстрой работы графической подсистемы, нужно всегда помнить о необходимости получить -- возможно, несвободный -- драйвер.

## Мультимедиа

### Музыкальные шкатулки

Программ-проигрывателей звуковых файлов в Linux не перечислить. Очевидный лидер по популярности среди них -- XMMS (**X Multi Media System**) -- стремительно сдаёт позиции, сказывается слишком жёсткая привязка к устаревшей интерфейсной библиотеке. У XMMS много "наследников" -- XMMS2, BMPx, Audacious. Помимо основной функции -- играть музыку (поддерживается множество форматов) -- в каждой из них реализовано немало звуковых и визуальных эффектов благодаря большому количеству расширений. Интерфейс этих программ аналогичен интерфейсу не менее популярного в системах Windows приложения WinAMP (как правило, они умеют использовать "шкурки" WinAMP2). Обычно каждая среда рабочего стола реализует собственный проигрыватель звуковых файлов, хотя бы для того, чтобы воспроизводить собственные звуковые эффекты, связанные с различными системными событиями, однако с их помощью прослушивать файлы может и пользователь.

С развитием музыкального вещания в сети Интернет и повсеместным распространением цифровых проигрывателей, интерес пользователя начинает смещаться в

сторону т. н. "ведения музыкальных коллекций": недостаточно просто проигрывать музыку, надо уметь каталогизировать имеющиеся записи и находить нужную среди них, осуществлять поиск по сотням интернет-радиостанций и т. п. Грамотным примером такого подхода может служить amaroK -- интегрированный менеджер музыкальных коллекций в KDE.

Очевидно, что для прослушивания звука совсем не обязательно использовать графический интерфейс, поэтому в Linux есть большое количество терминальных утилит для воспроизведения звука. Некоторые из них, например, mpg123, mpg321, ogg123 или splay, предназначены для проигрывания оцифрованного звука (возможно, в сжатых форматах), другие, такие как lazy или cd-console, управляют музыкальными лазерными дисками, есть утилиты, играющие музыку в нотном (midi) и других форматах -- timidity (она отличается тем, что преобразует ноты, записанные для инструментов в оцифрованное звучание этих инструментов, а значит, не требует MIDI-устройства), mikmod (распознаёт множество форматов: MOD, STM, S3M, XM и т.д.), sidplay и прочие. Чтобы пользователь не запутался, специальные оболочки, например mpfc или cplay, предоставляют общий интерфейс ко всем консольным проигрывателям.

## Музыкальные редакторы

Часть профессиональных музыкантов предпочитает использовать для работы со звуком дорогие специализированные музыкальные компьютеры: в этом повинна и реклама, и низкое, с точки зрения профессионала, качество звука большинства звуковых карт в компьютерах общего назначения. Несмотря на это и для таких компьютеров существует немало программ, работающих со звуком на профессиональном уровне. Такие программы можно разделить на две категории: нотные редакторы, задача которых -- создание, редактирование, запись и нотное представление *музыкальных композиций*, и звуковые редакторы для собственно звука, а также преобразования его, наложения эффектов и т. п.

### Нотные редакторы

В операционных системах, основанных на GNU/Linux, также присутствуют мощные программы для редактирования музыки и звука. Пожалуй, самым известным из них является Rosegarden. Программа изначально разрабатывалась для профессиональных **мультимедиа-**станций от Silicon Graphics и работала на операционной системе IRIX, потом она была перенесена на Linux, а исходные тексты программы были открыты. Сегодня Rosegarden представляет из себя развитый MIDI- и аудиосеквенцер, нотный редактор, а также редактор общего назначения для сочинения и редактирования музыки. Он прост в изучении и идеально подходит для композиторов, музыкантов или студентов музыкальных специальностей, работающих в маленькой студии или записывающихся дома.

Noteedit -- нотный редактор (редактор партитур), основанный на Midi-библиотеке TSE3. Он может писать и читать Midi файлы и сигналы от внешней Midi-клавиатуры. Системные Midi-устройства используются для воспроизведения нотной записи. Имеется возможность сохранить партитуры в формате MusiXTeX или Lilypond для последующего вывода на печать.



MusE -- это MIDI-секвенсер в стиле Cubase/Logic Audio, поддерживающий ввод MIDI-событий с клавиатуры и последующее их редактирование в нотном редакторе, матричном редакторе, редакторе списка событий и редакторе ударных инструментов.

## Редакторы и фильтры оцифрованного звука

Популярный свободный редактор звука -- это Audacity. Он умеет записывать звук сразу в форматы WAV, AIFF, AU, IRCAM или MP3. В нём есть всевозможные инструменты для редактирования записанного звука, в том числе встроенный редактор амплитуды, настраиваемый режим отображения спектрограммы и средства частотного анализа звуковых дорожек. Встроенные простейшие аудиоэффекты включают усиление баса, [WahWah](#), удаление шума и т.д. Audacity поддерживает модульные дополнения, в которых обычно поставляются более сложные аудиоэффекты. В список поддерживаемых форматов модулей входят VST, LADSPA и Nyquist.

Sweep -- это многоканальный звуковой редактор, в котором реализованы все основные операции, такие как удаление, копирование, вставка и применение эффектов, оформленных в виде плагинов, к любой части звукового файла. Примерно теми же возможностями обладают и другие редакторы звука -- ReZound, [WaveSurfer](#) и GNUSound.

Как и в случае с другими мультимедиа-форматами, в Linux существуют терминальные утилиты для обработки звука, не требующие графического интерфейса. Основной пакет терминальных утилит для работы со звуком называется SOX, в него входят утилиты для преобразования, записи и проигрывания звуковых файлов, поддерживается множество форматов.

При помощи консольных утилит можно также сжимать звуковые файлы в различные форматы с потерей качества. Содержимое файла, сжатого "с потерей качества", может быть неотличима на слух от содержимого исходного файла: алгоритмы преобразования учитывают человеческую физиологию, например, формат MP3 не воспроизводит слишком высоких звуков, а слишком низкие не разделяет на два канала. Смысл термина "потеря качества" -- в том, что из упакованного файла *исходный* восстановить уже нельзя. Сжатие с потерей качества можно настраивать на определённую мощность потока упакованных данных: чем больше данных можно передавать в единицу времени, тем чище звук, поэтому такие форматы подходят для передачи по сети (например, интернет-радио).

Основные форматы с потерей качества -- это MP3 (с ним работают упаковщики lame/toolame, bladeenc) и OGG Vorbis (утилита oggenc). Эти форматы (особенно OGG) хорошо подходят для упаковки качественной музыки. Файлы в формате OGG, упакованные семикратно (192 kbp/s), почти неотличимы на слух от исходных. Если необходимо сжать звук с ощутимой потерей качества (но без потери членораздельности и раз в двадцать), можно использовать другие форматы -- gsm, aiff, adpcm, speex (сжатие речи) или bonk. Последний формат -- нестандартный, он поддерживается одноимённой утилитой и отличается большой гибкостью, так как может работать и в режиме "сжатие без потерь". Для сжатия без потерь разработан специальный формат -- FLAC, его распознают многие программы (в том числе и утилиты от авторов этого формата -- flac и metaflac) и даже аудиоустройства.

## Видеопроигрыватели

Весьма мощным видеопроигрывателем для Linux является запускаемая из командной строки программ `mplayer`. Проект `MPlayer` "носит с собой" несколько библиотек, позволяющих использовать для декодирования видео несвободные кодеки, в том числе оформленные как динамические библиотеки Windows. Кроме того, `mplayer`-ом можно управлять из специально созданных графических оболочек, таких, как `smplayer`. Очень неплох видеопроигрыватель `Kaffeine`, встроенный в KDE. Он пользуется ещё одним видео-"движком", `Xine`, у которого есть также и собственный графический интерфейс. Активно развивается `VLC (Video Lan Client)`, изначально предназначавшийся для просмотра сетевого потокового вещания. `VLC` интересен ещё и тем, что может выступать "ретранслятором" видеопотока (направляя его нескольким пользователям), и имеет, в числе прочих, HTTP-интерфейс, что позволяет управлять им с удалённого компьютера посредством браузера. Все эти программы поддерживают как свободные, так и множество несвободных видео- и аудиоформатов, таких, как OGG, MPEG, DivX/XviD, DVD, формат контейнера VMW и др.

