IIT ACM Linux/Git Beginners Tutorial

Frank Lockom, Gregoire Scano

November 9, 2011

1 Introduction

This tutorial will cover basic use of the bash shell and the git version control system.

1.1 Why Use the Shell?

The shell is a direct interface to your operating system. It is a simple matter of more power and control at the cost of increased start-up cost. Using the shell will make you familiar with the flow of data through programs and the way in which programs interact. After you have used the shell for a while a GUI will feel very clumsy and inefficient.

```
 ------------------------
<  It's A UNIX system!  >
 ------------------------
        \   ^__^
         \  (OO)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

1.2 Why use Version Control?

- keep different versions of a program concurrently
- remember all changes made to a program
- collaborate with others

2 Some Prerequisite Material

2.1 File System Layout

The file system may be different than what you are used to (Micro$oft). All files are organized into a single tree. Everything in Linux is a file (or a process). Directories are special files, your hard drive is a file, your speaker is a file, your screen is a file...etc. Disk partitions are *mounted* into the

tree however you find convenient as opposed to seeing them separately (C drive, D drive). The root of the tree is noted as '/'. The top level directories are those located at the root of the tree. They are fairly consistent across different *NIX operating systems and the different Linux distros. Here are some of the more common ones as described by the Filesystem Hierarchy Standard Group [1]

bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
home	User home directories
lib	Essential shared libraries and kernel modules
media	Mount point for removable media (flash drives/cdrom)
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
proc	Kernel and process information virtual filesystem
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	temporary files
usr	Secondary hierarchy
var	Variable data (/var/log contains log files)

2.2 Users Groups and Permissions

Linux is a multi-user system. Every user has a username and a password. Additionally users can belong to groups. Every group can have multiple users and every user can belong to multiple groups. Users and groups are the basic mechanism for managing access rights on a Linux system. The file `/etc/passwd` contains a list of all users and maps usernames to IDs(uid)(among some other things). The file `/etc/group` contains a list of all groups, maps from group names to gids and the users which have membership in them.

Every file has associated with it an owner, a group and permissions. The owner of a file is a user on the system. Permissions consist of a list of access rights for the file which specify who can read/write/execute the file. These rights are specified for each file in relation to three categories of users: the owner, members of the file's group and everyone else. For example:

```
$ ls -l
total 152
-rw-rw---x 1 foo bars   1543 Nov  8 18:24 forkbomb
-rw-r--r-- 1 foo bars    206 Nov  8 18:09 ACM_linux_git_handout.tex
```

The file `forkbomb` can be read and written to by the user `foo` and members of the group `bars`. Everyone else is only allowed to execute the file. On the other hand the file `ACM_linux_git_handout.tex` can be read and written by `foo` and only read by everyone else.

3 The Shell

Lets open up a shell and work with some basic commands.

3.1 Directory Navigation and File Manipulation

The most fundamental thing you need to be able to do is navigate the filesystem. First we should figure out where we are. The command `pwd` (print/present working directory) accomplishes this task.

```
$ pwd
/home/foo
```

We see that we are in home directory of the user `foo`. The organization of users home directories in this way is just a convention, there's no magic (no 'registry' either). This is implemented using the `/etc/passwd` file on Linux.

For examples sake we need a file to work with. Lets create a file called foo.txt. We can use the following command to write into a file from standard input (your keyboard). Type the contents you would like to see in your file then hit the control key and D at the same time. <Ctrl>+D inserts a EOF (end of file) character into the terminal.

```
$ cat > foo.txt
Just fooing around
```

Possibly the most used command is `ls`, which lists the contents of the working directory.

```
$ ls
foo.txt
```

Now we know that the directory `/home/foo` contains one file. If we want to copy a file we use the command `cp`

```
$ cp foo.txt bar.txt
```

Now there is a file called bar.txt which is a copy of foo.txt. One way to view a file is to use the `cat` command. cat takes file names, concatonates their contents and prints the result to standard output(your terminal).

```
$ cat foo.txt
just fooing around
```

This is not always the best way to view a file, if it is really long you should use `less`. less will let you scroll through a file and only reads the file as you need it, so it can be faster than starting up a text editor.

Instead of copying a file you might want to 'cut' it. All you are really doing is *moving* the file from one place to another, hence the command `mv`.

```
$ mv foo.txt baa.txt
```

Now the file foo.txt is called baa.txt. However, baa is rather unconventional[1] so we should delete it. Files can be removed using the `rm` command.

[1] the correct name is baz

```
$ rm baa.txt
```

Careful! there is no magic trashcan(yet) once you rm a file it is gone for good.

We pretty much have all the basics here but we have omitted one important detail, i.e. directories. Lets make a directory, this is done with the command `mkdir`

```
$ mkdir unconventional_metasyntactic_variable_names
```

Lets change into our new directory. Changing directories is done using `cd`.

