UNIX/Linux Operating System

Shell scripts

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction to shell scripts

- Shell languages are interpreted languages
  - There is no explicit compilation
- Pros & Cons
  - Shell available in every UNIX / Linux environment
  - Faster production cycle
  - Lower run-time efficiency
  - Fewer debugging possibilities
  - Used for writing "Quick and dirty‟ software

Introduction to shell scripts

- Scripts
  - Are normally stored in files with **`.sh`** extension (**`.bash`**)
  - But recall that the extensions are not used UNIX/Linux to determine the file type
- They can be executed using two techniques
  - Direct execution
  - Indirect execution

Direct execution

```
./scriptname args
```

- The script is executed from the command line as a normal executable file
  - The script file must have the execute permission
    - `chmod + x ./scriptname`
  - The first line of the script can specify the name of the script interpreter
    - `#!/bin/bash` or `#!/bin/sh`
  - It is possible to execute the script using a specific shell
    - `/bin/bash ./scriptname args`

Direct execution

```
./scriptname args
```

- The script is executed by a sub-shell
  - i.e., by a new shell process
  - Environment (variables) of the original process and of the new one are not the same
  - Changes to the environment variables made by the script, and used within the script, are lost at exit

Indirect execution

```
source ./scriptname args
```

- The source command executes the script given as its argument
  - It is the current shell to run the script
    - "The current shell sources the script"
  - It is not necessary that the script is executable
  - The changes made by the script to environment variables remain in effect in the current shell

Example: direct and indirect execution

```
#!/bin/bash
# NULL Script
exit 0
```

# indicates a comment

Direct execution:

> scriptName.sh

The shell executes the script as a sub-shell. Executing **exit** the sub-shell terminates. **The initial process resumes control**.

Indirect execution :

> source scriptName.sh

The shell executes the script. Executing **exit the shell process terminates**

Script debugging

- A script can be debugged
  - Partially (only a few lines of the script)
  - Fully (the whole script)
- Partial debug
  - This is done using the **set** command
    - `set -v … set +v`
      - Displays the shell commands in … before running it
    - `Set –x … set +x`
      - Displays the execution trace, i.e., displays the **result** of the commands in …

Script debugging

- **Fully** debug the script
  - Use the options **`-v/-x`** with the script command
    - **`-v`**
      - Displays the commands executed by the script
      - Direct/indirect execution
        - `/bin/bash -v ./scriptname args`
        - `#!/bin/bash -v`
    - **`-x`**
      - Displays the **results** of the commands executed by the script
      - Direct/indirect execution
        - `/bin/bash -x ./scriptname args`
        - `#!/bin/bash -x`

Syntax: general rules

- The bash language is relatively "high level", offering
  - Standard shell commands
    - `ls, wc, find, grep, ...`
  - Standard constructs of the shell language
    - Input and output variables and parameters, operators (arithmetic, logic, etc.), control constructs (conditional, iterative), arrays, functions, etc.
- Often instructions/commands are written in separate lines
  - on the same line, they must be separated by '`;`'

Syntax: general rules

- Comments
  - Character **#** indicates the presence of a comment on the line
  - A comment begins by character **#** and terminates at the end of line
- **`exit`** allows terminating a script returning an error code
  - exit
  - exit 0/1
- In shell, 0 means TRUE

Example of shell commands

Absolute path

```
#!/bin/bash

# This line is a comment

rm -rf ../newDir/
mkdir ../newDir/
cp * ../newDir/
ls ../newDir/ ;

# 0 is TRUE in shell programming

exit 0
```

';' superfluous

Arguments

- The arguments of the command line are identified by **$**
- Positional parameters
  - **`$0`** is the script name
  - **`$1, $2, $3, ...`** indicate the arguments passed to the script on the command line
- Special parameters
  - **`$*`** Is the entire list (string) of arguments (excluding the script name)
  - **`$#`** Is the number of parameters (excluding the script name)
  - **`$$`** Is the process PID

Argument passing example

```
#!/bin/bash

# Using command line parameters in a script

echo "Process $0 is running"
echo "First argument: $1"
echo "Second argument: $2"
echo "Number of arguments $#"
echo "Argument list $*"
echo "Home directory $HOME"
echo "Path $PATH"
exit 0
```