Существует даже программный продукт `MythTV`, превращающий компьютер в полноценный мультимедийный центр, с телевидением, радио, проигрыванием и показом дисков и управлением с помощью дистанционного пульта. Что касается аппаратных видеоустройств (вроде платы видеозахвата или камеры), на низком уровне с ними взаимодействует комплект программ [Video4Linux \(v4l\)](#), предоставляя проигрывателям уровня `vlc` унифицированный программный интерфейс.

## Видеоредакторы и конвертеры

В Linux есть выбор средств для преобразования и обработки видео.

`GStreamer` представляет собой библиотеку для обработки медиапотоков, основанное на идее объединённых в графы фильтров, обрабатывающих медиаданные. Приложения, использующие эту библиотеку, смогут производить любую обработку медиаданных от обработки звука до проигрывания видео. Модульная архитектура позволяет реализовать поддержку любого нового формата данных, просто установив соответствующее расширение.

`Kino` -- это нелинейный редактор цифрового видео (DV) для GNU/Linux. Он хорошо интегрирован с IEEE 1394 и позволяет захватывать изображение, управлять VTR, и записывать на камеру. Этот редактор записывает видео на диск в формате AVI в кодировках `type-1 DV` и `type-2 DV`.

Другими примерами редакторов цифрового видео являются `Cinelerra` (полнофункциональная система нелинейного видеомонтажа и аудиопроектирования) и [PiTiVi](#) (видеоредактор попроще).

Существуют терминальные утилиты для обработки видеопотока, например, пакет `transcode`. Кодирование и декодирование видеопотока осуществляется с помощью загружаемых модулей. Также поддерживается загрузка внешних фильтров. В число модулей входят: модули импортирования из DVD, элементарных MPEG (ES) и программных потоков

(VOB), видео в формате MPEG, цифрового видео (DV), потоков YUV4MPEG, поддержка формата файлов [NuppelVideo](#) и необработанных потоков видео; модули для записи DivX;-), OpenDivX, DivX 4.xx или несжатых файлов AVI с MPEG, звука в форматах AC3 или PCM; дополнительные модули для записи отдельных кадров (PPM) или потоков YUV4MPEG. Пакет transcode содержит набор утилит для демультимплексирования (tcdemux), выделения (tctract) и декодирования (tcdecode) видеопотока, исследования (tcprobe) и сканирования (tcsscan) ваших файлов и пост-обработки файлов AVI, изменения заголовков файлов AVI (avifix), соединения нескольких файлов в один (avimerge) или разделения большого файла на несколько AVI-файлов меньшего размера (avisplit) для размещения на CD.

Ffmpeg -- это "сверхзвуковой" кодировщик/декодировщик видео и звука, работающий в режиме реального времени, а также потоковый сервер и преобразователь различных звуковых и видеоформатов. Ffmpeg умеет захватывать видеосигнал из источника [Video4Linux](#) и преобразовывать его в файлы различных форматов на основе компенсирующего кодирования DCT/motion. Звук при этом сжимается по алгоритму MPEG-2 или алгоритму, совместимому с AC3.

## Запись CD и DVD

Для записи дисков и сопровождающих запись задач в Linux есть как минимум два приложения с графическим интерфейсом: входящее в комплект приложений для KDE k3b и написанное на GTK xcdroast. Фактически, оба этих приложения -- это графические оболочки над терминальными утилитами для записи CD и DVD, в первую очередь cdrecord и cdrdao, которыми можно пользоваться и непосредственно из командной оболочки. cdrecord -- утилита для записи дисков с цифровыми данными, в нём реализована полная поддержка аудио-, смешанных, мультисессионных и CD+ дисков. cdrdao -- программа записи аудиодисков в одну сессию позволяет управлять областями в начале дорожек данных (длиной до 0, ненулевые аудиоданные) и, например, международными стандартными кодами записи. Все данные, которые будут записаны на диск, должны быть описаны в текстовом файле. Аудиоданные могут быть в форматах WAVE или raw.

Помимо того, для Linux есть множество программ, позволяющих производить обратную операцию: считывание данных с аудиодиска в файл, такие программы называются грабберами (grabber). Один из удобных грабберов с графическим интерфейсом -- Grip.

## Издательские системы

Подготовка печатных документов и оригинал-макетов изданий -- хоть и не очень распространённое, но важное приложение компьютера. В Linux самой известной и системой подготовки качественных документов, пригодных к печати в типографии, является TeX. TeX -- это фактически специализированный язык программирования, специально разработанный для описания типографского набора. Документ в TeX представляет собой текст, сопровождаемый командами, указывающими, какое форматирование следует произвести.

Возможности TeX очень широки, однако для того, чтобы их использовать в полной мере, требуются довольно серьёзные познания в нём. Чем шире познания -- тем легче, быстрее и удобнее готовить документы в [TeXe](#) и тем лучше их качество.

Обычно TeX используется совместно с пакетами форматирования более высокого уровня, например, LaTeX. LaTeX -- это набор написанных на языке TeX макропакетов, предоставляющих удобные средства для решения типичных задач оформления печатных изданий. В LaTeX определено оформление для нескольких стандартных классов документов.

LyX -- это современный подход к написанию документов, разрывающий с устаревшей парадигмой использования компьютеров как пишущих машинок, применяемой в большинстве других систем подготовки документов. Он разработан для тех, кто хочет получить профессиональное качество документа при печати, не тратя при этом много времени и усилий, и не становясь специалистом по полиграфическому оформлению. Основное новшество в LyX -- это WYSIWYM (What You See Is What You Mean -- Вы видите то, что вы имели в виду), которое означает, что автор сосредотачивается над своей работой, а не над деталями оформления документа. Это позволяет продуктивно работать, оставляя заключительное оформление специальному движку (такому как LaTeX), который специально разработан для подобных задач. С LyX автор может сконцентрироваться на содержании своей работы и позволить компьютеру взять большинство забот об оформлении на себя.

В Linux есть по крайней мере одна программа для визуальной подготовки оригинал-макетов, аналогичная **издательским системам** Adobe [PageMaker](#), QuarkXPress и подобным -- Scribus. Возможности его могут быть более ограничены, чем у перечисленных коммерческих аналогов с сегодняшнем их виде, однако он распространяется свободно и в настоящее время активно разрабатывается.

## Нельзя объять необъятного

В этот краткий и фрагментарный обзор не вошли собственно инструменты для разработки программного обеспечения, которые развиты в Linux чуть ли не лучше, чем все остальные приложения, поэтому написать краткий обзор для них гораздо сложнее. Не сказано ничего о серверах баз данных (не потому, что таких серверов нет!) и серверах приложений в составе сложных проектов. Обойдены вниманием и игры -- любой читатель этой лекции сможет самостоятельно решить, сколько внимания и каким из них уделять.

Напоследок повторим: главной целью приведённого обзора приложений для Linux было показать, что приложения есть и их много, нужно только достаточно внимательно искать -- и нужное обязательно найдётся. Мефодию для поиска доступен очень удобный инструмент, описанный в лекции [Управление пакетами](#), менеджер пакетов APT и команда `apt-cache search`. Поскольку в современные дистрибутивы Linux входят тысячи пакетов, почти наверняка среди них найдётся нужное приложение. Кроме того, любому пользователю Internet доступен поисковый сайт <http://google.com>, наиболее подходящий для поиска чего бы то ни было, а для поиска существующих приложений для Linux можно воспользоваться

специализированными сайтами -- <http://rpmfind.net>, <http://freshmeat.net>, <http://rpm.pbone.net> или сайтом, посвящённым выбранному дистрибутиву.

