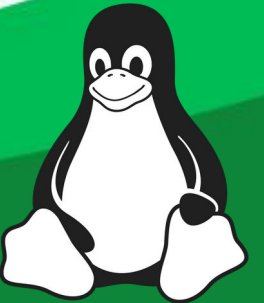


# Оболочка Linux



# Запуск программ

Синтаксис запуска программы выглядит таким образом:

/путь/к/файлу/программы параметры

Переменная PATH, в которой хранятся все пути к папкам где обычно находятся программы

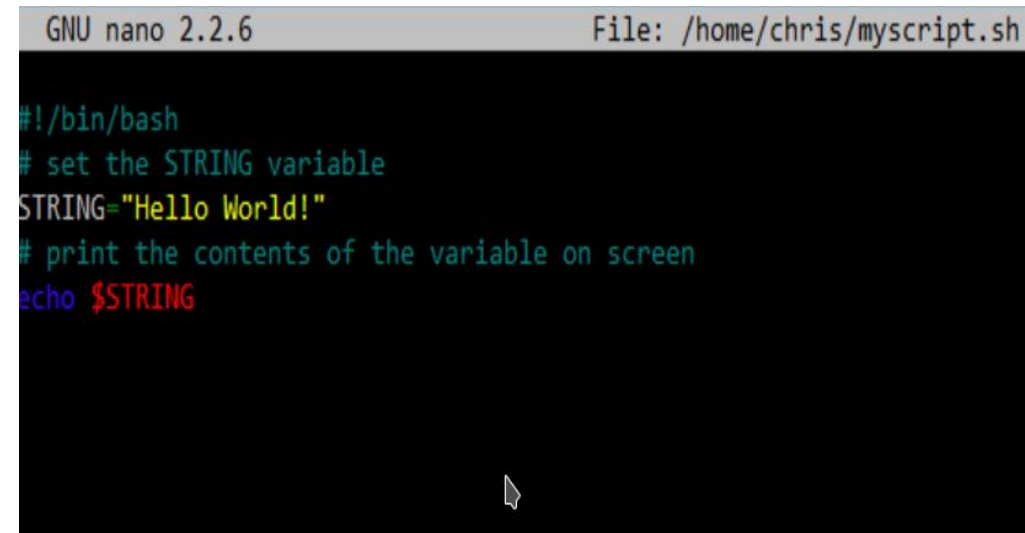
echo \$PATH

Программы делятся на бинарные и интерпретируемые

Создадим файл first.sh, напишем в нем и сохраним:

echo "Current directory is:"

pwd

A screenshot of a terminal window showing the GNU nano 2.2.6 text editor. The editor is open to a file named /home/chris/myscript.sh. The script contains four lines of code: a shebang line, a comment, a variable assignment, and an echo command. The text is color-coded: blue for the shebang, green for comments, yellow for the variable name, and red for the variable reference.

```
GNU nano 2.2.6 File: /home/chris/myscript.sh
#!/bin/bash
# set the STRING variable
STRING="Hello World!"
# print the contents of the variable on screen
echo $STRING
```

---

**Как вы думаете как можно обобщенно назвать цель создания bash-скриптов?**

# Код возврата

Процесс, завершившись, возвращает родительскому процессу какое-то значение — код возврата.

По соглашению разработчиков, нулевой код возврата означает успешное

завершение, а ненулевые — разнообразные ошибки. Переменной `$?` с каким кодом завершился последний процесс.

`exit <code>` - команда `bash` возвращающая код из скрипта

1	разнообразные ошибки
2	неверное использование встроенных команд
126	вызываемая команда не может быть выполнена
127	команда не найдена
128	неверный аргумент команды <code>exit</code>
128+n	фатальная ошибка по сигналу "n"
130	завершение по Control-C

# Объединение команд

---

`cmd1 ; cmd2` Последовательность команд ;

`cmd1 && cmd2` выполнить команду `cmd1` и затем `cmd2`. Если `cmd1` завершилась с ошибкой, то `cmd2` не выполнится

`cmd1 || cmd` выполнить или `cmd1` или `cmd2`. `cmd2` не будет выполнена, если `cmd1` выполнилась успешно

Примеры:

`cp student*.jpg /media/StudentFlash && rm student*.jpg`

`cd music/ || mkdir music`

# Конвейеры команд

---

Символ «|» — это и есть конвейер, аналогия - канал, в который один процесс может только писать, а другой — только читать из него. Выборка и помещение информации в такой канал происходит в порядке FIFO (First In/First Out)

Общий синтаксис конвейера команд такой: `cmd1 | cmd2 | .... | cmdN`

Перенаправление и `stdout`, и `stderr` в `pipe`:

Написание вместо | пары символов '|&', эквивалентно перенаправлению как `stdout`, так и `stderr` в конвейер

---

**Напишите конвейер, который выведет имя вашего дистрибутива на консоль  
Вам могут пригодиться `uname` и `cut`**

# Команда tr

---

tr, используется для замены, замещения или удаления символов из стандартного ввода, отправляя результат на стандартный вывод

tr [КЛЮЧ]... НАБОР1 [НАБОР2]

-c, -C Сначала получить дополнение НАБОРА1

-d --delete Удалить знаки из НАБОРА2

-s --squeeze-repeats Удалить повторы

-t --truncate-set1

Примеры: Заменить все x на z tr x z

Удалить все буквы в нижнем регистре tr -d [:lower:]

Уплотнить повторяющиеся буквы большого и малого регистров tr -s [:upper:][:lower:]



# Команда wc

---

Выводит количество строк, слов и байт введённой информации

Утилита может обрабатывать файлы: `wc file`

Или стандартный поток ввода если ввести `wc`

`-c` или `—bytes` Отобразить размер объекта в байтах

`-m` или `--count` Показать количество символов в объекте

`-l` или `--lines` Вывести количество строк в объекте

`-w` или `--words` Отобразить количество слов в объекте

# Команда sort

---

Сортировка строк текстовых файлов. Основные параметры представлены ниже

- `b --ignore-leading-blanks` игнорировать начальные пробелы в строках
- `-f --ignore-case` Выполнять сортировку без учета регистра символов
- `-n --numeric-sort` Выполнять сортировку, опираясь на числовые значения строк.
- `-r --reverse` Сортировать в обратном порядке
- `-k --key=поле1[,поле2]` Сортировать по ключевым полям
- `-m --merge` Интерпретировать каждый аргумент как имя предварительно отсортированного файла.
- `-o --output=файл` Записать результат сортировки не в стандартный вывод, а в указанный файл
- `-t --field-separator=символ` Определяет символ, разделитель полей.

Пожалуй, параметр `-n` требует пояснения, он используется для сортировки по числовым значениям. Этот параметр позволяет сортировать строки по их числовым значениям, а не по алфавитному порядку

# Команда uniq

---

Выявление или удаление повторяющихся строк

Чтобы `uniq` действительно выполнила свою работу, исходные данные нужно сначала отсортировать. Это объясняется тем, что `uniq` удаляет повторяющиеся записи, только если они следуют друг за другом.

Параметры:

- c Вывести список повторяющихся строк, предваряя их числом найденных дубликатов
- d Вывести только повторяющиеся, не уникальные строки
- f n Пропустить n начальных полей в каждой строке.
- i Сравнивать строки без учета регистра символов
- s n Пропустить n начальных символов в каждой строке
- u Вывести только уникальные строки (по умолчанию)

# Команда grep

grep просматривает текстовые файлы в поисках совпадений с указанным регулярным выражением и выводит в стандартный вывод все строки с такими совпадениями

grep [параметры] регулярное\_выражение [файл...]

Примеры: `ls /usr/bin | grep zip` список всех файлов из каталога `/usr/bin`, содержат подстроку `zip`

-i Игнорировать регистр символов

-v Инвертировать критерий

-c Вывести число совпадений (или «несовпадений» если -v)

-l Вместо строк с совпадениями выводить только имена файлов с найденными строками

-L выводит только имена файлов, где не найдено ни одного совпадения.

