A very brief guide to Linux

C. J. Rennie

5th April 2011

This document is for those who are just beginning with Linux. It covers the really essential commands in some detail, but also gives some impression of the great scope of commands that are available. It says nothing about email clients, browsers, compilers etc, which are either documented elsewhere or are sufficiently GUI not to need any documentation such as this.

Logging in and out

You always need to log in by specifying your user name and your password. Once that is done, the Desktop Environment (usually 'Gnome' or 'KDE') starts, and you will be able to do things like open terminal windows, run the file manager, or browse the Internet.

Log off via the menu options provided by the desktop.

To try out the things described in this document it will help to have a 'terminal' window open. How to do this should be obvious from whatever desktop you are using.

Directories

Whatever computing background you have, you will be familiar with the concept of directories (or 'folders'), which are arranged hierarchically, and act as containers for files.

There are between 5 and 10 thousand directories on a Linux machine, so it is good to have some idea of how they are organized. The directory tree consists of branches having certain functions. The details differ on the various Linux versions, but top level or 'root' directory is always designated '`/`', and contains the following directories

Path	Function
`/suphys/`*`your_name`*`/`	For the personal use of users
`/media/`	CD-ROMs, floppies, data directories on other machines
`/usr/`	Where most programs are located
`/bin/`	Core operating system programs
`/sbin/`	Core operating system programs
`/etc/`	Configuration files for operating system

amongst others.

The branches are sorted here according to relevance for everyday work, so concern yourself with the first couple do not try to mess with those near the bottom. (You will be prevented anyway.)

You are free to create directories under your home directory and certain other areas (e.g., under `/media/floppy`), while others area may be protected against modifications. When creating a directory (or file), the names are case-sensitive and you can use spaces Â— however spaces are often confusing and annoying (they have to be quoted), and I advise against the practice.

You can access directories by specifying their name explicitly (i.e., by giving some path beginning with '/', for example `/suphys/chris`), or by using shortcuts. The shortcuts are as follows: '.' means the current directory; '..' means the next directory up; '~' means your home directory. Thus '`~/data`' refers to one of your own directories, '`data`', located under your home directory.

Another great help is auto-completion. If you are part-way through typing a path, then pressing the TAB key will complete the directory or filename. If there is ambiguity in how to complete the name, then TAB will just fill in as much as it can, whereupon you can enter another character or two to resolve the ambiguity and press TAB again. Repeat as often as necessary. If you are unsure how to resolve the ambiguity, you can press CTRL-D at any time to see what the alternatives are.

If the full path of some directory or file is printed somewhere on screen, a common trick for using this name in a command is to cut and paste: highlight the block with left-click and drag; paste it with middle-click. (This is the Linux equivalent of CTRL-C and CTRL-V.) Another way to highlight is by multiple clicks.

Navigating directories on the command line is not as intuitive as with a graphical browser, but can still be done, and done surprisingly efficiently.

Essential commands

This section gives brief, illustrated descriptions of a few really essential commands. They are basic command for dealing with directories and files.

Change you current directory with `cd`:

`cd ..`	change to the next directory up
`cd /usr/bin`	Change to the directory `/usr/bin`
`cd ~`	Change to your home directory

Listing the contents of directories is done with `ls`:

`ls`	Compact listing of the current directory
`ls -l /usr/bin`	Detailed listing of `/usr/bin`
`ls ~`	Compact listing of your home directory
`ls -la ~`	Detailed listing of your home directory, including those the semi-hidden files beginning with '.'
`ls -lt`	Detailed listing of the current directory, in chronological order

Copying is done with `cp`:

`cp` *`filename dir`*	Copy a file to some other directory, resulting in *dir/filename*
`cp` *`filename newname`*	Make a copy, named `newname`, of the specified file
`cp` *`something* dir`*	Copy all files with names matching *`something*`* to the directory *`dir`*
`cp -r` *`dir1 dir2`*	Copy all files and subdirectories in *`dir1`* to a new directory *`dir2/dir1`*

`cp -r dir1/* dir2`	Copy all files and subdirectories in *dir1* into the existing directory *dir2*

Note the subtle but important difference between copying from *dir1* and from *dir1/**.