$1 (and the others) can also be written outside "…"

Usage of the predefined environment variables **HOME** and **PATH**

Variables

- Variables can be
  - **Local** (shell variables)
    - Available only in the current shell
  - **Global** (environment variables)
    - Available in all sub-shells
    - Are **exported** by the current shell

Variables

- Main features of shell variables
  - Are not declared
  - A variable is created by assigning a value to the variable name
  - Are case sensitive
    - **`Var`**, **`VAR`**, and **`var`** are different variables
  - Some names are reserved for special purposes
- The list of all defined variables and associated value is displayed by command **`set`**
- The unset command clears the value of a variable
  - **`unset name`**

Local (shell) variables

- Characterized by a name and associated content
  - The contents associated to a name are strings (even if a string can be interpreted as a numeric value)
  - The content specifies the type
    - Constant, string, integer, vector or matrix
  - Setting
    - **`name="value"`**
  - Usage
    - **`$name`**

No blanks around '='

Double quotes are mandatory if the string includes blank characters

Examples

```
> var= "Hello world"
> echo $var
Hello world

> var=7+5
> echo $var
7+5

➢ i=Hello world
➢ world: command not found
```

Variables are strings !!

Assign an arithmetic expression to a variable (more details later)

Assignment is incorrect (do to the blank)
Use quotes

```
➢let var=7+5
➢echo $var
12
```

Global (environment) variables

- The **`export`** command allows creating an environment variable visible by other processes
  - **`export name`**
- Notice that
- Some environment variable names are predefined and reserved
- These variable names are typically uppercase
- Can be displayed by means of the **`printenv`** command

Example: local and global variable

```
> v=one
> echo $v
one
> bash
> ps –l
… Two bashes running
> echo $v

> exit
> echo $v
one
```

This variable is not set

Current shell local variable

```
> v=one
> echo $v
one
> export v
> bash
> ps –l
… Two bashes running
> echo $v
one
> exit
> echo $v
one
```

Global variable because it has been exported by the sub-shell

Example: variables

Clear video

```
#!/bin/bash
clear
echo "Hello, $USER!"
echo
echo "List logged users"
who
echo "Set two local variables"
COLOR="black"; VALUE="9"

echo "String: $COLOR"
echo "Number: $VALUE"
echo
echo "Completed"

#exit
```

who: shows the logged users

Set commands on the same line

Also without explicit exit

Predefined variables

Partial list

Variable	Meaning
$?	Stores the return value of the last process: 0 if successful, other than 0 (between 1 and 255) in case of error. Value 0 corresponds to the TRUE value (unlike in C language)
$SHELL	Current shell
$LOGNAME	Username used for login
$HOME	User home directory
$PATH	List of the directories, delimited by ':' used for searching the executable files and commands
$PS1 $PS2	Main prompt (usually '$' for users, '#' for root) Auxiliary prompt (usually '>')
$IFS	Lists the characters that delimits the "words" in an input string (see **read** shell command)

Examples

```
$ PS1="> "
> echo $HOME
...
> v=$PS1
> echo $PS1
...
> PS1="myPrompt > "
myPrompt > echo $v
...
```

shell prompt modifications

```
> myExe
myExe: command not found
> PATH=$PATH:.
> myExe
... myExe running ...
```

PATH modification, adding current directory

```
> ls foo
ls: cannot access foo:
No such file or directory
> echo $?
2
> ls bar*
bar.txt
> echo $?
0
```

Return value (0=TRUE)

Read from `stdin`

- The **`read`** function allows reading a line from standard input
- Syntax
  - **`read [options]`** $\mathtt{var_1}$ $\mathtt{var_2}$ **`...`** $\mathtt{var_n}$
    - **`read`** can be possibly followed by a list of variables
    - The "words" of the read line will be assigned in turn to each variable
    - Possible excess words are **all** stored (as a string) in the last variable
    - If no variables are specified, the complete input string is stored in variable **`REPLY`**

Read from `stdin`

- Supported options
  - **`-n NCHARS`**
    - Returns after reading **`NCHARS`** characters without waiting for newline
  - **`-t timeout`**
    - Timeout on reading
    - Returns 1 if a string is not typed within **`timeout`** seconds

