Bash-скрипты: начало

<https://likegeeks.com/bash-script-easy-guide/>

* [Блог компании RUVDS.com](https://habr.com/company/ruvds/),
* [Настройка Linux](https://habr.com/hub/linux/),
* [Серверное администрирование](https://habr.com/hub/s_admin/)
* Перевод

[Bash-скрипты: начало](https://habrahabr.ru/company/ruvds/blog/325522/)  
[Bash-скрипты, часть 2: циклы](https://habrahabr.ru/company/ruvds/blog/325928/)  
[Bash-скрипты, часть 3: параметры и ключи командной строки](https://habrahabr.ru/company/ruvds/blog/326328/)   
[Bash-скрипты, часть 4: ввод и вывод](https://habrahabr.ru/company/ruvds/blog/326594/)  
[Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями](https://habrahabr.ru/company/ruvds/blog/326826/)   
[Bash-скрипты, часть 6: функции и разработка библиотек](https://habrahabr.ru/company/ruvds/blog/327248/)  
[Bash-скрипты, часть 7: sed и обработка текстов](https://habrahabr.ru/company/ruvds/blog/327530/)  
[Bash-скрипты, часть 8: язык обработки данных awk](https://habrahabr.ru/company/ruvds/blog/327754/)  
[Bash-скрипты, часть 9: регулярные выражения](https://habrahabr.ru/company/ruvds/blog/327896/)  
[Bash-скрипты, часть 10: практические примеры](https://habrahabr.ru/company/ruvds/blog/328346/)  
[Bash-скрипты, часть 11: expect и автоматизация интерактивных утилит](https://habrahabr.ru/company/ruvds/blog/328436/)

Сегодня поговорим о bash-скриптах. Это — [сценарии командной строки](https://ru.wikipedia.org/wiki/%D0%A1%D1%86%D0%B5%D0%BD%D0%B0%D1%80%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B9_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8), написанные для оболочки bash. Существуют и другие оболочки, например — zsh, tcsh, ksh, но мы сосредоточимся на bash. Этот материал предназначен для всех желающих, единственное условие — умение работать в [командной строке](https://likegeeks.com/main-linux-commands-easy-guide/) Linux.  
  
[](https://habrahabr.ru/company/ruvds/blog/325522/)  
  
Сценарии командной строки — это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые некоей общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. Сценарии — это мощный способ автоматизации часто выполняемых действий.  
  
[https://habrastorage.org/files/1ba/550/d25/1ba550d25e8846ce8805de564da6aa63.png](https://ruvds.com/ru-rub/#order)  
  
Итак, если говорить о командной строке, она позволяет выполнить несколько команд за один раз, введя их через точку с запятой:

pwd ; whoami

На самом деле, если вы опробовали это в своём терминале, ваш первый bash-скрипт, в котором задействованы две команды, уже написан. Работает он так. Сначала команда pwd выводит на экран сведения о текущей рабочей директории, потом команда whoamiпоказывает данные о пользователе, под которым вы вошли в систему.  
  
Используя подобный подход, вы можете совмещать сколько угодно команд в одной строке, ограничение — лишь в максимальном количестве аргументов, которое можно передать программе. Определить это ограничение можно с помощью такой команды:

getconf ARG\_MAX

Командная строка — отличный инструмент, но команды в неё приходится вводить каждый раз, когда в них возникает необходимость. Что если записать набор команд в файл и просто вызывать этот файл для их выполнения? Собственно говоря, тот файл, о котором мы говорим, и называется сценарием командной строки.

Как устроены bash-скрипты

Создайте пустой файл с использованием команды touch. В его первой строке нужно указать, какую именно оболочку мы собираемся использовать. Нас интересует bash, поэтому первая строка файла будет такой:

#!/bin/bash

В других строках этого файла символ решётки используется для обозначения комментариев, которые оболочка не обрабатывает. Однако, первая строка — это особый случай, здесь решётка, за которой следует восклицательный знак (эту последовательность называют [шебанг](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B1%D0%B0%D0%BD%D0%B3_(Unix))) и путь к bash, указывают системе на то, что сценарий создан именно для bash.  
  