```
$ cd unconventional_metasyntactic_variable_names
```

I hope you didn't write that all out. After typing `cd u` if you hit the <Tab> key bash will complete the name for you since it is the only file in the pwd which begins with a u.

Here are some additional things you should know:

- The file `~` refers to your home directory. (its the same as writing `/home/foo`)
- The file `.` refers to your pwd
- The file `..` refers to the parent of your pwd
- Any time you give a filename to a command you can give a relative or absolute/full path. An absolute path always starts from root (so it begins with `/`). A relative path is from the pwd

3.2 Environment Variables

Environment Variables are available to every process through the system call *environ*, they are inherited from the parent process through the *exec* system call. You don't really need to know what this means[2]. What you need to understand is that bash creates some environment variables and they are available to the programs that you run in bash.

You can view your environment variables using the command `env`. Some notable ones are listed below.

- USER - the user currently logged in
- HOME - the come directory of USER
- PATH - a list of directories used to search for incomplete filenames(typically programs)
- PWD - your present working directory
- SHELL - USER's login shell(specified in /etc/passwd)
- EDITOR - USER's preferred utility to edit files (should always be set to emacs)
- PS1 - can be used to customize your command prompt

In bash you can use an environment variable by prefixing its name with a '$'.

```
$ echo $USER
foo
```

You can set an environment variable using the bash built-in command `export`.

```
$ export PS1="what should I do now? "
```

[2]Not yet anyways, wait until you take CS351

3.3 Options and Parameters

Where do the commands that we have been using come from? In fact, they are just normal programs most of the are written in C and they are not part of bash[3].

when you enter a command, say `yes`, bash will first check if it is a built-in command and if not begin searching the PATH environment variable for the program and execute the first one it finds. You can use the `which` command to see which program is actually being executed by a command.

```
$ which yes
/usr/bin/yes
```

This means we can also execute yes by typing the whole pathname.

```
$ /usr/bin/yes "press Ctrl+C"
```

Anyways, options/flags/switches are used to change how a program runs. Parameters are the normal input to the program. Sometimes options are used as named parameters so that the order of parameters is irrelevant. Options may take an argument and options' arguments may be required or optional. Below is a partial synopsys of options, refer to the `getopt` manpage for full comprehension.

- short options are a single character. they are specified using a '−'
  - if the option has a required argument it can written immediately after the option character or with a whitespace in-between.
  - if the option has an optional argument it must be written immediately after the option character
  - multiple short options, used without arguments may be specified together after the '−'
- long options are a string. they are specified using '−−'
  - if the option has a required argument it can written after the option separated by whitepsace or by a '='.
  - if the option has an optional argument it must be written after the option string seperated by a '='

After you have specified your options you list your parameters separated by whitespace. Typically order matters with parameters but not options.

3.4 Pipes and IO redirection

Typically when you run a command its input is from your keyboard and its output is to your terminal. These are just files called standard input/output. It is possible to change the file that a program uses for stdin or stdout.

To change the file used for stdout for a command you add '> filename' to the end of the command. Suppose we want a file with 128 random bytes.

[3]There are some built-in commands in bash; cd is one example. See the SHELL BUILTIN COMMANDS section of the bash manpage

```
$ head -c 128 /dev/urandom > random128
```

Other forms of output redirection

- `prog >> out` - appends the output of prog to the file out
- `prog 2> out` - redirect standard error (stderr) from prog to the file out

To change the file used for stdin for a command you add '< filename' to the end of the command. Note that the commands we have been using do not require any input from stdin. Most command line utilities do not require interactive user input. However you have probably written many programs yourself which do. By changing stdin from the keyboard to a file you can automate the running of some interactive program. For example.

```
$ head -c 128 < /dev/urandom > random128
```

Pipes are used to make a ...err pipe from one command to another i.e. link the stdout of one command to the stdin of another.

```
$ cat ~/.bash_history | grep "^ls" | sort
```

3.5 Man Pages (RTFM)

Every program *should* come with a manual. These are in the form of a man page. Man pages have a fairly standard layout but can be difficult to read. However, if you're persistent you will find they tend to be more useful than the random blog post you found on the net. You can look at a manpage using the `man` command

```
$ man ls
```

Refer to the manpage for `man` for information about the general format for manpages

There are multiple *sections* to the manual pages. You will most commonly use sections 1(programs and shell commands), 2(system calls) and 3(library functions). Some names exist in multiple man page sections. So you must explicitly specify a section, you can do this by giving the section number before the name. If you want to search all the sections for an entry use the `whatis` command.

