

Chapter 1

UNIX Command Line Basics

The objective of this chapter is to configure the shell account so that printing, manuals, and editor functions are working normally. This will give us experience with the basic commands, environment variables, and workhorse tools needed.

1.1 Logging In To Your Account

Log into your system with the login name and password given to you. You will see something like

```
<bash>:
```

Your system should always prompt you with the name of the shell (bash) and your login name. This is a customizable feature in the **bash** shell which you are now using.

The most basic command in Linux is the directory listing “ls” command.

You can see the contents of your account by typing

```
<bash>: ls -al
```

You should see files like

```
drwxr-xr-x 13 joe      1024 Nov  9 17:06 .dt
-rwxr-xr-x  1 joe      5111 Sep 30 15:19 .dtprofile
drwxr--r--  2 joe           96 Dec  1 12:25 .elm
-rwxr--r--  1 joe      1178 Nov 19 10:58 .emacs
```

1.2 Command Structure

We've already seen `ls` which give a directory listing the current working directory (cwd).

Commands in Linux follow a general format:

Command *-options* Other parameters

for example:

```
<bash>: ls -l .bashrc
```

First comes the command name, followed by options. Options are normally preceded by a dash or minus sign. There is always a space between the command and the dash. Some commands use no options at all. After the options comes any other parameters or informations that command may need.

Let's talk about some of the workhorse commands. Please note that these definitions are purposely abbreviated and incomplete! Consult the `man` pages for each of the commands below. In the commands below, parameters that are enclosed in square brackets [...] are optional to that command. [-opts] refers to options in the style just mentioned.

Special Keys Strokes:

<code>q</code>	quits from many commands like <code>more</code> and <code>less</code>
<code>Ctrl+c</code>	also quits out of many commands
<code>Ctrl+L</code>	clears the screen
<code>Ctrl+a</code>	puts the cursor at the beginning of the command line
<code>Ctrl+e</code>	puts the cursor at the end of the command line

Help, Search, Info Tools:

<code>env [-opts] [exp]</code>	Print environment or run a command with another environment.
<code>find [path] [exp]</code>	Find files in <i>path</i> using <i>exp</i>
<code>info keyword</code>	List <i>info</i> help pages containing <i>keyword</i>
<code>locate keyword</code>	Locate all files of name <i>keyword</i> in a database
<code>man -k keyword</code>	List <i>man</i> pages with <i>keyword</i> (same as <code>apropos keyword</code>)
<code>man command</code>	Display the manual for <i>command</i>
<code>printenv</code>	Print environment variables (see <code>set</code>)
<code>set [vars]</code>	Print/Set environment vars and functions
<code>whatis keyword</code>	Search the <i>whatis</i> DB for <i>keyword</i>
<code>whereis command</code>	Locate source/binary and manuals for <i>command</i>
<code>which command</code>	Display path of <i>command</i>

Text Manipulation Tools

`awk|gawk [pgrm] [file]`ter file by *pgrm*

<code>cat file</code>	Display contents of <i>file</i> without paging
<code>clear</code>	Clears the screen. Same as Ctrl+L
<code>grep pattern file</code>	Finds <i>pattern</i> in <i>file</i>
<code>head file</code>	List the first few lines of <i>file</i>
<code>more file</code>	Display & page the text <i>file</i> (See less)
<code>sed [script] file</code>	Stream edit/filter <i>file</i> using <i>script</i>
<code>tail [-opts] file</code>	List the trailing lines of <i>file</i>
<code>tr chars1 chars2 file</code>	Change chars in <i>chars1</i> to <i>chars2</i>
<code>less file</code>	Display & page the text <i>file</i>

General Tools

<code>cd dir</code>	Change cwd to <i>dir</i> (home if <i>dir</i> omitted)
<code>chmod perms files</code>	Change file permissions of <i>files</i>
<code>chown owner.group file</code>	Change file owner and/or group
<code>chsh</code>	Change the default shell
<code>cp [-opts] f1 (f2 dir)</code>	Copy file <i>f1</i> to <i>f2</i> or directory <i>dir</i>
<code>date</code>	Displays the date
<code>kill pid</code>	Kills process ID <i>pid</i>
<code>ln [-opts] Old New</code>	Link <i>Old</i> to <i>New</i>
<code>login [username]</code>	Login to system with UID <i>username</i>
<code>lpr file</code>	Print <i>file</i> on the default printer
<code>ls [file]</code>	Listing for <i>file</i> (<i>cwd</i> if file omitted)
<code>mkdir dir</code>	Creates directory <i>dir</i>
<code>mv file1 file2</code>	Rename file1 to file2
<code>passwd [-opt] username</code>	Change password
<code>ps [-opts]</code>	Output a list of currently active processes
<code>pwd</code>	List the current working directory
<code>rm files</code>	Removes <i>files</i>
<code>startx</code>	Start the X-Windowing system
<code>tar [-opt][arch][file]</code>	Manage tar archives
<code>telnet [host [port]]</code>	Connect to the remote <i>host</i>
<code>uname [-opts]</code>	Output name and version number of OS
<code>who</code>	List users logged into this system
<code>xterm [-opts]</code>	Start a brand new X-terminal window