Команды оболочки отделяются знаком перевода строки, комментарии выделяют знаком решётки. Вот как это выглядит:

#!/bin/bash

*# This is a comment*

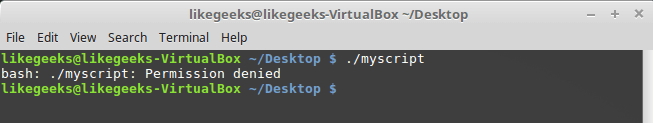
pwd

whoami

Тут, так же, как и в командной строке, можно записывать команды в одной строке, разделяя точкой с запятой. Однако, если писать команды на разных строках, файл легче читать. В любом случае оболочка их обработает.

Установка разрешений для файла сценария

Сохраните файл, дав ему имя myscript, и работа по созданию bash-скрипта почти закончена. Сейчас осталось лишь сделать этот файл исполняемым, иначе, попытавшись его запустить, вы столкнётесь с ошибкой Permission denied.



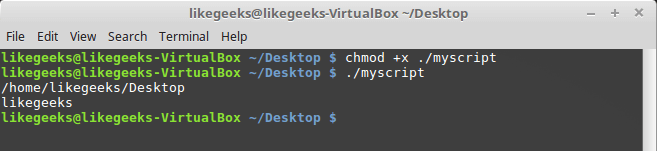
*Попытка запуска файла сценария с неправильно настроенными разрешениями*  
  
Сделаем файл исполняемым:

chmod +x ./myscript

Теперь попытаемся его выполнить:

./myscript

После настройки разрешений всё работает как надо.



*Успешный запуск bash-скрипта*

Вывод сообщений

Для вывода текста в консоль Linux применяется команда echo. Воспользуемся знанием этого факта и отредактируем наш скрипт, добавив пояснения к данным, которые выводят уже имеющиеся в нём команды:

#!/bin/bash

*# our comment is here*

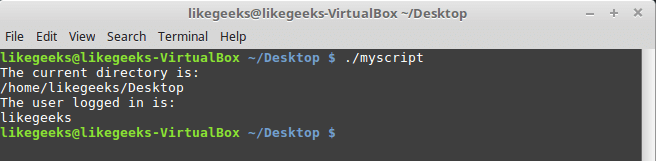
echo "The current directory is:"

pwd

echo "The user logged in is:"

whoami

Вот что получится после запуска обновлённого скрипта.



*Вывод сообщений из скрипта*  
  
Теперь мы можем выводить поясняющие надписи, используя команду echo. Если вы не знаете, как отредактировать файл, пользуясь средствами Linux, или раньше не встречались с командой echo, взгляните на [этот](https://likegeeks.com/basic-linux-commands-part2/) материал.

Использование переменных

Переменные позволяют хранить в файле сценария информацию, например — результаты работы команд для использования их другими командами.  
  
Нет ничего плохого в исполнении отдельных команд без хранения результатов их работы, но возможности такого подхода весьма ограничены.  
  
Существуют два типа переменных, которые можно использовать в bash-скриптах:

* Переменные среды
* Пользовательские переменные

Переменные среды

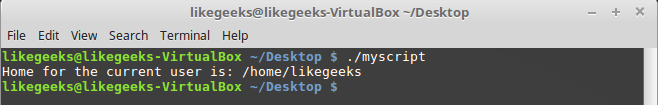
Иногда в командах оболочки нужно работать с некими системными данными. Вот, например, как вывести домашнюю директорию текущего пользователя:

#!/bin/bash

*# display user home*

echo "Home for the current user is: $HOME"

Обратите внимание на то, что мы можем использовать системную переменную $HOME в двойных кавычках, это не помешает системе её распознать. Вот что получится, если выполнить вышеприведённый сценарий.



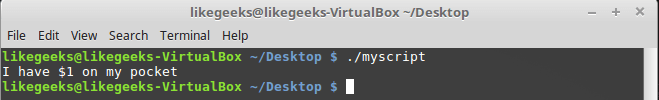
*Использование переменной среды в сценарии*  
  
А что если надо вывести на экран значок доллара? Попробуем так:

echo "I have $1 in my pocket"

Система обнаружит знак доллара в строке, ограниченной кавычками, и решит, что мы сослались на переменную. Скрипт попытается вывести на экран значение неопределённой переменной $1. Это не то, что нам нужно. Что делать?  
  
