Лабораторная работа № 1

Знакомство с операционными системами семейства *nix на примере ОС ALT Linux Server.

Командный интерпретатор и основы программирования на shell

Основы регулярных выражений

Copyright © 2008,2010 Nikolay A. Fetisov
Copyright © 2011,2012,2013,2014,2015,2017,2019,2020 Fedor A. Fetisov, Nikolay A. Fetisov
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available as
http://www.gnu.org/licenses/fdl.html

Copyright © Николай Фетисов, 2008,2010.
Copyright © Фёдор Фетисов, Николай Фетисов, 2011,2012,2013, 2014,2015,2017,2019,2020.
Настоящее пособие включает в себя документы, распространяющиеся на условиях GNU Free Documentation License, версия 1.1.
Каждый имеет право воспроизводить, распространять и/или вносить изменения в настоящий Документ в соответствии с условиями GNU Free Documentation License, Версией 1.2 или любой более поздней версией, опубликованной Free Software Foundation;
Данный Документ не содержит Неизменяемых разделов; Данный Документ не содержит текста, помещаемого на первой или последней страницах обложки.
Текст лицензии GNU FDL доступен по адресу:
http://www.gnu.org/licenses/fdl.html

Теоретические сведения.

*Основные понятия операционных систем семейства *nix.*

Первая система UNIX была разработана в 1969 - 1970 годах в подразделении Bell Labs компании AT&T. В 1973 году система была почти полностью переписана на также разработанном в Bell Labs языке высокого уровня C. Это позволило легко переносить UNIX на вычислительные системы различных архитектур и способствовало широкому её распространению. На настоящий момент в мире создано и используется большое число различных UNIX-систем. С юридической точки зрения только часть из них имеет право называться «UNIX», остальные же, хотя и используют сходные концепции и технологии, объединяются термином «UNIX-подобные» (*англ.* Unix-like). Для краткости всё семейство операционных систем класса Unix принято обозначать как *nix.

Некоторые отличительные признаки UNIX-систем включают в себя:

- использование простых текстовых файлов для настройки и управления системой;
- широкое применение утилит, запускаемых в командной строке;
- взаимодействие с пользователем посредством виртуального устройства — терминала;
- представление физических и виртуальных устройств и некоторых средств межпроцессового взаимодействия как файлов;
- использование конвейеров из нескольких программ, каждая из которых выполняет одну задачу.

Технически операционные системы *nix состоят из ядра системы и различных утилит и программ. Ядро обеспечивает общий интерфейс к оборудованию, управление выполнением программ, разделение между ними аппаратных ресурсов компьютерной системы. Основной концепцией, заложенной в архитектуру *nix, является представление различных аппаратных устройств как файлов и предоставление программам возможности работать с устройствами как с файлами.

Одной из ключевых особенностей операционных систем *nix является наличие большого количества разнообразных программ-утилит. Такие программы, запускаемые в командной строке, предназначены для выполнения определённого элементарного действия в системе — например, вывода текстового файла на экран, вывода содержимого каталога, записи текста в файл. Операционные системы *nix предоставляют удобные и гибкие механизмы объединения работы таких отдельных простых программ для выполнения конкретных задач пользователей. В данной лабораторной работе проводится рассмотрение и изучение этих механизмов.

В число основных задач современных вычислительных систем входит обработка текстовой информации, как в виде простого текста, так и в виде текста с форматированием. Хотя форматированный текст на персональных компьютерах обычно представляется в формате двоичных файлов, в последнее время намечается тенденция отказа от таких (часто закрытых) двоичных форматов и перехода к использованию основанных на обычном тексте языков разметки документов. Операционные системы *nix изначально разрабатывались для обработки текстовой информации, и обладают большим набором мощных и универсальных инструментов работы с текстами. Одним из таких инструментов являются регулярные выражения, примеры применения которых также рассматриваются в данной работе.

Структура файловой системы.

Для хранения данных в настоящий момент используются различные устройства — накопители на жестких и гибких магнитных дисках, накопители на микросхемах Flash-памяти, накопители на оптических носителях форматов CD, DVD, Blu-ray, и т.п. С точки зрения операционных систем, всё это — устройства с блочным вводом-выводом, которые далее мы будем обобщённо называть дисками.

Как правило, доступное для хранения информации место на дисках разбивается на разделы. В рамках каждого из разделов создаётся файловая система, позволяющая управлять размещением на дисках отдельных файлов. Это требование не является жёстким, возможно создание файловых систем непосредственно на дисках, без разбиения их на отдельные разделы. Кроме того, возможно хранение информации на дисках и без создания файловых систем — например, крупные системы управления базами данных (СУБД) могут сами управлять размещением баз данных на дисках, без использования промежуточных звеньев в виде файлов и файловых систем.

Задачей файловой системы является обеспечение эффективного выделения пространства для хранения данных, ведение списка файлов и каталогов, эффективный поиск файлов в каталогах и т.д. Существует большое количество файловых систем, обладающих теми или иными характеристиками. Выбор файловой системы для носителя данных зависит от конкретного случая. Операционные системы могут одновременно управлять несколькими устройствами хранения данных с разными файловыми системами на них.

К основным поддерживаемым в Linux файловым системам относятся:

- *Ext2* — файловая система, изначально разработанная для систем Linux. Сравнительно простая в реализации. Сейчас используется в основном во встраиваемых системах, например, в маршрутизаторах, сотовых телефонах, в качестве корневой файловой системы сетевых накопителей бытового уровня и т.п.
- *Ext3* — дальнейшее развитие *Ext2*, файловая система с поддержкой журналирования. Совместима с *Ext2*. При хранении большого числа файлов в каталогах использование *Ext3* неэффективно.

- *Ext4* — дальнейшее развитие *Ext3*, файловая система с поддержкой журналирования. Совместима с *Ext2* и *Ext3*. Более эффективна при работе с большим числом файлов в каталогах.
- *XFS* — журналируемая файловая система, разработанная для рабочих станций Silicon Graphics (SGI) с операционной системой IRIX. Изначально спроектирована для работы с мультимедийными файлами большого размера, эффективна при использовании расширенных списков контроля доступа к файлам. При использовании *XFS* крайне желательна надёжно работающая аппаратная платформа и наличие резервирования электропитания оборудования.
- *JFS* (*Journaled File System*) — журналируемая файловая система, изначально разработанная корпорацией IBM для ОС AIX.
- *ReiserFS* — журналируемая система, разработанная Гансом Рейзером (Hans Reiser). Оптимизирована для работы с каталогами, содержащими большое количество файлов, а также для хранения небольших по размеру файлов.
- *Btrfs* — журналируемая файловая система, разрабатываемая как замена файловых систем Ext3/Ext4. Обеспечивает эффективную работу с файлами небольшого размера, каталогами с большим числом файлов, имеет возможность прозрачного сжатия хранящихся данных. Поддерживает создание снимков состояния файловой системы, возможность размещения файловой системы на нескольких физических устройствах, оптимизирована под работу с твердотельными дисками (SSD). В настоящее время считается экспериментальной и для широкого использования не рекомендована.
- *ISOFS* (*iso9660*) — файловая система, разработанная для дисков CD, но достаточно часто встречающаяся и на дисках DVD. Имеет ограничение максимального размера файла в 2 Gb.
- *UDF* — файловая система, обычно используемая для дисков DVD.
- *VFAT* — развитие файловой системы MS DOS с добавленной поддержкой длинных имён файлов. Из достоинств файловой системы — простота её реализации. Используется в основном на съёмных носителях данных типа USB Flash. Для разделов дисков, больших 32 Gb, использование *VFAT* крайне неэффективно.
- *NTFS* — файловая система, используемая в ОС Microsoft Windows NT и более поздних. В отличие от *VFAT*, использует журналирование и имеет систему контроля прав доступа к файлам. Использование данной файловой системы в Linux ограничено из-за отсутствия открытой документации по архитектуре файловой системы и сильной зависимости её реализации от архитектуры ОС MS Windows.

Существуют также и специальные файловые системы, из которых можно отметить:

- *procfs* — файловая система, позволяющая обращаться к ряду струк-

тур данных внутри ядра *nix, как к файлам. В частности, в *procfs* можно посмотреть текущий список выполняющихся процессов, состояние оборудования, настройки и текущее состояние сетевых устройств, и т.п. Программы, предназначенные для вывода подобной информации, получают её из *procfs*. Кроме того, ряд файлов в *procfs* доступны для записи, и с их помощью можно изменить параметры работы ядра *nix.

- *sysfs* — файловая система, работающая со структурами ядра Linux и позволяющая получить данные об оборудовании системы. В частности, с использованием *sysfs* производится конфигурация устройств «горячего» подключения.
- *udevfs* — файловая система, предназначенная для хранения файлов устройств, с поддержкой создания/удаления файлов устройств «горячего» подключения.
- *tmpfs* — файловая система, предназначенная для хранения файлов в виртуальной памяти. Основное назначение системы — размещение временных файлов, которые можно потерять при перезагрузке системы. Использование *tmpfs* на современных компьютерных системах, с большим объемом ОЗУ и достаточно большим размером файлов подкачки, позволяет существенно ускорить, например, выполнение компиляции и сборки сложных программных продуктов. Кроме того, *tmpfs* используется во встраиваемых системах.
- *jffs2* (*Journalling Flash File System version 2*) — файловая система, оптимизированная для работы с программируемой Flash-памятью (ППЗУ) с учётом особенностей износа Flash-памяти при регулярной записи в неё данных. Применяется во встраиваемых системах для хранения настроек.
- *squashfs* — файловая система, обеспечивающая хранение данных и структур каталогов в сжатом состоянии. Предназначена только для чтения данных, широко используется во встраиваемых системах.

Существуют также сетевые файловые системы. Эти файловые системы предназначены для доступа к информации, хранящейся на других системах, через компьютерную сеть. Из сетевых файловых систем можно отметить:

- *CIFS* (*Common Internet File System, старое название SMB, Server Message Block*) — сетевая файловая система, использующаяся в сетях Microsoft Windows.
- *NFS* (*Network File System*) — сетевая файловая система, изначально появившаяся для систем *nix. По сравнению с *CIFS* существенно более простая в реализации, но и с существенно менее гибким управлением доступом к файлам.

Современные файловые системы организуют хранение файлов в иерархической структуре каталогов. Все перечисленные выше файловые

системы поддерживают длинные имена файлов. Как правило, максимальная длина имени файла составляет 255 символов — т.е., при использовании кодировки UTF-8, 127 символов русского алфавита. В именах файлов и каталогов допускаются любые символы, кроме символов `NULL` (символ с кодом ASCII 0) и `/` (слеш). Символ `NULL` используется как ограничитель строки, он следует за последним значащим символом строки переменной длины. Символ `/` (слеш) служит разделителем имён каталогов при указании пути к файлу. Отметим, что в операционных системах семейства Microsoft Windows для этой цели используется символ `\` (обратный слеш).

В имени файла может содержаться расширение — несколько (обычно до четырёх) символов, отделённых от основной части имени файла символом «.» (точка). Обычно через расширение указывается формат файла. К широко используемым расширениям относятся:

- `.jpg`, `.gif`, `.png`, `.tiff` — для графических файлов;
- `.html`, `.htm` — для файлов в формате HTML;
- `.pdf`, `.ps` — для файлов PDF и PostScript;
- `.c`, `.h`, `.cpp` — для исходных текстов и заголовков программ на C и C++;
- `.o` — для скомпилированного объектного кода;
- `.sh` — для скриптов командного интерпретатора;
- `.tar` – для файловых архивов в формате утилиты tar;
- `.gz`, `.bz2`, `.zip`, `.xz` – для файлов, сжатых утилитами `gzip`, `bzip2`, `zip` и `xz` соответственно, и т.п.

Часто в расширении указывается дополнительная информация — например, для библиотек в виде расширения может указываться номер версии библиотеки. Однако для самой операционной системы расширения файлов никакой смысловой нагрузки не несут, никаких ограничений ни на размер расширений, ни на их число не накладывается.

В *nix строчные и заглавные буквы в именах файлов различаются, т.е. файлы `file.txt`, `file.TXT`, `File.txt` — три разных файла, которые могут сосуществовать в одном каталоге. Однако могут быть и исключения — например, для файловой системы *VFAT* — это один и тот же файл, а для *CIFS* поведение зависит от настроек сервера *CIFS*.

В каждой файловой системе имеется каталог верхнего уровня. Существует понятие корневой файловой системы — это каталог верхнего уровня файловой системы системного диска. На системном диске размещаются основные файлы операционной системы и с него выполняется загрузка системы. Корневой каталог имеет путь `/`. Для каждого файла в системе можно указать полный путь — перечисление иерархии всех каталогов от корневого каталога до самого файла.

В отличие от систем семейства Microsoft DOS/Windows, для доступа к файловым системам, расположенным на других дисках, не используются названия этих дисков. Вместо этого файловые системы этих дисков монтируются (mount), или, иными словами, «прикрепляются» к одному из каталогов файловой системы. После монтирования файловая система смонтированного диска становится продолжением общего дерева каталогов в системе. Для прикладных программ не важно, на каком конкретно диске и типе файловой системы находится тот или иной файл — всеми подробностями организации дисков занимается операционная система. Это позволяет скрыть особенности организации дисковой системы от пользователя, легко добавлять и удалять диски из системы, переносить части существующих файловых систем на новые диски без изменения путей к файлам, или, например, разместить часть каталогов файловой системы не на локальных, а на сетевых дисках.

При монтировании можно указывать дополнительные параметры, влияющие на поведение смонтированной файловой системы. В частности, файловые системы могут быть смонтированы в режиме «только для чтения» – в этом случае изменения на них файлов и каталогов будут невозможны. Такой режим монтирования широко используется в операционных системах мобильных устройств, где монтирование в режиме «только для чтения» файловых систем с системными файлами обеспечивает дополнительную защиту от нежелательной модификации служебных данных пользователями. При необходимости обновления системных файлов в этом случае программа-установщик изменяет параметры монтирования файловых систем, делая их доступными для записи, вносит в системные файлы нужные изменения, и возвращает файловые системы обратно в режим доступа только на чтение.

Общая структура каталогов *nix-систем относительно стандартна, современные системы стараются придерживаться рекомендациям *FHS* (*Filesystem Hierarchy Standard*). Рассмотрим общую структуру каталогов *nix-систем на примере ALT Linux Branch 5.1. В корневом каталоге системы располагаются следующие каталоги верхнего уровня:

```
$ ls -1 /
bin
boot
dev
etc
home
lib
lib64
lost+found
media
mnt
opt
proc
root
sbin
srv
```

```
sys
tmp
usr
var
```

- `/bin/` — каталог с основными программами и утилитами. Расположенные в этом каталоге программы жизненно необходимы для функционирования операционной системы и её нормальной загрузки;
- `/boot/` — каталог с файлами ядра операционной системы;
- `/dev/` — каталог с файлами устройств. В старых *nix-системах в каталоге `/dev/` размещались файлы устройств для практически всего поддерживаемого системами оборудования, и размер этого каталога был достаточно большим. В современных дистрибутивах Linux в каталог `/dev/` монтируется файловая система *udevfs*, и в нём присутствуют только файлы реально подключенных и используемых в системе устройств;
- `/etc/` — каталог с файлами конфигурации системы и программ. Практически все настройки системы хранятся в текстовых файлах, которые можно легко просмотреть и изменить обычным текстовым редактором. Как правило, помимо самих настроек в файлах конфигурации размещаются комментарии и описания этих настроек. Обычно комментарии начинаются с символов `#` (октоторп) или `;` (точка с запятой);
  - `/etc/X11/` — каталог с конфигурацией *X Window System, version 11*;
  - `/etc/opt/` — каталог с конфигурацией программ из `/opt/`;
- `/home/` — каталог для домашних каталогов пользователей. Каждому пользователю системы выделяется т.н. домашний каталог — каталог, в котором хранятся личные файлы пользователя, персональные настройки программ и т.п. Например, если в системе есть пользователь `student`, то его домашний каталог будет находиться в `/home/student/` ;
- `/lib/` — основные библиотеки системы — т.е. библиотеки, использующиеся программами из каталогов `/bin/` и `/sbin/`. В каталоге `/lib/modules/` размещаются загружаемые модули ядра операционной системы;
- `/lib64/` — каталог для системных библиотек, с указанием архитектуры системы. В данном случае — это каталог для 64-битных библиотек 64-разрядных процессоров с архитектурой Intel. Такие каталоги не обязательны и в ряде систем могут отсутствовать;
- `/lost+found/` — каталоги с таким названием могут присутствовать в корне разделов файловых систем. В ходе проверок файловых систем на целостность после сбоев (например, после отключения питания работающей системы), в эти каталоги помещаются обнаруженные «потерянные» файлы. В нормально работающих системах должны быть пустыми; для некоторых файловых систем (*Ext2*, *Ext3*, *Ext4*) автоматически созда-

ются при монтировании файловой системы, для других (*XFS*) — при необходимости при проверке файловой системы.

- `/media/` — содержит каталоги, в которые монтируются при подключении съёмные носители данных. Для использования какого-либо диска (включая оптические диски CD/DVD, дискеты, накопители USB Flash) его файловая система должна быть смонтирована в общее дерево. Это может делаться как вручную (в подкаталоги `/mnt/`), так и автоматически, при использовании соответствующих системных утилит. Эти утилиты, при подключении нового диска к системе, создают для него подкаталог в каталоге `/media/` и монтируют в него файловую систему диска. Стоит обратить внимание на то, что вынуть съёмный диск из компьютера можно только после его размонтирования (отключения от общей файловой системы). Для ряда съёмных накопителей (например, дисков CD/DVD) система не позволит их извлечь без размонтирования их файловой системы. Размонтировать файловую систему можно только тогда, когда ни одна из работающих программ не обращается к съёмному диску. Если диск размонтировать не получается, надо найти и завершить такие программы. Если просто вынуть дискету или отключить USB Flash, то можно нарушить работу операционной системы и повредить файловую систему дискеты / накопителя USB Flash.
- `/mnt/` — содержит каталоги, в которые временно и вручную монтируются различные диски.
- `/opt/` — каталог для крупных и независимых программных пакетов. К таким пакетам обычно относятся коммерческие продукты, которые не могут использовать общесистемные настройки и библиотеки. Они размещаются в отдельных подкаталогах внутри каталога `/opt/`, со своими подкаталогами `etc/`, `bin/`, `lib/`, `var/`, `sbin/` и т.п;
- `/proc/` — каталог, в который монтируется файловая система *procfs*. Содержит информацию о состоянии системы;
- `/root/` — домашний каталог суперпользователя. Поскольку каталоги пользователей системы могут быть размещены на отдельных дисках, домашний каталог суперпользователя отделён от них и размещается в корне файловой системы, на системном диске;
- `/sbin/` — содержит основные системные утилиты. В отличие от утилит из `/bin/`, эти утилиты не предназначены для использования обычными пользователями, но также жизненно необходимы для загрузки системы;
- `/srv/` — каталог, предназначенный для сервисов системы. Обычно никак не используется;
- `/sys/` — каталог, в который монтируется файловая система *sysfs*. Содержит информацию об оборудовании системы;
- `/tmp/` — содержит временные файлы. Может быть смонтирован как *tmpfs*, с удалением всего содержимого при перезагрузке машины. В каталоге `/tmp/` может создавать файлы и подкаталоги любой пользователь

системы;

- `/usr/` — предназначен для размещения прикладных программ и их статических данных. Подкаталоги внутри `/usr/` образуют вторичную структуру каталогов, повторяющую структуру корневого каталога, и содержат:
  - `/usr/bin/` — каталог для прикладных программ, доступных пользователям;
  - `/usr/include/` — каталог для файлов заголовков C;
  - `/usr/lib/` — каталог для библиотек, использующихся программами в `/usr/bin/` и `/usr/sbin/`;
  - `/usr/sbin/` — каталог с некритичными системными программами — например, с различными сетевыми серверами;
  - `/usr/share/` — каталог с архитектурно-независимыми данными, в т.ч.:
    - `/usr/share/doc/` — каталог с документаций для пакетов программ;
    - `/usr/share/man/` — каталог со страницами справочных руководств по программам;
    - `/usr/share/info/` — каталог со справочными руководствами в формате info, предназначенных для просмотра одноимённой командой;
  - `/usr/src/` — каталог с исходными текстами программ;
  - `/usr/X11R6/` — структура каталогов третьего уровня для программ *X Window System, Version 11, Release 6* — в т.ч. `/usr/X11R6/bin/`, `/usr/X11R6/lib/`, `/usr/X11R6/man/`, и т.д.;
  - `/usr/local/` — структура каталогов третьего уровня для программ, устанавливаемых из исходных текстов. Содержит подкаталоги `/usr/local/bin/`, `/usr/local/lib/`, `/usr/local/sbin/` и т.д.;
- `/var/` — предназначен для различных изменяемых при работе системы файлов. Внутри `/var/` находятся:
  - `/var/cache/` — файлы с кэшированными данными приложений, например, полученные из сети пакеты программ, отформатированные и готовые для показа пользователю страницы справочных руководств, и т.п.;
  - `/var/lock/` — файлы блокировки, предназначенные для отслеживания использования ресурсов в системе;
  - `/var/log/` — журналы (*логи*). В файлы в этом каталоге выводится информация от работающих в системе неинтерактивных программ;
  - `/var/spool/` — каталог с данными, которые ожидают обработки,

например:

  - `/var/spool/mail/` — входящие почтовые ящики пользователей;
  - `/var/spool/cups/` — очереди документов для печати системы *CUPS* (*Common Unix Printing System*);
- `/var/run/` — информация о запущенных и работающих неинтерактивных программах;
- `/var/tmp/` — различные временные файлы. В отличие от `/tmp/`, содержимое этого каталога должно сохраняться между перезагрузками системы.

Пользователи и процессы в системе.

ОС UNIX изначально разрабатывалась как многопользовательская многозадачная система, и предусматривает одновременную работу многих программ разных пользователей. Каждому пользователю в системе соответствует уникальный числовой идентификатор — *UID* (*User ID*). Хотя для работы самой операционной системы достаточно иметь только числовой идентификатор UID, в целях удобства используются и символьные имена пользователей. Имя пользователя может содержать только латинские строчные буквы, цифры и символ - (дефис). Желательно, чтобы имя пользователя было не длиннее 8 символов, хотя допускаются и более длинные имена. Записи о соответствии символьных имён пользователей их числовым идентификаторам, а также дополнительная информация об учётных записях пользователей хранится в системных информационных базах, простейшим и наиболее распространённым на настольных системах вариантом которых является текстовый файл `/etc/passwd`.

Пользователи объединяются в группы, которые также имеют уникальные числовые идентификаторы — *GID* (*Group ID*), и символьные имена. Записи о группах – соответствие символьных имён числовым идентификаторам, а также списки входящих в группу пользователей, хранятся в своих информационных базах. В простейшем случае настольных систем такая база хранится в файле `/etc/group`. Каждый пользователь должен входить хотя бы в одну группу, которая носит название первичной группы. Также пользователь может входить в одну или несколько других групп, носящих название вторичных. Имя первичной группы пользователя указывается в его записи в файле `/etc/passwd`. Для вторичных групп в `/etc/group` указываются имена пользователей, входящих в них. В системе всегда существует пользователь с *UID*=0 и соответствующая группа с *GID*=0. Этот пользователь является администратором или суперпользователем системы. Традиционно имя суперпользователя — `root`. Принципиальным отличием суперпользователя от остальных пользователей является то, что система не применяет к нему правила контроля доступа, т.е. пользователь с *UID*=0 (`root`) имеет полный и неограниченный доступ ко всех функциям, файлам, устройствами и ресурсам системы.