1.3 The Linux Manuals and the *man* Utility

Virtually every command that is worth knowing has an entry in the *man* pages, and is accessed by doing a **man command** . To get all related commands to a *keyword* word, use **man -k keyword** as in the following example:

```
<bash>: man -k manual
```

man (1)	- format and display the on-line manual pages
man2html (1)	- format a manual page in html
perlxS (1)	- XS language reference manual
whereis (1)	- locate the binary, source, and manual page files
xman (1x)	- Manual page display program for the X Window System

When in doubt, use `man` and `man -k keyword` to get info on a command or UNIX related term.. Find the man page for the `ls` command. `ls` will list directory contents:

```
<bash>: man ls
```

<pre> LS(1) FSF LS(1) NAME ls - list directory contents SYNOPSIS ls [OPTION]... [FILE]... DESCRIPTION List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuSUX nor --sort. -a, --all do not hide entries starting with etc </pre>
--

Figure 1.1: Manual Page for `ls`

Here is a `man` entry for `cd` which changes your current working directory (folder).

```
<bash>: man cd
```

<pre> cd(n) Tcl Built-In Commands cd(n) NAME cd - Change working directory SYNOPSIS cd dirName DESCRIPTION Change the current working directory to dirName, or to the home directory (as specified in the HOME environment variable) if dirName is not given. Returns an empty string. </pre>

Figure 1.2: Manual Page for `cd`

1.3.1 Exercises

1. `<bash>: man man`
2. `<bash>: man 7 signal`
3. Man `cd` and look for information on “.” and “..”

4. `<bash>: cd .. ; pwd`
5. `<bash>: cd . ; pwd`
6. `<bash>: cd ; pwd`
7. Discuss what “.” and “..” are in terms of the filesystem.

1.4 Create, List, Copy, and Move

The most fundamental of all commands is to list, create, copy, move (rename), and remove files. This section will show you the basics. Lets start with creating some files now, so we can list them later.

1.4.1 Creating New Files

File creation can be done by invoking an editor on a new filename. Before we try this, there is an easier way to make a new empty file by using the `touch` command. This is how it works:

```
<bash>: touch file1
```

This will make a new file named `file1` that is empty. There are many other ways to make new files that we will see later.

1.4.2 Creating New Directories

Directories are created with the `mkdir` command. Thus

```
<bash>: mkdir dir1
```

will create a new directory named `dir1` located in your current working directory. List this file now with the “long” format we just spoke about. Check your files with

```
<bash>: ls -l
```

1.4.3 Listing Files and Directories

By far, listing a file is the most basic of all commands. As we have seen before, you list files with the `ls` command:

```
<bash>: ls file1
```

```
file1
```

```
<bash>: ls -l file1
```

```
-rw-r--r--  1 joe  users      0 Oct 28 22:05 file1
```

Using the `-l` flag causes a “long” listing that shows more information about `file1`.

You list your directories in a similar way (note that the second example shows that `dir1` is empty):

```
<bash>: ls -l
```

```
total 10
```

```
drw-r--r--  1 joe  users    0 Oct 28 22:05 dir1
<bash>: ls -l dir1
total 0
```

1.4.4 Copying Files and Directories

You can copy a file by using the `cp` command. The following statement

```
<bash>: cp file1 file2
```

will copy `file1` to `file2`. Copying a folder or directory requires the use of the *recursive* or `-r` flag indicating that `cp` should descend into the directory and copy all sub-files and sub-folders:

```
<bash>: cp -r dir1 dir2
```

Exercise 1.4.4: Please create a directory like this now. Explain clearly how the following examples are different from the above:

```
<bash>: cp file1 dir1
```

```
<bash>: cp -r dir1 dir2/
```

```
<bash>: cp -r dir1 dir2/dir3
```

