Principles of Computer Science II

Bash Shell Scripting

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 10

UNIX Shell

- The shell
  - Allows the execution of command scripts
  - Enables alternative methods to carry out complex tasks
  - Provides variables
- Various types of shells exist, e.g., korn, tcsh, zsh ...
- Every user has a preselected shell
  - The selection is stored in the file */etc/passwd*
    `ichatz:x:1000:1000:,,,:/home/ichatz:/bin/bash`
  - The command *chsh* allows to change the preselected shell
- Each shell uses a specific file for user settings

BASH Script Example

```
$ for dir in $PATH
>do
>  if [ -x $dir/gcc ]
>  then
>     echo Found $dir/gcc
>     break
>  else
>     echo Searching $dir/gcc
>  fi
>done
```

- For each folder within the variable `$PATH`
- Check if the folder contains the file gcc
  - If the file is found, print out the *path* and stop
  - Otherwise continue to the next folder.

CSH/TCSH Script Example

```
$ foreach dir ( $path )
>    if ( -x $dir/gcc ) then
>      echo Found $dir/gcc
>      break
>    else
>      echo Searching $dir/gcc
>    endif
>  end
```

- Looks more like a C code
- We will focus on BASH
  - It is based on the Bourne Shell
  - It is open source
  - To use it, simply execute the command: *bash*

Command line

```
# bash
bash-4.4.20#
```

- Left part of # can be changed.
- Right part of # is used to type in commands.
- Offers certain built-in commands
  - Implemented within the BASH source code
  - These commands are executed within the BASH process
- Allows to execute scripts
  - For this reason it is called a UNIX programming environment

Built-in Commands

Command	Description	Exception
cd	Change Folder	`cd ..`
declare	Set a variable	`declare myvar`
echo	Print out a text to the standard output	`echo hello`
exec	Replace bash with another process	`exec ls`
exit	Terminate shell process	`exit`
export	Set a global variable	`export myvar=1`
history	List of command history	`history`
kill	Send a message to a process	`kill 1121`
let	Evaluate an arithmetic expression	`let myvar=3+5`

Built-in Commands

Command	Description	Exception
local	Declare a local variable	`local myvar=5`
pwd	The current folder	`pwd`
read	Read a value from standard input	`read myvar`
readonly	Lock the contents of a variable	`readonly myvar`
return	Complete a function call and return a value	`return 1`
set	List declared variables	`set`
shift	Shifts the command parameters	`shift 2`
test	Evaluate an expression	`test -d temp`
trap	Monitor a signal	`trap "echo Signal" 3`

Input/Output Redirection

- Commands produce an output – using the descriptor > the output is redirected to a file
  `# ls > filelist`
  - A new file is created under the name filelist
  - If the file already exists, the new file will replace the old one.
  - We can use the descriptor >> to redirect the output to an existing file
    `# ls -lt /root/doc >> /root/filelist`
- The commands that require input – using the descriptor < the input is redirected from a file
  `# sort < /root/filelist`
- The output of a command can be redirector to the input of another – using the descriptor |
  `# ls | sort` – *sorting the files of a folder*
  `# ls /root | wc -l` – *counting files*

Processes

- We may execute commands in series by using the delimeter ;
  - Commands are executed one by one. When the first is completed, the next one starts. When the last command is completed, we get a new prompt
  - `# who | sort ; date`
- We may execute commands in the background using the delimeter &
  - The commands are executed and a new prompt is provided immediately
  - `# pr junk | lpr &`
- The execution of a command results to a new process
  - The command *ps* shows up in the list of active processes
  - The command *wait* is active until all the commands executed using the delimeter & complete.

List of processes

```
# ps -a
   PID TTY  TIME CMD
   106  c1  0:01 -sh
  4114  co  0:00 /bin/sh /usr/bin/packman
  2114  co  0:00 -sh
  6762  c1  0:00 ps -a
    87  c2  0:00 getty
    90  c3  0:00 getty
```

- Parameter a – list all the commands created by consoles
- Column PID – unique ID of the process
- Column TTY – the console ID that created the process
- Column TIME – total execution time
- Column CMD – the name of the command

Process management

- To terminate a process we use the command *kill [PID]*
- We may change the priority of a process
  - prefix *nice*
    `# nice pr junk | lpr &`
- We may delay the execution of a command
  - prefix *at*

```
# at 1500
ls -l / /root /dir | wc > allfiles
pr allfiles | lpr ; date > lpr-endtime &
date > lpr-starttime
^D
at: /usr/spool/at/07.111.1500.67 created
#
```

The echo command (1)

