Инструменты в Linux для программистов из Windows

Проект книги

Автор: Олег Цилюрик,

редакция 2.60,

04.12.2011г.

Оглавление

Введение 4
Структура книги 4
Что есть и чего нет в книге? 5
Соглашения и выделения, принятые в тексте 6
Код примеров и замеченные опечатки 6
Замечание о дистрибутивах и версиях ядра 6
Общие принципы 8
POSIX операционные системы - родовые черты 8
Операционная система Linux 9
Дистрибутивы Linux 10
Файловая система 13
Корневые каталоги 13
Важные системные файлы 15
Конфигурации (/etc) 15
Информация о состояниях (/proc и /sys) 16
Файловая система /sys 19
Данные и журналы (/var) 20
Каталог устройств (/dev) 21
Устройства хранения 22
Каталог загрузки (/boot) и коротко о загрузке 24
Монтирование файловых систем 26
Командный интерпретатор 29
Переменные окружения 30
Некоторые важные переменные 32
Встроенные переменные 33
Консольные команды 34
Формат командной строки 35
Уровень диагностического вывода команд 36
Фильтры, каналы, конвейеры 36
Справочные системы 37
Пользователи и права 38
Файловая система: структура и команды 42
Владельцы и права 43
Информация о файле 43
Дополнительные атрибуты файла 44

Навигация в дереве имён 45
Основные операции 45
Архивы 48
Устройства 48
Подсистема udev 49
Команды диагностики оборудования 50
Компиляция и сборка приложений 53
Компилятор GCC 53
Библиотеки 55
Библиотеки: использование 55
Библиотеки: связывание 55
Библиотеки: построение 59
Как это всё работает? 63
Конструктор и деструктор 63
Подмена имён 66
Данные в динамической библиотеке 67
Некоторые сравнения 69
Создание проектов, сборка make 69
Как существенно ускорить сборку make 71
Сборка модулей ядра 74
Прочий инструментарий создания программных проектов 74
Другие языки программирования 74
Интегрированные среды разработки 75
Установка программного обеспечения 76
Бинарная установка 76
Пакетная установка 77
Пакетная система rpm и менеджер yum 78
Пакеты исходных кодов 79
Создание собственного инсталляционного пакета 79
Инсталляция из исходников 80
Непосредственная сборка 81
Autoconf / Automake 82
Создание своего конфигурируемого пакета 84
Cmake 86
Портирование POSIX программного обеспечения 87
Инструменты удалённой работы 90
Сеть Linux 90
Сетевые интерфейсы 90
Инструменты управления и диагностики 92
Протокол ssh (Secure Shell) 93
Клиент telnet 93
Клиент rlogin 95
Средства ftp / tftp 95
Система nfs 96
Удалённые сессии в файловом менеджере mc 99
Удалённый X11 99
Нативный протокол X....100
Графическая сессия ssh 102
Сети Windows 103
Пакет Samba 103

Печать с Samba 104
Серверная часть Samba 104
Файловые системы smbfs и cifsfs 105
Библиотеки API POSIX 107
Сводный перечень по разделам API 107
Окружение процесса 108
Обработка опций командной строки 108
Параллельные процессы 109
Время клонирования 111
Загрузка нового экземпляра (процесса) 113
Сигналы 121
Модель ненадёжной обработки сигналов 122
Модель надёжной обработки сигналов 123
Модель обработки сигналов реального времени 125
Сигналы в потоках 127
Параллельные потоки 127
Создание потока 127
Параметры создания потока 129
Временные затраты на создание потока 131
Активность потока 131
Завершение потока 132
Данные потока 133
Собственные данные потока 133
Сигналы в потоках 135
Расширенные операции ввода-вывода 137
Неблокирующий ввод-вывод 138
Мультиплексирование ввода-вывода 138
Ввод-вывод управляемый сигналом 141
Асинхронный ввод-вывод 142
Терминал, режим ввода: канонический и неканонический 143
Приложения 146
Приложение А : Восстановление пароля root 146
Использование мультизагрузчика GRUB 146
Использование загрузочного Live CD 146
Соображения безопасности 147
Источники использованной информации 148

Введение

Весь представленный ниже текст был собран в ходе подготовки и проведения курса тренингов по программированию модулей ядра (драйверов) Linux, которые мне предложила организовать компания Global Logic (http://www.globallogic.com/) для сотрудников украинских подразделений компании. Но в ходе проведения этих занятий, первый тур которых проводился весной-летом 2011 года в городе Харькове, выяснилось следующее: значительная часть участников тренингов являются профессиональными разработчиками, с солидным опытом разработки программных проектов, но профессионализм этот наработан в других средах разработки (все варианты Windows систем, системы QNX, Solaris, встраиваемое оборудование и другое), а в Linux они обладают максимум уровнем добросовестного пользователя. И оказалось, что, при всём великом множестве, не так легко найти и посоветовать такому специалисту книгу, которая быстро восполняла бы этот пробел:

- есть множество изданий «для чайников», но смешно специалисту с многолетним опытом разработки начинать с объяснений что такое файл...
- есть множество изданий [2, 10, 17 и др.], посвящённых детальному и глубокому анализу, но отдельных аспектов Linux: структура файловой системы UNIX, программный API POSIX, сетевые средства и инструменты ... и так далее.

Но мне не удалось ни одно найти издание, которое бы очень бегло, в максимально сжатом объёме, «пробежалось» бы только по отличительным сторонам POSIX/Linux, и, опираясь на глубокие знания деталей из других операционных систем, связало бы аналогии и ассоциации разных систем в единую картину. И тогда мне пришлось, сверх планируемого курса по программированию модулей ядра Linux, написать и этот текст ;). Системы Windows (любая из них, как родовое понятие) в этом контексте названы только как наиболее распространённые, это может быть любая среда — принципиально то, что предполагается знание и понимание основных понятий и терминов, безотносительно к конкретной реализации.

В конечном счёте, то, что получилось - это и есть фрагментарная «памятка»: отдельные разрознённые фрагменты, которые, как мне казалось, нужно выделить, чтобы на **начальном этапе** работы в Linux иметь меньше хлопот (быстрее «въехать» в прямую программистскую деятельность). Ничего большего от этого текста и не следует ожидать. А в связи с специфичностью представленного текста (как по предназначению, так и по происхождению), хочется отметить ещё несколько получившихся производных его особенностей (ну, ... так получилось):

- Примеры и команды, их иллюстрирующие, в любом описании имеют выраженную направленность на определённую аудиторию. Большинство описаний Linux делают направленность на пользователя системы (начиная с установки, настройки). Данный текст отходит от этой традиции: здесь направленность на программиста, уже **работающего** в этой системе...
- Примеры, при их отборе для такого беглого обзора системы Linux, естественно, обладают выборочностью, фрагментарностью. И выборочность эта, в данном случае, мной направлялась на те подмножества команд, с которыми наиболее активно работают именно в ходе программной разработки. Почти полностью опущены команды администрирования системы — оставлено только то, что полезно программисту для настройки его **индивидуального** рабочего места.

Так что, я загодя прошу прощения у пользователей системы Linux, и администраторов систем и сетей, в том, что сознательно ущемил круг их их интересов в пользу программистов-разработчиков. Но так и ставилась цель написания...

Структура книги

«Скажешь ты, я мечтатель,
Но такой я не один»

Джон Леннон «Imagine».

Вводная (как это называется у армейских - «представь себе, что...»): вы прикладной программист-разработчик, имеющий изрядный опыт разработок в языке С, но не работавший непосредственно в операционной системе Linux. И вам нужно в минимальные сроки окунуться в среду **прикладной** разработки в этой системе ... возможно, перенос и адаптирование (портирование) уже готового ранее программного проекта в эту систему.

Задача: какие шаги вас должны интересовать в наибольшей мере, чтобы не требовать отвлечения на структуру, особенности операционной системы, не увязать в вопросах многочисленных настроек и администрирования? ... только тот минимум, который позволит начать работу? Я думаю (это **только** моё личное видение!), что это:

1. Беглое **перечисление** наиболее часто употребляемых в ходе программной разработки консольных команд. Не описание (это слишком громоздко), а именно перечисление в нескольких примерах и на интуитивно понятном уровне (дальнейшее понимание придёт в ходе использования: «винтовку добудете в бою»).
2. В меру **подробный** обзор программных инструментов, используемых при создании программных проектов на **языке C** используя **компилятор gcc** (все остальные языковые инструменты — это уже надстройка более высокого уровня, которая, во многом, системно независимая).
3. Обустройство рабочего места, **рабочего окружения** (для написания C кода, компиляции, запуска, отладки, ...). Поскольку в Linux-разработке особое место по значимости занимают возможности удалённой (сетевой) работы, то акцент делаем именно на таком инструментарии (локальный инструментарий более понятен и из аналогий других операционных систем).
4. Библиотеки **для разработчика** на C (именно C есть родной язык разработки Linux), то, что называется стандартами API POSIX. Причём, общеизвестные, переносимые из одной операционной системы в другую, вызовы API (а таких большинство, пожалуй 90%), такие как `strlen()` или `fopen()` - можно вообще даже не называть, а вот на UNIX-экзотике, специфических вещах, таких как `fork()` или сигналы UNIX — остановиться максимально подробно.

Вот из этого, и именно в такой последовательности и сложилась структура книги. Кое-где она в дальнейшем разбавлена некоторыми вторичными деталями, которые разрешают те мелкие, но досадные неприятности, которые возникают в практической работе. Ничего более здесь нет, и если вы ищите что либо, выходящее за рамки перечисленного, то вам вряд ли следует терять время на этот текст.

Что есть и чего нет в книге?

Цели, задачи и структура изложения, уже названные выше, и определяют отбор материала, вошедшего в книгу. Разработка драйверов (модулей ядра, если применительно к Linux), или перенос и адаптация мультиплатформенных проектов — две области программной деятельности, которые в наибольшей мере интересуют меня, и контингент моих коллег по тренингам. Эти виды деятельности могут и не требовать каких-то глубинных знаний, специфических для конкретной операционной системы. Но нужно было выбрать и осветить те стороны предстоящей деятельности, которые могли создать рутину и задержки в освоении приёмов работы, препятствующие прямой разработческой деятельности. Уже по итогу написания получилось, что такие вопросы отчётливо улеглись в небольшое число тематических частей (групп вопросов), а именно:

- Консольные команды, и именно те команды, которые максимально часто мелькают в программной работе. Кроме того то, как получить справочную информацию по таким командам. Всё (с небольшими расширениями относительно связанных архитектурных особенностей) составляют первую, начальную часть.
- Инструменты, непосредственно предназначенные для создания, компиляции и сборки проектов — это следующая группа вопросов.
- Удалённая (сетевая) работа, всегда широко использовавшаяся в UNIX, но менее популярная в других средах. Инструменты для такой работы.
- Набор библиотечных вызовов POSIX API, особенно применительно к таким ключевым для UNIX вызовам и понятиям (отсутствующим в других системах), как `fork()`, сигналы и некоторым другим.

Даже пусть поверхностного ознакомления с этими вопросами уже достаточно для создания начального окружения для развёртывания программной работы. Из-за акцента на этих ключевых группах вопросов, из рассмотрения сознательно совершенно опущены такие важные вопросы как:

- Инструменты, средства и приёмы отладки создаваемого кода, его профилирования.
- Средства создания и редактирования исходного кода, использование цветовой разметки кода и смежные вопросы.

– Доступные интегрированные среды разработки, а их в Linux великое множество: Eclipse, Solaris Studio, Kdevelop, IDEA, ...

– Средства контроля версий и организации групповой работы (Subversion/SVN, GIT, ...), их графические оболочки).

Это всё актуальные и интересные вопросы, но это уже вопросы ... «второго порядка малости», поэтому оставим их на дальнейшую самостоятельную проработку. Кроме всего прочего, в этих вопросах присутствуют много альтернативных, взаимно исключающих возможностей и средств, так что многое здесь зависит от субъективных индивидуальных предпочтений и привычек.

В завершение затянувшегося введения, хочу отметить следующее: ряд утверждений в последующем тексте могут быть спорными, а кому-то могут показаться и неприемлемыми. Но это — моё видение и мой текст, так, как я считаю нужным для себя его излагать. При этом я стараюсь в этих мнениях придерживаться (насколько мне это удаётся) объективности, и не подыгрывать ни одной из сторон в «религиозных войнах» и конкурентных взаимных происках, которые, как нигде, развернулись на IT пространствах.

Соглашения и выделения, принятые в тексте

Для ясности чтения текста, он размечен шрифтами по функциональному назначению. Для выделения фрагментов текста по назначению используется разметка:

– Отдельные ключевые понятия и термины в тексте, на которые нужно обратить особое внимание, будут выделены **жирным шрифтом**.

– Тексты программных листингов, вывод в ответ на консольные команды пользователя размечен `моноширинным шрифтом`.

– Программным листингам предшествует имя файла (отдельной строкой), где находится этот код, это имя файла выделяется ***жирным курсивом с подчёркиванием***.

– Таким же `моноширинным шрифтом` (прямо в тексте) будут выделяться: имена команд, программ, файлов ... т.е. всех терминов, которые должны оставаться неизменяемыми, например: `/proc`, `mkdir`, `./myprog`, ...

– Ввод пользователя в консольных командах (сами команды, или ответы в диалоге), кроме того, выделены жирным моноширинным шрифтом, чтобы отличать от ответного вывода системы.

– Текст, цитируемый из другого указанного источника, выделяется (для ограничения) *курсивным написанием*.

Код примеров и замеченные опечатки

Все протоколы выполнения команд и программные листинги, приводимые в качестве примеров, были опробованы и испытаны. Все примеры, обсуждаемые в тексте, предполагаю воспроизведение и повторяемость результатов. Примеры программного кода сгруппированы по темам в архивы, поэтому всегда будет указываться имя архива (например, `xxx.tgz`) и имя файла (например, `xxx.c`); некоторые архивы могут содержать подкаталоги, тогда указывается и подкаталог для текущего примера. Большинство архивов содержат файлы виды `xxx.hist` (для архива `xxx.tgz`) — в них содержится скопированные с терминала результаты выполнения примера, показывающие как этот пример должен выполняться, в сложных случаях здесь же могут содержаться команды, показывающие порядок компиляции и сборки примеров архива.

Конечно, и при самой тщательной выверке и вычитке, не исключены недосмотры и опечатки в таком объёмном тексте, могут проскочить мало внятные стилистические обороты и подобное. Да и в процессе вёрстки книги может быть привнесено много любопытного... О замеченных таких дефектах я прошу сообщать по электронной почте olej@front.ru, и я был бы признателен за любые указанные недостатки книги, замеченные ошибки, или высказанные пожелания по её доработке.

Замечание о дистрибутивах и версиях ядра

Примеры и команды, показываемые в тексте, отрабатывались на самых разных инсталляциях Linux. В первую очередь, здесь разделение может идти по признаку: реальная или виртуальная установка. Излагаемый материал

отрабатывался и опробовался на 2-х виртуальных (Virtual Box) и и 4-х реальных инсталляциях, на аппаратуре, принадлежащей к разным поколениям (указан год производства именно оборудования):

1. Fedora 14 - 64-бит инсталляция, 2011 год 4 ядра :

```
$ uname -r
2.6.35.13-91.fc14.x86_64
```

2. Fedora 12 - 32-бит инсталляция, 2008 год 2 ядра :

```
$ uname -r
2.6.32.9-70.fc12.i686.PAE
```

3. CentOS 5.2, 1998 год одноядерный Celeron :

```
$ uname -r
2.6.18-92.el5
```

4. Ubuntu 10.04.3 LTS, 2011 год 4 ядра Atom:

```
$ uname -r
2.6.32-33-generic
```

Сознательно выбраны разнородные линии дистрибутивов (RedHat/Fedora/CentOS & Debian/Ubuntu). Конечно, есть некоторые незначительные отличия (часть их будет отмечена), но, в итоге, работа подтвердила то, что резюмировано далее в тексте: не ищите принципиальных различий в дистрибутивах, и не гоняйтесь за обновлениями до самой свежей версии ядра — отличия мало существенны. Общее правило должно быть следующее: для **разработчика** (программиста) вид дистрибутива должен быть — дело второе, которое может «сыграть» на последних этапах подготовки проекта — создание инсталляционных пакетов и конфигураций.

К версии же **ядра** Linux (например, при работе над модулями-драйверами) вообще нужно подходить с очень большой осторожностью: ядро — это не пользовательский уровень, и разработчики не особенно обременяют себя ограничениями совместимости снизу вверх (в отличие от пользовательских API). Обновления версий ядра производится весьма часто — вот, для примера, короткий фрагмент хронологии выходов нескольких последовательных версий ядра (в последней колонке указано число дней от предыдущей версии до текущей), взято с http://en.wikipedia.org/wiki/Comparison_of_operating_system_kernels :

```
...
2.6.30    2009-06-09    78
2.6.31    2009-09-09    92
2.6.32    2009-12-02    84
2.6.33    2010-02-24    84
2.6.34    2010-05-15    81
2.6.35    2010-08-01    77
...
```

Среднее время до выхода очередного ядра на протяжении 5-ти последних лет (2005-2010) составляло 81 день, или около 12 недель (взято там же). Но итоги этих частых обновлений часто весьма мало ощутимы программистами прикладного уровня, и уж тем более пользователями...

Общие принципы

Операционная система Linux принадлежит к системам, называемым как UNIX-like, или POSIX[1] совместимыми. POSIX — это **набор из нескольких** стандартов. POSIX на сегодня является зарегистрированной торговой маркой комитета IEEE, для того, чтобы получить официальные тексты стандартов, вам придётся прежде заплатить IEEE. Но есть другой источник, который текстуально практически совпадает со стандартами POSIX - The Open Group Base Specifications Issue 7 [28][2]. Более подробно о стандартах POSIX мы будем говорить в последней части нашего рассмотрения, а сейчас только отметим, что под термином POSIX я буду называть (не следуя строгим формальным правилам) не какой-то строгий документ, а целый набор нескольких стандартов (POSIX, Open Group Base Specifications, UNIX 98 и др.), которые в основной части совпадают, а в деталях иногда и дополняют друг друга.

POSIX операционные системы - родовые черты

Основные общие признаки операционных систем, которые называют POSIX совместимые, или родовым именем UNIX:

1. Многопользовательские системы с разграничением прав по имени **пользователя и группы**, наличие пользователя `root` с неограниченными правами.
2. Древовидная файловая система, с единым корнем от `/`; большинство сущностей системы отображается как имя в дереве файловой системы; функциональное назначение каталогов файловой системы сохраняется примерно постоянным от одной системы к другой.
3. Приверженность символьным форматам: конфигурации системы и всех программных пакетов представляются в текстовых файлах (последнее время иногда в файлах XML), это позволяет изменять все конфигурации простым текстовым редактированием.
4. Единообразный базовый набор консольных утилит-команд (стандарт POSIX 2).
5. Единый API (вызовы программных функций) программирования языка C (стандарт POSIX 1, POSIX 1.b, POSIX 1.c, POSIX 1.j, UNIX 98 и другие).

По адресу http://gentoo.theserverside.ru/gentoo-doc/Gentoo_doc-1.5-6.html — можно посмотреть: история, классификация, перечисление стандартов (POSIX и другие); большое перечисление основных команд; рассмотрены основные дистрибутивы (Slackware, Debian, RedHat) Linux их хронология; краткий обзор линии BSD; рассмотрение лицензии GPL.

К этому роду (POSIX совместимые) принадлежат очень много принципиально **различающихся** операционных систем: Linux, все ветви BSD (FreeBSD, NetBSD, OpenBSD, ...), Sun/Oracle Solaris, Mac OS, QNX, MINIX3 и много других (проще, пожалуй, перечислить системы, которые **не** принадлежат к POSIX совместимым: **все** системы семейства Windows, Plan 9, Inferno, Blue Botle и некоторые другие).

Какие преимущества даёт совместимость операционной системы со стандартами POSIX (принадлежность её к роду UNIX)? Ответ будет выглядеть как очень протяжённое перечисление, среди которых минимум самых основных аргументов выглядит так:

- Простота переноса (портирование) программных проектов их одной операционной системы в другую: часто это достигается путём выполнения ряда чисто формальных действий, иногда требует некоторой изобретательности, но почти всегда это работа трудоёмкостью в несколько часов. Естественно, этот пункт срабатывает только для открытых программных проектов (под разнообразными публичными

1 http://www.intuit.ru/department/se/pposix/: POSIX - Portable Operating System Interface - мобильный (в смысле **переносимый**!) интерфейс операционной системы; название предложил основатель Фонда свободного программного обеспечения (FSF) Ричард Столмэн.

2 Это стандарт по состоянию на 2008 год, на момент написания этого текста более позднего варианта не существует.

лицензиями: GNU, BSD, Mozilla, Apache и другие).

– Простота для программиста-разработчика «пересаживаться» из одной операционной системы в другую: часто период адаптации в совершенно незнакомой операционной системе исчисляется в считанные дни: «... садится Гендель за рояль и играет Моцарта». Не треуется практически никакое дополнительное обучение.

Операционная система Linux

О Linux исписаны миллионы страниц, и я не стану пытаться здесь их пересказать... Остановлюсь только на некоторых, не столь очевидных и не столь однозначных (по разному толкуемых программистской общественностью) вопросах.

Программный код того, что мы в обиходе называем операционной системой Linux имеет несколько источников происхождения:

1. Код ядра Linux, который развивается и контролируется группой разработчиков под руководством Линуса Торвальдса.
2. Код программных утилит, который во многом развивается сообществом GNU, и, в частности, очень многое сделано под эгидой FSF (Free Software Foundation), основанной Ричардом Столменом.
3. Прямые или «по мотивам» заимствования из кода других открытых POSIX совместимых операционных систем, например, сетевой стек TCP/IP практически во всех таких системах заимствуется из NetBSD.
4. Независимые целевые проекты, из которых можно назвать (только для того, чтобы представить о чём идёт речь — перечислить миллионы таких проектов невозможно) те, которые стали целыми линиями развития (целая группа проектов) и самостоятельными сообществами: Apache, Mozilla, ...

Границы этих ... источников и составных частей — размыты. Существует мнение, что называть именем Linux имеет смысл только ядро. Всё остальное (в сотни раз превосходящее ядро объёмом) - обеспечение пользовательского пространства, разрабатываемое совершенно другими разработчиками, дополняет ядро до операционной системы, поэтому и операционную систему предлагают именовать: GNU/Linux. Но, наверное, это слишком изощрённо и громоздко... По крайней мере эти терминологические изыски не существенны для всего нашего дальнейшего рассмотрения.

В иллюстрацию отмеченной **разнородности** полезно указать на очень характерный пример: на систему графического интерфейса пользователя (GUI) - X Window System, называемой ещё просто системой X11 или X (я буду использовать эти разные названия без придания каких либо терминологических оттенков). Это система настолько показательна и значима для Linux (и это многое объясняет в структуре Linux), что о некоторых её деталях стоит здесь упомянуть отдельно:

- Система X11 не является как либо соотносящейся с Linux его составной частью - она начала развиваться в рамках вообще POSIX систем задолго до того, как прозвучало слово Linux[3]...
- Более того, система распространяется как свободно доступная система, но на условиях лицензии, отличной от GPL (под которой находится подавляющее большинство программного обеспечения Linux) — это лицензия MIT.
- Реализации X11 для для свободных UNIX-подобных операционных систем систем (и **в том числе** и Linux) развиваются в составе нескольких независимых проектов. Вплоть до 2004 года наиболее распространённым был проект XFree86[4]. На сегодня самой используемой реализацией является проект

3 Система X Window System была разработана в Массачусетском технологическом институте (MIT) в 1984 году. Нынешняя (по состоянию на начало 2009 года) версия протокола — X11 — появилась в сентябре 1987 года.

4 XFree86 возник как порт X на 386-совместимые персональные компьютеры. К концу 1990-х (ещё до зарождения Linux!) этот проект стал главным источником технических инноваций в X Window System и де-факто руководил разработкой X11. Однако в 2004 году XFree86 поменял условия лицензии, и реализация X.Org Server (которая является форком XFree86, но со свободной лицензией) стала более распространённой.

X.Org Server[5]. Кроме этого существуют и коммерческие проприетарные реализации.

- Вся система X11 (X-сервер и его драйверы конкретных разнообразных моделей видеоадаптеров) в реализации Linux работает в адресном пространстве пользователя (не в пространстве ядра!), обмен с портами видеоадаптера производится из пространства пользователя **получая привилегии I/O**, X11 вообще не подгружает модулей ядра (там есть более поздние доработки от сторонних исполнителей — модули ядра, но это уже сверх реализации X11, и направлено только на решение вопросов производительности).
- X-система - это сугубо **сетевая система**, в которой X-клиенты (графические приложения) взаимодействуют со средой ввода-вывода (X-сервер) через сетевой сокет. Это оказывается очень необычно для других GUI систем, или для реализаций в других операционных системах (Windows, например).
- Если бы на сегодня из операционной системы Linux исключить её «внешний» компонент — GUI окружения X11 с оконными менеджерами, оболочками рабочего стола, множеством X-программ - то систему **Linux покинули бы**, пожалуй, 90-95% её пользователей...

К системе X Window System мы ещё будем возвращаться позже не раз, поскольку она существенно значима для разработчика программных проектов.

Дистрибутивы Linux

Помимо собственно понятия операционной системы Linux, состав и структуру которой мы уже по-верхам затронули выше, существуют ещё такие понятия (градации) как дистрибутивы системы Linux — **те формы**, в которых операционная система Linux доходит до потребителя.

Во всех дистрибутивах Linux исходный код ядра **един**: он не может быть отличным по принципам и правилам Linux — тогда это уже будет другая операционная система, и она должна называться по-иному.

Примечание: В принципе, дистрибьюторы, как правило, в большей или меньшей степени на свой вкус вносят некоторые изменения в код ядра («патченное ядро»). Чаще всего это дополнения, не принятые в официальную версию ядра («ванильное ядро») командой разработчиков ядра. Поэтому ядро конкретного дистрибутива может **в мелких деталях** отличаться от официального.

Как уже было разграничено по составу: множество утилит для системы разрабатывают другие, совершенно независимые от разработчиков ядра и не координирующиеся с ними команды разработчиков (GNU сообщество, FSF и разработчики независимых проектов) — это вторая движущая сила развития системы.

Наконец, дистрибьюторы, как третья сторона, участвующая в процессе, только отбирает те программы и компоненты, которые, по их мнению, должны включаться в комплект поставки. Так появляются различные дистрибутивы. Существует (существовало: они то появляются, то исчезают) уже **несколько сот** различных дистрибутивов Linux[6], и такое их количество, как ничто другое, подтверждает то мнение, что различные дистрибутивы имеют потребительские отличия, но не имеют принципиальных функциональных[7]: всё, что можно сделать в одном дистрибутиве, можно сделать и во всех остальных — это **одна** операционная система Linux.

Основные различительные черты дистрибутивов (дистрибьюторы указывают заметно больше отличительных признаков, но они вытекают из этого набора основных):

---

5 Официально: «X.Org Foundation Open Source Public Implementation of X11»).

6 Вплоть до того, что разразились «религиозные войны» между приверженцами различных дистрибтивов Linux: какой дистрибутив «лучше»...

7 А ещё такое **неимоверное** количество дистрибутивов Linux — это бизнес-модель коммерческой деятельности дистрибьюторов, а «религиозные войны» - ничего более, чем конкурентная война на этом рынке.

1. Используемая пакетная система[8].
2. Принятые правила конфигурирования (каталог /etc).
3. Отношение к программным проектам со «спорным» лицензированием и включение их в дистрибутивы (XFree86 / Xorg, Qt4 / KDE, средства MP3 и многое другое).
4. **Как следствие**: некоторое разграничение целевой ниши дистрибутива (сервера, настольные рабочие станции, мультимедийное использование и так далее...).

Отчётливо выделяется несколько **семейств** дистрибутивов, основные из которых (оценочные цифры числа дистрибутивов в группе собраны по состоянию на май 2011г.):

- Дистрибутив Debian и дистрибутивы, производные от него, используют формат пакетов .deb и инсталлятор пакетов dpkg, существуют еще и средства пакетного мета-менеджмента, наиболее известное и распространенное из них — apt. В этом семействе около 98 дистрибутивов, самые распространённые дистрибутивы этого семейства (не считая самого Debian): Knoppix & Ubuntu.

- Дистрибутивы на базе RedHat или использующие формат пакетов .rpm и одноимённый инсталлятор rpm. Под явным влиянием apt (из Debian) возникли и иные системы пакетного менеджмента, для этого семейства это yum (в некоторых дистрибутивах urpmi). В этом семействе около 38 дистрибутивов, в частности: «Мобильная Система Вооруженных Сил» (МСВС) - специальный дистрибутив, разработанный для нужд МО РФ; дистрибутивы официально принятые в системе российского всеобщего среднего образования: ALT Linux и ASP Linux (система Linux, кстати, кроме России принята как официальная система сети среднего образования в Бразилии и Китае).

- Slackware подобные дистрибутивы. Это дистрибутивы серверной ориентации, традиционно установка пакетов в них практиковалась из архивов исходных кодов формата .tgz, но и в них стали применяться пакетные инсталляторы slapt-get (вариант apt) и slackpkg. Сторонники этих дистрибутивов считают, что это «самый чистый» Linux. В этом семействе около 11 дистрибутивов.

- Gentoo: дистрибутив, ориентированный на энтузиастов и профессионалов, с собственной системой управления пакетами Portage. Gentoo ориентируется на компилирование из исходного кода, а не на распространение бинарных (прекомпилированых пакетов). В этом семействе около 6 дистрибутивов.

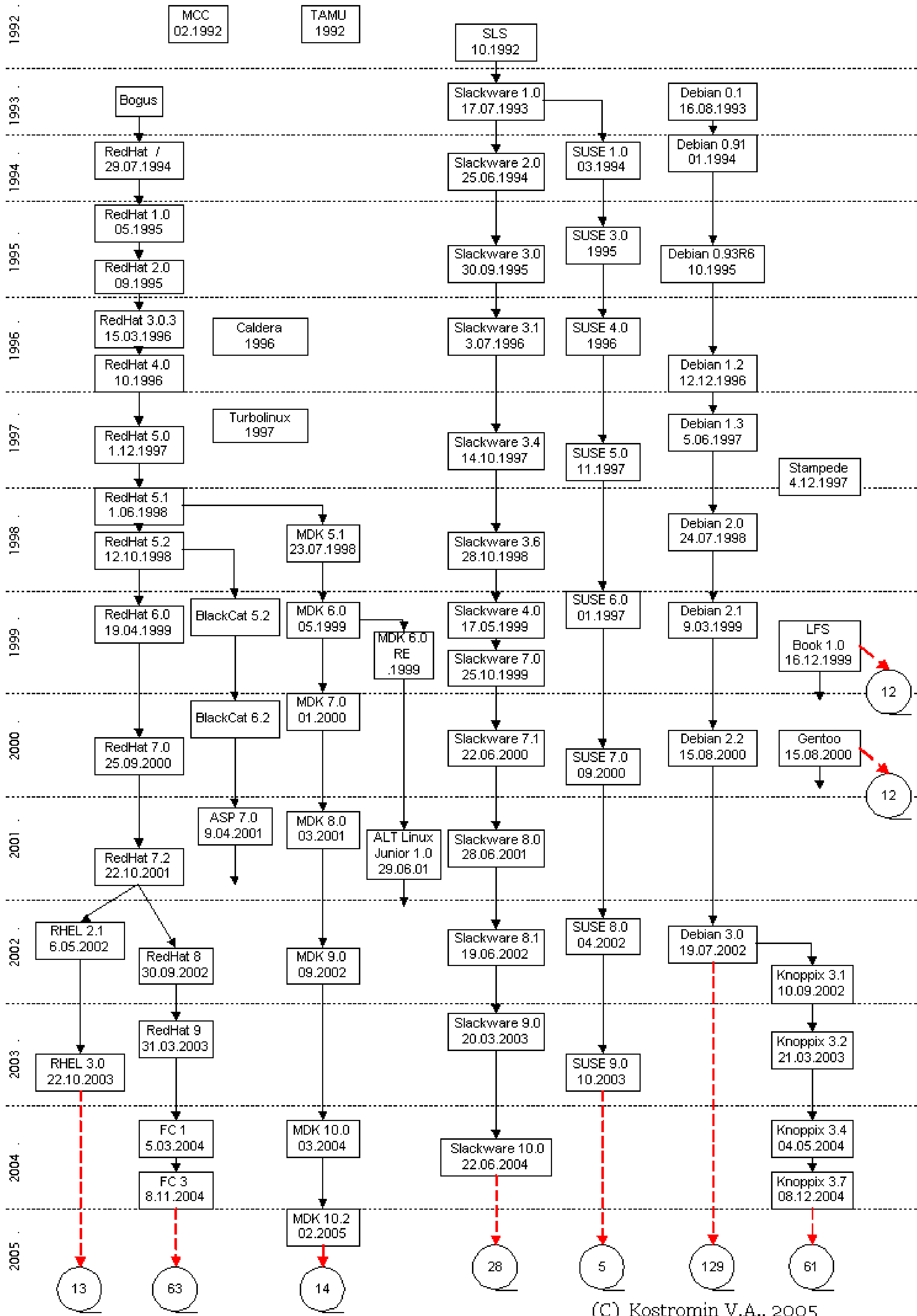
- SUSE: разработанный в Нюрнберге, Германия, SUSE (ранее SuSE) — один из наиболее популярных дистрибутивов в Европе. Клон Slackware, очень далеко отошедший от начального прототипа. Он содержит уникальную конфигурационную утилиту YaST. 4 ноября 2003 года SUSE приобретена Novell.

Это только несколько линий из большого набора. Некоторое ограниченное (но далеко не полное) представление о соотношении дистрибутивов даёт рисунок, показанный далее (рисунок заимствован из [26], и показывает состояние дел примерно на 2005 г.).

Особенностью некоторых дистрибутивов (дистрибутивы с отключенным локальным входом root - самым показательным в этом смысле является Ubuntu и его производные), есть то, что в них запрещено открытие сеанса (терминала, текстовой консоли) от имени пользователя root. Это делается из соображений безопасности, но ничего принципиально существенно нового в работу с системой не привносит: административные команды, или целые скрипты, выполняются посредством команды sudo. Это особенность только конфигурирования, и если она вам досаждает, вы можете её сменить настройками дистрибутива. Но и это вряд ли целесообразно делать специально - при необходимости, вы просто можете обойти это ограничение, войдя в файловый менеджер (командный интерпретатор) как root, вот так:

```
$ sudo mc
# whoami
root
```

8 Отказ от использования пакетной системы инсталляций — это тоже возможная идеология своей пакетной системы: всё программное обеспечение устанавливается компиляцией исходного программного кода (Linux — система с открытыми и доступными исходными кодами всего программного обеспечения). Те дистрибутивы (а таких подавляющее большинство), которые используют ту или иную систему инсталляционных пакетов, называют: пакетные дистрибутивы.


1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
MCC
02.1992
TAMU
1992
SLS
10.1992
Bogus
Slackware 1.0
17.07.1993
Debian 0.1
16.08.1993
RedHat /
29.07.1994
Slackware 2.0
25.06.1994
SUSE 1.0
03.1994
Debian 0.91
01.1994
RedHat 1.0
05.1995
RedHat 2.0
09.1995
Slackware 3.0
30.09.1995
SUSE 3.0
1995
Debian 0.93R6
10.1995
RedHat 3.0.3
15.03.1996
Caldera
1996
Slackware 3.1
3.07.1996
SUSE 4.0
1996
RedHat 4.0
10.1996
Debian 1.2
12.12.1996
Turbolinux
1997
RedHat 5.0
1.12.1997
Slackware 3.4
14.10.1997
SUSE 5.0
11.1997
Debian 1.3
5.06.1997
Stampede
4.12.1997
RedHat 5.1
1.06.1998
MDK 5.1
23.07.1998
RedHat 5.2
12.10.1998
Slackware 3.6
28.10.1998
Debian 2.0
24.07.1998
RedHat 6.0
19.04.1999
BlackCat 5.2
MDK 6.0
05.1999
MDK 6.0
RE
.1999
Slackware 4.0
17.05.1999
SUSE 6.0
01.1997
Debian 2.1
9.03.1999
LFS
Book 1.0
16.12.1999
Slackware 7.0
25.10.1999
12
BlackCat 6.2
MDK 7.0
01.2000
RedHat 7.0
25.09.2000
Slackware 7.1
22.06.2000
SUSE 7.0
09.2000
Debian 2.2
15.08.2000
Gentoo
15.08.2000
12
ASP 7.0
9.04.2001
MDK 8.0
03.2001
ALT Linux
Junior 1.0
29.06.01
Slackware 8.0
28.06.2001
RedHat 7.2
22.10.2001
RHEL 2.1
6.05.2002
RedHat 8
30.09.2002
MDK 9.0
09.2002
Slackware 8.1
19.06.2002
SUSE 8.0
04.2002
Debian 3.0
19.07.2002
Knoppix 3.1
10.09.2002
RedHat 9
31.03.2003
Slackware 9.0
20.03.2003
RHEL 3.0
22.10.2003
SUSE 9.0
10.2003
Knoppix 3.2
21.03.2003
FC 1
5.03.2004
MDK 10.0
03.2004
Slackware 10.0
22.06.2004
Knoppix 3.4
04.05.2004
FC 3
8.11.2004
Knoppix 3.7
08.12.2004
MDK 10.2
02.2005
13
63
14
28
5
129
61
(C) Kostromin V.A., 2005

Файловая система

Файловая система всех POSIX систем представляется иерархией **единого дерева** от корня, корень имеет имя `/`. Любой объект (каталог, файл, устрйство, ...) в файловой системе имеет своё **путевое имя**. Путевое имя объекта может быть указано как **абсолютное** — от корня файловой системы, или как **относительное** — относительно текущего рабочего каталога (посмотреть текущий каталог можно командой `pwd`, а сменить — `cd`). Пример:

```
$ ls -l /boot/vmlinuz-2.6.37.3
-rw-r--r-- 1 root root 7612704 Мар 13 19:37 /boot/vmlinuz-2.6.37.3
```

- здесь указано абсолютное имя файла загрузочного образа операционной системы. А далее указывается имя этого же (что видно по характеристикам файла) объекта в форме относительного имени:

```
$ cd /boot
/boot
$ pwd
/boot
$ ls -l vmlinuz-2.6.37.3
-rw-r--r-- 1 root root 7612704 Мар 13 19:37 vmlinuz-2.6.37.3
```

Важной отличительной особенностью UNIX файловой системы есть то, что в путевых именах большие и малые литеры считаются совершенно разными, поэтому:

```
$ touch _XXX
$ touch _xxx
$ ls -l _*
-rw-rw-r-- 1 olej olej 0 Июл 31 16:53 _xxx
-rw-rw-r-- 1 olej olej 0 Июл 31 16:53 _XXX
```

Это созданы в одном каталоге два совершенно разных файла!

В файловой системе UNIX очень широко используются ссылки: синонимы для имени объекта, имя, ссылающееся на другое имя. Из-за этого возникают далеко идущие последствия (не очевидные для пользователей с привычками из других систем), вот некоторые из них:

- у одного и того же объекта (файла) может быть сколь угодно много различающихся имён;
- но в системе не может быть двух объектов с точно совпадающими абсолютными их именами;
- из-за ссылок очень трудно (или неоднозначно) интерпретировать многие понятия, например: объём дискового пространства, занимаемого файлами текущего каталога...
- ссылки могут создавать циклические файловые структуры (это не ошибка, а нормальное явление) — это необходимо учитывать при планировании рекурсивных алгоритмов обхода деревьев файловой системы;

Наглядный пример ссылочности, который вы найдёте в любой инсталляции Linux:

```
$ ls -l /boot/vmlinuz*
lrwxrwxrwx 1 root root      22 Май 26 01:10 /boot/vmlinuz -> /boot/vmlinuz-2.6.37.3
-rwxr-xr-x 1 root root 3652704 Дек  1  2010 /boot/vmlinuz-2.6.32.26-175.fc12.i686.PAE
-rwxr-xr-x 1 root root 3645024 Мар  3  2010 /boot/vmlinuz-2.6.32.9-70.fc12.i686.PAE
-rw-r--r-- 1 root root 7612704 Мар 13 19:37 /boot/vmlinuz-2.6.37.3
```

Ссылки в Linux согут быть жёсткими (hard) и мягкими (soft), главное различие между ними в том, что первые могут ссылаться только на имена в пределах поддерева, размещённого на одном физическом устройстве хранения (диске), а вторые — на произвольное имя во всем дереве файловой системы. Мягкие ссылки появились исторически позже жёстких, и на сегодня гораздо более применимы. Но об этом позже...

Корневые каталоги

Основные каталоги корневого уровня файловой системы Linux:

```
$ ls /
```

```
bin   dev  home  lost+found  misc  net  proc  sbin     srv  tmp  var
boot  etc  lib   media       mnt   opt  root  selinux  sys  usr
```

Не всякий объект, который имеет имя в файловой системе UNIX является файлом (или каталогом как частным видом файла), многие объекты, именованные в дереве файловой системы, файлами не являются, а отображают некоторые логические сущности, модели, представляемые своими путевыми именами. Это один из главных и самых ранних принципов UNIX: «все сущности, что ни есть — представляются путевыми именами в файловой системе». Множественные примеры объектов, не являющихся файлами, дают нам:

- имена устройств в каталоге `/dev` : все имена здесь являются именами устройств, но никак не файлов, с ними выполняется совсем лругой набор операций
- имена псевдофойлов в какталогах `/proc` и `/sys` : вся иерархия имён здесь файлами не является, хотя, в отличие от предыдущего примера, над каждым именем здесь можно выполнять операции как над файлами (читать, писать, ...);

Хорошим подтверждением сказанному является наблюдение состояния файловой системы Linux, но не при загруженной системе (например, при загрузке с Live CD): на диске не будет никаких каталогов `/dev` , `/proc` , или `/sys`; в некоторых POSIX OS (QNX 6) не будет даже каталога `/bin` с командами-утилитами — таким логическим отображением решаются задачи построения пакетных систем.

Примечание: Всё таки не до конца все понятия в UNIX отображаются в имена файловой системы: нет, например, имени, соответствующего манипулятору мышь, которое можно было бы просто читать-писать операциями последовательного доступа... Но эту идею до идеального соответствия её действительности довели авторы первоначальной UNIX системы (из Bell Labs.) в своей последующей операционной системе Plan 9.

Назначение каталогов корня файловой системы UNIX (показанных в примере выше), при всей их многофункциональности[9], укрупнённо можно охарактеризовать в Linux так:

`/boot` – загрузочный каталог, содержит образ системы и, возможно, образ загрузочной файловой системы, и всё, что относится к загрузке (мультизагрузчик `grub` и его меню); часто размещается на отдельном физическом разделе диска.

`/etc` — каталог конфигураций (текстовых файлов конфигураций) всех подсистем (как при загрузке самой системы, так и при старте этих подсистемы).

`/dev` – каталог устройств.

`/proc` — каталог системных файлов (псевдофайлов).

`/sys` — более поздняя подсистема диагностики и управления системы, во многом то же, что и `/proc`.

`/usr` – каталог пользовательского программного обеспечения, часто сюда (или в подкаталог `/usr/local`) устанавливаются программные пакеты.

`/opt` — эквивалент `/usr` в некоторых операционных системах (Sun Solaris, Open Solaris, QNX) для умалчиваемой установки программ; сюда же могут по умолчанию устанавливаться и в Linux разнообразные программные от сторонних производителей[10] (например: `/opt/google/ chrome`, `/opt/cisco-vpnclient`, `/opt/VirtualBox`, `...`), вы и сами может устанавливать свои проекты сюда — на последнее время это начинает считаться хорошей тенденцией.

`/home` – домашние каталоги пользователей (всех ординарных пользователей, кроме пользователя `root`), здесь же будут накапливаться все рабочие файлы пользователей[11], поэтому этот каталог также имеет смысл размещать на отдельном физическом разделе диска (на случай разрушения, да и просто переустановки системы).

`/root` — домашний каталог пользователя `root`.

---

9 Всё это очень изменчиво: например, уже в мае 2011г. (версия ядра 2.6.39) был введен новый каталог корневого уровня `/run`, перенесенный сюда из `/var/run`: учёт PID запущенных программ и служб.

10 Часто говорят так: в `/opt` устанавливаются программные пакеты, не входящие в дистрибутивы (в репозитарии дистрибутивов), в этом смысле — это удачное и попадающее под определение место для установки собственного (ведомственного, узко специального, целевого ...) программного обеспечения.

11 Если не указано иное место, что может быть сделано администратором при создании учётной записи нового пользователя.

`/var` — каталог данных системы, важнейшим его подкаталогом является `/var/log` — каталог системных журналов.

Важные системные файлы

Конечно, все файлы важны в системе. Но толко системных файлов в своём Linux вы найдёте тысячи. Поэтому упомним кратким списком только тот мизерный набор системных файлов, с которыми сталкивается именно программист разработчик, и сталкивается каждодневно...

Примечание: Поскольку это информация справочная, не ищите никакого скрытого смысла ни в порядке рассмотрения каталогов, ни в порядке представления имён в них.

Конфигурации (`/etc`)

Конфигурации POSIX систем, как уже отмечалось ранее — текстовые файлы, и в этом их огромное преимущество.

Файл описания постоянно (при загрузке) смонтированных устройств системы - `/etc/fstab`, будет показан позже, при рассмотрении монтирования устройств.

Список и параметры локальных каталогов файловой системы, разделяемых через сетевую файловую систему NFS:

```
$ cat /etc/exports
/home/olej                    192.168.1.0/24(rw,sync,no_root_squash)
/home/olej                    192.168.0.0/16(rw,sync,no_root_squash)
```

Порядок (последовательность) использования механизмов разрешения сетевых имён:

```
$ cat /etc/host.conf
order hosts,bind
```

- в этом примере: сначала локальный файл `/etc/hosts`, а затем (при неудаче) DNS;

Сам файл разрешения имён сетевых хостов:

```
$ cat /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1       localhost.localdomain   localhost       home
192.168.1.9     notebook notebook.localdomain
192.168.1.7     home home.localdomain
192.168.1.5     smp smp.localdomain
192.168.1.3     rtp rtp.localdomain
192.168.1.8     minix minix.localdomain
192.168.1.1     adsl gate
192.168.1.20    wifi wifi.localdomain
::1     localhost6.localdomain6 localhost6
```

Символьные имена сетевых служб, используемые ними транспортные протоколы (TCP, UDP, ...) и порты:

```
$ cat /etc/services
...
echo            7/tcp
echo            7/udp
...
daytime         13/tcp
daytime         13/udp
```

```
...
ftp             21/tcp
ftp             21/udp          fsp fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp                          # SSH Remote Login Protocol
telnet          23/tcp
telnet          23/udp
...
```

Файлы /etc/passwd и /etc/shadow — учёт пользователей системы, будут показаны детально далее, в командах работы с пользователями системы.

Общей тенденцией есть то, что при развитии и усложнении конфигураций той или иной подсистемы, очень часто относящийся к такой подсистеме конфигурационный файл вида zzz.conf переходит со временем в каталог вида zzz.d, а все входящие в каталог конфигурационные файлы последовательно читаются и включаются в конфигурацию подсистемы zzz.

Конфигурации пакетного менеджера yum:

```
$ ls /etc/yum*
/etc/yum.conf  /etc/yumex.conf  /etc/yumex.profiles.conf
/etc/yum:
pluginconf.d  yum-updatesd.conf
/etc/yum.repos.d:
adobe-linux-i386.repo  CentOS-Media.repo  epel-testing.repo  livna.repo
CentOS-Base.repo       epel.repo          fedora10.repo      planetccrma.repo
```

Конфигурации сетевого суперсервера xinetd:

```
$ ls /etc/xinet*
/etc/xinetd.conf
/etc/xinetd.d:
chargen-dgram   daytime-stream  echo-stream   klogin       rsync
chargen-stream  discard-dgram   eklogin       krb5-telnet  tcpmux-server
cvs             discard-stream  ekrb5-telnet  kshell       time-dgram
daytime-dgram   echo-dgram      gssftp        ktalk        time-stream
```

Примечание: суперсервер xinetd пришёл на смену реализации inetd, который и на сегодня широко используется в некоторых POSIX системах, и в малых и встраиваемых конфигурациях, конфигурации inetd записываются в файл /etc/inetd.conf.

