Bash Shell

# Session 5

Monday, December 23, 2013

## Part I: Customize and use the shell environment

*Candidates should be able to customize existing scripts, or write simple new BASH scripts.*

### Key Knowledge Areas

* Set environment variables (e.g. PATH) at login or when spawning a new shell.
* Write BASH functions for frequently used sequences of commands.
* Maintain skeleton directories for new user accounts.
* **Set command search path with the proper directory.**

***Introduction***

The command line interface (CLI or terminal) may seem intimidating at first, but it's important to remember that the command line is really, truly your friend. An army of tools are at your disposal that can take what would be a tedious and lengthy job (like removing the last four characters from every line of a lengthy file) and turn it into a two minute job.

For every Linux distribution the command line prompt will look a little different. For example, on one system you might see your username, the '@' symbol, the machine name, your current directory and the prompt.

**user@linux ~/$**

This is a very common prompt. You may also see your username, a space, the fully qualified domain name of the computer, the full path to your present working directory followed by the prompt

user linux.box.com /home/user$

The prompt varies from system to system based on a number of things. For example, it may be the default configuration set by the creators of your particular Linux distribution. It could also have been configured by the person who administers the computer or by yourself.

The way you configure the look of your command prompt depends on what shell you use and the shell is the piece that most people commonly refer to as "the command line" when, in reality, it is simply a piece of software that provides an interface to the services of a kernel. The distinction between a 'shell' and the 'command line' is simply that a shell refers to a specific piece of software (e.g BASH, tcsh, ksh, etc) that provides a command line interface. Most modern Linux systems use BASH (Bourne Again SHell) as their default shell.

**A Little History**

The commands used at the command line may seem a little cryptic due to their tendency to be very short. This is because the roots of the Linux command line are from systems where a single letter entry could take a significant amount of time to travel from a terminal, to a central server and back to the terminal where it was printed onto a roll of paper. In those old systems, the shorter the input was, the better as it meant less time waiting to issue your command and receive output. The best thing you can do to remember what commands stand for is to find out what word the command is an abbreviation for. This can go a long way to remembering the command later.

***Summary of Common Commands***

* ls - This command 'lists' the contents of your present working directory.
* pwd - Shows you what your present working directory is.
* cd - Lets youchange directories.
* rm - removes one or more files.
* rmdir - Remove an empty directory.
* mkdir - Make a directory.
* ps - Provides a list of currently running processes.
* cp - Copy a file.
* mv - Move a file (this is also used to rename a file, "moving" it from one file name to another.)
* grep - The global regular expression print program lets you search through a file or output of another program.
* find - Find a file on the filesystem
* man - Displays the manual for most commands (including 'man').

**TIP**

For help about a command, use mancommandwhich will bring up the manual for it. Note that some commands are built into your shell and do not have a man page, use your interpreter internal command (should be help <command>).

**Login vs. Non-login shell**

Login shell: Shell started with login, bash -l or su command

Non-login shell: Shell started any other way

**Reason for 2 types of shell**

The login shell reads a series of configuration file as it is started.

The non-login shells inherit settings (environment variables) from the parent program which started it.

**Variable inheritance**

Variables declared inside a shell are inherited by child processes if the variable has been exported.

If a child process changes its own copy of an exported variable, the parent shell's copy is not changed. The changed value is exported to any sub-child processes.

All exported shell variables keep their export settings in the child process.

If a shell script is called from within a shell a new child non-login shell is started.

If a shell script is started with the '.' command within a shell, then the script is run within that current shell.

*Example:* /home/joe/bin/myscript

*Warning:* If the called script runs the command exit, the current shell will be terminated!

**Interactive and non-interactive shells**

Interactive shell: Provides a prompt where the user can type commands.

Non-Interactive shell: No shell prompt – started by calling a shell script or by the command:

sh -c 'command...'

Sequence of events when bash starts

**Interactive-login bash**

bash --login or su - username or from login

/etc/profile Executed first from interactive login shell. It contains system-wide environment settings.

~/.bash\_profile Individual user's shell settings.

~/.bash\_login Executed if ~/.bash\_profile doesn't exist.

~/.profile Executed if ~/.bash\_login or ~/.bash\_profile doesn't exist.

