Лекция 14. Файловая система. Командные интерпретаторы

Коновалов А. В.

14 ноября 2022 г.

Файловая система (определения)

Файловая система — способ хранения информации в долговременной памяти компьютера (жёсткие диски, флешки, ...) и соотвеющее API операционной системы.

API — application programming interface — интерфейс прикладного программирования. Так называют набор функций, посредством которых программист может взаимодействовать с операционной системой.

Файл — поименованная область диска. **Каталог, папка** — поименованная группа файлов. **Родительский каталог** для файла или папки — это папка, в которой находится данный файл или папка. **Корневой каталог** — каталог, у которого нет родительского каталога.

Путь к файлу — способ указания конкретного файла в файловой системе.

Файловая система в POSIX (1)

Далее мы будем рассматривать файловую систему UNIX-подобных операционных систем.

UNIX-подобная ОС — операционная система, реализующая стандарт POSIX. Примеры: Linux, macOS. На Windows имитируют окружение POSIX такие проекты как Cygwin и MinGW/MSys. В Windows 10 появилась подсистема WSL (Windows Subsystem for Linux), которая также имитирует окружение POSIX.

В отличие от Windows, в UNIX-подобных системах дерево файлов единое, т.е. содержимое отдельных устройств подключается в общее дерево папок как подпапки. (На Windows для каждого устройства выделяется отдельная буква диска.)

Файловая система в POSIX (2)

Для каждого процесса существует своего рода глобальная переменная — **текущая папка.** Как правило, текущая папка — это папка, которая была текущей в родительском процессе на момент запуска дочернего. Но процесс при желании может эту папку сменить.

Путь к файлу может быть **аблолютным** или **относительным.** Абсолютный путь к файлу указывается относительно корня операционной системы, относительный — относительно текущего каталога.

Файловая система в POSIX (3)

Имя файла в UNIX-подобных операционных системах может содержать любые знаки кроме знака `/` и знака с кодом `\0`. Символ с кодом `\0` запрещён, т.к. API для UNIX-подобных систем пишется на Си, а в Си этот символ является ограничителем строк. А знак `/` служит для разделения имён каталогов в пути к файлу.

По соглашению, имя файла может содержать точку, часть имени файла после точки определяет тип файла. Например, `example.c` — исходный текст на Си, `index.html` — веб-страница. Эта часть имени файла называется «расширение» или «суффикс».

Файловая система в POSIX (4)

Абсолютный путь к файлу записывается, начиная со знака /:

`/папка/папка/…/имя-файла-или-папки`

Относительный путь к файлу не начинается со /:

`папка/папка/…/имя-файла-или-папки`

Файловая система в POSIX (5) — . и ..

В путях можно использовать такие синонимы, как . и .. . Знак . является синонимом текущей папки, знак .. — родительской папки. Можно считать, что в каждой папке находится папка ., которая является синонимом для неё же самой и папка .. , которая является синонимом для родительской папки (ссылка на родительскую папку).

В корневой папке .. ссылается на неё же саму.

Файловая система в POSIX (6) — `.` и `..`

Т.е., например, следующие пути будут эквиваленты:

```
/usr/bin/gcc
/usr/./././bin/./gcc
/var/log/../../usr/bin/gcc
/../../../../usr/bin/gcc
```

Путь `/var/log/..` ссылается на папку `/var`, т.к. `..` в `/var/log` ссылаются на родительскую для неё папку. `/var/log/../..` ссылается на корень.

Ссылки `.` и `..`, как правило, используются в относительных путях.

Командная оболочка в POSIX (1)

Оболочка операционной системы — это программа, которая позволяет пользователю взаимодействовать с операционной системой: запускать программы, работать с файлами. Оболочки бывают текстовыми и графическими, текстовые появились исторически раньше.

Командный процессор — текстовая оболочка операционной системы. Пользователь вводит команду, операционная система команду выполняет, выводит что-то на экран и ожидает следующей команды. Примерно как в REPL.