В подобной ситуации поможет использование управляющего символа, обратной косой черты, перед знаком доллара:

echo "I have \$1 in my pocket"

Теперь сценарий выведет именно то, что ожидается.



*Использование управляющей последовательности для вывода знака доллара*

Пользовательские переменные

В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария.  
  
Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:  
TNW-CUS-FMP — промо-код на 10% скидку на наши услуги, доступен для активации в течение 7 дней

#!/bin/bash

*# testing variables*

grade=5

person="Adam"

echo "$person is a good boy, he is in grade $grade"

Вот что получится после запуска такого сценария.



*Пользовательские переменные в сценарии*

Подстановка команд

Одна из самых полезных возможностей bash-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.  
  
Сделать это можно двумя способами.

* С помощью значка обратного апострофа «`»
* С помощью конструкции $()

Используя первый подход, проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку. Команду нужно заключить в два таких значка:

mydir=`pwd`

При втором подходе то же самое записывают так:

mydir=$(pwd)

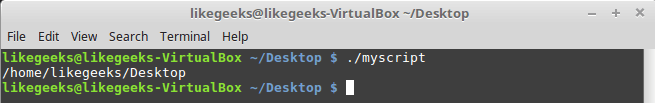
А скрипт, в итоге, может выглядеть так:

#!/bin/bash

mydir=$(pwd)

echo $mydir

В ходе его работы вывод команды pwdбудет сохранён в переменной mydir, содержимое которой, с помощью команды echo, попадёт в консоль.



*Скрипт, сохраняющий результаты работы команды в переменной*

Математические операции

Для выполнения математических операций в файле скрипта можно использовать конструкцию вида $((a+b)):

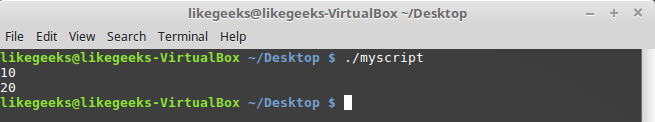
#!/bin/bash

var1=$(( 5 + 5 ))

echo $var1

var2=$(( $var1 \* 2 ))

echo $var2



*Математические операции в сценарии*

Управляющая конструкция if-then

В некоторых сценариях требуется управлять потоком исполнения команд. Например, если некое значение больше пяти, нужно выполнить одно действие, в противном случае — другое. Подобное применимо в очень многих ситуациях, и здесь нам поможет управляющая конструкция if-then. В наиболее простом виде она выглядит так:

if команда

then

команды

fi

А вот рабочий пример:

#!/bin/bash

if pwd

then

echo "It works"

fi

В данном случае, если выполнение команды pwdзавершится успешно, в консоль будет выведен текст «it works».  
  
Воспользуемся имеющимися у нас знаниями и напишем более сложный сценарий. Скажем, надо найти некоего пользователя в /etc/passwd, и если найти его удалось, сообщить о том, что он существует.

#!/bin/bash

user=likegeeks

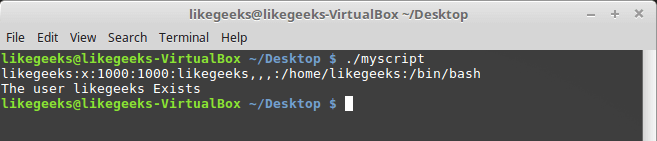
if grep $user /etc/passwd

then

echo "The user $user Exists"

fi

Вот что получается после запуска этого скрипта.



*Поиск пользователя*  
  
Здесь мы воспользовались командой grepдля поиска пользователя в файле /etc/passwd. Если команда grepвам незнакома, её описание можно найти [здесь](https://likegeeks.com/main-linux-commands-easy-guide/).  
  
В этом примере, если пользователь найден, скрипт выведет соответствующее сообщение. А если найти пользователя не удалось? В данном случае скрипт просто завершит выполнение, ничего нам не сообщив. Хотелось бы, чтобы он сказал нам и об этом, поэтому усовершенствуем код.