-n В начале каждой строки с совпадением вывести ее номер в файле

-h Подавить вывод имен файлов при поиске по нескольким файлам

# Перенаправление стандартного вывода

---

Механизм перенаправления ввода/вывода позволяет явно указать, куда должен осуществляться стандартный вывод

Чтобы перенаправить стандартный вывод в другой файл вместо экрана, нужно добавить в команду оператор перенаправления > и имя файла

```
ls -l /usr/bin > ls_output.txt
```

Добавить вывод в конец существующего файла, не затерев его, используем оператор перенаправления »

```
ls -l /usr/bin >> ls-output.txt
```

# Перенаправление стандартного вывода ошибок

---

Чтобы перенаправить стандартный вывод ошибок, нужно указать его дескриптор файла

Программа может производить вывод в любой из нескольких нумерованных файловых потоков — файловые дескрипторы стандартный ввод, вывод и вывод ошибок (0, 1 и 2 соответственно)

```
ls -l /bin/usr 2> ls-error.txt
```

# Перенаправление стандартного вывода и стандартного вывода ошибок в один файл

Иногда необходимо сохранить весь вывод команды в один файл. Для этого перенаправьте сразу два потока, стандартный вывод и стандартный вывод ошибок

```
ls -l /bin/usr > ls-output.txt 2>&1
```

Современные версии bash поддерживают второй, более простой метод выполнения перенаправления этого вида:

```
ls -l /bin/usr &> ls-output.txt
```

# dev/null

---

Иногда вывод команды нужно отбросить

Система дает такую возможность, предоставляя специальный файл /dev/null

Чтобы подавить вывод сообщений об ошибках, достаточно проделать следующее

```
ls -l /bin/usr 2> /dev/null
```



# Перенаправление стандартного ввода

---

Используя оператор перенаправления `<`, можно изменить источник данных для стандартного ввода с клавиатуры на файл. Этот способ не имеет никаких преимуществ в сравнении с передачей простого аргумента для некоторых программ, но он демонстрирует, как можно использовать файлы в роли источника данных для стандартного ввода.

Пример

```
cat < eat_more.txt
```

# Конвейеры

---

«Умение» команд читать данные со стандартного ввода и выводить результаты в стандартный вывод используется механизмом командной оболочки, который называется конвейером

команда1 | команда2

Конвейеры часто используются для выполнения сложных операций с данными.

Они позволяют объединить вместе несколько команд

ls /bin /usr/bin | sort | less

---

**Напишите конвейер, который выведет имя модели  
вашего процессора**

# Heredoc синтаксис

---

При написании сценариев оболочки вы можете оказаться в ситуации, когда вам нужно передать многострочный блок текста или кода интерактивной команде

```
[COMMAND] <<[-] 'DELIMITER'
```

```
HERE-DOCUMENT
```

```
DELIMITER
```

Первая строка начинается с необязательной команды, за которой следует специальный оператор перенаправления << и идентификатор-разделитель, чаще всего используются EOF или END для разделителей.

Если идентификатор разделителя не заключен в кавычки, оболочка подставит все переменные, команды и специальные символы перед передачей строк в команду

# Herestring синтаксис

---

<<< Herestring

здесь Herestring - это обычная строка, используемая для перенаправления ввода, а не особый вид строки

Пример:

```
cat <<< "ABCD Hello 321!"
```

# Команда tee

---

tee - чтение со стандартного ввода и запись в стандартный вывод и в файлы

Linux предоставляет команду tee, которая создает Т-образное разветвление в конвейере

```
ls /usr/bin | tee ls.txt | grep zip
```

# Подстановка переменной

---

присваивание значений переменным производится так

переменная=значение

```
b="a string"
```

```
c="a string and $b"
```

При использовании подстановки можно заключать в необязательные фигурные скобки {}.

Переименовать файл myfile в myfile1

```
[me@linuxbox ~]$ filename="myfile"
```

```
[me@linuxbox ~]$ touch $filename
```

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

# Командная подстановка

---

Подстановка команд позволяет использовать поток вывода команд в качестве аргументов других команд

```
echo $(ls)
```

```
ls -l $(which cp)
```

Можно использовать целые конвейеры

```
file $(ls /usr/bin/* | grep zip)
```

Альтернативный синтаксис:

```
ls -l `which cp`
```



# Подстановка процесса

Синтаксис для замены процесса:

<(commands) или >(commands)

wc -l <(ls \*sh) (эквивалентно ls \*sh | wc -l)

tar -cf >(ssh remote\_server tar xf -)