Командная оболочка в POSIX (2)

Исторически в UNIX первой оболочкой была оболочка, которая так и называлась, shell и располагалась по пути `/bin/sh`. Позже Борн создал оболочку Born Shell, затем был создан open source клон этой оболочки Born Again Shell — bash. Bash располагается по пути `/bin/bash`. Unix shell стандартизирован в POSIX.

Bash является расширением Unix shell, на большинстве дистрибутивов Linux `/bin/sh` является ссылкой на `/bin/bash`.

Работа в Bash (1)

Bash отображает приглашение командной строки, как правило, включающее имя пользователя, имя компьютера, путь к текущей папке и знак привилегий: $ для ограниченного пользователя, # для администратора (суперпользователя).

Знак ~ является сокращением для домашнего каталога пользователя, каталога вида `/home/username`.

Работа в Bash (2)

В командной строке можно вводить как встроенные команды Bash, так и имена программ. Если имя программы не включает знак /, то исполнимый файл программы ищется в стандартных путях поиска, как правило, включающих `/bin` и `/usr/bin`. Для суперпользователя — также `/sbin` и `/usr/sbin`.

Если указан путь к программе, включающий / (относительный или абсолютный), то стандартные пути поиска не учитываются, запускается программа по заданному пути. Т.е. если в текущей папке лежит программа, то её приходится запускать как

```
./progname
```

Работа в Bash (3)

Программы могут принимать аргументы командной строки. Т.е. после имени программы можно указать одно или несколько слов, эти слова запущенная программа может проанализировать и выполнить какие-либо действия:

```
./progname arg1 arg2 ...
```

Нулевым аргументом командной строки является само имя запущенной программы, последующие аргументы — те, что указаны пользователем.

Работа в Bash (4)

Программа `example.c`:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("program arguments:\n");

  for (int i = 0; i < argc; ++i) {
    printf("[%d] = %s\n", i, argv[i]);
  }

  return 0;
}
```

Работа в Bash (5)

Пример работы:

```
mazdaywik@Mazdaywik-NB10:~$ vim example.c
mazdaywik@Mazdaywik-NB10:~$ gcc example.c
mazdaywik@Mazdaywik-NB10:~$ ./a.out
program arguments:
[0] = ./a.out
mazdaywik@Mazdaywik-NB10:~$ ./a.out hello world
program arguments:
[0] = ./a.out
[1] = hello
[2] = world
mazdaywik@Mazdaywik-NB10:~$
```

Работа в Bash (6)

Программы `vim` (текстовый редактор) и `gcc` (компилятор Си) получали в качестве аргумента имя файла, программа `a.out` (результат трансляции) — произвольные строки.

Работа в Bash (7)

Список стандартных команд оболочки (встроенные команды и стандартные утилиты из `/bin`):

- `man команда` — показывает интерактивную справку по данной команде.
- `pwd` — вывести текущую папку.
- `cd имя-папки` — сменить текущую папку.
- `mkdir имя-папки` — создать папку.
- `rm файл ...`, `rmdir папка ...` — удаляет файлы и папки.
- `cp старый-файл новый-файл` — копирует файл.
- `cp файл ... папка` — копирует несколько файлов в папку.
- `mv старый-файл новый-файл`, `mv файл ... папка` — аналогично перемещает или переименовывает файлы и папки.
- `ls [папка]` — распечатывает содержимое папки на экран. По умолчанию — текущей папки.

Работа в Bash (8)

Список стандартных команд оболочки (встроенные команды и стандартные утилиты из `/bin`):

- `cat [файл ... ]` — распечатывает содержимое файлов на экран. Если имена файлов отсутствуют, то дублируется на экран ввод пользователя.
- `clear` — очищает экран.
- `more [файл]` — вывод содержимого файла постранично, утилита POSIX.
- `less [файл]` — улучшенный вариант `more`, в POSIX не входит, но обычно есть.
- `tree [папка]` — вывод дерева папок указанной папки.
- `wc [файл ... ]` — подсчёт символов, слов и строк в указанных файлах.
- `echo строка` — вывод строки на экран.

Работа в Bash (9)

Для аргументов командной строки существует соглашение, что параметры делятся на **ключи** и имена файлов. Ключи (опции) всегда начинаются на один или два знака `-`. Если аргумент не начинается с дефиса — он считается именем файла.

Ключи управляют режимом работы программы. Ключи, начинающиеся на `-`, как правило, однобуквенные, ключи на `--` записываются целым словом.

Работа в Bash (10)

Например, команда