Moving and/or renaming is done with `mv`:

`mv filename dir`	Move a file to some directory *dir*
`mv something* dir`	Moves all files with names matching *something** to the directory *dir*
`mv -r dir1 dir2`	Move all files and subdirectories in *dir1* to a new directory *dir2/dir1*
`mv -r dir1/* dir2`	Move all files and subdirectories in *dir1* into the existing directory *dir2*
`mv filename newname`	Rename a file or directory
`mv filename dir/newname`	Move a file to some directory *dir* and rename it

When moving and copying with `mv` and `cp`, the target directory *must* already exist.

Use `mkdir` and `rmdir` to create and remove directories:

`mkdir Aug21`	Make a new directory within the current directory
`mkdir ~/data/Aug21`	Make a new directory (`Aug21` in this case) in `~/data`
`rmdir ~/data/Aug21`	Remove a directory (`Aug21` in this case) from `~/data`

But note that Linux prevents you (as a precaution against lapses of judgment) from deleting directories that still contain files. This can be a lifesaver: bit it can also be a chore to delete all files before being allowed to issue the `rmdir` command. However I expect that you will eventually discover the way around this default behaviour, which will enable you to delete a directory and all its contents (including subdirectories) with a single command.

Removing files is done with `rm`:

`rm filename`	Delete a particular file from the current directory
`rm dir/filename*`	Delete files matching the pattern *filename** from the directory *dir*

The commands described above are just the start. You will find that each command has many, many options Â— and the way to learn about the other options is to consult the 'man' pages:

`man -k PDF`	Lists all commands connected in any way to PDF
`man ls`	Tells you *everything* about `ls`
`man rm`	Tells you *everything* about `rm`
`etc`	etc

With hundreds of commands available `man` might seem of little use to neophytes. The `man -k keyword` option is one way to find out what is available; others are the thematic lists in Some common commands, and the excellent *Linux in a Nutshell*.

It is also useful to know about *redirection*. Output is commonly listed on the terminal, but can be redirected to a file with the '>' command. Thus

```
ls -1 ~/data > names.txt
```

stores the output as a file named `names.txt`, overwriting the pre-existing file of this name, if any. (A variation of this output redirection command is '>>', which *appends* the output to an existing file, rather than overwriting.) The opposite is '<', which gets input from a file rather than the keyboard, e.g.

```
mail -s 'Hi' < letter.txt
```

Programs can be run in the 'background' by appending an ampersand ('&') to the command line. Thus

```
gv doc.ps &
```

launches ghostview and puts it in the background. What does 'background' mean, and why would you want to do this? Without the ampersand the program runs in the foreground, which means that no new commands can be entered on the command line. Launching the program in the background allows new commands to be entered on the command line. *The program operates identically in both cases.*

There are thousands of commands documented in the man pages. If you want to see for yourself, type

```
find /usr/share -name 'man' -type d | xargs ls -lR
```

which also gives you a taste of the sophisticated way in which all those commands can be used. In this example, `find` is used to look for all directories named '`man`' in the directory `/usr/share`, including all its subdirectories. That produces a list of directories that contain `man` pages. The list is not printed: instead it is 'piped' to the `xargs` command, which constructs commands like `ls -lR dirname`, each of which generates a recursive directory listing of all files in the man directory, and its subdirectories. The upshot is that the compound command above locates and lists all man pages, however scattered they are in the `/usr/share` branch of the directory tree. You can augment the above command slightly:

```
find /usr/share -name 'man' -type d | xargs ls -lR | wc -l
```

to get an approximate count of the number of man pages. But even this is a rather elementary example of the power of Unix/Linux commands.

Some common commands

Here are some more useful commands. They are a tiny fraction of the command-line programs available, but give a taste of what can be accomplished. Most of the commands below are core Linux utilities. See the 'man' pages for more about their purpose, and more about their options Â— wherein lies their real power. Also listed below are several large applications, which tend to have built-in instructions.