**Interactive non-login bash**

su username or bash -c command

~/.bashrc The only script executed when started. Inherits from parent bash environment.

**Non-Interactive non-login bash** (forked when scripts are run)

The above scripts are not executed but inherit environment from their parent.

BASH\_ENV Reads file in the variable BASH\_ENV.

ENV Reads file in the variable ENV if BASH\_ENV doesn't exist.

**Extra files**

/etc/inputrc System bash line editing (readline) configuration file

~/.inputrc Individual bash line editing (readline) configuration file

~/.bash\_logout Executed (if exists) when a login shell exits.

**Commands for shell/environment variables**

**Variablename=Value** - Assigns a value to a set (existing) or non-set variable.

**export Variablename** or

**declare -x Variablename** - Sets the export tag ON for an existing shell var.

**export Variablename=value** or

**declare -x Variablename=value** - Assign a value to a set (existing) or non-set variable and sets its export tag ON, all in one command.

**env** - Displays all the environment variables (export tag ON)

**export** - Same as env command except the display format is different eg. declare -x PAGER="less"

***Aliases***

Aliases are normally used to create command shortcuts (short names).

Aliases are NOT exportable: not passed-on to sub-shells or child process.

Aliases are not recognized in scripts.

An alias can call another alias within a command.

Example alias li="ls -l"; alias al="li -a" al calls the alias 'li'

Parameters added to alias will be added at the end of the real command.

The parameter variables ($1, $2, $3 ...etc) cannot be used within aliases.

Aliases are often defined in a file run within a script like ~/.bashrc or ~/.profile with the dot '.' command.

Alias commands:

alias - Displays all the current shell aliases.

alias AliasName="command(s)..." - Sets a new alias value.

Example*:*

|  |
| --- |
| # alias cp="cp -i" |

Replaces the original command cp with cp -i for interactive copying (asks before overwriting files).

**unalias** ***AliasName*** - Unsets (deletes) the alias.

***Functions***

* They are normally used like fast local mini-scripts within a shell which need to be called more than once within the interactive shell or script.
* Variables can be passed-on to functions and will be recognized as $1 $2 $3 etc. In fact the following variables are local within a function:

$1 - $9 - Positional parameters

$# - Number of positional parameters

$\* "$1 $2 $3 …"

$@ "$1" "$2" "$3" ...

**Special Parameters $\* and $@**

There are special parameters that allow accessing all the command-line arguments at once. $\* and $@ both will act the same unless they are enclosed in double quotes," ".

**Special Parameter Definitions**

* The $\* special parameter specifies all command-line arguments.
* The $@ special parameter also specifies all command-line arguments.
* The "$\*" special parameter takes the entire list as one argument with spaces between.
* The "$@" special parameter takes the entire list and separates it into separate arguments.
* The positional parameter $0 and all other variables stay global within the shell unless the command local Variable name is given within the function. Within a function, the variable FUNCNAME is used instead of the $0.
* Global shell or exported variables can be changed within the function.
* Functions do not return variables except for the return number, eg. Return 5. The return command will also terminate the function immediately. The return number can then be read as a normal exit value using $?.
* In scripts functions are normally included at the top so that they are read in first.
* Environment functions can be put into a file and read in with the . command.
* Functions may be recursive. No limit is imposed on the number of recursive calls.
* Functions can be exported, using the command: export -f FunctionName

**Function syntax**

FunctionName ()

{

command1 ;

command2 ;

}

**Command Search Priority**

When a command is run, bash tries to find the command in the following sequence:

1. Aliases
2. Functions
3. Builtin commands
4. searching the PATH

The first command found is the one which is run.

To force using a builtin command instead of an alias or a function (in the case the same command name exists as alias or function), use the command builtin.

|  |
| --- |
| # builtin cat /etc/fstab |

***set***

Syntax: set [--abefhkmnptuvxBCHP] [-o option] [arg ...]

The set command is used to:

-Display all bash variables and their values as well as the functions.

*Example:*

set [bash operating attributes] (using options).

-Assign values to positional parameters:

*Example:*

|  |
| --- |
| # set aaa bbb ccc $1 $2 $3 |

The above assigns the value aaa to $1, bbb to $2 and ccc to $3.

