Лабораторная работа № 2

Командный интерпретатор и основы программирования на shell
Основы регулярных выражений

Copyright (c) 2008 Nikolay A. Fetisov
Copyright (c) 2011 – 2014, 2016 – 2018, 2022 – 2026 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 (c) Николай Фетисов, 2008.
Copyright (c) Фёдор Фетисов, Николай Фетисов, 2011 – 2014, 2016 – 2018, 2022 – 2026 .
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.

Введение.

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.

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 продолжить работу в редакторе с того состояния, 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 выво-

дом 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). Передать сигнал процессу можно, используя функцию `kill()` из стандартной системной библиотеки. 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 прерываний 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.

Полный перечень доступных для использования сигналов можно получить, запустив утилиту `kill` с ключом `-l`:

```
$ kill -l
```

Перечень доступных сигналов и их номера могут изменяться в зависимости от версии ядра операционной системы и от аппаратной архитектуры системы, описание их доступно на странице справочного руководства `signal(7)`. Часть сигналов зарезервирована для использования программами. Например, работающие в неинтерактивном режиме демоны могут устанавливать обработчики сигналов USR1 и/или USR2; при получении таких сигналов от пользователя или других программ они могут, например, перечитать и обновить свою конфигурацию, или закрыть использующиеся ими в данный момент файлы журналов и заново открыть новые. Использование данных сигналов не стандартизировано, поведение конкретных программ нужно уточнять по документации к ним.

Перевод в фоновый режим и уничтожение заданий.

Рассмотрим управление заданиями на простом 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 отправки процессу какого-либо сигнала (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 комбинацию 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). Стыковку 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

стыковки 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
```

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 указать 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 выводиться 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. Задаваемые в шаблоны правила для поиска строк состоят из символов и метасимволов. В простейшем случае шаблон состоит только из обычных символов и для того, чтобы строка удовлетворяла шаблону, она должна содержать в себе эту последовательность символов – это обычный поиск подстроки в строке. Метасимволы в шаблоне позволяют задать дополнительные условия поиска. С использованием их в шаблоне можно определить такие операции, как:

- Перечисление: метасимвол «вертикальная черта» разделяет допустимые варианты. Например, шаблон «`one|two`» соответствует подстрокам *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. Наиболее распространены три варианта синтаксиса регулярных выражений:

- базовые регулярные выражения стандарта POSIX (basic regular expressions, BRE). Хотя этот вариант регулярных выражений на данный момент и определён *POSIX* как устаревший, но он до сих пор широко распространён из соображений обратной совместимости. Многие UNIX-утилиты используют такие регулярные выражения по умолчанию – также исходя из соображений обратной совместимости.
- расширенные регулярные выражения стандарта POSIX (extended regular expressions, ERE). Синтаксис в целом аналогичен BRE, за исключением записи ряда метасимволов. Приведённые выше операции перечисления, группировки и квантификации записаны в синтаксисе ERE, при использовании BRE использованные в них метасимволы `|`, `{`, `}`, `(`, `)`, `?` и `+` требуется предварять символом обратного слеша, «`\`».
- регулярные выражения, совместимые с Perl (Perl-compatible regular expressions, PCRE). Perl — интерпретируемый язык высокого уровня, изначально предназначавшийся для обработки символьной информации. Имеющая в нём реализация регулярных выражений имеет более богатый синтаксис и предоставляет существенно большие возможности по сравнению с регулярными выражениями BRE/ERE. Реализация регулярных выражений PCRE доступна в виде библиотеки и используется во многих языках программирования как стандартная.

Для базовых и расширенных регулярных выражений (BRE, ERE) большинство символов соответствуют сами себе («`a`» соответствует *a* и т.д.). Исключения из этого правила называются метасимволами, для синтаксиса BRE:

`.`	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 *ааа*, *аааа* или *ааааа*.

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 в синтаксисе ERE — «`^ *(#|$)`».

`grep` поддерживает синтаксис регулярных выражений BRE, ERE и PCRE, при этом по умолчанию используется синтаксис BRE. Как говорилось выше, при этом круглые скобки рассматриваются 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 ERE, вызвав его с ключом `-E` (`--extended-regexp` в полной записи имени ключа):