Преимущество

Пример сравнение списка файлов (лишние промежуточные\_файлы)

ls \*.java | cut -d. -f1 > java.txt

ls \*.out | cut -d. -f1 > class.txt

diff c.txt out.txt && rm c.txt out.txt

-----

diff <(ls \*.c | cut -d. -f1) <(ls \*.out | cut -d. -f1)

# Арифметическое расширение

Командная оболочка поддерживает также подстановку результатов арифметических выражений

```
echo $((2 + 2))
```

Для подстановки арифметических выражений используется следующий формат:

```
$((выражение))
```

**Только целые числа**

+ Сложение

— Вычитание

\* Умножение

/ Деление

% остаток от  
деления

\*\* Возведение в  
степень

# Раскрытие скобок (brace expansion)

С помощью этого механизма из одного шаблона, содержащего фигурные скобки, создается множество текстовых строк

```
echo Впереди-{A,B,C}-позади
```

```
echo Число_{1..5} echo {Z..A}
```

```
echo a{A{1,2},B{3,4}}b
```

Пример использования

```
mkdir Pics && cd Pics
```

```
mkdir {2009..2011}-0{1..9} {2009..2011}-{10..12}
```

```
ls
```

# Порядок выполнения

---

До начала выполнения команды `bash` осуществляет «грамматический разбор» порядок:

- раскрытие скобок (brace expansion);
- замена знака тильды (tilde expansion);
- подстановка параметров и переменных;
- подстановка команд;
- арифметические подстановки (выполняемые слева направо);
- раскрытие шаблонов имен файлов и каталогов (pathname expansion)

# Порядок выполнения

---

До начала выполнения команды `bash` осуществляет «грамматический разбор» порядок:

- раскрытие скобок (brace expansion);
- замена знака тильды (tilde expansion);
- подстановка параметров и переменных;
- подстановка команд;
- арифметические подстановки (выполняемые слева направо);
- раскрытие шаблонов имен файлов и каталогов (pathname expansion)

# Сценарии оболочки

---

Сценарий командной оболочки — это файл, содержащий последовательность команд. Командная оболочка — это одновременно и мощный интерфейс командной строки к системе, и интерпретатор языка сценариев

Чтобы успешно создать и запустить сценарий командной оболочки нужно:

1. Написать сценарий
2. Сделать сценарий выполняемым
3. Поместить сценарий в каталог

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

```
# Это наш сценарий.
```

```
echo 'Hello World!'
```

# Shebang

---

Последовательность символов `#!` — это на самом деле специальная конструкция, которая называется `shebang` и сообщает системе имя интерпретатора, который должен использоваться для выполнения следующего за ним текста сценария

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

```
#!/usr/bin/perl
```

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

# Возвращаемые значения, \$?

---

Команды (включая сценарии и функции, написанные нашими собственными руками) по завершении работы возвращают системе значение, которое называют кодом завершения. Целое число в диапазоне от 0 до 255

Echo \$?

Команда true (признаком успеха)

команда false (признаком ошибки)

```
if true; then echo "It's true."; fi
```

```
if false; then echo "It's true."; fi
```



# Операторы if

---

Если  $x = 5$ , тогда:

Сказать « $x$  равно 5».

Иначе:

Сказать « $x$  не равно 5».

$x=5$

if [  $x = 5$  ]; then

echo "x equals 5."

else

echo "x does not equal 5."

fi

# Операторы if

---

if команды; then

    команды

[elif команды; then

    commands...]

[else

    команды]

fi

Посмотрим, как командная оболочка определяет, успешно или нет выполнена команда. Мы рассмотрим это когда дойдем до команды test

# Оператор while

---

Представьте, что нам нужно вывести пять чисел по порядку, от 1 до 5

```
count=1
```

```
while [ $count -le 5 ]; do
```

```
    echo $count
```

```
    count=$((count + 1))
```

```
done
```

```
echo "Finished."
```

Команда while имеет следующий синтаксис:

```
while команды; do команды; done
```

# Оператор until

---

Команда `until` очень похожа на `while`, но завершает цикл не когда обнаружит ненулевой код завершения, а наоборот. Цикл `until` продолжается, пока не получит код завершения 0

```
count=1
```

```
until [ $count -gt 5 ]; do
```

```
echo $count
```

```
count=$((count + 1))
```

```
done
```

```
echo "Finished."
```

# Оператор for

---

Оригинальный синтаксис команды for имеет следующий вид:

```
for переменная [in слова]; do
```

```
команды
```

```
Done
```

переменная — значение которой будет увеличиваться в ходе выполнения цикла, слова — необязательный список элементов, которые последовательно будут присваиваться переменной, и команды — это команды, выполняемые в каждой итерации

```
for i in A B C D; do echo $i; done
```

# Команда test

---

Часто используется с инструкциями if while until используется команда test.

