Linux Boot Camp

UKZN SKA School 2013

1 What is Linux?

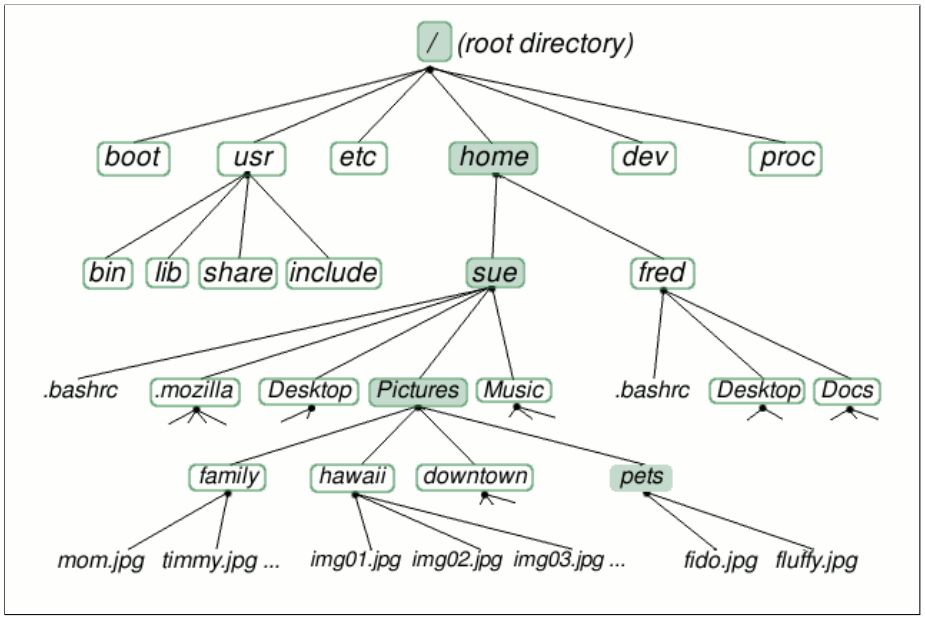
Linux is a free and open-source operating system. Almost all astrophysical computing relies on Linux, so it is essential to learn now to navigate comfortably in this system. There are many philosophical musings behind Linux that I won't get into here—as the title suggests, the main purpose of this document is to provide a pragmatic and condensed introduction to Linux essentials.

The first thing you need to do in order to get started on a Linux machine is to open a **terminal**, which is a window that has a prompt where you can type. This prompt is called the **command line**, and this is where all the action will happen. In contrast to e.g. Windows, Linux is largely about *typing commands* rather than pointing and clicking with the mouse.

2 File and directory structure

The figure below (image credit: Nadeem Oozeer) illustrates that files and directories in Linux are organised in an upside-down tree structure. All subdirectories stem from the **root directory = /**, which forms the base of the tree. The subdirectories can be loosely classified into two categories:

- **System directories = /bin, /etc, /proc, /usr, /var**, etc... These directories contain files that govern the operating system as a whole. You generally don't need to modify these directory contents unless you're e.g. installing packages or doing other system-wide tasks.
- **Home directories = /home/username**. This is where your personal files live, and most of your work will take place within your own home directory. In the example figure, there are two users, `sue` and `fred`, who have home directories in the system.



3 Navigating from the command line

All operations in Linux are performed from the command line. When you open a terminal, you are presented with a prompt where you type your commands. This section will give you a basic navigation tool kit, including specifying file and directory locations, and descriptions of the most essential Linux commands.

3.1 File and directory paths, full and relative

When you first open a Linux terminal, you are within your home directory, **/home/username**. In order to navigate the tree, you need to specify the location or "path" of files and directories with one of the following methods:

- **Full path, referenced from the root directory /.** The locations of all files and directories can be specified by "following the branch" from the root directory: starting with **/**, all subdirectories along the branch are appended, each separated by additional forward slashes. For example, in the above figure, the location of Sue's picture of Fido would be **/home/sue/Pictures/pets/fido.jpg**.
- **Relative path, referenced from your current directory**. If the location of the file or directory does not begin with a forward slash, then it is interpreted as being *relative* to the directory that you are working in. For example, in the above figure, if `sue` opened a terminal, she would start off in her home directory, `/home/sue`. From here, she could specify the location of Fido's picture as **Pictures/pets/fido.jpg**—note how this differs from the full path example.