Информация о состояниях (/proc и /sys)

Интерфейс к фаловым именам /proc (procfs) и более поздний интерфейс к именам /sys (sysfs) рассматривается как канал передачи диагностической (из) и управляющей (в) информации для модуля. Такой способ взаимодействия с модулем может полностью заменить средства вызова ioctl() для устройств, который устаревший и считается опасным.

Каталог /proc содержит, в первую очередь, множество подкаталогов вида /proc/<#>, где # - это PID исполняющегося процесса; в этих каталогах содержится системная информация времени выполнения о всех процессах с соответствующими им PID.

Другие файлы каталога /proc содержат (по чтению) диагностическую информацию о разных аспектах системы, например, информация о процессоре:

```
$ cat /proc/cpuinfo
```

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 8
model name      : Celeron (Coppermine)
stepping        : 3
cpu MHz         : 534.573
cache size      : 128 KB
...
flags           : fpu vme de pse tsc msr pae mce cx8 mtrr pge mca cmov pat pse36 mmx fxsr sse up
bogomips        : 1069.76
```

Информация о устройствах (дополняет `/dev`):

```
$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
...
Block devices:
  1 ramdisk
  2 fd
  3 ide0
  8 sd
  9 md
 33 ide2
...
```

Информация о линиях прерываний и, главное, счётчики обслуженных прерываний:

```
$ cat /proc/interrupts
           CPU0
  0:   22179473          XT-PIC  timer
  1:      38326          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  5:        158          XT-PIC  uhci_hcd:usb1, CS46XX
  6:          3          XT-PIC  floppy
  7:          0          XT-PIC  parport0
  8:          1          XT-PIC  rtc
  9:          0          XT-PIC  acpi
 11:    1394028          XT-PIC  ide2, eth0, mga@pci:0000:01:00.0
 12:     288594          XT-PIC  i8042
 14:         38          XT-PIC  ide0
NMI:          0
LOC:          0
ERR:          0
MIS:          0
```

Информация о каналах DMA:

```
$ cat /proc/dma
 2: floppy
 4: cascade
```

Детальная информация динамического распределителя памяти:

```
$ cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount>
<sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
...
anon_vma            2100   2286     12  254    1 : tunables  120   60    8 : slabdata      9      9      0
...
size-256             420    420    256   15    1 : tunables  120   60    8 : slabdata     28     28      0
size-128(DMA)          0      0    128   30    1 : tunables  120   60    8 : slabdata      0      0      0
size-128            2310   2310    128   30    1 : tunables  120   60    8 : slabdata     77     77      0
size-64(DMA)           0      0     64   59    1 : tunables  120   60    8 : slabdata      0      0      0
size-32(DMA)           0      0     32  113    1 : tunables  120   60    8 : slabdata      0      0      0
size-64             1182   1357     64   59    1 : tunables  120   60    8 : slabdata     23     23      0
size-32             3336   3390     32  113    1 : tunables  120   60    8 : slabdata     30     30      0
kmem_cache           141    153    224   17    1 : tunables  120   60    8 : slabdata      9      9      0
```

Информация о загруженных модулях ядра:

```
$ cat /proc/modules
mga 62145 3 - Live 0xd0b63000
drm 65493 4 mga, Live 0xd0b52000
cisco_ipsec 601788 0 - Live 0xd0c01000 (PU)
ne2k_pci 14625 0 - Live 0xd0ae0000
...
scsi_mod 134605 4 sg,usb_storage,libata,sd_mod, Live 0xd0870000
ext3 123593 1 - Live 0xd0895000
jbd 56553 1 ext3, Live 0xd0861000
uhci_hcd 25421 0 - Live 0xd0846000
ohci_hcd 23261 0 - Live 0xd0819000
ehci_hcd 33357 0 - Live 0xd083c000
```

Список всех символьных имён загруженного ядра Linux:

```
$ cat /proc/kallsyms | head -n10
c04011f0 T _stext
c04011f0 t run_init_process
c04011f0 T stext
c040122c t init_post
c04012e7 t rest_init
c0401308 t try_name
c0401485 T name_to_dev_t
c04016cc T calibrate_delay
c04019b0 T hard_smp_processor_id
c04019c0 t target_cpus
...
```

- в этом файле порядка 85 000 строк, поэтому пользоваться ним есть смысл только с некоторыми фильтрами отбора. Эта информация крайне нужна разработчикам модулей ядра, драйверов.

Существует распространённое заблуждение (мне приходилось не раз выслушивать), что файловая система `/proc` — для чтения (диагностики), а `/sys` — и для чтения и для записи. Это принципиально неверно! В файловой системе `/proc` есть множество псевдоимён, которые отображают определённые настроечные параметры системы, которые можно изменять, управляя «на ходу» конфигурированием системы. Особенно интересен в этом смысле каталог `/proc/sys` — там великое множество параметров конфигурирования системы. Например (обратите внимание на знаки приглашения ввода командной строки — права выполнения операций):

```
$ cat /proc/sys/vm/swappiness
60
# echo 10 > /proc/sys/vm/swappiness
$ cat /proc/sys/vm/swappiness
```

```
10
```

- вот таким образом мы можем изменить процент **свободной** RAM (с 60% до 10%), при которой система начинает виртуализировать страницы физической памяти в свап-пространство на диске, таким показанным изменением мы можем значительно поднять производительность рабочей станции (но не сервера). А вот таким образом:

```
$ cat /proc/sys/vm/vfs_cache_pressure
100
# echo 1000 > /proc/sys/vm/vfs_cache_pressure
$ cat /proc/sys/vm/vfs_cache_pressure
1000
```

- мы изменяем (увеличиваем) размер кэша файловой системы, а если мы используем твердотельный SDD диск, то удачным значением было бы, напротив:

```
# echo 50 > /proc/sys/vm/vfs_cache_pressure
```

Каталог `/proc/sys/net`, например, содержит всё множество переменных, определяющих в ядре работу сетевой подсистемы, большинство этих параметров доступны и по чтению и по записи, что позволяет реконфигурировать сетевую подсистему «на лету». Эти переменные собраны в каталоги по функциональной принадлежности:

```
$ ls -l /proc/sys/net
dr-xr-xr-x 0 root root 0 Сен 27 23:32 bridge
dr-xr-xr-x 0 root root 0 Сен 27 23:32 core
dr-xr-xr-x 0 root root 0 Сен 27 23:32 ipv4
dr-xr-xr-x 0 root root 0 Сен 27 12:55 ipv6
dr-xr-xr-x 0 root root 0 Сен 27 23:32 netfilter
-rw-r--r-- 1 root root 0 Сен 27 23:32 nf_conntrack_max
dr-xr-xr-x 0 root root 0 Сен 27 23:32 unix
```

Как пример использования таких переменных, команда:

```
$ echo 1 > proc/sys/net/ipv4/ip_forward
```

- включает функцию форвардинга (передачу транзитных пакетов между сетевыми интерфейсами), а команда:

```
$ echo 1 > proc/sys/net/ipv4/ip_forward
```

- его, соответственно, отключает; эта возможность позволяет компьютеру выступать в роли брандмауэра или маршрутизатора, эта переменная очень важна для NAT(Network Address Translation - Трансляция Сетевых Адресов). В этом каталоге сосредоточено управление практически всеми сетевыми параметрами, их настолько много, что относительно этих возможностей существует отдельное руководство [27].

Детальное назначение десятков параметров в `/proc` — это отдельная интереснейшая тема для изучения, но углубление в неё увело бы нас очень далеко от наших намерений. Разработчики модулей ядра (и, возможно, вы как разработчик модулей ядра) могут добавлять произвольно свои собственные имена в `/proc` для диагностики или управления внутренними состояниями модуля, но сейчас для этой цели они чаще используют систему `/sys`.

Файловая система /sys

Файловая система `/sys (sysfs)` появилась существенно позже системы `/proc` (фактически, в полной мере, только начиная с ядра Linux 2.6). Файловая система `/sys` возникла первоначально из нужды поддерживать последовательность действий в динамическом управлении электропитанием (иерархия устройств при включении-выключении) и для поддержки горячего подключения устройств (то есть в обеспечение последнего пункта перечисления). Но позже модель оказалась гораздо плодотворнее.

По функциональности и использованию `/sys` сильно напоминает `/proc`. В настоящее время сложилась тенденция многие управляющие функции переносить их `/proc` в `/sys`, отображения путевых имён модулем в эти две подсистемы по своему назначению и возможностям являются очень подобными. Содержимое имён-псевдофайлов в обоих системах является только **текстовым** отображением некоторых внутренних данных ядра. Но нужно иметь в виду и ряд отличий между ними:

- Файловая система /proc является общей, «родовой» принадлежностью всех UNIX систем (Free/Open/Net BSD, Solaris, QNX, MINIX 3, ...), её наличие и общие принципы использования оговариваются стандартом POSIX 2; а файловая система /sys является сугубо Linux «изоретением» и используется только этой системой.
- Так сложилось по традиции, что немногочисленные диагностические файлы в /proc содержат (часто) зачастую большие таблицы текстовой информации, в то время, как в /sys создаётся много больше по числу имён, но каждое из них даёт только информацию об ограниченном значении, часто соответствующем одной элементарной переменной языка C: `int, long, ...`

Но последнее утверждение — это не правило, а тенденция; сравните:

```
$ cd /sys/class/net/eth0
$ cat address
00:15:60:c4:ee:02
$ cat flags
0x1003
$ cat uevent
INTERFACE=eth0
IFINDEX=2
```

Данные и журналы (/var)

Главный журнал системы /var/log/messages, доступен для доступа только с правами root (пример показан с sudo):

```
$ sudo cat /var/log/messages | tail -n5
Mar 19 10:01:54 localhost xinetd[4249]: START: telnet pid=5325 from=192.168.1.9
Mar 19 10:02:18 localhost xinetd[4249]: EXIT: telnet status=1 pid=5325 duration=24(sec)
Mar 19 10:26:45 localhost xinetd[4249]: START: daytime-stream pid=0 from=192.168.1.9
Mar 19 10:26:57 localhost xinetd[4249]: START: echo-stream pid=5459 from=192.168.1.9
Mar 19 10:30:57 localhost xinetd[4249]: EXIT: echo-stream status=0 pid=5459 duration=240(sec)
```

Для просмотра системного журнала имеется команда, не требующая привилегий администратора, формат вывода той же информации несколько отличается:

```
$ dmesg | tail -n 2
audit(:0): major=340 name_count=0: freeing multiple contexts (16)
audit: freed 16 contexts
```

Журнал установки программного обеспечения в системе пакетным менеджером yum:

```
$ cat /var/log/yum.log | tail -n5
Nov 23 19:37:03 Installed: git - 1.5.5.6-4.el5.i386
Nov 23 19:37:04 Installed: perl-Git - 1.5.5.6-4.el5.i386
Nov 23 20:05:09 Installed: compat-libstdc++-296 - 2.96-138.i386
Nov 23 20:05:15 Installed: compat-libstdc++-33 - 3.2.3-61.i386
Мар 09 19:44:00 Installed: flash-plugin — 10.2.152.27-release.i386
```

Файлы конфигурирования сетевых репозитариев программных пакетов, которые будет использовать тот же менеджер yum для установки нового программного обеспечения:

```
$ ls -l /var/cache/yum/
drwxr-xr-x 3 root root 4096 Май 17  2010 addons
drwxr-xr-x 3 root root 4096 Фев 24  2010 adobe-linux-i386
drwxr-xr-x 3 root root 4096 Июл  5  2010 base
drwxr-xr-x 3 root root 4096 Мар  9 16:29 epel
drwxr-xr-x 3 root root 4096 Мар  9 16:31 extras
drwxr-xr-x 3 root root 4096 Фев 26  2010 fedora10
drwxr-xr-x 3 root root 4096 Дек  1  2009 livna
drwxr-xr-x 3 root root 4096 Дек  1  2009 planetccrma
drwxr-xr-x 3 root root 4096 Сен 22  2009 planetcore
-rw-r--r-- 1 root root 1673 Мар  9 16:28 timedhosts.txt
```

```
drwxr-xr-x 3 root root 4096 Мар  9 16:30 updates
```

PID-ы многих запущенных в системе программ-сервисов:

```
$ ls /var/run/*.pid
/var/run/cupsd.pid  /var/run/klogd.pid  /var/run/syslogd.pid  /var/run/xinetd.pid
/var/run/gdm.pid    /var/run/sshd.pid   /var/run/xfs.pid
$ cat /var/run/xinetd.pid
4249
```

Примечание: С мая 2011г. Этот важный подкаталог предложено вынести на корневой уровень файловой системы - `/run`, поэтому в доступных дистрибутивах этот каталог будет со временем мигрировать туда.

Каталог устройств (/dev)

Здесь представлены все доступные системе логические устройства (некоторые из них соответствуют реальным физическим устройствам, другие нет). Каждое именованное устройство в Linux однозначно характеризуется двумя (байтовыми: 0...255) номерами: старшим номером (`major`) — номером отвечающим за отдельный класс устройств, и младшим номером (`minor`) — номером конкретного устройства внутри своего класса.

Система терминалов (текстовых консолей, не графических терминалов X11 !):

```
$ ls -l /dev/tty*
crw-rw-rw- 1 root tty     5,  0 Июл 31 10:42 /dev/tty
crw--w---- 1 root root    4,  0 Июл 31 10:42 /dev/tty0
crw--w---- 1 root root    4,  1 Июл 31 10:42 /dev/tty1
crw--w---- 1 root tty     4, 10 Июл 31 10:42 /dev/tty10
crw--w---- 1 root tty     4, 11 Июл 31 10:42 /dev/tty11
...
crw--w---- 1 root tty     4, 63 Авг 31 16:04 /dev/tty63
...
crw-rw---- 1 root dialout 4, 64 Авг 31 16:04 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65 Авг 31 16:04 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66 Авг 31 16:04 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67 Авг 31 16:04 /dev/ttyS3
```

Присвоение номеров (`major, minor`) для большинства устройств строго регламентирован, перечень устройств и соответствующих им номеров поддерживается в постоянно обновляемом файле `devices.txt` (ищите его в каталоге `Documentation` исходных кодов ядра Linux, если вы себе их установили). В показанном выше примере видно, что в системе Linux может быть (зарезервированы имена и номера) до 63 текстовых консолей. Здесь же присутствуют 4 устройства последовательных каналов RS-232 / RS-485, имена вида `/dev/ttyS*`. Это число (4) соответствует максимальному числу зарезервированных последовательных линий, реальное число вы можете проверить, обратившись к линии:

```
# stty < /dev/ttyS0
speed 9600 baud; line = 0;
-brkint -imaxbel
# stty < /dev/ttyS1
stty: стандартный ввод: Ошибка ввода/вывода
```

Но могут быть и устройства, которые динамически запрашивают себе номера при загрузке модуля-драйвера, из числа незанятых в системе.

Каждое именованное устройство в Linux принадлежит к категории символьного (потокового) устройства, или блочного устройства (устройства прямого доступа), это отображается 1-м ведущим символом (`c`/`b`) в выводе команды: `ls -l ...` Блочные и символьные устройства могут иметь одинаковые старшие номера (`major`), они исчисляются из разных пространств нумерации, например:

```
$ ls -l /dev/ | grep 1,
```

```
crw-rw-rw-  1 root root        1,   7 Авг 31 16:04 full
crw-rw----  1 root root        1,  11 Авг 31 16:04 kmsg
crw-r-----  1 root kmem        1,   1 Авг 31 16:04 mem
crw-rw-rw-  1 root root        1,   3 Авг 31 16:04 null
crw-rw----  1 root root        1,  12 Авг 31 16:04 oldmem
crw-r-----  1 root kmem        1,   4 Авг 31 16:04 port
brw-rw----  1 root disk        1,   0 Авг 31 16:04 ram0
brw-rw----  1 root disk        1,   1 Авг 31 16:04 ram1
brw-rw----  1 root disk        1,  10 Авг 31 16:04 ram10
...
```

Но в каждом подмножестве не может быть двух устройств с полностью идентичным набором номеров (`major/minor`).

Таким же образом (как имена в `/dev`) представлены реально существующие в оборудовании USB концентраторы (хабы, шины):

```
$ ls -l /dev/usb*
crw-rw---- 1 root root 252, 0 2011-08-31 16:04 /dev/usbmon0
crw-rw---- 1 root root 252, 1 2011-08-31 16:04 /dev/usbmon1
crw-rw---- 1 root root 252, 2 2011-08-31 16:04 /dev/usbmon2
crw-rw---- 1 root root 252, 3 2011-08-31 16:04 /dev/usbmon3
crw-rw---- 1 root root 252, 4 2011-08-31 16:04 /dev/usbmon4
crw-rw---- 1 root root 252, 5 2011-08-31 16:04 /dev/usbmon5
$ lsusb
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Устройства хранения

Один из важнейших для пользователя класс устройств — это накопители прямого доступа.

Вот как выглядят дисковые накопители на контроллере EIDE:

```
$ ls -l /dev/hd*
brw------- 1 olej disk  3,  0 Мар 24 08:01 /dev/hda
brw-r----- 1 root disk 33,  0 Мар 24 08:01 /dev/hde
brw-r----- 1 root disk 33,  1 Мар 24 08:01 /dev/hde1
brw-r----- 1 root disk 33,  2 Мар 24 08:01 /dev/hde2
brw-r----- 1 root disk 33,  5 Мар 24 08:01 /dev/hde5
brw-r----- 1 root disk 33, 64 Мар 24 08:01 /dev/hdf
brw-r----- 1 root disk 33, 65 Мар 24 08:01 /dev/hdf1
brw-r----- 1 root disk 33, 66 Мар 24 08:01 /dev/hdf2
brw-r----- 1 root disk 33, 68 Мар 24 08:01 /dev/hdf4
brw-r----- 1 root disk 33, 69 Мар 24 08:01 /dev/hdf5
brw-r----- 1 root disk 33, 70 Мар 24 08:02 /dev/hdf6
```

Примечание: Здесь специально показана «странная» конфигурация, снятая с реальной Linux системы: отдельный аппаратный Ultra DMA контроллер отображает 2 диска HDD как `/dev/hde` и `/dev/hdf`, а IDE CD-RW тогда получает отображение как `/dev/hda` (master на 1-м канале встроенного IDE контроллера). Это иллюстрирует то, что не нужно полагаться на последовательную «раскладку» дисков, которую мы чаще всего наблюдаем.

Вот как осуществляется работа (диагностика геометрии, или создание разделов) с такими устройствами:

```
$ sudo /sbin/fdisk /dev/hdf
...
Команда (m для справки): p
Диск /dev/hdf: 20.0 ГБ, 20060135424 байт
255 heads, 63 sectors/track, 2438 cylinders
Единицы = цилиндры по 16065 * 512 = 8225280 байт
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/hdf1   *           1         501     4024251   4f  QNX4.x 3-я часть
/dev/hdf2            1394        2438     8393962+   f  W95 расшир. (LBA)
/dev/hdf4             502        1393     7164990    c  W95 FAT32 (LBA)
/dev/hdf5            1394        1456      506016   82  Linux своп / Solaris
/dev/hdf6            1457        2438     7887883+  83  Linux
Пункты таблицы разделов расположены не в дисковом порядке
Команда (m для справки):
```

Вот то же относительно приводов CD/DVD :

```
$ ls -l /dev/cd*
lrwxrwxrwx 1 root root 3 Мар 24 08:01 /dev/cdrom -> hda
lrwxrwxrwx 1 root root 3 Мар 24 08:01 /dev/cdrom-hda
```

Для устройств SCSI вместо `/dev/hd*` будет `/dev/sd*`, но в соответствии с стандартами SCSI оформлены также модули-драйверы блочных устройств SATA, твердотельных накопителей (SDD), или USB флеш-накопителей:

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8,  0 Авг 31 16:04 /dev/sda
brw-rw---- 1 root disk 8,  1 Авг 31 16:04 /dev/sda1
brw-rw---- 1 root disk 8,  2 Авг 31 16:04 /dev/sda2
brw-rw---- 1 root disk 8,  3 Авг 31 16:04 /dev/sda3
brw-rw---- 1 root disk 8,  4 Авг 31 16:04 /dev/sda4
brw-rw---- 1 root disk 8,  5 Авг 31 16:04 /dev/sda5
brw-rw---- 1 root disk 8,  6 Авг 31 16:04 /dev/sda6
brw-rw---- 1 root disk 8,  7 Авг 31 16:04 /dev/sda7
brw-rw---- 1 root disk 8, 16 Авг 31 16:30 /dev/sdb
brw-rw---- 1 root disk 8, 17 Авг 31 16:30 /dev/sdb1
```

- здесь `/dev/sda` — это SATA накопитель, а `/dev/sdb` — это съёмный USB-диск.

```
$ sudo fdisk /dev/sdb
...
Команда (m для справки): p
Диск /dev/sdb: 1031 МБ, 1031798272 байт
64 heads, 32 sectors/track, 983 cylinders
Units = цилиндры of 2048 * 512 = 1048576 bytes
Disk identifier: 0x00000000
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/sdb1               1         983     1006576    b  W95 FAT32
```

А вот как выглядит картина на другом компьютере с твердотельным SDD диском SATA:

```
$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Июн 16 11:03 /dev/sda
brw-rw---- 1 root disk 8, 1 Июн 16 11:04 /dev/sda1
brw-rw---- 1 root disk 8, 2 Июн 16 11:03 /dev/sda2
brw-rw---- 1 root disk 8, 3 Июн 16 11:03 /dev/sda3
$ sudo fdisk -l
```

```
Диск /dev/sda: 30.0 ГБ, 30016659456 байт
255 heads, 63 sectors/track, 3649 cylinders
Units = цилиндры of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00073858
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/sda1   *           1        3494    28056576   83  Linux
/dev/sda2            3494        3650     1254401    5  Расширенный
/dev/sda5            3494        3650     1254400   82  Linux своп / Solaris
```

Примечание: На настоящий момент применение в Linux в качестве системного диска твердотельного SDD — очень неплохая идея: ёмкости 30Gb более чем достаточно для любых системных нужд Linux, при стоимости такого диска порядка $80, но при его скорости на порядки превышающей HDD, загрузка Ubuntu 10.04.3 LTS в показанной выше конфигурации требует порядка 20 секунд от включения питания до регистрации пользователя, и порядка 8 секунд для восстановления графического десктопа GNOME после регистрации. В системе, размещённой на HDD на это требуется на порядок больше времени.

Но накопители на SD-карте уже будут представляться совсем по-другому (они поддерживаются совсем иным модулем-драйвером):

```
$ ls -l /dev/mm*
brw-rw---- 1 root disk 179, 0 Июл 31 10:42 /dev/mmcblk0
brw-rw---- 1 root disk 179, 1 Июл 31 10:42 /dev/mmcblk0p1
```

Блочные устройства `/dev/hd*` и `/dev/sd*` в каталоге `/dev` представляются как последовательный сырой (raw) поток байт — это ещё одна достопримечательность UNIX, сильно удивляющая пришельцев из других операционных систем. Поэтому могут копироваться (чем создаются загрузочные копии существующих разделов диска) целые диски:

```
# cp /dev/hdb /dev/hdc
```

... или их отдельные разделы (partition):

```
# cp /dev/hdb1 /dev/hdc3
```

Каталог загрузки (`/boot`) и коротко о загрузке

Возможный вид каталога `/boot` (Fedora 12):

```
$ ls /boot
config-2.6.32.9-70.fc12.i686.PAE
System.map-2.6.32.9-70.fc12.i686.PAE
grub
initramfs-2.6.32.9-70.fc12.i686.PAE.img
vmlinuz-2.6.32.9-70.fc12.i686.PAE
```

Другой вариант (CentOS 5.2):

```
$ ls /boot
config-2.6.18-92.el5
vmlinuz-2.6.18-92.el5
grub
initrd-2.6.18-92.el5.img
System.map-2.6.18-92.el5
```

В обоих (сильно различающихся) системах весь набор файлов системы имеет одинаковый суффикс (вида показанного — `2.6.32.9-70.fc12.i686.PAE`, обозначим его * в нашем изложении), набор файлов, относящихся к одному ядру, имеет фиксированный состав (4 основных файла):

`config-*` - текстовый файл конфигурационных параметров, при которых собрано текущее ядро, обычно он

используется как отправная точка для последующих изменений в конфигурации, при новых сборках ядра.

vmlinuz-* - загрузочный файл образа системы, на этот файл конкретной версии, загружаемой по умолчанию, обычно устанавливается ссылка /boot/vmlinuz.

System.map-* - файл таблицы символов соответствующего ядра, это очень близко динамической таблице символов, формируемой в /proc/kallsyms, но это статическая таблица символов, известных на момент сборки ядра (без загружаемых позже модулей).

initrd-* или initramfs — это образ стартовой корневой файловой системы (монтируемой как / на время загрузки) в двух альтернативных форматах (их существует больше двух, но это самые используемые). Во втором показанном варианте образ корневой файловой системы представлен в виде RAM-диска initrd-* (с поддержанием иерархической файловой системы). Во втором примере образ корневой файловой системы представлен архивом формата CPIO (один из самых старых и традиционных форматов архивирования UNIX) initramfs-*, содержащим требуемые файлы просто линейным списком — это более поздний, более современный способ представления.

Если вы будете обновлять ядро (пакетным менеджером), или собирать и устанавливать новое ядро из свежих исходных кодов, то у вас в каталоге /boot появятся каждый раз новая группа файлов в том же составе, но с отличающим их **суффиксом**.

Зачем нужен образ стартовой корневой файловой системы? Система грузится загрузкой файла-образа /boot/vmlinuz. Если вы соберёте монолитное ядро, не требующее динамической загрузки модулей (и на ранних этапах система собиралась только так, и так она собирается для малых специальных конфигураций), то никакая корневая система вам вообще не нужна. Но если это не так, то ядру могут потребоваться модули для их динамической загрузки, в том числе и модули драйверов дисковых и файловых систем... Но модули хранятся как загружаемые файлы в файловой системе ... для которой, возможно, ещё нет загруженных драйверов. Возникает проблема курицы и яйца... Образ стартовой корневой файловой системы и есть тот образ небольшой файловой системы, размещаемой полностью в RAM, в которой и лежат файлы требуемых модулей ядра и, возможно, некоторые конфигурационные файлы. В конечном итоге, если вы не пересобираете ядро, то вам никогда не придётся беспокоиться о стартовой корневой системе, а если вы пересобираете ядро, то там предусмотрены средства создавать и образы стартовых файловых систем путём простых формальных действий: без глубокого понимания что и как собирается.

Довольно заметно может отличаться вид и состав каталога /boot в других дистрибутивах (особенно Debian или Ubuntu). Но и здесь описанные тенденции сохраняются, и общая картина ясна. Вот как это выглядит в инсталляции Ubuntu 10.04.3 LTS:

```
$ ls /boot
abi-2.6.32-33-generic    grub          memtest86+.bin              vmcoreinfo-2.6.32-33-generic
config-2.6.32-33-generic initrd.img-2.6.32-33-generic System.map-2.6.32-33-generic
vmlinuz-2.6.32-33-generic
```

Что такое виденный выше (во всех дистрибутивах) каталог /boot/grub? Linux давно эксплуатируется с вторичными загрузчиками, допускающими мультизагрузку (и выбор загружаемой системы из начального меню). Такие загрузчики Linux развиваются как независимые открытые проекты (независимые и от разработки ядра, и от разработки утилитного окружения GNU/FSF). Самыми известными загрузчиками являются LILO (более старый проект) и GRUB (наиболее активно применяемый на сегодня). Домашняя страница каждого из проектов легко находится в интернет для получения исчерпывающей информации. Вот в каталоге /boot/grub и находится **ограниченное подмножество** средств пакета GRUB, необходимое для реализации мультизагрузки в Linux (GRUB широко применяется в других операционных системах с другой структурой разделов диска и файловых систем, например, в: Solaris, BSD; все такие расширенные средства не включаются в /boot/grub). Вот как выглядит конфигурационный файл (меню загрузки и другое) мультизагрузчика grub:

```
$ ls -l /boot/grub/grub.conf
-rw------- 1 root root 907 Дек  3  2009 /boot/grub/grub.conf
$ ls -l /boot/grub/menu.*
lrwxrwxrwx 1 root root 11 Окт 29  2008 /boot/grub/menu.lst -> ./grub.conf
```

```
$ sudo cat /boot/grub/grub.conf
default=1
timeout=5
...
title CentOS (2.6.24.3-1.rt1.2.el5.ccrmart)
        root (hd1,5)
        kernel /boot/vmlinuz-2.6.24.3-1.rt1.2.el5.ccrmart ro root=LABEL=/ rhgb quiet
        initrd /boot/initrd-2.6.24.3-1.rt1.2.el5.ccrmart.img
...
title QNX 6.3
        rootnoverify (hd1,0)
        chainloader +1
title Windows 98SE
        rootnoverify (hd0,0)
        chainloader +1
```

В отличие от загрузчика `LILO` и других более ранних систем мультизагрузки, GRUB знает структуру файловой системы, и после редактирования `grub.conf` **не требует** какого-то специального прописывания в загрузчик диска (изменения сразу вступают в силу). Сам `grub` (программа) имеет достаточно развитую командную оболочку, что позволит вам, например, восстанавливать повреждённую загрузку с диска в диалоге с программой (которая, помимо прочего, содержит в себе обширную справочную информацию по работе с программой):

```
# which grub
/sbin/grub
# grub
Probing devices to guess BIOS drives. This may take a long time.

    GNU GRUB  version 0.97  (640K lower / 3072K upper memory)

 [ Minimal BASH-like line editing is supported.  For the first word, TAB
   lists possible command completions.  Anywhere else TAB lists the possible
   completions of a device/filename.]

grub> help
blocklist FILE                         boot
cat FILE                               chainloader [--force] FILE
clear                                  color NORMAL [HIGHLIGHT]
configfile FILE                        device DRIVE DEVICE
displayapm                             displaymem
find FILENAME                          geometry DRIVE [CYLINDER HEAD SECTOR [
halt [--no-apm]                        help [--all] [PATTERN ...]
hide PARTITION                         initrd FILE [ARG ...]
kernel [--no-mem-option] [--type=TYPE] makeactive
map TO_DRIVE FROM_DRIVE                md5crypt
module FILE [ARG ...]                  modulenounzip FILE [ARG ...]
pager [FLAG]                           partnew PART TYPE START LEN
parttype PART TYPE                     quit
reboot                                 root [DEVICE [HDBIAS]]
rootnoverify [DEVICE [HDBIAS]]         serial [--unit=UNIT] [--port=PORT] [--
setkey [TO_KEY FROM_KEY]               setup [--prefix=DIR] [--stage2=STAGE2_
terminal [--dumb] [--no-echo] [--no-ed terminfo [--name=NAME --cursor-address
testvbe MODE                           unhide PARTITION
uppermem KBYTES                        vbeprobe [MODE]
```

Монтирование файловых систем

Принцип UNIX относительно устройств прямого доступа, представленные как последовательный сырой (raw) поток байт (о чём говорилось выше), для использования должны быть **монтированы**. Монтирование предполагает, что:

– на сырую байтовую последовательность диска будет «наложена» структуру одной из (многих) известных Linux файловых систем (EXT2, EXT3, EXT4, FAT32, NTFS, UFS, ZFS и великого множества других);

– для структурированного диска будет назначено имя каталога **точки монтирования**, далее иерархия имён диска будет выглядеть в файловой системе как поддерево имён от имени точки монтирования вниз;

Наиболее употребимая форма команды монтирования:

```
# mount [-fnrsvw] [-t vfstype] [-o options] <device> <dir>
```

- где `options` - это разделенный запятыми список конкретных опций монтирования, большинство которых зависит от конкретного типа монтируемой файловой системы (ключ `-t`). Это именно тот шаблон файловой системы, который будет «наложен» на сырой поток байт дискового раздела. Установить полный перечень поддерживаемых файловых систем, а уж тем более помнить правильное синтаксическое написание их в качестве значения опции `-t` — дело весьма хлопотное. Но нам в этом поможет:

```
$ man mount
...
-t vfstype
The argument following the -t is used to indicate the file system type.  The file system types  which  are  currently  sup-
ported  include:  adfs,  affs,  autofs,  cifs,  coda,  coherent,  cramfs, debugfs, devpts, efs, ext, ext2, ext3, hfs, hpfs,
iso9660, jfs, minix, msdos, ncpfs, nfs, nfs4, ntfs, proc, qnx4, ramfs, reiserfs, romfs, smbfs, sysv, tmpfs, udf, ufs,  ums-
dos, usbfs, vfat, xenix, xfs, xiafs.  Note that coherent, sysv and xenix are equivalent and that xenix and coherent will be
removed at some point in the future — use sysv instead. Since kernel version 2.1.21 the types ext and xiafs  do  not  exist
anymore. Earlier, usbfs was known as usbdevfs.
...
```

Пример:

```
# mount -t iso9660 /dev/cdrom /mnt/cd
$ ls -l /dev/cdrom
lrwxrwxrwx 1 root root 3 Мар 31 05:15 /dev/cdrom -> hda
```

Монтирование флеш-диска:

```
# mount -t vfat /dev/sda1 /mnt/usb1
# ls /mnt/usb1
...
```

Монтирование файловой системы, вообще то говоря, операция, требующая прав `root`. Но настройками системы полномочия на монтирование отдельных устройств могут быть делегированы администратором и ординарным пользователям.

При монтировании каталог монтирования (точка монтирования) **не обязательно** должен быть пуст. В различных POSIX ОС:

– каталог монтирования должен обязательно **существовать** ранее (Linux), в других — он будет создаваться по необходимости (QNX);

– монтируемые к не пустой точке монтирование каталоги устройства «дополняются» к существующим (Solaris, QNX), а в других — **временно** (до размонтирования) «замещают» их (Linux).

Пример повторного монтирования:

```
$ sudo mkdir /new
$ touch start.start.start
$ ls
start.start.start
$ sudo mount --bind `pwd` /new
$ ls /new
start.start.start
```

```
$ sudo umount /new
$ ls /new
$ sudo rmdir /new
```

Монтирования, которые не хочется постоянно повторять, могут быть вписаны (вами) в `/etc/fstab`, чтобы они выполнялись при начальной загрузке системы:

```
$ cat /etc/fstab
# <file system>         <mount point>           <type>  <options>  <dump> <pass>
...
sysfs                   /sys                    sysfs   defaults        0 0
proc                    /proc                   proc    defaults        0 0
...
/dev/cdrom              /mnt/cdrom              iso9660 ro,user,noauto,unhide
...
/dev/hde1               /mnt/win_c              vfat    defaults        0 0
/dev/hdf4               /mnt/win_d              vfat    defaults        0 0
/dev/hde5               /mnt/win_e              vfat    defaults        0 0
```

И последнее: для того, чтобы убедиться, что вы намонтировали на текущий момент — используем команду `mount` без параметров (вот здесь `root` не нужен):

```
$ mount
/dev/mapper/vg_notebook-lv_root on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sda1 on /boot type ext3 (rw)
/dev/sda4 on /mnt/arch type ext3 (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
nfsd on /proc/fs/nfsd type nfsd (rw)
gvfs-fuse-daemon on /home/olej/.gvfs type fuse.gvfs-fuse-daemon (rw,nosuid,nodev,user=olej)
/dev/mmcblk0p1 on /media/33D6-5316 type vfat
(rw,nosuid,nodev,uhelper=devkit,uid=500,gid=500,shortname=mixed,dmask=0077,utf8=1,flush) [12]
```

12 Две последние строки вывода (`nfsd` — демон сетевой файловой системы NFS и `/dev/mmcblk0p1` — вставленная SD-карточка памяти) - разорваны из-за ограниченности ширины страницы и плохо читаются.

Командный интерпретатор

Все консольные команды в Linux обрабатываются командным интерпретатором. Командный интерпретатор является такой же рядовой программой-утилитой, как всякая другая. По умолчанию в Linux определяется интерпретатор с именем bash, но может быть использован и любой другой (много их присутствует в дистрибутиве):

```
$ ls /bin/*sh*
/bin/bash  /bin/csh  /bin/dash  /bin/ksh  /bin/sh  /bin/tcsh  /bin/zsh
```

То, какой интерпретатор использовать, определяется при создании нового имени пользователя и зафиксировано в его записи в /etc/passwd. Позже это может быть изменено.

Работа с командами системы, переменными окружения и другое - могут существенно (для интерпретатора tcsh, ksh) или в деталях (для интерпретатора zsh), в зависимости от того, какой конкретно командный интерпретатор вы используете, и даже от его версии (для интерпретатора bash). Мы в обсуждениях будем предполагать, что используется самый широко используемый (по умолчанию) в Linux интерпретатор bash, который детальнейшим образом и многократно описан [19, 20, 21]. Если же вы сменили себе интерпретатор, то сверяйтесь по деталям в справочной странице по нему. Убедиться какой у вас активный интерпретатор можно так:

```
$ echo $SHELL
/bin/bash
```

Обратите внимание:

```
$ echo $shell
$
```

- возвращается «пустое значение» (переменная окружения не определена!) - это совершенно разные переменные. Как и везде в именовании: UNIX везде различает малые и большие буквы и считает их совершенно разными.

Примечание: Интерпретатор bash специально разрабатывался так, чтобы учесть уже сложившийся на то время общий синтаксис интерпретатора shell, но и подогнать его под требования стандарта POSIX 2 (IEEE POSIX Shell and Tools specification, IEEE Working Group 1003.2: http://gopher.std.com/obi/Standards/posix/1003.2/toc).

Из-за тщательности описания в литературе (например: [1], [4], [19]) синтаксических особенностей языка bash (а это сама по себе огромная тема), я не буду нигде описывать этот синтаксис. Мы будем рассматривать только отдельные выражения команд интерпретатора там, где это касается непосредственно рассматриваемого в этой теме аспекта системы.

Если вам нужно в текущем терминале (консоли) использовать другой из доступных интерпретатор (например, выполнить скрипт в его синтаксисе), то для этого не нужно изменять умолчания для этого пользователя, просто запускаете новую оболочку:

```
$ echo $SHELL
/bin/bash
$ ksh
$
```

- здесь вы уже в интерпретаторе ksh, будьте внимательны - обратите внимание, что при этом:

```
$ echo $SHELL
/bin/bash
```

- что может позже создать противоречия.

Переменные окружения

Переменные окружения (environment) известны из разных операционных систем. Особенностью их здесь есть только их запись: переменные окружения указываются в формате:

– L-value (объявление, присвоение) : NAME
– R-value (извлечение значения) : $NAME

Сравните запись одной и той же переменной в двух разных контекстах:

```
$ XXX=123
$ echo $XXX
123
```

В записи команд интерпретатора не следует вставлять пробел до/после знака присвоения: это украшательство приводит к ошибкам.

Примечание: Вот такое, как показанное выше, простое определение переменной, или присвоение нового ей значения - является локальным, и не будет наследоваться последующими запущенными программами. Для того, чтобы сделать его глобальным, используется внутренняя команда `export`, о чём будет подробно далее.

Многие правила синтаксиса, оперирующего с переменными окружения, неявно вытекают из общих правил синтаксиса интерпретатора `bash`, но не лишне показать итоги этих правил явно, из-за частоты оперирования с переменными окружения...

Имена большинства переменных окружения записывают большими литерами. Но это только традиция и ничего более. Значением любой переменной окружения (после формирования и присвоения ей этого значения) всегда является **текстовая строка** (ещё один факт приверженности UNIX символьным представлениям), которая может содержать любые символы, и которая далее самим интерпретатором никак не интерпретируется: значение переменной возвращается запрашивающей программе, которая сама уже знает, как ей разбираться с этим значением.

Если значение для переменной может вызвать синтаксические недоразумения (значение, содержащее спецсимволы, разделительные пробелы и подобное), то такое значение заключается в литеральные кавычки (одиночные или двойные):

```
$ XXX='это составное значение'
$ echo $XXX
это составное значение
$ XXX="это ещё одно составное значение"
$ echo $XXX
это ещё одно составное значение
$ XXX="литерал включающий '"
$ echo $XXX
литерал включающий '
```

Символы со специальным смыслом могут экранироваться (предшествующим '\', снимающим специальный смысл с символа, «экранирование»):

```
$ XXX="экранируется \$ символ"
$ echo $XXX
экранируется $ символ
```

Ещё одной особенностью UNIX нотации есть то, что **списки** значений (например, путевых имён) в качестве переменных окружения записываются через двоеточие (':') в качестве разделителя отдельных значений в списке:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/olej/bin
```

Переменным окружения могут быть присвоены не только константные значения (явно записанные в

присвоении), но и результат выполнения команд (традиционный вывод команд в `SYSOUT`) и целых командных скриптов:

```
$ pwd
/etc
$ CDIR=`pwd`
$ echo $CDIR
/etc
```

Примечание: Неудачи в попытках воспроизведения такой команды часто связаны с тем вопросом, какую литеральную кавычку использовать в записи команды? Кавычек достаточно много на клавиатуре, и с подобным начертанием... В данном случае нужен тот символ, который находится на клавише '~'/'Ё' (под `Esc`).

Это был показан «старый стиль» записи результата команды в качестве значения. Существует альтернативная форма, но это только специфическое расширение `bash` (не сработает в других интерпретаторах):

```
$ CDIR=$(pwd)
```

При определении нового значения для переменной могут использоваться значения уже ранее определённых переменных, в том числе и ранее установленное значение самой этой переменной, несколько примеров тому:

```
$ XXX=123
$ XXX=$XXX:456
$ echo $XXX
123:456

$ echo $XXX
123
$ XXX=AB${XXX}ab
$ echo $XXX
AB123ab
```

- здесь потребовалось выделить (`{...}`)имя переменной, чтобы вычленить его из конкатенируемой последовательности символов.

Посмотреть текущее содержимое окружения можно разнообразными способами, и в разных форматах, и последовательности:

```
$ export
declare -x COLORTERM="gnome-terminal"
...
declare -x TERM="xterm"
declare -x USER="olej"
...
$ env
ORBIT_SOCKETDIR=/tmp/orbit-olej
HOSTNAME=notebook.localdomain
IMSETTINGS_INTEGRATE_DESKTOP=yes
SHELL=/bin/bash
TERM=xterm
...
USERNAME=olej
...
CLASSPATH=.
DISPLAY=:0
...
$ set
BASH=/bin/bash
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
...
```

Вообще то, чаще бывает нужно не только определить новую переменную (или присвоить переменной новое значение), но и обеспечить её **экспортирование в среду** следующих выполняющихся команд. Это делается по-разному в зависимости от вида используемого командного интерпретатора. В интерпретаторе `bash` встроенная команда `export` помечает имена переменных для автоматического экспортирования в среду следующих выполняемых команд. Именно такое определение (а не простое присвоение значения) является основной практикой управления переменными окружения. Вот как могут выглядеть альтернативные варианты добавления текущего каталога (или любого другого) в список путей переменной PATH:

```
$ PATH=./:$PATH
$ export PATH
```

Или:

```
$ PATH=./:$PATH ; export PATH
```

Или просто:

```
$ export PATH=./:$PATH
$ echo $PATH
./:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/olej/bin
```

Примечание: Обратите внимание, что вот предыдущая запись и следующая запись будут иметь совершенно разный эффект (в качестве результата `pwd` будет вставлено абсолютное имя текущего каталога, а не обозначение «текущий каталог»):

```
$ export PATH=`pwd`:$PATH
```

Некоторые важные переменные

Некоторые из этих переменных устанавливаются самой системой, некоторые — совершенно специфическими программными пакетами, некоторые требуют установить от пользователя вручную программные пакеты для своего более комфортного использования. Я не буду разделять эти переменные по таким группам, мы только списком перечислим те переменные, на которые стоит обратить внимание, и те, которые своим некорректным значением могут вызвать странности в системе, и привести в замешательство...

`PATH` — список путей поиска для запуска исполнимых файлов:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/olej/bin
```

`LD_LIBRARY_PATH` – список путей каталогов поиска динамических библиотек для разрешения ссылок при компиляции и выполнении программ.

`LD_RUN_PATH` — список путей каталогов поиска динамических библиотек для загрузки времени выполнения.

`CLASSPATH` – путь поиска библиотек классов для программных систем на языке Java.

`DISPLAY` — указание дисплея, на который будут отображать свой ввод-вывод все программы графической подсистемы X11, по умолчанию обычно:

```
$ echo DISPLAY
DISPLAY=:0
```

Но это может быть и удалённый сетевой терминал (об этом мы подробно поговорим позже):

```
$ echo DISPLAY
DISPLAY=192.168.2.2:0.0
```

Прокси-переменные для утилиты `wget` и многих подобных программ обмена, выходящих во внешнюю сеть:

`http_proxy` - эта переменная окружения содержит URL сервера прокси для протокола HTTP, формат

переменной: `http_proxy=http://<login>:<password>@<ip-proxy>:3128`, например:

```
$ export http_proxy=http://olej:12345@192.168.2.2
```

`ftp_proxy` - эта переменная окружения содержит URL сервера прокси для протокола FTP. Достаточно общим является то, что стандартные переменные окружения `http_proxy` и `ftp_proxy` содержат один и тот же URL (но должны быть установлены при этом обе переменных).

`no_proxy` - эта переменная должна содержать разделенный **запятыми** список доменов, для которых сервер прокси не должен использоваться.

Встроенные переменные

Это внутренние переменные командного интерпретатора shell, который и устанавливает им значения. Имя такой переменной вне контекста получения его значения (R-value) не имеет смысла (не существует переменной `?`, имеет смысл лишь её значение `$?`). Имена таких переменных состоят из одного символа, некоторые из них:

`$1, $2, ... $9` — позиционные параметры скрипта;

`$#` - число позиционных параметров скрипта;

$`?` - код возврата последнего выполненного процесса;

`$$` - PID текущего shell;

`$!` - PID последнего процесса, запущенного в фоновом режиме (background, команды `bg` и `fg`);

Большинство таких переменных активно используется при написании скриптов `bash`. Но код возврата часто бывает полезным и для работы с консоли:

```
$ echo $?
0
$ cat /var/log/messages
cat: /var/log/messages: Отказано в доступе
$ echo $?
1
```

Консольные команды

Большинство консольных команд — это имена соответствующих программ-утилит (если таковые файлы программ найдены на путях поиска `$PATH`), но есть некоторая часть команд, которые являются внутренними командами интерпретатора `bash` (как и для других интерпретаторов). Полный их список команда, которые реализованы как внутренние, можно получить, указав в команде `man` **любую** из внутренних команд:

```
$ man set
BASH_BUILTINS(1)                                              BASH_BUILTINS(1)
NAME
   bash,  :,  .,  [,  alias, bg, bind, break, builtin, cd, command, compgen, complete, continue, declare, dirs, disown, echo, enable,
   eval, exec, exit, export, fc, fg, getopts, hash, help, history, jobs, kill, let, local, logout, popd, printf,  pushd,  pwd,  read,
   readonly, return, set, shift, shopt, source, suspend, test, times, trap, type, typeset, ulimit, umask, unalias, unset, wait - bash
   built-in commands, see bash(1)
...
    cd [-L|-P] [dir]
           Change the current directory to dir.
...
```

Эта же справочная страница даст вам подробные сведения об использовании каждой из внутренних команд интерпретатора.

Примечание: Обратите внимание, что справка по самому интерпретатору даст вам совсем другую страницу, а не ту, которую мы здесь обсуждаем:

```
$ man bash
BASH(1)                                                                BASH(1)
NAME
       bash - GNU Bourne-Again SHell
SYNOPSIS
       bash [options] [file]
COPYRIGHT
       Bash is Copyright (C) 1989-2009 by the Free Software Foundation, Inc.
...
```

Все прочие, не являющиеся внутренними, как уже сказано, являются именем **файла** программы, реализующей эту команду, если этот файл находится в одном из каталогов, перечисленных списком в переменной `$PATH`:

```
$ which cp
/bin/cp
```

Если для всех внутренних команд мы видели единую справочную страницу, то для внешних команд, естественно, они будут индивидуальными:

```
$ man cp
CP(1)                                                                    CP(1)
ИМЯ
       cp - копирование файлов и каталогов
ОБЗОР
       cp [опции] файл путь
       cp [опции] файл... каталог
      Опции POSIX: [-fiprR] [--]
      Дополнительные опции POSIX 1003.1-2003: [-HLP]
...
```

Примечание: Обратите внимание (когда-то это может стать сюрпризом), что имена некоторых внешних команд (файлов) может перекрываться такой же внутренней командой конкретного интерпретатора команд (а для другого интерпретатора такое перекрытие может и отсутствовать). Самый яркий тому пример:

```
$ which echo
/bin/echo
$ echo --help
```

```
--help
$ /bin/echo --help
Usage: /bin/echo [SHORT-OPTION]... [STRING]...
  or:  /bin/echo LONG-OPTION
Печатает СТРОКУ(СТРОКИ) на стандартный вывод.
...
ЗАМЕЧАНИЕ: ваша оболочка может предоставлять свою версию echo, которая
обычно перекрывает версию, описанную здесь.  Пожалуйста, обращайтесь к
документации по вашей оболочке, чтобы узнать, какие ключи она
поддерживает.
```

И ещё одно несоответствие в этом случае — очевидно, что это справочная страница «не той» команды, которая используется в системе по умолчанию:

```
$ which echo
ECHO(1)                          User Commands                          ECHO(1)
NAME
       echo - display a line of text
...
```

Формат командной строки

Формат командной строки определяет как, с какими параметрами и опциями, может быть записан вызов программ утилит. Детально формат определяется используемым **командным интерпретатором**, и даже его версией, но общие правила сохраняются:

```
$ <[путь/]команда> [ключи] [параметры] [ключи] [параметры]...
```

Порядок следования [ключи] [параметры] чаще всего произвольный, но в некоторых shell может требоваться именно такой!