**unset**

Syntax: unset [-fv] [name ...]

For each name, remove the corresponding variable or function.

Each unset variable or function is removed from the environment passed to subsequent commands. If any of RANDOM, SECONDS, LINENO, HISTCMD, FUNCNAME, GROUPS, DIRSTACK are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless a name does not exist or is readonly.

If no options are supplied, or the -v option is given, the name refers to a shell variable. Read-only variables may not be unset.

|  |
| --- |
| # unset -v  # unset -f |

The following examples delete the variable DISPLAY, and the function startx respectively.

|  |
| --- |
| # unset DISPLAY  # unset -f startx |

The following is a partial list of the files, terms and utilities that were used.

* /etc/profile
* env
* export
* set
* unset
* ~/.bash\_profile
* ~/.bash\_login
* ~/.profile
* ~/.bashrc
* ~/.bash\_logout
* function
* alias
* lists

## Part II: Customize or Write Simple Scripts

*Candidates should be able to customize existing scripts, or write simple new BASH scripts.*

### Key Knowledge Areas

* Use standard sh syntax (loops, tests).
* Use command substitution.
* Test return values for success or failure or other information provided by a command.
* Perform conditional mailing to the superuser.
* Correctly select the script interpreter through the shebang (#!) line.
* Manage the location, ownership, execution and suid-rights of scripts.

#### What is a shell script?

A shell script is a text file that tells the shell what to do.

It contains the name of the program that is used as the interpreter for the rest of the content of the script. The line starting with #!ProgramPath+Name (normally the first line) designates the interpreter to be used:

#!/bin/bash or #!/bin/sh or #!/usr/bin/perl -w

In reality when the system is asked to start a script, the line starting with #! is read and the appropriate script interpreter is started which in turns reads the script and executes the commands included in it.

**Conditions for running a script**

The script file must be runnable by the user running it (chmod ....) The interpreter must be where the script says it is: the default is to call bash.

**Language used in shell scripts**

The script language depends on the script interpreter used. bash has its own syntax which can be used interactively or in a script.

**Passing parameters to a script**

Scripts can be given up to 9 positional parameters (for all interpreters) or up to 99 parameters with bash.

Inside the script each parameter will be identified as $1 to $9 or ${10} to ${99}

scriptname param1 param2 param3 param4..... param47.....$0 $1 $2 $3 $4 ${47} .....

Some special parameters are automatically set by the Bourne shell, and usually cannot be directly set or modified.

Parameter $n can be modified by the set command inside the script.

(where n is 1-99 for bash)

set aaa bbb ccc ... $1 $2 $3

***Special Parameters***

$n - Positional parameter n max. n=9 ($0 is the name of the shell script)

${nn} - Positional parameter nn (for nn>9)

$# - Number of positional parameters (not including the script program)

$@, $\* - All positional parameters

"$@" - Same as "$1" "$2" . . . "$n"

"$\*" - Same as "$1c$2c . . . $n" c = content of $IFS (default is space)

$? - Exit status of the last command

$$ - Process ID of the current shell

$- - Current options in effect

$! - Process ID of the last background command

$ - Name of the current shell (in this case 'bash')

**The shift command**

The shift command moves the assignment of the positional parameters to the left. If a

script is called like this:

script1 aaa bbb ccc ddd

And the following commands are run inside the script

|  |
| --- |
| # echo $1 $2 $3  # shift  # echo $1 $2 $3 |

The result of the first echo command is:

aaa bbb ccc

The result of the second echo command is:

bbb ccc

**The set and unset commands**

The unset command is normally used to unset values of variables, and the set command to assign values to positional parameters from inside a script. Very useful if a script has been started without positional parameters and after verifying the script assigns default values to them.

set aa bb cc dd $1 $2 $3 $4 - assigns aa to $1, bb to $2, cc to $3 and dd to $4

The set command is also useful for changing properties of bash's behaviour.

One important option of set is:

|  |
| --- |
| # set -o noclobber |

The command causes the redirection symbol (>) to fail to overwrite the contents of an existing file.

***Conditional statements***

Below is a list of the most used conditional directives

The if conditional branching directive

if - allows certain commands to execute only if certain conditions are met.

