Лабораторная работа № 7. Ввод и вывод

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

Каждая программа работает с данными определённого типа: текстовыми, графическими, звуковыми и т. п. Как, наверное, уже стало понятно из предыдущих занятий, основной интерфейс управления системой в Linux -- это **терминал**, который предназначен для передачи текстовой информации от пользователя системе и обратно. Поскольку ввести с терминала и вывести на терминал можно только текстовую информацию, то ввод и вывод программ, связанных с терминалом[1], тоже должен быть текстовым. Однако необходимость оперировать с текстовыми данными не ограничивает возможности управления системой, а, наоборот, расширяет их. Человек может *прочитать* вывод любой программы и разобраться, что происходит в системе, а разные программы оказываются совместимыми между собой, поскольку используют один и тот же вид представления данных -- текстовый. Возможностям, которые даёт Linux при работе с данными в текстовой форме, и посвящено данное занятие.

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

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

---

[1] Т. е. в первую очередь командной оболочки и её процессов-потомков

```
[methody@localhost methody]$ hexdump -C /bin/cat | less
 00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
 00000010  02 00 03 00 01 00 00 00  90 8b 04 08 34 00 00 00  |............4...|
 00000020  e0 3a 00 00 00 00 00 00  34 00 20 00 07 00 28 00  |Ю:......4. ...(.|
  . . .
 00000100  00 00 00 00 00 00 00 00  00 00 00 00 06 00 00 00  |................|
 00000110  04 00 00 00 2f 6c 69 62  2f 6c 64 2d 6c 69 6e 75  |..../lib/ld-linu|
 00000120  78 2e 73 6f 2e 32 00 00  04 00 00 00 10 00 00 00  |x.so.2..........|
 00000130  01 00 00 00 47 4e 55 00  00 00 00 00 02 00 00 00  |....GNU.........|
  . . .
[methody@localhost methody]$ strings /bin/cat | less
 /lib/ld-linux.so.2
  Jv RegisterClasses
   gmon start
 libc.so.6
 stdout
  . . .
[methody@localhost methody]$ strings -n3 /bin/cat | less
 /lib/ld-linux.so.2
 GNU
 _Jv_RegisterClasses
   gmon start
 libc.so.6
 stdout
  . . .
```

Использование `hexdump` и `strings`

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

Перенаправление ввода и вывода

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

запись и закрытие, работают с дескриптором, как с *потоком данных*, а куда именно ведёт этот поток, неважно.

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

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

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

Стандартный вывод

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

```
[methody@localhost methody]$ cat > textfile
 Это файл для примеров.
 ^D
 [methody@localhost methody]$ ls -l textfile
 -rw-r--r--  1 methody methody 23 Ноя 15 16:06 textfile
```

Перенаправление стандартного вывода в файл

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

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

```
[methody@localhost methody]$ cat >> textfile
 Пример 1.
^D
[methody@localhost methody]$ cat textfile
 Это файл для примеров.
 Пример 1.
[methody@localhost methody]$
```

Недеструктивное перенаправление стандартного вывода

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

стандартный вывод	Поток данных, открываемый системой для каждого процесса в момент его запуска, и предназначенный для данных, выводимых процессом.

Стандартный ввод

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

```
[methody@localhost methody]$ sort < textfile
 Пример 1.
 Это файл для примеров.
[methody@localhost methody]$
```

Перенаправление стандартного ввода из файла

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

Стоит помнить, что операция ">" *деструктивна*: она всегда создаёт файл нулевой длины. Поэтому для, допустим, сортировки данных *в файле* надо применять последовательно `sort < файл > новый_файл` и `mv новый_файл файл`. Команда вида `команда < файл > тот_же_файл` просто урежет его до нулевой длины!

стандартный ввод	Поток данных, открываемый системой для каждого процесса в момент его запуска, и предназначенный для ввода данных.

Стандартный вывод ошибок

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

```
[methody@localhost methody]$ info cat > cat.info
 info: Запись ноды (coreutils.info.bz2)cat invocation...
 info: Завершено.
[methody@localhost methody]$ head -1 cat.info
 File: coreutils.info,  Node: cat invocation,  Next: tac invocation,  Up: Output of entire files
[methody@localhost methody]$
```

Стандартный вывод ошибок

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

в ходе выполнения программы, в Linux предусмотрен **стандартный вывод ошибок** (сокращённо -- stderr).

стандартный вывод ошибок	Поток данных, открываемый системой для каждого процесса в момент его запуска, и предназначенный для диагностических сообщений, выводимых процессом.

