Linux Command Line: Aliases, Prompts and Scripting

By Steven Gordon on Wed, 23/07/2014 - 7:46am

Here are a few notes on using the Bash shell that I have used as a demo to students. It covers man pages, aliases, shell prompts, paths and basics of shell scripting. For details, see the many free manuals of Bash shell scripting.

1. Man Pages

Man pages are the reference manuals for commands. Reading the man pages is best when you know the command to use, but cannot remember the syntax or options. Simply type `man cmd` and then read the manual. E.g.:

```
student@netlab01:~$ man ls
```

If you don't know which command to use to perform some task, then there is a basic search feature called `apropos` which will do a keyword search through an index of man page names and short descriptions. You can run it using either the command `apropos` or using `man -k`. E.g.:

```
student@netlab01:~$ man superuser
No manual entry for superuser
student@netlab01:~$ apropos superuser
su (1)               - change user ID or become superuser
student@netlab01:~$ man -k "new user"
newusers (8)         - update and create new users in batch
useradd (8)          - create a new user or update default new user information
```

Some common used commands are actually not standalone program, but commands built-in to the shell. For example, below demonstrates the creation of aliases using `alias`. But there is no man page for `alias`. Instead, alias is described as part of the `bash` shell man page. Unfortunately the Bash man page is very long. But you can search within man pages. A quick way is, once viewing the man page, press '/' and then type the keyword to search for. To keep searching, press '/' followed by Enter.

```
student@netlab01:~$ man bash
BASH(1)                                                                  BASH(1)

NAME
       bash - GNU Bourne-Again SHell

SYNOPSIS
       bash [options] [file]

COPYRIGHT
       Bash is Copyright (C) 1989-2011 by the Free Software Foundation, Inc.

DESCRIPTION
       Bash  is  an sh-compatible command language interpreter that executes commands read from the
       standard input or from a file.  Bash also incorporates useful features from the Korn  and  C
       shells (ksh and csh).

       Bash is intended to be a conformant implementation of the Shell and Utilities portion of the
       IEEE POSIX specification (IEEE Standard 1003.1).  Bash can be configured to be POSIX-confor-
       mant by default.

/aliases
       BASH_ALIASES
              An  associative  array  variable  whose  members  correspond  to the internal list of
              aliases as maintained by the alias builtin.  Elements added to this array  appear  in
              the  alias  list; unsetting array elements cause aliases to be removed from the alias
              list.
       BASH_ARGC
              An array variable whose values are the number of parameters in each frame of the cur-
              rent  bash  execution call stack.  The number of parameters to the current subroutine
              (shell function or script executed with . or source) is at  the  top  of  the  stack.
              When  a  subroutine  is  executed,  the  number  of  parameters passed is pushed onto
              BASH_ARGC.  The shell sets BASH_ARGC only when in extended debugging  mode  (see  the
```

```
       description of the extdebug option to the shopt builtin below)
BASH_ARGV
       An array variable containing all of the parameters in the current bash execution call
       stack.  The final parameter of the last subroutine call is at the top of  the  stack;
       the  first parameter of the initial call is at the bottom.  When a subroutine is exe-
       cuted, the parameters supplied are pushed onto BASH_ARGV.  The shell  sets  BASH_ARGV
       only  when  in extended debugging mode (see the description of the extdebug option to
       the shopt builtin below)
```

Some commands have a short description in the man page, and then a much longer manual available using `info` (usually they refer to the info page at the end of the man page). Info provides a different interface for exploring documents in a terminal. At first it can be a bit confusing, but if you are lost press 'h' for help. Info actually allows you to explore different programs. Just type:

```
student@netlab01:~$ info
```

For a good summary of the core commands try:

```
student@netlab01:~$ info coreutils
```

For more advanced help, it is probably easiest to search on the Internet.

2. Aliases

Create an alias or shortcut for a commonly used command using the `alias` built-in command of the Bash shell (to see detail, view the man page for `bash` and the search for 'ALIASES'). For example, instead of having to type `ls -R`, you can type (the slightly shorter) `lr` if the alias is defined:

```
student@netlab01:~$ ls -R
.:
its332

./its332:
linux-reference-card.pdf
student@netlab01:~$ alias lr='ls -R'
student@netlab01:~$ lr
.:
its332

./its332:
linux-reference-card.pdf
student@netlab01:~$ lr -l
.:
total 4
drwxr-xr-- 2 student student 4096 Jul 22 13:55 its332

./its332:
total 116
-rw-r--r-- 1 student student 115784 Jul 22 13:55 linux-reference-card.pdf
```