If you are a newcomer to Linux you may not understand the point of the commands listed below, or doubt their value. All I can say is that I have used them all, and consider each one useful or even invaluable.

File management

`a2ps`	Convert a text file to PostScript, and (by default) print it
`cat`	Concatenate text files Â— or simply display text files
`cd`	Change to some other directory
`chgrp`	Change the group to which a file belongs
`chmod`	Change access modes, e.g. `chmod go-rwx` *`file1`* to make a file invisible to others
`chown`	Change the owner of a file
`cp`	Copy files
`hd`	Dump a file, showing both hex and ASCII versions
`head`	Display the first few lines of a file
`less`	Display a file, page by page
`ln`	Create an alias ('link') for a file, e.g. `ln -s` *`current_name new_name`*
`ls`	List files contained in a directory
`mkdir`	Create a directory
`mv`	Move or rename files or directories
`pwd`	Print the present working directory (where am I?)
`rm`	Remove files
`rmdir`	Remove a directory
`tail`	Display the last few lines of a file
`wc`	Word count Â— also the line and character count

Communication

`scp`	File transfer
`sftp`	File transfer
`ssh`	For terminal-like sessions on remote hosts

File comparisons

`cmp`	Compare two files (usually binary files), byte by byte
`diff`	Compare two files (usually ASCII files), line by line
`kompare`	Compare two files graphically

Graphics

`display`	For image viewing and manipulation Â— moderately elaborate
`gimp`	For image manipulation Â— very elaborate
`ksnapshot`	For screen capture
`xfig`	For creating line drawings
`xv`	For image viewing and manipulation Â— not too elaborate

Printing

`lpq`	Show contents of printer queue
`lpr`	Send file to printer
`lprm`	Remove print job from queue

Searching

`apropos`	Search man pages for given topic
`find`	Search directory tree for specific filenames
`grep`	Search text files for specific words
`locate`	Find files based on partial names
`strings`	Extract all text from a file

Bundling of files

`bzip2`	Compress a file, producing a `.bz2` file
`bunzip2`	Uncompress a `.bz2` file
`compress`	Compress a file, producing a `.Z` file
`gunzip`	Uncompress either a `.gz` or `.Z` file
`gzip`	Compress file, producing a `.gz` file
`tar`	Bundle many files into one
`uncompress`	Uncompress a `.Z` file

Text processing

`awk`	Non-interactive editor (line-orientated)
`cut`	Select particular columns from a text file
`dos2unix`	Deal with the DOS vs Unix newline difference
`emacs`	Elaborate text editor
	Friendly text editor

`nedit`	
`sed`	Non-interactive editor (stream-oriented)
`sort`	Sort or merge files
`tr`	Translate (redefine or delete) characters, e.g. `tr -d ',' < fil > fil2`
`unix2dos`	Deal with the DOS vs Unix newline difference
`vi`	Classic text editor

Word processing

`kword`	Similar to WordÂ™
`ooffice`	Similar to WordÂ™
`latex`	Utterly unlike WordÂ™
`dvips`	Turns LaTeX output into PostScript
`dvipdf`	Turns LaTeX output into PDF
`ps2pdf`	Turns PostScript into PDF
`gv`	For viewing PostScript and PDF documents
`xpdf`	For viewing PDF documents
`acroread`	For viewing PDF documents

Status

`df`	Show the capacity and usage of each partition
`du`	Show the size of all subdirectories
`env`	Show all environment variables
`kill`	Terminate a running program, e.g. `kill -9 10283`
`renice`	Alter the priority of a running program, e.g. `renice -n 10 12872`
`ps`	Show all running programs, e.g. `ps -ef`
`quota`	Show your disk usage and quota
`top`	Show active programs

Miscellaneous

`clear`	Clear the terminal window
`finger`	Find out about a user, e.g. `finger paul`
`ispell`	Spell checker
`man`	Display a man page
`passwd`	Alter your password

`source`	Run a C shell script, e.g. `source ~/.cshrc`
`w`	Who is logged on
`which`	Show the full path of a program
`xargs`	Organizes multiple command line arguments
`xrdb`	Manage your X11 resources, e.g. `xrdb .Xdefaults`

Note: Linux is astonishingly diverse in its manifestations, and is evolving rapidly. Consequently you may find that as many as 5 to 10% of the above commands are unavailable on your system. The common commands are quite stable, however.