Управляющая конструкция if-then-else

Для того, чтобы программа смогла сообщить и о результатах успешного поиска, и о неудаче, воспользуемся конструкцией if-then-else. Вот как она устроена:

if команда

then

команды

else

команды

fi

Если первая команда возвратит ноль, что означает её успешное выполнение, условие окажется истинным и выполнение не пойдёт по ветке else. В противном случае, если будет возвращено что-то, отличающееся от нуля, что будет означать неудачу, или ложный результат, будут выполнены команды, расположенные после else.  
  
Напишем такой скрипт:

#!/bin/bash

user=anotherUser

if grep $user /etc/passwd

then

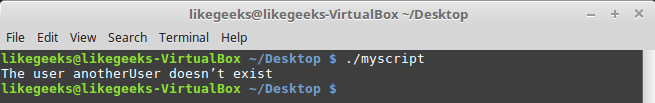
echo "The user $user Exists"

else

echo "The user $user doesn’t exist"

fi

Его исполнение пошло по ветке else.



*Запуск скрипта с конструкцией if-then-else*  
  
Ну что же, продолжаем двигаться дальше и зададимся вопросом о более сложных условиях. Что если надо проверить не одно условие, а несколько? Например, если нужный пользователь найден, надо вывести одно сообщение, если выполняется ещё какое-то условие — ещё одно сообщение, и так далее. В подобной ситуации нам помогут вложенные условия. Выглядит это так:

if команда1

then

команды

elif команда2

then

команды

fi

Если первая команда вернёт ноль, что говорит о её успешном выполнении, выполнятся команды в первом блоке then, иначе, если первое условие окажется ложным, и если вторая команда вернёт ноль, выполнится второй блок кода.

#!/bin/bash

user=anotherUser

if grep $user /etc/passwd

then

echo "The user $user Exists"

elif ls /home

then

echo "The user doesn’t exist but anyway there is a directory under /home"

fi

В подобном скрипте можно, например, создавать нового пользователя с помощью команды useradd, если поиск не дал результатов, или делать ещё что-нибудь полезное.

Сравнение чисел

В скриптах можно сравнивать числовые значения. Ниже приведён список соответствующих команд.

n1 -eq n2Возвращает истинное значение, если n1 равно n2.  
n1 -ge n2 Возвращает истинное значение, если n1больше или равно n2.  
n1 -gt n2Возвращает истинное значение, если n1 больше n2.  
n1 -le n2Возвращает истинное значение, если n1меньше или равно n2.  
n1 -lt n2Возвращает истинное значение, если n1 меньше n2.  
n1 -ne n2Возвращает истинное значение, если n1не равно n2.

В качестве примера опробуем один из операторов сравнения. Обратите внимание на то, что выражение заключено в квадратные скобки.

#!/bin/bash

val1=6

if [ $val1 -gt 5 ]

then

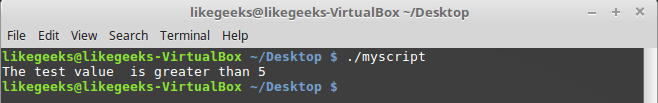
echo "The test value $val1 is greater than 5"

else

echo "The test value $val1 is not greater than 5"

fi

Вот что выведет эта команда.



*Сравнение чисел в скриптах*  
  
Значение переменной val1больше чем 5, в итоге выполняется ветвь thenоператора сравнения и в консоль выводится соответствующее сообщение.

Сравнение строк

В сценариях можно сравнивать и строковые значения. Операторы сравнения выглядят довольно просто, однако у операций сравнения строк есть определённые особенности, которых мы коснёмся ниже. Вот список операторов.

str1 = str2 Проверяет строки на равенство, возвращает истину, если строки идентичны.  
str1 != str2Возвращает истину, если строки не идентичны.  
str1 < str2Возвращает истину, если str1меньше, чем str2.  
str1 > str2 Возвращает истину, если str1больше, чем str2.  
-n str1 Возвращает истину, если длина str1больше нуля.  
-z str1Возвращает истину, если длина str1равна нулю.

Вот пример сравнения строк в сценарии:

#!/bin/bash

user ="likegeeks"

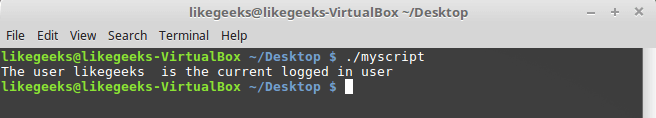
if [$user = $USER]

then

echo "The user $user  is the current logged in user"

fi

В результате выполнения скрипта получим следующее.