```
<ключи> := [<ключ>] [<ключи>]

<ключ> := { -p | -p[<пробелы>]<значение> | --plong }

<параметры> := [<параметр>] [<параметры>]

<параметры> := <значение>
```

Формат записи ключей и длинных ключей определяется POSIX программными вызовами `getopt()` и `getopt_long()` (настоятельно рекомендуется проработать). В «хорошем» (по реализации) командном интерпретаторе ключи (опции, опциональные параметры) и параметры в командной строке могут чередоваться произвольным образом, вот эти две команды должны быть эквивалентными:

```
$ gcc hello.c -l ld -o hello
$ gcc -o hello -l ld hello.c
```

Если опция (ключ), требует значения, то это значение может отделяться пробелом, а может записываться слитно с написанием ключа (одно символьным), вот ещё примеры эквивалентных записей:

```
$ gcc hello.c -o hello
$ gcc hello.c -ohello
```

Только простые ключи (односимвольные, начинающиеся с однократного '-') могут иметь значения, «длинные» опции (многосимвольные, начинающиеся с 2-х '-') значений не имеют:

```
$ gcc --help
```

Запись командной строки можно переносить на несколько строк обратным слэшем ('\') в конце каждой продолжаемой строки.

Уровень диагностического вывода команд

Во многих командах-утилитах реализован ключ `-v` - «детализировать диагностический вывод», причём этот ключ может повторяться в командной строке несколько раз, и число повторений его определяет уровень детализации диагностики: чем больше повторений, тем выше уровень детализации.

На уровне кода (в своих собственных приложениях) это реализуется примерно так:

```
int main( int argc, char *argv[] ) {
   int c, debuglevel = 0;
   while( ( c = getopt( argc, argv, "v" ) ) != EOF )
      switch( c ) {
         case 'v': debuglevel++; break;
      }
   // к этому месту в коде сформирован уровень диагностики  debuglevel
   ...
}
```

Фильтры, каналы, конвейеры

Большинство команд (утилит) UNIX (GNU, Linux) являются **фильтрами**:

1. они имеют неявный стандартный поток ввода (файловый дескриптор 0, `SYSIN`) и стандартный поток вывода (файловый дескриптор 1, `SYSOUT`) ... (и стандартный поток журнала ошибок, 2, `SYSERR`, который в этом рассмотрении нас будет меньше прочих интересовать);

2. часто (но не обязательно) `SYSIN` это клавиатура ввода, а `SYSOUT` это экран консоли или терминала, но, например, при запуске из-под суперсервера `inetd (xinetd)` `SYSIN` и `SYSOUT` это будут сетевые TCP/IP сокеты;

3. эти потоки ввода-вывода могут перенаправляться (`>`, `>>`, `<`), или через каналы (`|`) поток вывода одной утилиты направляется в поток ввода другой:

```
$ prog 2>/dev/null
```

- подавляется вывод ошибок (`SYSERR` направляется на `/dev/null` — псевдоустройство, которое поглощает любой вывод).

```
$ ps -Af | grep /usr/bin/mc | awk '{ print $2 }'
4920
4973
5013
5096
```

- выбираем только 2-е поле (`PID`) интересующих нас строк.

4. потоки могут сливаться (очень часто это характерно для потока ошибок 2):

```
$ prog >/dev/null 2>&1
```

- поток ошибок 2 направляется в поток вывода 1 (знак `&` отмечает, что это номер дескриптора потока, `SYSOUT`, а не новый файл с именем 1); изменение порядка записи операндов в этом примере изменит результат: поток ошибок будет выводиться;

5. чаще всего (но и это не обязательно) `SYSIN` и `SYSOUT` это символьные потоки, но это могут быть и потоки бинарных данных, например, аудиопотоки в утилитах пакетов: `sox, ogg, vorbis, speex` ...

```
$ speexdec -V male.spx - | sox -traw -u -sw -r8000 - -t alsa default
$ speexdec -V male.spx - | tee male3.raw | sox -traw -u -sw -r8000 - \
-t alsa default
```

Не представляет большого труда писать свои собственные консольные (текстовые) приложения так, чтобы они тоже соответствовали общим правилам утилит POSIX. К этому стоит стремиться.

Справочные системы

Значение онлайновых справочных систем (их несколько) в Linux велико, его трудно переоценить:

- практически всю справочную информацию (команды системы, конфигурационные файлы, программные API) можно получить непосредственно из справочной системы, за экраном терминала;
- из-за «свободности» системы Linux и программного обеспечения GNU, по ним нет, и никогда не будет упорядоченной и исчерпывающей технической документации, такой полноты, скажем, как по проприетарным операционным системам Solaris или QNX; техническую информацию приходится черпать из справочных систем.

При работе со справочными системами Linux нужно учитывать одно обстоятельство и проявлять известную осторожность: справочные системы обновляются нерегулярно и неравномерно, одновременно в них могут находиться статьи совершенно различающихся лет написания, и даже относящиеся к различным версиям обсуждаемых понятий — нужно пытаться отслеживать степень свежести справочной информации!

Основная онлайн справочная система Linux, это так называемая man-справка (manual), например:

```
$ man ifconfig
IFCONFIG(8)               Linux Programmer’s Manual              IFCONFIG(8)
NAME
       ifconfig - configure a network interface
SYNOPSIS
       ifconfig [interface]
       ifconfig interface [aftype] options | address ...
DESCRIPTION
...
```

Выход из страницы `man`: клавиша '`q`' (quit)! Стандартное завершение по Ctrl-C для этой утилиты не срабатывает.

Вся справочная система разбита на **секции** по принадлежности справки. Если не возникает неоднозначности (термин не встречается в нескольких секциях), номер секции можно не указывать, иначе номер секции указывается как параметр (номер секции может указываться всегда, это не будет ошибкой):

```
$ man 1 man
...
       The standard sections of the manual include:
      1      User Commands
      2      System Calls
      3      C Library Functions
      4      Devices and Special Files
      5      File Formats and Conventions
      6      Games et. Al.
      7      Miscellanea
      8      System Administration tools and Deamons
```

Здесь мы видим тематическое разделение всей справочной системы по секциям.

Другая справочная система — `info`:

```
$ info ifconfig
...
```

```
$ info info
-----Info: (*manpages*)ifconfig, строк: 169 --Top------------------------------------------
Добро пожаловать в Info версии 4.8. ? -- справка, m выбирает пункт меню.
File: info.info,  Node: Top,  Next: Getting Started,  Up: (dir)
Info: An Introduction
*********************
The GNU Project distributes most of its on-line manuals in the "Info
format", which you read using an "Info reader".  You are probably using
an Info reader to read this now.
...
```

Есть ещё база данных по **терминам** системы, и работающие с ней несколько команд:

```
$ whatis ifconfig
ifconfig             (8)  - configure a network interface
$ whatis whatis
whatis               (1)  - search the whatis database for complete words
$ apropos whatis
apropos              (1)  - search the whatis database for strings
makewhatis           (8)  - Create the whatis database
whatis               (1)  - search the whatis database for complete words
```

Базу данных для работы нужно предварительно сформировать :

```
# makewhatis
...
```

Это (форматирование базы данных) : а). делается с правами `root`, б). потребует существенно продолжительного времени, чтоб вас это не смущало (программа не зависла!), но потребует его один раз.

Разница между `whatis` и `apropos` :

```
$ whatis /dev
/dev: nothing appropriate
$ apropos /dev
MAKEDEV             (rpm) - Программа, используемая для создания файлов устройств в /dev.
swapoff [swapon]     (2)  - start/stop swapping to file/device
swapon               (2)  - start/stop swapping to file/device
```

Наконец, справочную информацию (подсказку) принято включать непосредственно в команды, и разработчики утилит часто следуют этой традиции:

```
$ rlogin --help
usage: rlogin host [-option] [-option...] [-k realm ] [-t ttytype] [-l username]
     where option is e, 7, 8, noflow, n, a, x, f, F, c, 4, PO, or PN
$ gcc --version
gcc (GCC) 4.1.2 20071124 (Red Hat 4.1.2-42)
Copyright (C) 2006 Free Software Foundation, Inc.
```

Пользователи и права

Эта группа команд позволяет манипулировать с именами пользователей (добавлять, удалять, менять им права). Управление учётными записями пользователей — это целый раздел искусства системного администрирования. Мы же рассмотрим эту группу команд в минимальном объёме, достаточном для администрирования **своего** локального рабочего места.

Основная команда добавления нового имени пользователя:

```
# adduser
Usage: useradd [options] LOGIN
Options:
```

```
  -b, --base-dir BASE_DIR       base directory for the new user account
                                home directory
  -c, --comment COMMENT         set the GECOS field for the new user account
  -d, --home-dir HOME_DIR       home directory for the new user account
...
# which adduser
/usr/sbin/adduser
```

При создании нового имени пользователя для него будут определены (в диалоге) и значения основных параметров пользователя: пароль, домашний каталог и другие. Команды той же группы:

```
# ls /usr/sbin/user*
/usr/sbin/useradd  /usr/sbin/userhelper   /usr/sbin/usermod
/usr/sbin/userdel  /usr/sbin/userisdnctl  /usr/sbin/usernetctl
```

С этими командами достаточно ясно без объяснений. Вот как мы меняем домашний каталог для нового созданного пользователя:

```
# adduser kernel
# usermod -d /home/guest kernel
# cat /etc/passwd | grep kernel
kernel:x:503:100:kernel:/home/guest:/bin/bash
```

А вот так администратор может сменить пароль любого другого пользователя:

```
# passwd kernel
Смена пароля для пользователя kernel.
Новый пароль :
НЕУДАЧНЫЙ ПАРОЛЬ: основан на слове из словаря
НЕУДАЧНЫЙ ПАРОЛЬ: слишком простой
Повторите ввод нового пароля :
passwd: все токены проверки подлинности успешно обновлены.
```

Уже находясь в системе (под каким-то, естественно именем), мы можем всегда перерегистрироваться (в одном отдельном терминале) под именем любого известного системе пользователя:

```
$ su - kernel
Пароль:
$ whoami
kernel
$ pwd
/home/guest
```

Некоторые дистрибутивы (Ubuntu и производные) запрещают регистрацию локального сеанса под именем `root`, в таких системах административные (привилегированные) команды выполняются с префиксной командой `sudo`.

Примечание: Особенностью таких систем может быть то, что, если вы попытаетесь выполнить привилегированную команду без префикса sudo, то она выполнится без всякого вывода (результата) и установки кода ошибочного результата, что может вводить в недоумение, сравните:

```
$ fdisk -l
$ echo $?
0
$ sudo fdisk -l
[sudo] password for olej:
Диск /dev/sda: 30.0 ГБ, 30016659456 байт
...
```

Сложнее обстоят дела с действиями над паролем пользователя. Вот как посмотреть **состояние** пароля пользователя (выполняется только с правами `root`):

```
$ sudo passwd -S olej
olej PS 2010-03-13 0 99999 7 -1 (Пароль задан, шифр SHA512.)
```

Но узнать (восстановить) пароль любого ординарного пользователя администратор не может, он может только

сменить его на новый — это один из основных принципов UNIX. А как следствие этого принципа: если вы утеряли пароль пользователя `root`, то легальных способов исправить ситуацию не существует ... да и нелегальные вряд ли помогут[13] — система достаточно надёжно защищена.

Ещё один часто задаваемый вопрос с не очевидным ответом: если команда создания пользователя `adduser` всегда создаёт пользователя с паролем входа, то как создать пользователя с пустым паролем? Для этого нам нужно удалить пароль у уже существующего пользователя:

```
# adduser guest
...
# passwd -d guest
Удаляется пароль для пользователя guest..
passwd: Успех
# passwd -S guest
guest NP 2011-03-23 0 99999 7 -1 (Пустой пароль.)
```

Все регистрационные записи пользователей (как созданных администратором, так и создаваемых самой системой) хранятся в файле:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
olej:x:500:500:O.Tsiliuric:/home/olej:/bin/bash
olga:x:501:501:olga:/home/olga:/bin/bash
```

- 3-е поле каждой записи — это численный идентификатор пользователя (UID), а 4-е - численный идентификатор основной группы (GID) этого пользователя. А вот значение 'x' во 2-м поле говорит о том, что пароль для пользователя хранится в файле теневых паролей:

```
$ ls -l /etc/shad*
-r-------- 1 root root 1288 Янв 24  2010 /etc/shadow
-r-------- 1 root root 1288 Янв 24  2010 /etc/shadow-
```

Обратите внимание на права доступа к этому файлу: даже `root` не имеет права записи в этот файл.

По умолчанию, при создании командой `adduser` нового пользователя с именем `xxx` создаётся и новая группа с тем же именем `xxx`. Кроме того, пользователя дополнительно можно включить в любое число существующих групп. Смотрим состав групп, их (групп) численные идентификаторы (GID) и принадлежность пользователей к группам:

```
$ cat /etc/group
...
nobody:x:99:
users:x:100:olej,games,guest,olga,kernel
...
olej:x:500:
...
```

- во 2-й показанной строке здесь `olej` — это имя пользователя в группе `users`, а в последней — имя группы, только совпадающее **по написанию** с именем `olej` пользователя: это результат отмеченного умолчания при создании пользователя, но от него можно и отказаться, задавая группу вручную.

Пример того, как получить информацию (если забыли) кто, как и где зарегистрирован и работает в системе на

13 Сказанное относится только к системе Linux, в которой администратором приняты адекватные меры для защиты и предотвращения несанкционированной **смены** пароля `root` (иногда называемой: восстановление утерянного пароля `root`). Практически в любой инсталляции, установленной из дистрибутива («из коробки»), это не так и пароль `root` может быть сменён. Средства восстановления пароля `root` описаны в приложении в конце текста.

текущий момент времени:

```
$ who
root     tty2         2011-03-19 08:55
olej     tty3         2011-03-19 08:56
olej     :0           2011-03-19 08:22
olej     pts/1        2011-03-19 08:22 (:0)
olej     pts/0        2011-03-19 08:22 (:0)
olej     pts/2        2011-03-19 08:22 (:0)
olej     pts/3        2011-03-19 08:22 (:0)
olej     pts/4        2011-03-19 08:22 (:0)
olej     pts/5        2011-03-19 08:22 (:0)
olej     pts/6        2011-03-19 08:22 (:0)
olej     pts/9        2011-03-19 09:03 (notebook)
```

- здесь: а). 2 (стр.1,2) регистрации в **текстовых консолях** (# 2 и 3) под разными именами (`root` и `olej`); б). X11 (стр.3) регистрация (консоль #7, CentOS 5.2 ядро 2.6.18); в). 7 открытых графических терминалов в X11, дисплей :0; г). одна удалённая регистрация по SSH (последняя строка) с компьютера **с сетевым именем** `notebook`.

А вот так узнать имя, под которым зарегистрирован пользователь на текущем терминале:

```
$ whoami
olej
```

- и это совсем не пустая формальность при одновременно открытых в системе нескольких десятков терминалов.

А как получают права `root`? На то есть несколько возможностей:

1. Перерегистрация[14]:

```
$ su -
Пароль:
#
```

2. Выполнение единичной команды от имени `root`:

```
$ su -c ls
Пароль:
build.articles
$ su -c 'ls -l'
Пароль:
итого 520
drwxrwxr-x 4 olej olej   4096 Мар 13 19:54 build.articles
...
```

3. Наилучший способ выполнения команды от имени `root`:

```
$ sudo makewhatis
...
```

Но иногда (в зависимости от дистрибутива), при первом употреблении команда `sudo` нещадно ругается... В этом случае нужно настроить поведение `sudo`:

```
cat /etc/sudoers
...
## Same thing without a password
# %wheel    ALL=(ALL)    NOPASSWD: ALL
%olej       ALL=(ALL)    NOPASSWD: ALL
```

14 Уже отмечалось, что такая возможность запрещена в Ubuntu и родственных дистрибутивах.

...
- здесь показана одна новая строка, добавленная по образцу закомментированной, разрешающая «беспарольный sudo» для пользователя с именем `olej`.

Файловая система: структура и команды

В файловой системы UNIX сверх того, что уже было сказано о файловой системе ранее, работаю правила:

1. каждый объект имеет **имя**, которое может состоять из нескольких доменов-имён, разделённых символом '.' (точка), понятие «расширение» или «тип» (как в системе именования «8.3», идущей ещё от MS-DOS) не имеет такого жёсткого смысла (определяющего функциональное назначение файла):

```
$ touch start.start.start
$ ls
start.start.start
```

2. каждый объект имеет полное **путевое имя** (путь от корня файловой системы, абсолютное имя), которое составляется из имени включающего каталога и собственно имени объекта (файла):

```
$ pwd
/home/guest
```

В данном случае полное путевое имя только-что созданного выше файла будет выглядеть так: `/home/guest/start.start.start`

В файловой системе не может быть двух элементов с полностью совпадающими путевыми именами.

Путевое имя может быть абсолютным (показано выше) и относительным: относительно текущего каталога (или/и каталогов промежуточного уровня): `/home/guest/start.start.start` и `./start.start.start` - это одно и то же имя (в условиях обсуждаемого примера). Когда мы находимся, например, в каталоге:

```
$ pwd
/var/cache/yum/updates
```

- то путевые имена: `../../../log/mail` и `/var/log/mail` — это одно и то же:

```
$ ls ../../../log/mail
statistics
$ ls /var/log/mail
statistics
```

В относительных именах часто используется знак '~' - **домашний** каталог текущего пользователя:

```
$ cd ~/Download/
$ pwd
/home/olej/Download
```

3. функциональное назначение имени (файла) может быть определено (не всегда точно![15]) командой `file` :

```
$ file start.start.start
start.start.start: empty
$ file KERNEL_11.odt
KERNEL_11.odt: OpenDocument Text
$ file /dev/hde
/dev/hde: block special (33/0)
$ file /dev/tty
/dev/tty: character special (5/0)
$ file mod_proc.ko
mod_proc.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

15 Команда `file` привлекает для «угадывания» формата и назначения файла несколько разнородных механизмов, таких как: начиная от магических символов (заголовочных байт) в начале файла, и до просмотра формата и содержимого файла.

```
$ file mod_proc.c
mod_proc.c: UTF-8 Unicode C program text
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.9, dynamically
linked (uses shared libs), for GNU/Linux 2.6.9, not stripped
```

4. каждый объект (имя) файловой системы имеет **2-х** владельцев: пользователя и группу:

```
$ ls -l
-rw-rw-r-- 1 guest guest 0 Мар 27 14:20 start.start.start
```

При создании файла обычно группой является **первичная** группа создающего пользователя (но в общем случае, элементы создаются процессами, которые могут накладывать свои правила).

Владельцы и права

В смысле прав доступа к объекту файловой системы определены 3 уровня (группы): владелец, группа, остальные. В каждой группе права определены триадой (прав): `r` — чтение, `w` — запись, `x` — исполнение (для для каталогов есть отличия в толковании флагов: `w` — это право создания и удаления объектов в каталоге, `x` — это право вхождение в каталог).

```
$ sudo chown olej:guest start.start.start
$ ls -l
-rw-rw-r-- 1 olej guest 0 Мар 27 15:00 start.start.start
$ sudo chgrp users start.start.start
$ ls -l
-rw-rw-r-- 1 olej users 0 Мар 27 15:00 start.start.start
$ sudo chown root *
$ ls -l
-rw-rw-r-- 1 root users 0 Мар 27 15:00 start.start.start
```

- пользователь и группа владения меняются, а установленное расположение флагов относительно владельца и группы — остаётся.

Изменение прав (`u` — владелец, `g` — группа владения, `o` — остальные, `a` — все):

```
$ sudo chmod a+x  start.start.start
$ ls -l start.start.start
-rwxrwxr-x 1 root users 0 Мар 27 15:00 start.start.start
$ sudo chmod go-x  start.start.start
$ ls -l start.start.start
-rwxrw-r-- 1 root users 0 Мар 27 15:00 start.start.start
$ sudo chmod go=r  start.start.start
$ ls -l start.start.start
-rwxr--r-- 1 root users 0 Мар 27 15:00 start.start.start
$ sudo chmod 765  start.start.start
$ ls -l start.start.start
-rwxrw-r-x 1 root users 0 Мар 27 15:00 start.start.start
```

Флаг `x` должен выставляться для любых файлов, подлежащих исполнению; его отсутствие — частая причина проблем с выполнением текстовых файлов содержащих скриптовые сценарии (на языках: `bash`, `perl`, `python`, ...).

Информация о файле

Детальную информацию о файле (в том числе и атрибутах) даёт команда `stat`:

```
$ stat start.start.start
  File: `start.start.start'
  Size: 0               Blocks: 0          IO Block: 4096   пустой обычный файл
Device: 2146h/8518d     Inode: 1168895     Links: 1
```

```
Access: (0765/-rwxrw-r-x)  Uid: (    0/    root)   Gid: (  100/   users)
Access: 2011-03-27 15:00:35.000000000 +0000
Modify: 2011-03-27 15:00:35.000000000 +0000
Change: 2011-03-27 15:38:06.000000000 +0000
```

У команды есть множество опций, позволяющих определить формат вывода команды `stat`:

```
$ stat -c%A start.start.start
-rwxrw-r-x
$ stat -c%a start.start.start
765
```

Дополнительные атрибуты файла

У файла могут устанавливаться дополнительные атрибуты, которые выставляются флагам. Такими реально употребляемыми атрибутами являются (для атрибута числом показано значение флага в 1-м байте атрибутов):

- установка идентификатора пользователя (`setuid`) - 4;

- установка идентификатора группы (`setgid`) - 2;

- установка `sticky`-бита - 1;

Последний атрибут устарел, и используется редко. А вот возможность установки `setuid` и/или `setgid` принципиально для UNIX и устанавливаются они для исполнимых файлов: при запуске файла программы с такими атрибутами, запущенная программа выполняется не с правами (от имени) запустившего её пользователя (зарегистрировавшегося в терминале, как обычно), а от имени того пользователя (группы) который установлен как владелец этого файла программы. Это обычная практика использования `setuid`, например, когда:

– владельцем файла программы является `root`;

– и такая программа должна иметь доступ к файлам данных, доступных только `root` (например `/etc/passwd`)...

– нужно дать возможность рядовому пользователю выполнять такую программу, но не давать пользователю прав `root`;

– элементарным примером такой ситуации является то, когда вы меняете **свой** пароль доступа к системе, выполняя от своего имени команду `passwd` (проанализируйте эту противоречивую ситуацию).

Принципиально важное значение имеет возможность установки `setuid` и/или `setgid` для исполнимых файлов, в числовой записи прав доступа это выглядит так:

```
$ stat -c%a start.start.start
765
$ sudo chmod 2765  start.start.start
$ stat -c%a start.start.start
2765
$ stat -c%A start.start.start
-rwxrwSr-x
$ sudo chmod 4765  start.start.start
$ stat -c%A start.start.start
-rwsrw-r-x
$ sudo chmod 1765  start.start.start
$ stat -c%A start.start.start
-rwxrw-r-t
```

В символической записи прав для `chmod` ключ `-s` устанавливает `setuid` и `setgid` (одновременно, нет возможности управлять ими раздельно):

```
$ stat -c%a start.start.start
765
```

```
$ sudo chmod a+s start.start.start
$ stat -c%a start.start.start
6765
$ stat -c%A start.start.start
-rwsrwSr-x
```

Навигация в дереве имён

Здесь мы вспомним как перемещаться по каталогам дерева, ориентироваться где мы находимся, и искать нужные нам места файловой системы:

```
$ pwd
/home/olej/2011_WORK/Linux-kernel
$ echo $HOME
/home/olej
$ cd ~
$ pwd
/home/olej
```

Поиск бинарных файлов может осуществляться:

- только исполняемых файлов на путях из списка переменной `$PATH` для запуска приложений:

```
$ which java
/opt/java/jre1.6.0_18/bin/java
```

- поиск бинарных и некоторых других типов файлов (`man`) в списке каталогов их основного нахождения:

```
$ whereis java
java: /usr/bin/java /etc/java /usr/lib/java /usr/share/java
```

Можно видеть, что в 1-м случае найден файл из списка каталогов в переменной `$PATH`, который будет запускаться по имени без указания пути, а во 2-м случае — файлы, не лежащие в этих каталогах:

```
$ echo $PATH
/opt/java/jre1.6.0_18/bin:/usr/lib/qt-3.3/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin
```

Наконец, есть утилита, позволяющая находить файл в любом месте файловой системе по любым самым сложным и комбинированным критериям поиска:

```
$ find /etc -name passwd
/etc/passwd
find: /etc/libvirt: Отказано в доступе
/etc/pam.d/passwd
find: /etc/lvm/cache: Отказано в доступе
...
```

- в данном примере найдено 2 файла по критерию «искать файл с именем...».

Основные операции

Основные операции над объектами файловой системы (чаще всего эти объекты — файлы, но это могут быть и файлы особого рода — каталоги, и даже вообще не файлы — псевдофайлы, устройства, путевые имена в файловой системе)...

Создание каталога:

```
$ mkdir newdir
$ cd newdir
```

Создание нового (пустого) файла, что бывает нужно достаточно часто:

```
$ touch cmd.txt
```

Другой способ создания нового файла — команда `cat`:

```
$ cat > new.file
123
^C
$ cat new.file
123
```

Копирование файла, или целога каталога файлов (для копирования всех файлов каталога указание ключа рекурсивности `-R` обязательно):

```
$ cp ../f1.txt cmd.txt
$ cp -R /etc ~
cp: невозможно открыть `/etc/at.deny' для чтения: Отказано в доступе
cp: невозможно открыть `/etc/shadow-' для чтения: Отказано в доступе
cp: невозможно открыть `/etc/gshadow-' для чтения: Отказано в доступе
...
```

Подсчитать суммарный объём, занимаемый всеми файлами в указанном каталоге:

```
$ du -hs ~/etc
89M     /home/olej/etc
```

Удалить **каталог**:

```
$ rmdir ~/etc
rmdir: /home/olej/etc: Каталог не пуст
```

Команда завершается ошибкой, так как в каталоге имеются файлы. Но команда рекурсивного удаления **файлов** (каталог — файл один из ...) справится с той же задачей:

```
$ rm -R ~/etc
rm: удалить защищенный от записи обычный файл `/home/olej/etc/sudoers'? Y
...
$ rm -Rf ~/etc
$ ls ~/etc
ls: /home/olej/etc: Нет такого файла или каталога
```

Перемещение или переименование файла (или целой иерархии файлов — каталога). Если перемещение происходит в пределах одного каталога, то это переименование (1-я команда), иначе — реальное перемещение (2-я команда):

```
$ mv cmd.txt list.txt
$ mv list.txt ../cmd.txt
```

Побайтовое сравнение файлов по содержимому:

```
$ cmp -s huck.tgz huck1.tgz
$ echo $?
0
```

Для сравнения текстовых файлов (файлов программного кода) с выделением различий для последующего применения команды `patch`, используется другая команда. Ниже показан короткий законченный пример создания такой заплатки, и последующего его наложения на файл — этого достаточно для понимания основ принципа:

```
$ diff --help
Использование: diff [КЛЮЧ]... ФАЙЛЫ
Построчно сравнивает два файла.
...
$ echo 1234 > f4
```

```
$ echo 12345 > f5
$ diff f4 f5
1c1
< 1234
---
> 12345
$ diff f4 f5 > 45.patch
$ patch -i 45.patch f4
patching file f4
$ cmp f4 f5
$ echo $?
0
$ ls -l
итого 12
-rw-rw-r-- 1 olej olej 23 Апр 14 07:12 45.patch
-rw-rw-r-- 1 olej olej  6 Апр 14 07:13 f4
-rw-rw-r-- 1 olej olej  6 Апр 14 07:10 f5
```

Команды с потоками и конвейерами:

```
$ cat url.txt >> cmd.txt
$ echo 111 > newfile
$ echo $?
0
$ ls -l newf*
-rw-rw-r-- 1 olej olej 4 Мар 19 15:17 newfile
```

Команда «размножения» потока вывода на несколько потоков, с записью каждого такого экземпляра потока в свой отдельный файл:

```
$ pwd
/home/olej/TMP
$ ls
$ echo 12345 | tee 1 2 3 4 > 5
$ ls
1  2  3  4  5
```

Некоторые служебные операции над файловой системой:

- сбросить буфера файловой системы на диск:

```
$ sync
```

- проверка (и восстановление) структуры файловой системы на носителе:

```
# fsck -c
fsck 1.39 (29-May-2006)
e2fsck 1.39 (29-May-2006)
/dev/hdf6 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.
```

Этот пример говорит о том, что проверку дисковых носителей (файловых систем) нужно производить в размонтированном состоянии. Единая команда `fsck` будет вызывать другую программу, соответствующую тому типу файловой системы, который обнаружен на этом носителе, вот из этого числа:

```
$ ls /sbin/fsck*
fsck  fsck.cramfs  fsck.ext2  fsck.ext3  fsck.msdos  fsck.vfat
```

Архивы

В Linux имеется множество программ архивирования и сжатия информации. Но почти на все случаи жизни достаточно средств архивирования, интегрированных в реализацию утилиты `tar`:

```
$ tar -zxvf abs-guide-flat.tar.gz
$ tar -jxvf ldd3_pdf.tar.bz2
```

- это показано создание разархивирование из форматов `.zip` и `.bz2`, соответственно.

А вот так сворачиваются в архив `.zip` файлы текущего каталога:

```
$ tar -zcvf new-arch.tgz *
```

А вообще, в Linux собраны архиваторы, работающие с форматами, накопившимися практически за всё время существования системы:

```
$ cd /usr/bin/
$ ls *zip*
bunzip2  bzip2recover  gpg-zip  gzip  p7zip     prezip      unzip     zip       zipgrep  zipnote
bzip2    funzip        gunzip   mzip  preunzip  prezip-bin  unzipsfx  zipcloak  zipinfo  zipsplit
```

Устройства

Как сказано ранее, все имена в каталоге `/dev` соответствуют устройствам системы. Каждое устройство (кроме имени в `/dev`) однозначно характеризуется двумя номерами: старший, `major` — родовой номер класса устройств, младший, `minor` — индивидуальный номер устройства внутри класса. Именно через эти два номера происходит связь устройства с ядром Linux (или, если совсем точно, с модулем-драйвером этого устройства в составе ядра). Сами имена устройства системе не нужны — они нужны человеку-пользователю для его удобства. Номера не произвольные. Все известные номера устройств описаны в документе `devices.txt` (лежит в дереве исходных кодов ядра[16]):

```
$ cd /usr/src/linux/Documentation
$ ls -l devices.txt
-rw-rw-r-- 1 olej olej 118626 Мар  8 01:05 devices.txt
$ cat devices.txt
                    LINUX ALLOCATED DEVICES (2.6+ version)
             Maintained by Alan Cox <device@lanana.org>
                      Last revised: 6th April 2009
...
```

Все устройства делятся на символьные и блочные (устройства прямого доступа, диски). Они различаются по первой литере в выводе содержимого каталога `/dev` :

```
$ ls -l /dev | grep ^c
crw-------  1 root video      10, 175 Июл 31 10:42 agpgart
crw-rw----  1 root root       10,  57 Июл 31 10:43 autofs
crw-------  1 root root        5,   1 Июл 31 10:42 console
...
$ ls -l /dev | grep ^b
...
brw-rw----  1 root disk        8,   0 Июл 31 10:42 sda
brw-rw----  1 root disk        8,   1 Июл 31 10:42 sda1
brw-rw----  1 root disk        8,   2 Июл 31 10:42 sda2
```

16 Когда говорят о исходных кодах Linux, нужно иметь в виду, что они не присутствуют в системе изначально: в некоторых дистрибутивах (Debian и др.), вы можете загрузить их из репозитариев вашего дистрибутива менеджером пакетов, в других (Fedora, CentOS и др.) это делается более «ручным» способом... это что касается «патченных» под дистрибутив ядер. Код официального вы можете загрузить самостоятельно с адреса http://www.kernel.org/ - для обсуждаемых нами целей (рассмотрение файлов) тонкие отличия не имеют значения: вы должны выбрать архив вашей версии ядра (архив вида `linux-2.6.37.3.tar.bz2`), разархивировать его в каталог `/usr/src` (это потребует около 500Mb) и, обычно, на полученный каталог устанавливают ссылку `/usr/src/linux` — это и есть дерево исходных кодов Linux ...

```
brw-rw----  1 root disk        8,   3 Июл 31 10:42 sda3
...
```

Старшие (major) номера символьных и блочных номеров могут совпадать: они принадлежат к разным пространствам номеров. Но вот внутри класса полной идентичности двух номеров (major+minor) не может быть.

Создание нового имени устройства в каталоге /dev:

```
# mknod -m 0777 /dev/hello c 200 0
```

- в команде указываются (кроме имени устройства): права доступа к имени, характер устройства (символьное или блочное), и два номера, характеризующих устройство.

Так же, как и любое путевое имя, имена устройств удаляются командной:

```
# rm /dev/hello
```

Примечание: в нарушение любых приличий, файл устройства можно создавать в произвольном месте, например, в текущем каталоге :

```
$ pwd
/home/olej
$ sudo mknod -m 0777 ./hello c 200 0
$ echo $?
0
$ ls -l hello
crwxrwxrwx 1 root root 200, 0 Мар 19 16:27 hello
$ sudo rm ./hello
$ echo $?
0
```

Самые разнообразные устройства представляются в /dev. Часто задаваемый вопрос: как представлены, например, последовательные линии связи RS-232 (RS-485)? Вот они:

```
$ ls -l /dev/ttyS*
crw-rw---- 1 root uucp 4, 64 Апр 27 06:19 /dev/ttyS0
crw-rw---- 1 root uucp 4, 65 Апр 27 06:19 /dev/ttyS1
crw-rw---- 1 root uucp 4, 66 Апр 27 06:19 /dev/ttyS2
crw-rw---- 1 root uucp 4, 67 Апр 27 06:19 /dev/ttyS3
```

Причём, представлены как терминальные линии все 4 (максимально возможные) каналы RS-232, но откликаться на команды (например, конфигурироваться командой stty) будут только линии, реально представленные в аппаратуре компьютера (часто /dev/ttyS0 и /dev/ttyS1 — COM1 и COM2 в терминологии MS-DOS).

Подсистема udev

udev - подсистема, которая заменяет devfs без потерь для функциональности системы. Более того, udev создаёт в /dev файлы только для тех устройств, которые присутствуют на данный момент в системе. Подсистема udev является надстройкой пространства пользователя над /sys. Задача ядра определять изменения в аппаратной конфигурации системы, регистрировать эти изменения, и вносить изменения в каталог /sys. Задача подсистемы udev выполнить дальнейшую интеграцию и настройку такого устройства в системе (отобразить его в каталоге /dev), и предоставить пользователю уже готовое к работе устройство.

Подсистема udev настраивает устройства в соответствии с заданными правилами. Правила содержаться в файлах каталога /etc/udev/rules.d/ (также файлы с правилами могут содержаться и в каталоге /etc/udev/). Все файлы правил просматриваются в алфавитном порядке.

```
$ ls /etc/udev/rules.d/
05-udev-early.rules  51-hotplug.rules  60-pcmcia.rules  61-uinput-stddev.rules  90-dm.rules          bluetooth.rules
```

```
40-multipath.rules   60-libsane.rules  60-raw.rules     61-uinput-wacom.rules   90-hal.rules
50-udev.rules        60-net.rules      60-wacom.rules   90-alsa.rules           95-pam-console.rules
$ cat 60-raw.rules
...
# An example would be:
#   ACTION=="add", KERNEL=="sda", RUN+="/bin/raw /dev/raw/raw1 %N"
...
```

Информация по `udev` :

```
$ man udev
UDEV(7)                                    udev                                    UDEV(7)
NAME
       udev - dynamic device management
...
```

Основной объём потребностей по работе с `udev` покрывает не очень широко известная команда `udevadm` с огромным множеством параметров и опций:

```
$ udevadm info -q path -n sda
/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda
$ udevadm info -a -p $(udevadm info -q path -n sda)
...
looking at device '/devices/pci0000:00/0000:00:1f.2/host0/target0:0:0/0:0:0:0/block/sda':
    KERNEL=="sda"
    SUBSYSTEM=="block"
...
$ udevadm info -h
Usage: udevadm info OPTIONS
  --query=<type>             query device information:
      name                     name of device node
      symlink                  pointing to node
      path                     sys device path
      property                 the device properties
      all                      all values
  --path=<syspath>           sys device path used for query or attribute walk
  --name=<name>              node or symlink name used for query or attribute walk
...
```

Разработчики прикладных систем часто сталкиваются с `udev` в разработке конфигурационных правил для своих систем (пример: системы SoftSwitch для VoIP PBX и их интерфейс к аппаратуре связи `zaptel/DAHDI`).

Команды диагностики оборудования

Характерное отличие потребностей программиста-разработчика (как, собственно, и системного администратора) от потребностей пользователя Linux состоит в том, что разработчику часто нужны средства детальной диагностики установленного в системе периферийного оборудования (диагностики по типу, производителю, модели, по функционированию и другое). В отношении анализа всего установленного в системе оборудования, начиная с анализа производителя и BIOS — существует достаточно много команд «редкого применения», которые часто помнят только заматерелые системные администраторы, и которые не всегда попадают в справочные руководства. Все такие команды, в большинстве, требует прав `root`, кроме того, некоторые из них могут присутствовать в некоторых дистрибутивах Linux, но отсутствовать в других (устанавливаются в составе системных пакетов, если утилиты нет, то можно легко определить нужный пакет и установить его из репозитария). Информация от этих команд в какой-то мере дублирует друг друга (а в какой-то - дополняет). Но сбор такой информации об оборудовании может стать ключевой позицией при работе с периферийными устройствами.

Ниже приводится только краткое перечисление (в порядке справки-напоминания) некоторых подобных команд (и несколько начальных строк вывода, для идентификации того, что это именно та команда о которой мы

говорим) — более детальное обсуждение увело бы нас слишком далеко от наших целей. Вот некоторые такие команды:

```
$ lspci
...
00:1c.0 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 1 (rev 01)
00:1c.2 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 3 (rev 01)
00:1c.3 PCI bridge: Intel Corporation 82801G (ICH7 Family) PCI Express Port 4 (rev 01)
00:1d.0 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #1 (rev 01)
00:1d.1 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #2 (rev 01)
00:1d.2 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #3 (rev 01)
00:1d.3 USB Controller: Intel Corporation 82801G (ICH7 Family) USB UHCI Controller #4 (rev 01)
00:1d.7 USB Controller: Intel Corporation 82801G (ICH7 Family) USB2 EHCI Controller (rev 01)
...
$ lsusb
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 003: ID 0461:4d17 Primax Electronics, Ltd Optical Mouse
Bus 004 Device 002: ID 0458:0708 KYE Systems Corp. (Mouse Systems)
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 006: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 001 Device 003: ID 046d:080f Logitech, Inc.
Bus 001 Device 002: ID 0424:2503 Standard Microsystems Corp. USB 2.0 Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
$ lshal
Dumping 162 device(s) from the Global Device List:
-------------------------------------------------
udi = '/org/freedesktop/Hal/devices/computer'
  info.addons = {'hald-addon-acpi'} (string list)
...
$ sudo lshw
notebook.localdomain
    description: Notebook
    product: HP Compaq nc6320 (ES527EA#ACB)
    vendor: Hewlett-Packard
    version: F.0E
    serial: CNU6250CFF
    width: 32 bits
    capabilities: smbios-2.4 dmi-2.4
...
```

Детальная информация, в том числе, по банках памяти, и какие модули памяти куда установлены:

```
$ sudo dmidecode
# dmidecode 2.10
SMBIOS 2.4 present.
23 structures occupying 1029 bytes.
Table at 0x000F38EB.
...
```

Пакет `smartctl` (предустановлен почти в любом дистрибутиве) - детальная информация по дисковому накопителю:

```
$ sudo smartctl -A /dev/sda
smartctl 5.39.1 2010-01-28 r3054 [i386-redhat-linux-gnu] (local build)
Copyright (C) 2002-10 by Bruce Allen, http://smartmontools.sourceforge.net

=== START OF READ SMART DATA SECTION ===
SMART Attributes Data Structure revision number: 16
```

```
Vendor Specific SMART Attributes with Thresholds:
ID# ATTRIBUTE_NAME          FLAG     VALUE WORST THRESH TYPE      UPDATED  WHEN_FAILED RAW_VALUE
  1 Raw_Read_Error_Rate     0x000f   100   100   046    Pre-fail  Always       -       49961
  2 Throughput_Performance  0x0005   100   100   030    Pre-fail  Offline      -       15335665
  3 Spin_Up_Time            0x0003   100   100   025    Pre-fail  Always       -       1
  4 Start_Stop_Count        0x0032   098   098   000    Old_age   Always       -       7320
...
```

Ещё один способ получения информации о дисковом накопителе:

```
$ sudo hdparm -i /dev/sda
/dev/sda:
 Model=WDC WD2500AAKX-001CA0, FwRev=15.01H15, SerialNo=WD-WMAYU0425651
 Config={ HardSect NotMFM HdSw>15uSec SpinMotCtl Fixed DTR>5Mbs FmtGapReq }
 RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=50
 BuffType=unknown, BuffSize=16384kB, MaxMultSect=16, MultSect=16
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=488397168
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio3 pio4
 DMA modes:  mdma0 mdma1 mdma2
 UDMA modes: udma0 udma1 udma2 udma3 udma4 udma5 *udma6
 AdvancedPM=no WriteCache=enabled
 Drive conforms to: Unspecified:  ATA/ATAPI-1,2,3,4,5,6,7
```

Все такие команды имеют разветвлённую систему опций, определяющих вид затребованной информации. Все они имеют онлайновую систему подсказок (ключи -v, -h, --help), позволяющую разобраться со всем этим множеством опций.

Компиляция и сборка приложений

Основным компилятором Linux является GCC, но могут использоваться и другие, примеры таких других: а).компилятор CC из состава IDE SolarisStudio, б).активно развивающийся в рамках проекта LLVM компилятор Clang (кандидат для замены GCC в FreeBSD, причина — лицензия), в).PCC (Portable C Compiler) — новая реализация компилятора 70-х годов, широко практикуемого в NetBSD и OpenBSD. GCC имеет значительные синтаксические расширения (главным из которых являются инлайновые ассемблерные вставки), не распознаваемые другими компиляторами - поэтому альтернативные компиляторы вполне пригодны для сборки приложений, но непригодны для пересборки ядра Linux и сборки модулей ядра.

Начало GCC было положено Ричардом Столлманом, который реализовал первый вариант GCC в 1985 на нестандартном и непереносимом диалекте языка Паскаль; позднее компилятор был переписан на языке Си Леонардом Тауэром и Ричардом Столлманом и выпущен в 1987 как компилятор для проекта GNU (http://ru.wikipedia.org/wiki/GCC).

Официальный сайт GCC http://gcc.gnu.org/ :

The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...).

Компиляция 2-х фазная:

– на начальной стадии (front end) лексического анализатора, в зависимости от языка программирования, происходит синтаксическое распознавание исходного кода и преобразование к структурам на основе деревьев, и промежуточному RTL-представлению (RTL - Register Transfer Language, язык межрегистровых пересылок), напоминающему S-выражения языка LISP.

– конечная стадия (back end) генерации кода принимает с предыдущей стадии языково независимые RTL-инструкции и создает код, работающий на заданной платформе.

Существуют дополнительные front-end'ы для языков Pascal, D, Модула-2, Modula-3, Mercury, VHDL и PL/1.

Компилятор GCC

Программа:

```
#include <stdio.h>
int main( int argc, char *argv[] ) {
   printf( "Hello, world!\n" );
};
```

Примечание: единственно понадобившийся мне `#include` здесь : `<stdio.h>`, без него:

```
$ gcc hello_world.c
hello_world.c: In function 'main':
hello_world.c:4: предупреждение: incompatible implicit declaration of built-in function 'printf'
```

Вот как мы делаем компиляцию-сборку показанной выше простейшей программы:

```
$ gcc hello_world.c
$ ls -l
-rwxrwxr-x 1 olej olej 4735 Мар 19 15:44 a.out
-rw-rw-rw- 1 olej olej   94 Мар 19 15:48 hello_world.c
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.9, dynamically linked (uses shared libs), for GNU/Linux 2.6.9, not stripped
$ g++ hello_world.c
$ ls -l
-rwxrwxr-x 1 olej olej 5180 Мар 19 15:49 a.out
-rw-rw-rw- 1 olej olej   94 Мар 19 15:48 hello_world.c
```

- в последнем примере мы скомпилировали ту же текстуально программу, но компилятором с языка C++, по

размерам выходого файла видно, что результаты различаются; но это не главное различие, главное различие скрыто, и состоит в том, что откомпилированный как C или C++ коды будут использовать совершенно разные разделяемые библиотеки периода выполнения.

У GCC великое множество опций (знаменитая книга Артура Гриффитса [10] «GCC: Complete Reference» имеет 624 стр.), ниже перечислены только ежедневно используемые...

Компилятор распознаёт язык программирования исходного кода, применённый в файле, по расширению файла, например: `*.c` — C, `*.cc` — C++, `*.S` — ассемблер в AT&T нотации (не Intel!). Но язык кода можно определить и ключом `-x <язык>`. В **одной** команде GCC в качестве входных файлов могут смешиваться файлы разных форматных представлений (исходные C, исходные ассемблерные, объектные):

```
$ gcc f1.c f2.S f3.o —o resfile
```

GCC может произвести только частичную обработку, произведя результат в зависимости от ключа:

`-c` — только компилировать в объектный формат (не вызывать компоновщик):

```
$ gcc fin.c -c —o fout.o
```

`-S` — компилировать в ассемблерный код:

```
$ gcc fin.c -S —o fout.S
```

`-E` — только выполнить препроцессорную обработку и разрешить макросы:

```
$ gcc fin.c -E —o fout.c
```

Справочная информация по GCC:

```
$ gcc --version
gcc (GCC) 4.1.2 20071124 (Red Hat 4.1.2-42)
$ gcc --help
Синтаксис: gcc [ключи] файл...
...
$ man gcc
GCC(1)                                GNU                                  GCC(1)
NAME
       gcc - GNU project C and C++ compiler
...
```

- текст `man` очень объёмный и может в меру долго загружаться.

Запуск и исполнение приложений (разбор отличий оставляем на самостоятельную проработку):

