

Working with the Command Line.

Julian Oliver, June 2011
<http://julianoliver.com>

Disclaimer: this manual is concerned with basic use of the command-line on UNIX and UNIX-like operating systems. It doesn't cover working with Windows DOS or the new Windows 7 PowerShell.

A vast number of networked computers in use today run a UNIX or UNIX-like operating system. Computers in this class include many smartphones, Linux and OS X based personal computers, servers, industrial machinery, robots and embedded computers. Each of these computers shares a common command-line interface, often neglected as a practical and powerful context for control and creation.

One advantage to text-based interaction with a computer is that the computer being manipulated doesn't require a display output or mouse pointer. When in the same room as the operator or at a remote location, it can be accessed only using a network connection.

The companion document, *Working with Computer Networks*, shows how to discover and connect to remote machines.

The kinds of command lines we are interested in working with are authored in a *shell*. The shell we will work with is the popular *Bourne Again Shell*, or **bash**. Other shells also exist, like **zsh** or **tcsh**. In almost all cases most usage of these shells will appear the same, if not identical.

Command lines on UNIX and UNIX-like operating systems often execute small programs. These programs are often vital to the overall functioning of the operating system. These programs are used to process text output, copy files, rename files, create directories, filter program output and manipulate text, among other things.

A shell is an application responsible for managing standard input and output to and from

command line applications and handling their execution context. An execution context includes the present working directory of the application, memory management, execution path and interaction with the permissions system at work on the host operating system. Shells can also be programmed, providing a powerful scripting interface to controlling the host operating system and spawning new processes. Shells are vital to the overall function of UNIX and UNIX-like systems and are also often resourced by GUI applications during installation, operation and removal.

Many of the applications invoked (*called*) in a shell on UNIX and UNIX-like systems are from a suite of tools known as the GNU suite. This is why many people refer to Linux as *GNU/Linux*.

On most Linux and all OS X systems the shell is accessed by an application called a Terminal Emulator, or Terminal for short.

When you start up a terminal, you will see a prompt. A prompt is the place that commands are written and feedback from them is returned. Here's the prompt on the computer on which I'm writing this text:

```
user@netbook:~$
```

Here we see '**user**' is my log in name on the computer called '**netbook**'. The '~' is short for **/home/user** (my *home directory*) while the '\$' sign indicates that I am a user, not administrator (root) in my current log in state. Every user on a UNIX or UNIX-like computer has a home directory in the form **/home/<username>** or **/Users/<username>**. The former is typical of almost all UNIX and UNIX-like systems whereas the latter is found on OS X.

The administrator on a UNIX or UNIX-like machine is called **root** and will always log in by default to their home directory, **/root**.

The administrator login prompt is different, looking like this:

```
root@netbook:~#
```

root is all powerful, able to delete, modify and control the filing and operating system aspects of a computer. For this reason, working as root is generally very dangerous, and so for the rest of this lesson we'll work as a user.

Command lines always have one basic form:

```
program <options> <arguments> <additional options> <additional arguments> [...]
```

Importantly, programs don't always need options and/or arguments. Below we'll see what these individual elements mean.

Normally one types a command and then hits ENTER to execute it. In many cases command lines will provide some output for the user to see whether the command was successful or not. In other cases, the output from command-lines are the very reason the user has typed it. One can think of some command lines as informative or diagnostic while others are concerned with a process, like copying files or converting image formats.

Moving around the command-line

Before we start working with the shell here are a few tips for making your experience of it easier. Firstly, every command you type in a given shell is stored in history, so you don't always need to type it out over and over. Use the UP and DOWN arrow to navigate the history of the shell you're working in.

When you've written a command you may want to move to the beginning or end, for instance to correct an element. Use CTRL-A to get to the beginning of your command and CTRL-E to get to the end.

If at any point you want to copy a line and use it later, move to the beginning of the line and use CTRL-K. That will cut it and copy it to the buffer. Later you can CTRL-Y to paste it.

To move rapidly across individual elements in you can use CTRL-LEFTARROW and CTRL-RIGHTARROW. This saves plenty of time.

Finding out where you are and what's there

Every modern computer has a *file system*. A file system describes the hierarchy of files and folders on a computer. When you open up a shell on a computer you are always placed in the home directory of the user you are logged in as. For instance if your user name is *murdock*, you will find yourself in the directory **/home/murdock**.

Regardless, it's always a good idea to know where you will be executing programs in the file system of the host computer. Use the program **pwd** to find the *Present Working Directory* by typing **pwd** and hitting ENTER. Here's what happens on the computer on which I'm writing this text:

```
user@netbook:~$ pwd
```

/home/user

As a rule, shell programs have a *standard output*, or **stdout**. The stdout of the program **pwd** is the current working directory. Some command-line utilities are also configured to receive *standard input*, or **stdin**.