**Syntax**: (see also the section 'CONDITIONAL EXPRESSIONS' later in this topic):

if <condition\_is\_true> ; then

run\_these\_commands

.................

elseif <condition\_is\_true> ; then

if first condition is not met and this one is met then:

run\_these\_commands

.................

else

if all conditions above are not met then:

run\_these\_commands\_instead

.................

fi

<condition\_is\_true> can be of the following types:

1. Test the status of files or directories.

if test -e /etc/fstab ; then

or

if [ -e /etc/fstab ] ; then

1. Command or script exit code.

if (ifconfig | grep 'ppp0') ; then

1. The contents of a variable corresponding to a certain value:

if $1 ; then - true if $1 has a value in it

if [ "$net" = "eth0" ] ; then - testing strings

if test ["$#" -eq 5 ] ; then

(integer testing)

The case conditional branching directive

case is normally used for conditionally branching to one of several choices depending on

the content of a variable.

**Syntax:**

case <Variable> in

<choice1>)

commands to run

;;

choice2)

commands to run

;;

choice3)

commands to run

;;

\*)

commands to run if none of the above conditions are met.

;;

esac

end of case directive block

Looping in scripts

Used whenever a sequence of commands must be executed more than once.

The while conditional loop directive

The while directive keeps looping and running the commands in its block for as long as its condition(s) (defined in the while statement) is/are met.

**Syntax**:

while <condition\_is\_true> ; do

run\_these\_commands

done

end of while directive block

*Note:* While is often used to ask the user for a keyboard entry of some sort and if the response is not adequate then the request is repeated until the proper information is entered. The while loop is then exited and the program resumes its execution.

The until conditional loop directive

The until loop works exactly the same way as the while loop except that the logic is the opposite:

The loop continues until condition(s) is/are met.

**Syntax**:

until <condition\_is\_true> ; do

run\_these\_commands

done

end of until directive block

The for loop directive

The for directive allows a sequence of commands to be executed as many times as there are items in a given list. Each time the loop runs through, the content of a specific variable becomes value of the current item in the given list.

**Syntax**:

for variable in list ; do

run\_these\_commands

done

end of for directive block

variable =the variable name which will have its content become the current item on each loop round in the given list. The list can also be a variable which contains a list of items.

for item in ~/file1 ~/file2 ~/file3 ; do

echo "------------ Content of $item -----------"

cat $item >> ~/allfiles

done

***Shell functions***

* Shell functions are a series of commands stored in one place that can be used from several points in the script. Parameters can be passed to functions via positional parameters.
* The positional parameters ($1, $2, $3 ...), which will become local to the function. They use the same syntax as for a script except that the first ($0) stays global.
* The variable FUNCNAME is used similarly to, for the same purpose as, $0.
* Special variables like $#, $\*, $@, are also local within the function.
* All other variables are global to the script and can be modified by the functions.
* The command return x (x=return code) can be used as a function exit command and to assign a function return code.

**Syntax**:

FunctionName () {

command ;

command ;

}

or

function FunctionName () {

command ;

command ;

}

*(See functions in the previous section Customize and use the shell environment for more details on shell Functions.)*

**Exit codes and the variable $?**

* All programs, including scripts, return an exit code when their process ends. The exit code helps determine the success or failure of the program or the script. This exit code can be read via the special variable $? and be used to make decisions further in the calling script.
* Generally the exit code of '0' means success and any other code (1-255) means some sort of failure. It is also often referred as the error code.

**The && and || conditional branching**

* The exit code can be used to execute another command (only one) depending upon its success or its failure. The double ampersand '&&' is used to designate the command to run if the exit code is success (0). The double pipe '||' designates the command to run if the exit code is not a success (1-255).

*Example*

|  |
| --- |
| # ifconfig ppp0 && echo "pppd running" || echo "pppd not running" |

If the command ifconfig ppp0 succeeds then the command echo "pppd running" will be executed (&&) otherwise the command echo "pppd not running" will be executed.

Mailing messages to root from a script

Sometimes it is useful to mail a message to root or tother users announcing some anomalies or success in the running of an automated script. The program normally used is 'mail'. See man mail for all the options it uses.