*Сравнение строк в скриптах*  
  
Вот одна особенность сравнения строк, о которой стоит упомянуть. А именно, операторы «>» и «<» необходимо экранировать с помощью обратной косой черты, иначе скрипт будет работать неправильно, хотя сообщений об ошибках и не появится. Скрипт интерпретирует знак «>» как команду перенаправления вывода.  
  
Вот как работа с этими операторами выглядит в коде:

#!/bin/bash

val1=text

val2="another text"

if [ $val1 \> $val2 ]

then

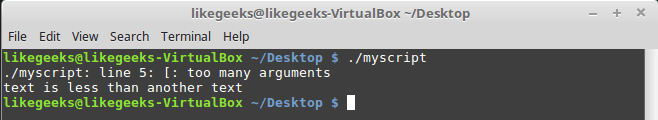
echo "$val1 is greater than $val2"

else

echo "$val1 is less than $val2"

fi

Вот результаты работы скрипта.



*Сравнение строк, выведенное предупреждение*  
  
Обратите внимание на то, что скрипт, хотя и выполняется, выдаёт предупреждение:

./myscript: line 5: [: too many arguments

Для того, чтобы избавиться от этого предупреждения, заключим $val2 в двойные кавычки:

#!/bin/bash

val1=text

val2="another text"

if [ $val1 \> "$val2" ]

then

echo "$val1 is greater than $val2"

else

echo "$val1 is less than $val2"

fi

Теперь всё работает как надо.



*Сравнение строк*  
  
Ещё одна особенность операторов «>» и «<» заключается в том, как они работают с символами в верхнем и нижнем регистрах. Для того, чтобы понять эту особенность, подготовим текстовый файл с таким содержимым:

Likegeeks

likegeeks

Сохраним его, дав имя myfile, после чего выполним в терминале такую команду:

sort myfile

Она отсортирует строки из файла так:

likegeeks

Likegeeks

Команда sort, по умолчанию, сортирует строки по возрастанию, то есть строчная буква в нашем примере меньше прописной. Теперь подготовим скрипт, который будет сравнивать те же строки:

#!/bin/bash

val1=Likegeeks

val2=likegeeks

if [ $val1 \> $val2 ]

then

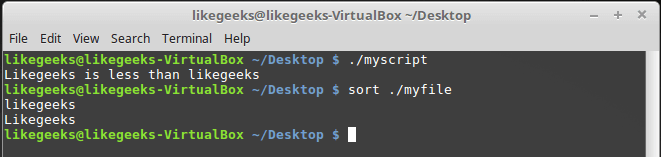
echo "$val1 is greater than $val2"

else

echo "$val1 is less than $val2"

fi

Если его запустить, окажется, что всё наоборот — строчная буква теперь больше прописной.



*Команда sort и сравнение строк в файле сценария*  
  
В командах сравнения прописные буквы меньше строчных. Сравнение строк здесь выполняется путём сравнения ASCII-кодов символов, порядок сортировки, таким образом, зависит от кодов символов.  
  
Команда sort, в свою очередь, использует порядок сортировки, заданный в настройках системного языка.

Проверки файлов

Пожалуй, нижеприведённые команды используются в bash-скриптах чаще всего. Они позволяют проверять различные условия, касающиеся файлов. Вот список этих команд.

-d fileПроверяет, существует ли файл, и является ли он директорией.  
-e fileПроверяет, существует ли файл.  
-f file Проверяет, существует ли файл, и является ли он файлом.  
-r fileПроверяет, существует ли файл, и доступен ли он для чтения.  
-s file Проверяет, существует ли файл, и не является ли он пустым.  
-w fileПроверяет, существует ли файл, и доступен ли он для записи.  
-x fileПроверяет, существует ли файл, и является ли он исполняемым.  
file1 -nt file2 Проверяет, новее ли file1, чем file2.  
file1 -ot file2Проверяет, старше ли file1, чем file2.  
-O file Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.  
-G fileПроверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Эти команды, как впрочем, и многие другие рассмотренные сегодня, несложно запомнить. Их имена, являясь сокращениями от различных слов, прямо указывают на выполняемые ими проверки.  
  