- Main way to produce output
- Prints out values of variables
- Recognizes special characters (or meta-characters)

```
bash-4.4.20# echo hello there
hello there
bash-4.4.20# let myvar=1; echo $myvar
1
bash-4.4.20# echo *
junk lpr-starttime temp
bash-4.4.20# echo print '*' "don't"
print * don't
```

The echo command (2)

- May contain more than 1 lines
- May also execute commands

```
bash-4.4.20# echo 'hello
there'
hello
there
bash-4.4.20# echo hello\
 there
hello there
bash-4.4.20# echo `date`
Mon Apr 30 16:12:21 GMT 2007
bash-4.4.20# echo -n `date` " "
Mon Apr 30 16:12:21 GMT 2007 bash-4.4.20#
```

Meta-characters

- The character *?* – defines any single character, e.g., `ls /etc/rc.????`
- The character *** – defines multiple characters, e.g., `ls /etc/rc.*`
- The array *[...]* – defines a specific set of characters, e.g. `ls [abc].c`
- The use of the above meta-characters is also called filename substitution
- We may use these meta-characters in any combination within command execution
- The following command is disabled `mv *.x *.y`

Shell Variables

- The shell allows the declaration of variables
- Initial values of variables are defined in the user settings file
- The scope of the variables is connected with the session
  - Or until the user removes them
- The variables with UPPER-case letters are global – they are transfered to all processes executed by the shell
- The variables with LOWER-case letters are local – they are accessible only by the shell process

```
HOME            # The path to your home directory
term            # The terminal type
```

Shell Variables

- We may use variables at the command line
- We use the descriptor $

```
bash-4.4.20# myvar="hello"; echo $myvar
hello
bash-4.4.20# myvar="ls -la"
bash-4.4.20# $myvar
lrwxrwxrwx  1 bin    operator   2880 Jun  1  1993 bin
-r--r--r--  1 root   operator    448 Jun  1  1993 boot
drwxr-sr-x  2 root   operator  11264 May 11 17:00 dev
...
```

Special Variables

- Some special variables are provided

Variable	Description
USER	User name
HOME	Home folder of user
TERM	Type of terminal
SHELL	Name of shell
PATH	List of folders to look for commands
MANPATH	List of folders to look for manual pages
PWD	Active folder
OLDPWD	Previously active folder
HOSTNAME	Name of the system

Variable Handling

- The commands *env*, *printenv* provide a list of GLOBAL variables
- The command *set* provides a list of LOCAL variables
- To declare a new GLOBAL variable we use the command *export*
- Variable type is define by content type
  - String variables – `myvar = "value"`
  - Integer variables – `declare -i myvar`
  - Constant variables – `readonly me="ichatz"`
  - Array variables – `declare -a MYARRAY`
    `MYARRAY[0]="one"; MYARRAY[1]=5; echo ${MYARRAY[*]}`
- The names of the variables are case-sensitive
- The command *unset* removes a variable

Local vs Global Variables

- A global variable is declared using *export*

```
bash-4.4.20# myvar="hello"
bash-4.4.20# set | grep myvar
myvar=hello
bash-4.4.20# bash                    --- 2nd Shell
bash-4.4.20# set | grep myvar
bash-4.4.20# exit                    --- End of 2nd Shell
bash-4.4.20# export myvar="hello"
bash-4.4.20# set | grep myvar
myvar=hello
bash-4.4.20# bash                    --- 2nd Shell
bash-4.4.20# set | grep myvar
myvar=hello
```

Creation of scripts

- Scripts are used as if they were commands/applications
  - Defined by a source file
- We execute the script using the command *sh*
  - Or directly by setting execute access permissions

```
bash-4.4.20# echo 'who | wc -l' > nu
bash-4.4.20# cat nu
who | wc -l
bash-4.4.20# sh nu
      1
bash-4.4.20# chmod a+x nu
bash-4.4.20# nu
      1
```

Handling (1)

- We may pass parameters to a script at command-line
  - These are called the command-line arguments
- We use arguments as variables

Argument	Description
$0	The name of the script
$1 ... $9	The value of 1st ... 9th argument
$#	Number of arguments
$	All the arguments as string

```
bash-4.4.20# cat nu
echo Files found: `ls -la $1* | wc -l` "($1\*)"
bash-4.4.20# nu /b
Files found: 57 (/b*)
```

Handling Parameters (2)

- In order to access more than 9 parameters
  - We may not use *$10*
- We need to use command *shift x*
  - Shifts the parameters left-wise by *x* positions
  - Shifted parameters are lost (!)

```
bash-4.4.20# cat ten
shift 10
echo $1
echo $* " -- " $#
bash-4.4.20# ten 1 2 3 4 5 6 7 8 9 10
10
10 -- 1
```