1.4.5 Moving Files and Directories

Moving a file is the same as renaming it. This allows for the possibility that you move the file to another directory as well:

```
<bash>: mv file2 file3
```

```
<bash>: mv file1 dir1/file2
```

You move a directory in EXACTLY the same way as a file:

```
<bash>: mv dir1 dir3
```

1.4.6 Changing Directories

Changing your current directory is done with the `cd` command:

```
<bash>: cd dir1
```

```
<bash>: cd ../dir2
```

1.4.7 Removing Files and Directories

Normal files are removed with the `rm` command:

```
<bash>: rm file1
```

Removing directories is also done with the `rm` command, but again you need to use the “recursive” or `-r` option:

```
<bash>: rm -r dir1
```

After you try the above, make sure `file1` and `dir1` are gone.

1.5 I/O, Redirection, and Pipes

I/O refers to Input (I) and Output (O). This section talks about input and output of commands and how you can manipulate these. These principals are very important because they are used constantly in Unix.

1.5.1 Standard I/O

In UNIX, Standard Input (*stdin*) and Standard Output (*stdout*) are mechanisms that allow you to input or output data from a command line. Simple commands like "`cat file1`" send their results to *stdout* (normally to your terminal screen) while the word `file1` is an example of *stdin* which is fed to the command `cat`.

Independent of *stdin* and *stdout*, there is the also standard error (*stderr*) which normally goes to your screen when the command detects an error. Its manipulation is shell specific.

Bash assigns special numbers, called *File Descriptors*, to *stdin*, *stdout*, and *stderr*:

Name	Abbreviation	File Descriptor	Standard Device
Standard Input	<i>stdin</i>	0	Keyboard
Standard Output	<i>stdout</i>	1	Console
Standard Error	<i>stderr</i>	2	Console

Table 1.1: Bash Standard I/O

1.5.2 Redirection

Redirection refers to the art of redirecting input and output traffic from commands. Shells like `bash` allow for redirection of *stdin* and *stdout* with the `<` and `>` operators respectively.

As an example, lets say you want to list some files and send (redirect) the output to a file instead of the screen. Do it the easy way:

```
<bash>: ls -al > output.txt
```

To check the output, you can use `cat` (short for concatenate). `cat` is useful when you want to view short files:

```
<bash>: cat output.txt
-rwx----- 1 carinhas users      1606 Aug 17 23:14 .acrorc
```

```
-rwx----- 1 carinhas users      153 Dec 20 08:47 .bashrc
-rwx----- 1 carinhas users     3189 Dec 23 15:34 .cshrc
.....
```

Examples of *stdin* redirecting:

```
<bash>: cat < output.txt
<bash>: wc -l output.txt
      7 output.txt
<bash>: wc -l < output.txt
      7
```

Note that “`cat output.txt`” and “`cat < output.txt`” give the same result, but the “`wc -l`” examples give something slightly different.

In `bash` *stderr* is redirected with with the `2>` operator, while in `tcsh`, the `>&` operator. Just relax and we will see real examples of this shortly.

Linux Warning: Please note that `>` will overwrite anything in the output file, if it exists, or create the file if it does not exist. In contrast, the `>>` operator will append to the existing file.

Here is a brief summary of the redirects:

Name	Operator	Description
Redirect <i>stdin</i>	<code><</code>	Feeds the file to input
Redirect <i>stdout</i>	<code>></code>	Creates or overwrites
Append <i>stdout</i>	<code>>></code>	Creates or appends
Redirect <i>stderr</i>	<code>>&</code>	Both <i>stdout</i> and <i>stderr</i>
Redirect <i>stderr</i>	<code>2></code>	Only <i>stderr</i> in bash

Table 1.2: Standard Redirection I/O

1.5.3 Pipes

When you want to take the output of one command and use that as input into another, use the “pipe operator” `|`. Think of actually connecting a metal pipe from one command to another. The following example sorts a simple `ls` command in reverse order (do it!):

```
<bash>: ls | sort -r
.doomrc
.cshrc
.bashrc
.acrorc
```

1.5.4 Examples to Try

```
<bash>: cat noname 2> error.txt      # Sends error to error.txt
<bash>: cat noname > error.txt 2>&1  # Send stdout + stderr to error.txt
<bash>: cat noname >& error.txt      # everything (as above) goes to error.txt
```

Try these examples in `tcsh`:

```
<tcsh>: tcsh
<tcsh>: (cat noname > output.txt) >& error.txt      # Send stderr to error.txt
<tcsh>: more output.txt                          # Same as in bash.
<tcsh>: more error.txt                          # Just like in bash.
<tcsh>: cat testfile.txt >& out.txt              # ditto
<tcsh>: cat < testfile.txt | sort | more         # ditto
<tcsh>: locate .bashrc | xargs grep alias        # find 'alias' in .bashrc.
<tcsh>: exit
```

Don't forget the last "exit" to get out of `tcsh` and back into fabulous `bash`.

