Introduction to Shell Programming in Bash

3. Introduction to Scientific Computing with Linux
Part III. Basic Computing

Nidish Narayanaa B[1]

[1]Department of Aerospace Engineering
Indian Institute of Space Science & Technology, Trivandrum

FOSS Club, IIST, 2016

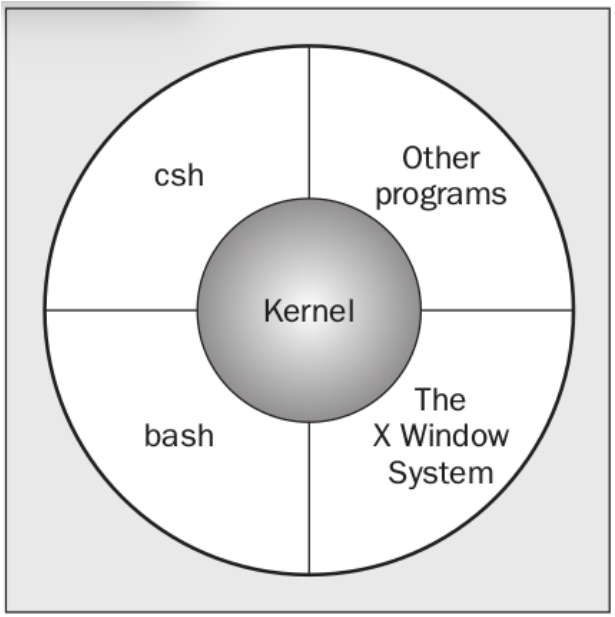
Outline

Outline

The Shell

A shell is a program acting as an interface between the user and the linux system, allowing one to enter other programs that the system should execute

- The GNU *B*ourne *a*gain *sh*ell (Bash) and the *C sh*ell (Csh) are the most common "flavours" of shells in linux
- Our discussion will be concentrated upon Bash



Programming in the Shell

- The shell uses an interpreted language, parsing each statement as one or more programs with their arguments
- One of the main reasons to use the shell is you can do your job simply and quickly
- Batch processing, bulk conversion, etc. of files are just a short script away
- Shell scripts are used when efficiency may take a back hand whilst maintining a coherence work sequence comes to the fore
- They may be used to run commands in predetermined (even conditioned) sequences

Outline

Starting Off

A Sample Program

```
#!/bin/sh

# A comment
for file in *
do
    echo $file
done

exit 0
```

Notes

- The first line of each script must contain the location to the shell program (`sh` in this case) following `#!`
- All statements following a # are ignored as comments
- The `exit` command produces a return value
- To make the script executable, run

  `sudo chmod +x script.sh`
- It is prudent to save shell scripts with a ".sh" extension

Variables

- By default variables are strings in Bash
- To assign a variable use the ‘‘=’’ sign

```
$> var1=Hello
$> var2="Hello World"
```

- Since by default space is the escape sequence, strings with space have to be enclosed in double quotes
- The value of a variable is accessed using the ‘‘$" character. For example, to print the value of `var2` onto the screen we use,

```
$> echo "$var2"
```

- To **read** from user input we use,

```
$> read var2
```

- For outputting text as is use single quotes or precede the special characters by a backslash (\)

Environment Variables

- Since the Linux system itself is presented through a Bash interface, there are several variables pre-defined in the "environment"
- Most of them denote to particular file addresses to aid ease of access
- For example the variable `HOME` points to `/home/nidish_ubuntu1604` in my pc. It points to the home directory
- Another example would be the `PATH` variable. It stores the locations the system searches in when programs are called. Programs stored in `PATH` may be invoked with their names from anywhere in the system.
  `$> echo $PATH$` throws back in my system,
  `/home/nidish_ubuntu1604/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/nidish_ubuntu1604/Tisean`

Environment Variables

- In some cases it may be necessary for us to **add** to existing variables without removing the current values
- For example, to add the path of a development folder to the `PATH` we use,

```
$> export PATH="$PATH:$HOME/Development/bin"
```

- This change will be valid only for the current terminal session. To make it global, you could type it into the file `$HOME/.bashrc` (You need superuser privileges for this)
- Note that the `.bashrc` file will not be visible in a graphical file explorer since the name starts with a "."
- To view all such configuration files, go to your terminal and use `ls` with the `-a` flag

```
$> cd $HOME
$> ls -a
```