You can view the current aliases, a specific one or delete an alias:

```
student@netlab01:~$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias lr='ls -R'
alias ls='ls --color=auto'
student@netlab01:~$ alias lr
alias lr='ls -R'
student@netlab01:~$ unalias lr
```

```
student@netlab01:~$ alias lr
-bash: alias: lr: not found
```

The aliases defined above are only available during the current terminal session. To have the alias permanently available during any terminal session, it is best to put it into one of the startup scripts which are called when a terminal is opened. On Ubuntu Linux there are several options. One easy option is to edit the file `.bashrc` in your home directory. This file is automatically loaded when your terminal starts. In fact you will see some example aliases already in there.

```
student@netlab01:~$ grep "alias" .bashrc
# enable color support of ls and also add handy aliases
    alias ls='ls --color=auto'
    #alias dir='dir --color=auto'
    #alias vdir='vdir --color=auto'
    alias grep='grep --color=auto'
    alias fgrep='fgrep --color=auto'
    alias egrep='egrep --color=auto'
# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
# Add an "alert" alias for long running commands.  Use like so:
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1
# ~/.bash_aliases, instead of adding them here directly.
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
```

As a 2nd option (which can be more portable across different computers) is to put your aliases (and other shell options) in a separate file called `.bash_aliases`. Note that the last two lines above show that `.bashrc` loads `.bash_aliases` if it exists. Then you can simply copy your `.bash_aliases` file across to other computer when necessary (despite its name, it doesn't have to contain just aliases; later we will use it to customize the prompt). After editing `.bash_aliases`, it may look like:

```
student@netlab01:~$ cat .bash_aliases
# Steve's aliases and shell customizations
alias lr='ls -R'
```

The alias will be available the next time you open a terminal. If you want to load it now, then use `source`:

```
student@netlab01:~$ source .bash_aliases
student@netlab01:~$ alias lr
alias lr='ls -R'
```

Finally, to make your own complex command you may use aliases with Shell functions. Here is an example, that should be included in `.bash_aliases`, that uses a Shell function to start a VirtualBox and then SSH into it. I use this for virtnet [3], where I have multiple Linux guests named node1, node2, node3, etc., and port forwarding setup so that they are accessible on parts 2201, 2202, 2203, etc. The function `sdg_ssh_node()` checks if the node exists, and if so, checks if the node is running, and if not, starts the VM. Then it SSH's into the node. Then `alias` is used to map that function to a command `snode`.

```
sdg_ssh_node() {
        # Check if the node exists in the list of available VMs
        VBoxManage list vms | grep "\"node$1\"" > /dev/null
        isvm=$?
        if [ "${isvm}" = "1" ]; then
                echo "node$1 does not exist. Use 'VBoxManage list vms' to see available nodes."
                return
        else
                # Check if the node is already running
                VBoxManage list runningvms | grep "\"node$1\"" > /dev/null
                isrunning=$?
                if [ "${isrunning}" = "1" ]; then
                        # Start the node
                        VBoxManage startvm --type headless node$1
                fi
```

```
        fi
        # Assumes nodes listen on ports 2201, 2202, 2203, ...
        if [ $1 -lt 10 ]; then
                port=220$1
        else
                port=22$1
        fi
        # SSH into the node, using private key stored in a specific directory
        ssh -o "StrictHostKeyChecking no" -l network -p ${port} -i /home/sgordon/svn/virtnet/data/defaults/h
}

alias snode=sdg_ssh_node
```

To start node1 I use:

```
sgordon@lime:~$ snode 1
```

The above example will not work for you as-is. It is specific to virtnet and my directory setup (I should make it more generic ...). But it may give you ideas for using `alias` to automate tasks.

3. Shell Prompt

The shell prompt is the text shown on the terminal before you type you command. In many of these example (performed on the Network Lab) the shell prompt is:

```
student@netlab01:~$
```

This example contains the logged in user ('student'), the host name of the computer ('netlab01') and the current working directory ('~', meaning home).

The format of the prompt is defined by the shell environment variable `PS1` and be changed by changing that variable. To see prompts explained in detail, view the man page of `bash` and search for 'PROMPTING'. Here I will show just a few examples. First lets view the current prompt:

```
student@netlab01:~$ echo $PS1
\[\e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\u@\h:\w\$
```

You see many escape codes, so it is quite hard to understand all of it. But the most important for this example is the last few characters '\u@\h:\w\$ '. These are escape codes for username (\u), hostname (\h) and working directory (\w). Lets change the prompt, first to a human friendly message, then to a shorter prompt and then back to the original (but without some of the 'debian_chroot' stuff - lets ignore that for now):

```
student@netlab01:~$ PS1='Hello \u. You are in \w. What is your command? '
Hello student. You are in ~. What is your command? ls
its332
Hello student. You are in ~. What is your command? PS1='[\h:\w] '
[netlab01:~] ls
its332
[netlab01:~] PS1='\u@\h:\w\$ '
student@netlab01:~$ ls
its332
```