1.6 Command Line Editing

A great shortcut in **bash** and **tcsh** is to use the command editing facilities. Just hit the up-arrow key to a previous command and move the cursor to edit that command. Just try it. Command editing is very useful in repeating and correcting previous commands.

Now we discuss other features of command line editing.

1.6.1 Command and File Completion

Most shells support command and file completion typing shortcuts. These shortcuts allow you to hit the Tab key to finish off the name of a command or file after only hitting a few keys. Try this example of a filename (directory):

```
<bash>: cd /usr/inc<TAB>    will produce
<bash>: cd /usr/include/
```

Now try this example for a command:

```
<bash>: ghostv<TAB>    should produce
<bash>: ghostview
```

Note that command and file completion can only complete upto a unique set of commands. This means you have to provide a unique start string.

Thus **<bash>: ghost<TAB>** won't work because it could complete to **ghostview** or **ghostscript**.

tcsh has the same file-name completion mechanism as **bash**. The **<TAB>** key will complete the filename and command up to unique names.

1.6.2 Possible Command Completion

bash will show the possible choices are by hitting **<TAB>**twice (**<Ctrl-D>** in **tcsh**):

```
<bash>: ls /etc/h<TAB><TAB>
host.conf  hosts      hosts.allow  hosts.deny  http/
<bash>: ls /etc/h
```

The shell tells you what your choices are and is again ready for more input. This also works on commands too:

```
<bash>: mo<TAB><TAB>
modemlights_applet    montage                mount.smbfs
modemtool              more                   mouse-properties-caplet
```

```

modinfo          morepgp          mouse-test
modprobe         mount           mouseconfig
<bash>: mo

```

This shows us the possibilities, and again returns us so we can continue typing a command.

Remember that **tcs**h has the same possible-completion mechanism as **bash** does above. Just use <Ctrl-D> instead.

1.6.3 Command Line Substitution and History

In both **tcs**h and **bash** we have the facility of *Command Line Substitution* and *Command History*.

Command Line Substitution allows you to substitute a dynamic expression inside a command:

```

<bash>: echo My name is $USER
My name is Joe
<bash>:

```

The \$ symbol allows bash to process the expression in-line and later provide the results to the **echo** command.

Command History allows you to use stuff from your old commands in your current command. Remember: recycling is good for the environment ♣. The following list shows the history reference syntax common to both **bash** and **tcs**h:

!!	Redo the last command. Same as !-1
!-N	Repeat the N^{th} most recent command (see next)
!-3	Repeat the 3rd most recent command
!N	Redo the N^{th} entry in the history list
!<u>string</u>	Redo last command starting with the text "<u>string</u>"
!<u>?string?</u>	The most recent command which contains the text "<u>string</u>"

Figure 1.3: History Referencing Specifications

On top of *history referencing*, you can add these modifiers to the shell command line by appending them to the history reference after a colon (:)

0	The first (command) word
n	The nth word on the command line besides the command.
^	The first word, equivalent to '1'
\$	The last word
%	The word matched by an ?s? search
x-y	A range of words. '-y' abbreviates '0-y'.
*	Equivalent to '^-\$', but returns nothing if the event contains only 1 word
x*	Equivalent to 'x-\$'
x-	Equivalent to 'x*', but omitting the last word ('\$')

Figure 1.4: History Modifiers

1.6.4 Exercises

You can get a list of your history by typing simply **history**. Create two files called `boogie.man` and `boogie.man.old`. The file contents are not important for this exercise. Try these examples of the above history machinery to try. The exact history will vary depending on what you do so please adapt these to your current situation:

```
<bash>: history | tail -4
    9  8:30  touch boogie.man
   10  8:31  cp boogie.man boogie.man.old
   11  8:36  echo hello >> boogie.man
   12  8:37  diff boogie.man.old boogie.man
```

```
<bash>: !-2
vi boogie.man
```

```
<bash>: diff !-2:2.old !-2:2
diff boogie.man.old boogie.man
```

```
<bash>: diff !:1.old !:2
diff boogie.man.old.old boogie.man
diff: boogie.man.old.old: No such file or directory
```

```
<bash>: !?cat?:0 !:2
cat boogie.man
```