Some examples

Some of the commands in Some common commands are so common and useful that they deserve a few more comments.

`gzip` and `gunzip`

These are used for compressing and uncompressing files. Typically you will have a file, for example `Study0223.tar`, that you want to compress before FTP'ing it to some other site. You just need to enter

```
gzip Study0223.tar
```

and the original file will be replaced by one named `Study0223.tar.gz`. Conversely, if you receive a file with the tell-tale extension `.gz` then entering

```
gunzip Study0223.tar.gz
```

will expand the file to one named `Study0223.tar`. The uncompression command `gunzip` is especially useful since it can deal with several compression formats: `gzip`, `zip`, `compress`, and `pack`. The detection of the input format is automatic.

`tar`

The name `tar` comes from Tape ARchive, but is more commonly used to bundle a group of files and write it to another file, rather than to a tape. A simple example would be:

```
tar cvf tarfile.tar file1.eeg file2.eeg ...
```

This takes any number of files *`file1.eeg file2.eeg ...`*, specified explicitly or using wildcard characters ('*', '?' etc), and bundles them into an output file `tarfile.tar`. The options are: `c` to create an archive, `v` to do so verbosely, and `f` to output to a file with the immediately following name. (The `v` is optional, but can be reassuring.)

It is probably more common (and more powerful) to specify a *directory* instead of a list of input files, as in:

```
tar cvf /media/usb/backup.chris.jan.tar /suphys/chris
```

which is what I might do to back up my home directory, including all subdirectories. A warning though: it is both illogical and *bad* to `tar` a directory and write the output file to that same directory!

If you receive a tar file, say `somefile.tar`, then I suggest you first look at its contents by

```
tar tf somefile.tar
```

This will produce a list of the enclosed files, including any subdirectories. When you untar the file, the listed files Â— including subdirectories Â— can be created in the current directory, and this might not be what you want. It is advisable to use the `t` option to anticipate exactly what will happen when you extract the contents of the tar file. Perhaps you should move the tar file into a suitably-named subdirectory before extracting its contents. What you *don't* want to do is extract lots of files, and then find that they are hopelessly mixed up with pre-existing filesÂ…

When you are satisfied that the tar file is in an appropriate location, you can extract everything with

```
tar xvf somefile.tar
```

where `x` means extract, and `v` and `f` have the usual meanings. Users will occasionally wish to extract a specific file or files: that is easily achieved by appending the names of the required files, as in

```
tar xvf somefile.tar go-nogo/10038259/slice5.dcm
```

The requested file '`slice5.dcm`' will then be created in a subdirectory '`go-nogo/10038259/`' under the current directory.

`find`

While the command `locate` may be adequate much of the time for tracking down files, the command `find` is far more powerful. The basic usage is `find` *`directory criterion`*. It searches in directory *`directory`*, and all subdirectories, for anything matching *`criterion`*. Thus you could enter

```
find ~/docs -name thesis.doc
```

to look for `thesis.doc` among your personal documents, or

```
find ~ -name 'note*'
```

to find files with names matching `note*`. (Note that the quotes are *essential* when using wildcard characters with `find`.)

You can search for particular directory names, as well as file names. If you wish to be specific, you could expand the criterion by appending `-type d` or `-type f`.

The criteria can go far beyond the simple cases above. Here are a few more examples:

`find ~ -atime -4`	All files under `~` accessed in last 4 days
`find ~ -mtime -4`	All files under `~` modified in last 4 days
`find / -user chris`	List all files owned by `chris`
`find ~ -name 'core.*' -exec rm {} \;`	Delete all core dumps

The `find` command is even more powerful in combination with others. A very common example is this: you want to identify a subset of files (`*.tex`) within some directory (`~/docs`) and all subdirectories; and to search within this subset of files for a particular string of characters ('needle'). The way to do this is:

```
find ~/docs -name '*.tex' | xargs grep -n 'needle'
```

The output shows, for each match, the file, line number and the line itself.