Generally you want a prompt that is short but contains useful information. On systems that support colour (most systems today), you can also change the colour of the font. Some examples using rather complex escape/colour codes (try them to see):

```
student@netlab01:~$ PS1='\[\e[1;34m\]\u@\h:\w\$\[\e[0m\] '
student@netlab01:~$ PS1='\[\e[1;31m\]\u@\[\e[1;32m\]\h:\[\e[1;35m\]\w\$\[\e[0m\] '
student@netlab01:~$ PS1='\u@\h:\w\$ '
```

4. Shell Scripting Basics

The shell is the software that interprets the commands you type in on a terminal. It is a program itself, and there are many different implementations: sh (the original), Bash, Csh, Tcsh, Zsh, Dash, Ksh, ... Bash is very common today and is the default on Ubuntu Linux and Mac OSX, and therefore we will focus on that.

The shell defines how you interact with the OS on the terminal. The most common interaction is simply typing the name of an application, followed by optional parameters. The shell then executes that application. However a shell has much more, including features that allow you to combine multiple commands to complete more complex tasks than what a single application can do on its own. For convenience, rather than typing a set of commands on the terminal, they are usually included in a file, and then the shell executes that file. Such a file is called a shell script.

The following is a very quick introduction to shell scripting via examples. There are many sources that explain shell scripting, including:

- `man bash` or `info bash` (also available online [4])
- Bash Reference Manual [5]
- Bash Beginners Guide [6]
- Introduction to Bash Programming [7]
- Advanced Bash Scripting [8]

4.1 Shell Scripts are Text Files

Lets create a first shell script. Use a text editor to create a file containing your commands; below I will show the complete file.

```
student@netlab01:~$ cat example1
#!/bin/bash
ls -l ~/
student@netlab01:~$ bash example1
total 8
-rw-rw-r-- 1 student student   21 Jul 23 08:02 example1
drwxr-xr-- 2 student student 4096 Jul 22 13:55 its332
```

The script `example1` is just a text file with two lines. The first line is a special line that indicates to the shell what interpreter (shell) should be used to execute the following commands. Although it is not necessary, it is good practice to include such a line. For my examples I will always include it. Note that later we will see everything after a # (hash) is a comment; however this is a special case where the first two characters of the file are #! (shebang), which means its not actually a comment.

The 2nd line of `example1` is the only command to execute in this script: list in long format the files in my home directory.

You can execute the script by passing its name as a parameter to `bash`. As a result the commands inside the file are executed.

4.2 Variables in Scripts

Variables can be used in shell scripts as demonstrated in the following example. You refer to the value by preceding the variable name with a $ (dollar sign). Optionally, you may enclose the variable name in {} (braces). Everything after a # (hash) is a comment and is not executed.

```
student@netlab01:~$ cat example2
#!/bin/bash
myname="Steven Gordon"
# Variable names can be enclosed in braces { }
echo ${myname}
echo "My name is $myname" # or optionally the braces can be omitted
# It is good practice to include the braces
student@netlab01:~$ bash example2
Steven Gordon
My name is Steven Gordon
```

4.3 For Loops

For loops can loop across numbers, using C-like syntax, as well as loop across lists, including lines in a file. Some examples:

```
student@netlab01:~$ cat data1.txt
123,456,abc
789,012,def
345,678,ghi
student@netlab01:~$ cat example3
#!/bin/bash
for ((i=1; i<=3; i++));
do
        echo $i
done

for name in Steve Thanaruk Pakinee;
do
        echo ${name}
done

for line in `cat data1.txt`;
do
        echo ${line} | cut -d "," -f 2
done
student@netlab01:~$ bash example3
1
2
3
Steve
Thanaruk
Pakinee
456
012
678
```

4.4 If/Then/Else

Conditional statements are possible using if/then/else style. The hardest part is the testing of conditions. This is normally done using the `test` command, which has a short form of enclosing the conditional statement in [] (square brackets). See `man test` to see the syntax for different conditions.

```
student@netlab01:~$ cat example4
#!/bin/bash
cutoff=2
for ((i=1; i<=3; i++));
do
        if [ $i -lt $cutoff ];
        then
                echo "$i is less than $cutoff"
        elif [ $i -eq $cutoff ];
        then
                echo "$i is is equal to $cutoff"
        else
                echo "$i is not less than $cutoff"
        fi
done

for name in Steve Thanaruk Pakinee;
do
        if [ "$name" = "Thanaruk" ];
        then
                echo "$name is the boss"
        fi
done

filename="data1.txt";
if [ -e ${filename} ];
then
        echo "${filename} exists"
fi
student@netlab01:~$ bash example4
1 is less than 2
2 is is equal to 2
3 is not less than 2
```

```
Thanaruk is the boss
data1.txt exists
```