Examples: read from stdin

```
> read v
input line string
> echo $v
input line string
```

Input string assigned to variable **v**

2 variables, but input string includes 3 words

Input string assigned to the default variable **REPLY**

```
> read v1 v2
input line string
> echo $v1
input
> echo $v2
line string

> read
> One two three
> echo $REPLY
One two three
> read
One two three
> v=$REPLY
> echo $v
One two three
```

Exercise

- Write a bash script that takes two numbers and prints their sum and product

```
#!/bin/bash
# Sum and product

echo -n "Reading n1: "
read n1
echo -n "Reading n2: "
read n2
let s=n1+n2
let p=n1*n2
echo "Sum: $s"
echo "Product: $p"

exit 0
```

-n no newline

from stdin

Arithmetic expression (more detail later)

No blanks around **=, +, ***

Exercise

- Write a bash script that reads a username, and displays her/his number of logins
  - The list of logged users is produced by command **`who`**

```
#!/bin/bash
# Number of login(s) of a specific user

echo -n "User name: "
read user

# who is logged | look for username | word count
times=$(who | grep $user | wc -l)

echo "User $user has $times login(s)"

exit 0
```

`-l` = # of lines

Exercise

- Write a bash script that reads a string, and displays its length

```
#!/bin/bash
# String length

echo "Type a word: "
read word

# echoing without newline | word count chars
l=$(echo -n $word | wc -c)

echo "Word $word is $l characters long"

exit 0
```

-c = # of char

Output

- Output on **`stdout`** can be performed using
  - **`echo`**
  - **`printf`**
- Function **`printf`** syntax is similar to C language printf
  - Uses escape characters
  - It is not necessary to delimit fields by `","`

Output

- **echo**
  - Displays its arguments, delimited by blank, and terminated by newline
  - Options
    - -n eliminates the newline
    - **-e** interprets escaped (\...) characters
      - **\b** backspace
      - **\n** newline
      - **\t** tab
      - \\ backslash
      - etc.

Examples: I / O

```
echo "Printing with a newline"
echo –n "Printing without newline"
echo –e "Deal with \n escape \t\t characters"
printf "Printing without newline"
printf "%s \t%s\n" "Hello. It’s me:" "$HOME"
```

```
#!/bin/bash
# Interactive input/output
echo –n "Insert a sentence: "
read w1 w2 others
echo "Word 1 is: $w1"
echo "Word 2 is: $w2"
echo "The rest of the line is: $others"
exit 0
```

Arithmetic expressions

- Several notations can be used for defining arithmetic expressions
  - Command **`let "..."`**
  - Double parentheses **`((...))`**
  - Square parentheses **`[...]`**
  - Syntactic statement **`expr`**
    - Evaluates an expression by means of a new shell
    - Less efficient
    - Normally not used

Notice that an arithmetic expression is evaluated as TRUE (exit status) IFF it is not 0
expression !=0 → TRUE exit status=0 → TRUE

Examples

Use of ((e))

```
> i=1
> ((v1=i+1))
> ((v2=$i+1))
> v3=$(($i+1))
> v4=$((i+1))
> echo $i $v1 $v2 $v3 $v4
1 2 2 2 2
```

Use of let

```
> i=1
> let v1=i+1
> let "v2 = i +  1"
> let v3=$i+1
> echo $i $v1 $v2 $v3
1 2 2 2
```

Use of [e]

```
> i=1
> v1=$[$i+1]
> v2=$[i+1]
> echo $i $v1 $v2
1 2 2
```

The expression can include blanks using "..."

Conditional statement: if-then-fi

- The conditional statement **`if-then-fi`**
  - Checks if the exit status of a sequence of commands is equal to 0
    - Recall: 0=TRUE in UNIX shell
  - If so, it executes one or more commands
- The statement can also include an else condition statement
    - **`if-then-else-fi`**
    - which allows also performing nested checks

Conditional statement: `if-then-fi`

```
# Syntax 1
if condExpr
then
  statements
fi
```

Standard format

Statement on a single line: ';' is mandatory

```
# Syntax 2
if condExpr ; then
  statements
fi
```

With else

```
# Syntax 3
if condExpr
then
  statements
else
  statements
fi
```

Nested **if-then-else-fi** can be written as **if-then-elif-fi**

```
# Sntax 4
if condExpr
then
  statements
elif condExpr
then
  statements
else
  statements
fi
```