Variables

Operations on Variables

- Almost all the scenarios where Bash scripting is necessary involve file conversions and other operations requiring the system to manipulate file names one by one
- We make a list of the most handy operations that may be performed on Variables. In all the below examples `par` is taken as a variable

`${par:-bar}`	Use value of `par` if it exists "bar" if not
`${par:=bar}`	Use value of `par` if it exists assign "bar" if not
`${par:+bar}`	Return "bar" if par exists Not equalling to `NULL`
`${#par}`	Returns length of the variable

Variables

Formatting

- We list some more operations

`${par%word}`	Returns `$par` with the smallest part from the end matching `word` removed
`${par%%word}`	Returns `$par$` with the longest part from the end matching `word` removed
`${par#word}`	Returns `$par$` with the smallest part from the beginning matching `word` removed
`${par##word}`	Returns `$par$` with the longest part from the beginning matching `word` removed

- These tend to come in very handy when we're dealing with converting a set of files and renaming them with appropriate extensions

Parameters

Command-Line Arguments to Bash Scripts

- Command-line arguments may be passed to a Bash Script
- The following list the symbols that may be used to access information about the passed arguments

`$#`	Number of Arguments passed
`$0,$1,$2..`	zeroth argument(script name), and other arguments
`$*`	comma-separated list of all the arguments(excluding script name)
`$@`	space-separated list of all the arguments(excluding script name)
`IFS`	This stores the default field separator used to parse the arguments

- There are ways to work with command line arguments more efficiently, for example by using the `getopt` binary. We wiill not go into this presently

Conditionals

- In Bash all the variables are fundamentally strings denoting file/folder names. The conditionals are built around this.
  The `test` Command
- This command (refer the man page) returns 0 if the condition is satisfied and 1 if not. The `if` command takes input in this format.
- Alternately, the `test` command may be called using the "[" character. Closing the test statement using "]" is a recommended practice.
- The options that may be passed to it are classified into,
  1. File Conditionals
  2. String Conditionals
  3. Arithmetic Conditionals

Conditionals

The test Command

File Conditionals

Call	True if
[-f file]	regular file
[-d file]	directory
[-x file]	executable
[-s file]	size non-zero
[-r file]	readable
[-w file]	writable
[-e file]	exists

Arithmetic Conditionals

Call	True if
e1 -eq e2	expressions equal
e1 -ne e2	expressions inequal
e1 -lt e2	lesser than
e1 -le e2	lesser-equal
e1 -gt e2	greater than
e1 -ge e2	greater-equal
! e1	expression false

String Conditionals

Call	True if
s1 = s2	strings equal
s1 != s2	strings inequal
-n s1	string not null
-z s1	string null

Conditionals

An Example

Example Program

```
if test -f fred.c
then
    echo "FOUND!"
else
    echo "NOT FOUND!"
fi
# (equivalently)
if [ -f fred.c ]
then
    echo "FOUND!"
else
    echo "NOT FOUND!"
fi
```

Notes

- The snippet shows two ways of conducting the same conditional using the `if-then-else-fi` construct
- The statements check for the existence of the file "fred.c" and print "FOUND!" or "NOT FOUND!" onto the screen accordingly
- More examples are attached in an adjoining folder

Conditionals

The case-esac construct

- **Syntax**

```
echo "Type yes :"
read var
case "$var" in
  [yY] | [yY][eE][sS] ) echo "Good Morning!";;
  [nN] | [nN][oO] ) echo "Bad Morning!";;
  * ) echo "Input not recognized";;
esac
```

- The script checks if input is only "y" or "Y" or "yes" with whatever case in the first line
- In the second, it checks if the input is no or it's derivatives
- `*`) denotes the default behavior
- Note the two semicolons ending each statement

Loops

- Loops are very useful in conducting repetitive processes over a similar set of files, strings or numbers
- A very useful command line utility is `sed`, used to produce a set of equispaced numbers as one column
- For example, to produce numbers 1 through 10, the command is,

  ```
  sed 1 1 10
  ```