```
# 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, эти утилиты могут сами прочитать файл с диска:

```
# grep -Ev '^ *(#|$)' /etc/lighttpd/lighttpd.conf
```

Для сокращения записи «grep -E» может встречаться версия grep с включённой по умолчанию поддержкой расширенных регулярных выражений — egrep, но в текущих версиях стандартных утилит, к которым относится grep, утилита egrep считается устаревшей, не рекомендованной к использованию и выдающей соответствующее сообщение при её использовании.

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.

По-умолчанию утилита sed также использует базовый вариант синтаксиса регулярных выражений, BRE. Переключение на использование расширенного варианта синтаксиса, ERE, выполнятся аналогично grep — заданием ключа -E (--extended-regexp).

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 (`$VAR`).
- `` (обратные апострофы) — служат выполнения команды, заключённой между ними, и подстановки в строку вывода этой команды. Другой вариант выполнения команды – использование знака доллара и круглых скобок, `$( команда )` .
- \ (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. Разрешено использование как заглавных, так и строчных латинских букв. Имена переменных регистрозависимые, переменные с именами `VAR`, `Var`, `var` – это три разные переменные. 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
```

Если переменная не существует, она создаётся при первом использовании оператора присваивания. Отдельное объявление переменной перед её использованием не требуется. Обращение к несуществующей переменной ошибкой не является, значение такой переменной – пустая строка.

Удалить переменную можно с помощью команды `unset`, передав ей имя удаляемой переменной.

Использование значений переменных.

Обращение к значению переменной осуществляется по её имени, перед которым ставится знак `$` (знак доллара). Язык командного интерпретатора относится к интерпретируемым, строки с командами обрабатываются последовательно по мере их выполнения. Если в строке встречается знак `$` (знак доллара), то следующее слово, состоящее из латинских букв, цифр и символов подчеркивания (по факту, удовлетворяющее регулярному выражению '`\$[A-Za-z0-9_]+`'), рассматривается как имя переменной – значение которой подставляется в строку вместо знака `$` (знак доллара) и этого имени до последующего выполнения команд из этой строки. Таким образом, значения переменных можно использовать при вводе команд, например:

```
$ TMP_DIR=/tmp/tmp_dir
$ mkdir $TMP_DIR
$ echo "TMP_DIR=$TMP_DIR" > $TMP_DIR/tmp_dir
$ cd $TMP_DIR; cat tmp_dir
TMP_DIR=/tmp/tmp_dir
```

Здесь переменной `TMP_DIR` было присвоено значение `/tmp/tmp_dir` (при этом, если такой переменной ранее не было, она была создана), далее был создан каталог `/tmp/tmp_dir` (командный интерпретатор при обработке строки "`mkdir $TMP_DIR`" раскрыл переменную `TMP_DIR` (подставил в строку значение переменной), и далее выполнил получившуюся команду "`mkdir /tmp/tmp_dir`"). Следующая строка после раскрытия переменных

стала "echo "TMP_DIR=/tmp/tmp_dir" > /tmp/tmp_dir/tmp_dir", что привело к записи в файл /tmp/tmp_dir/tmp_dir строки "TMP_DIR=/tmp/tmp_dir".

Раскрытие значений переменных выполняется в обычных строках команд и внутри символьных переменных, заключённых в двойные кавычки. Если раскрытие значений переменных не нужно, то нужно использовать или символьные переменные, заключённые в одинарные кавычки, или экранировать знака $ (знак доллара):

```
$ echo 'TMP_DIR=$TMP_DIR'
TMP_DIR=$TMP_DIR
$ echo \$TMP_DIR
$TMP_DIR
```

Раскрытие значений переменных выполняется один раз, если в записанной в переменную строке есть знак $ (знак доллара), то при раскрытии переменной он второй раз не обрабатывается:

```
$ A='TMP_DIR=$TMP_DIR'
$ echo $A
TMP_DIR=$TMP_DIR
```

При раскрытии значений переменных в строках может встречаться ситуация, когда сразу после имени переменной в строке идёт дополнительная последовательность символов. Командный интерпретатор при обработке строки с командной не учитывает существующие ранее определённые переменные, а обращение к несуществующей переменной ошибкой не является, значение несуществующей переменной – пустая строка. Чтобы указать конкретное имя переменной для оператора раскрытия значения $ и отделить его от последующих не относящихся к имени символов, имя переменной можно заключать в фигурные скобки:

```
$ TMP_DIR=/tmp/tmp_dir
$ mkdir $TMP_DIR1
mkdir: пропущен операнд
По команде «mkdir --help» можно получить дополнительную информацию.
$ mkdir ${TMP_DIR}1
$ ls -l ${TMP_DIR}1
итого 0
```

Здесь при обработке первой команды mkdir интерпретатор команд подставил в строку значение несуществующей переменной $TMP_DIR1 – без указания, какой каталог нужно создавать, mkdir выдал ошибку. Во второй команде mkdir было отдельной указано использовать переменную $TMP_DIR, раскрытие подстроки "${TMP_DIR}1" дало значение "/tmp/tmp_dir1" и успешно создало соответствующий каталог.

Заключать или нет имена переменных при их использовании в фигурные скобки для обычных переменных командного интерпретатора – вопросы используемого стиля программирования.

Вычисление выражений.

В интерпретаторе команд все переменные рассматриваются как строки. Однако есть возможность и вычисления арифметических выражений — через внешние программы.

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.

В интерпретаторе команд *Bash*, использующемся по-умолчанию в большинстве дистрибутивов Linux-систем, имеется аналогичный внешней команде `expr` встроенный оператор для целочисленных вычислений `$(( ))`. С его помощью вышеперечисленные операции можно записать в более привычном виде как:

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

Как операторе `$(( ))`, так и в команде `expr` поддерживаются операции сложения (`+`), вычитания (`-`), умножения (`*`), деления (`/`) и получения остатка от деления (`%`). Все операции проводятся 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.

Чтение потока ввода в переменную.

Для чтения строки из потока ввода и записи её в переменную используется встроенная команда интерпретатора `read`. Команда `read` читает строку из потока ввода – по-умолчанию с терминала. В качестве параметра задаётся имя переменной, в которую помещается полученная строка:

```
$ echo "Введите число:";read A
```

```
Введите число:
100
$ echo $A
100
```

Здесь после вывода командой echo приглашения с терминала вводится число *100*, ввод завершается нажатием на <Enter> – полученная строка запоминается в переменной A.

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!
```

Как и `test`, команда `[` – это обычная программа, запускаемая и выполняющаяся интерпретатором команд. В интерпретаторе команда Bash есть внутренняя (встроенная) команда `[[` , с аналогичным синтаксисом, не требующая запуска внешней программы и выполняющая несколько быстрее:

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

Для построения условных выражений можно использовать команды `true` и `false`. Они предназначены для получения кодов возврата `0` и `1` соответственно, при запуске они завершаются с выдачей соответствующих кодов возврата без вывода чего-либо в терминал или других действий:

```
$ /bin/true; echo $?
0
$ /bin/false; echo $?
1
```

Также есть встроенная команда интерпретатора *Bash* ":", также выдающая код возврата `0`. Чаще всего она используется в виде составного выражения '`||:`' в конце командной строки, где "`||`" - рассмотренный ранее оператор `ИЛИ`. Такое выражение позволяет игнорировать возникающие при

выполнении команды ошибки и всегда возвращать в результате её выполнения код возврата 0, например:

```
$ /bin/false ||:
$ echo $?
0
```

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. Какой именно ошибке 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. Как говорилось выше, последовательность символов '`||:`' интерпретируется как оператор «*или*» и пустой оператор '`:`', всегда возвращающий нулевое значение.