Обратите внимание, что все названные в этой лекции приложения -- это свободно распространяемые и разрабатываемые программы (см. лекцию [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)), если не оговаривается обратное. Характерная черта свободного программного обеспечения состоит в том, что если для решения какой-то задачи есть одно свободное приложение, то всегда есть и несколько других (о причинах этого см. лекцию [Политика свободного лицензирования. История Linux: от ядра к дистрибутивам](#)), так что пользователь всегда может выбрать себе приложение по вкусу, а если подходящего не обнаружится -- изменить для себя одно из уже существующих или даже написать новое. В конце концов, нет ничего дороже и милее сделанного собственными руками велосипеда.

---

(1) Пользователи, знакомые с MS-DOS, вспомнят Norton Commander, а пользователи помоложе -- Far Manager.

# 18. Политика свободного лицензирования. История Linux: от ядра к дистрибутивам

## История возникновения свободного ПО

### Разработка ПО как научное исследование

Особенность программного обеспечения состоит в том, что оно производится в одной форме -- в виде **исходного текста** (source code), а распространяется и используется в другой -- в виде двоичной программы, машинных кодов, по которым невозможно однозначно восстановить исходный текст. Чтобы изменять программу, исправлять ошибки или даже просто точно установить, что и как делает программа, необходимо располагать её исходным текстом.

Первоначально создание программного обеспечения для компьютеров было в первую очередь академическим занятием. Для специалистов в области компьютерной науки (computer science) каждая программа представляла собой результат научного исследования, в некотором смысле аналогичный публикации статьи. Это означает, что исходный текст программы был обязательно доступен всему научному сообществу, поскольку любой научный результат должен быть верифицируем, т. е. подтверждаться другими исследователями и быть открытым для критики. Таким образом, процесс разработки программного обеспечения более принципиально схож с научным процессом: учёный брал существующие программы, исправлял их в соответствии со своими идеями и публиковал исправленные программы -- новый результат.

Однако технология производства компьютеров развивалась не менее активно, чем программное обеспечение для них. В 1970-е годы существует огромное разнообразие различных архитектур вычислительных машин, различавшихся также и производительностью, и ценой. Естественно, для каждой архитектуры приходилось разрабатывать отдельный набор программного обеспечения. С середины 1970-х в большинстве американских университетов (где преимущественно и развивалась компьютерная наука) для академических разработок использовались компьютеры архитектуры PDP-10, что позволило сотрудникам разных университетов использовать разработки друг друга на своих машинах. Сотрудники лаборатории искусственного интеллекта массачусетского технологического института в конце 1970-х разработали для PDP-10 собственную операционную систему ITS (Incompatible Timesharing System, несовместимая система с разделением времени) и очень большой набор программ для неё. Исходные тексты написанных в МТИ программ были общедоступны, сотрудники других университетов пользовались их исходными текстами и присылали им исправления, всё

программное обеспечение в этих лабораториях было полностью академическим, а среди учёных-разработчиков был настоящий дух сотрудничества.

## ПО как "патентованный" продукт

В условиях огромного многообразия архитектур компьютеров программное обеспечение составляло неотъемлемую часть самой машины, причём далеко не самую дорогостоящую часть. Производители компьютеров поставляли их вместе с основным программным обеспечением, по крайней мере, с операционной системой. Производство компьютеров было наукоёмким, в основе своей коммерческим предприятием.

В ситуации, когда программное обеспечение является объектом продажи наравне с предметами обихода, на него автоматически распространяются уже не только законы научной разработки, но и свойства материальных предметов, которыми можно торговать, обмениваться, право владения и пользования которыми стоит охранять законодательно. Так, программное обеспечение попало в разряд интеллектуальной собственности: т. е. исходный текст программы стал рассматриваться как произведение, объект применения авторского права. Чтобы защитить свои интересы, производители компьютеров и программного обеспечения используют **лицензии** -- вид договора между обладателем авторских прав и пользователем (покупателем) программного обеспечения. Подобные договоры заключались и с университетами, например, университету передавались исходные тексты программ и право их изменять, но запрещалось распространять их за пределами университета. Подобные ограничения означали, что тексты соответствующих программ не могли открыто обсуждаться в сообществе, т. е. не существовали для научной разработки. Были у компьютеров и программного обеспечения покупатели и вне академической среды, например, банки. Таким пользователям не столь важно получить исходные тексты программ, они заинтересованы в программном обеспечении как в законченном продукте и готовы платить деньги за надёжные и удобные программы.

В европейской культуре столь долго вырабатывались правила собственности по отношению к материальным предметам, что распространение этих прав на предметы нематериальные -- программные продукты -- выглядит делом естественным и не вызывает сомнений. А поводов для сомнений немало. Главное отличие программного продукта от, допустим, табурета -- т. н. **безущербное копирование**. Если грабитель отбирает у крестьянина табурет, совершается злодеяние: крестьянин табурета лишается, терпит ущерб. Если крестьянин отдаёт кому-то табурет добровольно, он его также лишается, поэтому вправе требовать возмещения ущерба -- например, деньгами. Для того, чтобы ущерба у крестьянина не происходило, табурет нужно *воспроизвести*: добыть досок, позвать столяра, краснодеревщика и оплатить их работу, и один из двух получившихся предметов обихода отдать грабителю. В этом случае ущерб -- денежный -- терпит тот, кто оплачивает копирование табурета. Совершенно естественно при этом законодательно запрещать нанесение ущерба, то есть признавать право распоряжаться вещью только за одним человеком -- за её хозяином. Никаких дополнительных механических или юридических приспособлений, запрещающих воспроизводить табуреты, при этом не требуется.

Иное дело -- программный продукт. Сколько бы средств ни было вложено в его *разработку*, процедура его *копирования* (переписывания с одного носителя данных на другой) резко отличается от процедуры воспроизведения табурета. Она не требует участия *ни одного* из авторов программы, ни, по большому счёту, вообще человека. Единственная расходная статья при этом -- цена носителя данных и амортизация копируемого устройства. В результате такого копирования получается два экземпляра программы, создающие удобства уже для двух человек. Таким образом, если человек оценивает приносимые программой удобства выше стоимости носителя данных, копирование -- благо. Если же относиться к программному продукту, как материальной вещи, и закреплять право её использования за одним каким-то человеком, возникает множество неурядиц, каждую из которых приходится решать искусственными, а зачастую и противоестественными методами.

Например, придётся изыскивать, какой ущерб всё-таки наносится "хозяину" программы при её безущербном копировании. Обычно при этом фигурирует понятие "упущенная выгода", то есть та прибыль, которую хозяин мог бы получить, но не получил из-за того, что продукт скопировали. Вспоминается история 30-х годов, когда советский колхозник украл мешок колхозной пшеницы, и был осуждён за хищение в *крупных* размерах: если, мол, эту пшеницу всю посадить, да вырастить, да собрать, вышло бы несколько центнеров. Приходится изобретать хитроумную аппаратуру, *мешающую* копированию, или причиняющую при этом ущерб. Приходится вводить в законодательство особую категорию прав, условно назовём её "патент"<sup>(1)</sup>, ограничивающую злоупотребления -- и свободу -- всего человечества в пользу хозяина патента. Причём далеко не всегда хозяин патента и автор изобретения -- один и тот же человек!

Далее в лекции патентованные программные продукты и способ их разработки будут в чём-то противопоставлены свободно распространяемым программам. Термин "**патентованный программный продукт**" будет означать не наличие действительного патента на программу, а наличие у программы собственника, который относится к ней как к материальному объекту (в случае патентованных программ). Патентованные программы часто называют иногда "проприетарными" (от английского термина "proprietary") или просто "коммерческими" (что, строго говоря, неверно, так как "делать коммерцию" -- то есть получать выгоду -- можно различными способами, и многие успешные свободные проекты это подтверждают).

## Появление свободного ПО

Однако компьютеры развивались очень быстро, и бывшие вполне современными в 1970-е PDP-10 к началу 1980-х уже устарели, и значительно отставали по производительности от более современных машин. Однако ни для одной из новых архитектур уже не было операционной системы и прочего программного обеспечения, разработанного исключительно в академической среде. Университеты должны были покупать новые компьютеры с новым программным обеспечением и выполнять условия лицензии, ограничивающей их права на разработку и распространение ПО, иначе говоря, ограничивающей возможность научной модели разработки программного обеспечения.