Выполняющиеся в системе программы носят названия процессов. Каждый процесс имеет уникальный номер — идентификатор процесса (*PID, Process ID*), а также идентификаторы *UID* и *GID*, с правами которых он выполняется. Любой процесс может с помощью системного вызова `fork()` создать новый процесс. Новый процесс наследует от своего родителя значения *UID* и *GID*. Также процессам доступен системный вызов `chuser()`, который меняет *UID* выполняющего процесса. Вызов `chuser()` доступен только процессам с *UID*=0, т.е. запущенный с правами `root` процесс может один раз изменить свои полномочия на полномочия непривилегированного пользователя, а дальше и этот процесс, и все создаваемые им дочерние процессы изменить свои *UID* не могут. Имеется аналогичный системный вызов и для смены *GID*.

Первый процесс, запускаемый ядром операционной системы при загрузке системы, получает *PID*, равный 1, и выполняется с *UID*=0 и *GID*=0. Обычно этой программой является `/sbin/init`, которая, в свою очередь, запускает другие программы согласно настройкам в `/etc` . Процесс `init` постоянно находится в системе, вплоть до завершения работы.

Все процессы, работающие в системе, можно разделить на три группы. Во-первых, это системные процессы, которые, как и `init`, запускаются ядром. Эти процессы отвечают за работу таких подсистем ядра, как кэширование дисков, управление виртуальной памятью и т.п. Эти процессы запускаются и контролируются непосредственно ядром операционной системы, возможности управления ими весьма ограничены.

Вторая группа — это процессы неинтерактивных системных программ — различных сервисов, выполняющихся в системе. В качестве примера можно привести веб-серверы, серверы баз данных, серверы удалённого доступа к системе, и т.п. Непосредственно с пользователем эти программы не взаимодействуют, для работы с ними требуются дополнительные прикладные программы. Такие процессы, как правило, автоматически запускаются системой при её загрузке, и далее постоянно выполняются в фоновом режиме. Но для функционирования системы они не требуются, и имеется возможность управлять их выполнением — останавливать, запускать и т.п.

К третьей группе относятся прикладные процессы — т.е. программы, непосредственно запускаемые пользователем при его работе с системой.

Кроме суперпользователя и обычных пользователей в системе существует набор т.н. псевдопользователей — непривилегированных пользователей, с правами которых работают различные системные программы. Как правило, псевдопользователям не назначен командный интерпретатор, а их домашний каталог — это тот каталог, в который соответствующие программы могут писать свои данные. При этом в файле `/etc/passwd` вместо командного интерпретатора указывается пустое устройство `/dev/null`.

Системные программы обычно запускаются с привилегиями суперпользователя и после инициализации изменяют с помощью системного вызова

chuser() свой идентификатор пользователя на непривилегированный.

В качестве дополнительной меры безопасности существует возможность изменения в настройках запущенных процессов указателя на корневой каталог. В этом случае работающей программе доступно не всё дерево каталогов системы, а только некоторая его часть, обратится к файлам за пределы которой программа не может. Обычно при этом для программы создаётся каталог с именем /var/lib/<имя программы>, с подкаталогами etc/, lib/, tmp/, var/, в которые копируется минимально необходимый для работы данной программы набор конфигурационных файлов и системных библиотек. Смена общего корневого каталога на каталог /var/lib/<имя программы> выполняется системным вызовом chroot().

Системные вызовы chuser() и chroot() доступны только суперпользователю, поэтому после перехода в непривилегированный режим работы процесс не может вернуть себе полномочия root обратно. Все создаваемые им процессы также будут работать с правами псевдопользователя. В случае, если в программе будет обнаружена уязвимость, и злоумышленник получит над ней контроль (т.е. сможет запускать свои программы в системе), он не сможет получить права суперпользователя и выйти за границы *chroot-окружения*.

Как правило, идентификатор (*UID*) псевдопользователя меньше 500, а обычного пользователя — равен или больше. Однако данное разделение чисто условное и соблюдается по договорённости.

Для интерактивной работы пользователей с системой используется понятие терминала — символьного устройства, с которого поступают вводимые пользователем команды и на которое выводится результат их выполнения. Для начала работы с системой пользователь должен зарегистрироваться на одном из поддерживаемых системой терминальных устройств. При локальной работе терминалом являются подсоединённые к системе монитор и клавиатура. В случае удалённых сеансов работы, пользователь должен на своём рабочем месте запустить программу — эмулятор терминала и соединиться с удалённой системой. Разумеется, на удалённой системе при этом должен быть запущен соответствующий сервер, обеспечивающий такие соединения.

В начале работы с системой, система запрашивает имя пользователя и его пароль. При совпадении введённых значений со значениями, хранящимися в учётной записи в /etc/passwd, система разрешает работу пользователя с данным терминалом и запускает на нём командный интерпретатор, позволяя вводить команды.

При этом нет ограничений на число параллельно работающих с системой пользователей — каждый работает независимо в своём терминале. Можно одновременно открыть и несколько терминальных сессий с системой для одного и того же пользователя, и работать одновременно с несколькими терминалами.

Права доступа к файлам.

Поскольку система Linux с самого начала разрабатывалась как многопользовательская, в ней предусмотрен такой механизм, как права доступа к файлам и каталогам. Он позволяет разграничить полномочия пользователей, работающих в системе. В частности, права доступа позволяют отдельным пользователям иметь «личные» файлы и каталоги. Например, если пользователь `student` создал в своём домашнем каталоге файлы, то он является владельцем этих файлов и может определить права доступа к ним для себя и остальных пользователей. Он может, например, полностью закрыть доступ к своим файлам для остальных пользователей, или разрешить им читать свои файлы, запретив изменять и исполнять их.

Правильная настройка прав доступа позволяет повысить надёжность системы, защитив от изменения или удаления важные системные файлы. Наконец, поскольку внешние устройства с точки зрения системы также являются объектами файловой системы, механизм прав доступа применяется и для управления доступом к устройствам.

С точки зрения самой системы работа пользователя в ней — это выполнение программ (процессов) с идентификаторами UID/GID пользователя, которые осуществляют различные действия с файлами и каталогами, и запускают на выполнение другие процессы. Например, одна из таких программ — командная оболочка, которая считывает команды пользователя из командной строки и передаёт их системе на выполнение.

Каждая программа (процесс) выполняется от имени определённого пользователя (т. е. с определёнными идентификаторами UID/GID). Её возможности работы с файлами и каталогами определяются правами доступа, заданными для этого пользователя.

Содержимое файла программа может считывать или записывать, а если в файле хранится другая программа, то её можно запустить на выполнение и создать новый процесс. Из каталога можно считать список содержащихся в нём файлов и каталогов, или внести в этот список изменения — создать новую запись (файл или каталог), переименовать или удалить существующую. Кроме того, в каталог можно перейти — сделать его текущим для данного процесса.

У любого файла в системе есть владелец — один из пользователей. Однако каждый файл одновременно принадлежит и некоторой группе пользователей системы.

Права доступа определяются по отношению к трём типам действий: чтение, запись и исполнение. Эти права доступа могут быть предоставлены трём классам пользователей: владельцу файла (пользователю), группе, которой принадлежит файл, а также всем остальным пользователям, не входящим в эту группу. Право на чтение даёт пользователю возможность читать содержимое файла или, если такой доступ разрешён к каталогам, просматривать содержимое каталога (используя команду `ls`). Право на запись даёт пользователю возможность записывать или изменять файл, а право на

запись для каталога — возможность создавать новые файлы или удалять файлы из этого каталога. Наконец, право на исполнение позволяет пользователю запускать файл как программу или сценарий командной оболочки (разумеется, это действие имеет смысл лишь в том случае, если файл является программой или сценарием). Для каталогов право на исполнение имеет особый смысл — оно позволяет сделать данный каталог текущим, т.е. «перейти» в него, например, командой `cd`.

Чтобы получить информацию о правах доступа, можно использовать команду `ls` с ключом `-l`. При этом будет выведена подробная информация о файлах и каталогах, в которой будут, среди прочего, отражены права доступа. Рассмотрим несколько примеров:

```
$ ls -l ~
итого 8
drwx------ 2 student student 4096 Фев 19 17:30 Documents
-rw-r--r-- 1 student student    0 Фев 20 08:03 file.txt
drwx------ 2 student student 4096 Фев 19 15:59 tmp

$ ls -l /var
итого 72
drwxr-xr-x  2 root root   4096 Апр 19  2007 adm
drwxr-xr-x  4 root root   4096 Фев 15 08:32 cache
drwxr-xr-x  2 root root   4096 Апр 19  2007 db
dr-xr-xr-x  2 root root   4096 Апр 19  2007 empty
drwxr-xr-x 11 root root   4096 Фев  9 15:29 lib
drwxr-xr-x  2 root root   4096 Апр 19  2007 local
drwxr-xr-x  6 root root   4096 Фев 20 07:32 lock
drwxr-xr-x 14 root root   4096 Фев 20 07:32 log
lrwxrwxrwx  1 root root     10 Фев  5 13:22 mail -> spool/mail
drwxr-xr-x  2 root root   4096 Апр 19  2007 nis
drwxr-x---  2 root nobody 4096 Апр 19  2007 nobody
drwxr-xr-x  2 root root   4096 Апр 19  2007 opt
drwxr-xr-x  2 root root   4096 Апр 19  2007 preserve
drwxr-xr-x  5 root root   4096 Апр 19  2007 resolv
drwxr-xr-x  5 root root   4096 Фев 17 03:38 run
drwxr-xr-x  6 root root   4096 Фев 20 07:32 spool
drwxrwxrwt  2 root root   4096 Апр 19  2007 tmp
drwxr-xr-x  3 root root   4096 Фев 15 09:24 www
drwx------  2 root root   4096 Апр 19  2007 yp
$ ls -l /bin/su
-rws--x--- 1 root wheel 23712 Окт 18  2006 /bin/su
```

Для файла `~/file.txt` первое поле в строке (`-rw-r--r--`) отражает права доступа. Третье поле указывает на владельца файла (`student`), четвёртое поле указывает на группу, которая владеет этим файлом (`student`). Последнее поле — это имя файла (`file.txt`). Другие поля описаны в документации к команде `ls`.

Данный файл является собственностью пользователя `student` и группы `student`. Последовательность `-rw-r--r--` показывает права доступа для пользователя — владельца файла, пользователей — членов группы-владельца, а также для всех остальных пользователей.

Первый символ из этого ряда обозначает тип файла. Символ `-` (дефис) означает, что это — обычный файл, который не является каталогом (в этом

случае первым символом было бы d), символьной ссылкой (было бы l) или псевдофайлом устройства (было бы c или b). Следующие три символа (rw-) представляют собой права доступа, предоставленные владельцу student. Символ r — сокращение от read (*англ.* читать), а w — сокращение от write (*англ.* писать). Таким образом, student имеет право на чтение и запись (изменение) файла file.txt.

После символа w мог бы стоять символ x, означающий наличие прав на исполнение (*англ.* execute, исполнять) файла. Однако символ - (дефис), стоящий здесь вместо x, указывает, что student не имеет права на исполнение этого файла. Это разумно, так как файл file.txt не является программой. В то же время, пользователь, зарегистрировавшийся в системе как student, при желании может предоставить себе право на исполнение данного файла, поскольку является его владельцем. Для изменения прав доступа к файлу или каталогу используется команда chmod.

Следующие три символа (r--) отражают права доступа группы к файлу. Группой-владельцем файла в нашем примере является группа student. Поскольку здесь присутствует только символ r, все пользователи из группы student могут читать этот файл, но не могут изменять или исполнять его.

Наконец, последние три символа (это опять r--) показывают права доступа к этому файлу всех других пользователей, помимо собственника файла и пользователей из группы student. Так как здесь указан только символ r, эти пользователи тоже могут лишь читать файл.

Для ~/Documents первое поле содержит drwx------. Это каталог (на что указывает первый символ — буква d), владелец которого (student) может читать содержимое каталога (т. е. получать список содержащихся в нём файлов), писать в каталог (т. е. изменять его содержимое — создавать, удалять и переименовывать файлы) и переходить в него (для каталогов операцией выполнения считается возможность сделать данный каталог текущим). Другие пользователи — как члены группы student, так и все прочие — никаких прав не имеют и ни перейти, ни прочитать содержимое этого каталога, ни, тем более, что-либо в него записать не могут.

Для /var/adm первое поле содержит drwxr-xr-x. Это каталог, владелец которого — root — имеет права rwx — т.е. может читать, писать и переходить в этот каталог. Пользователи из группы root имеют права r-x — т.е. могут читать содержимое каталога и переходить в него. Те же права и у всех остальных пользователей.

Для /var/empty первое поле содержит dr-xr-xr-x. Это каталог, владелец которого (root), группа (root) и все остальные пользователи имеют одинаковые права r-x — т.е. могут читать содержимое каталога и переходить в него, что соответствует названию каталога (*англ.* empty — пустой). Правда, стоит отметить, что root записать что-либо в этот каталог всё-таки может, поскольку на суперпользователя права доступа не распространяются.

Для /var/nobody первое поле содержит drwxr-x---. Это каталог, владелец которого — root — имеет права rwx. Группа — nobody — имеет права r-x, т.е. может переходить в каталог и читать его. Прочие пользователи доступа к каталогу не имеют. Такие права объясняются тем, что nobody — это псевдопользователь, и /var/nobody — его домашний каталог (см. запись в /etc/passwd). Это бесправный пользователь, и даже прав на запись чего-либо в свой домашний каталог у него нет.

Для /var/mail в первом поле стоит lrwxrwxrwx. Первый символ l означает символьную ссылку. Согласно выводу команды ls -l, эта ссылка указывает на /var/spool/mail (путь spool/mail указан относительно каталога, где размещена ссылка — т.е. /var). Права доступа к файлу или каталогу, на который ссылается символьная ссылка, определяются правами на сам файл, а не правами ссылки. Поэтому здесь права доступа ничего не означают.

Также видно два особых случая. Первый — это /var/tmp. Права на этот каталог — rwxrwxrwt. Последний символ t означает наличие у каталога дополнительного флага — т.н. *sticky bit*. Это каталог для временных файлов, и в него разрешена запись всем пользователям. Однако удалять из него пользователи могут только свои файлы.

Второй случай — это /bin/su . Здесь права — rws--x---. Владелец файла (root) может его читать, записывать и запускать. Пользователи, включённые в группу wheel, могут только запускать этот файл, прочитать его и, тем более, записать в него они не имеют права. Все прочие пользователи никаких прав на этот файл не имеют. Буква s вместо x для прав владельца файла имеет особый смысл. Это т.н. *SUID bit*, и его наличие означает, что данная программа будет запускаться не с правами пользователя, а с правами владельца файла. Иными словами, непривилегированный пользователь (но входящий в группу wheel!) может запустить эту программу и получить права её владельца — т.е. суперпользователя.

Как правило, настройки современных Linux-систем в целях повышения безопасности запрещают удалённый вход в систему с правами суперпользователя, в ряде дистрибутивов для пользователя root запрещён и локальный вход в систему. Программа /bin/su является одним из способов повысить права обычного пользователя до администратора системы. Именно поэтому её выполнение разрешено только пользователям из группы wheel.

Возможность доступа к файлу зависит также от прав доступа к каталогу, в котором находится файл. Например, даже если права доступа к файлу установлены как rwxrwxrwx, другие пользователи не могут получить доступ к файлу, пока они не имеют прав на исполнение для каталога, в котором находится файл. Другими словами, чтобы воспользоваться имеющимися у вас правами доступа к файлу, требуется иметь право на исполнение для всех каталогов вдоль пути к файлу. Например, псевдопользователь nobody не сможет прочитать файл ~/file.txt, несмотря на то, что права на этот

файл — `rw-r--r--`, т.к. права доступа к домашнему каталогу `/home/student/` — `rwx------`.

Установка и поддержание оптимальных прав доступа является одной из важнейших задач системного администратора. Права должны быть достаточными для нормальной работы пользователей и программ, но не большими, чем необходимо для такой работы. Дистрибутивы ALT Linux обладают продуманной системой прав (предопределённые группы, псевдопользователи для различных программ-серверов, права доступа для системных файлов и каталогов). Прежде чем вносить существенные изменения в эту систему, целесообразно понять её логику и выяснить, нет ли другого способа достичь нужной цели.

Поскольку программы, исполняемые от имени суперпользователя (`root`), могут совершать любые действия с любыми файлами и каталогами, их выполнение может нанести системе серьёзный ущерб. Это может быть как следствием уязвимостей или ошибок в программах, так и результатом ошибочных действий самого пользователя. Поэтому работа с правами суперпользователя требует особой осторожности.

Понятие командного интерпретатора.

Чтобы обеспечить взаимодействие пользователя с операционной системой и с прикладными программами необходим интерфейс: система передачи команд пользователя операционной системе и ответов системы обратно пользователю. Такое взаимодействие представляет собой «диалог» пользователя с компьютером на специальном языке, будь то язык, использующий знаки, похожие на слова и высказывания естественного языка, или язык изображений. На сегодня известны две принципиальные возможности организации интерфейса: графический интерфейс и командная строка.

Командная строка — приглашение оболочки, обозначающее готовность системы принимать команду пользователя — в наиболее явной форме демонстрирует идею диалога. На каждую введенную команду пользователь получает ответ от системы: либо очередное приглашение, означающее, что команда выполнена, и можно вводить следующую, либо сообщение об ошибке, представляющее собой высказывание системы о произошедших в ней событиях, адресованное пользователю. При работе в операционной среде с графическим интерфейсом происходящий диалог пользователя с системой не столь очевиден, хотя с точки зрения системы клик мышью в определенной области на экране аналогичен команде, введенной с клавиатуры, а ответ системы пользователю может быть представлен в виде диалогового окна.

При работе с командной строкой для организации интерфейса используются специальные программы — командные интерпретаторы. Они принимают от пользователя выдаваемые им команды в виде строк текста, содержащих имена программы и параметры, с которыми эти программы

следует выполнить, производят разбор полученных строк, запускают необходимые программы и передают пользователю их вывод — также строки текста. Всё взаимодействие пользователя с системой происходит через командный интерпретатор, поэтому его часто называют оболочкой (*shell*). Последовательности команд для выполнения типовых действий оказываются одинаковыми. Такие последовательности команд можно записать в текстовый файл и далее передать этот текстовый файл командному интерпретатору для выполнения. Такие текстовые файлы называются скриптами. Для запуска они должны иметь соответствующие права (флаг `x`). Командные интерпретаторы поддерживают условное выполнение команд (структуры *if-then-else*), циклы, создание и вызовы подпрограмм и т.п. Язык командного интерпретатора исключительно мощный, и позволяет автоматизировать практически любую задачу в системе. Например, действия при загрузке системы могут осуществляться скриптами командного интерпретатора — при запуске системы выполняется скрипт `/etc/rc.d/rc.sysinit`, который, в свою очередь, вызывает большое количество других скриптов.

В системах *nix, в соответствии с их модульным построением, доступны несколько командных интерпретаторов. В основном сейчас используется интерпретатор `bash` (`/bin/bash`).

Команды операционной системы представляют из себя небольшие программы, расположенные в каталогах `/bin`, `/usr/bin`, `/sbin`, `/usr/sbin`. В дальнейшем, говоря о командах, мы будем понимать под этим именно указанные программы.

Общий формат вызова команды выглядит следующим образом:

```
$ command -f --flag --key=parameter argument1 agrument2 ...
```

Здесь `$` (знак доллара) — это приглашение операционной системы к вводу команды. Для обычных пользователей оно имеет вид `$`, для суперпользователя (`root`) — # (октоторп). В дальнейшем для команд, которые требуют привилегий `root`, будет использоваться запись вида `# command`.

`command` — имя команды. Для часто использующихся команд имена, как правило, короткие, состоящие из 2-3 букв.

После имени команды, при необходимости, указываются ключи. Ключ — параметр команды, который влияет на результат её выполнения. Часто используещиеся ключи — короткие, односимвольные; для требующихся реже длинных ключей используются слова или сокращения. Короткие ключи начинаются с символа `-` (дефис), длинные — с двух символов `-` (дефис). Короткие ключи часто дублируются длинными — для повышения удобства чтения и самодокументирования скриптов. После ключей может допускаться указание дополнительных параметров, для длинных ключей такие пара-

метры принято записывать через знак = (равно). Несколько односимвольных ключей разрешается объединять вместе: например, вместо

```
$ ls -l -a
```

можно записать:

```
$ ls -la
```

Порядок ключей, как правило, не важен.

После всех ключей следуют аргументы команды. Аргументы чаще всего представляют из себя пути к файлам или каталогам. При необходимости использовать аргументы, начинающиеся со знака - (дефис), от списка ключей они отделяются двумя символами - (дефис):

```
$ touch -- -file-with-
```

Команды могут использовать различные ключи и параметры. Запоминать все возможные комбинации формата вызова каждой программы невозможно и бессмысленно. Поэтому в системе доступны описания и подсказки по использованию практически каждой утилиты и программы.

Обычно программы поддерживают несколько стандартных ключей. По ключу `-h` или `--help` выдаётся краткая справка о программе. По ключу `-v` или `--version` — её версия. Если краткой справки недостаточно, то можно вызвать описание программы в справочной системе. Для работы со справкой используется команда `man` (сокращение от manual — *англ.* руководство). Команда `man` в качестве аргумента принимает имя команды или файла конфигурации, ищет и выводит на экран страницу справочного руководства. В справке, выдаваемой командой `man`, содержится информация о формате вызова программы, поддерживаемых ей ключах и параметрах, информация об авторах и лицензии программы, в ряде случаев — примеры использования, ссылки на сайты разработчиков с дополнительной документацией.

Для просмотра страниц руководства, не помещающихся на экране, следует использовать прокрутку клавишами перемещения курсором, `<Page Up>` и `<Page Down>`. Пробел перемещает руководство на страницу вперёд. Для выхода из `man` и продолжения работы с системой следует нажать клавишу `<q>` (от *англ.* quit, выйти).

Часть программ, помимо руководств в формате `man`, также имеют и более пространную документацию в формате `info` — с вызовом её через одноимённую утилиту.

В отличие от встроенной системы подсказки программ в операционной системе Windows, руководства `man` и `info` содержат полную подробную техническую информацию о работе команд.

Основные команды системы.

Перед рассмотрением команд системы стоит отметить возможности редактирования командной строки в bash. Указанием на то, что командный интерпретатор готов принимать команды, служит т.н. приглашение — строка вида:

```
[student@lab-100 ~]$
```

В ней указывается имя пользователя и системы, на которой выполняется интерпретатор, текущий каталог (в данном случае — ~ (тильда), что означает сокращение для домашнего каталога пользователя). Завершает приглашение символ $ (знак доллара). Для суперпользователя таким символом является # (октоторп).

Команда набирается как обычная строка в текстовом редакторе. Перемещать курсор по строке возможно с помощью клавиш управления им, работают клавиши <Home> (начало строки), <End> (конец строки), <Del> (для удаления символа под курсором) и <Backspace> (для удаления символа перед курсором). Клавиша <Tab> имеет особый смысл — при её нажатии bash попытается дополнить текущее слово до ближайшего имени файла. Т.е., если в каталоге есть файлы a.txt, a1.txt и b.txt, ввод b и нажатие <Tab> дополнит b до b.txt. Если bash не может однозначно дополнить строку (например, при вводе a и <Tab>), то повторное нажатие на <Tab> выдаст все возможные варианты (в данном случае — a.txt и a1.txt). Клавишей <Tab> можно дополнять и команды, поскольку они для системы также являются файлами.

bash поддерживает историю вводимых команд. Перемещаться по списку команд можно клавишами управления курсором <стрелка вверх> и <стрелка вниз>. Команды из истории можно редактировать, как и обычные. Т.е., если bash не распознал введённую команду и выдал ошибку, проще не набирать команду заново, а, нажав <стрелку вверх>, вызвать последнюю команду и отредактировать её.

Получить полный список команд из истории можно командой history. В качестве необязательного параметра можно задать число последних команд, которые и будут выведены на экран. В истории команд по-умолчанию хранится до 500 последних команд, в системе они сохраняются в файле ~/.bash_history .