Области видимости переменных

При входе пользователя в систему и запуске командного интерпретатора выполняется ряд конфигурационных скриптов, которые задают начальные значения переменных окружения. Когда интерпретатор команд запускает какую-либо программу, он создаёт копию имеющихся у него переменных окружения, и передаёт их программе. В процессе своей работы программа может как читать, так и изменять переданные ей при запуске переменные окружения, а также создавать новые переменные окружения – на значения переменных окружения в процессе интерпретатора команд эти действия влияния не оказывают.

Создаваемые с помощью оператора присваивания в командной строке значения переменных к переменным окружения не относятся и в запускаемые программы не передаются.

Для выполнения скрипта запускается отдельный экземпляр командного интерпретатора – соответственно, сказанное выше относится и к переменным, доступным и задаваемым внутри скриптов.

Рассмотрим данное поведение на примере запуска скрипта `test.sh`:

```
#!/bin/bash
while read l;do L=$(( L+1 )); echo "Строка $L: $l"; done
echo "Всего строк: $L"
```

Данный скрипт читает в цикле поток ввода, запоминая прочитанные строки в переменную `l`. Внутри цикла инкриминируется значение переменной `L`, и выводится текущее значение переменных `L` и `l`. Условие выхода из цикла является исчерпание данных в потоке ввода (при запуске скрипта и чтении строк с терминала – ввод последовательности «`Ctrl-D`»).

Пусть есть файл `test.txt` из двух строк:

```
$ cat test.txt
Первая строка
Вторая строка
```

Переменная `L` в скрипте `test.sh` умышленно не инициализирована, значение не инициализированной переменной – пустая строка. При запуске скрипта из командной строки с перенаправлением в его поток входа файла `test.txt` будет выведено:

```
$  cat test.txt | ./test.sh
Строка 1: Первая строка
Строка 2: Вторая строка
Всего строк: 2
```

Задаваемые в командной строке с помощью оператора присваивания переменные к переменным окружения не относятся и в запускаемый скрипт не передаются:

```
$ L=1000
$ cat test.txt | ./test.sh
Строка 1: Первая строка
Строка 2: Вторая строка
Всего строк: 2
```

Если нужно использовать переменную внутри скрипта, её надо определить как переменную окружения с использованием команды `export`:

```
$ export L=1000
$ cat test.txt | ./test.sh
Строка 1001: Первая строка
Строка 1002: Вторая строка
Всего строк: 1002
```

Здесь переменная `L` объявлена как переменная окружения, и была передана вместе с остальными переменными окружения в скрипт `test.sh` . Т.к. внутри скрипта переменная `L` не инициализирована перед использованием – её начальным значением стало значение из переменных окружения.

Второй способ передать переменную из командной строки в запускаемую программу – задать её непосредственно в строке запуска команды:

```
$ unset L
$ L=2000 ./test.sh < test.txt
Строка 2001: Первая строка
Строка 2002: Вторая строка
Всего строк: 2002
```

Здесь определённая ранее переменная `L` была удалена командной `unset`, и далее задана при запуске скрипта – и также стала доступна внутри него.

При этом в вариантах запуска скрипта

```
$ L=3000; ./test.sh < test.txt
Строка 1: Первая строка
Строка 2: Вторая строка
Всего строк: 2

$ L=3000 cat test.txt | ./test.sh
Строка 1: Первая строка
Строка 2: Вторая строка
Всего строк: 2
```

переменная `L` внутрь скрипта не попадает: в первом случае её определение и запуск скрипта — это две разные последовательно выполняющиеся команды, во втором – командный интерпретатор запускает две команды (`cat` и `./test.sh`), переменная `L` определяется для окружения команды `cat`.