Команда test может выполнять различные проверки и сравнения. Она имеет две эквивалентные формы:

test выражение

и более популярную [ выражение ]

где выражение возвращает истинное (true) или ложное (false) значение. Команда test возвращает код завершения 0, если выражение истинно, и код завершения 1, если выражение ложно

# Команда test

---

Часто используется с инструкциями if while until используется команда test.

Команда test может выполнять различные проверки и сравнения. Она имеет две эквивалентные формы:

test выражение

и более популярную [ выражение ]

где выражение возвращает истинное (true) или ложное (false) значение. Команда test возвращает код завершения 0, если выражение истинно, и код завершения 1, если выражение ложно

# Выражения для проверки файлов

---

файл1 -ef файл2 — истинно если, файлы имеют одинаковое число индексного узла

файл1 -nt файл2 — файл1 новее файла файл2

файл1 -ot файл2 — файл1 старше файла файл2

-b файл — файл существует и является блочным устройством

-c файл — файл существует и является файлом символьного устройства

-d файл — файл существует и является каталогом

-e файл — файл существует

-f файл — файл существует и является обычным файлом



# Выражения для проверки файлов

---

- g — файл файл существует и имеет атрибут set-group-ID (бит setgid)
- G — файл файл существует и принадлежит действующей группе
- k — файл файл существует и имеет атрибут «sticky bit»
- L — файл файл существует и является символической ссылкой
- O — файл файл существует и принадлежит действующему пользователю
- p — файл файл существует и является именованным каналом
- r — файл файл существует и доступен для чтения
- s — файл файл существует и имеет размер больше нуля
- S — файл файл существует и является сетевым сокетом

# Выражения для проверки файлов

---

-t дескриптор\_файла — дескриптор\_файла представляет файл, подключенный к терминалу. Это выражение можно использовать для проверки стандартных потоков ввода/вывода/ошибок

-u файл — файл существует и имеет атрибут setuid

-w файл — файл существует и доступен для записи

-x файл — файл существует и доступен для выполнения

# Выражения для проверки строк

Строка — Строка не пустая

-n строка — Длина строки больше нуля

-z строка — Длина строки равна нулю

строка1 = строка2 или строка1 == строка2 — строка1 и строка2 равны

строка1 != строка2 — строка1 и строка2 не равны

строка1 > строка2 — строка1 больше, чем строка2, в смысле алфавитной сортировки

строка1 < строка2 — меньше, чем строка2, в смысле алфавитной сортировки

# Выражения для проверки целых чисел

---

число1 -eq число2 — число1 и число2 равны

число1 -ne число2 — число1 и число2 не равны

число1 -le число2 — число1 меньше или равно числу2

число1 -lt число2 — число1 меньше, чем число2

число1 -ge число2 — число1 больше или равно числу2

число1 -gt число2 — число1 больше, чем число2

# Параметры сценариев

---

Параметры введенные с клавиатуры во время работы сценария и позиционные параметры

Встроенная команда `read` команду можно использовать для чтения ввода с клавиатуры или, в случае перенаправления, строки данных из файла

Синтаксис:

```
read [-параметры] [переменная...]
```

Если имя переменной не указано, строка с данными сохраняется в переменной `REPLY`