При работе с системой один из каталогов является текущим. В начале сеанса работы текущим каталогом становится домашний каталог пользователя. В приведённых ниже примерах $ или # означают приглашение командной строки. Вводить его не нужно. <имя файла> — имя произвольного файла в системе, абсолютное или относительное. Абсолютные имена файлов начинаются с символа / (слеш), включают в себя все родительские каталоги и отсчитываются от корня файловой системы. Относительные имена не начинаются с / и отсчитываются от

текущего каталога.

Текущий каталог может быть изменён командой `cd`:

`$ cd`	Перейти в домашний каталог.
`$ cd /home`	Перейти в каталог `/home/`.
`$ cd ..`	Перейти в каталог одним уровнем выше текущего.
`$ cd ~`	Перейти в домашний каталог (`~` (тильда) — сокращение для обозначения домашнего каталога).
`$ cd ~/Documents`	Перейти в подкаталог `Documents/` домашнего каталога.
`$ cd ../../etc`	Перейти в каталог `../../etc` с использованием относительного пути. При текущем каталоге `/home/student/` эта команда позволит перейти к каталогу `/etc/`.

Посмотреть текущий каталог можно командой `pwd`.

Для просмотра каталогов используется команда `ls`:

`$ ls`	Получить список файлов в текущем каталоге.
`$ ls -l`	Получить список файлов в полном формате, с указанием их прав, владельцев, групп, размера, даты создания и пр.
`$ ls -l /etc`	Получить список файлов в полном формате в каталоге `/etc/`.
`$ ls -a`	Получить список всех файлов в текущем каталоге. По-умолчанию, без флага `-a`, `ls` не показывает файлы, начинающиеся с символа `.` (точка). В таких файлах обычно хранятся настройки программ пользователя.
`$ ls -al`	Получить список всех файлов в полном формате в текущем каталоге.

Для создания каталога используется команда `mkdir <имя каталога>`.

Для удаления пустого каталога используется команда `rmdir <имя каталога>`.

Для удаления файла используется команда `rm`. Отменить результат выполнения команды `rm` и восстановить удалённые из системы файлы практически нельзя, штатных средств для этого не предусмотрено.

`$ rm file.txt`	Удалить файл `file.txt` в текущем каталоге.
`$ rm *.txt`	Удалить все файлы, заканчивающиеся на `.txt`, в текущем каталоге.

`$ rm -f *`	Удалить все файлы в текущем каталоге, не запрашивая разрешений.
`$ rm -r directory/`	Рекурсивно удалить все файлы в каталоге `directory/` и сам каталог `directory/`.
`$ rm -rf *`	Рекурсивно удалить все файлы и каталоги из текущего каталога, не запрашивая подтверждения. Данная команда, отданная от имени суперпользователя и в корневом каталоге, удалит всю файловую систему — **без дополнительных вопросов и возможности отмены действия** (в ряде дистрибутивов в `rm` специально внесены изменения, запрещающие такое поведение).

Для смены даты последнего изменения файла на текущую используется команда `touch <имя файла>`.Если файла не существует, `touch` создаст новый файл нулевого размера.

Для вывода на экран содержимого текстового файла или его части используются команды `cat`, `less`, `more`, `head`, `tail`.

`$ cat /etc/passwd`	Вывести на экран содержимое файла `/etc/passwd`.
`$ more /etc/passwd`	Вывести на экран содержимое файла `/etc/passwd`. Если вывод не будет помещаться на одном экране — вывести начало файла и ждать нажатия любой клавиши для следующей страницы.
`$ less /etc/passwd`	Вывести на экран содержимое файла `/etc/passwd`. Если вывод не будет помещаться на одном экране — вывести начало файла и позволить пользователю просмотреть его, используя прокрутку клавишами управления курсором. Для завершения работы команды `less` следует нажать клавишу `<q>`.
`$ head /etc/passwd`	Вывести первые 10 строк файла `/etc/passwd`.
`$ head -5 /etc/passwd`	Вывести первые 5 строк файла `/etc/passwd`.
`$ tail /etc/passwd`	Вывести последние 10 строк файла `/etc/passwd`.

`$ tail -1 /etc/passwd`	Вывести последнюю строку файла `/etc/passwd`.

Правами доступа к файлам можно управлять командой `chmod`:

`$ chmod u+rwx file.sh`	Добавить файлу `file.sh` право владельцу на чтение, запись и выполнение.
`$ chmod g+x files.sh`	Добавить для группы файла `files.sh` право на выполнение.
`$ chmod u=rw,g=r,o-rwx file.txt`	Для файла `file.txt`: установить права для владельца в `rw-`, для группы — в `r--`, для всех остальных — в `---`.
`$ chmod a+rx file.sh`	Добавить для `file.sh` права на чтение и выполнение для всех пользователей.
`$ chmod g-w,o-rwx file.txt`	Снять с файла `file.txt` права на запись для группы пользователей и все права для остальных пользователей.
`$ chmod -R g+w directory/`	Добавить для всех файлов и каталогов внутри каталога `directory/` прав на запись для группы.
`$ chmod -R g+X directory/`	Добавить для всех *каталогов* внутри каталога `directory/` право на выполнение, а для *файлов* право на выполнение оставить прежним.

Для смены своего пароля пользователь может использовать команду `passwd`. При запуске команда запросит у пользователя его текущий пароль, новый пароль и для подтверждения ввода — повторение нового пароля. Ввод паролей на экране не отображается. При совпадении введённых паролей, пароль пользователя будет изменён. Важно помнить, что пароль является средством, по которому система аутентифицирует пользователя. Короткие или легко угадывающиеся пароли очень быстро и просто находятся путём их перебора. Учитывая, что *nix — это сетевые операционные системы, слабые пароли пользователей легко позволяют злоумышленникам подбирать их и проникать в системы.

В настоящее время пароль не должен содержать менее 8 символов. Эти символы не должны быть одинаковыми, пароль не должен содержать только цифры или быть словарным словом. Все подобные, слабые пароли легко определяются современными программами для взлома систем в автомати-

ческом режиме. Если пароль покажется программе passwd слишком слабым, она не позволит его задать.

Учитывая особый статус учётной записи пользователя root, в Linux-системах обычно удалённый вход этого пользователя запрещён.

Как указывалось выше, для получения справки по любой команде используется команда man. Для получения справки по использованию man следует ввести man man.

Для удобного перемещения по дереву каталогов и работы с файлами возможно использование файловых менеджеров. Наиболее распространённый из них — *Midnight Commander*, запускаемый командой mc. Он использует стандартный двухпанельный интерфейс файловых менеджеров типа *Norton Commander*, встроенный текстовый редактор, программу просмотра текста, может работать с архивами в различных форматах, с файлами на удалённых серверах *ftp*, *cifs*, *ssh* и др.

В *Midnight Commander* существует встроенный интерфейс командной строки, вызываемый комбинацией клавиш <Ctrl>+<o>. Повторное нажатие <Ctrl>+<o> возвращает панели менеджера файлов. Для выхода из mc используется клавиша <F10> или последовательность клавиш <Esc>,<0>.

Помимо встроенного текстового редактора mc, в системе доступны также и другие текстовые редакторы. Для пользователей *nix-систем представляется полезным иметь хотя бы минимальные навыки работы с редактором vi — как стандартным редактором, имеющимся практически во всех системах.

При запуске редактора vi в командной строке ему указывается имя файла для редактирования:

```
vi file.txt
```

Если файл существует, то vi загружает его и отображает на экране, если нет — при сохранении создаётся новый файл с указанным именем. vi (*visual editor*) впервые появился в середине 70-х годов и имеет интерфейс, приспособленный для работы на самых простых терминалах. vi работает в двух основных режимах — в режиме «ввода текста» и в режиме «команд».

После запуска vi оказывается в режиме «команд». В этом режиме можно перемещаться по тексту с помощью клавиш управления курсором. На тех терминалах, где таких клавиш нет, можно использовать клавиши <h>, <l> (влево, вправо), <j>, <k> (вниз, вверх). Для перемещения курсора на следующий символ X в строке используется последовательность <f>,<X> (<f>,<"> - перемещение на символ "), в обратном направлении – <F>,<X> . Для перехода в начало строки используется команда <^>, в конец – <$> .

Для перехода на строку с номером N можно набрать её номер и команду <G> (<1>,<0>,<G> – переместить курсор на 10ую строку). Также можно осуществить поиск по тексту, нажав символ </> и введя нужный шаблон поиска. Повторный поиск в прямом и обратном направлении осуществляется клавишами <n> и <N> .

Найдя нужное место, можно перейти в режим «ввода текста». Для этого надо нажать <i> для вставки текста в текущей позиции, или <a> для добавления в конце строки. Для вставки новой строки в режиме ввода текста используется клавиша <Enter>. Выйти из режима ввода текста можно, нажав <Esc>.

В командном режиме можно удалять символы и строки текста. Для удаления символа под курсором используется клавиша <x>, для удаления строки — последовательность <d>,<d>. Удалённые символы или строки помещаются в буфер обмена. Содержимое буфера обмена можно вставить в текст клавишей <p>. Просто поместить текущую строку в буфер обмена можно последовательностью <y>,<y>.

Перед командой можно задать число её повторений. Например, последовательность клавиш <1>,<0>,<j> переместит курсор на 10 строк вниз, <7>,<l> - на 7 символов вправо, <8>,<x> удалит в буфер обмена 8 символов, начиная с текущей позиции, <d>,<5>,<k> - удалит в буфер обмена 5 строк вверх, начиная с текущей.

Для завершения редактирования файла и сохранения результата надо набрать в командном режиме :wq; для завершения редактирования без сохранения — :q!.

Возможности vi (и его улучшенной и расширенной версии vim) не ограничиваются простым редактированием текста, для их изучения можно использовать встроенное интерактивное руководство, вызываемое командой vimtutor.

Выйти из командного интерпретатора и завершить сеанс работы с системой можно, введя команду logout.

Следующие команды доступны администратору системы и позволяют управлять пользователями, их правами для доступа к файлам, и т.п.

$ su -l	Запустить командный интерпретатор с правами суперпользователя. Ключ -l обязателен. При выполнении команда запросит пароль суперпользователя. Завершить работу командного интерпретатора можно, как и в случае обычного пользователя, командной logout.

`# su -l user`	Запустить командный интерпретатор с правами пользователя `user`.
`# useradd user`	Добавить учётную запись пользователя `user`. При этом создаются необходимые записи в файлах `/etc/passwd` и `/etc/group`, а также домашний каталог пользователя. Пароль новому пользователю не назначается, и войти в систему до его задания он не может.
`# userdel user`	Удалить учётную запись пользователя `user`. Файлы, принадлежащие пользователю, при этом не удаляются.
`# passwd user`	Задать пароль пользователя `user`. Суперпользователю знать старый пароль `user` для его смены не нужно.
`# chmod <права> file`	Изменить права на файл. `root` может изменить права доступа к любому файлу.
`# chown user file`	Изменить владельца файла на пользователя `user`.
`# chown :group file`	Изменить группу файла на `group`.
`# chown user:group *`	Изменить владельца и группу всех файлов в текущем каталоге.
`# chown -R user:group *`	Рекурсивно изменить владельца и группу всех файлов в текущем каталоге и подкаталогах.
`# shutdown`	Выключить систему. Без дополнительных ключей команда `shutdown` останавливает систему, не отключая её питание. Удалённо выключить систему можно, но включить её после этого возможно только внешними средствами.
`# halt`	Выключить систему, не отключая её питание (аналогично вызову `shutdown`).
`# shutdown -h`	Выключить систему, и отключить её питание.
`# poweroff`	Выключить систему, в т.ч. отключить её питание (аналогично вызову `shutdown -h`).
`# shutdown -r`	Перезагрузить систему.
`# reboot`	Перезагрузить систему (аналогично вызову `shutdown -r`).

При работе с правами суперпользователя следует помнить, что никаких ограничений и прав доступа для этой учётной записи не существует. Поэтому неосторожная команда или опечатка может привести систему в

нерабочее состояние.

Управление выполнением программ.

Каждая выполняющаяся в Linux программа называется процессом. Linux, как многопользовательская многозадачная система характеризуется тем, что в ней одновременно может выполняться множество процессов, принадлежащих разным пользователям. Вывести список исполняющихся в текущее время процессов можно командой `ps`, например, следующим образом:

```
$ ps
PID TT STAT  TIME COMMAND
24  3  S     0:03 bash
161 3  R     0:00 ps
$
```

По-умолчанию команда `ps` выводит список только тех процессов, которые принадлежат запустившему её пользователю и выполняются в данной сессии. Чтобы посмотреть все исполняющиеся в системе процессы, нужно использовать ключ `-a`, т. е. запускать команду как `ps -a`. Наиболее полный вид списка процессов, с указанием их владельцев, времени запуска, потребляемых ресурсов (памяти и процессора) можно просмотреть командой `ps -aux`.

Номера процессов (*process ID*, или *PID*), указанные в первой колонке, являются уникальными номерами, которые система присваивает каждому работающему процессу. Последняя колонка, озаглавленная `COMMAND`, показывает имя работающей команды. Среди команд, запущенных данным пользователем, есть только `bash` и сама команда `ps`. (`bash` — это командный интерпретатор (командная оболочка, *англ.* shell), который обрабатывает вводимые пользователем с терминала команды и обеспечивает их выполнение в системе. Более подробно роль командного интерпретатора рассматривалась в предыдущей лабораторной работе.) Видно, что командная оболочка `bash` выполняется одновременно с командой `ps`. Когда пользователь ввёл команду `ps`, оболочка `bash` начала её исполнять. После того, как команда `ps` закончила свою работу (таблица процессов выведена на экран), управление возвращается процессу `bash`. Тогда оболочка `bash` выводит на экран приглашение и ждёт новой команды.

Работающий процесс также называют заданием (*англ.* job). Понятия процесс и задание являются взаимозаменяемыми. Однако обычно процесс называют заданием, когда имеют в виду управление заданием (*англ.* job control). Управление заданием — это функция командной оболочки, которая предоставляет пользователю возможность переключаться между несколькими заданиями.

В большинстве случаев пользователи в каждый момент времени запускают только одно задание — ту команду, которую они ввели и запустили из командной оболочки. Однако многие командные оболочки (включая `bash` и

`tcsh`) имеют функции управления заданиями, позволяющие запускать одновременно несколько команд или заданий и, по мере надобности, переключаться между ними.

Управление заданиями может быть полезно, если, например, при редактировании большого текстового файла возникает необходимость временно прервать редактирование и выполнить какую-нибудь другую операцию. С помощью функций управления заданиями можно приостановить работу с редактором, вернуться к приглашению командной оболочки и запустить какие-либо другие команды. Когда они будут выполнены, можно вернуться обратно к работе с редактором в то его состояние, на котором была прервана работа с редактируемым файлом.

Передний план и фоновый режим.

Задания могут выполняться или на переднем плане (*англ.* foreground), или в фоновом режиме (*англ.* background). На переднем плане в любой момент времени может быть только одно задание. Задание на переднем плане взаимодействует с пользователем, получает ввод с клавиатуры терминала и посылает вывод на экран. Задания в фоновом режиме не получают ввода с терминала и обычно ничего на него не выводят (в противном случае выводящиеся из них данные будут произвольным образом смешиваться с выводом из команды переднего плана). Как правило, это задания, которые не нуждаются во взаимодействии с пользователем.

Некоторые задания исполняются очень долго, и во время их работы не происходит ничего интересного. Пример таких заданий — компилирование программ, а также сжатие больших файлов. Нет никаких причин смотреть на экран и ждать, когда эти задания выполнятся. Такие задания вполне можно запускать в фоновом режиме, тогда во время их выполнения будет возможность продолжать работать с системой.

Для управления выполнением процессов в Linux предусмотрен механизм передачи сигналов. Сигналы предоставляют процессам возможность обмениваться стандартными короткими сообщениями непосредственно с помощью операционной системы. Сообщение-сигнал не содержит никакой информации, кроме номера сигнала (для удобства вместо номера можно использовать предопределённое системой имя). Для того, чтобы передать сигнал, процессу достаточно задействовать системный вызов `kill()`. Для обработки поступающих сигналов процесс может зарегистрировать в системе для интересующих его сигналов свои процедуры-обработчики, или воспользоваться предоставляемыми системой стандартными обработчиками сигналов. В зависимости от номера сигнала стандартные обработчики или не выполняют никаких действий, или приводят к немедленному завершению получившего сигнал процесса.

Обработчик сигнала запускается асинхронно, немедленно после получения сигнала, что бы процесс в это время ни делал. В этом механизм сигналов очень похож на механизм обработки прерываний от аппаратной части компьютера; сигналы являются одним из вариантов внутренних прерыва-

ний в системе – так называемыми программными прерываниями.

Сигналы с номерами 9 (KILL) и 19 (STOP) всегда обрабатываются операционной системой. Первый из них принудительно останавливает и уничтожает процесс (отсюда и название, *англ.* kill — убивать). Сигнал STOP приостанавливает процесс: в таком состоянии процесс не удаляется из таблицы процессов, но и не выполняется до тех пор, пока не получит сигнал 18 (CONT), после чего продолжает работу. В командной оболочке Linux сигнал STOP можно передать активному процессу с помощью управляющей последовательности клавиш <Ctrl>+<Z>.

Сигнал номер 15 (TERM) служит для прекращения (*англ.* terminate) работы задания. При поступлении этого сигнала процесс должен завершить свою работу. Командная оболочка позволяет отправить сигнал TERM активному процессу с помощью управляющей последовательности <Ctrl>+<C>. При этом, в отличие от сигнала KILL, программы могут перехватывать сигнал TERM и установить собственный обработчик этого сигнала, т. е. нажатие комбинации клавиш <Ctrl>+<C> может и не прервать процесс немедленно. Это сделано для того, чтобы программа могла корректно завершить свою работу: удалить временные файлы, осуществить запись изменённых данных и т. п., прежде, чем она будет завершена. На практике, некоторые программы прервать таким способом не получится.

Существует утилита kill, предназначенная для отправления того или иного сигнала произвольному процессу. Её формат вызова:

```
kill [-s SIGNAL | -SIGNAL] PID
```

где SIGNAL — это посылаемый процессу сигнал, а PID — соответствующий идентификатор процесса. Например, для посылки сигнала KILL процессу 1 можно записать:

```
$ kill -9 1
-bash: kill: (1) - Операция не позволена
```

Запущенная обычным пользователем, такая команда закончится с ошибкой: на отправление сигналов также распространяются соглашения о контроле доступа, и обычный пользователь может отправлять сигналы только процессам, запущенным им самим (т. е. процессам с *UID* этого пользователя). Как говорилось в предыдущей лабораторной работе, процесс с *PID*, равным 1 — это процесс init, запускающийся первым после загрузки ядра операционной системы и от имени суперпользователя. Сам суперпользователь (администратор системы) может отправить любой сигнал любому процессу.

Перевод в фоновый режим и уничтожение заданий.

Рассмотрим управление заданиями на простого примере. Существует команда yes, которая выводит бесконечный поток строк, состоящих из символа y . При запуске этой команды на экран начинают выводиться строки с

буквой 'y':

```
$ yes
y
y
y
y
y
```

Последовательность таких строк будет бесконечно продолжаться – пока выполняется команда `yes` . Остановить её выполнение можно, отправив команде сигнал прерывания, т.е. нажав `<Ctrl>+<C>`.

Чтобы на экран не выводилась эта бесконечная последовательность, перенаправим стандартный вывод команды `yes` на `/dev/null`. Устройство `/dev/null` — одно из специальных устройств в системе, оно действует как «чёрная дыра»: все данные, посланные в это устройство, пропадают. С помощью этого устройства очень удобно избавляться от слишком обильного вывода некоторых программ. Подробнее о перенаправлении устройств ввода-вывода рассказано ниже по тексту в соответствующем разделе.

```
$ yes > /dev/null
```

Теперь на экран ничего не выводится. Однако и приглашение командной оболочки также не возвращается. Это происходит потому, что команда `yes` все ещё работает и посылает свои сообщения, состоящие из букв `y`, в устройство `/dev/null`. Уничтожить это задание также можно, отправив ему сигнал прерывания.

Можно сделать так, чтобы команда `yes` продолжала работать, но при этом приглашение командной оболочки вернулось на экран и стало возможно работать с другими программами. Для этого можно команду `yes` перевести в фоновый режим, и она будет там выполняться параллельно с другими запускаемыми из командного интерпретатора программами.

Один из способов запустить процесс в фоновом режиме — дописать символ `&` (амперсанд) в конце строки запуска команды:

```
$ yes > /dev/null &
[1]+ 164
$
```

Сообщение `[1]` представляет собой номер задания (*англ.* job number) для процесса `yes`. Командная оболочка присваивает номер задания каждому исполняемому заданию. Поскольку `yes` является единственным исполняемым заданием в данном сеансе, ему был присвоен порядковый номер 1. Число 164 является идентификационным номером, соответствующим данному процессу (*PID*); он уникален для системы в целом. К запущенному в фоновом режиме процессу можно обращаться, указывая как его *PID*, так и номер задания.

Для того, чтобы проверить состояние запущенного и работающего в фоновом режиме процесса, можно использовать команду `jobs`, которая является внутренней командой оболочки.

```
$ jobs
[1]+  Running                 yes >/dev/null  &
$
```

В выводе команды `jobs` указывается, какие задания запущены, их номера, текущее состояние (выполняется, приостановлено, ожидает ввода-вывода) и вид командной строки. Также для того, чтобы узнать статус задания, можно воспользоваться командой `ps`, как это было показано выше.

Для того, чтобы передать процессу сигнал (чаще всего когда возникает потребность прервать работу задания) используется упомянутая выше утилита `kill`. В качестве аргумента этой команде даётся либо номер задания, либо *PID*. Необязательный параметр — номер сигнала, который нужно отправить процессу. По умолчанию отправляется сигнал `TERM`. Если к заданию нужно обратиться по его номеру (а не через *PID*), то номер задания в параметрах команды `kill` указывается через символ `%` (процент). В рассмотренном выше случае номер задания был 1, так что команда `kill %1` прервёт работу задания:

```
$ kill %1
$ jobs [1]
Terminated              yes       >/dev/null
```

Фактически, задание уничтожено, и при вводе команды `jobs` в следующий раз, на экране о нём не будет никакой информации.

Уничтожить задание можно также, используя идентификационный номер процесса (*PID*). Этот номер, наряду с идентификационным номером задания, указывается во время старта задания. В нашем примере значение *PID* было 164, так что команда `kill 164` была бы эквивалентна команде `kill %1`. При использовании *PID* в качестве аргумента команды `kill` вводить символ `%` (процент) не требуется.

Приостановка и продолжение работы заданий.

Запустим командой `yes` на переднем плане процесс, как это делалось раньше:

```
$ yes > /dev/null
```

Как и ранее, поскольку процесс работает на переднем плане, приглашение командной оболочки на экран не возвращается.

Теперь вместо того, чтобы прервать задание комбинацией клавиш `<Ctrl>+<C>`, приостановим его (suspend, *англ.* подвесить), отправив сигнал `STOP`. Для приостановки задания надо нажать соответствующую комбина-

цию клавиш, обычно это <Ctrl>+<Z>.

```
$ yes > /dev/null
Ctrl-Z[1]+  Stopped yes       >/dev/null
$
```

Приостановленный процесс попросту не выполняется, на него не тратятся вычислительные ресурсы процессора. Приостановленное задание можно вновь запустить на выполнение с той же точки, в которой оно было приостановлено, как будто бы этого не происходило.