В это время в лаборатории искусственного интеллекта МТИ разрабатывались так называемые LISP-машины, умевшие на аппаратном уровне интерпретировать язык программирования, похожий на LISP -- развитый и перспективный язык программирования. На LISP же была написана операционная система для таких машин и всё программное обеспечение для них. В начале 1980-х некоторые сотрудники лаборатории искусственного интеллекта выкупили у МТИ права на LISP-машины и математическую систему MACSIMA и основали собственные коммерческие компании для дальнейшей разработки в этой области. Очень многие сотрудники лаборатории перешли работать в эти компании, после чего все их дальнейшие разработки уже становились закрытыми для научного сообщества. Новые LISP-машины распространяются с лицензиями, запрещающими пользователям модифицировать и распространять исходные тексты программ. Программы, которые раньше для сотрудников МТИ были аналогом научных публикаций, стали принадлежащим кому-то патентованным продуктом.

Одному из сотрудников, оставшемуся в лаборатории искусственного интеллекта МТИ, Ричарду Столлману, такое положение дел казалось недопустимым нарушением открытого научного процесса разработки программного обеспечения. Он в одиночку пытался в рамках прежней академической модели развивать LISP-машины и открыто реализовывать изменения, аналогичные сделанным в рамках закрытой коммерческой разработки, чтобы LISP-машины МТИ могли конкурировать с патентованными аналогами. Конечно, эта попытка угнаться за активной разработкой целой компании была обречена на неудачу.

Тогда в поисках единомышленников Ричард Столлман создаёт некоммерческую организацию Фонд свободного программного обеспечения (Free Software Foundation, FSF). Своей основной целью Фонд ставит сохранение программного обеспечения, процесс разработки которого всегда будет гарантированно открытым, а исходные тексты всегда доступны. Более масштабная цель Фонда -- разработка операционной системы, *целиком* состоящей из открыто разрабатываемого программного обеспечения. Декларируя такую цель Столлман, фактически хотел вернуть представлявшееся ему идеальным состояние, когда в МТИ работали в собственной операционной системе для PDP-10.

Операционная система, разрабатываемая в рамках Фонда, должна была стать совместимой с операционной системой UNIX. Изначально UNIX был разработан в 1970-е годы Кеном Томпсоном и Деннисом Ричи в лаборатории компании AT&T и распространялся этой компанией (а впоследствии -- и другими) как патентованная операционная система. К началу 1980-х UNIX очень широко использовался, в том числе и в академической среде, для этой операционной системы существовало много программ, свободно распространявшихся в научном сообществе, поэтому хотелось, чтобы эти программы работали и в новой -- свободной -- операционной системе. Эта будущая операционная система получила название GNU(2).

## Определение свободного ПО

Для того, чтобы сохранить модель научного сотрудничества между разработчиками, необходимо было обеспечить, чтобы исходные тексты программ, написанных

разработчиками, оставались доступными для чтения и критики всему научному сообществу. Для этого Ричард Столлман сформулировал понятие **свободное программное обеспечение**, в котором отразились принципы открытой разработки программ в научном сообществе, сложившемся в американских университетах в 1970-е годы. Столлман явно сформулировал эти принципы, они же -- критерии свободного программного обеспечения. Эти критерии оговаривают те права, которые автор свободной программы передаёт любому пользователю.

- Программу можно использовать с любой целью ("нулевая свобода")<sup>(3)</sup>.
- Можно изучать, как программа работает и адаптировать её для своих целей ("первая свобода"). Условием этого является доступность исходного текста программы.
- Можно распространять копии программы -- в помощь товарищу ("вторая свобода").
- Программу можно улучшать и публиковать свою улучшенную версию, с тем чтобы принести пользу всему сообществу ("третья свобода"). Условием этого является доступность исходного текста программы.

Только удовлетворяющая всем принципам программа может считаться свободной, т. е. гарантированно открытой и доступной для научного сообщества. Нужно подчеркнуть, что эти принципы оговаривают только доступность программ для всеобщего использования, критики и улучшения, но никак не оговаривают связанные с распространением программ денежные отношения, в том числе не предполагают и бесплатности. В англоязычных текстах здесь часто возникает путаница, поскольку слово "free" обозначает не только "свободное", но и "бесплатное" и нередко употребляется по отношению к программному обеспечению, которое распространяется без взимания платы за *использование*, но которое при этом совершенно недоступно для изменения сообществом, просто потому, что его исходные тексты не опубликованы. Такое бесплатное ПО вовсе не является свободным. Наоборот, свободное ПО вполне можно распространять, взимая при этом плату, однако соблюдая при этом критерии свободы: каждому пользователю предоставляется право получить исходные тексты программ, изменять их и распространять далее. Всякое программное обеспечение, пользователям которого не предоставляется такого права, является *несвободным*.

Открытый доступ к исходным текстам программ, в действительности, является ключевым признаком свободного ПО, поэтому предложенный несколько позднее термин "open source software" (ПО с открытым исходным текстом) представляется даже более удачным для обозначения феномена свободного программного обеспечения, чем предложенный Столлманом "free software".

## Общественная лицензия GNU

Декларируя критерии свободного ПО, члены Фонда свободного ПО стали распространять свои программы в соответствии с этими принципами, никак не оформляя это документально, иначе говоря, первоначально свободные программы распространялись

вообще без **лицензии**. Однако произошедший с самим Ричардом Столлманом прецедент убедил его в том, что документальное оформление необходимо для свободного ПО.

Ричард Столлман занимался разработкой текстового редактора Emacs (о котором шла речь в лекции [Текстовые редакторы](#)) на основе исходных текстов Джеймса Гослинга (который впоследствии стал автором известного сегодня продукта Java). Тогда Гослинг свободно раздавал свои исходные тексты всем заинтересованным. Однако в некоторый момент Гослинг продал права на распространение Emacs компании [UniPress](#) (<http://www.unipress.com>) и компания попросила Столлмана прекратить распространение его версии Emacs, так как права принадлежат им. Этот инцидент заставил Столлмана переписать заново те части исходного текста Emacs, которые теперь принадлежали [UniPress](#), после чего он разработал собственную лицензию на программное обеспечение.

Лицензия, сформулированная Столлманом, должна была работать так же, как и лицензии на патентованное программное обеспечение: это типовой договор автора программы (обладателя авторских прав) с пользователем, в котором автор оговаривает права пользователя по отношению к программе. В отличие от коммерческой лицензии, в лицензии Столлмана оговариваются те права, которые пользователь получает по отношению к свободной программе: получать исходные тексты программ, изменять их, распространять изменённые и неизменённые версии (см. перечисленные выше критерии свободного ПО). Кроме того, в этой лицензии оговаривается принципиальное для Столлмана условие распространения свободного ПО: ни один пользователь не имеет права, сделав модифицированную версию свободной программы, распространять её, не соблюдая *всех* принципов свободного ПО, ограничивая тем самым права *других пользователей* по отношению к программе. Иначе говоря, нельзя модификацию свободной программы сделать несвободной.

Лицензия, содержащая такое условие, получила название "copyleft". Здесь игра слов: по-английски авторское право называется "copyright", буквально "копировать право", а "copyleft", соответственно, "копировать лево". Действительно, условие "copyleft" прямо противоположно по смыслу авторскому праву: авторское право призвано ограничить пользователя в копировании и распространении копий продукта, а "авторское лево", наоборот, строго запрещает его ограничивать. Впоследствии лицензия Столлмана получила название "Общественная лицензия GNU" (GPL, **GNU Public License**).

В настоящее время помимо GPL известны и другие лицензии, под которыми может распространяться свободное ПО. Самая распространённая из таких лицензий -- BSD(4) License. Лицензия BSD отличается от GPL главным образом тем, что в ней отсутствует условие "copyleft", т. е. на основании свободного ПО, распространяемого под этой лицензией, *можно* производить *несвободные* модификации. Однако лицензия BSD и другие лицензии до тех пор будут оставаться лицензиями на свободное программное обеспечение, пока они соответствуют условиям, оговорённым принципами свободного ПО, объявленными Фондом.