```
$ gcc hello_world.c -o hello_world
$ ./hello_world
Hello, world!
$ hello_world
bash: hello_world: команда не найдена
$ strace ./hello_world
execve("./hello_world", ["./hello_world"], [/* 54 vars */]) = 0
brk(0)                                  = 0x9b8a000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=114110, ...}) = 0
mmap2(NULL, 114110, PROT_READ, MAP_PRIVATE, 4, 0) = 0xb7fa5000
close(4)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 4
read(4, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000\277\266\0004\0\0\0"..., 512) = 512
...
write(1, "Hello, world!\n", 14Hello, world!)         = 14
```

```
exit_group(14)                          = ?
$ ltrace ./hello_world
__libc_start_main(0x8048384, 1, 0xbf8e7724, 0x80483c0, 0x80483b0 <unfinished ...>
puts("Hello, world!"Hello, world!
)                                                           = 14
+++ exited (status 14) +++
```

Библиотеки

Библиотеки для Linux программного обеспечения являются важнейшей составляющей (сравните с Windows, где всё вообще практически является DLL-библиотеками). С другой стороны, по мало понятным причинам, всё что связано с библиотеками Linux крайне скудно описано и недостаточно документировано. Поэтому на этой части рассмотрения мы остановимся особо обстоятельно.

Библиотеки: использование

Объектные модули, являющиеся результатом трансляции отдельных исходных файлов, в реальных крупных проектах компонуются в библиотеки объектных модулей (и их называют просто: библиотеки, предполагая, что библиотеки содержат именно объектные модули). Использованием библиотек достигаются несколько целей:

- раздельная компиляция может во много раз сократить время компиляции при внесении изменений в проект — ускоряется темп развития разработки;
- устраняется дублирование общих частей исходного кода, используемых в разных приложениях проекта (этого иногда можно добиться и включениями #include общих фрагментов, но это разные способы реализации);
- упрощается внесение изменений и сопровождение, за счёт избежания дублирования изменения вносятся только однократно; в итоге возрастает тщательность проекта и уменьшается число несоответствий при правках;

Библиотеки могут быть созданы как статические или как динамические (разделяемые) — это два альтернативных варианта, и (в подавляющем большинстве случаев) может с равным успехом может быть выбран любой из этих вариантов, а зависимости от требований задачи. Всё сказанное относится как к библиотекам, создаваемым в ходе разрабатываемого проекта, так и к библиотекам, предоставляемым с третьих сторон от независимых разработчиков, и используемых в проекте (в том числе, и библиотеки, устанавливаемые с самой операционной системой Linux).

Библиотеки: связывание

О создании собственных библиотек мы поговорим в следующем разделе, а пока о том, как использовать уже существующие библиотеки со своими приложениями... Для использования библиотеки со своим приложением, **объектный код** приложения должен быть скомпонован с отдельными объектными модулями, **извлекаемыми** из библиотеки. Этой работой занимается компоновщик (линкер) из комплекта программ проекта GCC, вот как проделывается это в два шага, детализировано, двумя командами — на примере любого простейшего приложения (это могут быть показанные две последовательные команды в терминале, или две строки сценария `Makefile`):

```
$ gcc -c hello.c -o hello.o
$ ld /lib/crt0.o hello.o -lc -o hello
```
[17]

Первая команда только компилирует (ключ `-c`) C-код приложения (`hello.c`) в объектный вид, а вторая команда вызывает компоновщик, имя которого `ld`, и который компонует полученный объектный файл с а). стартовым объектным модулем, указанным абсолютным именем `/lib/crt0.o` и б). со всеми необходимыми объектными модулями функций из стандартной библиотеки C — `libc.so` (о том, как соотносятся **имя библиотеки**, указываемое в ключах сборки, и **имя файла библиотеки** — смотрите чуть ниже).

Но обычно компоновщик не вызывается явно, он вызывается неявно из gcc и с требуемыми умалчиваемыми

17 Конкретный пути и имена стартового объектного файла, и указание стандартной библиотеки в командной строке - могут заметно разниться от версии к версии и дистрибутива системы, показанная командная строка взята из CentOS 5.2 (ядро 2.6.18).

параметрами. Вот форма, полностью эквивалентная предыдущей:

```
$ gcc -c hello.c -o hello.o
$ gcc hello.o -o hello
```

- здесь и имя стартового объектного модуля (`/lib/crt0.o`) и имя стандартной библиотеки С (`libc.so`) gcc, вызвав компоновщик `ld`, передаст ему как параметры по умолчанию. Эквивалентным (по результату) будет и вызов:

```
$ gcc hello.c -o hello
```

- свёрнутый в одну строку, в котором сначала вызывается компилятор, а затем компоновщик, эту форму мы уже неоднократно видели, но за ней скрывается весь механизм, который мы только-что развёрнуто рассмотрели.

В зависимости от того, какой формы библиотеки используются для сборки приложения, они могут (должны) компоноваться с приложением (способ компоновки определяется ключом `-B`):

- статически:

```
$ gcc -Bstatic -L<путь> -l<библиотека> ...
```

- динамически:

```
$ gcc [-Bdynamic] -L<путь> -l<библиотека> ...
```

- или смешано:

```
$ gcc -Bstatic -l<библиотека1> ... -Bdynamic -l<библиотека2> ...
```

Причём, в смешанной записи статические и динамические библиотеки могут чередоваться произвольное число раз:

```
$ gcc -Bstatic -l<библ1> <библ2> -Bdynamic -l<библ3> -l<библ4> -Bstatic -l<библ5> -l<библ6> ... \
-Bdynamic -l<библ7> ...
```

Если способ связывания не определён в командной строке (не указан ключ -B), то по умолчанию предполагается динамический способ связывания.

Теперь относительно имён библиотек и файлов... В качестве значения опции `-l` мы указываем имя библиотеки. Но сами то библиотеки содержатся в файлах! Имена файлов, содержащих статические библиотеки, имеет расширение `.a` (archive), а файлов, содержащих динамические библиотеки - `.so` (shared objects). А само имя файла библиотеки образуется конкатенацией префикса `lib` и имени библиотеки. Таким образом, в итоге, если мы хотим скомпоновать свою программу `prog.c` с разделяемой библиотекой с именем `xxx`, то мы предполагаем наличие (на путях поиска) библиотечного файла с именем `libxxx.so`, и записываем команду компиляции так:

```
$ gcc prog.c -lxxx -oprog
```

А если мы хотим проделать то же, но со статической библиотекой, то мы должны иметь библиотечный файл с именем `libxxx.a`, и записываем команду компиляции так:

```
$ gcc prog.c -Bstatic -lxxx -oprog
```

Посмотреть на какие **файлы** разделяемых библиотек ссылается уже собранная бинарная (формата ELF) программа можно командой:

```
$ ldd hello
        linux-gate.so.1 =>  (0x00f1b000)
        libc.so.6 => /lib/libc.so.6 (0x00b56000)
        /lib/ld-linux.so.2 (0x00b33000)
```

Здесь могут ожидать неожиданности: могут быть указаны те же имена библиотек, которые мы ожидаем, но не с тем полным путевым именем, которое мы имели в виду - это может оказаться не та версия библиотеки, обуславливающая не то поведение приложения.

Некоторую сложность могут вызывать только оставшиеся вопросы: а). где компоновщик ищет библиотеки при компоновке, и б). где и как исполняемая программ ищет библиотеки для загрузки... и как это всё грамотно определить при сборке. Пути **умалчиваемого** поиска библиотек при компоновке находим в описании:

```
$ man ld
LD(1)                      GNU Development Tools                      LD(1)
NAME
       ld - The GNU linker
SYNOPSIS
...
```

Это обычно `/lib` и `/usr/lib`.

Последовательность (в порядке приоритетов) поиска библиотек **компоновщиком**:

1. По путям, указанным ключом `-L`

2. По путям, указанным списком в переменной окружения `LD_LIBRARY_PATH`

3. По стандартным путям: `/lib` и `/usr/lib`

4. По путям, которые сохранены в кэше загрузчика (`/etc/ld.so.cache`)

Для статической компоновки процесс поиска на этом и заканчивается.

Смотреть текущее содержимое кэша загрузчика можно непосредственно так:

```
$ strings '/etc/ld.so.cache' | head -n8
ld.so-1.7.0
glibc-ld.so.cache1.1
libzrtpcpp-1.4.so.0
/usr/lib/libzrtpcpp-1.4.so.0
libzltext.so.0.13
/usr/lib/libzltext.so.0.13
libzlcore.so.0.13
/usr/lib/libzlcore.so.0.13
...
```

- это один из предлагаемых в литературе способов, в таком варианте мы достаточно произвольно выделяем символьные строки из, вообще-то говоря, не символьной последовательности байт, но это работает. Другой (легальный) способ посмотреть имена используемых библиотек с полными путями их размещения — это непосредственное использование утилиты для работы с библиотеками `ldconfig`; полный вывод такой команды может быть чрезвычайно велик:

```
$ ldconfig -p | head -n10
2341 библиотек найдено в кэше «/etc/ld.so.cache»
1482 библиотек найдено в кэше «/etc/ld.so.cache»
        libzrtpcpp-1.4.so.0 (libc6) => /usr/lib/libzrtpcpp-1.4.so.0
        libzltext.so.0.13 (libc6) => /usr/lib/libzltext.so.0.13
        libzlcore.so.0.13 (libc6) => /usr/lib/libzlcore.so.0.13
...
```

И здесь мы, как обычно, пользуемся фильтрами:

```
$ ldconfig -p | grep libxml2
        libxml2.so.2 (libc6) => /usr/lib/libxml2.so.2
        libxml2.so (libc6) => /usr/lib/libxml2.so
```

Как попадают пути в кэш загрузчика? Посредством всё той же утилиты обновления (добавления) `ldconfig` из файла `/etc/ld.so.conf` ... Но в новых системах этот файл фактически пустой:

```
$ cat /etc/ld.so.conf
include ld.so.conf.d/*.conf
```

- и включает в себя последовательно перечисленное содержимое всех файлов каталога `/etc/ld.so.conf.d`. С расширением `.conf`. Поэтому, информация для обновления кэша накапливается из файлов этого каталога:

```
$ ls /etc/ld.so.conf.d
mysql-i386.conf  qt4-i386.conf  qt-i386.conf  usr-local-lib.conf  xulrunner-32.conf
```

Где, для примера, содержимое одного из выбранных наугад конфигурационных файлов и включает новый путь поиска библиотек:

```
$ cat /etc/ld.so.conf.d/usr-local-lib.conf
/usr/local/lib
```

Таким образом и каталог /usr/local/lib попадает в пути загрузки.

Обычно ldconfig запускается последним шагом инсталляции программных пакетов (особенно из исходников), но не лишне бывает выполнить его и вручную, а детали вызова смотрим справкой:

```
$ ldconfig --help
Использование: ldconfig [КЛЮЧ...]
Конфигурирует связи времени выполнения
для динамического компоновщика.
...
```

Это всё касалось поиска для связывания **любых** (статических и динамических) библиотек на этапе компоновки. Но для динамических библиотек нужен ещё поиск периода старта приложения, использующего библиотеку (библиотека могла быть, например, перенесена со времени сборки приложения). Такой поиск производится функциями (динамического линкера), содержащимися в динамической **библиотеке** /lib/ld-2.13.so, с которой (естественно, номер версии в имени файла может меняться) компонуется по умолчанию (без нашего вмешательства) любая программа, использующая разделяемые библиотеки. Требуемые приложению библиотеки ищутся на путях, указанных в ключах -rpath и -rpath-link при сборке программы, а затем по значению путей в списке переменной окружения с именем LD_RUN_PATH. Далее проводится поиск по п.п. 3,4 показанных ранее.

Существует два различных метода использования динамической библиотеки из приложения (оба метода берут свое начало в Sun Solaris). Первый способ – это **динамическая компоновка** вашего приложения с совместно используемой библиотекой. Это более традиционный способ который мы уже неявно начали рассматривать. При этом способе загрузку библиотеки при запуске приложения берёт на себя операционная система. Второй называют **динамической загрузкой**. В этом случае программа явно загружает нужную библиотеку, а затем вызывает определенную библиотечную функцию (или набор функций). На этом втором методе обычно основан механизм загрузки подключаемых программных модулей – плагинов. Этот способ может быть иногда особенно интересен разработчикам встраиваемого оборудования. Компоновщик не получает никакой информации о библиотеках и обьектах в ходе компоновки. Библиотека libdl.so (компонуемая к приложению) предоставляет интерфейс к динамическому загрузчику. Этот интерфейс составляют 4 функции: dlopen(), dlsym(), dlclose() и dlerror(). Первая принимает на вход строку с указанием имени библиотеки для загрузки, загружает библиотеку в память (если она еще не была загружена), или увеличивает количество ссылок на библиотеку (если она уже найдена в памяти). Возвращает дескриптор, который потом используется в функциях dlsym() и dlclose(). Функция dlclose(), соответственно, уменьшает счетчик ссылок на библиотеку и выгружает ее, если счетчик становится равным 0. Функция dlsym() по имени функции возвращает указатель на ее код. Функция dlopen() в качестве второго параметра получает управляющий флаг, определяющий детали загрузки библиотеки. Он может иметь следующие значения:

RTLD_LAZY - разрешение адресов по именам объектов происходит только для используемых объектов (это не относится к глобальным переменным — они разрешаются немедленно, в момент загрузки).

RTLD_NOW - разрешение всех адресов происходит до возврата из функции.

RTLD_GLOBAL - разрешает использовать объекты, определенные в загружаемой библиотеке для разрешения адресов из других загружаемых библиотек.

RTLD_LOCAL - запрещает использовать объекты из загружаемой библиотеки для разрешения адресов из других загружаемых библиотек.

RTLD_NOLOAD - указывает не загружать библиотеку в память, а только проверить ее наличие в памяти. При использовании совместно с флагом RTLD_GLOBAL позволяет «глобализовать» библиотеку, предварительно загруженную локально (RTLD_LOCAL).

RTLD_DEEPBIND - помещает таблицу загружаемых объектов в самый верх таблицы глобальных символов. Это

гарантирует использование только своих функций и нивелирует их переопределение функций другими библиотеками.

`RTLD_NODELETE` - отключает выгрузку библиотеки при уменьшении счетчика ссылок на нее до 0.

Изложенной информации более чем достаточно, чтобы скомпоновать наше собственное приложение с любой, **уже имеющейся** в нашем распоряжении, библиотекой объектных модулей.

Библиотеки: построение

А теперь мы дополним изложение предыдущего раздела, позволяющее скомпоновать приложение с любой библиотекой, ещё и умением самим изготавливать такие библиотеки.

Когда у нас есть в приложении несколько (как минимум 2, или сколь угодно более) отдельно скомпилированных объектных модуля (один из которых — главный объектный модуль с функцией `main()`), то у нас есть, на выбор, целый спектр возможностей скомпоновать их в единое приложение, как минимум:

а). всё скомпоновать в единое монолитное приложение;

б). собрать модули в статическую библиотеку и прикомпоновывать её к приложению;

в). собрать модули в автоматически подгружаемую разделяемую библиотеку;

г). собрать модули в динамически подгружаемую по требованию разделяемую библиотеку;

Все эти возможности показаны в архиве примеров `libraries.tgz`.

Во всех случаях (кроме последнего 4-го) используем практически одни исходные файлы (см. архив примера), главным отличием будут `Makefile` для сборки (сравнивайте размеры аналогичных файлов!):

hello_main.c

```
#include "hello_child.h"
int main( int argc, char *argv[] ) {
   char *messg = "Hello world!\n";
   int res = put_my_msg( messg );
   return res;
};
```

hello_child.c

```
#include "hello_child.h"
int put_my_msg( char *messg ) {
   printf( messg );
   return -1;
};
```

hello_child.h

```
#include <stdio.h>
int put_my_msg( char* );
```

Собираем цельное монолитное приложение из отдельно компилируемых объектных модулей (никакая техника библиотек не используется):

Makefile

```
TARGET = hello
MAIN = $(TARGET)_main
CHILD = $(TARGET)_child

all: $(TARGET)
$(TARGET):      ../$(MAIN).c ../$(CHILD).c ../$(CHILD).h
```

```
                gcc ../$(MAIN).c ../$(CHILD).c -o $(TARGET)
                rm -f *.o
```

Выполнение:

```
$ make
gcc ../hello_main.c ../hello_child.c -o hello
rm -f *.o
$ ./hello
Hello world!
$ ls -l hello
-rwxrwxr-x 1 olej olej 4975 Июл 30 15:25 hello
```

Собираем аналогичное приложение с использованием статической библиотеки:

Makefile

```
TARGET = hello
MAIN = $(TARGET)_main
CHILD = $(TARGET)_child
LIB = lib$(TARGET)

all: $(LIB) $(TARGET)
$(LIB):         ../$(CHILD).c ../$(CHILD).h
                gcc -c ../$(CHILD).c -o $(CHILD).o
                ar -q $(LIB).a $(CHILD).o
                rm -f *.o
                ar -t $(LIB).a
$(TARGET):      ../$(MAIN).c $(LIB)
                gcc ../$(MAIN).c -Bstatic -L./ -l$(TARGET) -o $(TARGET)
```

Объектные модули в статическую библиотеку (архив) в Linux собирает (добавляет, удаляет, замещает, ...) утилита `ar` из пакета `binutils` (но если у вас установлен пакет `gcc`, то он по зависимостям установит и пакет `binutils`). Архив `.a` в Linux не является специальным библиотечным форматом, это набор отдельных компонент с каталогом их имён, он может использоваться для самых разных целевых назначений. Но, в частности, с ним, как с хранилищем объектных модулей умеет работать компоновщик `ld`.

Выполнение для этого варианта сборки:

```
$ make
gcc -c ../hello_child.c -o hello_child.o
ar -q libhello.a hello_child.o
ar: creating libhello.a
rm -f *.o
ar -t libhello.a
hello_child.o
gcc ../hello_main.c -Bstatic -L./ -lhello -o hello
$ ./hello
Hello world!
$ ls -l hello
-rwxrwxr-x 1 olej olej 4975 Июл 30 15:31 hello
```

Обращаем внимание, что размер собранного приложения здесь в точности соответствует предыдущему случаю (что совершенно естественно: собираются в точности идентичные экземпляры приложения, только источники одних и тех же объектных модулей для их сборки используются различные, в первом случае — это модули в отдельных файлах, во втором — те же модули, но в каталогизированном архиве).

Переходим к сборке разделяемых библиотек. Объектные модули для разделяемой библиотеки должны быть компилированы в позиционно независимый код (PIC - перемещаемый, не привязанный к адресу размещения — ключи `-fpic` или `-fPIC` компилятора).

Примечание: Документация говорит, что опция `-fPIC` обходит некоторые ограничения `-fpic` на некоторых аппаратных платформах (m68k, Spark), но в обсуждениях утверждается, что из-за некоторых ошибок в реализации `-fpic`, лучше указывать обе эти опции, хуже от этого не становится. Но всё это, наверное, очень сильно зависит от версии компилятора.

С учётом всех этих обстоятельств, собираем автоматически подгружаемую (динамическая компоновка) разделяемую библиотека[18]:

Makefile

```
TARGET = hello
MAIN = $(TARGET)_main
CHILD = $(TARGET)_child
LIB = lib$(TARGET)

all: $(LIB) $(TARGET)
$(LIB):         ../$(CHILD).c ../$(CHILD).h
                gcc -c -fpic -fPIC -shared ../$(CHILD).c -o $(CHILD).o
                gcc -shared -o $(LIB).so $(CHILD).o
                rm -f *.o
$(TARGET):      ../$(MAIN).c $(LIB)
                gcc ../$(MAIN).c -Bdynamic -L./ -l$(TARGET) -o $(TARGET)
```

Сборка приложения:

```
$ make
gcc -c -fpic -fPIC -shared ../hello_child.c -o hello_child.o
gcc -shared -o libhello.so hello_child.o
rm -f *.o
gcc ../hello_main.c -Bdynamic -L./ -lhello -o hello
$ ls
hello  libhello.so  Makefile
$ ls -l hello
-rwxrwxr-x 1 olej olej 5051 Июл 30 15:40 hello
```

Отмечаем, что на этот раз размер приложения отличается и, вопреки ожиданиям, в сторону увеличения. Конкретный размер, пока, нас не интересует, но различия в размере говорят, что это — совершенно другое приложение, в отличие от предыдущих случаев. На этот раз запустить приложение будет не так просто (на этот счёт смотрите подробное разъяснение о путях поиска библиотек: текущий каталог не входит и **не может входить** в пути поиска динамических библиотек):

```
$ ./hello
./hello: error while loading shared libraries: libhello.so: cannot open shared object file: No such
file or directory
```

Мы можем (для тестирования) обойти эту сложность следующим образом:

```
$ export LD_LIBRARY_PATH=`pwd`; ./hello
Hello world!
```

Отметим такие детали, как:

- после такого запуска переменная окружения уже установлена в текущем терминале, и в последующем может быть многократно использована приложением:

```
$ ./hello
Hello world!
```

- но это относится только к текущему терминалу, в любом другом терминале вся картина повторится сначала...

- пути поиска динамических библиотек не могут быть заданы относительными путями (`./`, `../`, ...), а могут

18 Компиляция и сборка самой библиотеки в этом и следующем примере записана 2-мя строками, командами вызова `gcc`: первая компилирует модуль, а вторая помещает его в библиотеку. Это сделано для наглядности, чтобы разделить опции (ключи) компиляции и сборки. На практике эти две строки записываются в один вызов `gcc`, который обеспечивает и то и другое действие.

быть только абсолютными (от корня файловой системы /):

```
$ echo $LD_LIBRARY_PATH
/home/olej/2011_WORK/GlobalLogic/my.EXAMPLES/examples.DRAFT/libraries/auto
```

- последнее (использование только абсолютных путей) и понятно, поскольку для динамических библиотек сборка и использование — это два совершенно различных акта во времени, и на сборке невозможно предугадать какой каталог будет текущим при запуске, и от какого отсчитывать относительные пути.

Наконец, собираем динамически подгружаемую по требованию (динамическая загрузка) разделяемую библиотека. В этом случае главный файл приложения имеет иной вид:

hello_main.c

```
#include <dlfcn.h>
#include "../hello_child.h"
typedef int (*my_func)( char* );
int main( int argc, char *argv[] ) {
char *messg = "Hello world!\n";
   // Открываем совместно используемую библиотеку
   void *dl_handle = dlopen( "./libhello.so", RTLD_LAZY );
   if( !dl_handle ) {
      printf( "ERROR: %s\n", dlerror() );
      return 3;
   }
   //  Находим адрес функции в библиотеке
   my_func func = dlsym( dl_handle, "put_my_msg" );
   char *error = dlerror();
   if( error != NULL ) {
      printf( "ERROR: %s\n", dlerror() );
      return 4;
   }
   // Вызываем функцию по найденному адресу
   int res = (*func)( messg );
   // Закрываем библиотеку
   dlclose( dl_handle );
   return res;
};
```

Примечание: Отметим важное малозаметное обстоятельство: вызов функции библиотеки `(*func)(messg)` по имени происходит без какой-либо синтаксической проверки на соответствие тому прототипу, который объявлен для типа функции `my_func`. Соответствие обеспечивается только «доброй волей» пишущего код. С таким же успехом функция могла бы вызываться с 2-мя параметрами, 3-мя и так далее. На это место нужно обращать пристальное внимание при написании реальных плагинов.

Makefile

```
TARGET = hello
MAIN = $(TARGET)_main
CHILD = $(TARGET)_child
LIB = lib$(TARGET)

all: $(LIB) $(TARGET)
$(LIB):         ../$(CHILD).c ../$(CHILD).h
                gcc -c -fpic -fPIC -shared ../$(CHILD).c -o $(CHILD).o
                gcc -shared -o $(LIB).so $(CHILD).o
                rm -f *.o *.so $(TARGET)
$(TARGET):      $(MAIN).c $(LIB)
                gcc $(MAIN).c -o $(TARGET) -ldl
```

Сборка:

```
$ make
gcc -c -fpic -fPIC -shared ../hello_child.c -o hello_child.o
gcc -shared -o libhello.so hello_child.o
rm -f *.o
gcc hello_main.c -o hello -ldl
$ ls
hello  hello_main.c  libhello.so  Makefile
$ ls -l hello
-rwxrwxr-x 1 olej olej 5487 Июл 30 16:06 hello
```

Выполнение:

```
$ ./hello
Hello world!
$ echo $?
255
```

- для дополнительного контроля код приложения написан так, что код возврата приложения в систему (255) равен значению, возвращённому функцией из динамической библиотеки.

Как это всё работает?

Теперь, когда мы умеем связать свою программу с любыми библиотеками, вернёмся к вопросу: что происходит при использовании того или иного сорта библиотеки, и в чём разница? Статическая библиотека не представляет из себя по структуре ничего более, чем просто линейный набор N объектных модулей, снабжённых каталогом для их поиска (чем и занимается утилита `ar`). При компоновке пользовательского процесса, из библиотеки выбираются (по внешним ссылкам) и извлекаются M требуемых программе объектных модулей, и сатически собираются в единое целое (часто M<<N). Объём полученного в результате процесса (занимаемая при его загрузке память) пропорционален объёму M модулей (но не N)! Если некоторая функция `xxx()` используется несколькими собираемыми процессами в проекте P1, P2, P3, ... PK, то экземпляр объектного модуля этой функции будет прикомпонован к каждому процессу, и если, вдруг, потребуется загрузить одновременно все эти процессы проекта, то они потребуют под загрузку функции `xxx()` в K раз больше памяти, чем сам размер модуля.

Автоматически загружаемая динамическая библиотека (наиболее частый на практике случай), если она не загружена в памяти, загружается при загрузке использующего её процесса. Если к загрузке этого процесса библиотека уже загружена, то новый экземпляр уже не загружается, а используется ранее загруженный; в этом случае только увеличивается число ссылок использования библиотеки (внутренний параметр). Автоматически загружаемая библиотека не может быть выгружена из памяти до тех пор, пока её счётчик ссылок использования не нулевой. Обратим внимание на то, что в отличие от того, что описано относительно статической библиотеки, если процессу нужен хотя бы одна точка входа из библиотеки, будет загружен весь объём N объектных модулей.

При загрузке по требованию, сама динамическая библиотека ведёт себя в точности также. Но по исполнению, со стороны вызывающего приложения, это очень похоже на оверлейную загрузку (мы это ещё обсудим позже). На этот способ стоит обратить особое внимание, так он является готовым механизмом для создания **динамических плагинов** к проекту.

Конструктор и деструктор

Посмотрим внешние имена (для связывания) созданной в предыдущем примере разделяемой библиотеки:

```
$ nm libhello.so
...
00000498 T _fini
000002ec T _init
...
00000430 T put_my_msg
...
```

Мы видим ряд имён (и как раз типа T для внешнего связывания), одно из которых `put_my_msg` — это имя нашей функции. Два других имени — присутствуют в любой разделяемой библиотеке: функция `_init()` вызывается при загрузке библиотеки в память (например, для инициализации некоторых структур данных), а `_fini()` - при выгрузке библиотеки (деструктор).

Примечание: Наличие функций конструктора и деструктора (`_init()` и `_fini()`) является общим свойством всех файлов формата ELF, к которому относятся и исполнимые файлы Linux, и файлы динамических разделяемых библиотек.

Эти функции (конструктор и деструктор библиотеки) могут быть переопределены из вашего пользовательского кода создания библиотеки. Для этого перепишем динамическую библиотеку из наших предыдущих примеров (каталог `init` архива `libraries.tgz`):

hello_child.c :

```
#include "../hello_child.h"
#include <sys/time.h>

static mark_time( void ) {
   struct timeval t;
   gettimeofday( &t, NULL );
   printf( "%02d:%06d : ", t.tv_sec % 100, t.tv_usec );
}

static mark_func( const char *f ) {
   mark_time();
   printf( "%s\n", f );
}

void _init( void ) {
   mark_func( __FUNCTION__ );
}

void _fini( void ) {
   mark_func( __FUNCTION__ );
}

int put_my_msg( char *messg ) {
   mark_time();
   printf( "%s\n", messg );
   return -1;
}
```

Но просто собрать такую библиотеку не получится:

```
$ gcc -c -fpic -fPIC -shared hello_child.c -o hello_child.o
$ gcc -shared -o libhello.so hello_child.o
...
hello_child.c:(.text+0x0): multiple definition of `_init'
/usr/lib/gcc/i686-redhat-linux/4.4.4/../../../crti.o:(.init+0x0): first defined here
hello_child.o: In function `_fini':
hello_child.c:(.text+0x26): multiple definition of `_fini'
/usr/lib/gcc/i686-redhat-linux/4.4.4/../../../crti.o:(.fini+0x0): first defined here
...
```

Совершенно естественно, так как мы пытались повторно переопределить имена, уже определенные в стартовом объектном коде. Желаемого результата мы достигнем сборкой с опциями, как показано в следующем сценарии сборки:

Makefile :

```
TARGET = hello
CHILD = $(TARGET)_child
```

```
LIB = lib$(TARGET)

TARGET1 = $(TARGET)_d
TARGET2 = $(TARGET)_a

all: $(LIB) $(TARGET1) $(TARGET2)

$(LIB):         $(CHILD).c ../$(CHILD).h
                gcc -c -fpic -fPIC -shared $(CHILD).c -o $(CHILD).o
                gcc -shared -nostartfiles -o $(LIB).so $(CHILD).o
                rm -f *.o

$(TARGET1):     $(TARGET1).c $(LIB)
                gcc $< -Bdynamic -ldl -L./ -l$(TARGET) -o $@

$(TARGET2):     $(TARGET2).c $(LIB)
                gcc $< -Bdynamic -L./ -l$(TARGET) -o $@
```

- здесь как `TARGET1` (файл `hello_d`) собирается приложение, самостоятельно подгружающее динамическую библиотеку `libhello.so` по требованию, а как `TARGET2` (файл `hello_a`) - приложение, опирающееся на автоматическую загрузку библиотек средствами системы. Сами исходные коды приложений теперь имеют вид:

hello_a.c :

```
#include "../hello_child.h"
int main( int argc, char *argv[] ) {
   int res = put_my_msg( (char*)__FUNCTION__ );
   return res;
};
```

hello_d.c :

```
#include <dlfcn.h>
#include "../hello_child.h"
typedef int (*my_func)( char* );

int main( int argc, char *argv[] ) {
   // Открываем совместно используемую библиотеку
   void *dl_handle = dlopen( "./libhello.so", RTLD_LAZY );
   if( !dl_handle ) {
      printf( "ERROR: %s\n", dlerror() );
      return 3;
   }
   // Находим адрес функции в библиотеке
   my_func func = dlsym( dl_handle, "put_my_msg" );
   char *error = dlerror();
   if( error != NULL ) {
      printf( "ERROR: %s\n", dlerror() );
      return 4;
   }
   // Вызываем функцию по найденному адресу
   int res = (*func)( (char*)__FUNCTION__ );
   // Закрываем библиотеку
   dlclose( dl_handle );
   return res;
};
```

Теперь собираем всё это вместе и приступаем к опробованию:

```
$ make
gcc -c -fpic -fPIC -shared hello_child.c -o hello_child.o
gcc -shared -nostartfiles -o libhello.so hello_child.o
```

```
rm -f *.o
gcc hello_d.c -Bdynamic -ldl -L./ -lhello -o hello_d
gcc hello_a.c -Bdynamic -L./ -lhello -o hello_a
$ export LD_LIBRARY_PATH=`pwd`
$ ./hello_a
65:074290 : _init
65:074387 : main
65:074400 : _fini
$ ./hello_d
68:034516 : _init
68:034782 : main
68:034805 : _fini
```

Смысл приложений в том, что каждая из функций, в порядке их вызова, выводи своё имя и метку времени, когда она вызвана (число после двоеточия — микросекунды, перед — секунды).

Подмена имён

Переопределение кода функций `_init()` и `_fini()` это не есть самая хорошая идея, потому что мы затрагиваем структуру ELF файла и можем породить нежелательные тонкие эффекты (как всегда с функциями, начинающихся в имени с _). Но есть другие способы: просто объявить свои произвольные функции как конструктор и деструктор (каталог `init` архива `libraries.tgz`). Перепишем реализацию библиотеки предыдущего примера, всё остальное, и сборка — остаются неизменными:

hello_child.h :

```
#include "../hello_child.h"
#include <sys/time.h>

static mark_time( void ) {
   struct timeval t;
   gettimeofday( &t, NULL );
   printf( "%02d:%06d : ", t.tv_sec % 100, t.tv_usec );
}

static mark_func( const char *f ) {
   mark_time();
   printf( "%s\n", f );
}

__attribute__ ((constructor))
   void my_init_1( void ) {
   mark_func( __FUNCTION__ );
}

__attribute__ ((constructor))
   void my_init_2( void ) {
   mark_func( __FUNCTION__ );
}

__attribute__ ((destructor))
   void my_fini_1( void ) {
   mark_func( __FUNCTION__ );
}

__attribute__ ((destructor))
   void my_fini_2( void ) {
   mark_func( __FUNCTION__ );
}

int put_my_msg( char *messg ) {
```

```
   mark_time();
   printf( "%s\n", messg );
   return -1;
}
```

Вот как выглядит выполнение примера в таком виде:

```
$ export LD_LIBRARY_PATH=`pwd`
$ ./hello_a
51:256261 : my_init_2
51:256345 : my_init_1
51:256365 : main
51:256384 : my_fini_1
51:256394 : my_fini_2
```

Мы определили сколь угодно конструкторов и деструкторов, в виде собственных функций, которые могут с равным успехом вызываться и из программного кода приложения (но вот порядком вызовов множественных конструкторов и деструкторов мы управлять не можем). Здесь работу за нас выполняет ключевое слово `__attribute__` — это одно из расширений компилятора `gcc`.

Данные в динамической библиотеке

До сих пор мы говорили только о функциях в библиотеке, вызываемых из кода приложения. А что, если библиотека содержит специфические данные, или даже модифицируются по ходу выполнения приложения? Сделаем библиотеку и соответствующую ей задачу, которые ответят нам на этот вопрос. Сначала смотрим код библиотеки:

lib.h :

```
#define BUF_SIZE 200
void put_new_string( const char *s );
void get_new_string( char *s );
```

lib.c :

```
#include <string.h>
#include "lib.h"

static char buffer[ BUF_SIZE + 1 ] = "initial buffer state!\n";

void put_new_string( const char *s ) {
   strcpy( buffer, s );
}

void get_new_string( char *s ) {
   strcpy( s, buffer );
}
```

Вот, собственно, и вся библиотека: она экспортирует два имени, но функция `put_new_string()` изменяет содержимое внутреннего статического (невидимого внаружу) буфера `buffer`. И соответствующий пользовательский процесс, который на каждом прохождении цикла индицирует значение, считанное им библиотечным вызовом `get_new_string()`, после чего обновляет по `put_new_string()` это содержимое считанной с консоли строкой:

prog.c :

```
#include "lib.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void put( char* msg ) {
   time_t t;
   time( &t );
   struct tm *m = localtime( &t );
   printf( "%02d:%02d :\t%s", m->tm_min, m->tm_sec, msg );
}
```

```
int main( int argc, char *argv[] ) {
   char buffer[ BUF_SIZE + 1 ] = "";
   while( 1 ) {
      get_new_string( buffer );
      put( buffer );
      fprintf( stdout, "> " );
      fflush( stdout );
      fgets( buffer, sizeof( buffer ), stdin );
      put( buffer );
      put_new_string( buffer );
      printf( "--------------------------\n" );
   }
}
```

Для ясности — сценарий и процесс сборки:

Makefile :

```
LSRC = lib
LNAME = new
LIB = lib$(LNAME)
PROG = prog

all: $(LIB) $(PROG)

$(LIB):         $(LSRC).c $(LSRC).h
                gcc -c -fpic -fPIC -shared $(LSRC).c -o $(LSRC).o
                gcc -shared -o $(LIB).so $(LSRC).o
                rm -f $(LSRC).o

$(PROG):        $(PROG).c $(LIB)
                gcc $< -Bdynamic -L./ -l$(LNAME) -o $@
```

```
$ make
gcc -c -fpic -fPIC -shared lib.c -o lib.o
gcc -shared -o libnew.so lib.o
rm -f lib.o
gcc prog.c -Bdynamic -L./ -lnew -o prog
$ ls
lib.c  lib.h  libnew.so  Makefile  prog  prog.c
```

А теперь выполняем с двух различных терминалов два независимых экземпляра полученной программы `prog`, которые используют единый общий экземпляр разделяемой библиотеки `libnew.so`:

```
$ export LD_LIBRARY_PATH=`pwd`
$ ./prog
34:41 : initial buffer state!
> 2-й терминал
35:15 : 2-й терминал
--------------------------
35:15 : 2-й терминал
> повторение со второго терминала
35:53 : повторение со второго терминала
--------------------------
35:53 : повторение со второго терминала
> ^C

$ export LD_LIBRARY_PATH=`pwd`
```

```
$ ./prog
34:52 : initial buffer state!
> 1-й терминал
35:05 : 1-й терминал
-------------------------
35:05 : 1-й терминал
> повторение с 1-го терминала
35:34 : повторение с 1-го терминала
-------------------------
35:34 : повторение с 1-го терминала
> ^C
```

Прекрасно видно (по чередующимся временным меткам операции, формат <минуты>:<секунды>), что каждый экземпляр программы работает со своей копией буфера, не затирая данные параллельно работающего экземпляра программы: при первой же модификации области данных экземпляру создаётся своя независимая копия данных (COW — copy on write).

Некоторые сравнения

Одинаково ли по производительности выполняются процессы, собранные со своими объектными модулями статически и динамически? Нет! И дело здесь не в том, что вызовы динамически связываемых функций будут иметь некоторый уровень косвенности через таблицы имён — это копейки... Существенно то, что объектные модули для помещения в динамическую библиотеку должны компилироваться с опцией «позиционно независимый код» (ключ `-fpic`), а такой код сложнее, менее производительный и хуже подлежит оптимизации компилятором. В результате может быть некоторая потеря производительности. Обычно это мало заметно, но в некоторых областях, особенно в алгоритмах цифровой обработки сигналов (digital signal processing - DSP: быстрые преобразования Фурье, авторегрессионные фильтры, кодаки ... и многое другое) это может стать существенным.

Относительно расходования памяти. Естественно, динамические библиотеки гораздо экономичнее расходуют память за счёт исключения дублирования кода. Но это происходит, если размер библиотеки поддерживается разумно небольшой, и в одну библиотеку не наталкивается всё, что только можно придумать: если программе нужен хотя бы единый вызов из библиотеки, то загружается **вся** библиотека. Кроме того, не следует забывать, что хотя для N программ, использующих динамическую библиотеку, загружается и одна копия самой библиотеки, но с каждым из N процессов загружается таблица имён используемой библиотеки, то есть, в итоге, загружается N экземпляров таблицы. При объёмных библиотеках это может быть существенная величина.

Для экономной работы с памятью (особо для встраиваемых и малых архитектур) может оказаться перспективным обсуждавшийся ранее способ загрузки библиотек по требованию: обширная библиотека расщепляется на несколько более мелких, и каждая из них загружается только на период времени её прямого использования. Таким образом библиотеки могут поочерёдно грузиться в одну и ту же область памяти, что реализует схему оверлейной загрузки фрагментов кода.

Создание проектов, сборка make

Многократно выполняемая сборка приложений проекта, с учётом зависимостей и обновлений, делается утилитой `make`, которая использует оформленный сценарий сборки. Мы уже неоднократно прибегали к `make` в рассматриваемых ранее примерах, а теперь посмотрим на эту утилиту подробнее.

Утилита `make` автоматически определяет какие части большой программы должны быть перекомпилированы, и выполняет необходимые для этого действия. На самом деле, область применения `make` не ограничивается только сборкой программ. Её можно использовать ее для решения любых задач, где одни файлы должны автоматически обновляться при изменении других файлов.

Утилита make существует в разных ОС, из-за особенностей выполнения наряду с «родной» реализацией во многих ОС присутствует GNU реализация gmake, и поведение этих реализаций может достаточно существенно отличаться (в некоторых ОС, например, Solaris), а в сценариях сборки указываться имя конкретной из утилит. В Linux эти два имени являются синонимами (реализованы через ссылку):

```
$ ls -l /usr/bin/*make
...
lrwxrwxrwx 1 root root      4 Окт 28  2008 /usr/bin/gmake -> make
...
-rwxr-xr-x 1 root root 162652 Май 25  2008 /usr/bin/make
$ make --version
GNU Make 3.81
...
```

По умолчанию имя файла сценария сборки - Makefile. Утилита make обеспечивает полную сборку указанной **цели** в сценарии сборки, например:

```
$ make
$ make clean
```

Если цель не указывается, то выполняется **первая последовательная** цель в файле сценария[19]. Может использоваться и любой другой сценарный файл сборки:

```
$ make -f Makefile.my
```

Простейший Makefile состоит из синтаксических конструкций всего двух типов: целей и макроопределений. Описание цели состоит из трех частей: имени цели, списка зависимостей и списка команд интерпретатора shell, требуемых для построения цели. Имя цели — непустой список файлов, которые предполагается создать. Список зависимостей — список файлов, в зависимости от которых строится цель. Имя цели и список зависимостей составляют заголовок цели, записываются в одну строку и разделяются двоеточием (':'). Список команд записывается со следующей строки, причем все команды начинаются **с обязательного символа табуляции**. Любая строка в последовательности списка команд, не начинающаяся с табуляции (ещё одна команда) или '#' (комментарий) — считается завершением текущей цели и началом новой.

Утилита make имеет много умалчиваемых значений, важнейшими из которых являются правила обработки суффиксов, а также определения внутренних переменных окружения. Эти данные называются базой данных make и могут быть рассмотрены так:

```
$ make -p >make.suffix
make: *** Не заданы цели и не найден make-файл.  Останов.
$ cat make.suffix
# GNU Make 3.81
# Copyright (C) 2006  Free Software Foundation, Inc.
...
# База данных Make, напечатана Thu Apr 14 14:48:51 2011
...
CC = cc
LD = ld
AR = ar
CXX = g++
COMPILE.cc = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
COMPILE.C = $(COMPILE.cc)
...
SUFFIXES := .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l .s .S .mod .sym .def .h .info .dvi
.tex .texinfo .texi .txinfo .w .ch...
# Implicit Rules
...
%.o: %.c
```

19 Бытует заблуждение, что по умолчанию выполняется некая цель all, но это неверно, просто этим именем часто называют первую по порядку цель, но её может попросту не быть среди других.