Также надо учитывать, что и внутри скриптов могут запускаться отдельные команды и дополнительные экземпляры интерпретатора команд, в частности, изменённый следующим образом скрипт `test.sh`

```
#!/bin/bash
L=0
cat test.txt | while read l;do L=$((L+1));echo "Строка $L: $l";done
echo "Всего строк: $L"
```

запускает внутри себя для цикла `while` отдельный экземпляр командного интерпретатора и изменяемое в цикле значение `L` будет доступно только внутри цикла:

```
$ ./test.sh
Строка 1: Первая строка
Строка 2: Вторая строка
Всего строк: 0
```

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 примере 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.
`@reboot /bin/false`	Запускать при запуске демона crond, т.е. обычно во время загрузки системы.
`@monthly /bin/false`	Запускать каждое первое число месяца, сокращение для "0 0 1 * *"
`@weekly /bin/false`	Запускать каждое воскресенье, сокращение для "0 0 * * 0"
`@dayly /bin/false`	Запускать каждый день в полночь, сокращение для "0 0 * * *"
`@hourly /bin/false`	Запускать каждый час, сокращение для "0 * * * *"

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
```

Имеет смысл отметить, что система инициализации systemd может обеспечивать периодическое выполнение заданий самостоятельно, без использования демона crond. Описание возможностей планировщика заданий systemd можно найти на странице справочного руководства man systemd.timer. По сравнению с демоном crond планировщик заданий systemd обеспечивает большую гибкость задания временных интервалов и частоты выполнения заданий, умеет отслеживать ход их выполнения, может учитывать зависимости между разными заданиями. Как следствие, создание заданий в планировщике systemd существенно сложнее, чем для демона crond, и в настоящей лабораторной работе не рассматривается.

Вполне возможно использовать одновременно и планировщик заданий systemd, и отдельный демон crond — для разных заданий.

Выполнение лабораторной работы.

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

В лабораторной работе требуется:

- 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.

В лабораторной работе требуется получить, записать и проанализировать ряд значений, относящихся к локальной системе виртуального сервер, к удалённому сетевому оборудованию, а также доступные из виртуального сервера параметры работы физического сервера, на котором выполняется виртуальная машина.

Для локальной системы требуется получить, сохранить и обработать

- число процессов в локальной 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 `ext` локальной системы данных в байтах. Эти значения содержатся в выводе команды `netstat -i`, в соответствующих полях выдаваемой таблицы.

Для сетевого оборудования нужно получить, сохранить и обработать

- число переданных и принятых через порт удалённого коммутатора пакетов и байтов данных. Данные величины могут быть получены по протоколу *SNMP* с использованием программы `snmpget`.

Для системы физического сервера нужно получить, сохранить и обработать

- определённые данные о текущем состоянии работы системы – загрузке процессора локальной системы, использования ОЗУ, параметры планировщика процессов и т. д. Соответствующие величины могут быть получены чтением данных из файловой системы `procfs`.

Вызов программы `snmpget` для получения данных по протоколу SNMP 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.

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`.

IP-адрес коммутатора и номер сетевого интерфейса, а также перечень требуемых данных о текущем состоянии работы физического сервера индивидуальны для каждого виртуального сервера. Они размещены в репозитории Git с именем `lab-NN`, соответствующем номеру виртуальной машины (см. лабораторную работу №1). Задание на лабораторную работу размещено в ветке репозитория `task/lab2`, для его просмотра требуется обновить репозиторий с сервера Git и перейти на указанную ветку. Данные для доступа к сетевому коммутатору размещены в файле `SNMP.txt`, перечень требуемых данных о текущем состоянии работы системы, а также

имена и описания формата содержащих их файлов в файловой системе `profcs` — в файле `HN.txt` . Обновление состояния репозитория с сервера Git и переключение на ветку `task/labs` выполняется командами вида

```
$ git pull --all
$ git checkout task/lab2
```

Для получения данных локальной системы предлагается использовать следующие программы:

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

```
#!/bin/bash
# Script for logging current system status:
# - number of processes
# - RX and TX bytes over ext 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 '^ext[[:blank:]]' | awk '{print "RX ",$4,"bytes, TX
",$8,"bytes."}'`