```
mkdir -p foo/bar/baz
```

создаст папки `foo`, `foo/bar` и `foo/bar/baz`, если их до этого не существовало. Без ключа `-p` программа выдаст ошибку, т.к. для папки `baz` родительской папки `foo/bar` не существует.

У большинства команд (программ) есть ключ `-h` или `--help`, который отображает краткую справку. Не для все команд есть справка, выдаваемая через `man`.

Работа в Bash (11)

Bash умеет раскрывать шаблоны имён файлов. Если среди аргументов присутствует аргумент со знаками `*` или `?`, то он считается шаблоном и вместо него помещаются файлы, чьи имена соответствуют шаблону.

В шаблоне знак `*` означает произвольную последовательность знаков, `?` — один знак.

Примеры: `*.c` — все файлы текущей папки с расширением `.c`, `backups/2020-12-*.zip` — архивы, датированные декабрём этого года из папки `backups`. Если в папке присутствуют файлы с расширениями `.cpp` и `.cxx`, то шаблон `*.c??` выберет их все.

Работа в Bash (12)

Пример раскрытия шаблона

```
mazdaywik@Mazdaywik-NB10:~$ ./a.out *.c*
program arguments:
[0] = ./a.out
[1] = example.c
[2] = hello.cpp
```

В текущей папке было только 2 файла, подпадающие под шаблон.

Работа в Bash (13)

Для того, чтобы записать аргумент, например, с пробелами или какими-то другими знаками, которые интерпретируются в Bash, используются кавычки.

Двойные кавычки " ... " допускают некоторую интерпретацию внутри них, например, раскрытие переменных или шаблонов. Одинарные ' ... ' — трактуют содержимое буквально.

Работа в Bash (14)

```
mazdaywik@Mazdaywik-NB10:~$ X=Foo
mazdaywik@Mazdaywik-NB10:~$ echo $X
Foo
mazdaywik@Mazdaywik-NB10:~$ ./a.out "Hello, $X"
program arguments:
[0] = ./a.out
[1] = Hello, Foo
mazdaywik@Mazdaywik-NB10:~$ ./a.out 'Hello, $X'
program arguments:
[0] = ./a.out
[1] = Hello, $X
```

Работа в Bash (15)

```
mazdaywik@Mazdaywik-NB10:~$ ./a.out Hello, $X
program arguments:
[0] = ./a.out
[1] = Hello,
[2] = Foo
mazdaywik@Mazdaywik-NB10:~$ ./a.out '*.c*'
program arguments:
[0] = ./a.out
[1] = *.c*
```

Процессы в POSIX (1)

«Философия Unix гласит:

- Пишите программы, которые делают что-то одно и делают это хорошо.
- Пишите программы, которые бы работали вместе.
- Пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс».

Дуг Макилрой, изобретатель каналов Unix и один из основателей традиции Unix

Процессы в POSIX (2)

Процесс — это экземпляр работающей программы.

Когда мы в Bash пишем команду, запускающую программу, запускается новый процесс, а сама оболочка ждёт его завершения. Но процесс можно запустить и в фоне:

```
$ ./program args &
```

Знак `&` означает, что процесс запущен в фоне. Список фоновых программ, запущенных в текущем сеансе, можно получить при помощи команды `jobs`, она выведет пронумерованные процессы. Команда `fg` переводит фоновый процесс на передний план.

Процессы в POSIX (3)

Процесс может быть приостановлен (заморожен, поставлен на паузу). Постановка текущей выполняемой программы на паузу выполняется комбинацией клавиш CTRL-Z. Процесс в этом случае приостанавливается и уходит в фон.

Команда `fg` к приостановленному процессу его возобновляет и переводит на передний план. Команда `bg` — возобновляет и отправляет в фон.

Процессы в POSIX (3)

Пример. Запустили архиватор, увидели, что он будет работать долго, решили послать его в фон:

```
mazdaywik@Mazdaywik-NB10:~$ tar czf archive.tar.gz *
^Z
[1]+  Остановлен    tar czf archive.tar.gz *
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]+  Остановлен    tar czf archive.tar.gz *
mazdaywik@Mazdaywik-NB10:~$ bg
[1]+ tar czf archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]+  Запущен          tar czf archive.tar.gz * &
```

Процессы в POSIX (4)

```
mazdaywik@Mazdaywik-NB10:~$ fg
tar czf archive.tar.gz *
^Z
[1]+  Остановлен    tar czf archive.tar.gz *
mazdaywik@Mazdaywik-NB10:~$ tar czf second-archive.tar.gz * &
[2] 25
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]+  Остановлен       tar czf archive.tar.gz *
[2]-  Запущен          tar czf second-archive.tar.gz * &
```

Процессы в POSIX (5)

```
mazdaywik@Mazdaywik-NB10:~$ bg 1
[1]+ tar czf archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$ jobs
[1]-  Запущен          tar czf archive.tar.gz * &
[2]+  Запущен          tar czf second-archive.tar.gz * &
mazdaywik@Mazdaywik-NB10:~$
```

Процессы в POSIX (6)

Для прерывания процесса используется комбинация клавиш CTRL-C:

```
mazdaywik@Mazdaywik-NB10:~$ fg 1
tar czf archive.tar.gz *
^C
mazdaywik@Mazdaywik-NB10:~$ fg 2
tar czf second-archive.tar.gz *
^C
mazdaywik@Mazdaywik-NB10:~$ jobs
mazdaywik@Mazdaywik-NB10:~$
```

Процессы в POSIX (7)

Процессы в unix-подобных системах идентифицируются по PID — целое число.

Для получения списка запущенных процессов используется команда `ps`, по умолчанию выводит список процессов текущего пользователя. Команда `ps aux` выводит все процессы в системе с выдачей подробных сведений.

Процессы в POSIX (8)

Процессам можно посылать сигналы. Для посылки сигналов используется команда `kill`. Синтаксис

```
kill [-N] pid
```

где `-N` — номер сигнала. По умолчанию посылается сигнал `SIGTERM`. Сигнал `SIGTERM` — просьба процессу завершиться. Аналогичную роль играет `SIGINT`, он как раз посылается с клавиатуры комбинацией клавиш CTRL-C. Сигнал `SIGSTOP` посылается как CTRL-Z.

Список доступных сигналов с номерами:

```
kill -l
```

Процессы в POSIX (9)

Сигнал `SIGKILL` — сигнал на безусловное прерывание программы, имеет код `9`. Поэтому, чтобы жёстко убить процесс, нужно набрать

```
kill -9 pid
```

Если в программе произошла ошибка доступа к памяти (например, из-за неверного указателя), операционная система посылает процессу сигнал `SIGSEGV` (segmentation violation, segmentation fault, ошибка сегментации).

Потоки ввода-вывода в POSIX (1)

Процесс может иметь несколько открытых дескрипторов (небольшие целые числа), из которых он может читать данные, либо в них писать. Обычно это дескрипторы открытых файлов или сетевых соединений.

Но есть 3 по умолчанию открытых дескриптора, которые соответствуют двум устройствам:

- Дескриптор `0` — чтение с клавиатуры.
- Дескриптор `1` — вывод на экран.
- Дескриптор `2` — вывод на экран.

Для того, чтобы ввести «конец файла» с клавиатуры, используется комбинация клавиш CTRL-D. На Windows «конец файла» вводится как CTRL-Z.

Потоки ввода-вывода в POSIX (2)

В языке Си тип `FILE*` — обёртка над дескрипторами ОС, обёртки над этими тремя дескрипторами доступны как константы `stdin`, `stdout` и `stderr`.