Для возобновления выполнения задания на переднем плане можно использовать команду fg (от *англ.* foreground — передний план).

```
$ fg
yes >/dev/null
```

Командная оболочка ещё раз выведет на экран название команды, чтобы пользователь знал, какое именно задание он в данный момент запустил на переднем плане. Приостановим это задание ещё раз нажатием клавиш <Ctrl>+<Z>, но в этот раз запустим его в фоновом режиме командой bg (от *англ.* background — фон). После перевода в фоновый режим процесс будет работать так, как если бы при его запуске использовалась команда с символом & (амперсанд) на конце (как это делалось в предыдущем разделе):

```
$ bg
[1]+ yes $>$/dev/null  &
$
```

При этом приглашение командной оболочки возвращается пользователю, а команда jobs будет показывать, что процесс yes действительно в данный момент работает. Этот процесс можно уничтожить командой kill, как показывалось ранее.

Для того, чтобы приостановить работающее в фоновом режиме задание, нельзя воспользоваться комбинацией клавиш <Ctrl>+<Z>. Прежде, чем приостанавливать задание, его нужно перевести на передний план командой fg, и лишь потом приостановить. Таким образом, команду fg можно применять либо к приостановленным заданиям, либо к заданию, работающему в фоновом режиме. Другой вариант приостановки работающего в фоновом режиме задания – это отправка ему сигнала STOP командой kill .

Задания, работающие в фоновом режиме, могут пытаться выводить некоторый текст на экран. Это будет мешать работать над другими задачами.

```
$ yes &
```

Здесь стандартный вывод не был перенаправлен на устройство /dev/null, поэтому на экран будет выводиться бесконечный поток символов y. Этот поток невозможно будет остановить, поскольку комбинация клавиш

<Ctrl>+<C> не воздействует на задания в фоновом режиме. Для того чтобы остановить эту выдачу, надо использовать команду fg, которая переведёт задание на передний план, а затем уничтожить задание комбинацией клавиш <Ctrl>+<C>.

Вызываемые без аргументов, команды fg и bg воздействуют на те задания, которые были приостановлены последними (если ввести команду jobs, эти задания будут помечены символом + (плюс) рядом с их номером). Если в одно и то же время работает одно или несколько заданий, задания можно помещать на передний план или в фоновый режим, задавая в качестве аргументов команды fg или команды bg их идентификационный номер (*англ.* job ID). Например, команда fg %2 помещает задание номер 2 на передний план, а команда bg %3 помещает задание номер 3 в фоновый режим. Использовать *PID* в качестве аргументов команд fg и bg нельзя.

Более того, для перевода задания на передний план можно просто указать его номер. Так, команда %2 будет эквивалентна команде fg %2.

Отметим также, что функции управления заданиями реализуются средствами командного интерпретатора. Команды fg, bg и jobs являются внутренними командами оболочки, т. е. одноимённых файлов с их программным кодом в файловой системе нет. В простых командных интерпретаторах, например на встраиваемых системах, эти команды могут не поддерживаться. В этих случаях управлять работой процессов можно, посылая им сигналы стандартной командой kill .

Код возврата команд.

Любая запускаемая в системе команда (программа) выполняет какие-то действия, операции, задачи или успешно и без ошибок, или же в процессе работы программы возникают какие-либо проблемы, и выполнить поставленную задачу программа не может. О результатах своей работы и возникших ошибках программа сообщает запустившему её пользователю, выдавая текстовые информационные сообщения на экран. И, помимо этого, программа сообщает о результатах своей работы и операционной системе — через выдаваемый в операционную систему в момент своего завершения код возврата. Код возврата команды — это целое число, или равное нулю в случае успешного завершения команды, или не равное нулю в случае возникновения каких-либо ошибок. Возможные значения кодов возврата в случае ошибок выполнения команды зависят от конкретной команды и, как правило, приводятся на странице справочного руководства (man) по этой команде.

Код возврата последней выполненной команды командный интерпретатор запоминает в переменной $? (подробнее о переменных командного интерпретатора рассказывается ниже). Посмотреть его можно через команду echo :

```
$ ls /tmp
$ echo $?
```

```
0
$ ls /tmp/0
ls: невозможно получить доступ к /tmp/0: Нет такого файла или каталога
$ echo $?
2
```

Здесь сначала успешно выводится список файлов из (пустого) каталога `/tmp`, а далее при попытке обратиться к несуществующему `/tmp/0` возникает ошибка. При этом `ls` как выводит сообщение об ошибке, так и возвращает ненулевой код возврата, сигнализирующий о ней.

Управление последовательностью выполнения команд.

В строке ввода интерпретатор команд позволяет ввести и запустить сразу несколько разных команд. Если команды требуется просто запускать последовательно одну за другой без учёта результата выполнения предыдущей команды перед запуском следующей, то их достаточно разделить точкой с запятой:

```
$ cd /bin; ls -l sh
-rwxr-xr-x 1 root root 486600 апр 19  2013 sh
```

Но также при запуске последующей команды можно и учитывать результат выполнения предыдущей. Если команда завершилась успешно (т. е. её код возврата равен нулю), то командный интерпретатор считает, что результат выполнения команды — логическая истина. Если код возврата отличен от нуля (т. е. произошла какая-либо ошибка), то результат выполнения команды – логическая ложь.

Для запуска следующей команды только в том случае, если предыдущая команда завершилась успешно, используется оператор «логическое И», записываемый как `&&` :

```
$ cd /tmp/ && touch file
```

Здесь команда `touch file` запускается только после успешного выполнения команды `cd /tmp`, т. е. после перехода в каталог `/tmp/` . В случае невозможности перехода в каталог команда `touch` запущена не будет.

Для запуска следующей команды только в том случае, если предыдущая завершилась с ошибкой, используется оператор «логическое ИЛИ», записываемый как `||` :

```
$ cd /tmp/0 || mkdir /tmp/0
```

Здесь делается попытка перехода в каталог `/tmp/0`, и если это не удаётся (например, такого каталога нет), запускается команда `mkdir /tmp/0` , создающая этот каталог.

Использование операторов «логического И» и «ИЛИ» для условий выполнения команды в зависимости от результата предыдущей команды основывается на логике оптимизации выполнения этих операций в языках программирования: результатом «логического И» будет логическая истина

в случае, если оба операнда равны логической истине. Если первый операнд – логическая ложь, то результат – логическая ложь при любом значении второго операнда, и его можно не вычислять. Аналогично, результатом «логического ИЛИ» будет логическая истина в случае, если один из операндов равен логической истине. Соответственно, если первый операнд равен логической истине, то результат уже известен, и значение второго операнда вычислять смысла нет.

Потоки ввода-вывода и их перенаправление.

Программы нужны для того, чтобы обрабатывать данные: принимать одно, на выходе выдавать другое, причём в качестве данных может выступать практически что угодно: текст, числа, звук, видео и т.д. Потоки входных и выходных данных для команды называются вводом и выводом. Потоков ввода и вывода у каждой программы может быть и по несколько. В Linux каждый процесс при создании в обязательном порядке получает так называемые стандартный ввод (*англ.* standard input, *stdin*), стандартный вывод (*англ.* standard output, *stdout*) и стандартный вывод ошибок (*англ.* standard error, *stderr*).

Программы работают с потоками ввода-вывода как с обычными файлами. С точки зрения программирования потоки ввода-вывода — это доступные сразу после запуска программы заранее открытые файловые дескрипторы с номерами 0, 1 и 2 для стандартного ввода, стандартного вывода и стандартного вывода ошибок соответственно. При необходимости программы могут переопределять эти файловые дескрипторы, закрывать их, и т.д.

Стандартные потоки ввода/вывода предназначены в первую очередь для обмена текстовой информацией. Тут даже не важно, кто общается с помощью текстов, человек с программой или программы между собой —главное, чтобы у них был канал передачи данных, и чтобы они говорили «на одном языке».

Текстовый принцип работы с машиной позволяет отвлечься от конкретных частей компьютера, вроде системной клавиатуры и видеокарты с монитором, рассматривая единое оконечное устройство, посредством которого пользователь вводит текст (команды) и передаёт его системе, а система выводит необходимые пользователю данные и сообщения (диагностику и ошибки). Такое устройство называется терминалом. В общем случае терминал — это точка входа пользователя в систему, обладающая способностью передавать текстовую информацию. Терминалом может быть отдельное внешнее устройство, подключаемое к компьютеру через порт последовательной передачи данных (*COM port* в терминологии персональных компьютеров). В роли терминала также могут работать и специальные программы: например, *PyTTY* и серверная часть — демон удалённого управления системой `ssh`. При работе с командной строкой стандартный ввод командной оболочки связан с клавиатурой, а стандартный вывод и вывод ошибок — с экраном монитора (или окном эмулятора терминала).

Рассмотрим в качестве примера одну из простейших команд — `cat`.

Обычно команда `cat` читает данные из всех файлов, которые указаны в качестве её параметров, и посылает считанное непосредственно в стандартный вывод (*stdout*). Следовательно, команда

```
$ cat /etc/hosts /etc/resolv.conf
127.0.0.1 lab-00.edu.cbias.ru lab-00 localhost.localdomain localhost
192.168.212.250 ftp-distr
nameserver 192.168.212.252
```

выведет на экран сначала содержимое файла `/etc/hosts`, а затем — файла `/etc/resolv.conf` .

Однако если имя файла не указано, программа `cat` читает входные данные из *stdin* и немедленно возвращает их в *stdout* (никак не изменяя). Данные проходят через `cat`, как через «трубу». Приведём пример:

```
$ cat
Hello there.
Hello there.
Bye.
Bye.
Ctrl-D$
```

Каждую строчку, вводимую с клавиатуры, программа `cat` немедленно возвращает на экран. При вводе информации со стандартного ввода конец текста отмечается вводом специальной комбинации клавиш, как правило — `<Ctrl>+<D>`.

Приведём другой пример. Команда `sort` читает строки вводимого текста (также из *stdin*, если не указано ни одного имени файла) и выдаёт набор этих строк в упорядоченном виде в *stdout*. Проверим её действие.

```
$ sort
bananas
carrots
apples
Ctrl-D
apples
bananas
carrots $
```

Как видно, после нажатия `<Ctrl>+<D>` команда `sort` вывела строки упорядоченными в алфавитном порядке.

Перенаправление ввода и вывода.

Допустим, нужно направить вывод команды `sort` в некоторый файл, чтобы сохранить упорядоченный по алфавиту список на диске. Командная оболочка позволяет перенаправить стандартный вывод команды в файл, используя символ `>` (больше). Приведём пример:

```
$ sort > list
bananas
carrots
apples
```

```
Ctrl-D$
```

Можно увидеть, что результат работы команды `sort` не выводится на экран, однако он сохраняется в файле с именем `list`. Выведем на экран содержимое этого файла:

```
$ cat list
apples
bananas
carrots
$
```

Пусть теперь исходный неупорядоченный список находится в файле `items`. Этот список можно упорядочить с помощью команды `sort`, если указать ей, что она должна читать данные из этого файла, а не из своего стандартного ввода, и, кроме того, перенаправить стандартный вывод в файл, как это делалось выше. Пример:

```
$ sort items > list
$ cat list
apples
bananas
carrots
$
```

Однако можно поступить иначе, перенаправив не только стандартный вывод в файл, но и стандартный ввод утилиты из файла, используя для этого символ < (меньше):

```
$ sort < items
apples
bananas
carrots
$
```

Результат команды `sort < items` эквивалентен команде `sort items`, однако при выдаче команды `sort < items` система ведёт себя так, как если бы данные, которые содержатся в файле `items`, были введены со стандартного ввода. Перенаправление ввода-вывода осуществляется командной оболочкой. Команде `sort` не сообщалось имя файла `items`, эта команда читала данные из своего стандартного ввода, как если бы их вводили с клавиатуры.

Введём понятие фильтра. Фильтром является программа, которая читает данные из стандартного ввода, некоторым образом их обрабатывает и результат направляет в стандартный вывод. Когда применяется перенаправление, в качестве стандартного ввода и вывода могут выступать файлы. Как указывалось выше, по умолчанию, *stdin* и *stdout* относятся к клавиатуре и к экрану соответственно. Программа `sort` является простым фильтром: она сортирует входные данные и посылает результат на стандартный вывод. Совсем простым фильтром является программа `cat`: она ничего не делает с входными данными, а просто пересылает их на выход.

Если вывод команды не интересен, его можно перенаправить на специальное устройство `/dev/null` — как говорилось выше, все данные, посланные в это устройство, удаляются. Также существуют специальные устройства `/dev/zero` – из которого можно прочитать неограниченное число нулевых символов, `/dev/random` – из которого можно прочитать случайные символы, `/dev/urandom` – для чтения последовательности псевдослучайных символов.

Использование состыкованных команд (конвейер).

Выше уже демонстрировалось, как использовать программу `sort` в качестве фильтра. В этих примерах предполагалось, что исходные данные находятся в некотором файле, или что эти исходные данные будут введены с клавиатуры (стандартного ввода). Однако часто требуется отсортировать данные, которые являются результатом работы какой-либо другой команды, например, `ls` .

Будем сортировать данные в обратном алфавитном порядке, это делается опцией `-r` команды `sort`. Если нужно перечислить файлы в текущем каталоге в обратном алфавитном порядке, один из способов сделать это будет следующим. Для получения списка файлов используем команду `ls` :

```
$ ls /bin
arch
awk
basename
bash
....
$
```

Теперь перенаправляем выход команды `ls` в файл с именем `file-list`, и далее сортируем этот файл с помощью команды `sort`:

```
$ ls /bin > file-list
$ sort -r file-list
zcat
ypdomainname
xargs
wc
...
$
```

Здесь вывод команды `ls` был сохранён в файле, а после этого файл был обработан командой `sort -r`. Однако этот путь является неэффективным — он требует использования временного файла для хранения выходных данных программы `ls` , лишних операций ввода-вывода для создания, записи и последующего чтения этого временного файла с диска.

Решением в данной ситуации может служить создание состыкованных команд (*англ.* pipelines). Состыковку осуществляет командная оболочка, которая *stdout* первой команды направляет на *stdin* второй команды. В данном случае мы хотим направить *stdout* команды `ls` на *stdin* команды `sort`.

Для состыковки используется символ | (вертикальная черта), как это показано в следующем примере:

```
$ ls /bin | sort -r
zcat
ypdomainname
xargs
wc
...
$
```

Эта команда короче, чем последовательность отдельных команд, и её проще набирать.

Рассмотрим ещё один пример. Команда

```
$ ls /usr/bin
```

выдаёт длинный список файлов. Большая часть этого списка выводится на экран слишком быстро, чтобы его содержимое можно было прочитать. Попробуем использовать команду `more` для того, чтобы выводить этот список частями:

```
$ ls /usr/bin | more
```

Теперь можно этот список «перелистывать».

Можно пойти дальше и состыковать более двух команд. Рассмотрим команду `head`, которая является фильтром, выводящим первые строки из входного потока (в нашем случае на вход будет подан выход от нескольких состыкованных команд). Если мы хотим вывести на экран последнее по алфавиту имя файла в текущем каталоге, можно использовать следующую длинную команду:

```
$ ls | sort -r | head -1 notes
```

где команда `head -1` выводит на экран первую строку получаемого ей входного потока строк (в нашем случае поток состоит из данных от команды `ls`), отсортированных в обратном алфавитном порядке.

Фильтры не обязательно используются только для обработки текста. Например, в пакете `netpbm` содержатся утилиты для обработки изображений, которые тоже являются фильтрами. Для увеличения иконки *Midnight Commander* в 5 раз и преобразования её из формата *PNG* в *JPEG* можно использовать такую связку команд:

```
$ pngtopnm /usr/share/icons/mc.png | pnmenlarge 5 | pnmsmooth | pnmtojpeg >
/tmp/mc.jpg
```

Здесь `pngtopnm` читает файл иконки (`/usr/share/icons/mc.png`) в формате *PNG*, преобразует его в формат *PNM* и выдаёт результат в стандартный вывод. `pnmenlarge` принимает файл в формате *PNM* из стандартного ввода, увеличивает (масштабирует) картинку в 5 раз и выдаёт результат в стандартный вывод. Далее `pnmsmooth` выполняет операцию сглаживания, а `pnmtojpeg` преобразует поток данных в формат *JPEG*. Итоговый результат

pnmtojpeg также выдаёт на стандартный выход, который средствами командного интерпретатора перенаправляется в файл /tmp/mc.jpg.

Другой пример: утилита mkisofs создаёт для файлов из заданного ей в качестве параметра каталога образ диска с файловой системой *ISO9660* для записи на оптические диски. А утилита cdrecord умеет записывать такие образы непосредственно на сами диски. Утилиты могут использоваться по-отдельности, с записью образа файловой системы в файл и последующей записью такого файла на диск. Однако их можно объединить в связку и записывать диски без создания временных файлов:

```
$ mkisofs ~/mydisk | cdrecord -
```

Здесь для того, чтобы приказать cdrecord использовать данные со стандартного входа, а не читать их из файла, мы в качестве имени файла указали – (дефис).

Недеструктивное перенаправление вывода и ввод до разделителя.

Эффект от использования символа > (больше) для перенаправления вывода в файл является деструктивным. Иными словами, команда

```
$ ls > file-list
```

уничтожит содержимое файла file-list, если этот файл ранее существовал, и создаст на его месте новый файл. Если вместо этого перенаправление будет сделано с помощью символов >>, то вывод будет дописан в конец указанного файла, при этом исходное содержимое файла не будет уничтожено. Например, команда

```
$ ls >> file-list
```

дописывает вывод команды ls в конец файла file-list.

Симметричная по виду запись перенаправления ввода (с помощью символов <<) используется для организации так называемого ввода до разделителя:

```
$ cat <<END
Hello, world!
END
Hello, world!
$
```

Здесь командный интерпретатор, встретив оператор перенаправления <<, запомнил последовательность символов после него (END) как разделитель потока ввода. Все последующие строки, вплоть до строки, содержащей только этот разделитель, были переданы на вход команды cat в виде потока ввода.

Следует иметь в виду, что перенаправление ввода и вывода и стыковка команд осуществляются командными оболочками, которые поддерживают использование символов >, >> , | и др. Сами команды специальным

образом эти символы не интерпретируют. Если нужно передать в команду один из этих символов в качестве параметра или использовать внутри передаваемой как параметр строки, то сделать это можно или «экранировав» одиночный спецсимвол с помощью символа обратного слеша (например, \<), или используя одинарные кавычки для выделения подстроки целиком.

Перенаправление потока вывода ошибок.

По-умолчанию операторы перенаправления > и >> изменяют передаваемый запускаемой программе файловый дескриптор с номером 1 – который соответствует потоку вывода. Возможно отдельно задать номер изменяемого файлового дескриптора, указав его перед операторами. Потоку вывода ошибок соответствует файловый дескриптор 2: т. е., например, для перенаправления вывода ошибок команды `mkdir` в `/dev/null` можно записать:

```
$ mkdir /etc/my-directory 2> /dev/null
```

Можно одновременно перенаправить и поток вывода, и поток ошибок:

```
$ ls -R /var/log/ 2>stderr >stdout
```

Здесь вывод команды `ls -R` (`-R` — рекурсивно по всем подкаталогам) выводится в файл `stdout`, а сообщения об ошибках – в файл `stderr`.

Также можно перенаправить стандартный поток ошибок в стандартный поток вывода – операторы перенаправления позволяют указать вместо имени файла номер файлового дескриптора в формате `&номер`:

```
$ ls -R /var/log/ 2>&1
```

При использовании одновременно перенаправления и стандартного потока вывода, и стандартного потока ошибок важен порядок операций:

```
$ ls -R /var/log/ 2>&1 >/dev/null
$ ls -R /var/log/ >/dev/null 2>&1
```

Первая команда присвоит файловому дескриптору потока ошибок значение файлового дескриптора потока вывода, и далее перенаправит поток вывода в `/dev/null`. В итоге сообщения об ошибках будут выводится в поток вывода (т. е. при запуске из терминала — на экран), а сам вывод команды будет перенаправляться в `/dev/null`.

Вторая команда сначала переопределит поток вывода, направив его в `/dev/null`, а потом присвоит потоку вывода ошибок значение файлового дескриптора потока вывода. В итоге, весь вывод команды – и стандартный, и сообщения об ошибках, - будет направлен в `/dev/null`.

Одновременное перенаправление в один и тот же файл и потока стандартного вывода, и потока ошибок встречается очень часто — для упрощения записи в ряде командных интерпретаторов, в т.ч. в Bash, есть дополнительный оператор перенаправления `&>` , переназначающий оба потока вывода сразу:

```
$ mkdir /etc/my-directory &> /dev/null
```

Узнать о результате выполнения команды при перенаправлении всего её вывода в устройство `/dev/null` можно, проанализировав код возврата.

Основы регулярных выражений.

Регулярные выражения (*англ.* regular expressions, сокращённо *regex*) — это система поиска фрагментов в тексте, основанная на специальной системе записи образцов для поиска. Образец (*англ.* pattern), задающий правило поиска, также называют шаблоном или маской.

Сейчас регулярные выражения используются многими текстовыми редакторами и утилитами для поиска и изменения текста на основе выбранных правил. Многие языки программирования имеют встроенную поддержку работы с регулярными выражениями, для других они доступны как внешние библиотеки. Набор утилит (включая редактор `sed` и фильтр `grep`), поставляемых в дистрибутивах *nix, одним из первых способствовал распространению регулярных выражений.

Регулярные выражения используются для сжатого описания некоторого множества строк с помощью шаблонов, без необходимости перечисления всех элементов этого множества. При составлении шаблонов используется специальный синтаксис, поддерживающий, обычно, следующие операции:

- Перечисление: вертикальная черта разделяет допустимые варианты. Например, «`one|two`» соответствует *one* или *two*.
- Группировка: круглые скобки используются для определения области действия и приоритета операторов. Например, шаблоны «`abd|acd`» и «`a(b|c)d`» описывают одно и то же множество: *abd* и *acd*.
- Квантификация: квантификатор после символа или группы символов определяет, сколько раз предшествующее выражение может встречаться. Например:
  - `{m,n}` — общее выражение, повторений может быть от `m` до `n` включительно.
  - `{m,}` — общее выражение, `m` и более повторений.
  - `{,n}` —общее выражение, не более `n` повторений.
  - `?` (вопросительный знак) означает `0` или `1` раз, то же самое, что и `{0,1}`. Например, «`colou?r`» соответствует и *color*, и *colour*.

- `*` (астериск) означает `0`, `1` или любое число раз (`{0,}`). Например, «`go*gle`» соответствует *ggle*, *gogle*, *google* и т.д.
- `+` (плюс) означает хотя бы `1` раз (`{1,}`). Например, «`go+gle`» соответствует *gogle*, *google* и т.д. (но не *ggle*).

Конкретный синтаксис регулярных выражений зависит от их программной реализации. Мы будем рассматривать синтаксис «базовых» регулярных выражений UNIX. Хотя он на данный момент и определён *POSIX* как устаревший, но до сих пор широко распространён из соображений обратной совместимости. Многие UNIX-утилиты используют такие регулярные выражения по умолчанию.

В этом синтаксисе большинство символов соответствуют сами себе («`a`» соответствует *a* и т.д.). Исключения из этого правила называются метасимволами:

`.`	Соответствует любому единичному символу.
`[ ]`	Соответствует любому единичному символу из числа заключённых в скобки. Символ – (дефис) интерпретируется буквально только в том случае, если он расположен непосредственно после открывающей или перед закрывающей скобкой: `[abc-]` или `[-abc]`. В противном случае он обозначает интервал символов. Например, `[abc]` соответствует *a*, *b* или *c*. `[0-9]` соответствует цифрам.
`[^ ]`	Соответствует единичному символу из числа тех, которых нет в скобках. Например, `[^abc]` соответствует любому символу, кроме *a*, *b* или *c*. `[^0-9]` соответствует любому символу, кроме цифр.
`^`	Используемое в начале регулярного выражения, соответствует началу строки текста.
`$`	Используемое в конце регулярного выражения, соответствует концу строки текста.
`\(\)`	Объявляет «отмеченное подвыражение», которое может быть использовано позже.
`\n`	`n` — цифра от `1` до `9`, соответствует *n*-му отмеченному подвыражению.
`*`	Астериск после выражения, соответствующего единичному символу, соответствует нулю или более копий этого выражения. Например, «`[xyz]*`» соответствует пустой строке, *x*, *y*, *zx*, *zyx*, и т.д.
`\{x,y\}`	Соответствует последнему блоку, встречающемуся не менее `x` и не более `y` раз. Например, «`a\{3,5\}`» соответствует *aaa*,

	aaaa или *aaaaa*.