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

log-local-rrd.sh — получение и запись в файл RRD статистики локальной системы:

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

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

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

# netstat info
NETBYTES=`netstat -i | grep '^ext' | awk '{print $4":"$8}'`
```

```
# Log all to the file
if [ -f "$LOG_FILE" ]; then
   rrdtool update "$LOG_FILE" N:$PROCNUM:$NETBYTES
else
   # Create file
   rrdtool create "$LOG_FILE" --step 60 \
       DS:procs:GAUGE:120:0:1000 \
       DS:RX:DERIVE:120:0:4294967295 \
       DS:TX:DERIVE:120:0:4294967295 \
       RRA:AVERAGE:0.5:1:2880 \
       RRA:AVERAGE:0.5:30:672 \
       RRA:AVERAGE:0.5:120:732 \
       RRA:AVERAGE:0.5:720:1460
fi
```

Данные, собираемые и записываемые скриптом `log-local.sh`, пригодны для вывода в текстовом виде. Данные, собираемые и записываемые скриптом `log-local-rrd.sh`, предназначены для построения графиков средствами утилит `RRDTOOLS`. Запуск скриптов получения данных из локальной системы предполагается осуществлять раз в минуту.

Для получения данных с сетевого оборудования по протоколу `SNMP` предлагается использовать следующие программы:

log-snmp.sh — получение и запись в текстовый 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"
#------------------------------------------------------------
```

log-snmp-rrd.sh — получение и запись в файл RRD 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.rrd

# Network interface number:
N=48

# SNMP host
HOST=192.168.222.144

# 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"

############################
# snmp info
RES='N'

for MIB in $MIB2 $MIB3 $MIB4 $MIB5; do
    LINE=`snmpget -c $COMMUNITY -v 1 $HOST $MIB`
    VALUE=`echo "$LINE" | sed "s/^IF-MIB::[[:alnum:]]\+\.$N = [[:alnum:]]\
+: //"`
    RES="$RES:$VALUE"
done

# Log all to the file
if [ -f "$LOG_FILE" ]; then
   rrdtool update "$LOG_FILE" "$RES"

else
```

```
   # Create file
   rrdtool create "$LOG_FILE" --step 120 \
       DS:ifInOctets:DERIVE:240:0:4294967295 \
       DS:ifInUcastPkts:DERIVE:240:0:4294967295 \
       DS:ifOutOctets:DERIVE:240:0:4294967295 \
       DS:ifOutUcastPkts:DERIVE:240:0:4294967295 \
       RRA:AVERAGE:0.5:1:2880 \
       RRA:AVERAGE:0.5:30:672 \
       RRA:AVERAGE:0.5:120:732 \
       RRA:AVERAGE:0.5:720:1460
fi

#---------------------------------------------------------
```

Запуск скриптов для получения данных с коммутатора через протокол *SNMP* предполагается осуществлять раз в две минуты.

Для получения данных о текущем состоянии работы физического сервера предлагается адаптировать скрипты получения данных локальной системы `log-local.sh` и `log-local-rrd.sh` с учётом информации об анализируемых параметрах системы и форматах данных в `procfs` из полученного задания. Для выделения нужных значений из считываемых файлов предполагается использование регулярных выражений и рассматриваемых в настоящей лабораторной работе утилит `grep`, `sed` и `awk`. Периодичность запуска скрипта получения данных указана в задании.

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

cgi-local.sh — отображение статистики локальной системы в текстовом виде.