```
#  команды, которые следует выполнить (встроенные):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
...
```

Все эти значения (переменных: `CC, LD, AR, EXTRA_CFLAGS, ...`) могут использоваться файлом сценария как неявные определения с значениями по умолчанию. Кроме этого, вы можете определить и свои правила обработки по умолчанию для указанных вами суффиксов (расширений файловых имён), как это показано на примере выше для исходных файлов кода на языке C: `*.c`.

Подавляющее большинство интегрированных сред разработки (IDE) или пакетов созданий переносимых инсталляций (таких как `automake` & `autoconf`) ставят своей задачей создание сценарного файла `Makefile` для утилиты `make`.

Как существенно ускорить сборку make

Сборка простых проектов происходит достаточно быстро. Но при естественном росте проекта в ходе его развития, сборка, основное время которой затрачивается на компиляцию, может значительно возрастать, становясь уже раздражающим фактором. Хорошо известным примером является сборка ядра Linux, которая, в зависимости от типа оборудования, может требовать от нескольких десятков минут до часов процессорного времени. Усугубляет ситуацию то, что при работе над проектом (доработка кода, отладка, поиск ошибок, тестирование, ...) может понадобится до нескольких десятков, а то и сотен, пересборок проекта за один рабочий день. Возможности ускорения этого процесса в таких условиях становится актуальной...

На сегодня, когда практически не осталось в обиходе (или выходят из обращения) однопроцессорных (одноядерных) настольных компьютеров, сборку многих проектов можно значительно (в разы) ускорить, используя умение `make` запускать несколько заданий в параллель (ключ `-j`):

```
$ man make
...
    -j [jobs], --jobs[=jobs]
     Specifies the number of jobs (commands) to run simultaneously.  If there is more than one -j
     last  one is effective.  If the -j option is given without an argument, make will not limit
     the number of jobs that can run simultaneously.
```

Проверим как это работает. В качестве эталона для сборки возьмём проект NTP-сервера (выбран проект, который собирается не очень долго, но и не слишком быстро):

```
$ pwd
/usr/src/ntp-4.2.6p3
```

Вот как это происходит на 4-х ядерном процессоре Atom (не очень быстрая модель, частота 1.66Ghz) но с очень быстрым твердотельным SDD:

```
$ cat /proc/cpuinfo | head -n10
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 28
model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz
stepping        : 2
cpu MHz         : 1596.331
cache size      : 512 KB
$ make clean
$ time make -j1
...
real    2m7.698s
user    1m56.279s
sys     0m12.665s
$ make clean
$ time make -j2
```

```
...
real    1m16.018s
user    1m58.883s
sys     0m12.733s
$ make clean
$ time make -j3
...
real    1m9.751s
user    2m23.385s
sys     0m15.229s
$ make clean
$ time make -j4
...
real    1m5.023s
user    2m40.270s
sys     0m16.809s
$ make clean
$ time make
...
real    2m6.534s
user    1m56.119s
sys     0m12.193s
$ make clean
$ time make -j
...
real    1m5.708s
user    2m43.230s
sys     0m16.301s
```

Это работает! А вот та же компиляция на гораздо более быстром 2-х ядерном процессоре, но с типовым HDD:

```
$ cat /proc/cpuinfo | head -n10
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Pentium(R) Dual-Core CPU E6600 @ 3.06GHz
stepping        : 10
cpu MHz         : 3066.000
cache size      : 2048 KB
...
$ pwd
/usr/src/ntp-4.2.6p3
$ time make
...
real    0m31.591s
user    0m21.794s
sys     0m4.303s
$ time make -j2
...
real    0m23.629s
user    0m21.013s
sys     0m3.278s
```

Итоговая скорость здесь в 3-4 раза лучше, но улучшение от числа процессоров только порядка 20%, и это потому, что тормозящим звеном здесь является накопитель, при записи большого числа `.obj` файлов.

Примечание: В порядке предупреждения! - не всякая сборка `make`, которая успешно идёт на одном процессоре (как это имеет место по умолчанию, или при `-j1`), будет успешно происходить при большем числе задействованных процессоров. Это связано с нарушениями синхронизации операций в случаях сложных сборок. Самым наглядным примером такой сборки, завершающейся по ошибке, является сборка ядра Linux. Возможность параллельного выполнения `make` нужно проверить экспериментально для собираемого вами

проекта. В большинстве случаев это работает!

Если предыдущий способ ускорения был мотивирован тем обстоятельством, что нынче в обиходе подавляющее большинство компьютеров многопроцессорные (многоядерные), то следующий наш способ использует то обстоятельство, что объём памяти RAM современных компьютеров (2-4-8 Gb) значительно превышает объём памяти, необходимый для компиляции программного кода. В таком случае, компиляцию, основным сдерживающим фактором для которой является создание многих объектных файлов, можно перенести в область электронного диска (RAM диск, `tmpfs`), организованного в памяти:

```
$ free
             total       used       free     shared    buffers     cached
Mem:       4124164    1516980    2607184          0     248060     715964
-/+ buffers/cache:     552956    3571208
Swap:      4606972          0    4606972
$ df -m | grep tmp
tmpfs                    2014          1      2014   1% /dev/shm
```

Теперь мы можем перенести файлы (поддерево) того же, что и в предыдущем случае, собираемого проекта в `tmpfs`:

```
$ pwd
/dev/shm/ntp-4.2.6p3
$ make -j
...
real    0m4.081s
user    0m1.710s
sys     0m1.149s
```

Здесь использованы оба обсуждаемых способа ускорения одновременно, но улучшение относительно исходной компиляции достигает почти порядка (это показаны результаты для того компьютера, на котором из-за медлительности HDD параллельны сборка практически не давала выигрыша, и требовала порядка 30 секунд)!

Показательно посмотреть этот способ ускорения применительно к сборке ядра Linux, где, как было уже сказано, параллельная сборка не работает. Для такой сборки скопируем дерево исходных кодов ядра в каталог `/dev/shm`:

```
$ pwd
/dev/shm/linux-2.6.35.i686
$ time make bzImage
...
  HOSTCC  arch/x86/boot/tools/build
  BUILD   arch/x86/boot/bzImage
Root device is (8, 1)
Setup is 13052 bytes (padded to 13312 bytes).
System is 3604 kB
CRC 418921f4
Kernel: arch/x86/boot/bzImage is ready  (#1)

real    9m23.986s
user    7m4.826s
sys     1m18.529s
```

Очень неплохой результат: сборка ядра Linux менее чем в 10 минут.

Резюме этого краткого экскурса: тщательно оптимизируйте условия сборки вашего проекта под оборудование, на котором это производится, и, учитывая, что в процессе отладки сборка выполняется сотни раз — вы сэкономите множество времени!

Сборка модулей ядра

Частным случаем сборки приложений есть сборка модулей ядра Linux (драйверов), для сборки модуля (в ядрах 2.6.x) составляется `Makefile` построенный на использовании макросов, нам остаётся только записать (для файла кода с именем `mod_params.c`), как шаблон для сборки модулей:

Makefile :

```
CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)
TARGET = mod_params
obj-m      := $(TARGET).o
default:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
...
```

В результате:

```
$ make
make -C /lib/modules/2.6.18-92.el5/build M=/home/olej/2011-work/Linux-kernel/examples/modules-done_1/hello_printk modules
make[1]: Entering directory `/usr/src/kernels/2.6.18-92.el5-i686'
  CC [M]  /home/olej/2011-work/Linux-kernel/examples/modules-done_1/hello_printk/hello_printk.o
  Building modules, stage 2.
  MODPOST
  CC      /home/olej/2011-work/Linux-kernel/examples/modules-done_1/hello_printk/hello_printk.mod.o
  LD [M]  /home/olej/2011-work/Linux-kernel/examples/modules-done_1/hello_printk/hello_printk.ko
make[1]: Leaving directory `/usr/src/kernels/2.6.18-92.el5-i686'
$ ls -l *.o *.ko
-rw-rw-r-- 1 olej olej 74391 Мар 19 15:58 hello_printk.ko
-rw-rw-r-- 1 olej olej 42180 Мар 19 15:58 hello_printk.mod.o
-rw-rw-r-- 1 olej olej 33388 Мар 19 15:58 hello_printk.o
$ file hello_printk.ko
hello_printk.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
$ /sbin/modinfo hello_printk.ko
filename:       hello_printk.ko
author:         Oleg Tsiliuric <olej@front.ru>
license:        GPL
srcversion:     83915F228EC39FFCBAF99FD
depends:
vermagic:       2.6.18-92.el5 SMP mod_unload 686 REGPARM 4KSTACKS gcc-4.1
```

Прочий инструментарий создания программных проектов

Основной контингент, на кого ориентировано данное изложение, как уже замечалось ранее — это разработчики драйверов (модулей ядра), поддержки специфических аппаратных средств, или расширения набора системных утилит диагностики и управления. Именно поэтому, всё предыдущее изложение было акцентировано на использование языка программирования C. Это обусловлено ещё и тем, что сама система Linux (как ядро, так и большая часть набора GNU утилит) написаны на языке C (для ядра с незначительными включениями ассемблерного кода). Но для прикладного программиста Linux предоставляет весь спектр известных в IT средств разработки.

Другие языки программирования

Известно такое определение: «UNIX — это операционная система, которую писали программисты и для программистов». Всё то же самое относится в равной мере к Linux. В системе представлены практически все существующие языки записи программного кода. Проще сказать именно так: «все из существующих», чем пытаться перечислить то великое множество языков (заведомо больше 100), доступных программисту Linux. Если интересующее вас языковое средство отсутствует непосредственно в репозитарии вашего дистрибутива,

ищите его на сайтах сторонних разработчиков, и вы его обязательно найдёте.

Интегрированные среды разработки

Интегрированные средства (среды) разработки (IDE) не являются критически необходимым компонентом программной разработки. Но их использование часто позволяет производительнее вести отработку программного кода, оперативнее выполнять в связке цикл: редактирование кода — сборка проекта — отладка. Под Linux доступно весьма много разных IDE, различной степени интегрированности. Их уже настолько много, что становится бессмысленным описывать все, или значительную их часть в деталях: использование тех или иных IDE становится, в значительной мере, вопросом субъективных предпочтений и привычек. Можно перечислить только несколько из[20], числа наиболее широко используемых IDE:

1. Kdevelop (http://kdevelop.org/) - среда разработки KDE, активно развивается с 1999 г., помимо проектов языка C, позволяет вести проекты на C++ и Pascal.
2. Eclipse SDK (Eclipse Integrated Development Environment, http://www.eclipse.org/) - одна из наиболее известных на сегодня сред, активно развивается с 2000 г., сначала как проприетарный проект IBM, который затем был превращён в открытый проект. Отличительной особенностью является возможность динамических расширений (которые может подготовить и рядовой пользователь), за счёт этого наработаны плагины для поддержки десятков языков программирования, среди которых: Java, C/C++, PHP, Python и многих других, число которых постоянно прирастает из-за лёгкости работы с плагинами.
3. Oracle Solaris Studio (http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html - бывший проект Sun Solaris Studio), один из старейших проектов, изначально ориентирован на операционную систему Solaris, но там же есть альтернативная реализация для Linux. Solaris Studio обладает особыми оптимизирующими свойствами и нередко генерирует более эффективный и быстродействующий код, чем GCC. Ориентирован на языки программирования: C, C++ и Fortran, с дополнительными плагинами от сторонних производителей (устанавливаются непосредственно из Solaris Studio): Java, PHP, Python, Ruby (но это может потребовать дополнительной установки Oracle JDK).
4. IntelliJ IDEA (http://www.jetbrains.com/idea/), проект, активно развиваемый с 2000 г., ориентированный на язык Java, но имеющий развитые инструменты разработки и отладки под Android.

Эти, а также и другие, IDE вы легко найдёте и установите в своей системе под свой вкус, пользуясь техникой установки программного обеспечения, описываемой далее. Вряд ли этот предмет стоит большего внимания.

20 Не по принципу «эти лучше других», а только потому, что эти попросту «под руку попали».

Установка программного обеспечения

В Linux принято (в виду его GPL лицензии) и распространено несколько типов инсталляции нового программного обеспечения (здесь традиции существенно отличаются, скажем, от Windows):

- Бинарная установка: достаточно редкий для Linux способ, весьма напоминающий инсталляцию в Windows, когда запускается на выполнение программа или скрипт, результатом которой является установленный пакет. Так устанавливаются некоторые виды проприетарного программного обеспечения.
- Установка из репозитария с помощью пакетной системы Linux (выбранного вами дистрибутива). В этом случае установка делается с помощью программы менеджера пакетной системы, который использует установочные пакеты в свойственном ему формате (в зависимости от дистрибутива). Файлы пакетов могут использоваться либо локальные, либо, в последнее время, ищутся на разнообразных ресурсах в глобальной сети. Преимущества пакетной установки: простота процесса (доступно любому начинающему пользователю) и то, что при установке пакетов контролируется их взаимозависимость: если для установки пакета ещё нет установленных в системе пакетов от которых данный зависим, то менеджер пакетной системы либо устанавливает всю иерархию пакетов, либо прерывает инсталляцию до тех пор, пока зависимости не будут восстановлены.
- Установка из исходных кодов. Это изначально традиционный для Linux путь наполнения системы программными пакетами. За время существования, здесь сложилось много разных подвидов того, как это делается, но общим у всех них остаётся одно: программные средства предоставляются в исходном коде (это требование лицензии GPL) и компилируются с целевой системе, после чего устанавливаются в соответствии со специфическими требованиями пакета. Преимущества: полный контроль за программным обеспечением системы, всегда самые свежие версии, отсутствие любых подозрений на «закладки», вирусы и другие неприятности. Недостатки: требует достаточно высокой квалификации, даже при наличии такой квалификации требует изрядной сообразительности и изобретательности, слабый контроль зависимостей пакетов и версий — можно легко создать неработоспособную инсталляцию.

Бинарная установка

Бинарная установка, то есть установка исполнимых файлов, готовых для выполнения, или выполнением исполнимых инсталляционных программ (скриптов) — крайне редкая практика в открытых POSIX системах:

1. Так реализовывался старый способ распространения бинарных пакетов «разархивированием от корня», который состоял в том, что нужно было разархивировать `tar`-архив в корень файловой системы, при этом нужные файлы из архива разлягутся в нужные каталоги файловой иерархии (ныне почти не используется, но можно встретить в других POSIX системах).

Такой способ бинарной установки мы можем наблюдать на примере популярного пакета Oracle Solaris Studio, когда вы получаете архив вида `SolarisStudio12.2-linux-x86-tar-ML.tar.bz2`, и разархивируете содержащийся в нём (помимо разнообразных README файлов) каталог `solstudio12.2` (больше 800Mb) в место установки (для Linux это обычно `/opt/oracle`, который нужно создать). В итоге получаем:

```
$ pwd
/opt/oracle/solstudio12.2
$ ls
bin  examples  include  LEGAL  lib  man  netbeans  prod  READMEs
$ du -hs
884M    .
```

Всё, что нам нужно сделать помимо этого, это прописать (например, в `bashrc`):

```
PATH=$PATH:/opt/oracle/solstudio12.2/bin ; export PATH
MANPATH=$MANPATH :/opt/oracle/solstudio12.2/man; export $MANPATH
```

После чего уже можете запускать интегрированную среду разработки:

```
$ solstudio &
```

Это был законченный пример такой странной для Linux бинарной инсталляции...

2. Ещё одним (но иным) способом бинарной установки до последнего времени распространялись Java средства от Sun: JDK и JRE (теперь это делает Oracle). Например, инсталляционный файл: `jre-6u18-linux-i586.bin` - это shell-скрипт до смещения примерно 0x3528, за которым следует огромный бинарный код. Точно так же распространяется свободное программное обеспечение NVIDIA, проприетарные системы драйверов видеокарт компании (в качестве образца файл: `NVIDIA-Linux-x86-280.13.run`), или весь набор программных средств (в качестве образца файл: `cudatoolkit_4.0.17_linux_32_fedora13.run`) - обеспечение технологии CUDA для параллельных многопроцессорных GPU вычислений. Всё это свободное, но не открытое программное обеспечение.

Для всех таких проприетарных установок нужно не упускать из виду, что перед их выполнением нужно предварительно (после загрузки дистрибутива из сети) установить флаг исполнимости (команда `chmod`) на такой инсталляционный файл, например вот так:

```
$ chmod a+x jre-6u18-linux-i586.bin
$ ./jre-6u18-linux-i586.bin
...
```

Или, если вы не хотите этого делать, нужно указать командному интерпретатору, что это файл ему для исполнения, так:

```
$ sh jre-6u18-linux-i586.bin
...
```

Или вот так:

```
$ bash jre-6u18-linux-i586.bin
...
```

Способы бинарной установки не имеют, с точки зрения потребителя, ни единого преимущества, за исключением, возможно, их исключительной простоты, и можно было бы стать в позу и принципиально не использовать такие продукты в Linux... Можно было бы, если бы не то, что таким форматом распространяются некоторое число весьма широко употребимых пакетов, например: Sun/Oracle JDK/JRE, IDE NetBeans, IDE Solaris Studio, Skype, уже названные программные средства от NVIDIA.

Примечание: Есть ещё одна особенность, свойственная бинарной инсталляции в Linux, которую следует держать в уме... Если бинарный программный пакет использует или строит **модуль ядра** Linux (чаще всего это драйверы), то такой установленный программный пакет потеряет работоспособность как только вы обновите версию ядра вашей инсталляции Linux, хотя бы только по наименованию, и хотя бы даже из репозитария обновлений вашей версии дистрибутива... Это происходит из-за щепетильности контроля соответствия версий ядра и его модулей. Эту неприятность можно обойти, но это а). достаточно хлопотно, б). не легально, в). с неконтролируемыми последствиями. Такая история наблюдается, например, из числа достаточно используемых, с средством Cisco Systems VPN Client.

Пакетная установка

Формат пакетной системы и программы пакетного менеджера — являются отличительной особенностью той дистрибутивной линии, к которой принадлежит ваш экземпляр Linux. Наиболее известными из таковых являются: формат пакета `.deb` и менеджер пакетов `apt` в дистрибутивах Debian и Ubuntu, и их альтернатива — пакетный формат `.rpm` и менеджеры `rpm` и `yum` в дистрибутивах ReaHat, Fedora, CentOS, Mandriva. Функционально и по принципам деятельности альтернативные пакетные системы подобны, и мы далее, для определённости, рассматриваем одну линию: `rpm` и `yum`.

Примечание: В самые последние годы наметилась тенденция предоставлять (даже в составе основного репозитария дистрибутива) программы (пакеты) для работы с альтернативными форматами пакетной системы (пакетами других дистрибутивов), например: работа с `rpm` в Debian, работа с `deb` в Fedora и так далее...

Логика пакетной системы состоит в том, что файл инсталляционного пакета содержит не только сами устанавливаемые файлы, но и требования по зависимостям данного пакета от ранее установленных пакетов: текущий пакет может быть установлен только если удовлетворены все требуемые для него зависимости. То же самое относится и к процессу удаления пакетов: может быть удалён только такой пакет, на который по зависимостям не ссылается уже ни один установленный в системе пакет. Для обеспечения таких функция поддержания целостности установленной системы программного обеспечения, менеджер пакетов ведёт в той или иной форме базу данных пакетной системы.

Пакетная система rpm и менеджер yum

Пакетная система `rpm` обслуживается набором утилит пакетного менеджера:

```
$ ls /usr/bin/rpm*
/usr/bin/rpm2cpio  /usr/bin/rpmbuild  /usr/bin/rpmdb  /usr/bin/rpmgraph  /usr/bin/rpmquery
/usr/bin/rpmsign  /usr/bin/rpmverify
$ rpm --version
RPM версия 4.4.2
```

Позже, над подсистемой пакетного менеджера `rpm`, и под явным влиянием пакетной `apt` системы Debian, была надстроен мета-менеджер `yum`. Если менеджер `rpm`, используется для установки пакетов из файлов пакетов вида `<имя>.rpm`, то мета-менеджер `yum` используется для поиска файла пакета в известных сетевых репозитариях, загрузки файла и установки его в систем (конечная фаза установки происходит при участии всё того же менеджера `rpm`, но это происходит скрыто для пользователя, и не имеет для него значения). Частным случаем сетевого места расположения файла пакета является файл на локальном хосте, поэтому `yum` может с таким же успехом использоваться и для локальной инсталляции, полностью заменяя в этом качестве менеджер `rpm`.

Вот так делается проверка наличия (поиск) требуемых пакетов (заданы маской имени) в сетевых репозитариях:

```
# yum list available djvu*
...
Доступные пакеты
djvulibre.i686                    3.5.21-3.fc12                fedora
djvulibre-devel.i686              3.5.21-3.fc12                fedora
djvulibre-mozplugin.i686          3.5.21-3.fc12                fedora
```

Вот так делается установка найденного пакета (или группы пакетов по маске, как в примере) из репозитария:

```
# yum install djvu*
...
New leaves:
  djvulibre-devel.i686
  djvulibre-mozplugin.i686
```

Перечень известных `yum` сетевых репозитариев — величина конфигурируемая, сам список репозитариев вы найдёте в виде файлов каталога `/etc/yum.repos.d`, а процесс конфигурирования `yum` - в сопутствующей ему обстоятельной документации.

Вот как производится установка пакета из локального файла (`*.rpm`) :

```
# yum --nogpgcheck localinstall djvulibre-3.5.18-1.fc7.i386.rpm
...
```

Особенностью здесь есть то, что может понадобиться указать не использовать PGP-сигнатуры, подтверждающие аутентичность пакета (при установке из сетевых репозитариев такая проверка установлена по умолчанию).

Установку из локального архива можно делать и непосредственно установщиком RPM-пакетов, утилитой `rpm`.

Примечание: Почему инсталляция `*.rpm` с помощью выше приведенной команды `yum` лучше, чем более

старой и традиционной командой установки?:

```
# rpm -i djvulibre-3.5.18-1.fc7.i386.rpm
```

Потому (предположительно?), что при установке `yum` установленный пакет учитывается в единой базе данных `yum`, и это сказывается при учёте зависимостей при установке и удалении других пакетов.

Пакеты исходных кодов

В некоторых случаях вы получаете (скачиваете из сети) файлы пакетов вида `*.src.rpm`. Это пакеты исходных кодов, которые собраны той же утилитой создания rpm-пакетов (`rpmbuild`), но результатом установки такого пакета будет исходный программный код, который нужно компилировать... мы об этом поговорим далее. Но при установке пакета исходных кодов вида `*.src.rpm` вам `yum` может не помочь, дело закончится таким примерно сообщением:

```
# yum localinstall esvn-0.6.12-1.src.rpm
...
Проверка esvn-0.6.12-1.src.rpm: esvn-0.6.12-1.src
Невозможно добавить пакет esvn-0.6.12-1.src.rpm в список действий. Несовместимая архитектура: src
Выполнять нечего
```

Для таких случаев нужно воспользоваться непосредственно командой `rpm` (у неё множество опций-возможностей, но они все достаточно полно описаны: `rpm --help` , `man rpm`, и др.):

```
# rpm -i -vvv esvn-0.6.12-1.src.rpm
D: ============== esvn-0.6.12-1.src.rpm
D: Expected size:      1930964 = lead(96)+sigs(180)+pad(4)+data(1930684)
D:   Actual size:      1930964
...
D: ========== Directories not explicitly included in package:
D:          0 /root/rpmbuild/SOURCES/
D:          1 /root/rpmbuild/SPECS/
D: ==========
...
```

Будут добавлены (или созданы) каталоги (в `$HOME`, в показанном случае это выполнялось от имени `root`):

```
# ls -R /root/rpmbuild
/root/rpmbuild:
SOURCES  SPECS
/root/rpmbuild/SOURCES:
esvn-0.6.12-1.tar.gz
/root/rpmbuild/SPECS:
esvn.spec
```

Дальше вы уже разбираетесь с установленным исходным кодом ... как и с исходным кодом любого другого происхождения (см. далее).

Создание собственного инсталляционного пакета

Для построения с целью дальнейшего распространения собственных инсталляционных пакетов используется утилита `rpmbuild`. Детально техника создания тиражируемых пакетов выходит за рамки рассмотрения, но очень кратко это выглядит так:

– создаётся текстовый файл сценария, с расширением `*.spec`, например, для уже упоминаемого пакета `esvn-0.6.12-1.src.rpm` это `esvn.spec`;

– редактируется текст сценария, который имеет фиксированную структуру секций, состоит из целей и макросов, начинаемых в записи с символа `%`:

```
$ cat esvn.spec
Summary: Graphical frontend for subversion
```

```
Name: esvn
Version: 0.6.12
Release: 1
License: GPL
...
Vendor: eSvn
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root
...
%build
%{__make} %{?_smp_mflags}
%install
%{__rm} -rf %{buildroot}
%{__install} -Dp -m0755 esvn %{buildroot}%{_bindir}/esvn
...
```

– этот сценарий создания пакета передаётся в качестве параметра утилите `rpmbuild`.

Этот процесс, внешне громоздкий, доступен для использования любому средней квалификации разработчику, и хорошо документирован для использования.

Инсталляция из исходников

Инсталляция из исходных текстов программного обеспечения — это основной путь обновления программных средств для разработчиков программистов. Когда-то изначально это был первый и единственный способ распространения программного обеспечения для Linux. Несомненными преимуществами инсталляции из исходных текстов являются: максимальная гибкость и то, что вы всегда имеете самую свежую реализацию нужных вам программных средств. Но и у этого способа есть свои недостатки:

- Установка программных средств способом компиляции исходных текстов и их последующей установки требуют, как минимум, некоторых навыков программиста (для компиляции) и некоторых навыков администратора (для установки). Но не все пользователи Linux являются и программистами и администраторами в одном лице...
- Если трудности предыдущего пункта, это дело наживное, при желании, то гораздо существеннее есть то, что при установке из исходных кодов производится только минимально ограниченный контроль (и это в лучшем случае, на этапе `./configure`) зависимостей, требований наличия других пакетов и библиотек, без которых ваш установленный пакет останется неработоспособным...
- Особым вариантом предыдущего пункта является конфликт версий, когда вы устанавливаете следующую версию пакета, при наличии установленной его предыдущей версии ... здесь могут быть совершенно любые чудеса. Особый подвид этого пункта являет собой установка 2-х версий пакета, но с разными базовыми каталогами инсталляции (параметр `--prefix` при `./configure`) — здесь уже тот случай, что «... хоть святых выноси»;
- Ещё одну проблему составляют коллизии инсталляций из исходных кодов, и инсталляций с помощью пакетной системы, и бинарных инсталляций. А без смешения (и потенциальных коллизий) здесь не обходится, потому как даже при полной вашей приверженности инсталляции из исходных кодов: а).огромное число начальных пакетов дистрибутива (при начальной CD-инсталляции) ставится пакетной системой, и б).есть ряд средств (JDK, Skype, ...), для которых придётся мириться с бинарной инсталляцией, поскольку других для них просто не существует в природе.

В любом случае, установка из исходных текстов остаётся всё равно больше искусством, чем штатной операцией. Но и относительно неё можно выделить несколько типовых случаев представления пакета, которые нужно идентифицировать по внешним признакам поставки. В любом случае, прежде всего читаем файл `README` в исходном каталоге, чтобы понять какой тип инсталляции перед нами. Из таких типовых случаем можно выделить несколько наиболее частых:

а). пакеты для непосредственной сборки;

б). пакеты, подготовленные средствами `Autoconf/Automake` (самый частый на сегодня случай);

в). пакеты, подготовленные `Cmake`;

Показать сборку пакета из исходных кодов для всех этих случаев, можно наилучшим образом только на примерах таких сборок.

Но прежде, чем приступить к инсталляции исходного пакета, вам, чаще всего, придётся ещё провести его некоторую пред подготовку. Основная масса пакетов исходных кодов поступает в виде архивов самых разных форматов, чаще это `*.tgz (*.tar.gz)`, или `*.tar.bz2` , или `*.tar.Z` (могут быть и существенно более экзотические случаи архивирования). Первым делом помещаем архив в место распаковки — удачным выбором будут `$HOME` или `/usr/src`. Следующим шагом разворачиваем такой архив в каталог (дерево) исходных кодов:

```
$ tar -zxvf xxx.tar.gz
...
```

Или (если это `*.bz2`):

```
$ tar -jxvf xxx.tar.gz
...
```

Теперь у вас есть дерево (поддерево) исходных кодов требуемого программного пакета.

Примечание: полученное на этом шаге дерево исходных кодов часто получают из репозитариев SVN или GIT, как их распространяют разработчики пакетов, операцией update клиента используемой системы контроля версий.

Непосредственная сборка

К этой категории я отнёс несколько различающихся вариантов, но общим для них будет то, что почти все они бывают представлены в пакетах программ, обычно, небольших: учебных иллюстрационных, проектов на ранних этапах развития и подобных... Но отличительной чертой их всё таки будет наличие файла сценария сборки `Makefile` (при отсутствии файла `./configure` или других признаков конфигурации пакета). Это не такой уж часто встречающийся случай, отличительной чертой такого пакета будет наличие в разархивированном каталоге файла `Makefile` с **датой создания** этого файла в некотором обозримом **прошлом** (файл не сгенерирован непосредственно в ходе манипуляций с каталогом). В таких пакетах можно попробовать просто выполнить последовательность команд:

```
$ make
...
$ sudo make install
...
```

В некоторых случаях, так же выглядят пакеты, предназначенные для обработки-сборки утилитой `qmake` (из пакета Qt3), которая либо должна быть у вас в системе, либо должна быть доустановлена для продолжения сборки. Хорошим примером такой непосредственной сборки есть установка пакета `eSVN` — удачная реализация GUI обёртки (может быть свободно получена из сети) для работы с системой поддержания версий `subversuon`:

```
$ make
qmake esvn.pro
make: qmake: Команда не найдена
make: *** [esvn] Ошибка 127
```

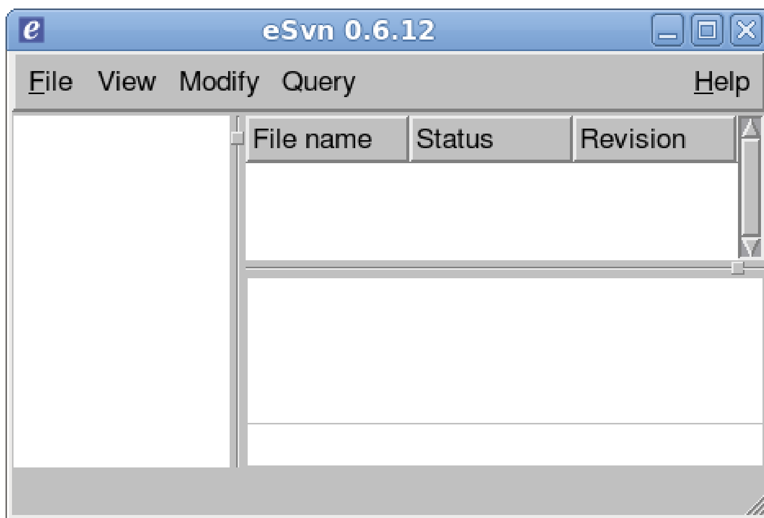
- у нас нет в системе утилиты `qmake` (не путать с `gmake`), которая часто является алиасом `make`, но не обязательно, и не во всех системах) и чаще всего именно так и бывает. Мы можем, судя уже по написанию (да и по назначению пакета), полагать, что это утилита из комплекта графических средств `Qt`. Проверяем это предположения:

```
# yum list all qt3*
...
Установленные пакеты
qt3.i686                        3.3.8b-28.fc12                        @fedora
Доступные пакеты
...
qt3-devel.i686                  3.3.8b-28.fc12                        fedora
# yum install qt3-devel*
```

```
...
# which qmake
/usr/lib/qt-3.3/bin/qmake
$ qmake
Usage: qmake [mode] [options] [files]
QMake has two modes, one mode for generating project files based on
some heuristics, and the other for generating makefiles.
...
```

- теперь у нас всё необходимое есть, и можно продолжить сборку:

```
$ make
qmake esvn.pro
make -f esvn.mak
make[1]: Entering directory `/usr/src/esvn'
...
make[1]: Leaving directory `/usr/src/esvn'
** done **
$ sudo make install
...
$ ./esvn
...
```



Мы получили непосредственной сборкой (без какого либо конфигурирования) пакет, пригодный для дальнейшего использования, как это показано на рисунке.

Autoconf / Automake

До настоящего времени это всё ещё остаётся наиболее частая форма поставки программного пакета в исходных кодах. Это весьма старый инструментарий (с 1991г.), который включил в себя за время развития целый ряд дополнительных пакетов (например `libtools` — пакет конфигурирования библиотек). Отличительным признаком таких пакетов является: наличие в каталоге файла скрипта `configure` с правами исполнения. Рассмотрим общие принципы такой сборки на примере очень крупного проекта VoIP PBX `FreeSwitch`, исходный пакет распакован в каталог:

```
$ pwd
/usr/src/freeswitch-1.0.6
```

В большинстве конфигурируемых пакетах файл `configure` допускает запуск с ключом `--help`, дающем подсказку по возможным параметрам установки, отличающихся от дефаултных:

```
$ ./ configure --help
`configure' configures freeswitch 1.0.6 to adapt to many kinds of systems.
Usage: ./configure [OPTION]... [VAR=VALUE]...
```

```
...
Installation directories:
 --prefix=PREFIX   install architecture-independent files in PREFIX [/usr/local/freeswitch]
 --exec-prefix=EPREFIX  install architecture-dependent files in EPREFIX [PREFIX]
By default, `make install' will install all the files in
`/usr/local/freeswitch/bin', `/usr/local/freeswitch/lib' etc.  You can specify
an installation prefix other than `/usr/local/freeswitch' using `--prefix',
for instance `--prefix=$HOME'.
...
Some influential environment variables:
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  CPPFLAGS    C/C++ preprocessor flags, e.g. -I<include dir> if you have
              headers in a nonstandard directory <include dir>
  CXX         C++ compiler command
  CXXFLAGS    C++ compiler flags
  CPP         C preprocessor
  CXXCPP      C++ preprocessor
...
```

Дальше (возможно определившись с опциями, которые мы укажем с `./configure ...`), выполняется достаточно типовая последовательность действий:

```
$ ./configure
...
```

Наиболее часто изменяемым параметром инсталляции является корневой путь установки пакета; весьма часто разработчики в качестве дефаултного пути установки задают `/usr/local`, но дистрибьюторы (или пользователи при установке) переопределяют этот путь в `/usr` или `/opt`. Достигается это выполнением с такими опциями, как:

```
$ ./configure --prefix=/usr
...
$ ./configure --prefix=/opt
...
```

После этой фазы (если она завершается без ошибок) должен быть создан файл сборки `Makefile`. Далее следует совершенно стандартная последовательность команд (для больших пакетов `make` я рекомендую выполнять с префиксной командой `time`, чтобы на будущее планировать сколько времени может занимать такая сборка):

```
$ time make
....
 +---------- FreeSWITCH Build Complete ----------+
 + FreeSWITCH has been successfully built.       +
 + Install by running:                           +
 +                                               +
 +               make install                    +
 +-----------------------------------------------+
real    15m25.832s
user    11m29.939s
sys     3m41.169s
$ su -c 'make install'
...
```

Примечание: Если команды `./configure` и `make` могут успешно выполняться от имени ординарного пользователя, то последняя операция инсталляции — требует прав `root`.

В некоторых пакетах между `./configure` и `make` может быть предусмотрена фаза конфигурирования состава

пакета:

```
$ make config
```

или

```
$ make menuconfig
```

Наличие такой возможности легко определить, просматривая Makefile, созданный ./configure, на наличие соответствующих целей.

Создание своего конфигурируемого пакета

При некоторых навыках и сообразительности, выполнять сборку и установку, как показано выше, обычно не составляет труда (возможно, на каждом шаге анализируя сообщения об ошибках и подправляя параметры). Гораздо больше изобретательности требуется чтобы сделать свой оригинальный проект в такой же степени конфигурируемым под разные варианты операционной системы. Рассматриваем, как наиболее употребимый, инструментарий Autoconfig : очень упрощённо и по шагам проделаем конфигурирование над ранее рассмотренным проектом сборки со статической библиотекой (архив libraries.tgz) программы hello (теперь это архив Autoconf.tgz):

```
$ ls
hello_child.c  hello_child.h  hello_main.c  Makefile.0
```

Здесь файл сценария сборки Makefile, использовавшийся при отработке целевого проекта переименован в Makefile.0, потому как в результате конфигурирования будет создаваться новый Makefile.

Прежде всего, нам предстоит составить файл configure.in, содержащий макросы для тестов проверок в новой операционной системе. Но мы не станем делать это сами, а воспользуемся утилитой autoscan, которая создаст нам configure.scan как прототип будущего configure.in :

```
$ autoscan
$ ls
autoscan.log  configure.scan  hello_child.c  hello_child.h  hello_main.c  Makefile.0
$ cat configure.scan
AC_PREREQ([2.63])
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
AC_CONFIG_SRCDIR([hello_child.h])
AC_CONFIG_HEADERS([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Это только прототип-шаблон, в который нам предстоит вписать много других макросов тестов зависимостей нашего пакета. Конечным действием этого шага должно стать:

```
$ cp configure.scan configure.in
$ ls
autoscan.log  configure.in  configure.scan  hello_child.c  hello_child.h  hello_main.c  Makefile.0
```

Далее создаём config.h.in, но опять же воспользуемся для этого генератором заготовки autoheader :

```
$ autoheader
$ ls
autom4te.cache  autoscan.log  config.h.in  configure.in  configure.scan  hello_child.c
hello_child.h  hello_main.c  Makefile.0
$ cat config.h.in
/* config.h.in.  Generated from configure.in by autoheader.  */
```

```
/* Define to the address where bug reports for this package should be sent. */
#undef PACKAGE_BUGREPORT
/* Define to the full name of this package. */
#undef PACKAGE_NAME
...
```

Здесь нет ничего интересного, потому как предполагается, что вы станете сознательно править `config.h.in`, наполняя его этим интересным содержанием сами...

И далее — ключевое действие:

```
$ autoconf
$ ls
autom4te.cache  autoscan.log  config.h.in  configure  configure.in  configure.scan  hello_child.c
hello_child.h  hello_main.c  Makefile
```

- создан файл `configure`, это скрипт с установленными флагами выполнимости!

Цель выполнения `configure` — создание из макета сценария сборки `Makefile.in` текущего сценария `Makefile` под конкретно сконфигурированное окружение операционной системы. Для этого предварительно скопируем этот файл макета из сценария сборки, который мы используем при отладке пакета:

```
$ cp Makefile Makefile.in
```

И выполняем лёгкое редактирование `Makefile.in` — выделяем в нём конфигурируемые параметры, в данном примере такой параметр один:

```
$ cat Makefile.in
...
LIB = lib$(TARGET)
CC = @CC@
all: $(LIB) $(TARGET)
$(LIB):   $(CHILD).c $(CHILD).h
          $(CC) -c $(CHILD).c -o $(CHILD).o
...
```

Фактически, это практически тот же `Makefile`, который мы использовали при компиляции проекта, за одним принципиальным исключением: переменной `@CC@`. Значения переменным вида `@XXXX@` будет присваиваться при выполнении скрипт `configure`, исходя из анализа тестов, которые скрипт проводит над операционной системой.

Всё! Мы можем проверять выполнение созданного конфигурационного скрипта:

```
$ ./configure
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
```

Конечным действием скрипта является создание нового `Makefile`, который мы тут же проверяем:

```
$ make
cc -c hello_child.c -o hello_child.o
ar -q libhello.a hello_child.o
```

```
ar: creating libhello.a
ar -t libhello.a
hello_child.o
cc hello_main.c -Bstatic -L./ -lhello -o hello
```

- это в точности то, что выполнял ранее созданный вручную сценарий `Makefile.0`.

Примечание: Основным содержательным действием, определяющим успех всего начинания, является наполнение файла `configure.in` (обсуждался выше) макросами тестирования возможностей операционной системы. Созданный файл начинается с макроса `AC_INIT` и завершается макросом `AC_OUTPUT` (как показано ранее в примере), между которыми можно вписывать свои собственные тесты-проверки, для очень многих требуемых случаем уже существуют предопределённые макросы. Примеры только некоторых таких макросов тестирования (для оценки разнообразия возможностей):

– `AC_CHECK_HEADERS()` используется для проверки существования конкретных заголовочных файлов (указаны параметрами). Пример:

  `AC_CHECK_HEADERS([stdlib.h string.h sys/param.h unistd.h])`

– `AC_TRY_COMPILE()` - который пытается откомпилировать небольшую вами представленную (прямо в качестве его параметра) программу, которая проверяет возможность использования заданной синтаксической конструкции текущим компилятором; также может использоваться для проверки структур и полей структур, которые присутствуют не во всех системах.

– `AC_TRY_LINK_FUNC()` - для проверки библиотеки, функции или глобальной переменной скрипт `configure` попытается скомпилировать и скомпоновать небольшую программу, которая использует тестируемые возможности: создается тестовая программа для того, чтобы убедиться, что программа, чье тело состоит их прототипа и вызова указанной функции (параметр макроса), может быть скомпилирована и скомпонована.

– `AC_TRY_RUN()` - тестирует поведение (отсутствие ошибок) периода выполнения указанной программы (параметр макроса).

– `AC_FUNC_MALLOC, AC_FUNC_VPRINTF` – проверка доступности функций из фиксированного набора.

– `AC_CHECK_FUNCS` – проверка доступности функций из списка параметров. Пример:

  `AC_CHECK_FUNCS([bzero strrchr])`

Это показана только весьма малая часть макросов, предоставляемых для выполнения проверок. Детальное описание всех возможных макросов смотрите в описаниях `Autoconf`.

Cmake

`Cmake` — это ещё одна, относительно новая, не зависящая от платформы (Linux, Windows, etc.) система конфигурирования пакетов исходных кодов под различия платформы. В проекте `KDE` после версии 3 система `Cmake` была выбрана основным инструментом конфигурирования, что обеспечило ей быструю динамику развития. `Cmake` должна быть дополнительно установлена как пакет. Имеет как консольный, так и GUI интерфейс.

```
$ which cmake
/usr/local/bin/cmake
$ cmake --help
cmake version 2.6-patch 3
Usage
  cmake [options] <path-to-source>
  cmake [options] <path-to-existing-build>
...
Generators
The following generators are available on this platform:
  Unix Makefiles              = Generates standard UNIX makefiles.
  CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.
  Eclipse CDT4 - Unix Makefiles
                              = Generates Eclipse CDT 4.0 project files.
```

```
  KDevelop3                   = Generates KDevelop 3 project files.
  KDevelop3 - Unix Makefiles  = Generates KDevelop 3 project files.
```

Особенно интересен последний абзац: `Cmake` может генерировать конфигурацию сборки для различных интегрированных средств разработки (`Eclipse, Kdevelop, ...`), дальнейшие действия по сборке после `Cmake` выполняются уже рассмотренными ранее средствами. Достаточно много пакетов предоставляются для конфигурирования и сборки средствами `Cmake`, признаком того часто является наличие в каталоге исходных кодов файла: `CMakeLists.txt`. Примером представления пакета для сборки может быть взят пакет GUI оболочки QEMU (дистрибутив: `qtemu-*.tar.bz2`), указание пути к сборке в команде (текущий каталог `./`) — обязательно:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/opt/qtemu .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
...
```

Портирование POSIX программного обеспечения

Портированием называют перенос программных пакетов с открытым кодом из одной операционной системы в другую (возможность существования которой, возможно, и не предполагалась во время создания переносимого пакета). В любом случае, компиляция и сборка чужого программного проекта всегда будет оставаться процессом поисковым, и не подлежит формальному выполнению. При этом процесс этот может завершиться неудачей, не взирая на любые затраченные усилия (он может просто потенциально не собираться для данной операционной системы). Тем не менее, на удивление часто и легко удаётся перенести в Linux программный пакет, написанный лет 10 назад и для какой-то несуществующей UNIX системы... Это проявление силы стандартизации POSIX. Ниже описаны несколько эмпирических пунктов (на уровне советов), которые могут значительно упростить перенос программных проектов в Linux, или наоборот, из Linux в другие системы.

Для облегчения определения характеристик используемой архитектуры оборудования и операционной системы используются файлы конфигурации `config.guess` и `config.sub`, поддерживаемые в актуальном (современном) состоянии на протяжении многих лет (поэтому нужно получить как можно более свежие копии этих скриптов):

```
$ cat config.guess | head -n7
# Attempt to guess a canonical system name.
#   Copyright (C) 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999,
#   2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010,
#   2011 Free Software Foundation, Inc.
timestamp='2011-02-02'
```

Этим скриптам нужно присвоить флаг исполнимости, или выполнять их как файлы данных для командного интерпретатора:

```
$ sh -e config.guess
i686-pc-linux-gnu
```

- как показано здесь, `config.guess` тестирует и определяет каноническое имя архитектуры и операционной системы где он выполняется. Файлы конфигураций обновляются очень часто, и включают определения для самых замысловатых операционных систем и аппаратных конфигураций. Пользующийся ними скрипт `./configure` (если он использует эти конфигурационные скрипты) может очень гибко настраиваться на конкретную конфигурацию. Грамотно написанные скрипты `./configure` часто используют `config.guess` и `config.sub`, а сами файлы файлы включаются в каталог проекта. В таких случаях бывает совсем не лишним обновить свежие копии этих скриптов.

Далее... Процесс сборки инородного проекта в Linux может прерваться (сообщением об ошибке) на различных этапах: а). `./configure`, б). компиляции, в). сборки (линковки). Рассмотрим эти случаи поочерёдно... Основной наш инструмент — это именно тщательный анализ полученного сообщения об ошибке:

Ошибки на этапе **конфигурирования**:

- Чаще всего выявляются в отсутствии других, ранее установленных пакетов (проектов, библиотек...), от которых зависим собираемый проект. В таком случае ищем (в сети) исходный код недостающего проекта и начинаем его сборку и установку. Этот пакет может, в свою очередь, потребовать недостающих зависимостей... Таким образом мы выстраиваем для себя дерево иерархий требуемых пакетов, и собираем их в обратном порядке, от листьев к корню... Вы мне не поверите, но как часто это приводит к успеху!
- Ещё одна несоответствие конфигураций — это несоответствие корневых каталогов установки разных зависимых пакетов (дефаултные корневые каталоги установки пакетов не совпадают). Например, большинство проектов Solaris по умолчанию устанавливается с префиксом `/opt`, и там же ищет библиотеки. Решаем это согласованным использованием опции `--prefix=...` команды `./configure`.
- Скрипт конфигурирования исходного исходного проекта может быть написан с использованием синтаксических расширений конкретного командного интерпретатора (`bash` сам по себе имеет очень значительные синтаксические расширения). В таких случаях совсем не лишним может оказаться попробовать выполнение под другим интерпретатором:

```
$ ksh
$ ./configure
...
$ exit
```

  Или:

```
$ tcsh
$ ./configure
...
$ exit
```

Примечание: Для использования дополнительных командных интерпретаторов, как показано выше, вам может понадобиться, возможно, их дополнительно установить:

```
$ sudo yum install ksh.i686
...
Установлено:
  ksh.i686 0:20100621-1.fc12
$ sudo yum install tcsh.i686
...
Установлено:
  tcsh.i686 0:6.17-5.fc12
```

Ошибки периода **компиляции**. Чаще всего причины тому бывают (в порядке частоты их проявления):

- Сообщения о неопределённых именах (определений функций) периода компиляции — это самая частая ошибка. Она указывает либо на то, что нужно включить дополнительные `#include` директивы (если удаётся найти определения требуемых функций в `*.h` файлах), либо сразу указывают на невозможность сборки, если поиск теперь уже известного вам имени по всем заголовочным файлам `/usr/include` не приводит к успеху. Найдите требуемый (содержащий нужное имя) файл-хедер в каталоге `/usr/include` и во всех его подкаталогах, и добавьте директиву `#include` с именем найденного файла.
- Часто простейший способ выяснить, какого заголовочного файла не хватает для требуемого имени — это выполнить команду справки относительно этого имени:

```
$ man <имя>
```

  И первой строке справки (если она найдётся) вы найдёте имя заголовочного файла, например, для

имени `strcat` в справке присутствует строка:

```
#include <string.h>
```

(А во второй строке справки будет указано имя библиотеки, где размещён требуемый вызов — возьмите его на заметку, он тут же нам может понадобиться для разрешения имён для следующего этапа связывания).

- Компиляторы GNU `gcc`, Solaris `cc` (Solaris Studio) и другие (например `pcc`), которыми, возможно, собирался проект, допускают синтаксические расширения (и значительные, и отличающиеся в каждом случае) относительно стандартов, и свои специфические прагмы компилятора — всё это может быть не распознано вашим компилятором (нестандартные расширения в `gcc` добавляются от версии к версии). Ищите замену нестандартной конструкции — она обычно проста, и лежит на поверхности. Простейший способ — попробовать просто убрать нестандартную конструкцию, часто такие синтаксические расширения, свойственные только одному конкретному компилятору, удаётся просто безболезненно комментировать. Вот пример из реального портирования:

```
void _db_enter_( const char *_func_, const char *_file_, uint _line_,
                 const char **_sfunc_, const char **_sfile_,
              /* uint * _slevel_, char ***_sframep_ __attribute__ ((unused))) */
                 uint * _slevel_, char ***_sframep_ )
```

Ошибки периода **связывания** (линковки). Здесь чаще всего:

- Объявлено внешнее неразрешённое имя. Сообщение примерно такого вида:

```
/usr/tmp/ccpQfRNO.o:(.text+0x9): undefined reference to `_mcount'
```

  Скорее всего, это означает, что в строке сборки gcc не указана явно какая-то из библиотек. В разных POSIX системах одни и те же широко используемые библиотеки должны или указываться явно, либо это необязательно. Пример: в Linux указание библиотеки `libpthread.so` (`-l pthread`) обязательно, но в QNX — нет; в QNX указание библиотеки `libsocket.so` (`-l socket`) обязательно, но в Linux — нет. Но даже это моё последнее утверждение может изменяться от версии к версии (gcc, не ядра!). Ищите недостающие библиотеки и определяйте их явно.

- Пути поиска разделяемых библиотек. В разных системах могут сильно различаться. Об этом очень обстоятельно рассказано раньше — определите доступные пути поиска ваших разделяемых библиотек.

- В некоторых POSIX сборка может производиться со статической библиотекой, но у вас может не быть в распоряжении статической библиотеки. Переопределите сборку на динамическую (`-B dynamic`).

Вы прошли все три этапа без сообщений об ошибках? Так я вас поздравляю: вы только-что уже собрали чужеродный проект для выполнения его в Linux!

Инструменты удалённой работы

Операционная система Linux (и вообще параллельные ей линии UNIX) и сеть TCP/IP (Internet) развивались одновременно, параллельно, и подпитывая друг друга идеями. Поэтому не удивительна та высокая степень их взаимной интегрированности, которую мы наблюдаем. В UNIX сложилось обычной практикой работа с одним хостом локальной сети, используя терминал другого сетевого хоста. Уделим некоторое внимание возможностям использования этой распространённой практики...

Сеть Linux

Прежде, чем обсуждать особенности работы с любым из сетевых протоколов, нужно убедиться, что использование этого протокола не запрещено в настройках файервола вашей системы. Соответствия протоколов их численным значениям портов UDP или TCP смотрим в уже обсуждавшемся файле `/etc/services` :

```
$ cat /etc/services
...
ftp             21/tcp
ftp             21/udp          fsp fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp                          # SSH Remote Login Protocol
telnet          23/tcp
telnet          23/udp
...
```

После этого (уточнив характеристики интересующего нас протокола) разрешаем его к использованию средствами конфигурирования файервола (утилиты `iptables` или GUI оболочки управления сетью).

Сетевая подсистема (стек протоколов TCP/IP) Linux — очень развитая подсистема, пронизывающая всю архитектуру системы Linux. Мы очень поверхностно пробежимся по сетевым инструментам Linux, но если вас это не интересует, вы можете переходить прямо к рассмотрению средств удалённой работы, начиная с обзора средств протокола `ssh`.

Сетевые интерфейсы

В отличие от всех прочих устройств в системе, которым соответствуют имена устройств в каталоге `/dev`, сетевые устройства создают сетевые интерфейсы, которые не отображаются как именованные устройства, но каждый из которых имеет набор своих характеристических параметров (MAC адрес, IP адрес, маска сети, ...). Интерфейсы могут быть физическими (отображающими реальные аппаратные сетевые устройства, например, `eth0` — адаптер Ethernet), или логическими (отражающими некоторые моделируемые понятия, например, `tap0` — тунельный интерфейс). Одному аппаратному сетевому устройству может соответствовать одновременно несколько различных сетевых интерфейсов.

Самым старым и известным инструментов диагностирования и управления сетевыми интерфейсами утилита:

```
$ ifconfig
...
cipsec0   Link encap:Ethernet  HWaddr 00:0B:FC:F8:01:8F
          inet addr:192.168.27.101  Mask:255.255.255.0
          inet6 addr: fe80::20b:fcff:fef8:18f/64 Scope:Link
          UP RUNNING NOARP  MTU:1356  Metric:1
          RX packets:4 errors:0 dropped:3 overruns:0 frame:0
          TX packets:18 errors:0 dropped:5 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:538 (538.0 b)  TX bytes:1670 (1.6 KiB)
...
wlan0     Link encap:Ethernet  HWaddr 00:13:02:69:70:9B
          inet addr:192.168.1.21  Bcast:192.168.1.255  Mask:255.255.255.0
```

```
          inet6 addr: fe80::213:2ff:fe69:709b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10863 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11768 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3274108 (3.1 MiB)  TX bytes:1727121 (1.6 MiB)
```

Здесь показаны два сетевых интерфейса: физическая беспроводная сеть Wi-Fi (`wlan0`) и виртуальный интерфейс (виртуальная частная сеть, VPN) созданный программными средствами (Cisco Systems VPN Client) от Cisco Systems (`cipsec0`), работающий через тот же физический канал (подтверждающие сказанное выше, о возможности нескольких сетевых интерфейсов над одним каналом). Эта команда имеет очень развитую функциональность, она позволяет не только диагностику, но и управление интерфейсами: запуск и останов интерфейса (операции up и down), присвоение IP адреса, маски, создание IP алиасов и многое другое. Для управления создаваемым сетевым интерфейсом (например, операции up или down), в отличие от диагностики, утилита `ifconfig` потребует прав `root`.

Гораздо менее известным (более поздним), но более развитым инструментом, является утилита `ip` (в некоторых дистрибутивах может потребоваться отдельная установка как пакета, известного под именем `iproute2`), вот результаты выполнения для той же конфигурации:

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN qlen 1000
    link/ether 00:15:60:c4:ee:02 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 00:13:02:69:70:9b brd ff:ff:ff:ff:ff:ff
4: vboxnet0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
5: pan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether ae:4c:18:a0:26:1b brd ff:ff:ff:ff:ff:ff
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
$ ip addr show dev cipsec0
6: cipsec0: <NOARP,UP,LOWER_UP> mtu 1356 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:0b:fc:f8:01:8f brd ff:ff:ff:ff:ff:ff
    inet 192.168.27.101/24 brd 192.168.27.255 scope global cipsec0
    inet6 fe80::20b:fcff:fef8:18f/64 scope link
       valid_lft forever preferred_lft forever
```

Утилита `ip` имеет очень разветвлённый синтаксис, но, к счастью, и такую же разветвлённую (древовидную) систему подсказок:

```
$ ip help
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
       ip [ -force ] -batch filename
where  OBJECT := { link | addr | addrlabel | route | rule | neigh | ntable |
                   tunnel | maddr | mroute | monitor | xfrm }
       OPTIONS := { -V[ersion] | -s[tatistics] | -d[etails] | -r[esolve] |
                    -f[amily] { inet | inet6 | ipx | dnet | link } |
                    -o[neline] | -t[imestamp] | -b[atch] [filename] }
$ ip addr help
Usage: ip addr {add|change|replace} IFADDR dev STRING [ LIFETIME ]
                                                      [ CONFFLAG-LIST]
       ip addr del IFADDR dev STRING
       ip addr {show|flush} [ dev STRING ] [ scope SCOPE-ID ]
                            [ to PREFIX ] [ FLAG-LIST ] [ label PATTERN ]
...
```

Инструменты управления и диагностики

Помимо множества общеизвестных инструментов тестирования сети, таких как `ping` или `traceroute` (не будем на них останавливаться в виду их общеизвестности), в Linux присутствует огромное число сетевых инструментов диагностики и управления, описание только их займёт не одну книгу. Ограничимся только простым беглым перечислением некоторых, наиболее известных, для того, чтобы стало понятно зачем они применяются...