Conditional statement: `if-then-fi`

- condExpr
  - Conditional expressions can use two syntactic flavors

```
# Syntax 1
test param op param
```

Different operators for
- Numbers
- Strings
- Logical values
- Files and directories

```
# Syntax 2
[ param op param ]
```

Square parentheses must be **delimited by a blank**

Conditional statement: `if-then-fi`

Operators for numbers	
-eq	==
-ne	!=
-gt	>
-ge	>=
-lt	<
-le	<=
!	! (not)

Operators for files and directories	
-d	Argument is a directory
-f	Argument is a regular file
-e	Argument exists
-r	Argument has read permission
-w	Argument has write permission
-x	Argument has execution permission
-s	Argument has non-null dimension

Operators for strings	
=	strcmp
!=	!strcmp
-n string	non NULL string
-z string	NULL (empty) string

Logical operators	
!	NOT
-a	AND (inside [])
-o	OR (inside [])
&&	AND (in a sequence of commands)
||	OR (in a sequence of commands)

Examples

```
if [ ]      # NULL is false
   [ str ] # a random string is true
```

Logical values

```
if [ $v1 –eq $v2 ]
then
  echo "v1==v2"
fi
```

or

```
if test $v1 –eq $v2
```

Test on numbers

```
if [ $v1 -lt 10 ]
then
  echo "$v1 < 10"
else
  echo "$v1 >= 10"
fi
```

Examples: file check

```
if [ "$a" -eq 24 -a "$s" = "str" ]; then
  ...
fi
```

AND of conditions

Equivalent format ([≡ test command)

```
if [ "$a" -eq 24 ] && [ "$s" = "str" ]
 if [[ "$a" -eq 24 && "$s" = "str" ]]
```

```
if [ $recursiveSearch -eq 1 -a -d $2 ]
then
  find $2 -name *.c > $3
else
  find $2 -maxdepth 1 *.c > $3
fi
```

Examples: string check

```
if [ $string = "abc" ]; then
  echo "string \"abc\" found"
fi
```

Test on strings

If $string is null (e.g., return from input) the syntax is incorrect because is evaluated as: **`[ = "abc" ]`**
Use double quotes for a error resistant syntax:
`if [ "$string" = "abc" ]; then`
which would be evaluated as: **`[ "" = "abc" ]`**

```
if [ -f foo.c ]; then
  echo "foo.c is in this directory"
fi
```

Test on file

Examples

```
#!/bin/sh

echo –n "Is it morning (yes/no)? "
read string
if [ "$string" = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

Examples

```
#!/bin/sh

echo -n "Is it morning (yes/no)? "
read string
if [ "$string" = "yes" ]; then
  echo "Good morning"
elif [ "$string" = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, wrong answer"
fi

exit 0
```

Uses **elif**

Iterative statement for-in

- Statement **`for-in`**
  - Executes the commands, for each value taken by variable **`var`**
  - The list of values can be given
    - Explicitly (list)
    - Implicitly (result of shell commands di shell, wild-cards, etc.)

```
# Syntax 1
for var in list
do
  statements
done
```

```
# Syntax 2
for var in list; do
  statements
done
```

Examples: for with list

```
for str in foo bar echo charlie tango
do
  echo $str
done
```

Displays a list of strings

Displays a list of "numbers"

```
for foo in 1 2 3 4 5 6 7 8 9 10
do
  echo $foo
done
```

Examples: for with shell commands

```
# Cycle using a variable
num="2 4 6 9 2.3 5.9"
for file in $num
do
  echo $file
done
```

Displays a list of "numbers" using a variable (array, see later)

```
for i in $(echo {1..50})
do
  echo -n "$i " >> number.txt
done
```

Append the numbers from 1 to 50 to file **`number.txt`**

'**>**' would overwrite **`number.txt`** at every iteration

Examples: for and wild-chars

Iteration on the script arguments

```
n=1
for i in $* ; do
  echo "arg#" $n = $i
  let n=n+1
done
```

Display all argument received on the command line

Remove files with name beginning by a OR b