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

```
[methody@localhost methody]$ info cat > cat.info 2> cat.stderr
[methody@localhost methody]$ cat cat.stderr
 info: Запись ноды (coreutils.info.bz2)cat invocation...
 info: Завершено.
[methody@localhost methody]$
```

Перенаправление стандартного вывода ошибок

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

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

```
[methody@localhost methody]$ info cat > cat.info 2>&1
[methody@localhost methody]$ head -3 cat.info
 info: Запись ноды (coreutils.info.bz2)cat invocation...
 info: Завершено.
 File: coreutils.info,  Node: cat invocation,  Next: tac invocation,  Up: Output of entire files
[methody@localhost methody]$
```

Объединение стандартного вывода и стандартного вывода ошибок

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

написана на shell, в ней бы использовались перенаправления вида "`>&4`". Чтобы не набирать громоздкую конструкцию "`> файл 2>&1`" в `bash` используются сокращения: "`&> файл`" или, что то же самое, "`>& файл`".

Перенаправление в никуда

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

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

Перенаправление в `/dev/null`

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

Обработка данных в потоке

Конвейер

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

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

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

Простейший конвейер

Можно последовательно обработать данные несколькими разными программами, перенаправляя вывод на ввод следующей программе и организуя сколь угодно длинный **конвейер** для обработки данных. В

результате получаются очень длинные командные строки вида "`cmd1 | cmd2 | ... | cmdN`", которые могут показаться громоздкими и неудобными, но оказываются очень полезными и эффективными при обработке большого количества информации, как мы увидим далее в этой лекции.

Организация конвейера устроена в shell по той же схеме, что и перенаправление в файл, но с использованием особого объекта системы -- **канала**. Если файл можно представить в виде Коробки с Данными, снабжённой Клапаном для Чтения или Клапаном для Записи, то **канал** -- это оба Клапана, приклеенные друг к другу вообще без Коробки. Для определённости между Клапанами можно представить Трубу, немедленно доставляющую данные от входа к выходу (английский термин -- "pipe" -- основан как раз на этом представлении, а в роли Трубы выступает, конечно же, сам Linux). Каналом пользуются сразу два процесса: один пишет туда, другой читает. Связывая две команды конвейером, shell открывает **канал** (заводится *два* дескриптора -- входной и выходной), подменяет по уже описанному алгоритму **стандартный вывод** первого процесса на входной дескриптор канала, а **стандартный ввод** второго процесса -- на выходной дескриптор канала. После чего остаётся запустить по команде в этих процессах и стандартный вывод первой попадёт на стандартный ввод второй.

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

Фильтры

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

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

В любом дистрибутиве Linux присутствует набор стандартных утилит, предназначенных для работы с файловой системой и обработки текстовых данных. Многими из них Мефодий уже успел воспользоваться: это `who`, `cat`, `ls`,

`pwd`, `cp`, `chmod`, `id`, `sort` и др. Мефодий уже успел заметить, что каждая из этих утилит предназначена для исполнения какой-то *одной* операции над файлами или текстом: вывод списка файлов в каталоге, копирование, сортировка строк, хотя каждая утилита может выполнять свою функцию несколько по-разному, в зависимости от переданных ей ключей и параметров. При этом все они работают со стандартными потоками ввода/вывода, поэтому хорошо приспособлены для построения конвейеров: последовательно выполняя простые операции над потоком данных, можно решать довольно нетривиальные задачи.

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

Структурные единицы текста

Работу в системе Linux почти всегда можно представить как работу с текстами. Поиск файлов и других объектов системы -- это получение от системы *текста* особой структуры -- списка имён. Операции над файлами: создание, переименование, перемещение, а также сортировка, перекодировка и прочее, означает замену одних *символов* и *строк* на другие либо в каталогах, либо в самих файлах. Настройка системы в Linux сводится непосредственно к работе с текстами -- редактированию **конфигурационных файлов** и написанию **сценариев**.

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

Строки

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

Поля

В одной строке может упоминаться и больше одного объекта. Если понимать объект как последовательность символов из определённого набора (например, букв), то строку можно рассматривать как состоящую из слов и **разделителей**[2]. В этом случае текст от начала строки до первого разделителя -- это первое **поле**, от первого разделителя до второго -- второе поле и т. д. В качестве разделителя можно рассматривать любой символ, который не может использоваться в объекте. Например, если в пути "`/home/methody`" разделителем является символ "`/`", то первое поле пусто, второе содержит слово "`home`", третье -- "`methody`". Некоторые утилиты позволяют выбирать из строк отдельные поля (по номеру) и работать со строками как с таблицей: выбирать и объединять нужные колонки и проч.