Ещё одно представление сетевых интерфейсов:

```
$ netstat -i
Kernel Interface table
Iface       MTU Met    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0       1500   0        0      0      0      0        0      0      0      0 BMU
lo        16436   0     5508      0      0      0     5508      0      0      0 LRU
wlan0      1500   0   154771      0      0      0   165079      0      0      0 BMRU
```

Установленные TCP соединения:

```
$ netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        0      0 notebook.localdomain:56223  2ip.ru:http                 TIME_WAIT
tcp        0      0 notebook.localdomain:45804  178-82-198-81.dynamic:31172 ESTABLISHED
tcp        0      0 notebook.localdomain:48314  c-76-19-81-120.hsd1.ct:9701 ESTABLISHED
tcp        0      0 notebook.localdomain:56228  2ip.ru:http                 TIME_WAIT
tcp        0      0 notebook.localdomain:56220  2ip.ru:http                 TIME_WAIT
tcp        0      0 notebook.localdomain:41762  mail.ukrpost.ua:imap        ESTABLISHED
tcp        0      0 notebook.localdomain:46302  bw-in-f16.1e100.net:imaps   ESTABLISHED
tcp        0      0 notebook.localdomain:56222  2ip.ru:http                 TIME_WAIT
tcp        0      0 notebook.localdomain:ssh    192.168.1.20:57939          ESTABLISHED
tcp        0      0 notebook.localdomain:56204  2ip.ru:http                 TIME_WAIT
tcp        0      0 notebook.localdomain:48861  mail1.ks.pochta.ru:imap     ESTABLISHED
```

Таблица маршрутизации хоста (основной источник информации):

```
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     *               255.255.255.0   U     2      0        0 wlan0
default         192.168.1.1     0.0.0.0         UG    0      0        0 wlan0
```

Диагностика и управление разрешением MAC адресов в адреса IP:

```
$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.20             ether   f4:6d:04:60:78:6f   C                     wlan0
192.168.1.1              ether   94:0c:6d:a5:c1:1f   C                     wlan0
```

Примеры запросов к DNS на прямое и обратное разрешение имени:

```
$ nslookup fedora.com
Server:         192.168.1.1
Address:        192.168.1.1#53

Non-authoritative answer:
Name:   fedora.com
Address: 174.137.125.92
$ nslookup  174.137.125.92
Server:         192.168.1.1
```

```
Address:        192.168.1.1#53

Non-authoritative answer:
92.125.137.174.in-addr.arpa   name = mdnh-siteboxparking.phl.marchex.com.

Authoritative answers can be found from:
125.137.174.in-addr.arpa      nameserver = c.ns.marchex.com.
125.137.174.in-addr.arpa      nameserver = d.ns.marchex.com.
125.137.174.in-addr.arpa      nameserver = a.ns.marchex.com.
125.137.174.in-addr.arpa      nameserver = b.ns.marchex.com.
$ host fedora.com
fedora.com has address 174.137.125.92
$ host 174.137.125.92
92.125.137.174.in-addr.arpa domain name pointer mdnh-siteboxparking.phl.marchex.com.
```

Одна из самых известных и широко описанных программ наблюдения сетевого трафика:

```
$ sudo tcpdump -i wlan0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlan0, link-type EN10MB (Ethernet), capture size 65535 bytes
21:29:58.638948 IP 125-252-88-77.dc-customer.top.net.ua.irdmi > notebook.localdomain.50414: Flags
[.], seq 714179229:714180617, ack 3453197392, win 8328, options [nop,nop,TS val 466775253 ecr
11426902], length 1388
21:29:58.639420 IP 125-252-88-77.dc-customer.top.net.ua.irdmi > notebook.localdomain.50414: Flags
[P.], seq 1388:1400, ack 1, win 8328, options [nop,nop,TS val 466775253 ecr 11426902], length 12
...
```

С помощью программы tcpdump, формируя сложные условия фильтра отбора пакетов для протокола (по сетевым интерфейсам, протоколам, адресам источника и получателя...) можно локализовать проблемы и изучить практически любой сетевой обмен.

Протокол ssh (Secure Shell)

Сетевой протокол сеансового уровня, позволяющий осуществлять защищённое тунелирование TCP-соединений (например, для передачи файлов). Схож по функциональности с протоколами telnet и rlogin, но, в отличие от них, шифрует весь трафик, включая и передаваемые пароли. Протокол ssh допускает выбор различных алгоритмов шифрования.

Прежде, чем рассчитывать на использование сервиса ssh, необходимо убедиться, что на серверном хосте работает демон sshd:

```
$ ps -Af | grep sshd
root      4144     1  0 05:53 ?        00:00:00 /usr/sbin/sshd
```

Далее создаём любое в отдельных терминалах нужное нам число ssh сессий:

```
$ ssh -l olej notebook
olej@notebook's password:
Last login: Sun Mar 13 23:17:00 2011 from home
$ uname -n
notebook.localdomain
```

Этим мы фактически создаём отдельную терминальную сессию к удалённому хосту, с шифрованным трафиком между хостами.

Клиент telnet

Для службы telnet не существует как такового выделенного запускаемого сервера, telnetd запускается по сетевому запросу суперсервером inetd или xinetd поэтому для его использования нужно разобраться с конфигурациями суперсерверов, например, для xinetd в файле /etc/xinetd.d/krb5-telnet :

```
        socket_type     = stream
        user            = root
        server          = /usr/kerberos/sbin/telnetd
        disable         = no
```

- возможно, вам придётся заменить значение `disable = yes` на `no`, сам сервер находим на пути: `/usr/kerberos/sbin/telnetd`.

Примечание: Даже если, зная путь к серверу `telnetd`, попытаться запустить прямой командой запуска, из этого ничего не получится: сервер для этого не предназначен; на него есть хорошая справка:

```
$  man telnetd
TELNETD(8)                                                         TELNETD(8)
NAME
       telnetd - DARPA TELNET protocol server
...
```

- оттуда можно почерпнуть, что автономно его можно запустить, но это только в том режиме, который у них называется отладочным.

Подключение `telnet` клиента:

```
$ telnet -l olej notebook
Trying 192.168.1.9...
telnet: connect to address 192.168.1.9: Connection refused
telnet: Unable to connect to remote host: Connection refused
```

- это как раз тот (частый) случай, когда а). сервер `telnetd` не конфигурирован в суперсервере, или б). когда порт `telnet` не разрешён в файерволе. Случай нормального подключения выглядит так:

```
$ telnet home
Trying 192.168.1.7...
Connected to home.
Escape character is '^]'.
home (Linux release 2.6.18-92.el5 #1 SMP Tue Jun 10 18:49:47 EDT 2008) (13)
login: olej
Password:
Last login: Sat Mar 19 09:03:04 from notebook
[olej@home ~]$ uname -n
home
[olej@home ~]$ exit
logout
Connection closed by foreign host.
```

На хосте `home` к которому подключаемся наблюдаем:

```
$ ps -A | grep netd
 4249 ?        00:00:00 xinetd
$ ps -A | grep tcp
$ ps -A | grep telnet
```

Значение удалённого текстового терминала `telnet` постепенно утрачивается (он заменяется `ssh`), но он широко используется в среде программистов разработчиков как незаменимый сетевой тестер (клиент) для разнообразных других портов TCP :

```
$ telnet notebook 13
Trying 192.168.1.9...
Connected to notebook.localdomain (192.168.1.9).
Escape character is '^]'.
28 APR 2011 10:31:07 EEST
Connection closed by foreign host.
```

- показано подключение по порту даты-времени к хосту, в подключение к которому по стандартному протоколу `telnet`, в примере выше, было отказано.

```
$ telnet notebook echo
Trying 192.168.1.9...
Connected to notebook.localdomain (192.168.1.9).
Escape character is '^]'.
123
123
asdfgh
asdfgh
... echo server ...
... echo server ...
^C
^]
telnet> ^C Connection closed.
```

- это подключение по порту эхо-повторителя, сессию `telnet` завершаем клавишной комбинацией '`^]`'.

Клиент rlogin

Эта, одна из самых старых, сетевая служба может пригодится для связи и обмена данными с другими POSIX, но не Linux, операционными системами.

```
$ rlogin -l olej home
home: Connection refused
```

Эта служба предоставляет удалённую текстовую консоль, подобно тому, как это делают `telnet` или `ssh`. Может успешно использоваться для подключения терминала Linux к другой операционной системе, например MINIX3.

Средства ftp / tftp

Одна из самых распространённых служб. Существует великое множество как клиентов `ftp`, так и серверов, запускаемых как по запросу к суперсерверу, так и автономно; одна из широко распространённых реализаций стороннего сервера для Linux:

```
$ ps -A | grep ftp
 1800 ?        00:00:00 proftpd
```

Простейший клиент — консольный, он же лучшее средство для проверки работоспособности службы:

```
$ ftp notebook
Connected to notebook.
220 FTP Server ready.
500 AUTH не распознано
500 AUTH не распознано
KERBEROS_V4 rejected as an authentication type
Name (notebook:olej):
331 Необходим пароль для пользователя olej
Password:
230 Пользователь olej подключён
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> help
Commands may be abbreviated.  Commands are:
!               cr              mdir            proxy           send
$               delete          mget            sendport        site
account         debug           mkdir           put             size
append          dir             mls             pwd             status
ascii           disconnect      mode            quit            struct
bell            form            modtime         quote           system
binary          get             mput            recv            sunique
```

```
bye             glob            newer           reget           tenex
case            hash            nmap            rstatus         trace
ccc             help            nlist           rhelp           type
cd              idle            ntrans          rename          user
cdup            image           open            reset           umask
chmod           lcd             passive         restart         verbose
clear           ls              private         rmdir           ?
close           macdef          prompt          runique
cprotect        mdelete         protect         safe
ftp> pwd
257 "/" является текущей директорией
ftp> quit
221 До свидания.
```

Имеется много разнообразных GUI клиентов `ftp`.

Система nfs

Протокол и подсистема `nfs` (Network File System), и сопутствующий ему протокол `nis` (Network Information System) очень удобны, но в сети, состоящей только из Linux/UNIX хостов (для других систем есть реализации, но они не получили распространения). Протокол позволяет монтировать поддеревья файловой системы удалённого хоста в дерево локальной файловой системы, и далее работать с ними как со своими собственными. Сама подсистема `nfs` достаточно громоздкая:

```
$ uname -r
2.6.35.11-83.fc14.i686
$ lsmod | grep nfs
nfsd                  201440  13
lockd                  56070  1 nfsd
nfs_acl                 1951  1 nfsd
auth_rpcgss            28808  1 nfsd
exportfs                2923  1 nfsd
sunrpc                165546  17 nfsd,lockd,nfs_acl,auth_rpcgss
$ ps -Af | grep nfs
root      1421     2  0 Mar22 ?        00:00:00 [nfsd4]
root      1422     2  0 Mar22 ?        00:00:00 [nfsd4_callbacks]
root      1423     2  0 Mar22 ?        00:00:00 [nfsd]
root      1424     2  0 Mar22 ?        00:00:00 [nfsd]
root      1425     2  0 Mar22 ?        00:00:00 [nfsd]
root      1426     2  0 Mar22 ?        00:00:00 [nfsd]
root      1427     2  0 Mar22 ?        00:00:00 [nfsd]
root      1428     2  0 Mar22 ?        00:00:00 [nfsd]
root      1429     2  0 Mar22 ?        00:00:00 [nfsd]
root      1430     2  0 Mar22 ?        00:00:00 [nfsd]
```

Видно, что подсистема `nfs` продолжает активно претерпевать изменения, и в последних ядрах реализована на потоках ядра Linux ([...] в выводе утилиты `ps`). Сетевая система требует проведения кропотливой настройки, но удобства её использования того стоят. В общих чертах это выглядит так:

1. Проверяем, что наше ядро вообще скомпилировано с поддержкой `nfs` (но обратное бывает редко) :

```
$ cat /proc/filesystems | grep nfs
nodev   nfsd
```

И убеждаемя в том, что у нас запущены службы протокола RPC и сам демон `nfsd` (как показано было выше):

```
# ps -A | grep rpc
 1278 ?        00:00:00 rpcbind
 1333 ?        00:00:00 rpc.statd
 1369 ?        00:00:00 rpciod/0
 1370 ?        00:00:00 rpciod/1
```

```
 1379 ?        00:00:00 rpc.idmapd
 1738 ?        00:00:00 rpc.rquotad
 1753 ?        00:00:00 rpc.mountd
```

Кроме того, убеждаемся что порты сетевой системы не запрещены файерволом, часто для управления этими настройками используются GUI скрипты вида: `/usr/bin/system-config-firewall` . Проверки этого пункта могут и не проверяться изначально, но если экспортирование каталогов не удастся, то к ним придётся вернуться. Проверить, какие службы (из которых нас в первую очередь интересует `nfs`) RPC запущены на хосте (в данном случае хост `home`):

```
$ rpcinfo -p home
   program vers proto   port  service
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp   1008  status
    100024    1   tcp   1011  status
    100011    1   udp    740  rquotad
    100011    2   udp    740  rquotad
    100011    1   tcp    743  rquotad
    100011    2   tcp    743  rquotad
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    4   udp   2049  nfs
    100021    1   udp  32808  nlockmgr
    100021    3   udp  32808  nlockmgr
    100021    4   udp  32808  nlockmgr
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100003    4   tcp   2049  nfs
    100021    1   tcp  52742  nlockmgr
    100021    3   tcp  52742  nlockmgr
    100021    4   tcp  52742  nlockmgr
    100005    1   udp    767  mountd
    100005    1   tcp    770  mountd
    100005    2   udp    767  mountd
    100005    2   tcp    770  mountd
    100005    3   udp    767  mountd
    100005    3   tcp    770  mountd
```

2. Логика сетевой системы `nfs` состоит в том, что каждый разделяемый данным хостом каталог должен быть экспортирован в списках экспорта, описан в файле `/etc/exports`. Для каждого экспортируемого каталога заводится одна строка запись, запись достаточно сложная, описывающая список (разделяемый пробелами) хостов и прав доступа, имеющих место при доступе из этих хостов. Это могут быть групповые адреса, определяющие условия доступа из целых подсетей. Доступ с хостов, не представленных в списках доступа — не разрешён. Пример такой строки списков доступа относительно одного из каталогов:

```
/home/olej   *(rw,insecure,sync,no_root_squash) 192.168.0.0/16(rw,insecure,sync,no_root_squash)
```

Заполнять (и проверить корректность) файла экспорта позволяют GUI скрипты вида подобного: `/usr/bin/system-config-nfs` , но такие скрипты конфигурации могут конфликтовать с ручным заполнением `/etc/exports`, что нужно тщательно выверить.

После завершения определения списков экспорта мы можем его проверить, из этого, или (на видимость) из другого хоста сети, утилитой:

```
# showmount -e home
Export list for home:
/home/olej (everyone)
# showmount -e notebook
Export list for notebook:
/home/olej (everyone)
```

3. Далее, мы можем по общим правилам выполнения команды `mount` (монтирование, выполняемое от имени `root`, к существующим заранее точкам монтирования, с указанием типа монтируемой файловой системы `nfs`, ...) выполнять монтирование в свою файловую систему сколь угодно многих каталогов, экспортируемых с различных хостов сети:

```
# mount -t nfs notebook:/home/olej /mnt/home1
$ du -hs /mnt/home1
16G     /mnt/home1
```

Особенностью для `nfs` системы есть то, что тип монтируемой системы (`-t nfs`) можно опускать в виду специфического синтаксиса указания монтируемого ресурса (`notebook:/home/olej`), утилита `mount` сама распознает этот случай, поэтому предыдущее монтирование можно записать так, что полностью эквивалентно:

```
# mount notebook:/home/olej /mnt/home1
```

4. Смонтированных в разные точки монтирования каталогов `nfs` может быть достаточно много, чтобы диагностировать что и куда смонтировано, используем:

```
# mount -t nfs
home:/home/olej on /mnt/home type nfs (rw,addr=192.168.1.7)
notebook:/home/olej on /mnt/home1 type nfs (rw,addr=192.168.1.9)
```

Такого же сорта дополнительную информацию мы можем получить и утилитой `df`:

```
# df -t nfs
Файловая система     1K-блоков      Исп  Доступно  Исп% смонтирована на
home:/home/olej        7640640   6705824    540416  93% /mnt/home
notebook:/home/olej   43135744  32713472   8231168  80% /mnt/home1
```

5. Так же, как мы монтируем каталог, обратным образом, мы его можем и отмонтировать когда он не нужен:

```
# umount /mnt/home1
$ du -hs /mnt/home1
4,0K    /mnt/home1
```

6. Наконец, сетевые каталоги, которые мы часто используем, мы можем сделать постоянно смонтированными в своей файловой системе. Для этого в `/etc/fstab` пропишем строку (по строке для каждого из монтируемых каталогов):

```
# device              directory       type    options     dump    fsckorder
notebook:/home/olej   /mnt/home1      nfs     defaults    0       0
```

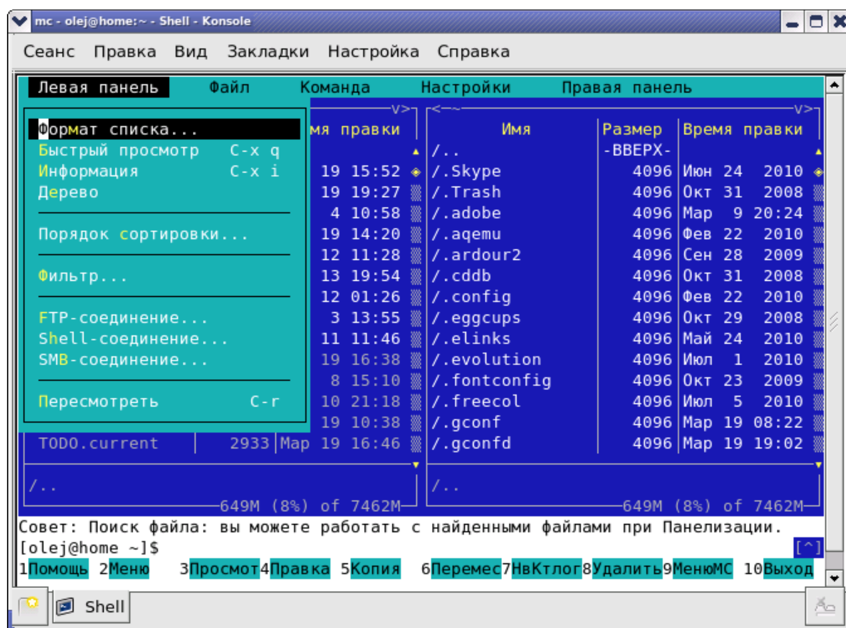
В дополнение к монтированию при загрузке, при создании такой записи монтирование и размонтирование каталога записывается командой ещё проще:

```
# mount notebook:/home/olej
```

Записи списков экспорта (`/etc/exports`) и параметры команд монтирования допускают большое множество опций и параметров, переопределяющих параметры сетевого соединения (таких, например, как реакция на временную потерю связи). Все эти параметры полно описаны в справочной системе и литературе.

Подсистема `nfs` может представить активный интерес в ходе разработок в области встраиваемого и мобильного Linux, переносимости системы на новые аппаратные платформы: в таких работах `nfs` является одним их протоколов, часто используемым для объединения целевой и инструментальной машин.

Удалённые сессии в файловом менеджере mc



Панели (левую, правую) менеджера mc можно настроить на отображение сессий удалённых подключений, причём то, что названо в меню mc (на рисунке) «Shell-соединение» - представляет собой (что совсем не очевидно) сессию ssh, а «FTP-соединение», соответственно — сессию ftp (на рисунке показана панель mc в момент выбора по F9 протокола для удалённой сессии).

При установлении таких соединений, параметры соединения (пользователь, пароль, хост) должны записываться в следующем синтаксисе:

```
olej:asdf55@192.168.1.9
olej:asdf55@home
olej@home
```

В третьем случае, если вы не хотите указывать пароль в адресной строке, пароль будет запрошен в диалоге в командной строке mc (внизу, в командной строке).

Достоинством таких соединений является не только то, что вы не только можете копировать или переносить (по F5, F6 в mc) файлы между локальным и удалённым каталогом, но и просматривать (F3) или редактировать (F4) файлы непосредственно на удалённом хосте. Этого, особенно одновременно с открытой в другом терминале сессией ssh к тому же удалённому хосту, вполне достаточно, чтобы редактировать, компилировать и собирать программные проекты на удалённом хосте. И всё это независимо от того, какая операционная система работает на том удалённом хосте: Linux, Solaris, QNX или MINIX3.

Удалённый X11

Уже упоминалось, что в Linux/UNIX графическая подсистема X11 не является составной частью операционной системы, более того — является чисто подсистемой пользовательского уровня, не использующей модули ядра системы, и работающей непосредственно с множеством типов видеоадаптеров средствами пользовательского пространства (не ядра). Система Linux вполне может быть установлена и сконфигурирована без наличия подсистемы X11 в своём составе. Ещё более удивляет многих при первом знакомстве с X11 (особенно после опыта Windows) то, что операционная система взаимодействует со своей графической подсистемой исключительно посредством сетевого протокола X, даже в совершенно локальной конфигурации оборудования, даже не имеющего сетевого адаптера.

Примечание: Именно поэтому большой неожиданностью бывает порой то, что графическая подсистема X11 может «отвалиться», и стать полностью неработоспособной, если неосторожно накрутить что-то в сетевых настройках системы.

Но если это так, то графическим приложениям (всем и любым!) совершенно одинаково работать ли, отображая информацию и осуществляя ввод с локального терминала, или терминала другого хоста локальной сети, или терминала компьютера, находящегося на другом конце света. Для этого потребуются только соответствующие настройки... В этой части мы проверим удалённый запуск любых графических приложений X11 (в качестве пробного приложения показан `xterm`, но это может быть **любое** графическое приложение Linux). Для определённости объяснений и примеров, приложение будет выполняться на хосте `notebook` (IP 192.168.1.9), а его графический вывод (ввод) производится на экран хоста `home` (IP 192.168.1.7). Оба хоста находятся в одном сегменте LAN, но это не имеет значения, и они так же будут выполняться в WAN.

Нативный протокол X

Для соединения с X-сервером используется TCP протокол, порт 6000 (нужно проверить, чтобы этот порт не был закрыт защищающими механизмами):

```
$ cat /etc/services
...
x11             6000/tcp        X               # the X Window System
...
```

Примечание: В системе X11 наблюдается, отмечаемая неоднократно, «инверсия терминов»: графический терминал за которым работает пользователь называется X-сервером, а программное приложение, с которым взаимодействует пользователь, и которое отвечает на его действия за терминалом — X-клиентом. Это происходит потому, что эта терминология отражает распределение ролей с точки зрения отрисовки графики, с позиции того, кто запрашивает графические примитивы, и кто их реализует. При таком рассмотрении всё становится на свои места...

На хосте `home` разрешаем X-серверу принимать подключения от указанного хоста :

```
$ uname -n
home
$ xhost
access control enabled, only authorized clients can connect
SI:localuser:olej
$ xhost +notebook
notebook being added to access control list
$ xhost
access control enabled, only authorized clients can connect
INET:notebook
SI:localuser:olej
```

Теперь экран этого компьютера готов для отображения удалённых приложений.

Примечание: в защищённой или изолированной LAN можно разрешить доступ по X-протоколу не для отдельного выбранного адреса, а для любого хоста LAN:

```
$ xhost +
access control disabled, clients can connect from any host
$ xhost
access control disabled, clients can connect from any host
SI:localuser:olej
```

Примечание: когда необходимость в доступе отпадёт, мы запретим его путём, обратным тому, как мы его разрешали:

```
$ xhost -notebook
notebook being removed from access control list
$ xhost
access control enabled, only authorized clients can connect
SI:localuser:olej
```

```
olej@notebook:/ <@notebook.localdomain>
[olej@notebook /]$ uname -n
notebook.localdomain
[olej@notebook /]$
```

Далее на хосте `notebook` указываем дисплей для X-протокола и запускаем приложение (для примера показан `xterm`) :

```
$ DISPLAY=192.168.1.7:0.0; export DISPLAY
$ echo $DISPLAY
192.168.1.7:0.0
$ xterm
^C
```

На рисунке, скопированном с экрана хоста `home` (X-сервера), хорошо видно, что `xterm` идентифицирует свой хост (на котором он выполняется) как `notebook`.

Другой способ указать X-клиенту (приложению) какой X-сервер (дисплей) использовать, без задействования переменной окружения:

```
$ DISPLAY=; export DISPLAY
$ echo $DISPLAY
$ xterm -display 192.168.1.7:0.0
...
```

Примечание: в некоторых инсталляциях Linux сервер X11 может запускаться так, что ему запрещено прослушивание протокола TCP/IP (обмен только через нативный домен UNIX). Для того, чтобы проверить это, смотрим строку запуска X-сервера:

```
$ ps ahx | grep Xorg
 4476 tty7     Ss+   34:26 /usr/bin/Xorg :0 -br -audit 0 -auth /var/gdm/:0.Xauth vt7
 6388 pts/4    S+     0:00 grep Xorg
```

Строка запуска без протокола TCP/IP будет содержать нечто подобное следующему (явно указывается «не прослушивать», по умолчанию соединение TCP прослушивается):

```
... /usr/bin/Xorg :0 -br -audit 0 -auth /var/gdm/:0.Xauth -nolisten tcp vt7
```

Если это так, и вам нужен удалённый X11, X-сервер нужно переконфигурировать и перезапустить... , сделать это можно разнообразными способами, например, так:

- Воспользоваться менджером:

```
# gdmsetup
```

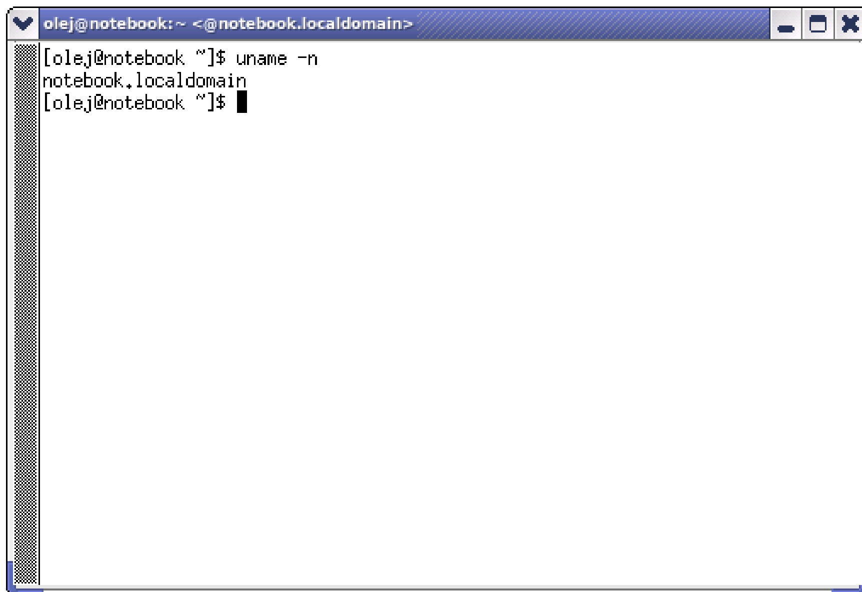
- Установить в менеджере разрешение TCP доступа, и перезапустить X систему:

```
# gdm-restart
```

- Не стоит здесь пугаться, что рабочая сессия X закроется, и будет запущена новая, начиная с начального `login`.

Графическая сессия ssh

Альтернативный способ выполнить X-приложение на хосте `notebook` находясь на хосте `home` (аналогично предыдущему случаю) — это тунелировать сообщения X-протокола внутрь протокола `ssh` (современные реализации `ssh` умеют это делать: пакеты протокола X инкапсулируются в сообщения `ssh`).



Для этого на хосте `home` (на той стороне, где мы хотим наблюдать выполнение, где будет находиться наш графический терминал - X-сервер) выполняем соответствующую `ssh` команду:

```
$ uname -n
home
$ ssh -nfX -l olej notebook xterm
olej@notebook's password:
$
```

Формат команды:

```
$ ssh -nfX -l<user> <host> <command>
```

Принципиальным здесь есть то, что (в отличие от создания текстовой сессии `ssh`), что вы обязательно должны указать в команде приложение, которое запускается. Меня часто спрашивают: «А как выбрать приложение (одно или несколько) уже после установления графической сессии `ssh`?». Да очень просто: запустите удалённый терминал `xterm`, например, как было показано выше, а уже в нём вы можете реализовывать любые свои капризы: становиться `root`-ом, устанавливать недостающие графические программы, если их там не окажется, и запускать эти программы (с отображением их на свой **удалённый** терминал), как показано на рисунке:

```
root@nvidia:~ (на nvidia.localdomain)

[olej@nvidia home]$ xclock
bash: xclock: command not found...
Установить пакет 'xorg-x11-apps' предоставляющий команду 'xclock'? [N/y]
 * Выполнение..
 * Разрешение зависимостей..
 * Ожидание авторизации.. Действие не удалось: not-authorized, Failed to obtain
authentication.

[olej@nvidia home]$ su -
Пароль:
[root@nvidia ~]# xclock
bash: xclock: command not found...
Установить пакет 'xorg-x11-apps' предоставляющий команду 'xclock'? [N/y]
 * Выполнение..

 * Разрешение зависимостей..
 * Ожидание авторизации..
 * Разрешение зависимостей..
 * Загрузка пакетов..
 * Проверка изменений..
 * Установка пакетов..
 * Сканирование приложений..
Warning: Missing charsets in String to FontSet conversion
Warning: Unable to load any usable fontset
^C
[root@nvidia ~]#
```

Сети Windows

Всё, что касается поддержки сетевых средств Windows (разделение файлов и использование принтеров) развивается из проекта, называемого Samba, поддержки протокола SMB (с 1996г. протокол переименован в CIFS).

Для проверки и демонстрации работы нам необходимо определить в своей системе IP адрес и имя хоста Windows, например так (следите внимательно далее за этим именем в командах):

```
$ cat /etc/hosts
...
192.168.1.3      rtp rtp.localdomain
$ ping rtp
PING rtp (192.168.1.3) 56(84) bytes of data.
64 bytes from rtp (192.168.1.3): icmp_seq=1 ttl=128 time=1.03 ms
64 bytes from rtp (192.168.1.3): icmp_seq=2 ttl=128 time=0.459 ms
...
```

Есть несколько альтернативных способов сетевого доступа к разделяемым ресурсам систем Windows.

Пакет Samba

FTP подобный программа-клиент, вот он перечисляет разделяемые Windows-ресурсы, каталоги на этих ресурсах, и обменивается файлами:

```
$ smbclient -L rtp -U Olej -N
        Sharename       Type      Comment
        ---------       ----      -------
        CDROM           Disk
        D               Disk
        C               Disk
        ADMIN$          Disk
        MY DOCUMENTS    Disk
...
$ smbclient //rtp/D -U olej -N
smb: \> dir
  Program Files                       D        0  Fri Nov 19 20:20:56 2004
```

```
  RECYCLED                          DHS        0  Sat Nov 20 12:54:58 2004
...
  FR6.install.hist                    A      218  Fri Oct 29 01:59:52 2010
...
                47975 blocks of size 65536. 5953 blocks available
smb: \> get FR6.install.hist
getting file \FR6.install.hist of size 218 as FR6.install.hist (71,0 KiloBytes/sec) (average 71,0 KiloBytes/sec)
smb: \> quit
$ ls FR6.*
FR6.install.hist
```

Печать с Samba

Если в Linux установлена подсистема печати (BSD) `lpr/lpd`, то печать на хостах Windows обеспечивается утилитой (скриптом) в составе Samba — `smbprint`:

```
$ which smbprint
/usr/bin/smbprint
```

Примерно с 2000-2001 годов на смену подсистемы печати BSD стала приходить подсистема печати CUPS (Common Unix Printing System) на основе демона управления буфером печати `cupsd`:

```
$ ps -A | grep cupsd
 1389 ?        00:00:00 cupsd
```

Теперь инструменты Samba могут отправлять задания по каналу прямо демону управления буфером печати `cupsd`. Для этого нужно конфигурировать разделяемые принтера Windows в Linux непосредственно с помощью инструментальных средств самой системы CUPS. Для тех случаев, когда это, в силу каких-либо условий, не подходит, существует средство консольного указания выполнения задания печати:

```
$ which smbspool
/usr/bin/smbspool
```

Эта утилита позволяет самые разнообразные комбинации наборов параметров в командной строке:

```
$ smbspool --help
Usage: smbspool [DEVICE_URI] job-id user title copies options [file]
       The DEVICE_URI environment variable can also contain the
       destination printer:
          smb://[username:password@][workgroup/]server[:port]/printer
```

Серверная часть Samba

Это тот случай, когда хост Linux должен быть использован в качестве **серверного** хоста для клиентов Windows. Для этого в Linux запускаются два демона `nmbd` (демон разрешения имён NetBIOS) и `smbd` (собственно сервер):

```
$ ps -A | grep mbd
22812 ?        00:00:00 smbd
22827 ?        00:00:49 nmbd
```

Настройки сервера Samba записаны в файле `smb.conf`, вы можете редактировать эти настройки, после того, как настройки отредактированы, корректность их проверяется утилитой:

```
$ which testparm
/usr/bin/testparm
$ testparm
Load smb config files from /etc/samba/smb.conf
rlimit_max: rlimit_max (1024) below minimum Windows limit (16384)
Processing section "[homes]"
Processing section "[printers]"
...
```

И далее анализируются все секции конфигурационного файла (кстати, `testparm` позволяет и определить местоположение `smb.conf` в вашем дистрибутиве, как показано на примере выше). Полную информацию по настройкам, требуемым в `smb.conf`, достаточную для настройки любого, самого замысловатого сервера, получаем:

```
$ man 5 smb.conf
SMB.CONF(5)              File Formats and Conventions             SMB.CONF(5)
NAME
       smb.conf - The configuration file for the Samba suite
...
```

Запуск серверной подсистемы Samba может производиться не только непосредственно (например, из скрипта `/etc/rc.local`), но и суперсервером `inetd/xinetd`, что может быть важно в малых конфигурациях.

Файловые системы smbfs и cifsfs

Компоненты Linux, приобретённые в несколько последних лет: доступ к разделяемым ресурсам Windows может осуществляться через файловую систему SMB/CIFS, таких реализаций существует независимо две (SMB и CIFS), но они могут и не быть собраны по умолчанию в составе ядра Linux. Выясняем (в каталоге `/boot`) с поддержкой каких из этих систем собрано текущее ядро - если нет никакой поддержки, может оказаться необходимым пересобрать ядро:

```
$ cd /boot
$ uname -r
2.6.32.9-70.fc12.i686.PAE
$ ls *`uname -r`
config-2.6.32.9-70.fc12.i686.PAE  System.map-2.6.32.9-70.fc12.i686.PAE  vmlinuz-2.6.32.9-
70.fc12.i686.PAE
$ grep CONFIG_SMB_FS config-2.6.32.9-70.fc12.i686.PAE
# CONFIG_SMB_FS is not set
$ grep CONFIG_CIFS config-2.6.32.9-70.fc12.i686.PAE
CONFIG_CIFS=m
CONFIG_CIFS_STATS=y
# CONFIG_CIFS_STATS2 is not set
CONFIG_CIFS_WEAK_PW_HASH=y
CONFIG_CIFS_UPCALL=y
CONFIG_CIFS_XATTR=y
CONFIG_CIFS_POSIX=y
# CONFIG_CIFS_DEBUG2 is not set
CONFIG_CIFS_DFS_UPCALL=y
CONFIG_CIFS_EXPERIMENTAL=y
```

- как и предупреждалось: в этом нашем дистрибутиве по умолчанию включена `cifs`, но не включена `smbfs`. Но могут быть всякие разнообразные комбинации...

Если какая-то из файловых систем (`smbfs` это более старая реализация, `cifs` отличается, главным образом, поддержкой кодировки UNICODE в именах) присутствует в ядре, то вы можете непосредственно монтировать Windows разделяемые директории в локальную файловую систему Linux:

```
$ cd ~
$ mkdir rtpdir
$ sudo mount -t smbfs //rtp/D ~/rtpdir -o username=olej,uid=olej,gid=users
Password:
...
$ sudo mount -t cifs //rtp/D ~/rtpdir -o user=olej,uid=olej,gid=users
Password:
...
```

Примечание: В примере показано монтирование, начиная с создания каталога (точки монтирования в домашнем каталоге - `~/rtpdir`), чтобы напомнить, что монтировать (в Linux) можно а).только к существующим точкам монтирования, б).в любое место файловой системы. При записи подобных команд монтирования, самая часто повторяемая ошибка: в записи списка значений **опции** `-o` - недопустимы пробелы (и любые другие разделители), всё что отделял бы такой разделитель должно уже считаться новым **параметром** команды.

Детальную информацию по опциям монтирования (все опции списком, разделённые запятой, в значении ключа `-o`) можно получить по запросу вида (для `mount.smbfs` аналогично):

```
$ man mount.cifs
MOUNT.CIFS(8)             System Administration tools            MOUNT.CIFS(8)
NAME
       mount.cifs - mount using the Common Internet File System (CIFS)
...
```

Библиотеки API POSIX

Наконец, мы добрались до последней части нашего обзора, которая и есть цель всей затеи: обзор отличительных особенностей программирования под операционной системой Linux. Основной объём доступного программисту API на языке C (базовый уровень API Linux) стандартизован несколькими стандартами POSIX, хотя библиотеки Linux предоставляют и некоторые специфические возможности, выходящие за пределы стандарта (например, всё, что касается `sysfs`, которой просто нет в POSIX/UNIX).

Доступный API POSIX делает программное обеспечение переносимым между множеством UNIX-подобных ОС. Наиболее полное и профессиональное описание основных вызовов POSIX более чем в 1000 страниц можно получить в [2]. Уже из этого понятно, что создать более или менее полный обзор механизмов POSIX в беглом обзоре невозможно... , да и не нужно — на то уже есть обстоятельные руководства. С другой стороны, большая часть механизмов POSIX уже знакома большинству программистов из других систем... только они ещё не знают, что это POSIX, и называют это словами: «стандартная C библиотека» (идущая, например, ещё от Borland C от MS-DOS). Поэтому большая часть API POSIX — известна и понятна. Но есть некоторые ключевые понятия этого API, свойственные только UNIX системам с их традициями. Вот на нескольких таких принципиально новых сторонах POSIX API, которые и вызывают основную трудность восприятия, мы и сконцентрируемся дальше, пусть это и будет выглядеть фрагментарно...

Сводный перечень по разделам API

Один только полный перечень вызовов POSIX API чрезвычайно велик. Ниже сделана попытка раскладки API по группам. В этом перечне имена функций, которые являются принципиальной основой для UNIX, или те которые могут быть совершенно новыми для программиста Windows — **выделены жирным шрифтом**. Только этот ограниченный круг API будет подробно рассмотрен далее, этого достаточно, чтобы сориентироваться и со всем окружающим их множеством API. Основными группами вызовов [2] POSIX API можно считать:

1. Файловый ввод вывод. Дескрипторы файлов. Вызовы: `open, create, close, lseek, read, write, dup, dup2, fcntl, ioctl`.
2. Файлы и каталоги. Вызовы: `stat, fstat, access, chmod, fchmod, chown, link, unlink, symlink, mkdir, opendir, readdir`.
3. Стандартная библиотека ввода-вывода. Потоки и объекты FILE, буферизация. Вызовы: `setbuf, fopen, fclose, getc, getchar, gets, putc, putchar, puts, fread, fwrite, fseek, sprintf, printf, scanf`.
4. Окружение процесса. Запуск процессов. Терминальная система, управляющий терминал, группы процессов. Демоны. Вызовы: `uname, gethostname, time, nice, gettimeofday,` **`getopt`**`, getopt_long`.
5. Управление процессами. Основные вызовы: **`fork`**`, exit,` **`wait`**`, system, popen, pclose, exec`.
6. Терминальный ввод/вывод. Канонический и неканонический режим. Вызовы и структуры: `stty, termcap, terminfo`. Псевдотерминалы.
7. Сигналы. Вызовы: **`signal`**, **`alarm`**, `kill, raise,` **`pause`**. Ненадёжная и надёжная модель обработки. Наборы **`sigset_t`**, вызовы: **`sig*set`**, `sigprocmask, sigpending, sigaction, sigsetjmp, siglongjmp`. Сигналы реального времени.
8. Потоки `pthread_t`. Сигналы и потоки.
9. Расширенные операции ввода-вывода. Неблокирующий ввод-вывод. Асинхронные операции. Основные вызовы: `select, poll, readv, writev`.
10. Межпроцессное взаимодействие. Каналы, очереди сообщений, разделяемая память.
11. Синхронизации. Семафоры, мьютексы, барьеры, условные переменные.

Окружение процесса

Обработка опций командной строки

Стандартные утилиты Linux используют, в основном, единый формат опций указания (ключей) и параметров командной строки, который обеспечивается функцией `getopt()`, как показано в примере (архив `hello-prog.tgz`):

mgetopt.c :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main( int argc, char *argv[] ) {
   char sopt[] = "d:t:v";
   int c, dev = 0, tim = 0, debug_level = 0;
   while( -1 != ( c = getopt( argc, argv, sopt ) ) )
      switch( c ) {
         case 'd':
            dev = atoi( optarg );
            break;
         case 't':
            tim = atoi( optarg );
            break;
         case 'v':.
            debug_level++;
            break;
         default :
            fprintf( stdout, "option must be: %s\n", sopt );
      }
   printf( "options value was:" );
   printf( "\td:%d\tt:%d\tv:%d\n", dev, tim, debug_level );
   printf( "parameters was:" );
   for( c = optind; c < argc; c++ ) printf( "\t<%s>", argv[ c ] );
   printf( "\n" );
   return 0;
};
```

Вот как выполняется этот пример:

```
$ ./mgetopt  -t 3 -d2
options value was:    d:2    t:3    v:0
parameters was:
$ ./mgetopt  -s
./mgetopt: invalid option -- 's'
option must be: d:t:v
options value was:    d:0    t:0    v:0
parameters was:
```

И вот как он разделяет опции (ключи) от параметров, заданных в командной строке, даже если они указаны вперемешку:

```
$ ./mgetopt -d 1 arg1 -t 2 arg2 -vvv
options value was:    d:1    t:2    v:3
parameters was:       <arg1> <arg2>
```

Хорошим стилем было бы, если **все** ваши консольные программы следовали подобным правилам взаимодействия с пользователем.

Параллельные процессы

Для всех UNIX/POSIX операционных систем классическим способом создания параллельного процесса в системе является вызов `fork()` (относящиеся к делу определения — в `<unistd.h>`). Простейший пример способа создания в UNIX параллельных процессов может выглядеть так (все примеры этого раздела в архиве `fork.tgz`, в этом разделе все иллюстрируемые вызовы API будут отмечены в коде жирным шрифтом) :

p2.c :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include "libdiag.h"
int main( int argc, char *argv[] ) {
   long cali = calibr( 1000 );
   uint64_t t = rdtsc();
   pid_t pid = fork();        // процесс разветвился
   t = rdtsc() - t;
   t -= cali;
   if( pid == -1 ) perror( "fork" ), exit( EXIT_FAILURE );
   if( pid == 0 ) {
      printf( "child with PID=%d finished, start delayed %lu cycles\n", getpid(), t );
      exit( EXIT_SUCCESS );
   }
   if( pid > 0 ) {
      int status;
      wait( &status );
      printf( "parent with PID=%d finished, start delayed %lu cycles\n", getpid(), t );
      exit( EXIT_SUCCESS );
   };
};
```

Показательно выполнение такого простейшего примера:

```
$ ./p2
child with PID=19044 finished, start delayed 855235 cycles
parent with PID=19041 finished, start delayed 109755 cycles
$ ./p2
child with PID=30908 finished, start delayed 166435 cycles
parent with PID=30904 finished, start delayed 106025 cycles
```

Здесь фиксируются и выводятся временные засечки старта каждой из ветвей разветвлённого процесса (относительно момента, предшествующего вызову `fork()`), эти времена могут быть совершенно разными, и, самое главное, здесь нельзя утверждать кто раньше (родительский или дочерний процесс) начнёт выполняться, или даже оба они продолжаться на разных процессорах SMP. В подтверждение сказанного, рассмотрим результаты на однопроцессорном компьютере (предыдущий показанный результат получен на 2-х ядерном SMP), результаты здесь той же программы совершенно противоположны (дочерний процесс активируется значительно быстрее родительского, порядок временных величин в единицах процессорных циклов примерно сохраняется):

```
$ ./p2
child with PID=6172 finished, start delayed 253986 cycles
parent with PID=6171 finished, start delayed 964611 cycles
$ ./p2
child with PID=6174 finished, start delayed 259164 cycles
parent with PID=6173 finished, start delayed 940884 cycles
```

А вот для сравнения тот же тест :

```
$ ./p2
child with PID=26466 finished, start delayed 232627 cycles
parent with PID=26465 finished, start delayed 183480 cycles
$ ./p2
child with PID=26468 finished, start delayed 234885 cycles
parent with PID=26467 finished, start delayed 184555 cycles
```

- выполнение на 4-х ядерном процессоре, частотой в разы превосходящей выше показанный случай:

```
$ cat /proc/cpuinfo
...
processor       : 3
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Quad  CPU   Q8200  @ 2.33GHz
stepping        : 7
cpu MHz         : 1998.000
...
```

Общая картина сохраняется: порядок временных задержек сохраняется, но порядок активации параллельных процессов может быть произвольным!