```
for file in [ab]* ; do
  rm -rf $file
  echo "Removing file $file"
done
```

Changes the privileges of files with name including digit 7

```
for f in $(ls | grep 7); do chmod g+x $f; done
```

Iterative statement while-do-done

- Iterates while the condition is true

```
# Syntax 1

while [ cond ]
do
  statements
done
```

```
# Syntax 2

while [ cond ] ; do
  statements
done
```

Example

```
#!/bin/bash

limit=10
var=0
while [ "$var" -lt "$limit" ]
do
  echo "Here var is equal to $var"
  let var=var+1
done

exit 0
```

Displays 10 times a message

Example

Displays a message until the correct string is given

```
#!/bin/bash

echo "Enter password: "

read myPass
while [ "$myPass" != "secret" ]; do
  echo "Sorry. Try again."
  read myPass
done

exit 0
```

Example of read with stdin redirection

```
#!/bin/bash

n=1
while read row
do
   echo "Row $n: $row"
   let n=n+1
done < in.txt > out.txt

exit 0
```

Normally, reads complete lines form stdin

Constant filenames.

Since the while-do-done statement is considered to be unique, the redirection must be done at the end of the statement

Exercise

- Write a bash script that
  - Takes two integers **`n1`** and **`n2`** from command line, otherwise reads them from **`stdin`**
  - Display a matrix of **`n1`** rows and **`n2`** columns of increasing integer values starting from **`0`**
  - Example

```
> ./myScript 3 4
0  1  2  3
4  5  6  7
8  9  10  11
```

Solution

```
#!/bin/bash
if [ $# -lt 2 ] ; then
  echo -n "Values: "
  read n1 n2
else
  n1=$1
  n2=$2
fi
```

Reads input data

```
k=0
while [ $i -lt $n1 ] ; do
  j=0
  while [ $j -lt $n2 ] ; do
    echo -n "$k "
    let k=k+1
    let j=j+1
  done
  let i=i+1
  echo
done
exit 0
```

Double loop for displaying the values

Break, continue and ':'

- **`break`** and **`continue`** statements have the same meaning in shell and in C language
- Character ':' can be used
  - For creating "null instructions"
    - `if [ -d "$file" ]; then`
    - `  :  # Empty instruction`
    - `fi`
  - For indicating a TRUE condition
    - `while :`

Arrays

- **bash** define also one-dimensional arrays
  - A variable can be defined as an array
    - Explicit declaration is not required (but possible with the **`declare`** construct)
  - Indices start from 0, as in C language

Arrays

- Definition
  - Element-wise
    - `name [index] = "value"`
  - By means of a list of values
    - `name =` (list of values separated by blanks)
- Reference
  - A single element
    - `${name[index]}`
  - All elements
    - `${name[*]}`

A new element can be created at any time

`*` or `@`

The use of **`{}`** is mandatory

Arrays

- Number of elements
  - `${#name[*]}`
- Length of the i-th element (number of characters)
  - `${#name[i]}`

- Statement **`unset`** eliminates an element or an array
  - `unset name[index]`
  - `unset name`

Examples: arrays

Initialized by a list

```
> vet=(1 2 5 hello)
> echo ${vet[0]}
1
> echo ${vet[*]}
1 2 5 hello
> echo ${vet[1-2]}
2 5
> vet[4]=bye
> echo ${var[*]}
1 2 5 hello bye
```

Elimination

```
> unset vet[0]
> echo ${var[*]}
2 5 hello bye
> unset vet
> echo ${var[*]}

> vet[5]=100
> vet[10]=50
> echo ${var[*]}
100 50
```

Non contiguous indexes

Exercise

- Write a bash script that
  - Reads a sequence of numbers, one per line, ending by 0
  - Displays the values read in inverse order
  - Example

```
Input n1: 10
...
Input n10: 100
Input n11: 0
Output: 100  ...  10
```

Solution

```
#!/bin/bash
i=0
while true; do
  echo -n "Input $i: "
  read v
  if [ "$v" -eq "0" ] ; then
    break;
  fi
  vet[$i]=$v
  let i=i+1
done
```

or :

```
echo
let i=i-1
while [ "$i" -ge "0" ]
do
  echo "Output $i: ${vet[$i]}"
  let i=i-1
done
exit 0
```

Output
in inverse order