```
#!/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 в текстовом виде.

```
#!/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 как приведённых выше скриптов, так и скриптов, обеспечивающих вывод данных в табличной форме и построение графиков,, размещён в репозитории `scripts4lab2` на сервере Git (см. лабораторную работу №1). Примеры работы скриптов приведёны на странице с примерами к настоящей лабораторной работе – `http://lab-00.edu.cbias.ru/` .

Для отображения данных о текущем состоянии работы системы предлагается адаптировать скрипты отображения данных локальной системы `cgi-local.sh` и `cgi-local-html.sh`. Если настроен сбор данных статистики локальной системы в графическом виде, то соответствующий скрипт отображения их рекомендуется создавать на базе `cgi-local.rrd` .

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 создаваемых скриптами отображения данных временных графических файлов используется подкаталог внутри `document_root` веб-сервера. В составе репозитория `scripts4lab2` есть скрипт `install.sh`, который создаёт каталоги по описанной схеме и размещает в них скрипты сбора и отображения данных.

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 подключеном 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" )
```

Если такого параметра в основном конфигурационном файле веб-сервера lighttpd нет, проверьте наличие параметра server.document-root в подключаемых через директивы include дополнительных файлах конфигурации или обратитесь к документации веб-сервера lighttpd.

Изменения настроек вступают в силу после перезапуска lighttpd.

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

В ходе лабораторной работы требуется установка в систему и редактирование скриптов сбора и сохранения данных, а также отображения собранных данных в ответ на удалённые запросы, поступающие из браузера и обрабатываемыми веб-сервером. Потенциальные ошибки, допущенные при редактировании скриптов, или их некорректной работе могут привести к нежелательным последствиям – например, удалению или перезаписи каких-либо файлов системы. Поэтому следует минимизировать использование при работе учётной записи суперпользователя выполнением необходимых системных настроек, а создание, редактирование и отладку работы скриптов проводить под обычной непривилегированной учётной записью пользователя.

Кроме того, для обеспечения безопасности системы должны соблюдаться определённые правила выполнения скриптов.

Сбор статистики должен выполняться от имени отдельного 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.

Использование описанной схемы разграничения прав доступа позволит:

- проводить необходимые изменения скриптов сбора и сохранения данных, а также скриптов отображения собранных данных под учётной записью пользователя;
- выполнение скриптов сбора и сохранения данных под отдельной учётной записью псевдопользователя, имеющей права только на сбор и запись собираемых данных в соответствующие файлы. Какие-либо ошибки в скриптах при этом не могут привести к изменениям ни самих скриптов, ни других файлов виртуального сервера;
- выполнение скриптов отображения данных под отдельной учётной записью веб-сервера lighttpd, имеющей только права на чтение для файлов собираемой статистики. Какие-либо ошибки в скриптах отображения данных, или в самом веб-сервере при этом не могут позволить внести какие-либо изменения в запускаемые от других пользователей процессы, или как-либо изменить собираемые данные.

Конкретная реализация этой схемы разграничения прав доступа входит в задание на лабораторную работу.

Для удобного доступа к различным скриптам в каталоге `document_root` веб-сервера предлагается разместить индексный файл с названием `index.html` , также приведённый в репозитории `scripts4lab2`.

Доработанные в ходе лабораторной работы скрипты требуется разместить в репозитории Git `lab-NN`, в отдельной ветке с именем «`lab2`». Изменения скриптов в ходе доработки должны отражаться в коммитах репозитория. По завершению работы требуется опубликовать изменения репозитория `lab-NN` на сервере Git. При желании можно также доработать скрипт `install.sh` для учёта выбранной схемы разграничения прав доступа.

По итогам выполнения лабораторной работы требуется подготовить отчёт. Требования к содержанию отчёта приведены ниже. Отчёт в формате PDF требуется разместить в репозитории Git lab-NN, ветка master – рядом с отчётом по лабораторной работе № 1. Требуемое имя файла отчёта — `report-2.pdf` .

После размещения отчёта в каталоге рабочей копии репозитория его следует добавить в рабочую копию Git, запомнить изменения в новом коммите и передать изменения на сервер репозиториев Git:

```
$ git add report-2.pdf
$ git commit
$ git push --all
```

В случае необходимости скорректировать отчёт требуется повторить вышеперечисленные шаги по передаче файла отчёта на сервер, запоминанию его в репозитории и публикации изменений.

Задания на лабораторную работу.