Input from the standard input

- We may use the standard input using *read*
  - The syntax is `read var-name`
  - We may use multiple variables
    `read var1 var2 ...`
  - We may output a message before requesting the input
    `read -p "Enter value:" var`

```
bash-4.4.20# read -p "Enter values:" i j k;\
echo i=$i, j=$j, k=$k
abc d e f
i = abc, j = d, k = e f
```

Mathematical Expressions

- Allows the evaluation of mathematical expressions using integers
  - Similar with C programming language
  - No need to explicitely declare a variable as an integer
  - We use *expr* rather than *int*

```
((a = a + 1))
a=$((a+1))
a=$(($a+1))
let a = a + 1
let a++
a=`expr $a + 1`
```

If Expressions

```
if [ condition 1 ]; then
  if [[ condition 2 && condition 3]]; then
  ...
  fi
elif [ condition 4 ] || [ condition 5 ] ; then
  ...
else
  ...
fi
```

- The command *test* allows the evaluation of an expression
  - Returns either true or false
  - Supports broad range of expressions
  - e.g., we might check if we have write access to a given file
    `if test -w "$1"; then echo "File $1 is writable" fi`

Evaluation using test

Expression	Description
-gt	Greater or equal
-ge	Greater
-lt	Smaller
-le	Smaller or equal
-eg	Equal
-ne	Not Equal
-n str	Size of the string bigger than 0
-z str	Empty string
-d file	The file is a folder
-s file	A non empty file
-f file	The file exists
-r file	Read access to file
-w file	Write access to file
-x file	Execution access to file

Evaluation Example (1)

```
bash-4.4.20# cat check.sh
#!/bin/bash
read -p "Enter a filename: " filename
if [ ! -w "$filename" ]; then
  echo "File is not writeable"
  exit 1

elif [ ! -r "$filename"  ] ; then
  echo "File is not readable"
  exit 1
fi
...
```

Evaluation Example (2)

```
bash-4.4.20# cat check.sh
#!/bin/bash
TMPFILE = "diff.out"

diff $1 $2 > $TMPFILE

if [ ! -s "$TMPFILE" ]; then
  echo "Files are the same"

else
  more $TMPFILE

fi

if [ -f "$TMPFILE" ]; then
  rm -rf $TMPFILE
fi
```

Boolean expressions

```
if [ condition 1 && condition a]; then
  if [ condition 2 || condition b]; then
  ...
  fi
elif [ ! condition 3 ] ; then
  ...
else
  ...
fi
```

Case

```
case STRING in

pattern 1 )
  ... ;;

pattern 2 | pattern 3)
  ... ;;

*)
  echo "None of the above";
  ...
esac
```

Case: An Example

```
#!/bin/bash
read -p "Enter command: " command
case $command in
all | ALL )
  echo "Display all files..."
  ls -la;;

list | LIST)
  echo "Display all non-hidden files ..."
  ls -l;;

*)
  echo "Invalid choise"
  ls;;
esac
```

For Loop

```
for VAR in <list>
do
  ...
done
```

```
for i in 6 3 1 2
do
  echo $i
done | sort -n
```

```
for i in *.c
```

While Loop

```
while [ expression ];
do
  ...
done
```

```
i = 1
while [[ $i -lt 10 ]];
do
  echo $i
  ((i++))
done
```

Until Loop

```
until [ expression ];
do
  ...
done
```

```
Stop = "N"
until [[ $Stop = "Y" ]];
do
  ps -ef
  read -p "Do you want to stop? (Y/N)" Stop
done
echo "Stopping..."
```

- Within loops we may use *break* continue

Functions

```
function name [()]
{
  ...
  [return]
}
```

- All functions declaration must be location at the top of the script
- A function may not have any parameters
- Parameters and Return value can be of any type
- Parameters defined within the function are global!
  - We need to explicitely define them as local

Functions: An Example

```
#!/bin/bash
outside = "a global variable"

function mine() {
  local inside="this is local"
  echo $outside
  echo $inside
  outside = "a global with new value"
}

echo $outside
mine
echo $outside
echo $inside
```

3^{th} Assignment

- https://www.rosalind.info/
  - Complete the following challenges:
    fibo, ins, maj, mer, 2sum, bins, ms, par, 3sum, inv, par3, med
  - http://rosalind.info/problems/{challenge}
- Create a GitHub repository and upload the code for each exercise.
- Email `ichatz@diag.uniroma1.it`
  Subject: [PCS2] Homework 3
  A .zip or a .tar.gz file with your python solutions, for all challenges.
  Also send your account user account link:
  http://rosalind.info/users/{username}
- **Deadline: 12/November/2019, 23:59**