COMP 311 Linux Operating System Lab

Lab 5: *Shell Usage and Configuration – Part 1*

Ahmed Tamrawi

جامعة بيرزيت
BIRZEIT UNIVERSITY

Computer Science Department

Linux OS Laboratory Manual (V 1.2)

COMP311

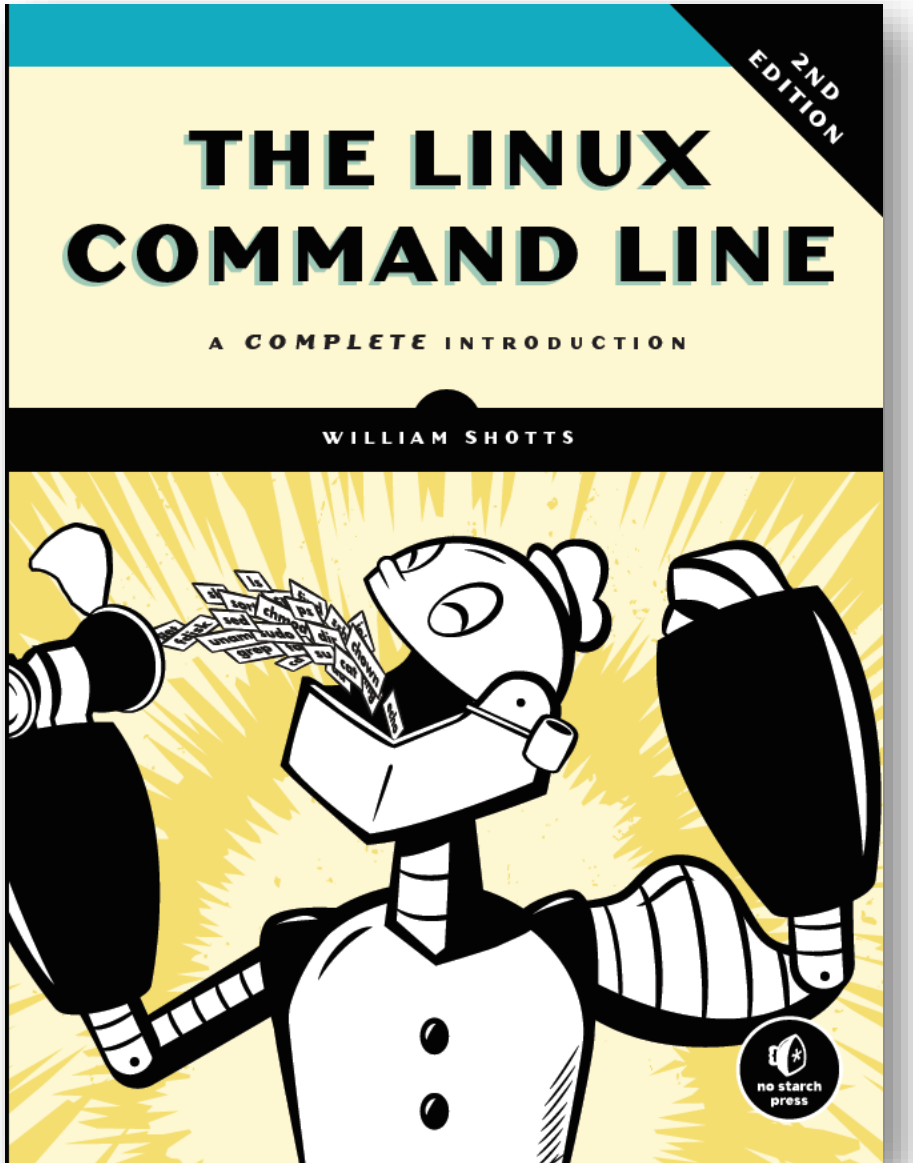
Nael I. Qaraeen

Approved By:

Computer Science department

May 2015

+



What is Shell?

- A **Shell** provides you with an interface to the Unix system. It **gathers input** from you and **executes programs** based on that input. When a program finishes executing, it **displays that program's output**.
- Shell is an **environment** in which we can **run our commands**, programs, and **shell scripts**.