1. Обновить систему из репозиториев *APT*, доставить всё необходимое программное обеспечение.
2. Адаптировать приведённые в описании работы скрипты, получающие значения статистических параметров и записывающие их в журналы.
3. Создать скрипт для сбора данных о текущем состоянии работы системы согласно полученного с тестового сервера SSH задания.
4. Обеспечить периодическое регулярное выполнение скриптов.
5. Адаптировать приведённые в описании работы скрипты для отображения записываемых в пп. 2-4 данных из журналов, обеспечить их выполнение из командной строки.
6. Настроить `lighttpd` для удалённого обращения из браузера к указанным скриптам и отображения собираемых данных в веб-браузере на удалённом рабочем месте.
7. Обеспечить безопасное выполнение скриптов.
8. Разместить доработанные скрипты в репозитории и опубликовать их на сервере Git.

Контрольные вопросы.

1. Что такое потоки ввода/вывода? Как можно перенаправить поток ввода, поток вывода?
2. Что такое скрипт, как создать скрипт и разрешить его выполнение?
3. Что такое переменная окружения, как посмотреть значение переменной окружения?
4. Как определить и использовать переменную *shell*?
5. Какие управляющие конструкции доступны в языке командного интерпретатора?
6. Что такое регулярное выражение?
7. Какие основные конструкции используются в регулярных выражениях?
8. Как организовать периодическое выполнение программ?
9. Объясните порядок работы скриптов, использованных в лабораторной работе для получения и вывода данных.
10. Поясните, под какими учётными записями пользователей выполняются установленные в лабораторной работе скрипты и как ограничен доступ к используемым ими каталогам выбранными правами доступа.

Содержание отчёта о выполнении лабораторной работы

В состав отчёта о выполнении лабораторной работы должны входить:

0. Титульная страница отчёта, с правильным полным названием учебного заведения, подразделения и кафедры, учебного курса, названия лабораторной работы, данных о выполнившем лабораторную работу студенте, данных о проверяющем лабораторную работу преподавателе. Титульная страница отчёта не должна содержать фактических, грамматических и синтаксических ошибок, стиль оформления должен соответствовать общеинститутским требованиям.
1. Снимок экрана с выводом списка файлов из каталога (или каталогов) с размещёнными в нём скриптами сбора значений статистических показателей. На снимке экрана должен быть виден абсолютный путь к использованному для размещения скриптов каталогу, а также права доступа к этому каталогу и к файлам в нём.
2. Снимок экрана с выводом списка файлов из каталога (или каталогов) с размещёнными в нём файлами журналов собираемых значений статистических показателей. На снимке экрана должен быть виден абсолютный путь к использованному для размещения файлов журналов каталогу, а также права доступа к этому каталогу и к файлам в нём.
3. Снимок экрана с несколькими (5-20) записями из системного журнала journald, относящиеся к регулярному запуску скриптов сбора данных и показывающих запуск этих скриптов в системе. Лишние и не относящиеся к запуску скриптов сбора данных строки журнала требуется отфильтровать. На снимке экрана должна быть видна использовавшаяся для получения данного списка записей системного журнала команда.
4. Снимок экрана с выводом списка файлов из каталога (или каталогов) с размещёнными в нём скриптами вывода значений статистических показателей. На снимке экрана должна быть виден абсолютный путь к использованному для размещения скриптов каталогу, а также права доступа к этому каталогу и к файлам в нём.
5. Снимок экрана браузера с любой страницей, содержащей данные о текущем состоянии работы физического сервера. В случае настройки вывода данных в графическом виде должна быть представлена страница с графиками получаемых данных. Снимок экрана должен включать в себя адресную строку браузера с отображением в ней URL запрошенной страницы, и собственно полученную страницу.

Формат отчёта о выполнении лабораторной работы – PDF, требуемое имя файла отчёта — `report-2.pdf` .

Отчёт требуется разместить в репозитории Git и опубликовать изменения на сервере репозиториев `git.edu.cbias.ru` .

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

8. 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