The **ls** command, without arguments or options, will simply list the contents of the present working directory. **ls** is a program in the directory `/bin/` on most UNIX and UNIX-like systems. Here's the output of **ls** on the computer on which I'm writing this text.

```
user@netbook:~$ ls
file1.txt
Development
Documents
Music
Video
```

The syntax of a command line will often determine whether it will or won't be successfully executed. The legality of the syntax is determined by the command-line program being executed in the shell or by the shell itself, or both. One can think of a shell as an *interpreter*.

Like numerous other shell programs, **ls** has many different options. Options are often known as *switches*; just as with switches on a machine, they alter the functioning and/or output of the program used.

The switches for each command-line program can be found by invoking the manual page. Almost all command-line programs have a manual page visible in the terminal itself by typing:

man <program>

Reading the manual for **ls** we see that it takes the switches (options) **-l** and **-t**. The switch **-l** means long-list, providing information about modification date, file size and ownership. **-t** means sort-by-modification-time.

Switches that take no arguments can, in almost all cases, be combined. So, the command:

```
ls -l -t
```

.. is the same as

```
ls -lt
```

This is the standard output of running **ls** with the switches **-lt** on the system on which I'm

writing this text:

```
user@netbook:~$ ls -lt
total 32
-rw-r-r--  1 user user  517 2011-05-02 00:24 file1.txt
drwxr-xr-x  2 user user 4096 2011-04-22 00:24 Music
drwxr-xr-x  2 user user 4096 2011-04-01 14:04 Desktop
drwxr-xr-x  2 user user 4096 2011-03-12 07:21 Videos
drwxr-xr-x  2 user user 4096 2011-03-07 11:11 Documents
```

These are all the files in my Present Working Directory. The various columns relate to file permissions, users and group membership, file sizes, modification time and the files themselves.

Let's say we *only* want a long-listing for the **Documents** directory in this folder. We can use the filtering program **grep** combined with the special character '|', otherwise known as a *pipe*.

Just like a physical pipe, | connects the output of one program and passes it as input to another program. In this way one can create chains of programs. This is a very powerful feature of UNIX shells. Any number of pipes can be used between programs in a command, as long as the inputs and outputs match up well. Here's how we pass the output of **ls -lt** to the filtering program **grep**:

```
user@netbook:~$ ls -lt | grep Documents
drwxr-xr-x  2 user user 4096 2011-03-07 11:11 Documents
```

Moving around the file-system

I can 'move' into other directories using a tool called **cd**. **cd** is short for 'change directory'. Notice how the prompt changes when we move into another directory.

```
user@netbook:~$ cd Videos
user@netbook:~/Videos$ pwd
/home/user/Videos
```

To move back to where I was I can type:

```
user@netbook:~/Videos$ cd ../
user@netbook:~$ pwd
/home/user
```

The '../' means 'go up one level in the file system'. It has the same result as typing:

```
user@netbook:~/Videos$ cd /home/user
```

Relatedly, you can always get back to the home directory of the user you are logged in as by typing **cd** with no arguments.

```
user@netbook:~/Videos$ cd
user@netbook:~$ pwd
/home/user
```

You can also use **cd -** to return to the last present working directory you were in.

```
user@netbook:~$ cd -
user@netbook:~/Videos$ pwd
/home/user/Videos
```

Basic logging

By using the arrow symbol ('>') we can simultaneously create a file called **my-directory-listing.txt** and direct program output to be written into it.

```
user@netbook:~$: ls -lt > my-directory-listing.txt
```

The file now contains the same output as seen above. To view this output we can use another program called **cat**. **cat** will output all the content of a file to the standard output.

```
user@netbook:~$ cat my-directory-listing.txt
total 32
-rw-r--r-- 1 user user 517 2011-05-02 00:24 file1.txt
drwxr-xr-x 2 user user 4096 2011-04-22 00:24 Music
drwxr-xr-x 2 user user 4096 2011-04-01 14:04 Desktop
drwxr-xr-x 2 user user 4096 2011-03-12 07:21 Videos
drwxr-xr-x 2 user user 4096 2011-03-07 11:11 Documents
```

Creating and manipulating files and directories.

An empty text file can be created using the program **touch**.

```
user@netbook:~$ touch file2.txt
user@netbook:~$ ls
file1.txt  file2.txt  Documents  Music  Videos
```

Content can be added to the file using the program **echo**.

echo normally however does just what the name implies, it echoes the input given to it to the

standard output:

```
user@netbook:~$ echo "test"  
test
```

Just as with **ls** we can send (*redirect*) the output of **echo**, or in fact any other program.

```
user@netbook:~$ echo "this is a line of text" > file2.txt  
user@netbook:~$ cat file2.txt  
this is a line of text
```

New content can be appended to this file using the double output redirection, '>>'. This avoids writing over the current contents.

```
user@netbook:~$ echo "this is another line of text" >> file2.txt  
user@netbook:~$ cat file2.txt  
this is a line of text  
this is another line of text
```

Files can be copied using the program **cp** ('copy') and given a new name on the same line:

```
user@netbook:~$ cp file2.txt file3.txt  
user@netbook:~$ ls  
file1.txt  file2.txt  file3.txt  Documents  Music  Videos
```

The file **file3.txt** has the same contents as **file2.txt**.