Shell Types

- There are **different flavors** of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.
- In Unix, there are two major types of shells:
  - **Bourne shell** – **$** character is the default prompt.
    - Bourne shell (`sh`)
    - Korn shell (`ksh`)
    - Bourne Again shell (`bash`)
    - POSIX shell (`sh`)
  - **C shell** – **%** character is the default prompt.
    - C shell (`csh`)
    - TENEX/TOPS C shell (`tcsh`)
- To change from one shell to another simply type the name of the shell on the command line and hit `Enter`.

Shell Features: *Expansion*

- Each time we type a command and press the enter key, bash performs **several substitutions upon the text** before it carries out our command.
- We have seen a couple of cases of how a simple character sequence, for example *, can have a lot of meaning to the shell.
- The process that makes this happen is called **expansion**. With expansion, we enter something, and it is expanded into something else before the shell acts upon it.

Shell Features: *Expansion*

- To demonstrate what we mean by this, let's look at the echo command.
- echo is a shell built-in that performs a very simple task. It prints its text arguments on standard output.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

Shell Features: *Expansion*

- Why didn’t echo print *?
- The simple answer is that the shell expands the * into something else before the echo command is executed.
- When the enter key is pressed, the shell **automatically expands any qualifying characters on the command line** before the command is carried out, so the echo command never saw the *, only its expanded result. Knowing this, we can see that echo behaved as expected.

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

```
[me@linuxbox ~]$ echo ~
/home/me
```

Shell Features: *Arithmetic Expansion*

```
$((expression))
```

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion supports only integer arithmetic, results are integers)
%	Modulo, which simply means "remainder"
**	Exponentiation

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
75
```

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

Shell Features: *Command Substitution*

- Command substitution allows us to use the output of a command as an expansion.

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Shell Features: *Command Substitution*

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2:      symbolic link to `bzip2'
/usr/bin/bzip2:        ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/funzip:       ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip:      Bourne shell script text executable
/usr/bin/gunzip:       symbolic link to `../../bin/gunzip'
/usr/bin/gzip:         symbolic link to `../../bin/gzip'
/usr/bin/mzip:         symbolic link to `mtools'
```

Shell Features: *Aliasing*

- We will create a command of our own using the `alias` command. But before we start, we need to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon.

```
command1; command2; command3...
```

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games  include  lib  local  sbin  share  src
/home/me
[me@linuxbox ~]$
```

Shell Features: *Aliasing*

```
alias name='string'
```

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

```
[me@linuxbox ~]$ foo
bin  games  include  lib  local  sbin  share  src
/home/me
[me@linuxbox ~]$
```

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls; cd -'
```

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

Shell Features: *Aliasing*

- To see all the aliases defined in the environment, use the `alias` command **without arguments**.

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

Shell Environment

- The shell maintains a body of information during our shell session called the **environment**.
- Programs use the data stored in the environment to determine facts about the system's configuration.
- The shell stores **three basic types of data in the environment**:
  - **Shell variables** are bits of data placed there by bash. To see those variables use the command: `set | less` (it also displays the **exported** environment variables)
  - **Programmatic data** such as aliases and shell functions. To see those variables use the command: `set | less` (it also displays the **exported** environment variables)
  - **Environment variables** are everything else To see those variables use the command: `env | less` (it displays ONLY the environment variables)

Some Interesting Variables

Variable	Contents
DISPLAY	The name of your display if you are running a graphical environment. Usually this is :0, meaning the first display generated by the X server.
EDITOR	The name of the program to be used for text editing.
SHELL	The name of your shell program.
HOME	The pathname of your home directory.
LANG	Defines the character set and collation order of your language.
OLDPWD	The previous working directory.
PAGER	The name of the program to be used for paging output. This is often set to */usr/bin/less*.
PATH	A colon-separated list of directories that are searched when you enter the name of a executable program.
PS1	Stands for "prompt string 1." This defines the contents of the shell prompt. As we will later see, this can be extensively customized.
PWD	The current working directory.
TERM	The name of your terminal type. Unix-like systems support many terminal protocols; this variable sets the protocol to be used with your terminal emulator.
TZ	Specifies your time zone. Most Unix-like systems maintain the computer's internal clock in *Coordinated Universal Time* (UTC) and then display the local time by applying an offset specified by this variable.
USER	Your username.