4.5 Input Parameters

A script can take input arguments/parameters, in the same way most commands do. These are called positional parameters and referred to using a number of the position listed on the command line, e.g. $1 is the first parameter, $2 is the second parameter, ...

```
student@netlab01:~$ cat example5
#!/bin/bash
ls -l $1 | grep $2
student@netlab01:~$ bash example5 /usr/bin vlc
-rwxr-xr-x 1 root   root         45 Aug  2  2013 cvlc
-rwxr-xr-x 1 root   root         47 Aug  2  2013 nvlc
-rwxr-xr-x 1 root   root         43 Aug  2  2013 qvlc
-rwxr-xr-x 1 root   root         42 Aug  2  2013 rvlc
-rwxr-xr-x 1 root   root         46 Aug  2  2013 svlc
-rwxr-xr-x 1 root   root      13872 Aug  2  2013 vlc
-rwxr-xr-x 1 root   root       9744 Aug  2  2013 vlc-wrapper
```

4.6 Executing Shell Scripts

So far we have executed the shell scripts by passing the file name as a parameter to bash. Another way is to make the script file executable:

```
student@netlab01:~$ chmod u+x example1
```

And now you can run the script like other programs:

```
student@netlab01:~$ ./example1
total 28
-rw-rw-r-- 1 student student   36 Jul 23 08:27 data1.txt
-rwxrw-r-- 1 student student   21 Jul 23 08:02 example1
-rw-rw-r-- 1 student student  209 Jul 23 08:51 example2
-rw-rw-r-- 1 student student  182 Jul 23 08:28 example3
-rw-rw-r-- 1 student student  423 Jul 23 08:37 example4
-rw-rw-r-- 1 student student   31 Jul 23 08:43 example5
drwxr-xr-- 2 student student 4096 Jul 22 13:55 its332
```

But we need to include "./" in front of the name to tell the shell that the command/program `example1` can be found in 'this' directory. If you want to avoid including "./" then the directory that stores the script must by in the PATH. Lets saw we create a directory that contains all our scripts (`/home/student/bin`) and move them into that directory. Lets also make them executable.

```
student@netlab01:~$ mkdir bin
student@netlab01:~$ mv example* bin/
student@netlab01:~$ chmod u+x bin/*
student@netlab01:~$ ls -l bin/
total 20
-rwxrw-r-- 1 student student  21 Jul 23 08:02 example1
-rwxrw-r-- 1 student student 209 Jul 23 08:51 example2
-rwxrw-r-- 1 student student 182 Jul 23 08:28 example3
-rwxrw-r-- 1 student student 423 Jul 23 08:37 example4
-rwxrw-r-- 1 student student  31 Jul 23 08:43 example5
```

Now lets add our directory to the PATH environment variable. First we show the current PATH, and then add our directory to it:

```
student@netlab01:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
student@netlab01:~$ PATH=/home/student/bin:$PATH
student@netlab01:~$ echo $PATH
/home/student/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Now we can execute our scripts from any directory by just typing the name.

```
student@netlab01:~$ example1
total 12
drwxrwxr-x 2 student student 4096 Jul 23 08:58 bin
-rw-rw-r-- 1 student student   36 Jul 23 08:27 data1.txt
drwxr-xr-- 2 student student 4096 Jul 22 13:55 its332
```

But be careful: some of the example scripts above referred to relative files (e.g. data1.txt), so may longer work. Try to fix them.

Content: Howto [9]
Interest: Linux [10]
Ubuntu Linux [11]

Source URL: http://sandilands.info/sgordon/aliases-prompts-and-scripting-in-linux

Links:
[1] http://sandilands.info/sgordon/aliases-prompts-and-scripting-in-linux
[2] http://sandilands.info/sgordon/user/2
[3] http://sandilands.info/sgordon/virtnet
[4] http://linux.die.net/man/1/bash
[5] https://www.gnu.org/software/bash/manual/bashref.html
[6] http://www.tldp.org/LDP/Bash-Beginners-Guide/html/
[7] http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
[8] http://www.tldp.org/LDP/abs/html/
[9] http://sandilands.info/sgordon/taxonomy/term/212
[10] http://sandilands.info/sgordon/taxonomy/term/300
[11] http://sandilands.info/sgordon/taxonomy/term/302