## Сообщество разработчиков и пользователей

Однако главное условие существования свободного ПО -- не лицензия, а люди, которые готовы делиться текстами своих программ и совершенствовать тексты чужих. Свободное ПО унаследовало модель открытой научной разработки, а вместе с ней -- и специфическую организацию *сообщества* разработчиков и пользователей, в некоторых отношениях напоминающую академическое сообщество. Чтобы лучше продемонстрировать специфику этого сообщества, сравним социальные отношения, сопровождающие использование свободного и патентованного ПО.

У любого пользователя программного обеспечения непременно возникают вопросы, когда он пытается применить его для решения своих задач. Традиционная коммерческая модель разработки и использования программного обеспечения основана на том, что исходные тексты программ являются коммерческой тайной производителя, а пользователь получает готовый продукт -- скомпилированную программу. Такая программа является несвободной. Пользователь несвободной (патентованной) программы платит за неё производителю, который взамен предоставляет ему некоторые гарантии, одна из которых -- отвечать на вопросы о работе программы. Специально для этого производитель организует *службу поддержки*, которая по телефону и по электронной почте отвечает на вопросы пользователей.

Пользователь свободно распространяемой программы не получает вместе с ней никаких гарантий: автор сделал её исходный текст открытым для общества, но при этом не брал на себя обязательств объяснять всем, как работает программа(5). Поэтому получить ответ на свой вопрос пользователь может из двух источников: из документации, а если её недостаточно -- от более опытных пользователей. Хорошо, если такие пользователи есть среди знакомых, а если нет? В этом случае их всегда можно найти в **списке рассылки** в Internet, посвящённом данной программе.

Письмо, пришедшее на электронный адрес списка рассылки, будет отправлено всем подписчикам списка, любой из них может ответить на него в списке, и ответ также получат все подписчики и т. д. Так организуется нечто вроде виртуальной общей комнаты для разговоров(6). В настоящее время сложилось неписанное правило, что для каждой свободно распространяемой программы существует отдельный **список рассылки**. Найти адрес этого списка и подписаться на него можно в Internet (обычно на сайте, посвящённом данной программе). Любой пользователь свободной программы может направить свой вопрос в список рассылки. Списки рассылки читают разработчики программы и её активные пользователи, и обычно среди них находится тот, кто ответит на вопрос(7). Так получается, что пользователи свободных программ, в отсутствие централизованной службы поддержки, организуются в сообщество для взаимопомощи.

У пользователей программ вновь и вновь возникают одни и те же вопросы и сложности. Постоянным читателям списков рассылки это особенно очевидно, поскольку им приходится на эти вопросы отвечать не по одному разу. В таких ситуациях у них нередко возникает инициатива записать ответы на самые распространённые вопросы и открыть их для

всеобщего обозрения. Так к свободной программе появляется новая документация в жанре FAQ (**F**requently **A**sken **Q**uestions, **Ч**Асто задаваемые **В**Опросы), представляющая собой список вопросов с ответами. Пользователи патентованных программ тоже задают одни и те же вопросы, только не в списке рассылки, а службе поддержки, в результате так же появляется документация типа FAQ, которая, почему-то, редко выходит за пределы внутреннего пользования производителя программы.

В любой программе непременно имеются ошибки (bugs). Производитель патентованной программы оплачивает работу отдела контроля качества, который занимается поиском ошибок. Тем не менее, некоторые ошибки этот отдел пропускает, и они достигают пользователя. Пользователь несвободной программы, столкнувшись с ошибкой, не может выявить её причину (поскольку ему недоступны исходные тексты программы), но, скорее всего, способен описать ошибку и условия, в которых она происходит. Он может сообщить об ошибке производителю программы (обычно посредством обращения всё в ту же службы поддержки), и если там решат, что ошибка действительно в программе, а не в работе пользователя, о ней будет сообщено разработчикам. В итоге пользователь может ожидать, что в следующей версии программы ошибка будет исправлена.

У свободно распространяемой программы обычно нет оплачиваемого отдела контроля качества. Значит, пользователь может столкнуться с ещё большим количеством ошибок, чем в патентованной программе. Тем актуальнее для него возможность сообщить об ошибке разработчикам программы. Раньше в сопровождающей программу документации было принято указывать электронный адрес, по которому разработчики принимали сообщения об ошибках, **bug report**. Некоторые вводили стереотипную форму для таких сообщений, чтобы облегчить и автоматизировать их обработку. Уже это требует существенно более высокой *связности* сообщества во всём мире, существенно большей, чем достаточно для закрытой разработки.

Разработчики и контролёры-испытатели патентованного продукта могут ходить на службу в один и тот же офис, и там обмениваться информацией или тратить определённую долю рабочего времени на составление и анализ строгих отчётов, содержащих сообщения о ошибках и рапорты об устранении неисправностей. Такая организация труда эффективна, если круг разработчиков невелик, а ввести общую дисциплину относительно легко (например, угрожая рублём). Для открытого проекта круг *потенциальных* разработчиков не ограничен ничем, поэтому эффективность разработки в гораздо большей степени зависит от того, насколько просто всем членам сообщества договариваться между собой, а также от "сознательности" пользователей. Заметив ошибку в программе, сознательный пользователь не просто исправит её самостоятельно (что не всегда ему по силам), а оформит внятное сообщение об ошибке, а если исправление готово, приложит к сообщению и его.

Простому и упорядоченному приёму и перенаправлению сообщений об ошибках служат системы отслеживания ошибок (**Bug Tracking System**), самые известные из которых разработаны участниками больших проектов для себя, а благодаря свободным лицензиям

используются повсеместно. Таковы GNUTS (разработанная в GNU), Bugzilla (mozilla.org), [JitterBug](#) (проект Samba) или Debian BTS. Более ранние версии ориентируются на электронную почту, более поздние включают в себя WWW-интерфейс. Например, при помощи Bugzilla организуется сайт в Internet, на котором пользователь может заполнить форму сообщения об ошибке. Каждое сообщение имеет свой номер, по которому можно попасть на "персональную" страницу данной ошибки, где отражаются все происходящие по её поводу события, от первоначального сообщения (открытия) до исправления (закрытия). При каждом изменении в состоянии ошибки Bugzilla рассылает всем заинтересованным лицам (включая, естественно, сообщившего об ошибке и занимающихся данной программой разработчиков) письма по электронной почте. Поскольку Bugzilla позволяет оставлять комментарии и прикладывать файлы, она является полноценным средством для общения пользователя с разработчиком по поводу ошибки в программе.

Принципиальное преимущество пользователя свободной программы заключается в том, что у него, в отличие от пользователей несвободных программ, всегда есть возможность заглянуть в исходные тексты. Конечно, для многих пользователей исходные тексты не более понятны, чем двоичные исполняемые файлы. Однако при достаточном уровне познаний в программировании пользователь может установить причину ошибки в программе и устранить её, исправив соответствующим образом исходный текст. А если пользователь заинтересован в развитии программы, то с его стороны будет разумно не только сообщить автору об ошибке, но и прислать ему свои исправления к исходному тексту программы: автору останется только применить эти исправления к тексту программы, если он найдёт их корректными и уместными. Пересылать автору исправленный текст программы целиком непрактично: он может быть очень большим (десятки тысяч строк), и автору будет нелегко разобраться, что же изменено (а вдруг изменения сделаны неграмотно?).

Чтобы облегчить и автоматизировать процесс внесения исправлений, Ларри Уолл (Larry Wall) в 1984 году разработал утилиту `patch` ("заплатка"), которая в формализованном (но хорошо понятном человеку) виде описывает операции редактирования, которые нужно произвести, чтобы получить новую версию текста. С появлением этой утилиты пользователь, обнаруживший и исправивший ошибку в программе, мог прислать автору небольшую заплатку, по которой автор мог понять, какие изменения предлагаются, и автоматически "приложить" их к своему исходному тексту. С появлением `patch` гораздо больше пользователей стало включаться в разработку программ с доступным исходным текстом, немалую роль и здесь сыграла сеть Usenet (см. об этом статью Тима ОРейли [http://tim.oreilly.com/articles/paradigmshift\\_0504.html](http://tim.oreilly.com/articles/paradigmshift_0504.html)). Файлы-заплатки с исправлениями -- обязательный атрибут сегодняшней разработки свободных программ.