При использовании диапазонов символов следует учитывать, что они могут зависеть от выбранных настроек локализации. Например, диапазон «`[b-e]`» означает символы от *b* до *e* включительно. В английском языке, где сортировка букв идёт по-порядку (...*XYZabcdefg*...), ему соответствует набор символов *b*,*c*,*d*,*e*. По правилам русского языка, сортировка тех же символов идёт в другом порядке (...*эЭюЮяЯaAbBcCdDeEfFgG*...), и тому же диапазону соответствуют символы *b*,*B*,*c*,*C*,*d*,*D*,*e*.

Для решения таких проблем в стандарте *POSIX* имеются объявления некоторых классов и категорий символов:

Класс	Диапазон для английского языка	Описание
`[:upper:]`	`[A-Z]`	Латинские буквы верхнего регистра.
`[:lower:]`	`[a-z]`	Латинские буквы нижнего регистра.
`[:alpha:]`	`[A-Za-z]`	Латинские буквы верхнего и нижнего регистра.
`[:alnum:]`	`[A-Za-z0-9]`	Цифры, латинские буквы верхнего и нижнего регистра.
`[:digit:]`	`[0-9]`	Цифры.
`[:xdigit:]`	`[0-9A-Fa-f]`	Шестнадцатеричные цифры.
`[:punct:]`	`[.,!?:…]`	Знаки пунктуации.
`[:blank:]`	`[ \t]`	Пробел и табуляция.
`[:space:]`	`[ \t\n\r\f\v]`	Символы пропуска.
`[:cntrl:]`	-	Символы управления.
`[:graph:]`	`[^\t\n\r\f\v]`	Символы печати.

Способ представить сами метасимволы — `.`, `-` `[` `]` и другие — в регулярных выражениях без интерпретации, т.е. в качестве простых (не специальных) символов — предварить их («экранировать») символом `\` (обратный слеш). Например, чтобы представить сам символ «точка» (просто точка, и ничего более), надо написать `\.` (обратный слеш, а за ним — точка). Чтобы представить символ открывающей квадратной скобки `[`, надо написать `\[` (обратный слеш, и следом за ним скобка `[`) и т.д. Сам метасимвол `\` (обратный слеш) тоже может быть экранирован, то есть

представлен как \\ (два обратных слеша), и тогда интерпретатор регулярных выражений воспримет его как простой символ обратного слеша \.

При составлении регулярных выражений следует также учитывать их две основные черты: они являются т.н. «ленивыми» и «жадными». Первое означает, что в строке, где есть несколько совпадений с шаблоном, шаблон совпадёт с первым из них. Например, регулярное выражение «*шаблон\(..\)*» для строки

в строке, где есть несколько совпадений с шаблоном, шаблон совпадёт с первым из них

вернёт в подвыражении \1 символы *ом*, соответствующие первому встретившемуся подходящему совпадению (*шаблоном*). Второе возможное место совпадения (*шаблон с*) рассмотрено не будет.

«Жадность» регулярных выражений заключается в том, что, при использовании квантификаторов * (астериск) и + (плюс), шаблон будет совпадать с максимально длинным из возможных вариантов. Для той же строки шаблон «*шаблон.*н*», означающий подстроку, начинающуюся с «*шаблон*», заканчивающуюся на «*н*» и с произвольным количеством (*) любых (.) символов между «*шаблон*» и «*н*», совпадёт с подстрокой

шаблоном, шаблон совпадёт с первым из н,

а не с более короткой

шаблоном, шаблон

Рассмотрим далее применение регулярных выражений на примерах использования утилит grep и sed.

Утилита grep.

Одной из программ, использующих регулярные выражения для работы с текстом, является утилита grep. Она читает текст из файла и выводит те строки, которые совпадают с заданным регулярным выражением. Общий формат вызова утилиты:

```
grep [options] PATTERN [FILE...]
```

где PATTERN — регулярное выражение, а FILE — один или несколько файлов, к содержимому которых будет применено это регулярное выражение.

Если файл не задан, то grep читает текст со стандартного ввода. С помощью опций (*англ.* options) можно управлять поведением grep, например. опция -v приводит к выводу всех строк, не совпадающих с заданным регулярным выражением.

Рассмотрим некоторые примеры использования grep и регулярных выражений. Как говорилось в предыдущей лабораторной работе, команда ls выводит список файлов в каталоге. Команда ls /bin выведет список файлов из каталога /bin. Вывод команда ls осуществляет в *stdout*.

Предположим, нас интересуют те программы (файлы) из /bin, которые содержат подстроку *zip*. Этой подстроке соответствует простейшее регулярное выражение «zip». Перенаправляем вывод из ls в grep и получаем:

```
$ ls /bin | grep 'zip'
bunzip2
bzip2
bzip2recover
gunzip
gzip
```

Здесь регулярное выражение заключено в одиночные кавычки '', которые указывают bash, что внутри них — обычная строка. Такой синтаксис позволяет использовать в регулярном выражении пробелы, и его разумно придерживаться во всех случаях (например, регулярное выражение 'a b' описывает шаблон для строк, содержащих последовательно *a*, пробел и *b*. Если этот шаблон указать grep без кавычек, т.е. grep a b, то командный интерпретатор, разобрав строку, вызовет grep с двумя параметрами, и grep будет искать строки с буквами *a* в файле b. При использовании кавычек командный интерпретатор будет считать выражение 'a b' одним параметром, и передаст его grep целиком, вместе с пробелом внутри).

Файлы из /bin, которые кончаются на *2*:

```
$ ls /bin | grep '2$'
bash2
bunzip2
bzip2
```

Файлы из /bin, которые начинаются на *b*:

```
$ ls /bin | grep '^b'
basename
bash
bash2
bunzip2
bzcat
bzip2
bzip2recover
```

Файлы из /bin, начинающиеся на *b* и содержащие в своём имени букву *a*:

```
$ ls /bin | grep '^b.*a'
basename
bash
bash2
bzcat
```

Здесь в регулярном выражении указано, что оно:

- должно совпадать с началом строки — `^`
- в начале строки должна быть буква *b* — `^b`
- дальше может быть любой символ — `^b.`
- и таких символов может быть сколько угодно — 0 или больше — `^b.*`
- а дальше должна быть буква *a* — `^b.*a`

Файлы из `/bin`, начинающиеся на *b* и содержащие в своём имени буквы *a*, *e* или *k*:

```
$ ls /bin | grep '^b.*[aek]'
basename
bash
bash2
bzcat
bzip2recover
```

Здесь используется описание набора символов — `[aek]`.

Рассмотрим более полезный пример.

На предыдущей лабораторной работе производилась настройка сервера `lighttpd`. Его конфигурационный файл — `/etc/lighttpd/lighttpd.conf`. Как было видно, в нём (как и в большинстве других конфигурационных файлов) содержится большое количество комментариев, как с поясняющим текстом, так и с примерами различных опций настройки. Предположим, нам нужно посмотреть текущую конфигурацию сервера. Однако посмотреть её простой командой `cat /etc/lighttpd/lighttpd.conf` неудобно: текст не помещается на экране. Мы можем, конечно, использовать команду `less` для прокрутки текста, но комментарии при этом всё равно будут мешать. Мы можем удалить их из файла, но тогда сложно будет что-либо изменять в нём в дальнейшем.

Проще отфильтровать ненужный текст непосредственно при выводе файла на экран.

Комментарии в `lighttpd.conf` начинаются с символа `#` (октоторп). Перед ним в начале строки может или не быть ничего, или быть один или несколько пробелов.

Таким образом, регулярное выражение для выделения строк с комментариями — «`^ *#`»: начало строки, ноль или несколько пробелов, и затем — *#*.

Кроме того, нас не очень интересуют просто пустые строки, в которых нет никакого текста. Такие строки можно описать выражением «`^$`»: начало строки, и сразу — её конец. Может быть и другой вариант: строка, состоящая из одних пробелов, которая также не несёт никакой информации. Таким образом, общее регулярное выражение приобретает вид «`^ *$`».

Итого, строкам комментариев соответствует выражение «^ *#», а пустым строкам — «^ *$». Как было отмечено ранее, фильтру grep можно приказать выводить строки, которые не совпадают с регулярным выражением, вызвав его с ключом -v.

Выводим файл lighttpd.conf в *stdout* и последовательно пропускаем вывод через два фильтра:

```
# cat /etc/lighttpd/lighttpd.conf | grep -v '^ *#' | grep -v '^ *$'
```

Этот вариант не очень эффективен, хотя и приносит желаемый результат. Можно избежать двух последовательных вызовов grep, объединив шаблоны. Видно, что они очень похожи: возможные пробелы в начале строки и или # (октоторп), или конец строки. Т.е. общий шаблон — «^ *(#|$)».

grep поддерживает несколько вариантов синтаксиса регулярных выражений и в варианте по умолчанию рассматривает круглые скобки как обычные символы. Поэтому надо или приказать grep'у рассматривать их как оператор выбора, экранировав скобки символом \ (обратный слеш), или переключить grep в режим работы с расширенным синтаксисом регулярных выражений, вызвав его с ключом -E, или использовать версию grep с включённой по умолчанию поддержкой расширенных регулярных выражений — egrep:

```
# cat /etc/lighttpd/lighttpd.conf | grep -v '^ *\(#\|$\)'
# cat /etc/lighttpd/lighttpd.conf | grep -E -v '^ *(#|$)'
# cat /etc/lighttpd/lighttpd.conf | egrep -v '^ *(#|$)'
```

Ну и наконец, нам не обязательно передавать файл lighttpd.conf на стандартный вход grep/egrep, эти утилиты могут сами прочитать файл с диска:

```
# egrep -v '^ *(#|$)' /etc/lighttpd/lighttpd.conf
```

Утилита sed.

Программа grep выполняет только поиск строк и выводит найденные результаты без изменений. Однако часто бывает необходимо не только найти какой-либо текст, но и изменить его. Для редактирования потока текста можно использовать утилиту sed (от *англ.* Stream EDitor, потоковый редактор). sed используется для выполнения основных преобразований текста, читаемого из файла или поступающего из стандартного потока ввода, и совершает одно действие над вводом за проход. Общий формат вызова sed:

```
sed [options] COMMAND [FILE...]
```

Из большого числа возможных команд sed мы рассмотрим только команду поиска и замены текста. Эта команда имеет вид s/PATTERN/EXPRESSION/ и осуществляет поиск в каждой из входящих строк текста регулярного

выражения PATTERN. Результаты совпадения заменяются на выражение EXPRESSION. Результирующий текст выводится в стандартный поток вывода.

Рассмотрим использование команды замены в sed на примерах.

В простейшем случае просто поменяем один фрагмент текста на другой:

```
$ ls -1 /var/cache
apt
fontconfig
man
$ ls /var/cache/ | sed 's/apt/APT/'
APT
fontconfig
man
```

В каталоге /var/cache есть несколько файлов, список их можно получить командной ls. Регулярное выражение «apt» совпадает с одной из строк вывода, и мы меняем совпадение на APT.

```
$ ls /var/cache/ | sed 's/a/A/'
Apt
fontconfig
mAn
```

В этом случае мы заменили в выводе ls букву *a* на *A*. sed применяет свои команды для каждой из строк вывода, поэтому в обеих строках, где была буква *a*, она была заменена.

Утилита uptime выдаёт определённую статистику по работе системы:

```
$ uptime
 07:48:42 up 27 days, 22:13,  1 user,  load average: 0.00, 0.00, 0.00
```

Для того, чтобы выделить из этой строки текущее число пользователей в системе, используем sed. Число пользователей — это одна или несколько цифр — «[0-9]\+», за которыми после пробела (или нескольких пробелов в общем случае) — «[0-9]\+ \+» следует слово *user* (или *users*). Нам интересно число пользователей — выберем его в подвыражении: «\([0-9]\+\) \+user». В начале строки идёт некоторый текст, отделённый от числа пользователей пробелом: «^.* \([0-9]\+\) \+user». Конец строки тоже может быть любой: «^.* \([0-9]\+\) \+user.*».

Данное выражение совпадает со всей строкой и выделяет в подстроку \1 число пользователей. Заменив целиком строку на \1, мы получим в результате только это число:

```
$ uptime | sed 's/^.* \([0-9]\+\) \+user.*/\1/'
1
```

Аналогично можно получить, например, время работы системы (подстроку вида *27 days, 22:13*):

```
$ uptime | sed 's/^.* up \+\(.\+\), \+[0-9]\+ \+user.*/\1/'
27 days, 22:13
```

Здесь мы отметили, что время работы системы начинается за словом *up*, а после него идёт число пользователей. Соответственно, требующееся регулярное выражение для помещения времени работы системы в подстроку можно описать как:

- любое число любых символов от начала строки, далее пробел и слово *up* — `^.* up`
- за которым следует через один или несколько пробелов время работы системы — `^.* up \+\(\)`
- само время работы системы может содержать фактически любые символы, в т.ч. пробелы, знаки пунктуации и пр. — `^.* up \+\(.\+\)`
- однако за ним через запятую и один или несколько пробелов — `^.* up \+\(.\+\), \+`
- следует количество пользователей (число, одна или несколько цифр) — `^.* up \+\(.\+\), \+[0-9]\+`
- и слово *user* (или *users*). Далее до конца строки может быть что угодно — `^.* up \+\(.\+\), \+[0-9]\+ \+user.*`

Отметим, что то же самое мы могли бы сделать и по-другому: просто удаляя из вывода ненужный нам текст. Например:

```
$ uptime | sed 's/user.*//'
 08:18:07 up 27 days, 22:43,  2
```

убирает весь текст от *user* включительно и до конца строки. Также убираем в полученном результате и всё в конце строки от запятой включительно:

```
$ uptime | sed 's/user.*//'| sed 's/,[^,]*$//'
 08:24:13 up 27 days, 22:49
```

Отметим, что более простой вариант без привязки к концу строки

```
$ uptime | sed 's/user.*//'| sed 's/,[^,]*//'
 08:24:18 up 27 days,  2
```

из-за «ленивости» регулярных выражений совпадёт с первым вхождением запятой (*, 22:43*), а ещё более простой вариант

```
$ uptime | sed 's/user.*//'| sed 's/,.*$//'
 08:25:11 up 27 days
```

из-за «жадности» будет совпадать с текстом от первой запятой до конца строки (*, 22:43, 2*).

Далее нам нужно удалить текст от начала строки до *up* включительно:

```
$ uptime | sed 's/user.*//'| sed 's/,[^,]*$//' | sed 's/^.*up \+//'
27 days, 22:54
```

и мы получаем требуемый результат. (Символ `\` (обратный слеш) в конце

строки здесь означает, что команда будет продолжена на следующей строке).

Утилита awk.

AWK — интерпретируемый скриптовый язык, предназначенный для обработки текстовой информации. Первая версия *AWK* была написана в 1977 году в AT&T Bell Laboratories и получила название по фамилиям своих разработчиков: Альфреда Ахо (Alfred V. Aho), Питера Вейнбергера (Peter J. Weinberger) и Брайана Кернигана (Brian W. Kernighan).

AWK рассматривает входной поток как набор записей, каждая из которых состоит из набора полей. По умолчанию для *AWK* записью является строка, а разделителями полей в строке — пробелы. Внутри программы на *AWK* значение поля можно получить как значение переменной `$1`, `$2`, `$3`, ... Переменная `$0` содержит в себе всю запись.

Программа на *AWK* имеет вид

```
BEGIN{ДЕЙСТВИЕ}
ШАБЛОН {ДЕЙСТВИЕ}
ШАБЛОН {ДЕЙСТВИЕ}
…
{ДЕЙСТВИЕ}
END{ДЕЙСТВИЕ}
```

Для каждой строки, совпадающей с шаблоном, выполняется указанное действие. Если шаблон не указан, то действие выполняется для всех строк. Опционально можно указать блоки кода `BEGIN{}` и `END{}`, которые будут выполняться один раз, до первой входной строки и после последней входной строки соответственно.

Шаблон — это регулярное выражение, из большого числа возможных действий мы рассмотрим только команду `print`.

Рассмотрим использование команды `awk` на примерах.

Список файлов с указанием их владельцев, прав, и даты последнего изменения можно получить командой `ls -l`. Он имеет вид:

```
$ ls -l /bin | head -n 5
total 5596
lrwxrwxrwx 1 root root       4 Feb 25 05:30 awk -> gawk
-rwxr-xr-x 1 root root   19064 Apr 20  2008 basename
-rwxr-xr-x 1 root root  549368 Mar 27  2008 bash
lrwxrwxrwx 1 root root       4 Feb 25 05:30 bash2 -> bash
```

Преобразуем этот список в формат

`<имя файла> <владелец>:<группа> <права>`

`awk` обрабатывает каждую строку списка отдельно, и самостоятельно разбивает её на поля по границам слов. Права файла — поле 1, владелец и группа — поля 3 и 4, имя файла — поле 9. Тогда:

```
$ ls -l /bin | awk '{print $9,$3":"$4,$1;}' | head
 : total
awk root:root lrwxrwxrwx
basename root:root -rwxr-xr-x
bash root:root -rwxr-xr-x
bash2 root:root lrwxrwxrwx
bunzip2 root:root lrwxrwxrwx
bzcat root:root lrwxrwxrwx
bzip2 root:root -rwxr-xr-x
bzip2recover root:root -rwxr-xr-x
cat root:root -rwxr-xr-x
```

Можно отфильтровать список и вывести только файлы. Для файлов первый символ поля прав — - (дефис). Для форматирования вывода разделим выводящиеся значения символами табуляции (код символа `\t`). С учётом этого получаем:

```
$ ls -l /bin | awk '/^-/ {print $9"\t->\t"$3":"$4"\t"$1;}' | head
basename        ->      root:root       -rwxr-xr-x
bash    ->      root:root       -rwxr-xr-x
bzip2   ->      root:root       -rwxr-xr-x
bzip2recover    ->      root:root       -rwxr-xr-x
cat     ->      root:root       -rwxr-xr-x
chgrp   ->      root:root       -rwxr-xr-x
chmod   ->      root:root       -rwxr-xr-x
chown   ->      root:root       -rwxr-xr-x
clock_unsynced  ->      root:root       -rwxr-xr-x
cp      ->      root:root       -rwxr-xr-x
```

Создание скриптов.

До сих пор нами рассматривался запуск программ из командной строки оболочки. Однако для повторяющихся последовательностей команд это неудобно. В таких случаях можно сохранить последовательность команд в файл и запускать их не из командной строки, а из такого файла. Обычно такие файлы с записанными командами называют скриптами.

В простейшем случае, скрипт можно создать, например, так:

```
$ echo "ls | grep script" > script
$ cat script
ls | grep script
$ sh script
script
```

Здесь мы создали текстовый файл, содержащий команды `ls` и `grep`, и далее выполнили эти команды, вызвав интерпретатор команд и передав ему в качестве аргумента имя скрипта. Интерпретатор команд, получив в качестве аргумента имя файла, считал из него команды и выполнил их.

Такой способ запуска скриптов не очень удобен. Он отличается от вызова команд системы: здесь требуется в командной строке указывать имя интерпретатора команд и, в общем случае, полный путь к выполняемому скрипту, в то время как для скомпилированных команд системы достаточно ввести имя самой команды. Кроме того, для операционных систем *nix

существует несколько альтернативных командных интерпретаторов с различным синтаксисом команд. Существует и большое количество различных интерпретирующих языков программирования, программы для которых также оформляются в виде скриптов и запускаются с помощью соответствующих программ-интерпретаторов. Таким образом, требуется способ указать системе, каким именно интерпретатором следует выполнять тот или иной скрипт.

Имя программы, которая должна интерпретировать записанную в текстовый файл (скрипт) последовательность команд, можно указать в самом скрипте. Это делается с помощью специальным образом оформленной первой строки скрипта, которая обычно выглядит примерно как

```
#!/bin/bash
```

Первая строка состоит из двух символов `#!` (октоторп и восклицательный знак) и следующим за ними полном пути к программе, которая будет обрабатывать данный скрипт. В данном случае это интерпретатор команд `bash`. Как правило, интерпретируемые языки программирования (и командный интерпретатор в частности) используют символ `#` (октоторп) для выделения комментариев, т. е. интерпретировать подобным образом оформленную строку они не будут.

Как рассматривалось в предыдущей лабораторной работе, в операционных системах *nix существуют права доступа к файлам. Если для файла задано право его выполнения, то интерпретатор команд откроет его и прочитает несколько первых символов файла. Если там обнаружится начало скомпилированной программы, то она будет запущена, если же там обнаружится последовательность символов `#!`, то будет запущен указанный после неё интерпретатор, которому будет передано в качестве аргумента имя файла.

Итого:

```
$ echo '#!/bin/bash' > script
$ echo 'ls | grep script' >> script
$ chmod a+x script
$ cat script
#!/bin/bash
ls | grep script
$ ls -l script
-rwxr-xr-x 1 student student 29 Map 20 09:35 script
$ ./script
script
```

Здесь мы создали путём вызова двух команд `echo` файл (обратите внимание, что во второй команде мы дописали строку в имеющий файл), задали этому файлу право на выполнение, проверили результат (выведя файл через `cat` и проверив права на него через `ls -l`) и запустили его на выполнение.

Отметим, что командный интерпретатор ищет выполняемые файлы в определённых каталогах: `/bin`, `/usr/bin` и т.п. Для запуска программы из

нестандартного каталога требуется указывать путь к ней, т.е., в данном случае, запустить программу как `script` нельзя — вместо созданного нами скрипта командный интерпретатор запустит стандартную утилиту `script` из `/usr/bin`.

Часто простого последовательного выполнения недостаточно: для эффективного программирования требуются переменные, условное выполнение команд и т.п. Командный интерпретатор имеет собственный язык, который по своим возможностям приближается к высокоуровневым языкам программирования. Этот язык позволяет создавать программы (*shell*-файлы, *shell*-скрипты), которые могут включать операторы языка и команды UNIX.

Такие файлы не требуют компиляции и выполняются в режиме интерпретации, но они, как отмечалось ранее, должны обладать правом на исполнение (устанавливается с помощью команды `chmod`).

Скрипту могут быть переданы аргументы при запуске. Каждому из первых девяти аргументов ставится в соответствие позиционный параметр от `$1` до `$9` (`$0` — имя самого скрипта), и по этим именам к ним можно обращаться из текста скрипта.

Прежде чем начать рассмотрение некоторых операторов *shell*, следует обратить внимание на использование в командах некоторых символов.

- $ (знак доллара) – используется для подстановки в строку значения переменной, имя которой указывается сразу за ним (`$VAR`).
- `` (обратные апострофы) — служат выполнения команды, заключённой между ними, и подстановки в строку вывода этой команды.
- \ (обратный слеш) — знак отмены специального значения («экранирования») следующего за ним символа, такого как $ или `. Будучи последним символом в строке, обратный слэш экранирует символ перевода строки и позволяет разбивать запись команд с многочисленными и длинными аргументами на несколько строк
- "" (двойные кавычки) — используются для обрамления текста, внутри которого командная оболочка выполняет поиск и интерпретацию специальных символов.
- '' (одинарные кавычки или апострофы) — используются для обрамления текста, передаваемого как единый аргумент команды или присваиваемого переменной без интерпретирования в нём специальных символов.