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

User-Defined Variables

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool

[me@linuxbox ~]$
```

```
[me@linuxbox ~]$ foo="This is some "
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo"text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

Shell Features: *File Name Completion*

- Another useful shell function is file name completion where the shell completes long file names for you when you type commands.
- Suppose I have a file called : `abcdefghijklmnopqrstuvwxyz` and I need to copy it. All I have to do is type:

```
cp abcESCESC newfilename
```

- If there are no other files starting with `abc` then the shell will complete the long name for me.
- If there are other files that start with `abc` then the shell will display them, and I need to specify the first different character and then press `ESCESC` for the shell to complete the name.
- `TAB` button would result on the same completion behavior.

Shell Features: *Command Line Editing*

- The commands you enter on the command line are stored by the shell in a **history file** called `.bash_history` under the bash shell.
- To use or modify commands you executed earlier you can use the arrow keys. The **up** and **down** keys are used to get commands and the **left** and **right** arrows are used to move and modify a command if needed.
- `.bash_history` file is available under your home directory.

How Is the Environment Established?

- When we log on to the system, the bash program starts and reads a series of configuration scripts called **startup files**, which define the **default environment** shared by all users. This is followed by more startup files in our home directory that define our **personal environment**.
- The exact sequence depends on the type of shell session being started. There are two kinds:
  - **Login Shell Session:** This is one in which we are prompted for our username and password. This happens when we start a virtual console session, for example.
  - **Non-login Shell Session:** This typically occurs when we launch a terminal session in the GUI.

How Is the Environment Established?

- Login shells read one or more startup files.

Table 11-2: Startup Files for Login Shell Sessions

File	Contents
/etc/profile	A global configuration script that applies to all users.
~/.bash_profile	A user's personal startup file. It can be used to extend or override settings in the global configuration script.
~/.bash_login	If *~/.bash_profile* is not found, bash attempts to read this script.
~/.profile	If neither *~/.bash_profile* nor *~/.bash_login* is found, bash attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

How Is the Environment Established?

- Non-login shell sessions read the following **startup files** and **inherit the environment from their parent process**, usually a login shell.

Table 11-3: Startup Files for Non-Login Shell Sessions

File	Contents
/etc/bash.bashrc	A global configuration script that applies to all users.
~/.bashrc	A user's personal startup file. It can be used to extend or override settings in the global configuration script.

What's in a Startup File?

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
```

Lines that begin with a # are comments and are not read by the shell

If the file "~/.bashrc" exists, then read the "~/.bashrc" file.

The original PATH variable is often (but not always, depending on the distribution) set by the /etc/profile startup file

PATH is modified to add the directory $HOME/bin to the end of the list.

The export command tells the shell to make the contents of PATH available to child processes of this shell.

```
[me@linuxbox ~]$ foo="This is some "
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo"text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

Modifying the Startup File

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase
# history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Things to Remember

- Login shell modifications take effect after logging out and logging in again.
- Non-login shell modifications take effect after closing and reopening the shell.

Important Notes for Lab Solution Submission

- **What do you need to submit for each lab?**
  - Reply to the lab message on Ritaj with the following two items:
    - **[OPTIONAL]** *Your lab solution on a separate document.*
    - **[MUST]** *Script log file*; the `script` log file will prove that you have literally solved the lab tasks yourself as it contains timestamps as well as your username and machine name.
  - If you do not send your `script` log file, then you will be rewarded with **zero**.
- **How to generate the script log file?**
  - While solving lab tasks, perform the following per each task part:
    - `script –a UniversityID_FirstnameLastName.log`
    - solve the lab task using proper command, scripts, etc.
    - `exit`