Ещё один образец использования ветвления процессов это следующий пример, тестирующий запуск в системе как можно большего числа идентичных процессов:

p4.c :

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main( int argc, char *argv[] ) {
   unsigned long n = 1;
   pid_t pid;
   while( ( pid = fork() ) >= 0 ) {
      if( pid < 0 ) break;
      n++;
      if( pid > 0 ) {
         waitpid( pid, NULL, 0 );
         exit( EXIT_SUCCESS );
      };
   };
   printf( "exit with processes number: %lu\n", n );
   if( pid < 0 ) perror( NULL );
   return 0;
};
```

Вот как происходит запуск такого теста на 2-х процессорном компьютере:

```
$ time ./p4
exit with processes number: 913
Resource temporarily unavailable
real    0m0.199s
user    0m0.013s
sys     0m0.161s
$ uname -r
2.6.32.9-70.fc12.i686.PAE
```

Система была в состоянии запустить одновременно 913 процессов в дополнение к существующим в системе:

```
$ ps -A | wc -l
208
```

Но вот выполнение того же теста на 1-но процессорном компьютере (квази-параллельность!), частотой процессора всего в 3 раза ниже, и объёмом RAM меньше в 4 раза:

```
$ time ./p4
exit with processes number: 4028
Resource temporarily unavailable
real    2m59.903s
user    0m0.325s
```

```
sys     0m35.891s
$ uname -r
2.6.18-92.el5
```

Эта система оказалась в состоянии запустить одновременно 4084 дополнительных процесса, но это потребовало от неё затрат времени в сотни раз больше чем в предыдущем случае, при этом всё это время система была загружена близко к 100% и с большим трудом откликалась на команды с терминала, и это при том, что в ней стационарно сконфигурировано намного меньше выполняющихся процессов:

```
$ ps -A | wc -l
109
```

Во время этого длительного выполнения можно «подсмотреть» состояние таблицы процессов в системе:

```
$ ps -A
...
 7012 pts/1    00:00:00 p4
 7013 pts/1    00:00:00 p4
 7014 pts/1    00:00:00 p4
 7015 pts/1    00:00:00 p4
 7016 pts/1    00:00:00 p4
 7017 pts/1    00:00:00 p4
...
```

На этих механизмах, совместно с отображением созданных адресных пространств на исполнимые файлы, базируются все базовые механизмы выполнения заданий UNIX/POSIX/Linux.

Время клонирования

Интересно проследить скорость (измеряем в периодах частоты процессора) создания нового экземпляра процесса (позже сравнить её со скоростью создания потока):

p2-1.c :

```
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include <unistd.h>
#include <sys/wait.h>
#include "libdiag.h"

static uint64_t tim;
//#define data_size 1 // размер области данных в пространстве процесса: 1, 10, ... MB
#define data_size 10
#define KB        1024
#define data_byte KB*KB*data_size
static struct mbyte {
#pragma pack( 1 )
   uint8_t array[ data_byte ];
#pragma pack( 4 )
} data;
int main( int argc, char *argv[] ) {
   tim = rdtsc();
   pid_t pid = fork();
   if( pid == -1 ) perror( "fork" ), exit( EXIT_FAILURE );
   if( pid == 0 ) {
      tim = rdtsc() - tim;
      printf( "process create time : %llu\n", tim );
      if( argc > 1 ) {
         long i;
         tim = rdtsc();
         for( i = 0; i < data_byte; i += KB * 4 )
            data.array[ i ] = 0;
         tim = rdtsc() - tim;
```

```
         printf( "process write time : %llu\n", tim );
      }
      exit( EXIT_SUCCESS );
   }
   if( pid > 0 ) {
      int status;
      wait( &status );
   };
   exit( EXIT_SUCCESS );
};
```

Выполнение программы (разброс значений будет очень значителен, из-за загрузки системы и из-за кеширования областей памяти, повторяем выполнение по несколько раз):

```
$ ./p2-1
process create time : 348140
$ ./p2-1
process create time : 326090
$ ./p2-1
process create time : 216020
$ ./p2-1
process create time : 327290
```

Позже мы увидим, что время создания клона процесса практически не отличается от времени создания нового потока в процессе.

А теперь выполнение той же программы, но с модификацией страниц памяти, когда значительная область данных процесса прописывается значением (только 1-й байт каждой 4KB страницы):

- размер области данных 1 MB:

```
$ ./p2-1 w
process create time : 490670
process write time : 1877010
$ ./p2-1 w
process create time : 320200
process write time : 3956830
$ ./p2-1 w
process create time : 1558240
process write time : 2294780
$ ./p2-1 w
process create time : 291210
process write time : 2468000
```

- время записи 250 байт потребовало времени на порядок больше, чем запуск процесса — это иллюстрация работа механизма COW (copy on write).

- размер области данных 10 MB (требуется перекомпиляция задачи):

```
$ ./p2-1 w
process create time : 426220
process write time : 26742080
$ ./p2-1 w
process create time : 166930
process write time : 18489920
$ ./p2-1 w
process create time : 479890
process write time : 31890280
```

- возросло на порядок число переразмещаемых (посредством MMU) страниц адресного пространства процесса — возросло на порядок время записи.

Загрузка нового экземпляра (процесса)

Вызов `fork()` создаёт новое адресное пространство процесса (то, что до выполнения записи пространства 2-х процессов могут перекрываться в силу работы механизма «копирование при записи» - принципиально не меняет картину). Только после этого, если это необходимо, это вновь созданное адресное пространство может быть отображено на исполнимый файл (что можно толковать как загрузка новой задачи в адресное пространство). Выполняется это действие целым семейством библиотечных вызовов :

```
$ man 3 exec
EXEC(3)                    Linux Programmer's Manual                   EXEC(3)
NAME
       execl, execlp, execle, execv, execvp - execute a file

SYNOPSIS
       #include <unistd.h>
       extern char **environ;
       int execl(const char *path, const char *arg, ...);
       int execlp(const char *file, const char *arg, ...);
       int execle(const char *path, const char *arg,
                  ..., char * const envp[]);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
...
```

Но все они являются обёртками для единого системного вызова :

```
$ man 2 execve
EXECVE(2)              Руководство программиста Linux              EXECVE(2)
ИМЯ
       execve - выполнить программу
ОБЗОР
       #include <unistd.h>
       int execve(const char *filename, char *const argv [], char *const envp[]);
...
```

Но кроме целого семейства функций `exec*()`, **после** `fork()` загружающие новые процессы, предоставляются ещё упрощённые механизмы запуска новых процессов через новый экземпляр командного интерпретатора: `system()`, `popen()`, ... Ниже различные способы-механизмы рассматриваются на сравнительных примерах. Для того, чтобы лучше оценить мощь механизмов POSIX, в примерах будет использоваться не перенаправление символьной информации в потоках (что достаточно привычно, например, из использования конвейеров консольных команд), а потоков аудиоинформации и использование дочерних процессов из пакетов процесса `sox, ogg, speex` (что достаточно необычно).

Примечание: Проверьте прежде наличие в вашей системе этих установленных пакетов, это хотя все и широко распространённые пакеты, но они не является составной частью дистрибутива, и может потребовать дополнительной установки с помощью пакетного менеджера `yum`:

```
$ sox
sox: SoX v14.2.0
...
$ sudo yum install ogg*
...
$ ls /usr/bin/ogg*
/usr/bin/ogg123      /usr/bin/oggCut   /usr/bin/oggenc   /usr/bin/oggLength  /usr/bin/oggSlideshow
/usr/bin/oggCat      /usr/bin/oggdec   /usr/bin/ogginfo  /usr/bin/oggResize  /usr/bin/oggSplit
/usr/bin/oggconvert  /usr/bin/oggDump  /usr/bin/oggJoin  /usr/bin/oggScroll  /usr/bin/oggThumb
$ sudo yum install speex*
...
$ speexdec
Usage: speexdec [options] input_file.spx [output_file]
Decodes a Speex file and produce a WAV file or raw file
...
```

В архив примеров (`fork.tgz`) включены два файла образцов звуков — фрагменты женской и мужской речи (заимствованные из проекта `speex`):

```
$ ls *.wav
female.wav  male.wav
```

Проверить их звучание, и работоспособность аудиопакетов, можно утилитой play из состава пакета sox:

```
$ play -q male.wav
```

Теперь мы готовы вернуться к сравнительным примерам запуска дочерних процессов трансформации и воспроизведения аудиопотоков. Первый пример простейшим образом запускает вызовом `system()` программу `sox` в качестве дочернего процесса, для воспроизведения списка файлов, заданных в качестве параметров строки, с возможностью изменения темпо-ритма воспроизведения без искажения тембра:

s5.c :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main( int argc, char *argv[] ) {
   double stret = 1.0;
   int debug_level = 0;
   int c;
   while( -1 != ( c = getopt( argc, argv, "hvs:" ) ) )
      switch( c ) {
         case 's':
            if( 0.0 != atof( optarg ) ) stret = atof( optarg );
            break;
         case 'v': debug_level++; break;
         case 'h':
         default :
            fprintf( stdout,
                     "Опции:\n"
                     " -s - вещественный коэффициент темпо-коррекции\n"
                     " -v - увеличить уровень детализации отладочного вывода\n"
                     " -h - вот этот текст подсказки\n" );
            exit( 'h' == c ? EXIT_SUCCESS : EXIT_FAILURE );
      }
   if( optind == argc )
      fprintf( stdout, "должен быть указан хотя бы один звуковой файл\n" ),
      exit( EXIT_FAILURE );
   char stretch[ 80 ] = "";
   if( 1.0 != stret ) sprintf( stretch, " stretch %f", stret );
   else sprintf( stretch, "" );
   const char *outcmd = "sox%s -twav %s -t alsa default %s";
   int i;
   for( i = optind; i < argc; i++ ) {
      char cmd[ 120 ] = "";
      sprintf( cmd, outcmd,
               0 == debug_level ? " -q" : debug_level > 1 ? " -V" : "",
               argv[ i ],
               stretch );
      if( debug_level > 1 ) fprintf( stdout, "%s\n", cmd );
      system( cmd );
   }
   return EXIT_SUCCESS;
};
```

И выполнение этого примера:

```
$ ./s5
должен быть указан хотя бы один звуковой файл
```

```
$ ./s5 -h
Опции:
 -s - вещественный коэффициент темпо-коррекции
 -v - увеличить уровень детализации отладочного вывода
 -h - вот этот текст подсказки
$ ./s5 male.wav female.wav
$ ./s5 male.wav female.wav -s 0.7
$ ps -Af | tail -n10
...
olej     10034  7176  0 14:07 pts/10   00:00:00 ./s5 male.wav female.wav -s 2
olej     10035 10034  0 14:07 pts/10   00:00:00 sox -q -twav male.wav -t alsa default stretch
2.000000
...
```

Этот пример я представляю ещё и для того, чтобы остановить внимание на том факте, что и простейшего вызова `system()` порой достаточно для построения достаточно сложных конструкций, и что не нужно бывает для иных задач привлечения механизмов избыточной мощности, о которых пойдёт речь далее.

Следующий пример использует для создания входного и выходного потоков вызовы `popen()`: программа запускает посредством `popen()` два дочерних процесса-фильтра (теперь у нас в итоге 3 работающих процесса): входной процесс трансформирует несколько предусмотренных входных форматов (RAW, WAV, Vorbis, Speex) в единый «сырой» поток отсчётов RAW, головная программа считывает этот поток поблочно (размер блока можно менять), и передаёт эти блоки в темпе считывания выходному дочернему процессу, который, используя `sox`, воспроизводит этот поток (возможно делая для него темпо-коррекцию). Понятно, что теперь каждый отсчёт аудио потока последовательно протекает через цикл головного процесса, и в этой точке в коде процесса к потоку могут быть применены любые дополнительные алгоритмы цифровой обработки сигнала. Но прежде, чем испытывать программу, мы должны заготовить для него входной тестовый файл, в качестве которого создадим сжатый Speex файл:

```
$ speexenc male.wav male.spx
Encoding 8000 Hz audio using narrowband mode (mono)
$ ls -l male.*
-rw-rw-r-- 1 olej olej 11989 Май 12 13:47 male.spx
-rw-r--r-- 1 olej olej 96044 Авг 21  2008 male.wav
```

- при умалчиваемых параметрах сжатия программы `speexenc` размер файла ужался почти в 10 раз без потери качества, варьируя параметрами `speexenc` можно это сжатие сделать ещё больше.

Теперь собственно сам пример (пример великоват, но он стоит того, чтобы с ним поэкспериментировать):

o5.c :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
static int debug_level = 0;
static char stretch[ 80 ] = "";
u_long buflen = 1024;
u_char *buf;
// конвейер, которым мы читаем RAW файлы (PCM 16-бит), пример :
// $ cat male.raw | sox -u -s -b16 -r8000 -traw - -t alsa default stretch 0.9
// конвейер, которым мы читаем WAV файлы (или OGG Vorbis), пример:
// $ sox -V male.wav -traw -u -sw - | sox -u -s -b16 -r8000 -traw - -t alsa default stretch 0.9
// конвейер, которым мы читаем OGG SPEEX файлы, пример:
// $ speexdec -V male.spx - | sox -u -s -b16 -r8000 -traw - -t alsa default stretch 0.9
void play_file( char *filename ) {
   struct stat sbuf;
   if( stat( filename, &sbuf ) < 0 ) {
      fprintf( stdout, "неверное имя файла: %s\n", filename );
```

```
      return;
   }
   // форматы файла различаются по имени, но должны бы ещё и по magic
   // содержимому с начала файла: "RIFF..." - для *.wav, "Ogg ... vorbis" - для
   // *.ogg, "Ogg ... Speex" — для *.spx, или отсутствие magic признаков
   // для *.pcm, *.raw
   const char *ftype[] = { ".raw", ".pcm", ".wav", ".ogg", ".spx" };
   int stype = sizeof( ftype ) / sizeof( ftype[ 0 ] ), i;
   for( i = 0; i < stype; i++ ) {
      char *ext = strstr( filename, ftype[ i ] );
      if( NULL == ext ) continue;
      if( strlen( ext ) == strlen( ftype[ i ] ) ) break;
   }
   if( i == stype ) {
      fprintf( stdout, "неизвестный формат аудио файла: %s\n", filename );
      return;
   };
   char cmd[ 120 ];
   const char *inpcmd[] = {
      "cat %s",
      "sox%s %s -traw -u -s -",
      "speexdec%s %s -"
   };
   const int findex[] = { 0, 0, 1, 1, 2 };
   const char* cmdfmt = inpcmd[ findex[ i ] ];
   if( 0 == findex[ i ] )
      sprintf( cmd, cmdfmt, filename );
   else if( 1 == findex[ i ] )
      sprintf( cmd, cmdfmt,
               0 == debug_level ? " -q" : debug_level > 1 ? " -V" : "",
               filename, stretch );
   else
      sprintf( cmd, cmdfmt, debug_level > 1 ? " -V" : "", filename );
   if( debug_level > 1 ) fprintf( stdout, "%s\n", cmd );
   FILE *fsox = popen( cmd, "r" );
   const char *outcmd = "sox%s -u -s -b16 -r8000 -traw - -t alsa default %s";
   sprintf( cmd, cmdfmt = outcmd,
            0 == debug_level ? " -q" : debug_level > 1 ? " -V" : "",
            stretch );
   if( debug_level > 1 ) fprintf( stdout, "%s\n", cmd );
   FILE *fplay = popen( cmd, "w" );
   int in, on, s = 0;
   while( in = fread( buf, 1, buflen, fsox ) ) {
      if( debug_level ) fprintf( stdout, "read : %d - ", in ), fflush( stdout );
      on = fwrite( buf, 1, in, fplay );
      if( debug_level ) fprintf( stdout, "write : %d\n", on ), fflush( stdout );
      s += on;
   }
   if( debug_level ) fprintf( stdout, "воспроизведено: %d байт\n", s );
}
int main( int argc, char *argv[] ) {
   int c;
   double stret = 1.0;
   while( -1 != ( c = getopt( argc, argv, "vs:b:" ) ) )
      switch( c ) {
         case 's':
            if( 0.0 != atof( optarg ) ) stret = atof( optarg );
            break;
         case 'b': if( 0 != atol( optarg ) ) buflen = atol( optarg ); break;
```

```
         case 'v': debug_level++; break;
         case 'h':
         default :
            fprintf( stdout,
                     "Опции:\n"
                     " -s - вещественный коэффициент темпо-коррекции\n"
                     " -b - размер аудио буфера\n"
                     " -v - увеличить уровень детализации отладочного вывода\n"
                     " -h - вот этот текст подсказки\n" );
            exit( 'h' == c ? EXIT_SUCCESS : EXIT_FAILURE );
      }
   if( optind == argc )
      fprintf( stdout, "должен быть указан хотя бы один звуковой файл\n" ),
      exit( EXIT_FAILURE );
   if( 1.0 != stret ) sprintf( stretch, " stretch %f", stret );
   else sprintf( stretch, "" );
   buf = malloc( buflen );
   int i;
   for( i = optind; i < argc; i++ ) play_file( argv[ i ] );
   free( buf );
   return EXIT_SUCCESS;
};
```

Исполнение примера на различных форматах аудиофайлов:

```
$ ./o5  male.wav
$ ./o5  male.raw
```

Интересно сравнить времена исполнения:

```
$ time ./o5 -b7000 male.spx
Decoding 8000 Hz audio using narrowband mode (mono)
Encoded with Speex 1.2rc1
real    0m0.093s
user    0m0.000s
sys     0m0.001s
$ time play -q male.wav
real    0m8.337s
user    0m0.009s
sys     0m0.011s
```

Время проигрывания эталонного входного файла более 8 секунд, но в представлнной параллельной реализации главный запускающий процесс запускает процесс проигрывания и завершается через время менее 0.1 секунды.

Наконец последний пример. Предыдущий показанный код получает поток данных извне (из входного фильтра) и, возможно подвергшись некоторым трансформациям, отправляется вовне (в выходной фильтр). Противоположная картина происходит в этом последнем примере: аудио поток (он может генерироваться в этом процессе, в примере он, например, считывается из внешнего файла) из вызывающего процесса передаётся на вход дочернего процесса-фильтра (порождаемого `execvp()`), а результирующий вывод этого фильтра снова, через перехваченный поток, возвращается в вызвавший процесс. Этот пример, в отличие от предыдущих, показан на C++, но это сделано только для того, чтобы изолировать все рутинные действия по созданию дочернего процесса и перехвате его потоков ввода-вывода в отдельный объект класса `chld`. В этом коде есть много интересного из числа POSIX API называвшихся выше: `fork()`, `execvp()`, создание неименованных каналов `pipe()` связи процессов, переназначение на них потоков ввода/вывода `dup2()`, неблокирующий ввод устанавливаемый вызовом `fcntl()` и другие:

e5.cc :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;
using std::flush;
class chld {
   int fi[ 2 ], // pipe – для ввода в дочернем процессе
       fo[ 2 ]; // pipe – для вывода в дочернем процессе
   pid_t pid;
   char** create( const char *s );
public:
   chld( const char*, int* fdi, int* fdo );
};
// это внутренняя private функция-член : построение списка параметров запуска процесса
char** chld::create( const char *s ) {
   char *p = (char*)s, *f;
   int n;
   for( n = 0; ; n++ ) {
     if( ( p = strpbrk( p, " " ) ) == NULL ) break;
     while( *p == ' ' ) p++;
   };
   char **pv = new char* [ n + 2 ];
   for( int i = 0; i < n + 2; i++ ) pv[ i ] = NULL;
   p = (char*)s;
   f = strpbrk( p, " " );
   for( n = 0; ; n++ ) {
      int k = ( f - p );
      pv[ n ] = new char[ k + 1 ];
      strncpy( pv[ n ], p, k );
      pv[ n ][ k ] = '\0';
      p = f;
      while( *p == ' ' ) p++;
      if( ( f = strpbrk( p, " " ) ) == NULL ) {
         pv[ n + 1 ] = strdup( p );
         break;
      }
   }
   return pv;
};
// вот главное "действо" класса – конструктор, здесь переназначаются
// потоки ввода вывода (SYSIN & SYSOUT), клонируется вызывающий процесс,
// и в нём вызывается новый процесс-клиент со своими параметрами:
chld::chld( const char* pr,  int* fdi, int* fdo ) {
   if( pipe( fi ) || pipe( fo ) ) perror( "pipe" ), exit( EXIT_FAILURE );
   // здесь создаётся список параметров запуска
   char **pv = create( pr );
   pid = fork();
   switch( pid ) {
      case -1: perror( "fork" ), exit( EXIT_FAILURE );
      case  0:   // дочерний клон
         close( fi[ 1 ] ), close( fo[ 0 ] );
         if( dup2( fi[ 0 ], STDIN_FILENO ) == -1 ||
             dup2( fo[ 1 ], STDOUT_FILENO ) == -1 )
            perror( "dup2" ), exit( EXIT_FAILURE );
         close( fi[ 0 ] ), close( fo[ 1 ] );
```

```
        // запуск консольного клиента
        if( -1 == execvp( pv[ 0 ], pv ) )
           perror( "execvp" ), exit( EXIT_FAILURE );
        break;
     default:   // родительский процесс
        for( int i = 0;; i++ )
           if( pv[ i ] != NULL ) delete pv[ i ]; else break;
        delete [] pv;
        close( fi[ 0 ] ), close( fo[ 1 ] );
        *fdi = fo[ 0 ];
        int cur_flg;
        // чтение из родительского процесса должно быть в режиме O_NONBLOCK
        cur_flg = fcntl( fo[ 0 ], F_GETFL );
        if( -1 == fcntl( fo[ 0 ], F_SETFL, cur_flg | O_NONBLOCK ) )
           perror( "fcntl" ), exit( EXIT_FAILURE );
        *fdo = fi[ 1 ];
        // для записи O_NONBLOCK не обязательно
        break;
  };
}; // конец определения класса chld
static int debug_level = 0;
static u_long buflen = 1024;
static u_char *buf;
static char stretch[ 80 ] = "";
int main( int argc, char *argv[] ) {
  int c;
  double stret = 1.0;
  while( -1 != ( c = getopt( argc, argv, "vs:b:" ) ) )
     switch( c ) {
        case 's':
           if( 0.0 != atof( optarg ) ) stret = atof( optarg );
           break;
        case 'b': if( 0 != atol( optarg ) ) buflen = atol( optarg ); break;
        case 'v': debug_level++; break;
        case 'h':
        default : cout <<
                    argv[ 0 ] << "[<опции>] <имя вх.файла> <имя вых.файла>\n"
                    "опции:\n"
                    " -s - вещественный коэффициент темпо-коррекции\n"
                    " -b - размер аудио буфера\n"
                    " -v - увеличить уровень детализации отладочного вывода\n"
                    " -h — вот этот текст подсказки\n";
           exit( 'h' == c ? EXIT_SUCCESS : EXIT_FAILURE );
     }
  if( optind != argc - 2 )
     cout << "должно быть указаны имена входного и выходного звуковых файлов"
          << endl, exit( EXIT_FAILURE );
  // файл с которого читается входной аудиопоток
  int fai = open( argv[ optind ], O_RDONLY );
  if( -1 == fai ) perror( "open input" ), exit( EXIT_FAILURE );
  // файл в который пишется результирующий аудиопоток
  int fao = open( argv[ optind + 1 ], O_RDWR | O_CREAT,  // 666
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |  S_IWOTH );
  if( -1 == fao ) perror( "open output" ), exit( EXIT_FAILURE );
  char stretch[ 80 ] = "";
  if( 1.0 != stret ) sprintf( stretch, " stretch %f", stret );
     else sprintf( stretch, "" );
  char comstr[ 120 ] = "sox -V -twav - -twav - ";
  strcat( comstr, stretch );
```

```
// сформирована командная строка дочернего процесса
if( debug_level > 1 ) cout << comstr << endl;
int fdi, fdo;
chld *ch = new chld( comstr, &fdi, &fdo );
// дескриптор с которого читается вывод в stdout дочернего процесса
if( -1 == fdi ) perror( "pipe output" ), exit( EXIT_FAILURE );
// дескриптор куда записывается то, что читает из stdin дочерний процесс
if( -1 == fdo ) perror( "pipe output" ), exit( EXIT_FAILURE );
buf = new u_char[ buflen ];
int sum[] = { 0, 0, 0, 0 };
while( true ) {
   int n;
   if( fai > 0 ) {
      n = read( fai, buf, buflen );
      sum[ 0 ] += n > 0 ? n : 0;
      if( debug_level > 2 )
         cout << "READ from audio\t" << n << " -> " << sum[ 0 ] << endl;
      if( -1 == n ) perror( "read file" ), exit( EXIT_FAILURE );
      if( 0 == n ) close( fai ), fai = -1;
   };
   if( fai > 0 ) {
      n = write( fdo, buf, n );
      sum[ 1 ] += n > 0 ? n : 0;
      if( debug_level > 2 )
         cout << "WRITE to child\t" << n << " -> "
              << ( sum[ 1 ] += n > 0 ? n : 0 ) << endl;
      if( -1 == n ) perror( "write pipe" ), exit( EXIT_FAILURE );
      // передеспетчеризация - дать время на обработку
      usleep( 100 );
   }
   else close( fdo ), fdo = -1;
   n = read( fdi, buf, buflen );
   if( debug_level > 2 )
      cout << "READ from child\t" << n << " -> "
           << ( sum[ 2 ] += n > 0 ? n : 0 ) << flush;
   if( n >= 0 && debug_level > 2 ) cout << endl;
   // это может быть только после закрытия fdo!!!
   if( 0 == n ) break;
   else if( -1 == n ) {
      if( EAGAIN == errno ) {
         if( debug_level > 2 )
            cout << " : == not ready == ... wait ..." << endl;
         usleep( 300 );
         continue;
      }
      else perror( "\nread pipe" ), exit( EXIT_FAILURE );
   }
   n = write( fao, buf, n );
   if( debug_level > 2 )
      cout << "WRITE to file\t" << n << " -> "
           << ( sum[ 3 ] += n > 0 ? n : 0 ) << endl;
   if( -1 == n ) perror( "write file" ), exit( EXIT_FAILURE );
};
close( fai ), close( fao );
close( fdi ), close( fdo );
delete [] buf;
delete ch;
cout << "считано со входа " << sum[ 0 ] << " байт - записано на выход "
     << sum[ 0 ] << " байт" << endl;
```

```
   return EXIT_SUCCESS;
};
```

Детали и опциональные ключи программы оставим для экспериментов, покажем только как программа трансформирует аудио файл в другой аудио файл, с темпо-ритмом увеличенным вдвое (установлен максимальный уровень детализации диагностического вывода, показано только начало вывода диагностики):

```
$ ./e5 -vvv -s0.5 -b7000 male.wav male1.wav
sox -V -twav - -twav -  stretch 0.500000
READ from audio>7000 -> 7000
WRITE to child<>7000 -> 14000
READ from child>-1 -> 0 : == not ready == ... wait ...
READ from audio>7000 -> 14000
WRITE to child<>7000 -> 28000
READ from child>-1 -> 0 : == not ready == ... wait ...
READ from audio>7000 -> 21000
WRITE to child<>7000 -> 42000
READ from child>-1 -> 0 : == not ready == ... wait ...
READ from audio>7000 -> 28000
WRITE to child<>7000 -> 56000
...
```

В выводе видны строки неблокирующего вывода когда данные ещё не готовы (`== not ready == ... wait ...`). Убеждаемся, что это именно то преобразование (темпокоррекция), которое мы добивались, простым прослушиванием:

```
$ play male1.wav
...
```

Результат трансформации аудио файла смотрим ещё и таким образом:

```
$ ls -l male*.wav
-rw-rw-r-- 1 olej olej 48044 Май 12 19:58 male1.wav
-rw-r--r-- 1 olej olej 96044 Авг 21  2008 male.wav
```

Что совершенно естественно: результирующий файл `male1.wav` является копией исходного (по содержимому), с темпокоррекцией в 2 раза в сторону ускорения (число отсчётов и размер файла уменьшились вдвое).

Сигналы

Сигналы UNIX являются специфической и важнейшей составной частью таких систем (достаточно обратить внимание на то, что без сигналов мы не смогли бы завершить ни один процесс в системе). Для начала, в каждой системе, мы посмотрим, какие сигналы в ней обрабатываются:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Примеры кода для этой группы находятся в архиве `usignal.tgz`, все примеры этого архива написаны на C++ (для разнообразия), и используют общий заголовочный файл:

head.h :

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
#define _SIGMIN  SIGHUP
#define _SIGMAX  SIGRTMAX
```

Первый пример показывает как рекомендуют проверять реализован ли тот или иной сигнал в конкретной POSIX OS:

s1.cc :

```
#include "head.h"

int main( int argc, char *argv[] ) {
   cout << "SIGNO";
   for( int i = _SIGMIN; i <= _SIGMAX; i++ ) {
      if( i % 8 == 1 ) cout << endl << i << ':';
      int res = sigaction( i, NULL, NULL );
      cout << '\t' << ( ( res != 0 && errno == EINVAL ) ? '-' : '+' );
   };
   cout << endl;
   return EXIT_SUCCESS;
};
$ ./s1
SIGNO
1:      +       +       +       +       +       +       +       +
9:      +       +       +       +       +       +       +       +
17:     +       +       +       +       +       +       +       +
25:     +       +       +       +       +       +       +       -
33:     -       +       +       +       +       +       +       +
41:     +       +       +       +       +       +       +       +
49:     +       +       +       +       +       +       +       +
57:     +       +       +       +       +       +       +       +
```

Модель ненадёжной обработки сигналов

Модель ненадёжной обработки сигналов (старая модель) строится на вызове `signal()`, устанавливающем новую диспозицию сигнала (новую функцию-обработчик, `SIG_IGN`, или `SIG_DFL`) :

s2.cc :

```
#include "head.h"

static void handler( int signo ) {
   signal( SIGINT, handler );
   cout << endl << "signal #" << signo << endl;
};

int main() {
   signal( SIGINT, handler );
   signal( SIGSEGV, SIG_DFL );
   signal( SIGTERM, SIG_IGN );
   while( true ) pause();
};
$ ./s2
^C
```

```
signal #2
^C
signal #2
Убито
```

После установки нового обработчика сигнала (`SIGINT = 2`) процесс невозможно остановить по ^C, и мы останавливаем его, посылая ему с другого терминала `SIGKILL = 9` :

```
$ ps -A | grep s2
18364 pts/7    00:00:00 s2
$ kill -9 18364
```

Из этой же области модели ненадёжной обработки сигналов и широко используемый вызов `alarm()` (тайм-аут):

```
#include "head.h"

int main( void ) {
    alarm( 5 );
    cout << "Waiting to die in 5 seconds ..." << endl;
    pause();
    return EXIT_SUCCESS;
};
$ time ./s3
Waiting to die in 5 seconds ...
Сигнал таймера
real    0m5.002s
user    0m0.000s
sys     0m0.003s
```

На такой модели строится перехват сигнала завершения процесса для сохранения данных прежде завершения:

s4 .cc :

```
#include "head.h"

static void handler( int signo ) {
   cout << endl << "Saving data ... wait" << endl;
   sleep( 2 );
   cout << " ... data saved!" << endl;
   exit( EXIT_SUCCESS );
};

int main() {
   signal( SIGINT, handler );
   while( true ) pause();
};
$ ./s4
^C
Saving data ... wait
 ... data saved!
```

Модель надёжной обработки сигналов

Модель надёжной обработки сигналов на основе новой системы понятий:

1. Сигнальной маски типа: `sigset_t` — по одному биту на каждый представляемый сигнал;

2. Набор функций заполнения/очистки сигнальной маски:

```
int sigemptyset( sigset_t* );
int sigfillset( sigset_t* );
```

```
int sigaddset( sigset_t*, int signo );
int sigdelset( sigset_t*, int signo ), ...
```

3. Маскирование реакции на сигнал:

```
int sigprocmask ( int how, const sigset_t* set, sigset_t* oset );
```

- где `how` может быть:

`SIG_BLOCK` — добавить сигналы к сигнальной маске процесса (заблокировать доставку);

`SIG_UNBLOCK` — сбросить сигналы из сигнальной маски процесса (разблокировать доставку);

`SIG_SETMASK` — установить как сигнальную маску процесса;

`set` и `oset` — устанавливаемая и ранее установленная (для сохранения) маска процесса.

4. Структура описывающая диспозицию сигнала:

```
struct sigaction {
   union {    /* Signal handler.  */
      void (*sa_handler) ( int ) {                    /* Used if SA_SIGINFO is not set.  */
      void (*sa_sigaction) ( int, siginfo_t*, void*); /* Used if SA_SIGINFO is set.  */
   }
   sigset_t sa_mask;                  /* Additional set of signals to be blocked.  */
   int sa_flags;                      /* Special flags.  */
...
};
```

Маска `sa_mask` содержит сигналя, которые будут автоматически заблокированы в обработчике сигнала.

Возможные значения поля флагов:

`SA_RESETHANG` — после срабатывания обработчика сигнала будет восстановлен обработчик по умолчанию (`SIG_DFL`, что соответствует духу ненадёжной модели и позволяет воспроизвести её поведение);

`SA_NOCLDSTOP` — используется только для сигнала `SIGCHLD` и указывает системе не генерировать для родительского процесса `SIGCHLD` если дочерний процесс завершается по `SIGSTOP`;

`SA_SIGINFO` — будет организована очередь доставки сигналов (модель сигналов реального времени), при этом обработчику будет доставляться дополнительная информация о сигнале — структура `siginfo_t` и дополнительные параметры пользователя (при этом используется другой прототип обработчика `sa_sigaction`);

5. Функция установки диспозиции:

```
/* Get and/or set the action for signal SIG.  */
extern int sigaction( int signo, const struct sigaction* act, struct sigaction* oact );
```

где: `act` и `oact` — новая устанавливаемая, и прежняя ранее установленная (для сохранения) диспозиции, соответсвенно.

s8.cc :

```
#include "head.h"
void catchint( int signo ) {
   cout << "SIGINT: signo = " << signo << endl;
};

int main() {
   static struct sigaction act = { &catchint, 0, 0 }; /* 0 = (sigset_t)NULL }; */
   sigfillset( &(act.sa_mask) );
   sigaction( SIGINT, &act, NULL );
   for( int i = 0; i < 20; i++ ) sleep( 1 ), cout << "Cycle # " << i << endl;
};
```

```
$ ./s8
Cycle # 0
```

```
^CSIGINT: signo = 2
Cycle # 1
^CSIGINT: signo = 2
Cycle # 2
^CSIGINT: signo = 2
Cycle # 3
^CSIGINT: signo = 2
Cycle # 4
^CSIGINT: signo = 2
Cycle # 5
^CSIGINT: signo = 2
Cycle # 6
Cycle # 7
...
```

Модель обработки сигналов реального времени

Последняя модель - модель обработки сигналов реального времени — уже описана ранее, она определяется флагом `SA_SIGINFO` в структуре `struct sigaction`. Вот пример, в котором родительский процесс посылает дочернему «пачки» сигналов и завершается, только после чего дочерний процесс принимает сигналы; хорошо видно, что принимается вся последовательность посланных сигналов:

s5.cc :

```
#include "head.h"

static void handler( int signo, siginfo_t* info, void* context ) {
   cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
        << "received signal " << signo << endl;
};

int main( int argc, char *argv[] ) {
   int opt, val, beg = _SIGMAX, num = 3, fin = _SIGMAX - num, seq = 3;
   bool wait = false;
   while ( ( opt = getopt( argc, argv, "b:e:n:w") ) != -1 ) {
      switch( opt ) {
         case 'b' : if( atoi( optarg ) > 0 ) beg = atoi( optarg ); break;
         case 'e' :
            if( ( atoi( optarg ) != 0 ) && ( atoi( optarg ) < _SIGMAX ) ) fin = atoi( optarg );
            break;
         case 'n' : if( atoi( optarg ) > 0 ) seq = atoi( optarg ); break;
         case 'w' : wait = true; break;
         default :
            cout << "usage: " << argv[ 0 ]
                 << " [-b #signal] [-e #signal] [-n #loop] [-w]" << endl;
            exit( EXIT_FAILURE );
            break;
      }
   };
   num = fin - beg;
   fin += num > 0 ? 1 : -1;
   sigset_t sigset;
   sigemptyset( &sigset );
   for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) ) sigaddset( &sigset, i );
   pid_t pid;
   if( pid = fork() == 0 ) {
      // дочерний процесс: здесь сигналы обрабатываются
      sigprocmask( SIG_BLOCK, &sigset, NULL );
      for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) ) {
         struct sigaction act, oact;
```

```
         sigemptyset( &act.sa_mask );
         act.sa_sigaction = handler;
         act.sa_flags = SA_SIGINFO;          // вот оно - реальное время!
         if( sigaction( i, &act, NULL ) < 0 ) perror( "set signal handler: " );
      };
      cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
           << "signal mask set" << endl;
      sleep( 3 );                            // пауза для отсылки сигналов родителем
      cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
           << "signal mask unblock" << endl;
      sigprocmask( SIG_UNBLOCK, &sigset, NULL );
      sleep( 3 );                            // пауза для получения сигналов
      cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
           << "finished" << endl;
      exit( EXIT_SUCCESS );
   }
   // родительский процесс: отсюда сигналы посылаются
   sigprocmask( SIG_BLOCK, &sigset, NULL );
   sleep( 1 );                                // пауза для установок дочерним процессом
   for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) ) {
      for( int j = 0; j < seq; j++ ) {
         kill( pid, i );
         cout << "PARENT\t[" << getpid() << ":" << getppid() << "] : "
              << "signal sent: " << i << endl;
      };
   };
   if( wait ) waitpid( pid, NULL, 0 );
   cout << "PARENT\t[" << getpid() << ":" << getppid() << "] : "
        << "finished" << endl;
   exit( EXIT_SUCCESS );
};
```

```
$ ./s5
CHILD  [20934:20933] : signal mask set
PARENT [20933:5281] : signal sent: 64
PARENT [20933:5281] : signal sent: 64
PARENT [20933:5281] : signal sent: 64
PARENT [20933:5281] : signal sent: 63
PARENT [20933:5281] : signal sent: 63
PARENT [20933:5281] : signal sent: 63
PARENT [20933:5281] : signal sent: 62
PARENT [20933:5281] : signal sent: 62
PARENT [20933:5281] : signal sent: 62
PARENT [20933:5281] : signal sent: 61
PARENT [20933:5281] : signal sent: 61
PARENT [20933:5281] : signal sent: 61
PARENT [20933:5281] : finished
$
CHILD  [20934:1] : signal mask unblock
CHILD  [20934:1] : received signal 64
CHILD  [20934:1] : received signal 64
CHILD  [20934:1] : received signal 64
CHILD  [20934:1] : received signal 63
CHILD  [20934:1] : received signal 63
CHILD  [20934:1] : received signal 63
CHILD  [20934:1] : received signal 62
CHILD  [20934:1] : received signal 62
CHILD  [20934:1] : received signal 62
CHILD  [20934:1] : received signal 61
CHILD  [20934:1] : received signal 61
```

```
CHILD   [20934:1] : received signal 61
CHILD   [20934:1] : finished
```

Хорошо видно, что к моменту получения сигналов, родительским процессом для получателя является процесс `init` (PID=1), то есть родительский процесс к этому времени уже завершился.

Более того, сигналы по схеме реального времени могут отправляться не вызовом `kill()`, а вызовом `sigqueue()`, который позволяет к сигналам, отправляемым в порядке очереди присоединять данные:

```
$ man sigqueue
SIGQUEUE(2)                Linux Programmer's Manual               SIGQUEUE(2)
NAME
       sigqueue, rt_sigqueueinfo - queue a signal and data to a process
SYNOPSIS
       #include <signal.h>
      int sigqueue(pid_t pid, int sig, const union sigval value);
...
    union sigval {
               int   sival_int;
               void *sival_ptr;
           };
```

Обычно указатель `sival_ptr` и используют для присоединения к сигналу поля данных.

Сигналы в потоках

Всё, что показано выше, относится к посылке сигналов однопоточному приложению. Модель посылки сигналов приложению из многих потоков введена POSIX 1003.b (расширение реального времени), и будет рассмотрено после рассмотрения техники потоков.

Параллельные потоки

Реализация потоков в Linux выполнена в соответствии с POSIX 1003.b (POSIX реального времени). Все определения находятся с `<pthread.h>`, развитие этой линии API а).достаточно позднее, б).достаточно продолжительное и в).продолжается:

```
/* Copyright (C) 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
   Free Software Foundation, Inc.
   This file is part of the GNU C Library.
... */
```

Всё, что касается API и определений потоков POSIX, является общим стандартом, **намного шире** по детализации и возможностям, чем, например, механизм потоков ядра Linux, этот API насчитывает многие десятки вызовов. Этот механизм принципиально отличается от API потоков, принятый в Windows. Кроме собственно определения потоков и операций с ними, в `<pthread.h>` описываются реализация и примитивов синхронизации в соответствии с стандартом реального времени POSIX 1003.b : мьютексы — `pthread_mutex_t`, блокировки чтения/записи — `pthread_rwlock_t`, условные переменные — `pthread_cond_t`, спин-блокировки — `pthread_spinlock_t`, барьеры — `pthread_barrier_t`, а также все API для работы с ними. Здесь же определено всё, что относится к такой специфической части как :

```
int pthread_atfork( void(*prepare)(void), void(*parent)(void), void (*child)(void) );
```

Создание потока

Новый поток создаётся вызовом :

```
int pthread_create( pthread_t *newthread, const pthread_attr_t *attr,
                    void *(*start_routine)(void*), void *arg );
```

Все примеры кода для этой группы находятся в архиве `upthread.tgz`. Простейший пример (но на котором можно очень много увидеть из области работы с потоками) :

ptid.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void put_msg( char *title, struct timeval *tv ) {
   printf( "%02u:%06lu : %s\t: pid=%lu, tid=%lu\n",
           ( tv->tv_sec % 60 ), tv->tv_usec, title, getpid(), pthread_self() );
}

void *test( void *in ) {
   struct timeval *tv = (struct timeval*)in;
   gettimeofday( tv, NULL );
   put_msg( "pthread started", tv );
   sleep( 5 );
   gettimeofday( tv, NULL );
   put_msg( "pthread finished", tv );
   return NULL;
}

#define TCNT 5
static pthread_t tid[ TCNT ];
int main( int argc, char **argv, char **envp ) {
   pthread_t tid[ TCNT ];
   struct timeval tm;.
   int i;.
   gettimeofday( &tm, NULL );
   put_msg( "main started", &tm );
   for( i = 0; i < TCNT; i++ ) {
      int status = pthread_create( &tid[ i ], NULL, test, (void*)&tm );
         if( status != 0 ) perror( "pthread_create" ), exit( EXIT_FAILURE );
   };
   for( i = 0; i < TCNT; i++ )
      pthread_join( tid[ i ], NULL ); // это обычная техника ожидания!
   gettimeofday( &tm, NULL );
   put_msg( "main finished", &tm );
   return( EXIT_SUCCESS );
}
```

В отличие от многих других UNIX, в Linux компиляция примеров с таким вызовом завершится ошибкой:

```
$ g++ ptid.cc -o ptid
/tmp/ccnW2hnx.o: In function `main':
ptid.cc:(.text+0x6e): undefined reference to `pthread_create'
collect2: выполнение ld завершилось с кодом возврата 1
```

Необходимо явное всключение библиотеки `libpthread.so` в сборку:

```
$ ls /usr/lib/*pthr*
/usr/lib/libgpgme-pthread.so.11       /usr/lib/libgpgme++-pthread.so.2.4.0
/usr/lib/libgpgme-pthread.so.11.6.6   /usr/lib/libpthread_nonshared.a
/usr/lib/libgpgme++-pthread.so.2      /usr/lib/libpthread.so
```

В строку компиляции нужно дописать:

```
$ gcc ptid.c -lpthread -o ptid
```

Выполнение этого примера:

```
$ ./ptid
```

```
50:259188 : main started      : pid=13745, tid=3079214784
50:259362 : pthread started   : pid=13745, tid=3079211888
50:259395 : pthread started   : pid=13745, tid=3068722032
50:259403 : pthread started   : pid=13745, tid=3058232176
50:259453 : pthread started   : pid=13745, tid=3047742320
50:259466 : pthread started   : pid=13745, tid=3037252464
55:259501 : pthread finished  : pid=13745, tid=3079211888
55:259501 : pthread finished  : pid=13745, tid=3068722032
55:259525 : pthread finished  : pid=13745, tid=3058232176
55:259532 : pthread finished  : pid=13745, tid=3047742320
55:259691 : pthread finished  : pid=13745, tid=3037252464
55:259936 : main finished     : pid=13745, tid=3079214784
```

Параметр (1-й) `newthread` вызова является адресом идентификатором создаваемого потока (куда будет возвращён идентификатор), типа `pthread_t`, определённого в </usr/include/bits/pthreadtypes.h> :

```
typedef unsigned long int pthread_t; /* Thread identifiers. */
```

Этот идентификатор принципиально отличается от идентификатора присваиваемого ядром, для которого предусмотрен вызов `gettid()` (показан вывод одновременно с выполнением примера выше):

```
$ ps -efL | grep ptid
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
olej     13745  8859 13745  0    6 17:14 pts/10   00:00:00 ./ptid
olej     13745  8859 13746  0    6 17:14 pts/10   00:00:00 ./ptid
olej     13745  8859 13747  0    6 17:14 pts/10   00:00:00 ./ptid
olej     13745  8859 13748  0    6 17:14 pts/10   00:00:00 ./ptid
olej     13745  8859 13749  0    6 17:14 pts/10   00:00:00 ./ptid
olej     13745  8859 13750  0    6 17:14 pts/10   00:00:00 ./ptid
```

Этот же идентификатор потока (типа `pthread_t`) может быть позже быть получен в самом потоке вызовом:

```
pthread_t pthread_self( void );
```

Параметры создания потока

Созданный поток может иметь много параметров, определяющих его поведение. Эти параметры описываются в атрибутной записи потока — параметр `attr` (2-й) при создании потока. Если в качестве этого параметра указывается NULL, то создаётся поток с параметрами по умолчанию. Основные определения (константы) для таких параметров:

```
enum  { /* Detach state.  */
  PTHREAD_CREATE_JOINABLE,
  PTHREAD_CREATE_DETACHED
};
enum { /* Mutex protocols.  */
  PTHREAD_PRIO_NONE,
  PTHREAD_PRIO_INHERIT,
  PTHREAD_PRIO_PROTECT
};
enum { /* Scheduler inheritance.  */
  PTHREAD_INHERIT_SCHED,
  PTHREAD_EXPLICIT_SCHED
};
enum { /* Scope handling.  */
  PTHREAD_SCOPE_SYSTEM,
  PTHREAD_SCOPE_PROCESS
};
```

Параметры определяются в структуре типа `pthread_attr_t`. В Linux этот тип определён в </usr/include/bits/pthreadtypes.h>, примерно так:

```
#define __SIZEOF_PTHREAD_ATTR_T 36
typedef union {
  char __size[__SIZEOF_PTHREAD_ATTR_T];
  long int __align;
} pthread_attr_t;
```

Непосредственно с работа не производится, есть множество API для установки и чтения разных параметров из атрибутной записи потока.