```
$ whatis exit
exit (3p)            - terminate a process
exit (3)             - cause normal process termination
exit (2)             - terminate the calling process
exit (1p)            - cause the shell to exit
$ man 1p exit
```

The p stands for POSIX by the way.

3.6 Editors

This tutorial is not about editors but they must be mentioned. Most of your time will invariably be spent in a editor not cding around on the shell. There are only two choices *emacs* and *vim*. There is an unlimited amount of argument about which is better but the important thing is that you pick one and try to master it.

4 Git

Git is a decentralized Version Control System wrote mainly by Linus Torvalds for the development of Linux. It is a CVS like tool, and unlike Subversion has a strong branching system. The only draw back is the fact that it uses an automerge tool which is efficient but can bypass real semantic conflicts and defects. Indeed, it can only work on the syntax but in CVS, you have to merge all the files by hand which one can find more reliable when it comes to avoiding bugs.

Actually, CVS is old and Git is the "best" alternative, especially facing Subversion (does not have a real branching stuff is has a main server).

Well, lets have a deeper view in Git. By the way, you can access man pages of git commands using "man git branch" for instance.

First, you have to set up a new environment using "git init", it will initiate a new local repository.

At any time, you can use "gitg" to have a graphical representation of your repository.

Git is using a concept called the stagging area. It is a space between unstagged and committed. It allows to have a good interaction with your files, meaning that you cannot directly commit them.

The first step is to "add" a file which is going to add it to the stagged area. To remove a file from the stagging area, you can use the command "reset". Then use commit to put them in the local repo.

Two useful commands are :

- "log" which shows the log entered while committing
- "blame", one can use to see who has changed a part of a file, typically to check who has introduced a bug!!

5 Bash Scripts

A shell is used to enter commands with a high level perspective. The user does not known anything about the operating system and the kernel interface. One could say that it is the barebone of the user interface because it does not need any sophisticated screen management and run directly in a terminal.

On Linux systems, you can switch to terminals using Ctrl-Alt and F1 to F8 keys. The terminal TTY7 is often used to run the X server while the TTY8 shows log of the system.

A shell is made of an Read-Eval-Print loop and commands are build on a top of it. Typically, commands are C programs for the sake of efficiency and portability which are written so as to comply with as many operating system as possible. Thus their code is not easy to read and they could be long to test and correct as the developper has to switch platform.

GNU defined the Coreutils package which provide a high level representation of the operating system as well as devices. The purpose of this package is to provide an interface, whoever uses it is almost certain that it will be portable, and this is why it is also restricted. No fancy commands are allowed and mandatory options are required to release the program such as the "–version" option. Indeed, this option is used by configuration scripts (Makefile, apt) to know the current environment of the user and adapt internal options to match those versions.

Knowing that, shell scripts are programming languages based with the same orientation.

They are not portable but it is easy to change shell from one to another if needed.

There exist several shell interpreters such as :

- Bourne Shell : bash, sh, ksh
- C-Shell : csh, tcsh
- Other shells

Each of them has its own syntax but the concepts are similar.

Let's start with a scripts!!. We are going to write a simple trashcan script which you may use instead of the rm command because what goes with rm is gone except if you are up to recover deleted inodes on your hard drive using foremost!

You can find help here : **http://tldp.org/LDP/abs/html/index.html**

5.1 ONE

Write a simple script which print all its parameters using "for do done" and "echo".

5.2 TWO

Concat all those parameters in one variable called ALL and print it.

5.3 THREE

Save all options given into a second variable called ARGS and print it, use the "if [[ ]]; then fi" statement.

5.4 FOUR

Add a variable DBL_ARGS to save complex arguments. Add a feature to save the argument of the –interactive option.

5.5 FIVE

At this point you will add a variable RMV to store all the files/directories to delete. Make sure that you can save all parameters given to the command.

5.6 SIX

Add a TRASH variable to handle the destination of the deleted files, your trashcan. Use the "export" feature if you can.

5.7 SEVEN

Copy all the RMV nodes to the trash, keeping the tree context using "cp";

5.8 EIGHT

Remove all the old files using "rm".

5.9 NINE

Append the time (date + time) the directories/files while copying them to the trash using “date” and “sed” to remove any space in the string.

5.10 TEN

Test your script and make corrections if needed.

5.11 MORE

Now you have something working pretty well and you have learnt the basics of shell scripting as well as gone over several basic and powerful commands. You can notice that it is painful to write. Well, if your script is simple you may see that scripting is easy to write and fast to debbug. But, if the script begin to be more complicated, you can break it in several part or write it in C.

To have fun, you can now modify your script to add features.

References

[1] Filesystem Hierarchy Standard Group. *Filesystem Hierarchy Standard*. `http://pathname.com/fhs/pub/fhs-2.3.pdf`.