Кроме того, для удобства работы с файлами почти все командные интерпретаторы интерпретируют символы `?` (знак вопроса) и `*` (астериск), используя их как шаблоны имен файлов (т.н. метасимволы):

- `?` — один любой символ;
- `*` — произвольное количество любых символов.

Например, `*.c` обозначает все файлы с расширением `c`, `pr???.*` обозначает файлы, имена которых начинаются с `pr`, содержат пять символов и имеют любое расширение.

Переменные языка shell.

Язык *shell* позволяет работать с переменными без предварительного объявления. Имена переменных начинаются с латинской буквы и могут содержать латинские буквы, цифры и символ подчеркивания. Обращение к переменным начинается со знака `$` (знак доллара).

Имеется большое количество уже определённых переменных — т.н. переменных окружения. Их полный список можно получить командой `set`. Переменные окружения используются для настройки различных параметров окружения пользователя, например, в переменной `TMP` задаётся каталог для временных файлов, используемый рядом программ:

```
$ echo $TMP
/tmp/.private/student
$ ls $TMP
mc-student
```

Переопределить (в т.ч. случайно) такие системные переменные можно, но стоит учесть, что это может привести к нежелательным последствиям.

Для разных пользователей могут быть разные наборы переменных окружения с разными значениями. Например, как говорилось ранее, командный интерпретатор ищет выполняемые файлы в определённых каталогах: `/bin`, `/usr/bin` и т.п. Перечень этих каталогов командный интерпретатор берёт из переменной окружения `$PATH` . Для суперпользователя в этой переменной, помимо каталогов с программами пользователя, также указываются каталоги системных программ - `/sbin`, `/usr/sbin`. Или, для обычного пользователя в нескольких переменных окружения с именами вида LC_* задаются настройки локали — с учётом родного языка пользователя. Для суперпользователя используется английская локаль – для избежания проблем с поведением регулярных выражений в системных скриптах.

Как говорилось в предыдущей лабораторной работе, для повышения привилегий пользователя до уровня администратора системы требуется использовать команду '`su -l`' — с ключом '`-l`' . Данный ключ обеспечивает задание переменных окружения запускаемого от имени суперпользователя интерпретатора команд из настроек суперпользователя — без этого ключа остаются переменные окружения обычного пользователя. Как следствие, командный интерпретатор после этого не сможет найти системные программы, будет записывать временные файлы администратора в каталог

обычного пользователя, и т.п.

Оператор присваивания.

Присвоение значений переменным осуществляется с помощью оператора = (знак равенства). Пробелов между именем переменной, = и значением быть не должно. Например:

```
$ A=5
$ B=пять
$ C=$A+$B
$ echo A
A
$ echo B=$B
B=пять
$ echo C=$C
C=5+пять
```

Как мы видим, интерпретатор команд все переменные рассматривает как строки. Однако есть возможность и вычисления арифметических выражений — через внешние программы.

Вычисление выражений.

Вычисление выражений осуществляется с помощью команды `expr` и арифметических и логических операторов:

```
$ a=5 b=12
$ a=`expr $a + 4`
$ d=`expr $b - $a`
$ echo $a $b $d $A
9 12 3 5
```

Для `expr` аргументы и операции обязательно разделяются пробелами (они должны передаться команде как отдельные параметры). Кроме того, мы видим, что имена переменных чувствительны к регистру, `a` и `A` — разные переменные.

Команда `expr` позволяет производить операции только над целочисленными значениями. Для выполнения вычислений с числами с фиксированной точностью или с вещественными значениями можно использовать другие команды (например, калькуляторы `dc` или `bc`) – хотя, в целом, язык *shell* не предназначен для решения вычислительных задач.

Условные выражения.

Ветвление вычислительного процесса осуществляется с помощью оператора `if`:

```
if список_команд1; then
     список_команд2
[else
     список_команд3]
fi
```

(В квадратных скобках указывается необязательная часть команды.)

Список_команд — это одна или несколько команд, для задания пустого списка используется : (двоеточие). Список_команд1 передает оператору if код возврата последней команды из списка. Если код равен 0, то выполняются команды из списка_команд2, таким образом нулевой код возврата эквивалентен значению «истина». В противном случае выполняются команды из списка_команд3, если он указан.

Проверка условия может осуществляется с помощью команды test. Аргументами этой команды могут быть имена файлов, числовые и нечисловые строки. Она используется в следующих режимах:

- Проверка файлов: `test -ключ имя_файла`

  Ключи: `-r` файл существует и доступен для чтения;

  `-w` файл существует и доступен для записи;

  `-x` файл существует и доступен для исполнения;

  `-f` файл существует и является обычным файлом (т. е. не каталогом, не файлом устройства и т.п.);

  `-s` файл существует, является обычным файлом и не пуст, т. е. его размер больше 0 байт;

  `-d` файл существует и является каталогом.

- *Сравнение чисел:* `test число1 -ключ число2`

  Ключи: `-eq` равно;

  `-ne` не равно;

  `-lt` меньше;

  `-le` меньше или равно;

  `-gt` больше

  `-ge` больше или равно.

- Сравнение строк: `test [строка1] выражение строка2`

  `[-n] строка` строка не пуста;

  `-z строка` строка пуста;

  `строка1 = строка2` строки равны;

  `строка1 != строка2` строки не равны.

В качестве альтернативой записи test можно использовать команду [(открывающая квадратная скобка), при этом, например, для проверки существования файла вместо

```
$ if test -f /bin/bash; then echo 'bash найден!'; fi
bash найден!
```

можно использовать более аккуратно выглядящую конструкцию

```
$ if [ -f /bin/bash ]; then echo 'bash найден!'; fi
bash найден!
```

Построение циклов.

В языке командного интерпретатора существует три типа циклов: `while`, `until` и `for`.

Цикл `while`:

```
while список_команд1; do
     список_команд2
done
```

В условии учитывается код возврата последней выполненной команды из `списка_команд1`, при этом 0 интерпретируется как «истина».

Цикл `until`:

```
until список_команд1; do
     список_команд2{;|перевод строки}
done
```

Проверка условия выполняется перед выполнением цикла. Учитывается код возврата последней выполненной команды из `списка_команд1`, при этом цикл выполняется до тех пор, пока код возврата не примет значение «истина», т. е. будет равным нулю.

Цикл `for`:

```
for переменная [in список_значений]; do
     список_команд
done
```

Переменной присваивается значение очередного слова из `списка_значений`, и для этого значения выполняется `список_команд`. Количество итераций равно количеству цепочек символов в `списке_значений`, разделённых пробелами. Если ключевое слово `in` и `список_значений` опущены как необязательные, то переменной поочередно присваиваются значения параметров, переданных при запуске программы-скрипта. В качестве передаваемых параметров можно использовать шаблоны имён файлов, тогда интерпретатор превращает эти шаблоны в список имён файлов, удовлетворяющих шаблону.

Например,

```
$ A=1; for i in `ls /bin | grep '^b'`; do
> echo "$A  :$i"
> A=`expr $A + 1`
```

```
> done
1  :basename
2  :bash
3  :bash2
4  :bunzip2
5  :bzcat
6  :bzip2
7  :bzip2recover
```

Здесь мы получили список файлов из `/bin` (`ls /bin`), отфильтровали из него файлы, начинающиеся на `b` (`ls /bin | grep '^b'`), и передали полученный список в качестве параметра оператору цикла `for`. В самом цикле мы вывели текущее значение переменной цикла и номер записи.

Код возврата.

Как говорилось ранее, каждая программа по результату своего выполнения возвращает в операционную систему определённый код возврата. Нулевое значение подразумевает успешное выполнение программы, ненулевое – наличие каких-либо возникших ошибок. Каким ошибка именно соответствует ненулевое значение – определяется самой программой.

Скрипты командного интерпретатора также возвращают коды возврата. По-умолчанию, это код возврата последней выполненной команды скрипта. Есть возможность завершить скрипт с заданным кодом возврата – для этого можно использовать команду `exit`.

Например, такой скрипт проверит возможность выполнение команды '`su`' под текущим пользователем, в случае недостаточности прав – выведет сообщение об ошибке в поток вывода ошибок и завершится с кодом возврата 1:

```
#!/bin/bash
if [ -x /bin/su ]; then
  echo "Нет прав на выполнение команды su" >&2
  exit 1
fi
/bin/su -c 'ls -l'
```

При возможности запуска '`su`' будет запрошено выполнение под учётной записью суперпользователя команды '`ls /root`'. Код возврата скрипта в этом случае совпадёт с кодом возврата команды '`su`', например, если будет неверно введён пароль суперпользователя – код возврата будет содержать ошибку.

При написании скриптов на языке командного интерпретатора, так же как и в программах на других языках программирования, хорошим тоном является проверка успешности выполнения действий, которые могут быть завершены с ошибками, и обработка таких ошибок. Это касается операций с файлами, вызовов внешних программ и т. п.

Для скриптов на языке командного интерпретатора есть возможность указать оболочке автоматически прекратить работу скрипта при возникновении ошибки при выполнении любой команды. Данный режим включается командой 'set -e' в начале скрипта.

Например, скрипт

```
#!/bin/bash
# Clear /root/tmp/log
cd /root/tmp
rm -r log/
mkdir log/
```

опасен — при запуске не от суперпользователя команда cd не сможет перейти в каталог /root/tmp, и может быть удалён каталог log со всем содержимым в текущем каталоге.

Можно явно проверить результат выполнение команды cd /root/tmp и завершить выполнение скрипта с выдачей кода ошибки:

```
#!/bin/bash
# Clear /root/tmp/log
cd /root/tmp || exit 1
rm -r log/
mkdir log/
```

Другой вариант – использовать 'set -e':

```
#!/bin/bash
# Clear /root/tmp/log
set -e
cd /root/tmp
rm -r log/ ||:
mkdir log/
```

Здесь скрипт будет завершён автоматически при невозможности выполнить смену каталога. При этом, если в /root/tmp нет подкаталога log/, то команда рекурсивного удаления подкаталога также завершится в ошибкой. Чтобы при этом не было прервано выполнение скрипта, этот код ошибки надо обработать — в данном случае проигнорировать. Последовательность символов '||:' интерпретируется как оператор «или» и пустой оператор ':', всегда возвращающий нулевое значение.

Установка, удаление и обновление программных компонентов в системе.

Для операционной системы Linux доступно огромное количество программного обеспечения. Основная масса этого ПО доступна под свободными лицензиями, и с сайтов разработчиков можно загрузить архивы с исходными текстами программ. Однако пригодные для запуска скомпилированные бинарные файлы разработчиками программ или предоставляются для очень ограниченного круга дистрибутивов, или не предоставляются совсем.

Хотя обычно процесс сборки программ из исходных текстов автоматизирован, включая нахождение необходимых библиотек и заголовочных файлов на конкретной системе, всё-таки данное занятие требует достаточно глубоких знаний. При наличии нескольких систем собирать и устанавливать программу придётся вручную на каждой по-отдельности, т.к. набор и версии установленных системных библиотек на разных системах могут отличаться. Кроме того, перед использованием программу мало собрать — её надо установить в соответствующие данному дистрибутиву каталоги, внести изменения в её настройки (и, возможно, в настройки других программ) в соответствии с принятыми в конкретном дистрибутиве соглашениями, убедиться в правильности прав на установленные файлы, при необходимости создать псевдопользователей, добавить скрипты для запуска программы и т.п. Дополнительные проблемы возникают при появлении новых версий установленного на системе программного обеспечения — так, при обновлении какой-либо системной библиотеки, требуется заново собрать и установить не только её, но и все использующие её программы.

Для решения этой проблемы были разработаны системы, позволяющие компилировать программы и распространять результат в виде пакетов — архивов специального формата. В заголовке пакета указывается информация о названии программы, краткое её описание, номер версии программы и версии самого пакета. Для каждого пакета указываются его зависимости от других пакетов — т.е. пакеты с теми программами и библиотеками, которые используются программой в пакете и нужны ей для работы.

На данный момент существуют и широко используются две системы сборки пакетов программ. Первая — *Debian package management system (dpkg)*, использующая для установки и обновления пакетов программу `dpkg`, работающую с форматом `.deb` . Данная система используется в проекте Debian и вышедших из него дистрибутивах семейства Ubuntu.

Вторая система — *RPM Package Manager* (изначально *Red Hat Package Manager*), разработанная компанией Red Hat. В этой системе для управления пакетами в формате *RPM* используется одноимённая утилита. Пакеты в формате *RPM*, в частности, используются в дистрибутивах Red Hat, SUSE, Mandriva, в проектах Fedora Core, PLD, в отечественных проектах ASP Linux и ALT Linux.

Пакеты разделяются на две категории — пакеты с исходными текстами и пакеты с исполняемым кодом (бинарные пакеты). В первых содержится исходный код программы и инструкции для системы управления пакетами по сборке из этого исходного кода пакетов с исполняемым кодом. В системе *RPM* такие пакеты имеют расширение `.src.rpm`. Пакеты с исполняемым кодом содержат скомпилированные программы и предназначены для установки этих программ в системе.

Исполняемый код, очевидно, зависит от архитектуры системы. Одному пакету с исходным кодом, таким образом, соответствует несколько пакетов с двоичным. На данный момент в составе ALT Linux поддерживаются архитектуры 32-битных процессоров с командами Intel Pentium (с расширением

файлов пакетов .i586.rpm), 64-битных процессоров Intel и AMD (с расширением файлов пакетов .x86_64.rpm), архитектуры ARMv5 (с расширением файлов пакетов .arm.rpm) и ARMv7 (с расширением файлов пакетов .armh.rpm), а также архитектуры RISC-V, MIPS, Эльбрус v3 и Эльбрус v4. Кроме того, существуют программы, являющиеся архитектурно-независимыми — например, написанные на интерпретируемых языках. Для того, чтобы избежать дублирования пакетов с такими программами для каждой из архитектур, они упаковываются в пакеты с архитектурой *noarch*. Таким образом, для систем с 32-битными процессорами с системой команд x86 нужно использовать пакеты .i586.rpm и .noarch.rpm, для 64-битных систем — .x86_64.rpm и .noarch.rpm.

Управление пакетами в системе *RPM* осуществляется с помощью команды rpm. Полный формат её вызова можно посмотреть в соответствующем руководстве (man rpm).

Используя команду rpm, можно получать информацию о пакетах, устанавливать, обновлять и удалять их, а также собирать пакеты с исходным кодом и компилировать их в бинарные пакеты. Для получения информации о пакетах предназначается ключ -q.

rpm -q <имя пакета> выведет краткую информацию о версии и релизе установленного пакета:

```
$ rpm -q rpm
rpm-4.0.4-alt77.M40.1
```

Здесь 4.0.4 — версия программы *RPM*, а alt77.M40.1 — релиз пакета.

Более подробную информацию можно получить, добавив ключ -i:

```
$ rpm -qi rpm
Name        : rpm                               Relocations: (not relocateable)
Version     : 4.0.4                                  Vendor: ALT Linux Team
Release     : alt77.M40.1                        Build Date: Втр 28 Авг 2007
21:15:45
Install date: Пнд 22 Окт 2007 00:35:45      Build Host:
ldv.hasher.altlinux.org
Group       : System/Configuration/Packaging   Source RPM: rpm-4.0.4-
alt77.M40.1.src.rpm
Size        : 406252                              License: GPL
Packager    : Dmitry V. Levin <ldv@altlinux.org>
URL         : http://www.rpm.org/
Summary     : The RPM package management system
Description :
The RPM Package Manager (RPM) is a powerful command line driven
package management system capable of installing, uninstalling,
verifying, querying, and updating software packages.  Each software
package consists of an archive of files along with information about
the package like its version, a description, etc.
```

Как видно, в пакете указывается, помимо его версии и релиза, также время компиляции пакета, время его установки в системе, лицензия, URL

сайта разработчика программы, краткое и полное описание программы в пакете.

Список пакетов, установленных в системе, можно получить с помощью команды:

```
$ rpm -qa
vzquota-3.0.9-alt1
mktemp-1.5-alt2
libshhopt-1.1.7-alt4
....
alterator-users-8.0-alt2
kdebase-libkonq-3.5.8-alt11.M40.1
libxine-1.1.10.1-alt1.M40.1
```

Для каждого файла в системе, установленного из пакета, в кэше `rpm` хранится соответствующая запись. Всегда можно посмотреть, какому пакету принадлежит тот или иной файл или каталог. Для этого используется команда `rpm -qf`:

```
$ rpm -qf /usr/bin/rpm
rpm-4.0.4-alt77.M40.1
$ rpm -qf /bin/cp
coreutils-5.97-alt6
$ rpm -qf /home
filesystem-2.3.2-alt1
$ rpm -qf /root
filesystem-2.3.2-alt1
$ rpm -qf /home/student
предупреждение: файл /home/student не принадлежит ни одному из пакетов
```

Как видно, домашний каталог суперпользователя `/root` был создан в системе при установке пакета `filesystem`, а домашний каталог пользователя `student`, разумеется, ни одному из пакетов не принадлежит.

Для установки, обновления и удаления пакетов команду `rpm` использовать не очень удобно. Дело в том, что `rpm` (как и `dpkg`) предназначена для работы с одиночными пакетами. Однако пакеты, как правило, зависят от других пакетов, и для установки пакета требуется также установка и всех тех пакетов, от которых он зависит. Такие зависимости образуют цепочки, и вручную определить весь список необходимых пакетов сложно. Поэтому поверх систем *RPM* и *dpkg* используются системы управления репозиториями пакетов.

Под репозиторием понимается набор пакетов программ, предназначенный для конкретного дистрибутива и связанный общими зависимостями.

Репозитории пакетов существуют практически для всех крупных дистрибутивов. Они подразделяются на официальные, на базе которых и выпускаются дистрибутивы, и неофициальные, поддерживаемые конкретными разработчиками и/или группами разработчиков. Официальные репозитории крупных дистрибутивов насчитывают десятки тысяч пакетов.

Для уже выпущенных версий дистрибутивов изменения в их репозитории не вносятся, а новые версии программ, в т.ч. с исправлением выявленных ошибок, включаются в отдельные репозитории обновлений.

Одной из систем, позволяющих работать с репозиториями пакетов, является *APT* (*Advanced packaging tool*). Изначально разработанная для *dpkg*, в настоящее время она также может работать и с репозиториями пакетов *RPM*.

Перед использованием системы *APT* ей следует указать, какие репозитории она должна использовать. Эти настройки хранятся в каталоге `/etc/apt/`, в файле `/etc/apt/sources.list` и в файлах в каталоге `/etc/apt/sources.list.d/`. Запись о репозитории выглядит следующим образом:

```
 rpm [p9] ftp://ftp.altlinux.org/pub/distributions/ALTLinux/p9/branch x86_64
classic
 rpm [p9] ftp://ftp.altlinux.org/pub/distributions/ALTLinux/p9/branch noarch
classic
```

`rpm` указывает на тип репозитория. Для систем ALT Linux других значений в этом поле быть не должно. В квадратных скобках указывается имя открытого ключа, которым подписан репозиторий. Если цифровая подпись репозитория не будет соответствовать ключу, *APT* откажется работать с таким репозиторием. Далее указан URL самого репозитория. Как видно, в данном случае репозиторий доступен по протоколу FTP и размещён на сервере ftp.altlinux.org в каталоге `/pub/distribution/ALTLinux/p9/branch` . В четвёртом поле указывается архитектура пакетов в репозитории. В данном примере используются пакеты для 64-битных систем и архитектурно-независимые пакеты. Для 32-битных систем вместо `x64_86` должно указываться `i586`, для систем на архитектуре Эльбрус — `e2kv3` или `e2kv4`, и т.п. Последнее поле — используемый набор пакетов в репозитории.

Строки, начинающие с `#` — комментарии. Т.е. указанные в них репозитории не используются. Если их надо подключить, то символ `#` (октоторп) следует удалить.

В системе обычно используются удалённые репозитории, размещённые где-либо в Internet. Хотя можно разместить репозиторий локально в самой системе (тогда URL с путями к нему будут начинаться с file:///), чаще всего это нецелесообразно. Репозитории занимают довольно много места, причём значительная часть файлов в них для конкретной системы не нужна: в репозитории содержатся как пакеты с исходными текстами программ для самостоятельной сборки, так и пакеты для разных архитектур, из которых требуется только одна. Например, по состоянию на сентябрь 2020 г. приведённый выше репозиторий занимал порядка 322 GiB, из них около 64 GiB занимали пакеты с исходными кодами, по 44 GiB — пакеты для архитектур ARMv7 и MIPS, 44 GiB — для i586, 50 GiB — для x86_64, 33 GiB – архитектурно-независимые пакеты. Кроме того, пакеты в репозитории постоянно

обновляются (где-то 10-15 GiB в неделю для указанного репозитория). На конкретной же системе установлены только некоторые из пакетов в репозитории, и регулярно скачивать из Internet все изменения просто не нужно. Система *APT* поддерживает работу с удалёнными репозиториями и позволяет минимизировать сетевой трафик.

Для того, чтобы система *APT* узнала текущее состояние репозитория и список доступных пакетов в нём, требуется обновить её локальный кэш списка пакетов. Это делается командной `apt-get update`. В случае, если какие-либо репозитории недоступны, будут выведены сообщения об ошибках. Примерный вид работы `apt-get update` выглядит следующим образом:

```
# apt-get update
Get:1 ftp://ftp-distr x86_64 release [730B]
Get:2 ftp://ftp-distr noarch release [728B]
Fetched 1458B in 0s (13.5kB/s)
Get:1 ftp://ftp-distr x86_64/classic pkglist [2081kB]
Hit ftp://ftp-distr x86_64/classic release
Get:2 ftp://ftp-distr noarch/classic pkglist [942kB]
Hit ftp://ftp-distr noarch/classic release
Fetched 3023kB in 1s (1906kB/s)
Reading Package Lists... Done
Building Dependency Tree... Done
```

В данном случае система *APT* успешно обновила список пакетов.

Стоит обратить внимание на то, что выполнение операций по установке и обновлению пакетов в системе — это задача системного администратора. Поэтому `apt-get` должна вызываться с привилегиями суперпользователя.

Для обновления уже установленных пакетов в системе используется команда `apt-get upgrade`:

```
# apt-get upgrade
Reading Package Lists... Done
Building Dependency Tree... Done
The following packages will be upgraded
  libpcre3 shadow-convert shadow-utils startup
The following packages have been kept back
  pam0_passwdqc
4 upgraded, 0 newly installed, 0 removed and 1 not upgraded.
Need to get 564kB of archives.
After unpacking 4266B of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 ftp://ftp-distr x86_64/classic shadow-convert 1:4.0.4.1-alt8 [53.6kB]
Get:2 ftp://ftp-distr x86_64/classic shadow-utils 1:4.0.4.1-alt8 [351kB]
Get:3 ftp://ftp-distr noarch/classic startup 0.9.8.18-alt1 [33.6kB]
Get:4 ftp://ftp-distr x86_64/classic libpcre3 7.6-alt1 [126kB]
Fetched 564kB in 0s (2385kB/s)
Committing changes...
Preparing...        ########################################### [100%]
1: shadow-convert   ###########################################  [ 25%]
....
4: libpcre3         ########################################### [100%]
Done.
```

В данном случае было найдено 5 устаревших пакетов. При выполнении операции `upgrade` система *APT* не устанавливает новые и не удаляет из системы старые пакеты. Поэтому обновить пакет `pam0_passwdqc` она не могла, и предложила обновить только 4 пакета из 5. Подготовив список пакетов, команда `apt-get` задала пользователю вопрос о продолжении операции: '`Do you want to continue? [Y/n]`'. Получив утвердительный ответ (`y`), `apt-get` получила новые версии пакетов и установила их в системе.