Опробуем одну из команд на практике:

#!/bin/bash

mydir=/home/likegeeks

if [ -d $mydir ]

then

echo "The $mydir directory exists"

cd $ mydir

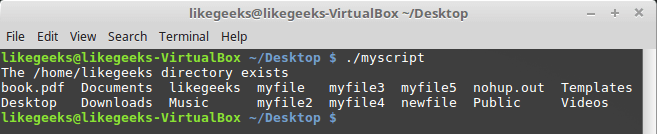
ls

else

echo "The $mydir directory does not exist"

fi

Этот скрипт, для существующей директории, выведет её содержимое.



*Вывод содержимого директории*  
  
Полагаем, с остальными командами вы сможете поэкспериментировать самостоятельно, все они применяются по тому же принципу.

Итоги

Сегодня мы рассказали о том, как приступить к написанию bash-скриптов и рассмотрели некоторые базовые вещи. На самом деле, тема bash-программирования огромна. Эта статья является переводом первой части большой серии из 11 материалов. Если вы хотите продолжения прямо сейчас — вот список оригиналов этих материалов. Для удобства сюда включён и тот, перевод которого вы только что прочли.

1. [Bash Script Step By Step](https://likegeeks.com/bash-script-easy-guide/) — здесь речь идёт о том, как начать создание bash-скриптов, рассмотрено использование переменных, описаны условные конструкции, вычисления, сравнения чисел, строк, выяснение сведений о файлах.
2. [Bash Scripting Part 2, Bash the awesome](https://likegeeks.com/bash-scripting-step-step-part2/) — тут раскрываются особенности работы с циклами for и while.
3. [Bash Scripting Part 3, Parameters & options](https://likegeeks.com/linux-bash-scripting-awesome-guide-part3/) — этот материал посвящён параметрам командной строки и ключам, которые можно передавать скриптам, работе с данными, которые вводит пользователь, и которые можно читать из файлов.
4. [Bash Scripting Part 4, Input & Output](https://likegeeks.com/shell-scripting-awesome-guide-part4/) — здесь речь идёт о дескрипторах файлов и о работе с ними, о потоках ввода, вывода, ошибок, о перенаправлении вывода.
5. [Bash Scripting Part 5, Sighals & Jobs](https://likegeeks.com/linux-bash-scripting-awesome-guide-part5/) — этот материал посвящён сигналам Linux, их обработке в скриптах, запуску сценариев по расписанию.
6. [Bash Scripting Part 6, Functions](https://likegeeks.com/bash-functions/) — тут можно узнать о создании и использовании функций в скриптах, о разработке библиотек.
7. [Bash Scripting Part 7, Using sed](https://likegeeks.com/sed-linux/) — эта статья посвящена работе с потоковым текстовым редактором sed.
8. [Bash Scripting Part 8, Using awk](https://likegeeks.com/awk-command/) — данный материал посвящён программированию на языке обработки данных awk.
9. [Bash Scripting Part 9, Regular Expressions](https://likegeeks.com/regex-tutorial-linux/) — тут можно почитать об использовании регулярных выражений в bash-скриптах.
10. [Bash Scripting Part 10, Practical Examples](https://likegeeks.com/write-shell-script/) — здесь приведены приёмы работы с сообщениями, которые можно отправлять пользователям, а так же методика мониторинга диска.
11. [Bash Scripting Part 11, Expect Command](https://likegeeks.com/expect-command/) — этот материал посвящён средству Expect, с помощью которого можно автоматизировать взаимодействие с интерактивными утилитами. В частности, здесь идёт речь об expect-скриптах и об их взаимодействии с bash-скриптами и другими программами.

Полагаем, одно из ценных свойств этой серии статей заключается в том, что она, начинаясь с самого простого, подходящего для пользователей любого уровня, постепенно ведёт к довольно серьёзным темам, давая шанс всем желающим продвинуться в деле создания сценариев командной строки Linux.  
  
Уважаемые читатели! Просим гуру bash-программирования рассказать о том, как они добрались до вершин мастерства, поделиться секретами, а от тех, кто только что написал свой первый скрипт, ждём впечатлений.