We can check for differences between the files by running the command **diff**. **diff** always prints the differences between text files if they exist.

```
user@netbook:~$ diff file2.txt file3.txt  
user@netbook:~$
```

No output from **diff** means that the files are, in fact, the same.

Perhaps later we decide to rename the **file3.txt** as **file2_backup.txt**. It can be done with the command-line utility **mv**, which stands for 'move'.

```
user@netbook:~$ mv file3.txt file2_backup.txt  
user@netbook:~$ ls  
file1.txt  file2.txt  file2_backup.txt  Documents  Music  Videos
```

Now let's create a backup directory, to store our backups. This can be done with the command-line tool **mkdir**, which stands for 'make directory'.

```
user@netbook:~$ mkdir Backups
```

```
user@netbook:~$ ls
file1.txt  file2.txt  file2_backup.txt  Backups  Documents  Music
Videos
```

Let's move our backup file into that new directory with **mv**:

```
user@netbook:~$ mv file2_backup.txt Backups
```

Because **Backups** is a directory, the file will be copied into it rather than over it.

```
user@netbook:~$ ls Backups
file2_backup.txt
```

Working with variables

As with other programming languages, variables can not start with a number. Here is a valid variable assignment:

```
user@netbook:~$ a="fox"
```

For portability across different shells, avoid spaces either side of the assigning sign '='.

Variables can then be called using a '\$' sign. We can use **echo** to *expand* the variable for us, revealing the value assigned to it.

```
user@netbook:~$ echo $a
fox
```

The variable **\$a** can be copied.

```
user@netbook:~$ b=$a
user@netbook:~$ echo $b
fox
```

It can be overwritten:

```
user@netbook:~$ b="fox and wolf"
user@netbook:~$ echo $b
fox and wolf
```

We can select just one word from our new string using the program **awk**. **awk** is a very powerful text processing program. It is very useful for selecting and printing text elements. To use **awk** to print the third word we need to pass the output of **echo** to **awk** via a UNIX pipe.

awk uses curly brackets as part of its special selection syntax. Here we select the 3rd word in the variable:

```
user@netbook:~$ echo $b | awk '{ print $3 }'  
wolf
```

Commands can also be combined to make new variables all on the same line. Consider the creation of the new variable **\$c** using the output of the above command:

```
user@netbook:~$ c=$(echo $b | awk '{ print $3 }')  
user@netbook:~$ echo $c  
wolf
```

See **man awk** for more details.

We can also manipulate characters in the command line using the *stream editor* **sed**. **sed** is a very powerful program for manipulating text. Here is a very simple demonstration of using **sed** to manipulate a word in a sentence using a substitution expression:

```
user@netbook:~$ echo $a | sed 's/wolf/elf/'  
fox and elf
```

Note the trailing **'/'**. This terminates the substitution expression.

sed can also be used on files, provided as an input argument. If you wanted to change all the instances of the word **wolf** to **shewolf** in a document without risking alteration of the original, you could do it like so, writing out a new file:

```
user@netbook:~$ sed 's/wolf/shewolf/' DungeonsAndDragons.txt >  
DungeonsAndDragQueens.txt
```

See **man sed** for more details.

System diagnostics and administration tools

df (*Disk Free*) will output the available disk space on all mounted partitions. With the **-h** switch the output will be more *human readable*, ie, in *kilobytes*, *megabytes* and *gigabytes* rather than just *bytes*.

top will output an updating ordered list of processes and the system resources they consume (memory, % of CPU).

du (*Disk Usage*) This is the tool most commonly used to determine the size of a file. Similar to **df**, it will print *human readable* output when invoked with the **-h** switch.

ps lists running processes of types selected by switches. Unlike most other command-line utilities, **ps** can be invoked in a BSD style (without '-') or a standard UNIX style (with '-'). The most commonly used **ps** switches to use are **a** and **x**, listing all running executable processes. Here are the first few lines of **ps ax** on my system:

```
user@netbook:~$ ps ax
  PID TTY          STAT       TIME COMMAND
    1 ?           Ss        0:02 /sbin/init
    2 ?           S          0:00 [kthreadd]
    3 ?           S          0:00 [ksoftirqd/0]
    6 ?           S          0:00 [migration/0]
   17 ?          S<         0:00 [cpuset]
   18 ?          S<         0:00 [khelper]
   19 ?          S<         0:00 [netns]
   21 ?           S          0:00 [sync_supers]
   22 ?           S          0:00 [bdi-default]
   23 ?          S<         0:00 [kintegrityd]
   24 ?          S<         0:00 [kblockd]
   25 ?          S<         0:00 [kacpid]
```

The only two columns we're interested in here are **PID** (*Process ID*) and **COMMAND**. PIDs are unique numbers used to identify running processes listed under **COMMAND**. Starting at 1 we see `/sbin/init`, the first and most important process when a Linux system is booting.