Для обновления пакетов, у которых изменились зависимости, служит команда `apt-get dist-upgrade`:

```
# apt-get dist-upgrade
Reading Package Lists... Done
Building Dependency Tree... Done
Calculating Upgrade... Done
The following packages will be upgraded
  pam0_passwdqc
The following NEW packages will be installed:
  libpasswdqc passwdqc-control
1 upgraded, 2 newly installed, 0 removed and 0 not upgraded.
Need to get 52.6kB of archives.
After unpacking 7100B of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 ftp://ftp-distr x86_64/classic passwdqc-control 1.1.0-alt0.4 [6583B]
Get:2 ftp://ftp-distr x86_64/classic libpasswdqc 1.1.0-alt0.4 [30.3kB]
Get:3 ftp://ftp-distr x86_64/classic pam0_passwdqc 1.1.0-alt0.4 [15.7kB]
Fetched 52.6kB in 0s (462kB/s)
Committing changes...
Preparing...              ########################################### [100%]
1: passwdqc-control       ########################################### [100%]
....
3: pam0_passwdqc          ########################################### [100%]
Done.
```

В данном случае у новой версии пакета `pam0_passwdqc` появились зависимости на два новых пакета, которые команда `apt-get` и предложила установить.

Для установки программы используется команда `apt-get install`. В качестве аргумента ей передаются имена пакетов, которые нужно установить. `apt-get` определяет зависимости пакетов и выдаёт полный список всех пакетов, которые будут установлены в системе:

```
# apt-get install rrd-perl
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  libart_lgpl libfreetype libpng12 librrd
The following NEW packages will be installed:
  libart_lgpl libfreetype libpng12 librrd rrd-perl
0 upgraded, 5 newly installed, 0 removed and 0 not upgraded.
Need to get 774kB of archives.
After unpacking 1563kB of additional disk space will be used.
Do you want to continue? [Y/n] n
Abort.
```

В данном случае была запрошена установка пакета `rrd-perl`. Этот пакет зависит от библиотеки `librrd`, которая, в свою очередь, использует библиотеки `libpng12` и др. Всего установка пакета потребовала бы установку дополнительно ещё четырёх — что и предложила сделать `apt-get`. Получив отрицательный ответ на вопрос о продолжении операции, `apt-get` отменила её.

Для удаления пакетов используется команда `apt-get remove`. Ей также передаётся список пакетов. Если от удаляемого пакета зависят какие-либо ещё из установленных в системе, `apt-get` предложит удалить их все. При удалении пакетов `apt-get` всегда запрашивает подтверждение операции. Списки удаляемых пакетов следует внимательно просматривать во избежание нежелательных последствий. Например, команда

```
# apt-get remove openssh-server
Reading Package Lists... Done
Building Dependency Tree... Done
The following packages will be REMOVED:
  openssh-server ve-basic
0 upgraded, 0 newly installed, 2 removed and 0 not upgraded.
Need to get 0B of archives.
After unpacking 560kB disk space will be freed.
Do you want to continue? [Y/n] n
Abort.
```

удалит сервер *SSH*. После её выполнения удалённо зайти в систему уже не получится.

В ряде случаев удаляемый пакет критически необходим для системы:

```
# apt-get remove filesystem
Reading Package Lists... Done
Building Dependency Tree... Done
The following packages will be REMOVED:
  SysVinit alternatives apt apt-conf-sisyphus basesystem bash bzip2 bzlib
....
   vitmp vixie-cron zlib
WARNING: The following essential packages will be removed
This should NOT be done unless you know exactly what you are doing!
  apt sed (due to apt) libapt (due to apt) rpm (due to apt)
....
  vitmp (due to basesystem) mktemp (due to basesystem)
0 upgraded, 0 newly installed, 145 removed and 0 not upgraded.
Need to get 0B of archives.
After unpacking 328MB disk space will be freed.
You are about to do something potentially harmful
To continue type in the phrase 'Yes, do as I say!' n
Abort.
```

В данном случае при попытке удалить пакет, содержащий каталоги корневой файловой системы, команда `apt-get` обнаружила 145 зависящих от него пакетов, включая критически необходимые, и выдала соответствующее грозное предупреждение. Отвечать утвердительно на такие вопросы `apt-get` не стоит.

При выполнении установки и обновления *APT* получает из Internet необходимые пакеты. Эти пакеты сохраняются в локальном кэше. Для сохранения места на диске данный кэш можно очистить командой `apt-get clean`.

Для поиска нужного пакета в репозитории используется команда `apt-cache search`. Указав ей в качестве параметров нужное имя программы или её описание, можно получить список пакетов. Например, на запрос о пакете с web-сервером выдаётся примерно такой список:

```
$ apt-cache search webserver
apache - Самый популярный веб-сервер Internet
apache-mod_perl - Веб-сервер Russian Apache со встроенным интерпретатором Perl
apache2 - Самый популярный веб-сервер Internet
dvdrip - DVD ripping graphical tool using transcode
furl - Display the HTTP headers returned by webservers
httpd-alterator - Apache HTTP Server (alterator edition)
lighttpd - A fast webserver with minimal memory-footprint
lighttpd-rrdtool - rrdtool support lighttpd module
mod_dav - Модуль DAV под Apache 1.3.x
php5-dbase - dBase database file access functions
freevo - Freevo
jetty5 - The Jetty Webserver and Servlet Container
mailgraph-common - Simple mail statistics for Postfix
maven-plugin-webserver - Optional webserver plugin for maven
perl-LWPx-ParanoidAgent - subclass of LWP::UserAgent that protects you from
harm
```

В репозиториях дистрибутивов содержится огромное количество программ. Если появляется необходимость установить какую-либо новую программу, её прежде всего стоит поискать в готовом виде в репозитории. Чаще всего найти её удастся. Программы, отсутствующие в репозиториях, не стоит собирать из исходных кодов и ставить в систему напрямую, без создания пакета, поскольку в этом случае сильно затрудняется дальнейшее сопровождение системы. Также не стоит особенно переживать, если на сайте разработчика есть более новая версия программы, чем та, что доступна в репозитории. Как правило, в таких случаях у собирающего пакет администратора есть какие-либо причины не обновлять версию в пакете.

Дополнительно можно отметить, что в случае установки программ из готовых пакетов на рабочих системах не требуется наличие компиляторов, заголовочных файлов и прочих инструментов, применяемых при разработке и сборке программ. Это, с одной стороны, позволяет уменьшить место, занимаемое системой на диске, а с другой — создать дополнительные сложности потенциальным злоумышленникам, часто собирающим необходимые им для взлома систем программы непосредственно на этих системах.

Запуск и остановка сервисов, настройка их автоматического запуска при загрузке системы.

Как правило, неинтерактивные программы — демоны или сервисы — должны запускаться при загрузке системы и корректно останавливаться при её выключении/перезагрузке. Для этого в *nix-системах используется система инициализации, в состав которой входит процесс `init`. Как говорилось ранее, процесс `init` запускается ядром операционной системы при загрузке системы. Далее этот процесс, согласно настройкам системы инициализации, запускает другие процессы, выполняющие настройку оборудования, проверку и монтирование файловых систем, запуск демонов, и т.д.

Традиционной и достаточно широко распространённой в настоящее время системой инициализации является система *sysvinit*, представляющая собой достаточно сложную систему скриптов. Для описания состояния системы в *sysvinit* вводится понятия уровня загрузки (уровня выполнения). В любой момент времени система находится на некотором определённом уровне загрузки, и в ней выполняется соответствующий этому уровню набор сервисов. Имеется возможность отдать системе команду и перевести её с текущего уровня на другой. Управляет переключениями уровней загрузки процесс `init`. При переходе с одного уровня на другой `init` последовательно запускает скрипты, останавливающие работающие на текущем уровне загрузки системы сервисы, и затем запускает сервисы, которые должны работать на новом уровне.

Уровней выполнения 7, из них:

- 0 — уровень остановки системы. На этот уровень система переходит по командам `poweroff`, `shutdown`, `halt`. Если подобное поддерживает аппаратная платформа, то после перехода на этот уровень компьютер выключается.
- 1 — однопользовательская система. Используется только в режиме восстановления системы, обычно на этом уровне запускается только командный интерпретатор суперпользователя.
- 2 — многопользовательская система без сетевой поддержки. Как правило, в настоящее время этот уровень не используется.
- 3 — многопользовательская система. Это основной уровень работы системы, используемый по умолчанию.
- 4 — предоставлено для настройки конкретных систем. Обычно то же самое, что и уровень 3, и не используется.
- 5 — многопользовательская система с поддержкой графики. Изначально для настольных системы предусматривалась загрузка на 3-ий уровень, если не требовался запуск графической среды, и на 5-ый — если графическая среда использовалась. В настоящий момент в настольных системах графическая среда запускается и на 3-ем уровне, т.е. 5-ый уровень практически не используется.

- 6 — уровень перезагрузки системы. На этот уровень система переходит по командам `reboot` и `shutdown -r`. После завершения перехода компьютерная система должна перезагрузиться.

Как правило, переходы между уровнями в работающей системе не производятся, и нужны только при её загрузке или остановке.

В ALT Linux и ряде других дистрибутивов сервисы запускаются скриптами, расположенными в каталоге `/etc/rc.d/init.d/`. На этот каталог есть символьная ссылка `/etc/init.d/`, использование путей `/etc/init.d/` и `/etc/rc.d/init.d/` равнозначно. Для управления тем, какой скрипт и на каком уровне запускается, используется команда `chkconfig`.

Посмотреть, какие сервисы должны выполняться при загрузке системы, можно командной `chkconfig --list`:

```
# chkconfig --list
crond           0:off   1:off   2:on    3:on    4:on    5:on    6:off
fbsetfont       0:off   1:off   2:off   3:off   4:off   5:off   6:off
ifrename        0:off   1:off   2:on    3:on    4:on    5:on    6:off
klogd           0:off   1:off   2:on    3:on    4:on    5:on    6:off
lighttpd        0:off   1:off   2:off   3:off   4:off   5:off   6:off
netfs           0:off   1:off   2:off   3:on    4:on    5:on    6:off
network         0:off   1:off   2:on    3:on    4:on    5:on    6:off
portmap         0:off   1:off   2:off   3:off   4:off   5:off   6:off
random          0:off   1:off   2:on    3:on    4:on    5:on    6:off
rawdevices      0:off   1:off   2:off   3:off   4:off   5:off   6:off
sshd            0:off   1:off   2:on    3:on    4:on    5:on    6:off
syslogd         0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

Например, демон `sshd` выключен на уровнях 0, 1 и 6, и включён на уровнях 2, 3, 4 и 5. Посмотреть на состояние конкретного сервиса можно, указав его имя:

```
# chkconfig --list lighttpd
lighttpd        0:off   1:off   2:off   3:off   4:off   5:off   6:off
```

Как видно, сервис `lighttpd` выключен и при перезагрузке системы запускаться не будет.

Для включения сервиса следует выполнить команду `chkconfig <имя сервиса> on`:

```
# chkconfig lighttpd on
# chkconfig --list lighttpd
lighttpd        0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

Выключается сервис командой `chkconfig <имя сервиса> off`.

Включение и выключение сервисов в конфигурации запуска системы не запускает и не останавливает их в работающей системе. Как правило, перезагрузка *nix-систем — это очень редкое и обычно вынужденное событие. Для запуска и остановки сервисов в работающей системе в ALT Linux используется команда `service`. Формат её вызова:

```
service <имя сервиса> <команда>.
```

Имя сервиса то же, что и для команды `chkconfig` (и, на самом деле, это имя скрипта из `/etc/init.d/`). Все сервисы поддерживают команды `start` (для запуска неработающего сервиса), `stop` (для остановки работающего сервиса), `restart` (для остановки и последующего запуска сервиса), `status` (для получения статуса сервиса). Возможны и дополнительные команды, которые можно узнать, запустив `service <имя сервиса>` без указания команды.

Подавляющее большинство скриптов в `/etc/init.d` отслеживают состояние запускаемых ими программ и не позволят повторно запустить их.

Например, для сервиса lighttpd:

```
 # service lighttpd
 Usage: lighttpd {start|stop|restart|condstop|condrestart|condreload|reload|
status}
 # service lighttpd status
 lighttpd is stopped
 # service lighttpd start
 Starting lighttpd service: [ DONE ]
 # service lighttpd status
 lighttpd is running
 [root@lab-100 ~]# service lighttpd restart
 Stopping lighttpd service: [ DONE ]
 Starting lighttpd service: [ DONE ]
 # service lighttpd status
 lighttpd is running
 # service lighttpd stop
 Stopping lighttpd service: [ DONE ]
 # service lighttpd status
 lighttpd is stopped
 # service lighttpd restart
 Service lighttpd is not running.[PASSED]
 Starting lighttpd service: [ DONE ]
 #
```

Видно, что сервис `lighttpd` поддерживает команды `start`, `stop`, `restart`, `status`. Команды `condstop`, `condrestart` и `condreload` в основном предназначены для перезапуска сервиса при обновлении пакета с ним.

Изначально сервис не был запущен. По команде `service lighttpd start` он был запущен, что подтвердил последующий вывод команды `service lighttpd status`. Также успешно прошёл перезапуск сервиса (в процессе которого он остановился и заново запустился, перечитав свою конфигурацию), и его остановка. Последняя команда `restart` не нашла работающего сервиса `lighttpd`, о чём сообщила, и потом его успешно запустила.

Помимо *sysvinit* существуют и другие системы инициализации: *Upstart*, *Runit*, *systemd*, и т.д. Выбор той или иной системы инициализации зависит от предпочтений составителей конкретного дистрибутива и направленности конкретного дистрибутива для решения тех или иных задач. Описанная

выше *sysvinit* отличается малой требовательностью к ресурсам для своей работы, и широко используется для серверных и встраиваемых систем. Основными её недостатками являются последовательное выполнение операций запуска или остановки демонов в процессе перехода с одного уровня выполнения на другой, и сложность задания правильной последовательности запуска и остановки зависящих друг от друга демонов.

Для современных настольных и серверных Linux-систем в настоящее время преимущественно выбирается система инициализации *systemd*, обеспечивающая параллельный запуск демонов при загрузке системы и, тем самым, существенно уменьшающая время загрузки. Для просмотра и управления конфигурацией в *systemd* используются консольная команда `systemctl` и её графический аналог `systemadm`.

В отличие от *sysvinit* в системе инициализации *systemd* для конфигурации сервисов используются не наборы скриптов на языке командного интерпретатора, а файлы конфигурации, описывающие порядок и параметры запуска сервисов. Вместо уровней выполнения используется понятие целей (*target*), для достижения нужной цели *systemd* определяет по файлам конфигурации нужный порядок остановки или запуска сервисов и выполняет соответствующие операции. Цель по-умолчанию носит название «`multi-user.target`».

Настройка и получение информации о работе сервисов выполняются с использованием команды `systemctl`. Для запуска сервиса в режиме командной строки используется команда `systemctl start <имя сервиса>` , для остановки - `systemctl stop <имя сервиса>` , для перезапуска - `systemctl restart <имя сервиса>`

Включить автоматический запуск сервиса используется можно командой `systemctl enable <имя сервиса>` , выключить автоматический запуск - `systemctl disable <имя сервиса>` . Также можно получить информацию о состоянии сервиса — `systemctl status <имя сервиса>` . Например,

```
 # systemctl enable lighttpd
 Synchronizing state of lighttpd.service with SysV service script with
/lib/systemd/systemd-sysv-install.
 Executing: /lib/systemd/systemd-sysv-install enable lighttpd
 Created symlink /etc/systemd/system/multi-user.target.wants/lighttpd.service →
/lib/systemd/system/lighttpd.service.

 # systemctl start lighttpd

 # systemctl status lighttpd
 ● lighttpd.service - Lighttpd Daemon
    Loaded: loaded (/lib/systemd/system/lighttpd.service; enabled; vendor
preset: disabled)
    Active: active (running) since Thu 2019-10-17 03:59:54 UTC; 2s ago
   Process: 32096 ExecStartPre=/usr/sbin/lighttpd -tt -f
/etc/lighttpd/lighttpd.conf (code=exited, status=0/SUCCESS)
  Main PID: 32097 (lighttpd)
```

```
     Tasks: 1 (limit: 4915)
    Memory: 968.0K
    CGroup: /system.slice/lighttpd.service
            └─32097

Oct 17 03:59:54 lab-00.edu.cbias.ru systemd[1]: Starting Lighttpd Daemon...
Oct 17 03:59:54 lab-00.edu.cbias.ru systemd[1]: Started Lighttpd Daemon.
```

В отличие от *sysvinit*, *systemd* может отслеживать выполнение нужных сервисов и в случае каких-либо сбоев автоматически перезапускать их.

Также в рамках *systemd* имеется централизованный сервис сбора и хранения журналов работы системы и сервисов — *journald*. Получить логи работы системы и сервисов можно, используя команду `journalctl`. По-умолчанию `journalctl` выводит все логи с начала последнего запуска систем, можно ограничить его вывод последними N строками (команда вида `journalctl -n 100`), или посмотреть логи запуска и выполнения конкретного сервиса (`journalctl -u <имя сервиса>`):

```
 # journalctl -u lighttpd -n 5
 -- Logs begin at Wed 2019-10-16 16:59:50 UTC, end at Thu 2019-10-17 04:01:11
UTC. --
 Oct 17 03:59:46 lab-00.edu.cbias.ru systemd[1]:
/lib/systemd/system/lighttpd.service:7: PIDFile= references a path below legacy
directory /var/run/, updating /var/run/lighttpd.pid → /run/lighttpd.pid; please
update the unit file accordingly.
 Oct 17 03:59:47 lab-00.edu.cbias.ru systemd[1]:
/lib/systemd/system/lighttpd.service:7: PIDFile= references a path below legacy
directory /var/run/, updating /var/run/lighttpd.pid → /run/lighttpd.pid; please
update the unit file accordingly.
 Oct 17 03:59:52 lab-00.edu.cbias.ru systemd[1]:
/lib/systemd/system/lighttpd.service:7: PIDFile= references a path below legacy
directory /var/run/, updating /var/run/lighttpd.pid → /run/lighttpd.pid; please
update the unit file accordingly.
 Oct 17 03:59:54 lab-00.edu.cbias.ru systemd[1]: Starting Lighttpd Daemon...
 Oct 17 03:59:54 lab-00.edu.cbias.ru systemd[1]: Started Lighttpd Daemon.ervice
lighttpd
```

Другие варианты использования `journalctl` можно посмотреть в его справочном руководстве man.

Нужно отметить, что часть программ не использует системные сервисы журналов, и самостоятельно записывают журналы работы в свои подкаталоги внутри каталога `/var/log/` .

В целях совместимости в системе инициализации systemd поддерживаются также скрипты *sysvinit* в `/etc/rc.d/init.d/` и команды `service` и `chkconfig`.

В используемом в настоящей лабораторной работе дистрибутиве ALT Linux Server P9 в качестве системы инициализации используется *systemd*, в рамках выполнения лабораторной работы можно использовать как команды `service` и `chkconfig` в режиме совместимости с *sysvinit*, так и напрямую команду `systemctl` .

Периодическое (регулярное) выполнение задач.

Скрипты можно использовать для автоматизации тех или иных задач. Очень часто при этом требуется организовать выполнение скрипта в заданное время или через определённые интервалы времени. Для этого существует специальный демон — `crond`.

Для настройки программ на регулярное выполнение используется файл конфигурации, который можно посмотреть командной `crontab -l` и изменить командой `crontab -e`.

Рассмотрим такой файл:

```
$ crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/.private/student/crontab.6WaeT9 installed on Mon Mar 17 12:39:10 2008)
# (Cron version V5.0 -- vixie-cron-4.1.20060426-alt3)
#minute (0-59),
#|      hour (0-23),
#|      |       day of the month (1-31),
#|      |       |       month of the year (1-12),
#|      |       |       |       day of the week (0-6 with 0=Sunday).
#|      |       |       |       |       commands
*/1     *       *       *       *       /var/www/bin/log-local.sh
*/2     *       *       *       *       /var/www/bin/log-snmp.sh
```

Строки, начинающиеся с `#` — как обычно, комментарии. Для каждой из запускаемых команд указывается, когда её надо выполнить. Для этого используются пять полей: минуты, часы, дни месяца, месяцы и дни недели. Для каждого из полей можно указать или какое-либо определённое значение, или `*` (астериск), что означает «для всех».

Для выбора дня выполнения задачи можно использовать или поля «день месяца» и «месяц», или поле «день недели». При указании для задачи и дня месяца, и дня недели, эти условия объединяются через логическое сложение (через «логическое ИЛИ»).

Рассмотрим значения этих полей на примера вызова программы `/bin/false`:

`* * * * * /bin/false`	Запускать каждую минуту (каждого часа, каждого дня, каждого месяца, в любой день недели).
`*/3 * * * * /bin/false`	Запускать каждые три минуты (каждого часа, каждого дня, каждого месяца, в любой день недели).
`*/3 1-2 * * * /bin/false`	Запускать каждые три минуты первого и второго часа ночи (каждого дня, каждого месяца, в любой день недели).
`1 1,6 * * * /bin/false`	Запускать в первую минуту первого и

	шестого часа ночи, т.е. в 01:01 и 06:01 (каждого дня, каждого месяца, в любой день недели).
`1 1 * * 1 /bin/false`	Запускать в 01:01 каждый понедельник.
`1 1 * 2 1 /bin/false`	Запускать в 01:01 каждый понедельник или в 01:01 каждого дня февраля.
`* * 31 10 5 /bin/false`	Запускать каждую минуту каждого часа 31 октября, или в каждую минуту каждого часа каждой пятницы.

При выполнении по `cron`'у задач, которые потенциально могут выполняться длительное время, следует предусмотреть и блокировать повторный запуск `cron`'ом скрипта в то время, когда ещё не успел завершиться предыдущий. Обычно такое можно делать, создавая и анализируя при запуске скрипта файл блокировки. Например:

```
$ cat lock.sh
#!/bin/bash
LOCK=/tmp/file.lock

if [ -f "$LOCK" ]; then
        echo 'Скрипт уже работает'
        exit 1
fi
touch "$LOCK"

sleep 1m

rm -f "$LOCK"
```

Здесь при запуске скрипта проверяется существование файла, и если он существует, то выполнение скрипта завершается. Иначе файл создаётся, выполняется некое действие (в данном случае — просто ожидание на 1 минуту), и перед завершением работы файл блокировки удаляется.

Пример выполнения:

```
$ ./lock.sh &
[1] 7702
$ ./lock.sh &
[2] 7704
$ Скрипт уже работает
[2]+  Exit 1                  ./lock.sh
```

Выполнение лабораторной работы.

Лабораторная работа посвящена изучению основ работы с операционной системой семейства *nix, основ взаимодействия команд в операционных системах семейства *nix, использованию перенаправления потоков ввода-вывода, регулярных выражений, написанию простых программ на языке командного интерпретатора и выполнению основных задач по администрированию операционной системы. Выполнение лабораторной работы предусматривает работу с удалённым сервером. Для доступа к серверу используется терминальная программа *PuTTY*.

В лабораторной работе требуется:

- ознакомится с основами работы в операционной системе ALT Linux Server, изучить работу основных команд операционной системы;
- провести начальную настройку и подготовку операционной системы к использованию;
- организовать периодическое получение данных о работе определённых систем;
- записывать их в файл для последующего анализа;
- организовать получение текущих значений через веб-интерфейс;
- построить графики изменения наблюдаемых величин и предоставить к ним доступ через веб-интерфейс.

Поскольку, как правило, под решение практически любой задачи в Linux можно найти в Internet или готовое решение, или набор рецептов, то выполнение лабораторной работы предусматривает использование готовых скриптов для выполнения поставленных задач. С другой стороны, данные скрипты надо установить на конкретную систему, адаптировать их под задачу и обеспечить их выполнение в рамках выделенного виртуального сервера.

Перед началом выполнения работы необходимо получить у преподавателя индивидуальные данные, содержащие:

- учётную запись пользователя на удалённом сервере: имя (идентификатор пользователя) и пароль;
- сетевой адрес сервера и номер порта для удалённого входа на него;
- имя сервера для доступа по протоколу http;
- список репозиториев для настройки системы *APT*.