Однако почему ограничивать сферу применения `patch` исправлением ошибок? Если пользователю программы не хватает в ней какой-то функции, то при должной квалификации он вполне может запрограммировать её сам и включить в исходный текст программы. Естественно, ему выгодно, чтобы его дополнение попало в "главный", авторский вариант программы (его называют "upstream") и появлялось во всех последующих версиях: можно

точно так же оформить его в виде patch и выслать автору. Этой возможности лишён пользователь несвободной программы, даже если он достаточно квалифицирован. Единственный способ включить в программу нужную ему функцию -- обратиться к производителю (если программа патентованная) с соответствующей просьбой, и надеяться, что производитель сочтёт предложенную функцию действительно необходимой.

Чем больше у свободной программы активных пользователей, готовых вносить исправления и дополнения и делиться ими, тем надёжнее работает и быстрее развивается программа. Причём такая свободная модель отслеживания и исправления ошибок для программы, у которой тысячи активных пользователей, может оказаться гораздо более эффективной, чем у любой патентованной программы: ни одна компания не может себе позволить такой огромный штат сотрудников в отделе контроля качества. Поэтому действительно популярная свободная программа может оказаться гораздо надёжнее патентованных аналогов.

Написать большую программу в одиночку довольно сложно и даже не всегда возможно, особенно если автор занимается этим в свободное от работы время. Большинство современных свободных программ пишется группой разработчиков. Даже если начинал писать программу один человек, и она оказалась интересной, к разработке могут присоединиться активные пользователи. Чтобы они могли не только вносить отдельные исправления, но и вообще всю разработку вести совместно, нужны специальные инструменты. Помимо patch, для организации совместной разработки ПО применяются системы контроля версий. Функции системы контроля версий состоят в том, чтобы организовать доступ к исходным текстам программы для нескольких разработчиков и хранить историю всех изменений в исходных текстах, позволяя объединять и отменять изменения и пр. Самая ранняя свободная система контроля версий, RCS использовалась ещё на заре свободного ПО абонентами сети Usenet, затем на смену ей пришла более развитая CVS, но сегодня и она считается во многом устаревшей, и всё чаще заменяется Subversion, Arch и другими. К слову, названные системы контроля версий сегодня активно используются и разработчиками патентованного ПО для организации совместной разработки.

Нужно заметить, что преимущества свободной разработки для пользователя не следует преувеличивать. Не все свободные программы в равной степени доступны для изменения пользователям, и это совершенно не связано с лицензией на их распространение. Важный фактор здесь -- объём программы: если в ней десятки тысяч строк (как, например, в [OpenOffice.org](http://OpenOffice.org)), то даже квалифицированному пользователю потребуется слишком много времени, чтобы разобраться, что к чему. А если при этом ещё нет толковой документации... Рассчитывать же на то, что разработчики ответят на все замечания и предложения пользователя немедленным исправлением программы тоже нельзя, поскольку они не несут перед пользователем никаких обязательств по качеству программы. В этом отношении пользователь патентованной программы может быть даже в лучшем положении.

Очень многие свойства сообщества разработчиков и пользователей свободных программ проистекают из того, что все его участники обычно занимаются этой программой

из интереса или потому, что эта программа -- необходимый для них инструмент (например, зарабатывания денег). Время, потраченное ими на программу, не оплачивается, поэтому нет никакой надежды, что обстоятельства не переменятся и разработка не прекратится вовсе. Нередки случаи, когда разработка программы начинается благодаря одному автору-энтузиасту, который привлекает многих к участию в разработке, а потом энтузиазм лидера гаснет, а вместе с ним затухает и разработка. К сожалению, сегодня существуют тысячи свободных программ, так никогда и не достигших версии 1.0, хотя "выгорание" лидеров и не единственная этому причина. Кроме того, программа может быть необходимой, но "неинтересной", а потому не найдётся и свободных разработчиков.

Место свободных программ на сегодняшнем рынке ПО очень значительно, и многие коммерческие и государственные предприятия используют свободное ПО прямо или опосредованно. Собственно, опосредованно все пользователи Internet задействуют, например, свободную программу Bind, предоставляющую службу DNS. Многие организации, особенно предоставляющие услуги через Internet, используют свободный web-сервер Apache, от работы которого непосредственно зависит их прибыль, не говоря уже о серверах на платформе Linux. Выгода использования свободного ПО очевидна: за него не приходится платить, а если приходится -- оно стоит гораздо дешевле патентованных аналогов. Главный недостаток с точки зрения коммерческого пользователя: разработчики свободных программ не несут никаких обязательств по качеству программы, кроме моральных. Поэтому сегодня большие корпорации, например, Intel или IBM, находят необходимым поддерживать проекты по разработке свободного ПО, оплачивая сотрудников, которые работают в рамках этих проектов.

## История Linux

### GNU без Linux

К 1990 году в рамках проекта GNU были разработаны и постоянно развивались свободные программы, составляющие основной инструментарий для разработки программ на языке Си: текстовый редактор Emacs, компилятор языка Си gcc, отладчик программ gdb, командная оболочка bash, библиотека важнейших функций для программ на Си libc. Все эти программы были написаны для операционных систем, похожих на UNIX. Это означает, что в них использовался стандартный для UNIX механизм запроса ресурсов компьютера, необходимых программе -- **системные вызовы**, которые исполняются **ядром** операционной системы. При помощи системных вызовов программы получают доступ к оперативной памяти, файловой системе, устройствам ввода и вывода. Благодаря тому, что системные вызовы выглядели более-менее стандартно во всех реализациях UNIX, программы GNU могли работать (с минимальными изменениями или вообще без изменений) в любой UNIX-подобной операционной системе.

С помощью имевшихся инструментов GNU можно было бы писать программы на Си, пользуясь *только* свободными программными продуктами, однако свободного UNIX-совместимого **ядра**, на основе которого могли бы работать все эти инструменты, не



существовало. В такой ситуации разработчики GNU вынуждены были использовать одну из патентованных реализаций UNIX, т. е. вынуждены были следовать принятым в этих операционных системах архитектурным решениям и технологиям и основывать на них свои собственные разработки. Идеал Столлмана о научной разработке ПО, свободной от решений, движимых коммерческими целями, был недоступен, пока в основе свободной разработки лежало патентованное UNIX-совместимое **ядро**, исходные тексты которого оставались тайной для разработчиков.

## Linux -- ядро

В 1991 году Линус Торвальдс, финский студент, чрезвычайно увлёкся идеей написать совместимое с UNIX ядро операционной системы для своего персонального компьютера с процессором ставшей очень широко распространённой архитектуры Intel 80386. Прототипом для будущего ядра стала операционная система MINIX: совместимая с UNIX операционная система для персональных компьютеров, которая загружалась с дискет и умещалась в очень ограниченной в те времена памяти персонального компьютера. MINIX был создан Энди Танненбаумом в качестве учебной операционной системы, *демонстрирующей* архитектуру и возможности UNIX, но непригодной для полноценной работы с точки зрения программиста. Так же, MINIX можно было использовать только в некоммерческих целях. Именно полноценное ядро для своего ПК и хотел сделать Линус Торвальдс. Название для своего ядра он соорудил из собственного имени, заменив последнюю букву и сделав его похожим на анаграмму слова UNIX.

Совместимость с UNIX в этот момент означала, что операционная система должна поддерживать стандарт POSIX. POSIX -- это **функциональная модель** совместимой с UNIX операционной системы, в которой описано, как должна вести себя система в той или иной ситуации, но не приводится никаких указаний, как это следует реализовать программными средствами. POSIX описывал те свойства UNIX-совместимых систем, которые были общими для разных реализаций UNIX на момент создания этого стандарта. В частности, в POSIX описаны системные вызовы, которые должна обрабатывать операционная система, совместимая с этим стандартом.