Символы

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

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

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

Примеры задач

Этот раздел посвящён нескольким примерам использования стандартных утилит для решения разных типичных (и не очень) задач. Эти примеры не следует воспринимать как исчерпывающий список возможностей, они приведены просто для демонстрации того, как можно организовать

---

[2] Например, в командной строке разделителями являются символы пробела и табуляции

обработку данных при помощи конвейера. Чтобы освоить их, нужно читать руководства и экспериментировать.

Подсчёт

В европейской культуре очень большим авторитетом пользуются точные числа и количественные оценки. Поэтому пользователю часто бывает любопытно и даже необходимо точно посчитать что-нибудь многочисленное. Компьютер как нельзя более удобен для такой процедуры. Стандартная утилита для подсчёта строк, слов и символов -- `wc` (от англ. "word count" -- "подсчёт слов"). Однако Мефодий запомнил, что в Linux многое можно представить как слова и строки, и решил с её помощью посчитать свои файлы.

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

Подсчёт файлов при помощи `find` и `wc`

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

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

Отбрасывание ненужного

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

Допустим, Мефодию потребовалось получить список всех файлов и подкаталогов в "`/etc`", которые принадлежат группе "`adm`". И при этом ему почему-то нужно, чтобы найденные файлы в списке были представлены не

полным путём, а только именем файла (скорее всего, это требуется для последующей автоматической обработки этого списка).

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

Извлечение отдельного поля

Если команда получается такой длинной, что её неудобно набирать в одну строку, можно разбить её на несколько строк, не передавая системе: для этого в том месте, где нужно продолжить набор со следующей строки, достаточно поставить символ "`\`" и нажать `Enter`. При этом в начале строки `bash` выведет символ "`>`", означающий, что команда ещё не передана системе и набор продолжается[3]. Для системы безразлично, в сколько строк набрана команда, возможность набирать в несколько строк нужна только для удобства пользователя.

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

Выбор нужного

Поиск

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

```
[methody@susanin methody]$ ps aux | grep bash
 methody   3459  0.0  3.0  2524 1636 tty2    S    14:30   0:00 -bash
 methody   3734  0.0  1.1  1644  612 tty2    S    14:50   0:00 grep bash
```

Поиск строки в выводе программы

---

[3] Вид этого приглашения определяется значением переменной окружения "`PS2`"

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

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

Поиск по регулярному выражению

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

```
[methody@susanin methody]$ info grep > grep.info 2> /dev/null
[methody@susanin methody]$ grep -on "\*Note[^:]*::" grep.info
 324:*Note Grep Programs::
 684:*Note Invoking::
[methody@susanin methody]$
```

Поиск ссылок в файле info

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

В регулярном выражении большинство символов обозначают сами себя, как если бы мы искали обыкновенную текстовую строку, например, "`Note`" и "`::`" в регулярном выражении соответствуют строкам "`Note`" и "`::`" в тексте.

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

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

На языке регулярных выражений можно также обозначить "любой символ" ("."), "одно или более совпадений" ("+"), начало и конец строки ("`^`" и "`$`" соответственно) и т. д. Благодаря регулярным выражениям можно автоматизировать очень многие задачи, которые в противном случае потребовали бы огромной и кропотливой работы человека. Более подробные сведения о возможностях языка регулярных выражений можно получить из руководства `regex(7)`. Однако руководство -- это не учебник по использованию, поэтому чтобы научиться экономить время и усилия при помощи регулярных выражений, полезно прочесть соответствующие главы книги `[[Курячий:2004]]` и книгу `[[Фридл:2000]]`.

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

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

Замены

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

Для замены одних символов на другие предназначена утилита `tr` (сокращение от англ. "translate", "преобразовывать, переводить"), работающая как фильтр. Мефодий решил употребить её прямо по назначению и выполнить при её помощи транслитерацию -- замену латинских символов близкими по звучанию русскими.

```
[methody@localhost methody]$ cat cat.info | tr abcdefghijklmnopqrstuvwxyz абцдефгийклмнопкрстуввсиз
> | tr ABCDEFGHIJKLMNOPRSTUVWXYZ АБЦДЕФГХИЙКЛМНОПКРСТУВВСИЗ | head -4
 Филе: цореутилс.инфо,  Ноде: цат инвоцатион,  Нест: тац инвоцатион,  Тп: Оутпут оф ентире филес

 `цат': Цонцатенате амд врите филес
 ==================================
[methody@localhost methody]$
```