При создании дефаултной атрибутной записи потока (`PTHREAD_JOINABLE, SCHED_OTHER, ...`) она должна быть инициализирована:

```
int pthread_attr_init( pthread_attr_t *attr );
```

После старта потока атрибутная запись уже не нужна и может быть переинициализирована (если предполагается ещё инициировать потоки), или должна быть уничтожена:

```
int pthread_attr_destroy( pthread_attr_t *attr ) ;
```

После создания атрибутной записи потока к ней применяются множество функций, подготавливающих нужный набор параметров атрибутов запуска, функции вида `pthread_attr_*()`, смысл большинства из них понятен без комментариев:

```
int pthread_attr_getschedparam( const pthread_attr_t *attr, struct sched_param *param) ;
int pthread_attr_setschedparam( pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_getschedpolicy( const pthread_attr_t *attr, int *policy );
int pthread_attr_setschedpolicy( pthread_attr_t *attr, int policy );
...
int pthread_attr_setaffinity_np( pthread_attr_t *attr,
                                 size_t cpusetsize, const cpu_set_t *cpuset ) ;
int pthread_attr_getaffinity_np( const pthread_attr_t *attr,
                                 size_t cpusetsize, cpu_set_t *cpuset ) ;
...
int pthread_attr_getdetachstate( const pthread_attr_t *attr, int *detachstate );
int pthread_attr_setdetachstate( pthread_attr_t *attr, int detachstate) ;
int pthread_attr_getguardsize( const pthread_attr_t *attr, size_t *guardsize );
```

- получить размер охранной области, создаваемой для защиты от переполнения стека.

```
extern int pthread_attr_setguardsize( pthread_attr_t *attr, size_t guardsize) ;
```

- установить размер охранной области, создаваемой для защиты от переполнения стека.

```
int pthread_attr_getinheritsched( const pthread_attr_t *attr, int *inherit );
```

- получить характер наследования (`PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED`) параметров для потока.

```
int pthread_attr_setinheritsched( pthread_attr_t attr, int inherit );
```

- установить характер наследования (`PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED`) параметров для потока.

```
int pthread_attr_getscope( const pthread_attr_t *attr, int *scope );
```

- получить область деспетчирования для потока (`PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS`);

```
int pthread_attr_setscope( pthread_attr_t *attr, int scope) ;
```

- установить область деспетчирования для потока (`PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS`);

```
int pthread_attr_getstackaddr( const pthread_attr_t *attr, void **stackaddr ) ;
```

- получить адрес, ранее установленный для стека;

```
int pthread_attr_setstackaddr( pthread_attr_t *attr, void *stackaddr ) ;
```

- установить адрес стека, минимальный размер кадра стека `PTHREAD_STACK_MIN`;

```
int pthread_attr_getstacksize( const pthread_attr_t *attr, size_t *stacksize ) ;
```

– получить текущий установленный минимальный размер стека;

```
int pthread_attr_setstacksize( pthread_attr_t *attr, size_t __stacksize)
```

- добавить информацию о минимальном стеке, необходимом для старта потока; этот размер не может быть менее `PTHREAD_STACK_MIN`, и не должен превосходить установленные в системе пределы;

```
int pthread_getattr_np( pthread_t th, pthread_attr_t *attr );
```

- инициализировать атрибутную запись нового потока в соответствии с атрибутной записью ранее существующего;

Поток, созданный как присоединённый (по умолчанию это так), может быть позже отсоединён вызовом:

```
int pthread_detach( pthread_t th );
```

Но переведен обратно в состояние присоединённости (`PTHREAD_JOINABLE`) он более быть не может.

Временные затраты на создание потока

Теперь проделаем то же, что уже делалось при клонировании процесса, и сравним времена создания нового процесса и нового потока:

p2-2.c :

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "libdiag.h"

static uint64_t tim;
void* threadfunc ( void* data ) {
   tim = rdtsc() - tim;
   pthread_exit( NULL );
   return NULL;
};

int main( int argc, char *argv[] ) {
   tim = rdtsc();
   pthread_t tid;
   pthread_create( &tid, NULL, threadfunc, NULL );
   pthread_join( tid, NULL );
   printf( "thread create time : %llu\n", tim );
   exit( EXIT_SUCCESS );
};
```

Несколько циклов сравнительного выполнения (`p2-1` - создание процесса, `p2-2` - создание потока, запуски чередуем, чтобы уменьшить влияние кэширования страниц памяти):

```
$ ./p2-1
process create time : 525430
$ ./p2-2
thread create time : 314980
$ ./p2-1
process create time : 2472100
$ ./p2-2
thread create time : 362210
$ ./p2-1
process create time : 342490
$ ./p2-2
thread create time : 333800
```

Результаты абсолютно идентичный, в пределах статистической погрешности. Вывод: сам процесс создания и потока и процесса — требуют одинаковых затрат времени (вопреки многим утверждениям в учебниках).

Активность потока

Вот таким вызовом поток может передать управление другому потоку (какому — неизвестно):

```
int pthread_yield( void );
```

Какому потоку будет передана активность (этого процессора) — вопрос непредсказуемый! Это может быть даже тот же самый наш поток, только что выполнивший `pthread_yield()`.

К **такому же** результату (передача активности иному потоку) приведёт и **любой** вызов, переводящий поток в блокированное (пассивное) состояние, например `sleep()`, `pause()` и подобные им.

Завершение потока

Условия и возможности завершения потока гораздо сложнее и разнообразнее, чем его запуск. Новый созданный поток завершается в одном из следующих случаев:

- Сам поток вызывает `pthread_exit()`, и завершается со статусом завершения, доступным другому потоку по вызову ожидания `pthread_join()`;
- Поток осуществляет возврат из функции потока, это эквивалентно `pthread_exit()`, возвращаемое значение является кодом возврата;
- Поток завершается по `pthread_cancel()` (это отдельный вопрос, рассматриваемый далее);
- Какой либо поток процесса вызывает `exit()` , или сама главная программа `main` завершается — все порождённые потоки процесса также завершаются.

Это вызывается в потоке при его завершении:

```
void pthread_exit( void *retval );
```

А это — в вызывающем потоке, ожидающем завершения:

```
int pthread_join( pthread_t th, void **return );
```

Детально поведение потока при завершении определяется ещё одной группой параметров, задаваемых в атрибутной записи потока `pthread_attr_t` :

```
enum { /* Cancellation – состояние завершаемости */
  PTHREAD_CANCEL_ENABLE,
  PTHREAD_CANCEL_DISABLE
};
enum { /* тип завершаемости */
  PTHREAD_CANCEL_DEFERRED,
  PTHREAD_CANCEL_ASYNCHRONOUS
};
```

И соответствующие API:

```
int pthread_setcancelstate( int state, int *oldstate );
int pthread_setcanceltype( int type, int *oldtype);
```

Отметка очередной точки отмены потока:

```
void pthread_testcancel( void );
```

Отменить поток немедленно, или при ближайшей возможности:

```
int pthread_cancel( pthread_t th );
```

И, наконец, последнее: стек процедур завершения.

```
void pthread_cleanup_push( void(*routine)(void*), void *arg );
void pthread_cleanup_push( int exec );
```

Примечание: На самом деле такие вызовы определены как макросы, что не меняет техники их использования:

```
#define pthread_cleanup_push( routine, arg )
```

```
#define pthread_cleanup_pop( execute )
```

Но это требует, чтобы использования в коде `pthread_cleanup_push` и `pthread_cleanup_push` были парными. Первый из этих вызовов добавляет новую процедуру завершения в стек, а второй — выталкивает последнюю находящуюся процедуру завершения из стека, и если параметр не нулевой — выполняет эту процедуру.

Данные потока

Собственные данные потока

Техника создания собственных данных потоков (thread specific data) создаёт по одному экземпляру каждого вида данных. Стандарт POSIX указывает, что это число видов данных (тип `pthread_key_t`) не превышает 128. Последовательность действий при создании TSD:

1. Поток запрашивает `pthread_key_create()` для создания ключа доступа к блоку данных определённого типа; если потоку нужно иметь несколько блоков данных разной типизациии (назначения), он делает такой вызов нужное число раз.

2. Некоторая сложность здесь в том, что запросить распределение ключа должен только **один** поток, первым достигший точки распределения. Последующие потоки должны только воспользоваться ранее распределённым значением ключа. Для разрешения этой сложности вводится вызов `pthread_once()`.

3. Теперь каждый поток, использующий блок данных, должен запросить специфический экземпляр данных по `pthread_getspecific()` и, если он убеждается, что это `NULL`, то запросить распределение блока для этого значения ключа по `pthread_setspecific()`.

4. В дальнейшем поток (и все вызываемые из него функции) может работать со своим экземпляром, запрашивая его по `pthread_getspecific()`.

5. При завершении любого потока система уничтожает и его экземпляр данных. При этом вызывается деструктор пользователя, который устанавливается при создании ключа `pthread_key_create()`. Деструктор единый для всех экземпляров данных во всех потоках для этого значения ключа (`pthread_key_t`), но он получает параметром значение указателя на экземпляр данных завершаемого потока.

Всё это гораздо легче показать на примере кода:

```
static  pthread_key_t key;
static pthread_once_t once = PTHREAD_ONCE_INIT;
typedef struct data_bloc {                 // наш собственный тип данных
   //...
} data_t;
static void destructor( data_t *db ) {     // деструктор собственных данных
   free( db );
}
static void once_creator( void ) {         // создаёт единый на процесс ключ для данных data_t
   pthread_key_create( &key, destructor );
}
void* thread_proc( void *data ) {         // функция потока
   pthread_once( &once,  once_creator );  // гарантия единичности создания ключа
   if( pthread_getspecific( key ) == NULL )
      pthread_setspecific( key, malloc( sizeof( data_t ) ) );
   // теперь везде в вызываемых потоком функциях:
```

```
   data_t *db = pthread_getspecific( key );
   // ...
}
```

Далее пример показывает разного рода данные, используемые потоком: а). параметр, передаваемый функции потока в стеке (и точно так же локальные данные функции потока), б). глобальные данные доступные всем потокам, в). экземпляр собственных данных потока.

own.c :

```
include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

static int global = 0;
static  pthread_key_t key;
typedef struct data_bloc {                 // наш собственный тип данных
   pthread_t tid;
} data_t;

void put_msg( int param ) {
   printf( "global=%u , parameter=%u , own=%lu\n",
           global, param, ((data_t*)pthread_getspecific( key ))->tid );
}

static pthread_once_t once = PTHREAD_ONCE_INIT;
static void destructor( void* db ) {       // деструктор собственных данных
   data_t *p = (data_t*)db;
   free( p );
}

static void once_creator( void ) {         // создаёт единый на процесс ключ для данных data_
   pthread_key_create( &key, destructor );
}

void* thread_proc( void *data ) {         // функция потока
   int param = (int)data;
   global++;
   pthread_once( &once,  once_creator );  // гарантия единичности создания ключа
   pthread_setspecific( key, malloc( sizeof( data_t ) ) );
   data_t *db = pthread_getspecific( key );
   db->tid = pthread_self();
   put_msg( param );
   return NULL;
}

int main( int argc, char **argv, char **envp ) {
#define TCNT 5
   pthread_t tid[ TCNT ];
   int i;
   for( i = 0; i < TCNT; i++ )
      pthread_create( &tid[ i ], NULL, thread_proc, (void*)( i + 1 ) );
   for( i = 0; i < TCNT; i++ ).
      pthread_join( tid[ i ], NULL );
   return( EXIT_SUCCESS );
}
```

... и весьма неожиданные и поучительные результаты выполнения такого примера (два последовательно выполненные прогона, которые существенно отличаются выполнением):

```
$ ./own
global=1 , parameter=4 , own=3047005040
global=2 , parameter=3 , own=3057494896
global=3 , parameter=1 , own=3078474608
global=4 , parameter=5 , own=3036515184
global=5 , parameter=2 , own=3067984752
$ ./own
global=4 , parameter=1 , own=3078527856
global=5 , parameter=4 , own=3042863984
global=4 , parameter=3 , own=3057548144
global=4 , parameter=2 , own=3068038000
global=5 , parameter=5 , own=3030383472
```

Сигналы в потоках

Сигналы не могут направляться отдельным потокам процесса — сигналы направляются процессу в целом, как оболочке, обрамляющей несколько потоков. Точно так же, для каждого сигнала может быть переопределена функция-обработчик, но это переопределение действует глобально в рамках процесса.

= = = = = = = = = =

здесь Рис. : прохождение сигнала сквозь многопотоковый процесс.

= = = = = = = = = =

Тем не менее, каждый из потоков (в том числе и главный поток процесса `main() {...}`) могут независимо определить (в терминах модели надёжной обработки сигналов, сигнальных наборов) собственную маску реакции на сигналы. Таким образом оказывается возможным: а). распределить потоки, ответственные за обработку каждого сигнала, б). динамически изменять потоки, в которых (в контексте которых) обрабатывается реакция на сигнал и в). создавать обработчики сигналов в виде отдельных потоков, специально для того предназначенных.

Ниже показан многопоточный пример (3 потока сверх главного), в котором направляемая извне (из другой консоли) последовательность повторяемого сигнала поочерёдно обрабатывается каждым из дочерних потоков по 1-му разу, после чего реакция на сигнал блокируется:

s6.cc :

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
using namespace std;

static void handler( int signo, siginfo_t* info, void* context ) {
   cout << "sig=" << signo << "; tid=" << pthread_self() << endl;
};

sigset_t sig;
void* threadfunc ( void* data ) {
   sigprocmask( SIG_UNBLOCK, &sig, NULL );
   while( true ) {
      pause();
      sigprocmask( SIG_BLOCK, &sig, NULL );
   }
   return NULL;
};
```

```
int main() {
   const int thrnum = 3;
   sigemptyset( &sig );
   sigaddset( &sig, SIGRTMIN );
   sigprocmask( SIG_BLOCK, &sig, NULL );
   cout << "main + " << thrnum << " threads : waiting fot signal " << SIGRTMIN
        << "; pid=" << getpid() << "; tid(main)=" << pthread_self() << endl;
   struct sigaction act;
   act.sa_mask = sig;
   act.sa_sigaction = handler;
   act.sa_flags = SA_SIGINFO;
    if( sigaction( SIGRTMIN, &act, NULL ) < 0 ) perror( "set signal handler: " );
   pthread_t pthr;
   for( int i = 0; i < thrnum; i++ )
      pthread_create( &pthr, NULL, threadfunc, NULL );
   pause();
};
```

Вот как происходит выполнение этого процесса:

```
$ ./s6
main + 3 threads : waiting fot signal 34; pid=7455; tid(main)=3078510288
sig=34; tid=3078503280
sig=34; tid=3068013424
sig=34; tid=3057523568
^C
$ kill -34 7455
$ kill -34 7455
$ kill -34 7455
$ kill -34 7455
$ kill -34 7455
```

Хорошо видно, что:

- главный поток процесса не реагирует на получаемые извне сигналы, его реакция изначально заблокирована;

- `tid` потока, который принимает каждый последующий сигнал, отличается от предыдущего;

- после 3-х реакций, обслуженных в каждом из 3-х потоков, реагирование на этот сигнал прекращается (блокируется).

Пользуясь гибкостью API расширения реального времени POSIX 1003.b, можно построить реакцию на получаемые сигналы в отдельных обрабатывающих потоках, вообще без обработчиков сигналов (со своими ограничениями на операции в контексте сигнального обработчика). В следующем примере процесс приостанавливается (или мог бы выполнять другую полезную работу) до тех пор, пока поток обработчика сигналов не сообщит о завершении.

sigthr.c :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <pthread.h>

int quitflag = 0;
sigset_t mask;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wait = PTHREAD_COND_INITIALIZER;

void* threadfunc ( void* data ) {
   int signo;
```

```
   while( 1 ) {
      if( sigwait( &mask, &signo ) != 0 )
         perror( "sigwait:" ), exit( EXIT_FAILURE );
      switch( signo ) {
         case SIGINT:
            printf( "  ... signal SIGINT\n" );
            break;
         case SIGQUIT:
            printf( "  ... signal SIGQUIT\n" );
            pthread_mutex_lock( &lock );
            quitflag = 1;
            pthread_mutex_unlock( &lock );
            pthread_cond_signal( &wait );
            return NULL;
         default:
            printf( "undefined signal %d\n", signo ), exit( EXIT_FAILURE );
      }
   };
};

int main() {
   printf( "process started with PID=%d\n", getpid() );
   sigemptyset( &mask );
   sigaddset( &mask, SIGINT );
   sigaddset( &mask, SIGQUIT );
   sigset_t oldmask;
   if( sigprocmask( SIG_BLOCK, &mask, &oldmask ) < 0 )
      perror( "signals block:" ), exit( EXIT_FAILURE );
   pthread_t tid;
   if( pthread_create( &tid, NULL, threadfunc, NULL ) != 0 )
      perror( "thread create:" ), exit( EXIT_FAILURE ); ;
   pthread_mutex_lock( &lock );
   while( 0 == quitflag )
      pthread_cond_wait( &wait, &lock );
   pthread_mutex_unlock( &lock );
   /* SIGQUIT был перехвачен, но к этому моменту снова заблокирован */
   if( sigprocmask( SIG_SETMASK, &oldmask, NULL ) < 0 )
      perror( "signals set:" ), exit( EXIT_FAILURE );
   return EXIT_SUCCESS;
};
```

Примечание: Изменения флага `quitflag` производится под защитой мьютекса `lock`, чтобы главный поток не мог пропустить изменение значения флага.

Выполнение задачи:

```
$ ./sigthr
^C  ... signal SIGINT
^C  ... signal SIGINT
^C  ... signal SIGINT
^\  ... signal SIGQUIT
```

Расширенные операции ввода-вывода

К этой части обычно относят рассмотрение синхронного-асинхронного ввода-вывода, и вызовы `select()`, `pselect()`, `poll()`, `epoll()`. Пожалуй, самую ясную и строгую классификацию моделей ввода-вывода в UNIX дал У. Р. Стивенс в [3] :

Прежде чем начать описание функций `select` и `poll`, мы должны вернуться назад и уяснитьосновные различия между пятью моделями ввода-вывода, доступными нам в Unix:

- *блокируемый ввод-вывод;*

- *неблокируемый ввод-вывод;*

- *мультиплексирование ввода-вывода (функции `select` и `poll`);*

- *ввод-вывод, управляемый сигналом (сигнал `SIGIO`);*

- *асинхронный ввод-вывод (функции POSIX.1 `aio_`).*

Блокируемый ввод — это самый часто используемый, и самый известный вариант, когда выполняется операция `read()`, или даже элементарные вызовы `getchar()` или `gets()`, выполняемые в каноническом режиме ввода с терминала (консоли). Эта модель ввода-вывода не нуждается в детальных комментариях.

Неблокирующий ввод-вывод

Неблокирующий ввод-вывод не ожидает наличия данных (или возможности вывода), результат выполнения операции, или невозможность её выполнения в данный момент определяется по анализу кода возврата. Пример (файл `e5.cc` архива `fork.tgz`) неблокирующего ввода был показан выше (в примере запуска дочернего процесса-фильтра). Схематично (убрано всё лишнее) это выглядит так:

```
int fo[ 2 ];                              // pipe - для чтения из дочернего процесса
if( pipe( fo ) ) perror( "pipe" ), exit( EXIT_FAILURE );
close( fo[ 1 ] );
int cur_flg = fcntl( fo[ 0 ], F_GETFL );  // чтение должно быть в режиме O_NONBLOCK
if( -1 == fcntl( fo[ 0 ], F_SETFL, cur_flg | O_NONBLOCK ) )
   perror( "fcntl" ), exit( EXIT_FAILURE );
...
while( 1 ) {
   int n = read( fdi, buf, buflen );
   if( n > 0 ) {
                                         // считаны данные ... обработка
   }
   else if( -1 == n ) {
      if( EAGAIN == errno ) {            // данные не готовы
         printf( "not ready!\n" );
         usleep( 300 );
         continue;
      }
      else perror( "\nread pipe" ), exit( EXIT_FAILURE );
   }
}
```

Целая подборка примеров, относящихся к неблокирующему вводу-выводу, применительно к сетевым сокетам, заимствованных из [3] (потребовавших минимальных изменений), находится в архиве `ufd.tgz` каталог `nonblock`.

Мультиплексирование ввода-вывода

Один из самых старых API:

```
int select( int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout );
```

И более поздний эквивалент:

```
int  pselect( int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
              const struct timespec *timeout, sigset_t *sigmask );
```

Различия:

- `select()` использует тайм-аут в виде `struct timeval` (с секундами и микросекундами), а `pselect()` использует `struct timespec` (с секундами и наносекундами);

- `select()` может обновить параметр `timeout`, чтобы сообщить, сколько времени осталось. Функция `pselect()` не изменяет этот параметр ;

- `select()` не содержит параметра `sigmask`, и ведет себя как `pselect()` с параметром `sigmask`, равным `NULL`. Если этот параметр `pselect()` не равен `NULL`, то `pselect()` сначала замещает текущую маску сигналов на ту, на которую указывает `sigmask`, затем выполняет `select()`, и восстанавливает исходную маску сигналов.

Параметр тайм-аута может задаваться несколькими способами:

- NULL, что означает ожидать вечно;

- ожидать инициированное структурой значение времени;

- не ожидать вообще (программный опрос, pooling), когда структура инициализируется значением {0, 0}.

Функции возвращают значение больше нуля — число готовых к операции дескрипторов, ноль — в случае истечения тайм-аута, и отрицательное значение при ошибке.

Вводится понятие набора дескрипторов, и макросы для работы с набором дескрипторов:

```
FD_CLR( int fd, fd_set *set );
FD_ISSET( int fd, fd_set *set );
FD_SET( int fd, fd_set *set );
FD_ZERO( fd_set *set );
```

С готовностью дескрипторов чтения и записи `readfds, writefds` — относительно ясно интуитивно. Очень важно, что вариантом срабатывания исключительной ситуации `exceptfds` на дескрипторе сетевых сокетов — является получение внеполосовых данных TCP, что очень широко используется в реализациях (конечных автоматов) сетевых протоколов (например SIP, VoIP сигнализаций PRI, SS7 — на линиях E1/T1, ...).

Примечание: Большинство UNIX систем имеют определение численной константы `FD_SETSIZE`, но её численное значение сильно зависит от констант периода компиляции совместимости с стандартами (такими, например, как `__USE_XOPEN2K`, ...).

Ещё один вариант мультиплексирования ввода-вывода вывода — функция `poll()`. Представление набора дескрипторов заменено на массив структур вида:

```
struct pollfd {
   int fd;           /* файловый дескриптор */
   short events;     /* запрошенные события */
   short revents;    /* возвращенные события */
};
```

- где: `fd` — открытый файловый дескриптор, `events` — набор битовых флагов запрошенных событий для этого дескриптора, `revents` — набор битовых флагов возвращенные событий для этого дескриптора (из числа запрошенных, или `POLLERR, POLLHUP, POLLNVAL`). Часть возможных битов, описаны в `<sys/poll.h>`:

```
#define POLLIN      0x0001    /* Можно читать данные */
#define POLLPRI     0x0002    /* Есть срочные данные */
#define POLLOUT     0x0004    /* Запись не будет блокирована */
#define POLLERR     0x0008    /* Произошла ошибка */
#define POLLHUP     0x0010    /* Разрыв соединения */
#define POLLNVAL    0x0020    /* Неверный запрос: fd не открыт */
```

Ещё некоторая часть описаны в `<asm/poll.h>`: `POLLRDNORM, POLLRDBAND, POLLWRNORM, POLLWRBAND` и `POLLMSG`.

Сам вызов оперирует с массивом таких структур, по одному элементу на каждый интересующий дескриптор:

```
#include <sys/poll.h>
int poll( struct pollfd *ufds, unsigned int nfds, int timeout );
```

- где: `ufds` - сам массив структур, `nfds` - его размерность, `timeout` - тайм-аут в миллисекундах (ожидание при положительном значении, немедленный возврат при нулевом, бесконечное ожидание при значении, заданном специальной константой `INFTIM`, которая определена просто как отрицательное значение).

Пример того, как используются (и работают) вызовы `select()` и `poll()` - позаимствованы из [3] (архив `ufd.tgz`), оригиналы кодов У. Стивенса несколько изменены (оригиналы относятся к 1998 г. и проверялись на совершенно других UNIX того периода). Примеры достаточно объёмные (это полные версии программ TCP клиентов и серверов), поэтому ниже показаны только фрагменты примеров, непосредственно относящиеся к вызовам `select()` и `poll()`, а также примеры того, что реально эти примеры выполняются и как это происходит (вызовы функций в коде показаны как у У. Стивенса — с большой буквы, вызов этот — это полный аналог соответсвующего вызова API, но обрамлённый выводом сообщения о роде ошибки, если она возникнет):

tcpservselect01.c (TCP ретранслирующий сервер на `select()`):

```
...
   int                              nready, client[ FD_SETSIZE ];
   fd_set                           rset, allset;
   socklen_t                        clilen;
   struct sockaddr_in      cliaddr, servaddr;
   ...
   listenfd = Socket( AF_INET, SOCK_STREAM, 0 );
   bzero( &servaddr, sizeof(servaddr) );
   servaddr.sin_family      = AF_INET;
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   servaddr.sin_port        = htons(SERV_PORT);
   Bind( listenfd, (SA*)&servaddr, sizeof(servaddr) );
   Listen( listenfd, LISTENQ );
   maxfd = listenfd;                          /* initialize */
   maxi = -1;                                 /* index into client[] array */
   for( i = 0; i < FD_SETSIZE; i++ )
      client[i] = -1;                         /* -1 indicates available entry */
      FD_ZERO( &allset );
      FD_SET( listenfd, &allset );
      for ( ; ; ) {
         rset = allset;                       /* structure assignment */
         nready = Select( maxfd + 1, &rset, NULL, NULL, NULL );
         if( FD_ISSET( listenfd, &rset ) ) {    /* new client connection */
         connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
...
```

tcpservpoll01.c (TCP ретранслирующий сервер на `poll()`):

```
...
   struct pollfd           client[ OPEN_MAX ];
   struct sockaddr_in      cliaddr, servaddr;
...
   listenfd = Socket( AF_INET, SOCK_STREAM, 0 );
   bzero( &servaddr, sizeof(servaddr) );
   servaddr.sin_family      = AF_INET;
   servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
   servaddr.sin_port        = htons( SERV_PORT );
   Bind( listenfd, (SA*)&servaddr, sizeof(servaddr) );
   Listen( listenfd, LISTENQ );
   client[0].fd = listenfd;
   client[0].events = POLLRDNORM;
   for( i = 1; i < OPEN_MAX; i++ )
```

```
      client[i].fd = -1;                           /* -1 indicates available entry */
      maxi = 0;                                    /* max index into client[] array */
      for ( ; ; ) {
         nready = Poll( client, maxi + 1, INFTIM );
         if( client[0].revents & POLLRDNORM ) {   /* new client connection */
            for( i = 1; i < OPEN_MAX; i++ )
               if( client[i].fd < 0 ) {
                  client[i].fd = connfd;  /* save descriptor */
                  break;
               }
...
               client[i].events = POLLRDNORM;
               if( i > maxi )
                  maxi = i;
...
```

Как выполнять эти примеры и на что обратить внимание? Запускаем выбранный нами сервер (позже мы остановим его по Ctrl+C), все сервера этого архива прослушивают фиксированный порт 9877, и являются для клиента ретрансляторами данных, получаемых на этот порт:

```
$ ./tcpservselect01
...
^C
```

или

```
$ ./tcpservpoll01
...
^C
```

В том, что сервер прослушивает порт и готов к работе, убеждаемся, например, так:

```
$ netstat -a | grep :9877
tcp        0      0 *:9877                      *:*                           LISTEN
```

К серверу подключаемся клиентом (из того же архива примеров), и вводим строки, которые будут передаваться на сервер и ретранслироваться обратно:

```
$ ./tcpcli01 127.0.0.1
1 строка
1 строка
2 строка
2 строка
последняя
последняя
^C
```

Указание IP адреса сервера (не имени!) в качестве параметра запуска клиента — обязательно. Клиентов может быть много — сервера параллельные. Во время выполнения клиента можно увидеть состояние сокетов — клиентского и серверных, прослушивающего и присоединённого (клиент не закрывает соединение после обслуживания каждого запроса, как, например, сервер HTTP):

```
$ netstat -a | grep :9877
tcp        0      0 *:9877                      *:*                           LISTEN
tcp        0      0 localhost:46783             localhost:9877                ESTABLISHED
tcp        0      0 localhost:9877              localhost:46783               ESTABLISHED
```

Ввод-вывод управляемый сигналом

В этом случае на сетевом сокете включается режим управляемого сигналом ввода-вывода, и устанавливается обработчик сигнала при помощи `sigaction()`. Когда UDP дейтаграмма готова для чтения, генерируется сигнал `SIGIO`. Обработать данные можно в обработчике сигнала вызовом `recvfrom()`. Пример того, как это работает, заимствован из [3], и находится в архиве `ufd.tgz` каталог `sigio`, он слишком громоздкий для

детального обсуждения, но может быть изучен и в коде и в работе. Краткая сводка о запуске примера:

Запуск ретранслирующего сервера UDP (в конце выполнения останавливаем его по Ctrl+C):

```
$ ./udpserv01
^C
```

Убедиться, что сервер готов и прослушивает порт, можно так:

```
$ netstat -a | grep :9877
udp        0      0 *:9877                      *:*
```

Запуск клиента:

```
$ ./udpcli01 127.0.0.1
qweqert
qweqert
134534256
134534256
^C
```

Примечание: Особо интересен запуск (например из скрипта) нескольких одновременно (6) клиентов, которые плотным потоком шлют серверу на ретрансляцию большое число строк (у У. Стивенса — 3645 строк). После этого серверу можно послать сигнал `SIGHUP`, по которому он выведет гистограмму, которая складывалась по числу одновременно читаемых дейтаграмм:

```
$ ps -A | grep udp
 2692 pts/12   00:00:00 udpserv01
$ kill -HUP 2692
$ ./udpserv01
cntread[0] = 0
cntread[1] = 8
cntread[2] = 0
cntread[3] = 0
cntread[4] = 0
cntread[5] = 0
cntread[6] = 0
cntread[7] = 0
cntread[8] = 0
^C
```

Асинхронный ввод-вывод

Асинхронный ввод-вывод добавлен только в редакции стандарта POSIX.1g (1993г., одно из расширений реального времени). В вызове `aio_read()` даётся указание ядру начать операцию ввод-вывода, и указывается, каким сигналом уведомить процесс о завершении операции (включая копирование данных в пользовательский буфер). Вызывающий процесс не блокируется. Результат операции (например, полученная UDP дейтаграмма) может быть обработан, например, в обработчике сигнала. Разница с предыдущей моделью, управляемой сигналом, состоит в том, что в той модели сигнал уведомлял о возможности начала операции (вызове операции чтения), а в асинхронной модели сигнал уведомляет уже о завершении операции чтения в буфер пользователя.

Всё, что относится к асинхронному вводу-выводу в Linux описано в <aio.h>. Управляющий блок асинхронного ввода-вывода — видны все поля, которые обсуждались выше:

```
struct aiocb {                 /* Asynchronous I/O control block.  */
   int aio_fildes;             /* File desriptor.  */
   int aio_lio_opcode;         /* Operation to be performed.  */
   int aio_reqprio;            /* Request priority offset.  */
   volatile void *aio_buf;     /* Location of buffer.  */
   size_t aio_nbytes;          /* Length of transfer.  */
```

```
   struct sigevent aio_sigevent; /* Signal number and value.  */
...
}
```

Того же назначения блок для 64-битных операций:

```
struct aiocb64 {
...
}
```

И некоторые операции (в качестве примера):

```
int aio_read( struct aiocb *__aiocbp );
int aio_write( struct aiocb *__aiocbp );
```

Инициализация выполнения целой цепочки асинхронных операций (длиной `__nent`) :

```
int lio_listio( int __mode,
                struct aiocb* const list[ __restrict_arr ],
                int __nent, struct sigevent *__restrict __sig ) ;
```

Как и для потоков `pthread_t`, асинхронные операции значительно легче породить, чем позже остановить... для чего также потребовался отдельный API:

```
int aio_cancel( int __fildes, struct aiocb *__aiocbp );
```

Можно предположить, что каждая асинхронная операция выполняется как отдельный поток, у которого не циклическая функция потока.

Терминал, режим ввода: канонический и неканонический

Частным случаем блокирующего-неблокирующего выводя является ввод с терминала — часто задаваемый вопрос: как реализовать неблокирующий посимвольный ввод с терминала/консоли (такой режим часто используется, например, визуальными редакторами)? Какие вызовы API для этого использовать? Ответ состоит в том, что для неблокирующего ввода не существует какого-то специального набора вызовов POSIX, а используется соответствующий набор параметров терминала, и, в частности, канонический или неканонический режим ввода. Текущие установленные параметры терминала можно посмотреть (наиболее интересующие нас параметры в контексте данного рассмотрения: `icanon, echo, min, time`):

```
$ stty -a < /dev/tty
speed 38400 baud; rows 33; columns 93; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = M-^?;
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
```

Примечание: Вся терминальная система (и команда `stty`) реализована в те давние времена, когда в большинстве случаев подключение терминала производилось через последовательные линии RS-232, поэтому очень много параметров ориентированы на параметры такой линии. Но, если возникает необходимость работать с RS-232 (для связи с устройствами), то оказывается полезной и ещё одна команда, вот как она возвращает, например, диагностику:

```
$ sudo setserial -bg /dev/ttyS*
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

Режим обмена (то, что мы видели по команде `stty`) с терминалом описывается (`<bits/termios.h>`) программной структурой:

```
#define NCCS 32
struct termios {
   tcflag_t c_iflag;           /* input mode flags */
   tcflag_t c_oflag;           /* output mode flags */
   tcflag_t c_cflag;           /* control mode flags */
```

```
   tcflag_t c_lflag;           /* local mode flags */
   cc_t c_line;                /* line discipline */
   cc_t c_cc[NCCS];            /* control characters */
   speed_t c_ispeed;           /* input speed */
   speed_t c_ospeed;           /* output speed */
};
```

Основные функции (`<termios.h>`) работы с режимами терминала:

```
int tcgetattr( int fd, struct termios *termios );
int tcsetattr( int fd, int optional_actions, const struct termios *termios );
```

- где `optional_actions` указывает как поступать с вводом и выводом, уже поставленным в очередь. Это может быть (`<bits/termios.h>`) одно из следующих значений:

```
/* tcsetattr uses these */
#define TCSANOW         0
#define TCSADRAIN       1
#define TCSAFLUSH       2
```

`TCSANOW` - делать изменения немедленно; `TCSADRAIN` - делать изменения после ожидания, пока весь поставленный в очередь вывод не выведен (обычно используется при изменении параметров, которые воздействуют на вывод); `TCSAFLUSH` - подобен `TCSADRAIN`, но отбрасывает любой поставленный в очередь ввод.

Этой информации достаточно, чтобы рассмотреть следующий пример (архив `terminal.tgz`): прямое управление курсором экрана терминала (нажатием клавиш 'd', 'u', 'l', 'r' — вверх, вниз, влево, вправо, соответственно, 'q' — выход из программы):

move.c :

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

int main ( int argc, char **argv ) {
   struct termios savetty, tty;
   char ch;
   int x, y;
   printf( "Enter start position (x y): " );
   scanf( "%d %d", &x, &y );
   if( !isatty( 0 ) ) {
      fprintf( stderr, "stdin not terminal\n" );
      exit( EXIT_FAILURE );
   };
   tcgetattr( 0, &tty );             // получили состояние терминала
   savetty = tty;
   tty.c_lflag &= ~( ICANON | ECHO | ISIG );
   tty.c_cc[ VMIN ] = 1;
   tcsetattr( 0, TCSAFLUSH, &tty ); // изменили состояние терминала
   printf( "%c[2J", 27 );           // очистили экран
   fflush( stdout );
   printf( "%c[%d;%dH", 27, y, x ); // установили курсор в позицию
   fflush( stdout );
   for( ; ; ) {
      read( 0, &ch, 1 );
      if( ch == 'q' ) break;
      switch( ch ) {
      case 'u':
         printf( "%c[1A", 27 );
```

```
         break;
      case 'd':
         printf( "%c[1B", 27 );
         break;
      case 'r':
         printf( "%c[1C", 27 );
         break;
      case 'l':
         printf( "%c[1D", 27 );
         break;
      };
      fflush( stdout );
   };
   tcsetattr( 0, TCSAFLUSH, &savetty ); // восстановили состояние терминала
   printf( "\n" );
   exit( EXIT_SUCCESS );
}
```

Примечание: Обязательная часть подобных программ (не показанная в примере, чтобы его не усложнять) это перехват сигналов завершения (`SIGINT`, `SIGTERM`) для восстановления перед завершением программы канонического режима ввода; в противном случае терминал будет «испорчен» для дальнейшего использования в качестве терминала. Для восстановления режима с успехом может быть использован вызов `atexit()`, как это сделано в примере (в том же архиве):

ncan.c :

```
...
struct termios saved_attributes;
void reset_input_mode( void ) {
   tcsetattr( STDIN_FILENO, TCSANOW, &saved_attributes);
}
void set_input_mode( void ) {
   struct termios tattr;
   ...
   tcgetattr( STDIN_FILENO, &saved_attributes );
   atexit( reset_input_mode );
   ...
   tcsetattr( STDIN_FILENO, TCSAFLUSH, &tattr );
}
...
```

Приложения

Приложение А : Восстановление пароля root

Все описываемые далее способы предназначены для восстановления забытого пароля `root` в установленной вами системе. Но они с равным успехом могут быть использованы и для взлома (смены) пароля `root`. Я достаточно колебался в том, помещать ли такую информацию, так как она небезопасна. Но, в конечном итоге, раз такие возможности существуют, то они должны быть названы: «кто предупреждён, тот вооружён».

С другой стороны, любой из описанных способов — это достаточно нелегальный способ входа в систему, и он может быть заблокирован в любом дистрибутиве, начиная с какой-то версии: используйте эти способы только в рассчёте на метод проб и ошибок... Ниже схематично описаны две группы действий для достижения цели, каждый из них может иметь вариации (в зависимости от вида дистрибутива), которые описаны в источниках, приведенных в конце текста.

Использование мультизагрузчика GRUB

Общий механизм восстановления заключается в осуществлении загрузки системы в однопользовательском (восстановительном) режиме и редактировании (смене) пароля. Этот доступ можно получить отредактировав конфигурацию менеджера загрузки ОС:

- В окне загрузчика GRUB (меню) выделите строку с нужной версией Linux, для которой вам нужно сбросить пароль; нажмите '`e`' для редактирования меню. Выберите строку ядра. Добавьте '`single`' в конец строки. Нажмите '`b`' для загрузки. Если система продолжает запрашивать пароль `root`, добавьте в конец строки `'rw init=/bin/bash'`. Снова нажмите '`b`' для загрузки.
- Вариант: таким же редактированием допишите в такую же строку, начинающуюся с '`kernel`' просто односимвольное слово '`1`' (это однопользовательский режим восстановления - Single Mode с правами `root`).

В любом из вариантов далее жмите '`b`' для загрузки. После загрузки вы попадаете в текстовую консоль, где используете команду `passwd` для изменения текущего пароля на новый.

Если по какой-то причине этого сделать нельзя (установлен пароль на изменение запуска, используется «самописный» загрузчик и другое), то следует использовать LiveCD любого доступного дистрибутива Linux.

Использование загрузочного Live CD

Здесь нам нужно знать имя раздела диска, в который установлен Linux, здесь нам поможет команда без параметров:

```
$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
...
```

Загрузите любой Linux дистрибутив с Live CD (вид дистрибутива произвольный, может не совпадать с установленным Linux). Здесь мы загружаемся как root. Уточняем (подтверждаем) информацию о инсталляции Linux на диск:

```
$ sudo fdisk -l
...
Устр-во Загр     Начало       Конец       Блоки   Id  Система
/dev/sda1   *           1        3494    28056576   83  Linux
/dev/sda2            3494        3650     1254401    5  Расширенный
/dev/sda5            3494        3650     1254400   82  Linux своп / Solaris
```

В нашем случае интересующий раздел — это: `/dev/sda1`.

Монтируем корневой раздел диска, в качестве точки монтирования используется директория `/mnt`:

```
$ sudo mount /dev/sda1 /mnt
```

Далее добавляем `root`-окружение системе LiveCD:

```
$ sudo chroot /mnt
```

Меняем стандартной командой пароль `root`:

```
$ sudo passwd root
```

Соображения безопасности

Показанные способы восстановления пароля `root` дают ещё один лишний раз повод задуматься о безопасности вашей системы, если к ней физически (к терминалу, к CD приводу) имеют доступ другие лица.

От взлома системы способами всех родов, основанных на редактировании меню GRUB, может обезопасить установка пароля доступа к редактированию конфигурации менеджера загрузки системы. Все нюансы установки паролей на GRUB, в том числе, и на редактирование меню GRUB, исчерпывающе описаны в [24].

В отношении использования LiveCD: многие из обсуждающих не считают это уязвимостью. Имея физический доступ, можно, в любом случае, запуститься с LiveCD, и сменить права к любому файлу. Здесь нужно ограничивать физический доступ к компьютеру. Намного важнее уязвимости, которые можно использовать удалённо, а значит на них надо обращать особое внимание.

В конечном счёте, у вас всегда остаётся возможность установки шифрования на раздел диска, причём она присутствует прямо «из коробки дистрибутива».

Источники использованной информации

Я воздержался от более привычного и академического названия для этого раздела - «Библиография», исходя из нескольких соображений:

– помимо публикаций (книг, статей) здесь указываются электронные публикации в Интернет, по которым, обычно, доступно гораздо меньше информации;

– указанные ниже позиции никак не упорядочены, как это принято в настоящей библиографии; это связано не только с тем, что мне просто лень это делать, но ещё и с тем, что я просто не представляю как упорядочить смесь традиционных бумажных источников с электронными публикациями, когда всё это представлено единым списком.

Итак, вот этот единый список:

[1]. А. Робачевский «Операционная система UNIX», Спб.: BHV-СПб, изд. 2, 2005, ISBN 5-94157-538-6, стр. 656.

[2]. У. Ричард Стивенс, Стивен А. Раго, «UNIX. Профессиональное программирование», второе издание, СПб.: «Символ-Плюс», 2007, ISBN 5-93286-089-8, стр. 1040. Полный архив примеров кодов к этой книге может быть взят здесь: http://www.kohala.com/start/apue.linux.tar.Z

[3]. У. Р. Стивенс, «UNIX: Разработка сетевых приложений», СПб.: «Питер», 2003, ISBN 5-318-00535-7, стр. 1088. Полный архив примеров кодов к этой книге может быть взят здесь: http://www.kohala.com/start/unp.tar.Z

[4]. «Искусство программирования на языке сценариев командной оболочки» («Advanced Bash-Scripting Guide»), автор: Mendel Cooper, перевод: Андрей Киселёв.

http://www.opennet.ru/docs/RUS/bash_scripting_guide/

[5]. «GNU Make. Программа управления компиляцией. GNU `make` Версия 3.79. Апрель 2000», авторы: Richard M. Stallman и Roland McGrath, перевод: Владимир Игнатов, 2000.

http://linux.yaroslavl.ru/docs/prog/gnu_make_3-79_russian_manual.html

[6] - «Отладчик GNU уровня исходного кода. Восьмая Редакция, для GDB версии 5.0. Март 2000», авторы: Ричард Столмен, Роланд Пеш, Стан Шебс и др.».

http://linux.yaroslavl.ru/docs/altlinux/doc-gnu/gdb/gdb.html

[7] - «Autoconf. Создание скриптов для автоматической конфигурации, Редакция 2.13», авторы: David MacKenzie и Ben Elliston.

http://www.linux.org.ru/books/GNU/autoconf/autoconf-ru_toc.html

[8] - «GNU Automake, Для версии 1.4», авторы: David MacKenzie и Tom Tromey

http://public.ttknn.net/mirrors/www.linux.org.ru/books/GNU/automake/automake-ru_toc.html

[9] - URL для получения свежих копий конфигурационных скриптов `config.guess` и `config.sub` :

http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;hb=HEAD

http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.sub;hb=HEAD

[10] А. Гриффитс, «GCC. Полное руководство. Platinum Edition», М.: «ДиаСофт», 2004, ISBN 966-7992-33-0, стр. 624.

[11]. «Linux From Scratch, Версия 4.0», автор: Gerard Beekmans © 1999-2002, перевод: Денис Каледин, Ник Фролов, Алекс Казанков.

http://www.linux.org.ru/books/Distro/lfsbook/index.html

[12]. Олег Цилюрик, Егор Горошко, «QNX/UNIX: анатомия параллелизма», СПб.: «Символ-Плюс», 2005, ISBN 5-93286-088-X, стр. 288. Книга по многим URL в Интернет представлена для скачивания, например, здесь: http://bookfi.org/?q=Цилюрик&ft=on#s

[13]. Проект Wine - исполняющая система Windows-приложений:

http://www.winehq.org/

«Руководство пользователя Wine» :

http://docstore.mik.ua/manuals/ru/wine_guide/wine-ug-1.html

[14]. Арнольд Роббинс, «Linux: программирование в примерах», 3-е издание, М.: «Кудиц-Пресс», 2008, ISBN 978-5-91136-056-6 , стр. 656.

[15]. Сандра Лузмор (Sandra Loosemore), Ричард Сталлман (Richard M. Stallman), Роланд Макграх (Roland MacGrath), Андрей Орам (Andrew Oram), «Библиотека языка C GNU glibc. Справочное руководство по функциям, макроопределениям и заголовочным файлам библиотеки glibc.»:

http://docstore.mik.ua/manuals/ru/glibc/glibc.html#toc12

[16]. W. Richard Stevens' Home Page (ресурс полного собрания книг и публикаций У. Р. Стивенса):

http://www.kohala.com/start/

[17]. У. Р. Стивенс, «UNIX: взаимодействие процессов», СПб.: «Питер», 2003,
ISBN: 5-318-00534-9, стр. 576.

[18]. Маттиас Калле Далхаймер, Мэтт Уэлш, «Запускаем Linux, 5-е издание», СПб.: «Символ-Плюс», 2008, ISBN: 5-93286-100-2, стр. 992.

[19]. Григорий Строкин, «BASH конспект»: http://www.ods.com.ua/koi/unix/bash-conspect.html

[20]. OpenXS Russian Man Pages (документация Solaris 8). bash - командный интерпретатор GNU Bourne-Again Shell : http://ln.com.ua/~openxs/projects/man/solaris8/bash.html

[21]. Machtelt Garrels (9.02.2010 Revision 1.12), перевод: Н. Ромоданов (февраль-март 2011 г.), «Руководство по Bash для начинающих» :

http://rus-linux.net/nlib.php?name=/MyLDP/BOOKS/Bash-Guide-1.12-ru/bash-guide-index.html

[22]. «Восстановление пароля для root или угроза безопасности из коробки в Linux» :

http://itshaman.ru/articles/12/passwd-root-linux

Статья опубликована17.05.2009, Автор статьи: Mut@NT

[23]. «Как сбросить пароль в Linux» : http://habrahabr.ru/blogs/linux/54103/

Перевод от 11 марта 2009, Оригинал: http://www.makeuseof.com/tag/how-to-reset-any-linux-password/

Автор: Varun Kashyap, Индия.

[24]. «Установка и настройка пароля на менеджер ОС GRUB» :

http://itshaman.ru/articles/14/password-grub

[25]. «Linux Network Configuration» :

http://www.yolinux.com/TUTORIALS/LinuxTutorialNetworking.html#CONFIGFILES

[26]. Костромин В. А., «Свободная система для свободных людей (обзор истории операционной системы Linux)», март 2005 г.:

http://rus-linux.net/kos.php?name=/papers/history/lh-00.html#toc

[27]. Bert Hubert, Thomas Graf, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spaans, Pedro Larroy, «Linux Advanced Routing & Traffic Control HOWTO», перевод Андрей Киселёв, Иван Песин:

http://www.opennet.ru/docs/RUS/LARTC/index.html

[28]. «The Open Group Base Specifications Issue 7 IEEE Std 1003.1™- 2008» - стандарты POSIX:

http://pubs.opengroup.org/onlinepubs/9699919799/