**Syntax1**:

|  |
| --- |
| # mail -s "subject" destination\_mail\_address "message.." |

**Syntax2**:

|  |
| --- |
| # program | mail -s "subject" destination\_mail\_address |

**Syntax3**:

|  |
| --- |
| # mail -s "subject" destination\_mail\_address <<EOM  message body.......  EOM |

*Example*:

|  |
| --- |
| # df | mail -s "HD Space on $(date)" root |

Mails the result of the command df to the local root user.

Location and security for bash scripts

Administration scripts are normally stored in the PATH which is either /usr/local/bin or /root/bin. The normal access rights are 755(rwx r-x r-x) or for more protection by preventing any other user than root to run it: 700(rwx --- ---).

Although the SUID doesn't have any effect on scripts, very old versions of Linux may be affected by SUID being set.

***Conditional Expressions***

The test and [...] commands are used to evaluate conditional expressions with file attributes, strings, and integers. The basic format is: test expression or [ expression ], where expression is the condition you are evaluating. There must be whitespace after the opening bracket, and before the closing bracket. Whitespace must also separate the expression arguments and operators. If the expression evaluates to true, then a zero exit status is returned, otherwise the expression evaluates to false and a non-zero exit status is returned.

**Test File Operators**

**-a** <file> True if file exists.

**-b** <file> True if file exists and is a block special file.

**-c** <file> True if file exists and is a character special file.

**-d** <file> True if file exists and is a directory.

**-e** <file> True if file exists.

**-f** <file> True if file exists and is a regular file.

**-g** <file> True if file exists and is set-group-id.

-**h** <file> True if file exists and is a symbolic link.

**-k** <file> True if file exists and its ``sticky'' bit is set.

**-p** <file> True if file exists and is a named pipe (FIFO).

**-r** <file> True if file exists and is readable.

**-s** <file> True if file exists and has a size greater than zero.

**-t** <fd> True if file descriptor fd is open and refers to a terminal.

**-u** <file> True if file exists and its SUID bit is set.

**-w** <file> True if file exists and is writable.

**-x** <file> True if file exists and is executable.

**-O** <file> True if file exists and is owned by the effective UID.

**-G** <file> True if file exists and is owned by the effective GID.

**-L** <file> True if file exists and is a symbolic link.

**-S** <file> True if file exists and is a socket.

**-N** <file> True if file exists and has been modified since it was last read.

**file1 -nt file2** True if file1 is newer (according to the modification date) than file2, or if file1 exists and file2 does not.

**file1 -ot file2** True if file1 is older than file2, or if file2 exists and file1 does not.

**file1 -ef file2** True if file1 and file2 refer to the same device and inode numbers.

**Test String Operators**

**-n** **string** True if length of string is not zero

**-z** **string** True if length of string is zero

**string** True if string is not set to null

**string1 = string2** True if string1 is equal to string2

**string1 = string2** True if string1 is equal to string2

**string1 = string2** True if string1 is not equal to string2

**string1 < string2** True if string1 sorts before string2 lexicographically in the current locale.

**string1 > string2** True if string1 sorts after string2 lexicographically in the current locale.

**string = pattern** True if string matches pattern

**string = pattern** True if string does not match pattern

**Test Integer Operators**

**exp1 -eq exp2** True if exp1 is equal to exp2 eg. [ "$#" -eq 4 ]

**exp1 -ne exp2** True if exp1 is not equal to exp2 eg. test "$#" -ne 3

**exp1 -le exp2** True if exp1 is less than or equal to exp2

**exp1 -lt exp2** True if exp1 is less than exp2

**exp1 -ge exp2** True if exp1 is greater than or equal to exp2

**exp1 -gt exp2** True if exp1 is greater than exp2

**Other test Operators**

**! exp** True if the given expression is false eg. [ ! -r /etc/motd ]

**exp1 -a exp2** True if both exp1 and exp2 evaluate to true (see example below)

**exp1 -o exp2** True if either exp1 or exp2 evaluate to true

**\( exp \**) True if exp is true; used to group expressions

The \ used to escape parentheses. Use spaces before and after this character

[ "$A" = "$B" -a \( "$C" = "$D" -a "$E" = "$F" \) ]

The following is a partial list of files, terms and utilities that were used.

* for
* while
* test
* if
* read
* seq