```
fprintf(stdout, "Hello!\n");
```

эквивалентно

```
printf("Hello!\n");
```

Потоки ввода-вывода в POSIX (3)

Оболочка `bash` может перенаправлять дескрипторы. Для запущенной программы можно связать `stdin`, `stdout` и `stderr` с файлом или каналом.

Канал (`pipe`) — особый тип файла. Если один процесс откроет канал для чтения, а второй — для записи, то всё, что запишет второй, будет читать первый. Когда пишущий процесс канал закроет, читающий увидит «конец файла».

Потоки ввода-вывода в POSIX (4)

Перенаправление стандартного ввода

```
$ ./program args ...  < input.txt
```

Если исходно программа запрашивала у пользователя ввод с клавиатуры, то теперь она читает файл `input.txt`.

Потоки ввода-вывода в POSIX (5)

Перенаправление стандартного вывода:

```
$ ./program args ...  > output.txt
```

На экран ничего не выводится, а то, что программа печатает на экран, на самом деле пишется в файл `output.txt`. Если до запуска программы файл `output.txt` существовал, то он перезапишется. Если использовать знак >>, то запись будет осуществляться в конец файла.

Пример:

```
$ echo hello > hello.txt
$ echo world >> hello.txt
```

Потоки ввода-вывода в POSIX (6)

Перенаправление стандартного потока ошибок:

```
$ ./program args ... 2> errors.txt
```

Программа может выводить на `stdout` полезные данные, а на `stderr` ошибки. Тогда, если `stdout` перенаправлен и возникнет что-то, о чём нужно уведомить пользователя, (а) сообщение об ошибке пользователь увидит (`stderr` по-прежнему связан с экраном), (б) сообщения об ошибках не перепутаются с полезными данными.

Потоки ввода-вывода в POSIX (7)

Пример. Программа `cat`, предназначенная для конкатенации файлов, получает в командной строке имена файлов и их содержимое последовательно пишет на `stdout`. Перенаправив `stdout`, мы получим файл с конкатенацией содержимого исходных файлов:

```
$ cat header.txt body.txt footer.txt > document.txt
```

Потоки ввода-вывода в POSIX (8)

Несколько программ можно объединять в конвейер:

```
$ prog1 args ... | prog2 args ... | prog3 args
```

В этом случае `stdout` каждой из программ будет связан со стандартным вводом (`stdin`) следующей программы.

Потоки ввода-вывода в POSIX (9)

Задача: найти в файлах с расширением `.c` все строки, содержащие `#include` и вывести их в алфавитном порядке и без повторяющихся строк:

```
$ cat *.c | grep "#include" | sort | uniq
```

Многие утилиты в unix-подобных ОС или принимают список файлов в качестве аргументов, или, если файлов не указано, читают стандартный ввод.

Написание скриптов (1)

Исполнимые файлы в UNIX-подобных ОС отличаются от обычных флагом исполнимости. У каждого файла есть три набора флагов `rwxrwxrwx`, `r` — доступ на чтение, `w` — доступ на запись, `x` — доступ на исполнение. Первая группа — права владельца файла, вторая — права группы пользователей, владеющих файлом, третья — права для всех остальных.

Права доступа типичного файла: `rw-r--r--`, т.е. владелец может в файл писать, все остальные — только читать.

Права доступа: `--x--x--x` — файл нельзя прочитать, но можно запустить.

Написание скриптов (2)

Установка и сброс атрибутов выполняется командой `chmod`:

```
chmod +x prog          # добавить флаг исполнимости
chmod +w file.dat      # разрешить запись
chmod -w file.dat      # запретить запись
chmod go-r file.dat    # запретить чтение (r) группе (g)
                       # и всем остальным (o)
```

Написание скриптов (3)

Исполнимые файлы могут быть либо двоичными в формате исполнимых файлов ОС (ELF для Linux, формат Mach-O для macOS), либо **скриптами (сценариями).** Скрипты должны начинаться со строки с указанием интерпретатора (так называемый *shebang*).