# Параметры сценариев. Параметры команды read

---

- a массив — Сохранить ввод в указанный массив, начиная с элемента с индексом 0.
- d разделитель — Использовать в качестве признака конца ввода первый символ из строки разделитель
- e — Использовать Readline для обработки ввода. Это позволяет редактировать ввод так же, как в командной строке
- n число — Прочитать указанное число символов, а не всю строку
- p — приглашение Показывать указанное приглашение к вводу
- r — Режим без промежуточной обработки. Не интерпретировать символы обратного следа как экранирующие символы
- s — Безмолвный режим. Не производить эхо-вывод символов на экран в процессе ввода. Этот режим может пригодиться для организации ввода паролей и другой конфиденциальной информации
- t секунды — Предельное время ожидания. Завершить ввод по истечении указанного числа секунд
- u дескриптор — Произвести ввод из файла с указанным дескриптором вместо стандартного ввода

# Параметры сценариев. Команда read

В простейшем случае read сохраняет значения полей, прочитанные со стандартного ввода, в указанные переменные. Можно также указать несколько переменных

```
read var1 var2 var3 var4 var5
```

```
echo "var1='$var1'"
```

```
echo "var2='$var2'"
```

```
echo "var3='$var3'"
```

```
echo "var4='$var4'"
```

```
echo "var5='$var5'"
```

# Позиционные параметры

Командная оболочка поддерживает множество переменных, которые называются позиционными параметрами и содержат отдельные слова из командной строки.

Эти переменные имеют имена от 0 до 9

```
echo "
```

```
\$0 = $0
```

```
\$1 = $1
```

```
\$2 = $2
```

```
\$3 = $3 "
```

Команда shift. Значение \$# - кол-во аргументов



# Функции

Функции — это «мини-сценарии», находящиеся внутри другого сценария, которые работают как автономные программы

Функции имеют две синтаксические формы. Обе формы эквивалентны и могут использоваться одна вместо другой

Первая выглядит так:

```
function имя {  
    команды  
    return  
}
```

Вторая так:

```
имя () {  
    команды  
    return  
}
```

# Функции

---

Использование функций командной оболочки:

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

```
function funct {
```

```
    echo "Step 2"
```

```
    return
```

```
}
```

```
echo "Step 1"
```

```
funct
```

```
echo "Step 3"
```

# Функции Return

---

Команда return позволяет задавать возвращаемый функцией целочисленный код завершения

```
function myfunc {  
    read -p "Enter a value: " value  
  
    echo "adding value"  
  
    return $(( $value + 10 ))  
}
```

myfunc

echo "The new value is \$?"

# Функции Return

---

Ещё один способ возврата результатов работы функции заключается в записи данных, выводимых функцией, в переменную. Рассмотрим пример:

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

```
function myfunc {
```

```
read -p "Enter a value: " value
```

```
echo $(( $value + 10 ))
```

```
}
```

```
result=$( myfunc)
```

```
echo "The value is $result"
```

# Функции. Аргументы функций

---

Функции могут использовать стандартные позиционные параметры, в которые записывается то, что передаётся им при вызове. Имя функции хранится в параметре \$0, первый переданный ей аргумент — в \$1, второй — в \$2. Количество аргументов - \$#

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

```
function myfunc {  
    echo $(( $1 + $2 ))  
}
```

# Функции. Работа с переменными

---

Существуют два вида переменных:

Глобальные переменные

Локальные переменные

Глобальные переменные — это переменные, которые видны из любого места bash-скрипта. По умолчанию все объявленные в скриптах переменные глобальны

Переменные, которые объявляют и используют внутри функции, могут быть объявлены локальными. Для того, чтобы это сделать, используется ключевое слово `local`

# Функции. Работа с переменными

Глобальные переменные — это переменные, которые видны из любого места bash-скрипта. По умолчанию все объявленные в скриптах переменные глобальны

```
function myfunc {  
  
    value=$(( $value + 10 ))  
  
}  
  
read -p "Enter a value: " value  
  
myfunc  
  
echo "The new value is: $value"
```

# Функции. Работа с переменными

---

Переменные, которые объявляют и используют внутри функции, могут быть объявлены локальными. Для того, чтобы это сделать, используется ключевое слово `local`. Если за пределами функции есть переменная с таким же именем, это на неё не повлияет. Ключевое слово `local` позволяет отделить переменные, используемые внутри функции, от остальных переменных.

```
function myfunc {  
  
    local temp=$(( $value + 5 )  
  
    echo "The Temp from inside function is $temp"  
  
}  
  
temp=4  
  
myfunc  
  
echo "The temp from outside is $temp"
```



**Спасибо за внимание**