В ходе данной лабораторной работы требуется изменить идентификатор пользователя и пароль для доступа к серверу. Остальные данные остаются постоянными для последующих работ в рамках данного курса.

Для входа на сервер требуется загрузить терминальную программу *PuTTY*. Внутри сети МЭИ на период проведения данного курса она доступна со страницы http://edu.cbias.ru/ или по прямой ссылке

http://edu.cbias.ru/files/putty.exe.

После запуска программы появляется окно с настройкой параметров соединения. Полученные сетевой адрес сервера и номер порта следует ввести в поля *Host Name (or IP address)* и *Port* соответственно. Для корректной работы с разными кодировками сервера и клиентской системы, требуется указать кодировку поступающих от сервера символов. Эти настройки задаются на вкладке *Windows → Translation*, выбираемой из списка слева в окне настроек. В выпадающем списке *Received data assumed to be in which character set:* требуется задать нужную кодировку (в данном случае - UTF-8), для старых версий *PuTTY* вместо *KOI8-U* следует указать *UTF-8*.

Для подключения к серверу и начала сеанса нажмите кнопку *Open* внизу окна настроек.

В появившемся окне консоли на запрос *login as:* введите идентификатор пользователя, на запрос *password* — пароль. Пароль при вводе не отображается. В случае ошибки повторите ввод пароля или перезапустите *PuTTY*.

После успешного входа в систему в окне терминала появляется приглашение вида:

```
[student@lab-100 ~]$
```

Дальнейшие команды, вводимые в терминале, выполняются на удалённом сервере.

Изучите структуру каталогов сервера, пользуясь командами `ls` (в т.ч. с ключами `-l`, `-la`, `-a`), `cd`, `pwd` .

Запустите менеджер файлов *Midnight Commander* (команда `mc`). Для перехода из оконного режима `mc` в консольный и обратно используйте сочетание клавиш `<Ctrl>+<o>`. Используйте `mc` для копирования, перемещения и удаления файлов. Повторите те же операции из командной строки, используя `cp`, `mv`, `rm`. Используйте возможности командного интерпретатора по автоматическому дополнению имени файлов при нажатии клавиши `<Tab>`.

Перейдите в каталог `~/Documents`, создайте пустой файл командой `touch`. Для получения справки по параметрам команды используйте команду `man`.

Для выхода из справочного руководства используйте клавишу `<q>`.

Введите какой-либо текст в созданный файл, используя встроенный редактор `mc` (`<F4>`).

Выйдите из *Midnight Commander*.

Внесите произвольные изменения в ~/Documents/file.txt с помощью редактора vim. Для завершения работы с vim используйте последовательность команд <Esc>:wq .

Изучите список пользователей и групп, находящийся в файлах /etc/passwd и /etc/group. Изучите права на файлы в домашнем каталоге пользователя, каталогах */etc*, */sbin*, */var/log*. Попробуйте прочитать записи в системном журнале /var/log/messages .

Получите права суперпользователя, используя команду su -l. Ключ -l обязателен. Рекомендуется выйти из Midnight Commander перед запуском su. Изначально пароль пользователя root отсутствует, после задания пароля su будет его запрашивать. Без указания выполняемой команды в качестве параметра su запускает командный интерпретатор с правами суперпользователя. Приглашение для root выглядит так:

```
[root@lab-100 ~]#
```

Запустите командный интерпретатор с правами root.

Задайте пароль на пользователя root. Задайте пароль для пользователя student, запустив passwd с соответствующим параметром.

Завершите сеанс root, выйдя из командного интерпретатора (exit или <Ctrl>+<D>).

Снова запустите командный интерпретатор с правами root.

Создайте нового пользователя. Имя пользователя выберите самостоятельно. Имя пользователя может содержать латинские строчные буквы, цифры и символы - (дефис) и _ (нижнее подчёркивание), и по возможности не должно превышать 8-ми символов. Для создания пользователя используйте команду useradd.

Проверьте список пользователей и групп в системе.

Проверьте, какие пользователи имеют право на запуск команды su (полное имя файла команды — /bin/su). Внесите созданного пользователя в нужные группы, отредактировав файл /etc/group.

Задайте пароль для созданного пользователя.

Запустите вторую терминальную сессию. Для этого в меню окна *PuTTY* (доступного при нажатии на иконку приложения слева в заголовке окна) выберите пункт *New Session*. Повторите настройки подключения и осуществите вход в систему под учётной записью созданного пользователя. Убедитесь в возможности получения им прав суперпользователя, запустив команду su -l.

Удалите учётную запись пользователя `student`, используя команду `userdel`. Убедитесь в успешном выполнении команды, проверив содержимое файлов `/etc/passwd` и `/etc/group`, а также попробовав запустить терминальную сессию под этим пользователем.

Найдите в `/home` домашние каталоги созданного и удалённого пользователей. Перенесите созданный в `Documents/` текстовый файл в каталог созданного пользователя.

При необходимости поменяйте права на файл с помощью команд `chmod` и `chown`.

Удалите домашний каталог пользователя `student`.

Получите полный список пакетов *RPM*, установленных в системе, командой `rpm -qa`. Получите детальную информацию об одном или нескольких пакетах, выполнив команды `rpm -qi <имя пакета>`.

Рассмотрите настройки списка репозиториев системы *APT*, находящиеся в файле `/etc/apt/sources.list`. Внесите в него записи о репозиториях, согласно выданным преподавателем рекомендациям.

Обновите локальные списки пакетов системы *APT*, выполнив команду `apt-get update`. В случае появления сообщений об ошибках проверьте и исправьте список репозиториев, и снова выполните обновление списка пакетов.

Обновите систему до текущего состояния репозиториев, выполнив команды `apt-get upgrade` и `apt-get dist-upgrade`. Обратите внимание на перечень обновлённых пакетов. Получите информацию о последних изменениях какого-либо пакета, выполнив команду `rpm -q --changelog <имя пакета>`.

Используя команду `apt-cache search <строка для поиска>`, найдите пакет, содержащий веб-сервер `lighttpd`. Установите пакет через вызов команды `apt-get install`.

Найдите в основном конфигурационном файле веб-сервера `lighttpd` (`/etc/lighttpd/lighttpd.conf`) путь к каталогу с файлами, доступными веб-серверу (параметр `server.document-root`).

Поместите в указанный каталог (при необходимости создав его) произвольный текстовый файл. Имя файла должно иметь расширение `.txt`.

Браузер подключается к веб-серверу, устанавливая сетевое соединение по протоколу TCP/IP. Выбор нужного веб-сервера осуществляется по определённому адресу IP и порту TCP. По-умолчанию, для схемы http://

используется порт 80, для схемы https:// - 443, и, как правило, вместо адреса IP в браузере указывается доменное имя, соответствующее нужному адресу IP. (Подробнее вопросы работы протоколов TCP/IP рассматриваются в лабораторной работе № 3.)

Чтобы браузер мог подключиться к веб-серверу по адресе IP и порту TCP, веб-сервер должен ожидать входящие подключения на этот адрес IP и порт TCP. Как правило, на сервере есть несколько адресов IP на разных интерфейсах, и по каким из этих адресов веб-сервер ожидает подключение, указывается в его конфигурации.

В конфигурации lighttpd адреса, подключения по которым ожидает lighttpd, задаются в параметре конфигурации `server.bind`. По-умолчанию lighttpd принимает соединения только на локальный адрес сервера («`localhost`»).

Чтобы можно было подключиться к веб-серверу из любых внешних сетей, в данном параметре требуется задать значение «`0.0.0.0`».

Запустите веб-сервер `lighttpd`, выполнив команду `service lighttpd start`. Проверьте, работает ли сервер, выполнив команду `service lighttpd status`. Проверьте наличие сервера в списке выполняемых процессов, выполнив команду `ps aux`. Обратите внимание на пользователя, под которым выполняется процесс веб-сервера. Получите файл из браузера, указав имя сервера и имя файла.

Получите список зарегистрированных в системе сервисов, командой `chkconfig --list`. Убедитесь в присутствии в списке `lighttpd`. В случае его отсутствия, добавьте сервис командой `chkconfig lighttpd --add`. Для автоматического запуска при загрузке системы сервис должен быть включён на уровнях выполнения 2-5. Если он выключен, включите его командой `chkconfig lighttpd on`.

Перезапустите систему командой `reboot`. Дождитесь загрузки сервера (время перезагрузки находится в пределах 5 минут). Войдите в систему. Проверьте, работает ли `lighttpd`.

Проведя таким образом начальную настройку операционной системы, можно приступать к установке набора скриптов для сбора и отображения данных о работе системы, что является основной целью данной лабораторной работы.

Поскольку, как правило, под решение практически любой задачи в Linux можно найти в Internet или готовое решение, или набор рецептов, то выполнение лабораторной работы предусматривает использование готовых скриптов для выполнения поставленных задач. С другой стороны, данные

скрипты надо установить на конкретную систему, адаптировать их под задачу и обеспечить их выполнение в рамках выделенного виртуального сервера.

В лабораторной работе требуется получить, записать и проанализировать следующие значения:

- число процессов в системе. Данный параметр может быть получен путём вывода полного списка выполняющихся в системе процессов и подсчёта числа строк в этом списке.
- суммарный объем переданных и принятых через сетевой интерфейс `venet0` данных в байтах. Эти значения содержатся в выводе команды `netstat -i`, в соответствующих полях выдаваемой таблицы.
- Число переданных и принятых через порт удалённого коммутатора пакетов и байтов данных. Данные величины могут быть получены по протоколу *SNMP* с использованием программы `snmpget`.

Вызов программы snmpget имеет вид:

```
$ snmpget -c public -v 1 192.168.250.1 IF-MIB::ifDescr.2 \
> IF-MIB::ifInOctets.2 \
> IF-MIB::ifInUcastPkts.2 \
> IF-MIB::ifOutOctets.2 \
> IF-MIB::ifOutUcastPkts.2
IF-MIB::ifDescr.2 = STRING: eth0
IF-MIB::ifInOctets.2 = Counter32: 120684456
IF-MIB::ifInUcastPkts.2 = Counter32: 1215812
IF-MIB::ifOutOctets.2 = Counter32: 1559547791
IF-MIB::ifOutUcastPkts.2 = Counter32: 1341129
```

Здесь было произведено обращение к коммутатору `192.168.250.1`, с которого были запрошены параметры:

`IF-MIB::ifDescr.2` — имя 2-го сетевого интерфейса;
`IF-MIB::ifInOctets.2` — число принятых интерфейсом байтов;
`IF-MIB::ifInUcastPkts.2` — число принятых интерфейсом пакетов;
`IF-MIB::ifOutOctets.2` — число переданных интерфейсом байтов;
`IF-MIB::ifOutUcastPkts.2` — число переданных интерфейсом пакетов.

Вывод команды приведён выше.

IP-адрес коммутатора и номер сетевого интерфейса индивидуальны для каждого виртуального сервера, их можно найти в файле `/root/SNMP.data` . В случае отсутствия такого файла в системе эти данные должны быть получены у преподавателя.

В ходе лабораторной работы используются программы `netstat` и `snmpget`, которые можно установить из пакетов `net-snmp-clients` и `net-tools`. Для отображения графиков используется набор утилит `RRDTOOLS` из пакета `rrd-utils`.

Для получения данных предлагается использовать следующие программы:

log-local.sh — получение и запись в файл локальной статистики.

```
#!/bin/bash
# Script for logging current system status:
# - number of processes
# - RX and TX bytes over venet0 network interface

# Log to this file:
LOG_FILE=/var/www/stat/local.log

# Timestamp:
TS=`date '+%Y-%m-%d %H:%M:%S'`

# Process number
PROCNUM=`ps aux | wc -l`
PROCNUM=$(($PROCNUM-1))

# netstat info
NETBYTES=`netstat -i | grep '^eth0' | awk '{print "RX ",$4,"bytes, TX ",
$8,"bytes."}'`

# Log all to the file
echo "$TS => Procs: $PROCNUM, $NETBYTES" >> "$LOG_FILE"
#------------------------------------------------------------
```

log-snmp.sh — получение и запись в файл SNMP-статистики.

```
#!/bin/bash
# Script for logging current SNMP information:
# - RX and TX bytes over some network interface

# Log to this file:
LOG_FILE=/var/www/stat/snmp.log

# Network interface number:
N=8

# SNMP host
HOST=192.168.222.100

# SNMP community
COMMUNITY=public

# MIBS
MIB1="IF-MIB::ifDescr.$N"
MIB2="IF-MIB::ifInOctets.$N"
MIB3="IF-MIB::ifInUcastPkts.$N"
MIB4="IF-MIB::ifOutOctets.$N"
MIB5="IF-MIB::ifOutUcastPkts.$N"

############################
# Timestamp:
TS=`date '+%Y-%m-%d %H:%M:%S'`

# snmp info
RES=''

for MIB in $MIB1 $MIB2 $MIB3 $MIB4 $MIB5; do
    LINE=`snmpget -c $COMMUNITY -v 1 $HOST $MIB`

    NAME=`echo $LINE | sed "s/^IF-MIB::\([[:alnum:]]\+\).*/\1/"`
    VALUE=`echo "$LINE" | sed "s/^IF-MIB::[[:alnum:]]\+\.$N = [[:alnum:]]\
+: //"`

    RES="$RES  $NAME:$VALUE"
done

# Log all to the file
echo "$TS => $RES" >> "$LOG_FILE"
#-------------------------------------------------------------
```

Запуск скриптов получения данных предполагается осуществлять раз в минуту для получения локальной информации, и раз в две минуты — для получения информации с коммутатора через *SNMP*.

Для вывода данных по запросам браузера предлагается установить в систему для запуска с помощью `lighttpd` следующие скрипты:

cgi-local.sh — отображение локальной статистики.

```
#!/bin/bash
# Simple CGI script

echo Content-type: text/plain
echo ""

LOG_FILE=/var/www/stat/local.log

# Show NUM lines
if [ -n "$QUERY_STRING" ]; then
    NUM=$QUERY_STRING
else
    if [ -n "$1" ]; then
        NUM=$1
    else
     NUM=10
    fi
fi

echo "Current statistic:"
tail -n $NUM "$LOG_FILE" | sort -r
#---------------------------------------------------------
```

cgi-snmp.sh — отображение SNMP-статистики.

```
#!/bin/bash
# CGI script for SNMP statistic

echo Content-type: text/plain
echo ""

LOG_FILE=/var/www/stat/snmp.log

# Show NUM lines
if [ -n "$QUERY_STRING" ]; then
    NUM=$QUERY_STRING
else
    if [ -n "$1" ]; then
        NUM=$1
    else
     NUM=10
    fi
fi

echo "Current statistic:"
tail -n $NUM "$LOG_FILE" | sort -r
#---------------------------------------------------------
```

Текст скриптов, обеспечивающих вывод данных в табличной форме и построение графиков, приведён на странице с примерами к данной лабораторной работе – http://lab-00.edu.cbias.ru/ .

Скрипты предполагается размещать в каталогах внутри /var/www, с использованием для скриптов получения данных каталога /var/www/bin, для веб-интерфейса — document_root веб-сервера, для хранения журналов — /var/www/stat. Для хранения графиков используется подкаталог внутри document_root веб-сервера.

Для запуска скриптов как веб-программ следует разрешить это в настройках lighttpd (расположенных в каталоге /etc/lighttpd/):

- нужно подключить модуль mod_cgi веб-сервера, раскомментировав строку «include "conf.d/cgi.conf"» в файле modules.conf;
- задать в подключенном файле конфигурации модуля mod_cgi (conf.d/cgi.conf) секцию параметров вида:

```
cgi.assign       = ( ".pl"  => "/usr/bin/perl",
                                …
                     ".rrd" => "/usr/bin/rrdcgi",
                     ".sh" => "/bin/bash" )
```

- добавить расширения файлов скриптов в параметр static-file.exclude-extensions в файле lighttpd.conf:

```
static-file.exclude-extensions = ( ".php", ".pl", ".fcgi", ".sh", ".rrd" )
```

Изменения настроек вступают в силу после перезапуска lighttpd.

Также для запуска скриптов на выполнение веб-сервер lighttpd должен иметь права на чтение использования каталога с ними.

Для обеспечения безопасности системы должны соблюдаться определённые правила выполнения скриптов.

Сбор статистики должен выполняться от имени непривилегированного пользователя. Обычно для подобных задач создаётся отдельный псевдо-пользователь с ограниченными по сравнению с обычными пользователями системы правами. Псевдопользователь не должен иметь возможности удалённого входа в систему и не должен иметь возможности изменения скриптов.

Отображающие информацию скрипты выполняются веб-сервером. Пользователь, под которым работает веб-сервер, не должен иметь возможности записи как в файлы скриптов, так и в файлы с сохранённой статистикой (файлы логов).

Временные файлы, создаваемые веб-сервером, не должны быть доступны для записи или удаления остальным пользователям системы.

Остальные пользователи системы не должны иметь возможности чтения и записи файлов логов.

Для удобного доступа к различным скриптам в `/var/www/html` предлагается разместить индексный файл с названием `index.html` вида:

```
<html>
<head>
<title>index</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
<h1>Текущая статистика</h1>
<ul>
<li><a href="/cgi-local.sh">Простой скрипт статистики локальной
системы</a></li>
<li><a href="/cgi-local-html.sh">HTML-скрипт статистики локальной
системы</a></li>
<li><a href="/cgi-local-html-table.sh">HTML-скрипт с выводом таблицей
статистики локальной системы</a></li>
<li><a href="/cgi-local.rrd">Выдача графиков статистики локальной
системы</a></li>
<li><a href="/cgi-snmp.sh">Простой скрипт статистики интерфейса SNMP</a></li>
<li><a href="/cgi-snmp-html.sh">HTML-скрипт статистики интерфейса
SNMP</a></li>
<li><a href="/cgi-snmp-html-table.sh">HTML-скрипт с выводом таблицей
статистики интерфейса SNMP</a></li>
<li><a href="/cgi-snmp.rrd">Выдача графиков статистики интерфейса
SNMP</a></li>
</ul>
</body>
</html>
```

Исходные тексты скриптов для сбора данных, скриптов для форматирования и вывода собранных данных, и приведённого выше индексного файла доступны для просмотра и скачивания на сайте http://edu.cbias.ru. Там же, на странице http://lab-00.edu.cbias.ru/ , можно посмотреть примеры результатов работы этих скриптов.

Задания на лабораторную работу.

1. Выполнить удалённую регистрацию в системе.
2. Провести ознакомление с операционной системой. Изучить структуру каталогов сервера. Посмотреть доступные команды в системе, вызвать справочное руководство по каким-либо из них. Создать текстовый файл, используя редактор `vi`.
3. Используя команду `su`, получить привилегии суперпользователя системы.
4. Изменить пароли пользователя и суперпользователя системы.
5. Создать новую учётную запись пользователя.
6. Зарегистрироваться в системе под созданным в п. 5 пользователем, убедиться в возможности использования им команды `su`.
7. Удалить учётную запись пользователя `student`.
8. Изучить список пакетов, установленных в системе.
9. Настроить список репозиториев пакетов для системы *APT*. Провести обновление системы до текущего состояния репозитория.
10. Установить веб-сервер `lighttpd`, запустить сервер. Проверить работу веб-сервера. Настроить его автоматический запуск при загрузке системы.
11. Перезагрузить систему. Убедиться, что веб-сервер `lighttpd` автоматически запустился после перезагрузки системы.
12. Доставить в систему всё необходимое для работы скриптов сбора и отображения статистики программное обеспечение.
13. Адаптировать приведённые в описании работы скрипты, получающие значения статистических параметров и записывающие их журналы.
14. Обеспечить периодическое регулярное выполнение скриптов.
15. Адаптировать приведённые в описании работы скрипты для отображения записываемых в пп. 13-14 данных из журналов, обеспечить их выполнение из командной строки.
16. Настроить `lighttpd` для удалённого обращения из браузера к указанным скриптам и отображения собираемых данных в веб-браузере на удалённом рабочем месте.
17. Обеспечить безопасное выполнение скриптов.

Контрольные вопросы.

1. Какие основные каталоги есть в файловой системе *nix?
2. В каких каталогах хранятся настройки системы?
3. В каких каталогах можно найти установленные в системе программы, доступные для пользователя?
4. В каких каталогах можно найти установленные системные программы и программы, предназначенные для выполнения суперпользователем?
5. Где хранится список пользователей и групп пользователей?
6. Как можно создать нового пользователя в системе?
7. Как можно включить пользователя в новую группу?
8. Когда вступают в силу изменения в списке групп пользователя?
9. Какие пользователи могут запускать команду `su` в ALT Linux?
10. Что такое суперпользователь системы?
11. Что такое псевдопользователи, и зачем они нужны?
12. Как посмотреть права доступа к конкретному файлу?
13. Как изменить права доступа к файлу?
14. Что такое потоки ввода/вывода? Как можно перенаправить поток ввода, поток вывода?
15. Что такое скрипт, как создать скрипт и разрешить его выполнение?
16. Что такое переменная окружения, как посмотреть значение переменной окружения?
17. Как определить и использовать переменную *shell*?
18. Какие управляющие конструкции доступны в языке командного интерпретатора?
19. Что такое регулярное выражение?
20. Какие основные конструкции используются в регулярных выражениях?
21. Что такое пакет *RPM*?
22. Какая информация хранится в заголовке пакета *RPM*?
23. Что такое репозиторий пакетов?
24. Как найти в репозитории пакет, содержащий нужную программу?
25. Как запустить, остановить, перезапустить сервис?
26. Какие сервисы настроены на автоматическое выполнение при загрузке системы?
27. Как включить и исключить сервис из списка автоматического выполнения?
28. Как организовать периодическое выполнение программ?
29. Объясните порядок работы скриптов, использованных в лабораторной работе для получения и вывода данных.
30. Поясните, под какими учётными записями пользователей выполняются установленные в лабораторной работе скрипты и как ограничен доступ к используемым ими каталогам выбранными правами доступа.

Литература

1. Георгий Курячий, Кирилл Маслинский
   «Введение в ОС Linux» - учебное пособие по работе с операционной системой Linux, распространяется на условиях лицензии GNU FDL:
   http://heap.altlinux.org/alt-docs/modules/LinuxIntro.george/index.html

2. ALT Linux снаружи. ALT Linux изнутри. Под ред. Кирилла Маслинского, М.: ALT Linux; Издательский дом ДМК-пресс, 2006 г. - 416 стр. Доступна на условиях лицензии GNU FDL,
   http://heap.altlinux.org/alt-docs/compactbook/index.html

3. Робачевский А.М., Немнюгин С.А., Стесик О.Л. Операционная система UNIX. – 2 изд., СПб.: BHV – Санкт-Петербург, 2005. – 636 с.

4. Забродин Л.Д. UNIX. Введение в командный интерфейс. – М.: ДИАЛОГ-МИФИ, 1994. – 144 с.

5. Керниган Б.В., Пайк Р. UNIX – универсальная среда программирования: Пер. с англ. – М.: Финансы и статистика, 1992. – 304 с.

6. Дансмур М., Дейвис Г. Операционная система UNIX и программирование на языке Си: Пер. с англ. – М.: Радио и связь, 1989. – 192 с.

7. Торвальдс Л., Даймонд Д. Ради удовольствия: рассказ нечаянного революционера. — М.: ЭКСМО-Пресс, 2002. — 288 с.
   http://www.lib.ru/LINUXGUIDE/torvalds_jast_for_fun.txt

8. Advanced Bash-Scripting Guide, перевод на русский язык
   http://www.opennet.ru/docs/RUS/bash_scripting_guide/

9. Advanced Bash-Scripting Guide
   http://tldp.org/LDP/abs/html/

Текст лицензии GNU FDL можно найти по адресу:
http://www.gnu.org/licenses/fdl.html