Важнейшую роль в развитии Linux сыграли глобальные компьютерные сети Usenet и Internet. На самых ранних стадиях он обсуждал свою работу и возникающие трудности с другими разработчиками в телеконференции comp.os.minix в сети Usenet, посвящённой операционной системе MINIX. Ключевым решением Линуса стала публикация исходных текстов ещё малоработоспособной первой версии ядра под свободной лицензией GPL. Благодаря этому и получавшей всё большее распространение сети Internet очень многие получили возможность самостоятельно компилировать и тестировать это ядро, участвовать в обсуждении и исправлении ошибок, и присылать исправления и дополнения к исходным текстам Линуса. Теперь над ядром работал уже не один человек, разработка пошла быстрее и эффективнее.

В 1992 году версия ядра Linux достигла 0.95, а в 1994 году вышла версия 1.0, что свидетельствует о том, что разработчики наконец сочли, что ядро в целом закончено и все

ошибки (теоретически) исправлены. В настоящее время разработка ядра Linux -- дело уже гораздо большего сообщества, чем во времена до версии 0.1, изменилась и роль самого Линуса Торвальдса, который теперь не главный разработчик, но главный авторитет, который традиционно оценивает исходные тексты, которые должны быть включены в ядро и даёт своё добро на их включение. Тем не менее, общая модель свободной разработки сообществом сохраняется. В настоящее время параллельно всегда разрабатывается два варианта ядра. Стабильная версия, считающаяся достаточно надёжной и пригодной для пользователей, её номер заканчивается на чётное число, например, "2 . 4". Номер соответствующей экспериментальной версии ядра оканчивается на нечётное число -- "2 . 5". Экспериментальная версия адресована в первую очередь разработчикам ядра, тестирующим новые возможности.

## GNU и Linux

Однако как нельзя сделать операционную систему без ядра, так и ядро будет бесполезно без утилит, которые использовали бы его возможности. Благодаря проекту GNU Линус Торвальдс сразу имел возможность использовать в Linux свободные утилиты: `bash`, компилятор `gcc`, `tar`, `gzip` и многие другие уже известные и широко используемые приложения, которые могли работать с его UNIX-совместимым ядром. Так Linux сразу попал в хорошее окружение и в сочетании с утилитами GNU представлял собой очень интересную среду для разработчиков программного обеспечения даже на самой ранней стадии своего развития.

Принципиальным шагом вперёд было именно то, что из ядра Linux и утилит и приложений GNU *впервые* стало возможно сделать полностью свободную *операционную систему*, т. е. работать с компьютером и, более того, разрабатывать новое программное обеспечение, пользуясь только свободным программным обеспечением. Идеал полностью некоммерческой разработки Столлмана теперь мог быть реализован в жизни.

Однако появление теоретической возможности воплощения идеала не означало его немедленной практической реализации. Совместимость Linux и утилит GNU была обусловлена тем, что и то, и другое писалось с ориентацией на одни и те же стандарты и практику. Однако, в рамках этой практики (множество различных UNIX-систем) оставался большой простор для несовместимости и различных решений. Поэтому на начальном этапе разработки ядра каждое заработавшее под Linux приложение GNU было для Линуса очередным достижением: первыми стали `bash` и `gcc`. Таким образом, сочетание GNU и Linux было возможностью создать свободную операционную систему, но само по себе ещё не составляло такой системы, потому что Linux и различные утилиты GNU оставались разрозненными программными продуктами, которые писали разные люди, не всегда принимая в расчёт то, что делают другие. Основное же качество системы -- согласованность её компонентов.

## Возникновение дистрибутивов

После определённого периода разработки под Linux уже стабильно работал ряд важнейших утилит GNU. Скомпилированное ядро Linux с небольшим комплектом скомпилированных уже в Linux утилит GNU составляло набор инструментов для разработчика программного обеспечения, желающего использовать свободную операционную систему на своём персональном компьютере. В таком виде Linux уже не только годился для разработки Linux, но и представлял собой операционную систему, в которой можно было уже выполнять какие-то прикладные задачи. Конечно, первое, чем можно было заниматься в Linux -- писать программы на Си.

Первоначально, чтобы получить компьютер с работающей системой Linux, разработчики пользовались специальными комплектами дискет со скомпилированным ядром Linux и утилитами: с этих дискет можно было загрузить Linux и работать в нём. Однако это не слишком удобно, когда нужно работать в Linux постоянно, да и объём дискет накладывал очень сильные ограничения на дальнейшее расширение системы и включение новых утилит.

Когда задача получить компьютер с постоянно работающей на нём системой Linux стала востребованной и довольно распространённой, разработчики в хельсинкском и техасском университетах создают собственные наборы дискет, с которых скомпилированное ядро и основные утилиты можно записать на жёсткий диск, после чего загружать операционную систему прямо с него. Эти наборы дискет -- первые прототипы современных **дистрибутивов** Linux -- комплекты программного обеспечения, на основе которых можно получить работающую операционную систему на своём компьютере. Нужно отметить, что в дистрибутив Linux с самого начала входили программные продукты GNU. На самом деле, всякий раз, когда говорится "операционная система Linux", подразумевается "ядро Linux и утилиты GNU". Фонд свободного ПО даже рекомендует называть это операционной системой GNU/Linux.

Однако скопировать все нужные программы на жёсткий диск ещё недостаточно, чтобы получить подходящую для нужд пользователя операционную среду (пусть даже это очень профессиональный пользователь). Поэтому первые наборы дискет можно только условно назвать дистрибутивами. Чтобы получить работающую операционную систему, требуются какие-то специальные средства установки и настройки программного обеспечения. Именно наличие таких средств и отличает современные дистрибутивы Linux. Другое важнейшая задача дистрибутива -- регулярное обновление. Программное обеспечение, особенно свободное, -- одна из самых быстро развивающихся областей, поэтому мало один раз установить Linux, нужно ещё регулярно его обновлять. Первым дистрибутивом в современном понимании, получившим широкое распространение, стал Slackware, созданный Патриком Фолькердингом (кстати, этот дистрибутив сохранился и до наших дней). Он был широко известен пользователям Linux уже к 1994 году.

Несмотря на то, что с появлением первых дистрибутивов установка Linux уже не требует самостоятельной компиляции всех программ из исходных текстов, использование Linux оставалось уделом разработчиков: пользователь этой операционной системы в тот

период её развития мог заниматься почти исключительно программированием. По крайней мере, чтобы решать в ней другие повседневные прикладные задачи (например, чтение электронной почты, написание статей и т. п.), он должен был сначала некоторое время позаниматься программированием и даже разработкой самой системы Linux, чтобы создать для себя соответствующие прикладные программы или заставить их работать в Linux.

Однако разработчики -- тоже люди, которые пишут и электронные письма, и статьи, и даже рисуют картинки. Всё программное обеспечение для Linux было открытым, поэтому вскоре стало появляться всё больше прикладных программ для Linux, которые использовались всё большим сообществом, отчего становились надёжнее и получали всё новую функциональность. В конце концов возникает идея, что из Linux и GNU-приложений для Linux целенаправленными усилиями небольшой группы разработчиков можно делать целостные операционные системы, подходящие для очень широкого круга пользователей и продавать эти системы пользователям за деньги как аналог и альтернативу существующим патентованным операционным системам.

Выгода операционной системы, целиком состоящей из свободного программного обеспечения, очевидна -- собирающие эту систему не должны никому платить за входящие в неё программы. Более того, дальнейшая разработка и обновление имеющихся программ ведётся сообществом разработчиков также совершенно бесплатно, не нужно платить сотрудникам, которые занимались бы этим. В итоге затраты фирмы, собирающей **дистрибутив** Linux для пользователя, ограничиваются оплатой программистов, интегрирующих разрозненные приложения в систему и пишущих программы для стандартизации процедур установки и настройки системы, чтобы облегчить эти задачи неподготовленному пользователю, а также затратами на само издание получившегося дистрибутива. Для конечного покупателя это означает принципиальное снижение цены на операционную систему.