Say I wish to terminate the process Firefox on a remote machine with which I only have network access to (see the section *Remote Connections* in the document *Working with Networks*).

Using **grep** and a UNIX pipe I can find that process easily:

```
user@netbook:~$ ps ax | grep firefox
5867 ?           Sl         100:26 /usr/lib/firefox-4.0.1/firefox-bin
8300 pts/1       S+          0:00 grep --color=auto firefox
```

Note that **grep** itself turns up in the filtered output. This can be avoided using square brackets over a section of the search string.

It is possible to terminate this process using the **kill** utility. The manual page for **kill** shows that the switch **-9** is a signal that cannot be blocked.

```
user@netbook:~$ kill -9 5867
```

vi or **vim** are editors pre-installed on the vast majority of UNIX and UNIX-like operating systems. vim stands for 'vi improved' and is the successor to the ancient editor.

vim and vi are often described as being a little difficult to use at first. They are nonetheless very useful and powerful, if not only for their ubiquity on other machines.

Here's how to open a file in **vim** and work with it.

```
user@netbook:~$ vim a-file.txt
```

Now press 'i' to go into *Insert Mode*. You can now type freely. To get out of *Insert Mode* and save your changes to the file, hit ESC. Now you need to go into *Command Mode*. To do this, type ':' and then 'wq'. 'wq' stands for *Write and Quit*. Lines can be deleted with 'dd', regions selected with SHIFT-v then cursor keys, copied with 'y' and pasted in Insert Mode with 'p'.

As always, hit ESC if ever you find yourself 'stuck' in a mode in vim.

Scripting in the shell

Commands can be joined together to form a line comprised of many commands. The below command creates a directory, copies files into it and then archives it as a compressed file using the program **tar**. See **man tar** for a description of all the different switches and compression options.

```
user@netbook:~$ mkdir new-folder; cp file1.txt file2.txt file3.txt  
new-folder && tar cvzf folder-backup.tar.gz new-folder
```

The ';' symbol means 'run the next command regardless of whether it fails or not' while the '&&' symbols mean 'only run this command if the previous one doesn't throw an error'.

Commands can also be put into a *shell script* for use later and in a variety of different contexts. Shell scripts begin with the special characters, right at the top of the file:

```
#!/bin/sh
```

When the shell environment encounters these characters, it knows to run everything underneath as shell code.

The below script will takes a folder as an input argument, compresses the contents of that folder into an archive with a name that is the combination of the word 'backup' and the current date. There is no error checking of return values involved. That's beyond the scope of this

document.

```
#!/bin/sh

# create a variable of today's date using the date program (see man
# date)

DATE=$(date '+%d-%m-%y')

# make a directory with the date as its name

mkdir $DATE

# create a variable to hold the folder-name given by the user running
# the script

FOLDER=$1

# copy all the contents of that folder to the folder whose name is
# the date

cp -r $FOLDER/* $DATE &&

# let the user know the archiving process is about to start.

echo "Creating archive.."

# archive all the contents of this folder into an archive with a name
# made from the word 'backup' and today's date

tar cvzf backup_$DATE.tar.gz $DATE

echo "Complete"
```

Now let's make the script executable. We can do this using the program **chmod**, or change mode. By using x with '+' we are assigning the mode executable to the file

```
user@netbook:~$ chmod +x my-script
```

I can now execute that script like so, compressing and archiving the directory **Photos**:

```
user@netbook:~$ ./my-script Photos
```

I can also execute the script like so, without need to make it executable.

```
user@netbook:~$ sh my-script Photos
```

If you were to copy your executable script into the executable directory on a UNIX or UNIX-

like system, like **/bin/** or **/usr/bin/** or **/usr/local/bin** you would then be able to call that script from anywhere in the filesystem, like so:

```
user@netbook:~/Images/Colombia/2011$ my-script MayPhotos
```

To find where the directory for executable programs is on your system, type:

```
user@netbook:~$ echo $PATH
```

\$PATH is an *environment variable* set when you open up any shell on your system. This variable informs the shell as to directories to find executables.

Further reading

To learn more about working and programming in the shell I thoroughly recommend reading the below free manual. If you were to read it end to end, you would become a fluent command-line user:

<http://www.flossmanuals.net/command-line/>