- The arguments are start number, spacing, and end number, respectively
- Bash has the `for`, `while` and `until` loop constructs
- In syntax, the `while` and `until` constructs are very similar

Loops

The for Loop Construct

Syntax

```
for <var_name> in <set_of_values>;
do
  <loop_body>
done
```

- The for loop operates by assigning a variable by a fixed sequence of numbers
- For example, to list out a set of numbers, either of the following may be used

```
for i in 1 2 3 4 5;
do
  echo $i;
done;
```

```
for i in `seq 1 1 5`;
do
  echo $i;
done;
```

Loops

The while Loop Construct

Syntax

```
while <test_statement>;
do
  <loop_body>
done
```

- The `while` construct works by evaluating the test statement and breaks the loop if it evaluates to false
- It is upto the user to ensure that the loop does not end up ad infinitum. An example,

```
read invar;
while [ "$invar" != "password" ]; do
  echo "Sorry, try again!"
  read invar
done
```

Loops

The until Loop Construct

Syntax

```
until <test_statement>;
do
  <loop_body>
done
```

- The `until` construct works by evaluating the test statement and breaks the loop if it evaluates to true
- It is upto the user to ensure that the loop does not end up ad infinitum. An example,

```
read invar;
until [ "$invar" = "password" ]; do
  echo "Sorry, try again!"
  read invar
done
```

Lists

- Often, we would like a set of statements to be evaluated in a particular secuence conditioned upon the behavior of the others
- In Bash, we have the `AND` and the `OR` lists whose functions are pretty obvious
- The `AND` list is a set of statements separated by a `&&` which will be executed from left to right upon successive return of true. If false is returned in any one statement, false is returned as the return value of the compound statement
- The `OR` list is a set of statements separated by a `||` which will be executed from left to right until one succeeds
- Some striking examples may be found in the directory attached herewith

Functions

Syntax

```
<fn_name>() {
  local <local_var>=<val>;
  <fn_body>
  return <ret_val>;
}
```

- A function is called by its name - arguments may be passed to it as command line arguments are passed while calling a regular program
- The `echo` strings of a function may be captured into a separate variable during the call by `res="$(func args)"`
- The return value of the last function called may be accessed through `$?`
- The `local` keyword may be used for local variables
- Check the adjoining folder for examples

Commands

- In `bash`, in addition to normally encountered constructs, there are what are called, `commands`
- These help in improving comfort in terms of short hands and other such constructs. We list some out here
- The `eval` command may be used to evaluate the value of a variable who's name is stored in another variable,
- The `:` command can replace `true` for more efficient usage

The eval command

```
foo=10
x=foo
eval y='$'$x
echo $y
```

The : command

```
if [ -f file ]; then
    :
else
    echo "Inexistant!"
fi
```

Commands

- The `exec <cmd_name> <args>` command may be used to replace the current shell with a different program
- The `exit` command is used to return an exit value to the shell. Values -1 to 125 are blocked for programs and may be used. *126 means teh file was not executable; 127 means a command was not found; 128 and above mean that a signal occurred*
- The `export <var>` ensures that a variable defined in a shell script is available in any script called from the script
- The `expr` command evaluates its arguments as an expression. In Bash, all arithmetic are done in integers. The following are equivalent

```
x=$(expr $x + 1)
x=‘expr $x + 1‘
x=&(($x+1))
```

Here Docs

Write a script right here

- In some situations, we might want to execute a script from another interpreter and we would like to have the script in the current one itself
- The `here documents` come to our rescue. We set the program to be called, and an end string. All text written in between the first call and the end string are passed into the interpreter specified

cat

```
cat <<!FUNKY!
hello
this is a here
document
!FUNKY!
```

gnuplot

```
gnuplot -p<<EOF
set grid
set zeroax lt -1
plot sin(x)
EOF
```

Thank You[1]

- The current tutorial was made in Ubuntu 16.04, Unity
- The presentation was typeset using Latex (Beamer)

Reference


Mathew, N. & Stones, R. *Beginning Linux Programming*, 3rd ed. Wiley Publishing, Inc., 2004 ISBN: 0-7645-4497-7

[1] Do check out the adjoining directory for a small introduction to GUI design with the dialog command