Замена символов (транслитерация)

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

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

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

```
[methody@localhost methody]$ ls -l | sed s/^-[-rwx]*/Файл:/ | sed s/^d[-
rwx]*/Каталог:/
 итого 124
 Файл:  1 methody methody  2693 Ноя 15 16:09 cat.info
 Файл:  1 methody methody    69 Ноя 15 16:08 cat.stderr
 Каталог:  2 methody methody  4096 Ноя 15 12:56 Documents
 Каталог:  3 methody methody  4096 Ноя 15 13:08 examples
 Файл:  1 methody methody 83459 Ноя 15 16:11 grep.info
 Файл:  1 methody methody    26 Ноя 15 13:08 loop
 Файл:  1 methody methody    23 Ноя 15 13:08 script
 Файл:  1 methody methody    33 Ноя 15 16:07 textfile
 Каталог:  2 methody methody  4096 Ноя 15 12:56 tmp
 Файл:  1 methody methody    32 Ноя 15 13:08 to.sort
[methody@oblomov methody]$
```

Замена по регулярному выражению

У `sed` очень широкие возможности, но довольно непривычный синтаксис, например, замена выполняется командой "`s/что_заменять/на_что_заменять/`". Чтобы в нём разобраться, нужно обязательно прочесть руководство `sed(1)` и знать регулярные выражения.

Упорядочение

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

```
[methody@localhost methody]$ cat grep.info | tr "[:upper:]" "[:lower:]" | tr
"[:space:][:punct:]" "\n" \
> | sort | uniq -c | sort -nr | head -8
  15233
    720 the
    342 of
    251 to
    244 a
    213 and
    180 or
    180 is
[methody@localhost methody]$
```

Получение упорядоченного по частотности списка словоупотреблений

Мефодий (вернее, компьютер по плану Мефодия) подсчитал, сколько раз какие слова были употреблены в файле "`grep.info`" и вывел несколько самых частотных с указанием количества употреблений в файле. Для этого потребовалось сначала заменить все большие буквы маленькими, чтобы не было разных способов написания одного слова, затем заменить все пробелы и знаки препинания концом строки (символ "`n`"), чтобы в каждой строке было ровно по одному слову (Мефодий всюду взял параметры `tr` в кавычки, чтобы `bash` не понял их неправильно). Потом список был отсортирован, все повторяющиеся слова заменены одним словом с указанием количества

повторений ("`uniq -c`"), затем строки снова отсортированы по убыванию чисел в начале строки ("`sort -nr`") и выведены первые 8 строк ("`head -8`").

Запуск команд

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

```
[methody@localhost methody]$ find /bin -type f -perm +a=x \
> | xargs grep -l -e '^#! \?/' 2> /dev/null
 /bin/egrep
 /bin/fgrep
 /bin/unicode_start
 /bin/bootanim
[methody@localhost methody]$
```

Поиск всех исполняемых файлов, которые точно являются сценариями

Здесь Мефодий решил определить, какие из исполняемых файлов в каталоге "`/bin`" являются сценариями. Для этого он нашёл все обычные исполняемые файлы (указывать "`-type f`" -- "обычные файлы" потребовалось, чтобы в результат не попали каталоги, которые все исполняемые), а затем для каждого найденного файла вызвал `grep`, чтобы поискать в нём сочетание символов "`#!/`" в начале строки. Ключ "`-l`" велел `grep` выводить не обнаруженную строку, а имя файла, в котором найдено совпадение. Так Мефодий получил список исполняемых файлов, в которых есть строка с указанием интерпретатора -- несомненных сценариев[5].

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается опасность вывода нетекстовых данных на терминал?
2. Назначение утилит `hexdump` и `strings`
3. Понятие дескриптора потока. Какие дескрипторы выдаются потоку при его запуске?
4. Отличия операций “>” и “>>”.
5. Понятие стандартный вывод
6. Назначение операции “<”.
7. Понятие стандартны ввод
8. Понятие стандартный вывод ошибок
9. Понятие перенаправления в никуда
10. Понятие конвейер. Назначение символа “|”
11. Понятие канал
12. Понятие фильтр
13. Пояснить особенности структурных единиц текста: строка, поле, символ

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

14. Описать порядок выполнения: подсчета, отбрасывания ненужного, поиска, поиска по регулярному выражению, замены, упорядочения, запуск команд