Первой успешной компанией, работающей по такой схеме, стала [RedHat](#), появившаяся в 1995 году. [RedHat](#) адресовала свои разработки не только программистам профессионалам, но и обыкновенным пользователям и системным администраторам, для которых компьютер -- в первую очередь офисное рабочее место или рабочий сервер. Ориентируясь на уже существующие на рынке предложения для такого класса пользователей, [RedHat](#) всегда уделял большое внимание разработке приложений с графическим интерфейсом для выполнения типичных задач по настройке и администрированию системы. Бизнес [RedHat](#) развивался довольно успешно, в 1999 году эта компания акционировалась -- сразу после выпуска акции росли в цене очень энергично, однако потом ажиотаж улегся. В настоящее время доля [RedHat](#) на рынке серверов и рабочих станций Linux очень велика. Благодаря [RedHat](#) в сообществе пользователей Linux очень широкое распространение получил формат пакетов RPM.

Практически одновременно с [RedHat](#) появился проект Debian. Его задача была примерно той же -- сделать целостный дистрибутив Linux и свободного программного обеспечения GNU(8), однако этот проект был задуман как принципиально некоммерческий, проводимый в жизнь сообществом разработчиков, нормы взаимодействия в котором

полностью соответствовали бы идеалам свободного ПО. Сообщество разработчиков Debian -- международное, участники которого взаимодействуют через Internet, а нормы взаимодействия между ними определяются специальными документами -- **полиси** (policy).

Сообщество разработчиков не извлекает никакой прибыли от продажи Debian, его версии распространяются свободно, доступны в Интернет, могут распространяться и на твёрдых носителях (CD, DVD), но и в этом случае их цена редко сильно превышает стоимость носителя и наценку, окупающую затраты на издание. Первоначально разработка Debian спонсировалась Фондом свободного программного обеспечения. Адресатами дистрибутивов Debian всегда в первую очередь были профессиональные пользователи, так или иначе связанные с академической разработкой программного обеспечения, которые готовы читать документацию и собственными руками организовать нужный **профиль** системы, соответствующий именно их задачам. Ориентация на такую аудиторию предопределила некоторые тенденции развития Debian: в нём никогда не было обилия "простых" графических средств настройки среды, всевозможных **мастеров**, однако всегда уделялось много внимания средствам последовательной и единообразной интеграции программного обеспечения в единую систему. Именно в Debian появился менеджер пакетов (APT). В настоящее время Debian -- самый популярный дистрибутив Linux среди пользователей, являющихся профессионалами в области информационных технологий.

Всякий раз, когда свободное программное обеспечение оказывается востребованным, немедленно возникает множество альтернативных решений -- так произошло и с дистрибутивами Linux. После 1995 года возникло (и продолжает возникать) огромное количество коммерческих компаний и свободных сообществ, которые ставят своей задачей подготовку и выпуск дистрибутивов Linux. У каждого из них -- свои особенности, своя целевая аудитория, свои приоритеты. К настоящему времени на рынке дистрибутивов выделилось несколько лидеров, которые предлагают более или менее универсальные решения и наиболее широко известны и используются. Помимо уже названных [RedHat](#) и Debian следует назвать в ряду дистрибутивов, ориентированных на рядового пользователя, немецкий SuSE и французский Mandrake, среди адресованных специалистам -- Gentoo. Но помимо "крупных" игроков на рынке дистрибутивов есть гораздо большее количество менее распространённых дистрибутивов. Теперь перед пользователем, желающим установить Linux, встаёт вопрос выбора дистрибутива. Критерии выбора -- и задачи, которые предполагается решать с помощью Linux, и уровень подготовки пользователя, и технологии, и предстоящие контакты с тем сообществом, которое занимается разработкой дистрибутива.

## История Linux в России

Получилось так, что в международном сообществе разработчиков, начинавших и продолжавших развивать Linux, все в той или иной степени могли объясниться по-английски. Это и неудивительно, поскольку исторически английский оказался языком компьютерной науки и операционной системы UNIX, глобальной сети Internet, программирования. В международном сообществе разработчиков программного обеспечения английский выполнял и выполняет роль, подобную латыни в научном сообществе средневековой Европы. Но если

Linux предполагается использовать не только для программирования и общения с программистами, но и для повседневных задач, необходима локализация -- т. е. возможность общаться с компьютером и при помощи компьютера и на других языках.

Локализация -- комплексный процесс, затрагивающий самые разные стороны системы. Для полноценной поддержки того или иного языка в системе необходимо обеспечить возможность ввода на этом языке (поддержка раскладок клавиатуры и кодировок), вывода (экранных шрифтов), печати, а затем уже необходимо переводить интерфейс различных приложений на данный язык, разрабатывать средства подготовки электронных и бумажных публикаций на этом языке и т. д. В этой лекции мы кратко рассмотрим только историю локализации Linux в России для русского языка, т. е. русификации Linux.

Первой компанией, поставившей своей целью выпуск дистрибутивов Linux для русскоговорящих пользователей, была [УрбанСофт](#), открытая в Петербурге в 1992 году. Весь её бизнес состоял в выпуске и продаже CD-дисков с дистрибутивами свободного программного обеспечения. В первую очередь это были дистрибутивы [RedHat](#), а также Debian, в которые включались разработанные силами [УрбанСофт](#) пакеты для русификации.

Несколько позже в Москве IPLabs Linux Team выпускает Linux Mandrake Russian Edition -- модифицированный (чтобы соответствовать нуждам русского пользователя) вариант дистрибутива Mandrake Linux. Впоследствии эта команда начинает выпускать дистрибутивы, которые отличаются от Mandrake уже не только наличием пакетов для русификации, но и другими принципиальными возможностями. В конце концов команда разработчиков создаёт фирму ALT Linux и начинает выпускать дистрибутивы под маркой ALT Linux.

Также появляется компания ASPLinux, целью которой является выпуск [RedHat](#) с модификациями для поддержки русского языка, название продукта совпадает с именем компании.

Все перечисленные Российские производители дистрибутивов Linux существуют и по сей день, и продолжают с большей или меньшей активностью выпускать дистрибутивы.

---

(1) Условно -- потому, что далеко не во всех странах разрешено выдавать патенты на программное обеспечение, однако везде отношения собственности на исходные тексты программ регулируются общими или специальными разделами законов об авторском праве (разных в разных государствах).

(2) Это псевдоаббревиатура, для которой сам Столлман предлагал рекурсивную расшифровку: GNUS Not Unix ("Гну -- Не UNIX").

(3) Для российских пользователей эта свобода действительно "нулевая", в том смысле, что она присутствует всегда, в том числе и у пользователей патентованного ПО. По российскому законодательству обладатель прав на интеллектуальную собственность может передавать или не передавать пользователю право на *распространение* копий своего произведения (в данном случае -- программного обеспечения), однако у него нет никаких

прав каким бы то ни было образом ограничивать владельца копии в *использовании* программы.

(4) BSD -- **Berkeley Software Distribution**, пакет совместимого с UNIX программного обеспечения, разработанный в университете Беркли и распространявшийся свободно.

(5) В общественной лицензии GNU есть даже стандартная формулировка, закрепляющая отсутствие гарантий: "Настоящая программа поставляется на условиях "как есть". Если иное не указано в письменной форме, автор и/или иной правообладатель не принимает на себя никаких гарантийных обязательств, как явно выраженных, так и подразумеваемых, в отношении программы, в том числе подразумеваемую гарантию товарного состояния при продаже и пригодности для использования в конкретных целях, а также любые иные гарантии". Текст лицензии приводится в переводе Елены Тяпкиной.

(6) Списки рассылки в Internet -- наследники телеконференций сети Usenet, возникшей до появления Internet. В Usenet существовали телеконференции буквально на любую тему, и, конечно же, многие были посвящены программному обеспечению.

(7) Задавший вопрос пользователь должен принимать в расчёт, что все подписчики списка рассылки участвуют в нём добровольно, и никто из них не обязан отвечать на какие-либо вопросы, поэтому предъявлять претензии на этот счёт бессмысленно и невежливо.

(8) Официальное название дистрибутива -- Debian GNU/Linux.