The following shows progressive elaboration of a `find`, which ultimately performs a selective archive of all recently modified files.

`find ./progs ./tex -ctime -75 -type f`	All files under `./progs` and `./tex` that have been created, modified, or had their status changed in the last 75 days

`find ./progs ./tex -ctime -75 -type f \`
`-o -path ./progs/corejsf -prune \`
`-o -path ./progs/jeda/build -prune`
	Ditto, but exclude two subdirectories

`find ./progs ./tex -ctime -75 -type f \`
`-o -path ./progs/corejsf -prune \`
`-o -path ./progs/jeda/build -prune \`
`| xargs tar cvf backup.tar`
	Ditto, but additionally bundle the files as `backup.tar`

`make`

It is very common to use the `make` utility when compiling programs, although `make` can be used for doing all sorts of operations. This utility looks for a file, conventionally named `Makefile`, and performs any one of several sets of commands.

An example `Makefile` is as follows:

```
PROG    =eegfit
CC      =gcc

# For profiling support add -pg, for which -g3 is a prerequisite
CFLAGS_G        = -Wall
CFLAGS_D        = -g3 -ggdb
CFLAGS_R        = -O
CFLAGS  =$(CFLAGS_G) $(CFLAGS_D)

# For profiling support add -pg
LFLAGS_G        = -lX11 -lm -lncurses -L/usr/X11R6/lib
LFLAGS_D        =
LFLAGS_R        =
LFLAGS  =$(LFLAGS_G) $(LFLAGS_D)

SOURCES = eegfit.c spectfit.c xlib.c gammq.c \
          gser.c gcf.c gammln.c ran1.c gasdev.c
OBJECTS = eegfit.o spectfit.o xlib.o gammq.o \
          gser.o gcf.o gammln.o ran1.o gasdev.o

%.o: %.c
        $(CC) $(CFLAGS) -c $<

eegfit: $(OBJECTS)
        $(CC) -o $@ $(OBJECTS) $(LFLAGS)

clean:
        rm -f *.o $(PROG)

all:
        $(CC) $(CFLAGS) -c $(SOURCES)
        $(MAKE)

tar:
```

```
            tar -cvf eegfit.tar $(SOURCES) *.h makefile

    # Dependencies
    eegfit.o:       parameter.h
    spectfit.o:     parameter.h xlib.h
    xlib.o:         xlib.h
```

This example makefile specifies all the files involved in a program, `eegfit`, and all the operations required to achieve certain ends: compilation, starting afresh, and creating a tar archive. Having created this makefile, it is then just a matter of typing one of the following:

`make eegfit`	Compile the application
`make`	Ditto
`make clean`	Delete the object files and the executable
`make tar`	Bundle all source files into `eegfit.tar`

Environment

The key to convenient program execution is understanding your `$PATH`. This is an 'environment' variable: it can be examined by typing `echo $PATH`, and it lists all directories that are searched automatically whenever you try to run a program. If a program with the name you specify is found in any of the directories listed in `$PATH`, then it is run Â— otherwise you get a message `Command not found`.

As an example of displaying the `$PATH`:

```
    echo $PATH
    /bin:/usr/bin:/usr/bin/X11:/usr/local/bin:/usr/X11R6/bin:.
```

The presence of '`/bin`' is the reason you can type `vi` instead of having to type the full, explicit path `/bin/vi`, whenever you want to edit a file. For the same reason the '.' at the end means that programs in your current directory can be run without any directory prefix.

You are free to alter the `$PATH` variable, and the best way to do so is in your `.cshrc` file. This is a configuration file for your command interpreter or 'shell', and can be viewed by `less ~/.cshrc`. You will generally see in it references to `setenv` and much else besides. If you want to prepend some directory to your `$PATH` (your personal program directory `~/bin`, say) then type either of

```
    setenv PATH ~/bin:$PATH
    set path=(~/bin $path)
```

(both do the same thing), and from then on you will be able to run your program `~/bin/model` (say) just by typing `model`, irrespective of your current directory.