```
#!/путь/до/интерпретатора
```

Для Bash это

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

или

```
#!/bin/sh
```

Если shebang не указан, то на Linux по умолчанию вызывается `/bin/sh`.

Написание скриптов (4)

В сценарии последовательно записываются команды Bash. Среди них могут быть как вызовы программ, так и встроенные команды включая операторы.

Процессы при завершении устанавливают код возврата. В языке Си кодом возврата является возвращаемым значением функции `main()`:

```c
int main() {
  return 100;
}
```

По соглашению, успешное завершение работы соответствует коду `0`, неуспешное — ненулевому числу, при этом разные значения соответствуют разным ошибкам.

Написание скриптов (5)

Несколько команд можно объединять знаками `(`, `)`, `&&`, `||`.

```
prog1 && prog2
```

Код возврата будет нулевым, если обе программы завершились успешно. Если `prog1` завершилась неуспешно, `prog2` даже не запустится.

```
./gen-source > source.c && gcc source.c
```

```
prog1 || prog2
```

Соответственно, логическое ИЛИ. `prog2` вызовется, если `prog1` завершилась неуспешно.

```
./get-info > info.txt || rm info.txt
```

Команды в Bash разделяются переводом строки или знаком `;`.

Написание скриптов (6) — оператор `if`

Оператор ветвления имеет вид:

```
if команда аргументы ...  ; then
  команда
   ...
elif команда аргументы ...  ; then
  команда
   ...
else
  команда
   ...
fi
```

Написание скриптов (7)

Код после `then` выполняется, если команда в условии завершилась успешно.

`grep` возвращает успех, если что-то нашлось, иначе — неуспех.

```
if grep ERROR file.txt > /dev/null; then
  echo Были ошибки
else
  echo Ошибок не было
fi
```

Встроенные команды `true` и `false` они всегда завершаются, соответственно, успешно и неуспешно.

Написание скриптов (8) — цикл `while`

Цикл `while`

```
while команда аргументы ... ; do
  команда
   ...
done
```

Написание скриптов (9) — цикл `for`

Цикл `for`:

```
for перем in строка ... ; do
  команда $перем
   ...
done
```

Переменные окружения (1)

Переменные окружения. В UNIX есть понятие *переменных окружения* или *переменных среды* (environment variables) — набора некоторых глобальных переменных, которые хранят некоторые строки.

Например, переменная `PATH` хранит список стандартных путей, в которых ищутся исполнимые файлы. Типичное содержимое: `/bin:/usr/bin:/usr/local/bin` (пути разделяются двоеточием). `HOME` — путь к домашнему каталогу пользователя, `USER` — имя пользователя.

Переменные окружения (2)

В Bash можно устанавливать переменные среды при помощи синтаксиса

```
VAR=VALUE
```

Получить значение переменной можно при помощи `$VARNAME` или `${VARNAME}`.

```
MY_NAME="Vasiliy Pupkin"
echo $MY_NAME
```

Переменные окружения (3)

Можно писать так:

```
RESULT=false

if …; then
  …
  RESULT=true
  …
fi

if $RESULT; then
  …
fi
```

Переменные окружения (4)

Особые переменные среды:

- `$!` — PID процесса, запущенного в фоне предыдущей командой.
- `$?` — код возврата предыдущей команды.
- `$1`, `$2`, … — параметры командной строки скрипта.
- `$*` и `$@` — список всех параметров. Посмотрите в руководстве, чем они отличаются. Желательно их указывать в кавычках `"$1"`, тогда при раскрытии параметры с пробелами не рассыпятся на кусочки.
- `$0` — имя скрипта.

Команда `shift` (встроенная в Bash) сдвигает аргументы командной строки: `$2` становится `$1`, `$3` → `$2` и т.д., значение `$1` теряется.

Программы-фильтры (1)

Программы-фильтры — это программы, которые принимают какой-то текст на `stdin`, фильтруют его как-то и выводят на `stdout`. Либо, если указаны файлы в командной строке, они читают каждый файл последовательно.

Программы-фильтры как правило используются в конвейерах.

- `sort` — сортирует строки в алфавитном порядке (в соответствии с кодами символов). У команды есть множество дополнительных ключей, выполняющих числовую сортировку, сортировку в обратном порядке, сортировку по номеру поля и т.д. Ключи можно посмотреть в `man sort` или `sort --help`.
- `uniq` — удаляет последовательные повторяющиеся строки. Комбинация `sort | uniq` позволяет получить поток, в котором нет вообще повторяющихся строк.

Программы-фильтры (2)

- `grep`, `egrep` — выбирает из потока строки, содержащие некоторый шаблон.
- `head [-N]`, `tail [-N]` — выбирают первые `N` строк или последние `N` строк файла или потока. По умолчанию `N` равно 10.
- `sed команда` — позволяет выполнять некоторые операции по редактированию потока или файла. Наиболее распространённое использование — делать замену одной подстроки на другую `sed 's/from/to/'` (заменяется первое вхождене), `sed 's/from/to/g'` — все вхождения. Пример: `man cat | sed 's/cat/dog/g'`

Программы-фильтры (3)

- `more` — выводит текст постранично, можно перематывать только вперёд (POSIX).
- `less` — выводит текст постранично, его можно перематывать вверх и вниз стрелками (утилита проекта GNU). Утилиты `more` и `less` используются в конце конвейера.
- `cat` — ничего не делает с потоком, но может в поток положить содержимое нескольких файлов.
- `tac` — выводит поток задом наперёд.
- `awk` — скриптовый язык программирования, ориентированный на фильтрацию потока. Описание языка: `man awk`.

Команда `test` (она же `[`) (1)

Команда `test` позволяет проверить некоторое условие, относящееся к файлам или значениям. Если условие верное, код возврата нулевой, иначе — ненулевой.

Может быть вызвана как `test условие` или как `[ условие ]`. Примеры:

```
[ -e filename ]     # проверяет, существует ли файл
[ 100 -lt 200 ]     # проверяет, что 100 меньше 200
[ "ab" ≠ "cd" ]     # проверяет, что строка "ab" не равно "cd"
[ 100 -ne 200 ]     # числа не равны
```

Команда `test` (она же `[`) (1)

```
[ -e filename.txt ] && [ 100 -ge 100 ]
[ -e filename.txt -a 100 -ge 100 ]
test -e filename.txt -a 100 -ge 100
```

Ключи команды `test` см. в `man test` (для самостоятельного изучения).

Но есть и особый синтаксис. В Bash есть встроенная команда `[[`, которая по поведению эквивалентна `test`, но обрабатывается самим Bash.

Функции в скриптах (1)

В Bash можно объявлять функции. Синтаксис:

```
funcname() {
  тело функции
}
```

Функция вызывается как обычная команда, параметры функции доступны в её теле как $1, $2, ….

Функции в скриптах (2)

Если команду записать внутри обратных кавычек или внутри скобок `$(…)`, то весь вывод программы на `stdout` превратится в поледовательность аргументов.

```
echo `cat file.txt`
echo $(cat file.txt)
```

Этот синтаксис часто используют при написании функций. Функция возвращает результат на `stdout`, её вызывают обратными кавычками или `$(…)` и получают её вывод как строку.

Функции в скриптах (3)

Bash может вычислять арифметические выражения: `$((2 + 2 * 2))` → 6.

Встроенная команда `read VARNAME` считывает из стандартного ввода одну строчку и кладёт её в переменную `VARNAME`. Если достигнут конец файла, программа завершается неуспешно. Использование:

```
... | while read X; do
  ...
  ...
done
```

Функции в скриптах (4)

Пример. Рекурсивный обход папок:

```
#!/bin/bash

rec() {
  if [ -d "$1" ]; then
    ls "$1" | while read name; do
      rec "$1/$name"
    done
  else
    echo File "$1"
  fi
}

rec "$1"
```

Shebang и /usr/bin/env

Для того, чтобы запустить интерпретатор скриптового языка, доступный в PATH, но при этом располагающемся по неизвестному пути, в начало файла добавляют /usr/bin/env:

```
#!/usr/bin/env python

# дальше код какой-то на Python
...

#!/usr/bin/env node

// Дальше код на JavaScript
...
```