Actually it is better to place the above line in your `.cshrc` file Â— otherwise the change is limited to the terminal window in which you typed it.

Typing `printenv` will give you a full list of environment variables: those you specified in your `.cshrc` file, plus those that the operating system sets automatically. You will see `$PATH`, discussed above, the environment variable `$PRINTER`, which specified your preferred printer, `$DISPLAY`, which says where graphics are to appear, and much else.

Note the `alias` commands in `.cshrc`, which define personal keyboard shortcuts. For example if you routinely type

```
   ssh joe.physics.usyd.edu.au
```

to access the computer '`joe`' then you can add

```
alias ssh-joe 'ssh joe.physics.usyd.edu.au'
```

to your `.cshrc` and then access `joe` just by typing `ssh-joe`.

Shell programming

Once you are comfortable with Unix/Linux commands, you can considering even more interesting operations.

Consider the hypothetical problem of separating a collection of serially-numbered files with names like `mr2366.dcm`, `mr2367.dcm`, `mr2368.dcm`, Â…. You want to copy just the *odd*-numbered slices to a neighbouring directory named `0dd`, while also renaming the copied files to `mr0dd2367.dcm`, `mr0dd2369.dcm`, Â…. If there were just three or four files needing to be moved, then you could issue a series of explicit `mv` commands, one per file. But for any more you should consider using the programming power of the shell.

In the above hypothetical case, the solution would be to enter the following three lines at the shell prompt:

```
foreach f ( *[13579].dcm )
mv $f ../0dd/`echo $f | sed 's/mr/mr0dd/'`
end
```

The first line looks for all odd-numbered slices, and for each creates a `mv` command defined by the second line. As a result, a series of commands, like those that you could have typed manually, is executed automatically:

```
mv mr2367.dcm ../0dd/mr0dd2367.dcm
mv mr2369.dcm ../0dd/mr0dd2369.dcm
:
```

Shell programming is undoubtedly an advanced topic. However it is often the perfect solution to a problem. Long or frequently-used sets of commands can be bundled into a *shell script*, and the script can then be run as a single command. (You will need to `chmod` it as well, to make it executable.) In the above example, the 3-line script might be named `extract_odd`, and run simply by typing this name. It doesn't get much easier than that.

Scripting can be about moving files and manipulating filenames. It can also be about modifying the contents of files. `sort`, `cut` and `join` can be used to rearrange tabular material very simply. The utilities `sed` and `gawk` provide more powerful programmatic methods for modifying, collating, and extracting values in text files. For example, if a file contains columns of numbers, and you want to see just the first column and the ratio of the next two, then

```
gawk '{if(NF>=3) print $1,$2/$3}' dat.txt
```

will do the job.

Finally, the code below is a realistic example of a shell script, whose goal should be obvious. To maximize its utility the script (named `t2ps`) is placed in `~/bin`, which should be made part of your `$PATH`.

```
#!/bin/tcsh
if ($#argv == 0) then
```

```
  echo -n "File(s): "
  set filelist = ($<)
else
  set filelist = ($argv[*])
endif

# Strip extensions, if present
set filelist = ($filelist:gr)

# Process each file, checking first that each exists
foreach filename ( $filelist )
  if ( -e $filename.tex && -f $filename.tex ) then
    latex $filename.tex && dvips $filename -o && rm $filename.{aux,dvi,log}
  else
    echo "t2ps: input file $filename.tex not found"
  endif
end
```

Shell programming is a big topic. Reading `man tcsh` (or `man bash`) is a start, but a difficult one. I recommend books like *Linux in a Nutshell* and *Redhat Linux Toolbox*, which concisely cover all topics and provide brief examples. Also I like and recommend the many freely-available reference cards: see, for example, the collection at www.cheat-sheets.org.