3.2 Basic commands for dealing with directories and files

At any given time within a Linux terminal, you're working within a particular directory. You can find out where you are, navigate to other directories, and create brand new directories with the following commands:

- **pwd** = **p**resent **w**orking **d**irectory. Use this command to figure out which directory you're working in. This command doesn't take arguments, and you just type it by itself:
  `pwd`
- **cd** = **c**hange **d**irectory. Use this command to change to a different directory. This command requires an *argument*, which specifies the directory that you want to change to. As described in the previous section, you can specify the directory location with either a full or relative path. For example:
  `cd /home/sue/Pictures/pets`
  `cd Pictures/pets` (if working in Sue's home directory)
- **mkdir** = **m**ake **d**irectory. Use this command to create a new directory. This command requires an argument that specifies the name of the new directory. For example:
  `mkdir /home/sue/Pictures/durban`
  `mkdir Pictures/durban` (if working in Sue's home directory)

Now that you know how to navigate within the directory tree, here are some basic commands for dealing with files. Some of these commands also work on directories.

- **ls** = **l**i**s**t files and directories. If you type this command by itself (with no arguments), it lists all of the contents of your present working directory. You can optionally provide a directory location as an argument in order to determine its contents:
  `ls /home/sue/Pictures/pets`

- **rm** = **rem**ove files and directories. This command requires an argument that specifies the name of the file to be deleted. *Note that file deletion in Linux is IRREVERSIBLE. Once you delete a file, it's gone forever. There is no "recycle bin" for recovering accidentally deleted files.* Here's an example that illustrates deleting Sue's picture of Fido:
  `rm /home/sue/Pictures/pets/fido.jpg`

- **rmdir** = **rem**ove empty **dir**ectories. This command is similar to `rm`, but it works only on directories that contain no files.

- **cp** = **cp**y files and directories. This command requires two arguments: the first specifies the *source* that you wish to copy, and the second specifies the *destination* location or name. Note that you can use the second argument to create a copy that has a different name. For example, the following commands copy Sue's picture of Fido to (1) Sue's `family` subdirectory and (2) to a new file with a different name, `dog.jpg`:
  `cp /home/sue/Pictures/pets/fido.jpg /home/sue/Pictures/family`
  `cp /home/sue/Pictures/pets/fido.jpg /home/sue/Pictures/pets/dog.jpg`

- **mv** = **m**o**v**e files and directories to different locations or new names. Like `cp`, this command also requires two arguments that specify that source file and the destination location. The `mv` command can be used to *rename* files by moving an existing file to a new, different name. For example, the following commands take Sue's picture of Fido and (1) move it to Sue's `family` subdirectory or (2) rename it to `dog.jpg`:
  `mv /home/sue/Pictures/pets/fido.jpg /home/sue/Pictures/family`
  `mv /home/sue/Pictures/pets/fido.jpg /home/sue/Pictures/pets/dog.jpg`

- **cat** = con**cat**enate the contents of a file, i.e. dump the contents to your screen. This command takes a file name as an argument and is most useful for viewing the contents of text files. For example:
  `cat /home/sue/.bashrc`

- **more** or **less** = view file contents with scrolling. These commands are similar to `cat` in that they dump file contents to your screen, but they show only one screen's worth at a time. To scroll forward through the file contents, press `<space>`, and to scroll backward, press `b` (for **b**ack). You can exit at any time by pressing `q` (for **q**uit). For example:
  `more /home/sue/.bashrc`
  `less /home/sue/.bashrc`

3.3 Extended command options

In general, most Linux commands can be followed by *option flags* that extend their functionality. To obtain a comprehensive list of flags for any Linux command (in addition to a brief usage instructions), use the **man** = **man**ual command. For example:
`man ls`
`man rm`

The following list in this section will describe just a few of the most commonly used option flags for the commands featured in the previous section. The world of options extends far beyond this brief list, and it's well worth using `man` to explore all the possibilites. In all cases, the options are called by adding the flags immediately after the command. For example:
`ls -l /home/sue/Pictures/hawaii` (long file list)
`ls -t /home/sue/Pictures/hawaii` (list files sorted by modification time)
`ls -lt /home/sue/Pictures/hawaii` (you can use more than one flag at once)

- Flags for **ls**:
  -a = list **a**ll contents, including "hidden" files
  -d = list only **d**irectories, not files
  -l = list **l**ong content descriptions
  -t = list contents sorted by modification **t**ime
  -r = list contents sorted by **r**everse modification time

- Flags for **rm**:
  -i = remove **i**nteractively. You will be presented with a prompt for every file that you're trying to delete, and `rm` will ask if you're absolutely sure that you want to proceed.
  -r = remove **r**ecursively, i.e. include all subdirectories and their contents
  -f = **f**orcibly remove contents, even those that are write-protected

- Flags for **cp**:
  -r = copy **r**ecursively, i.e. include all subdirectories and their contents. Note that running `cp` on a directory will not work without the `-r` flag, and the command will report an error.

- Flags for **mkdir**:
  -p = create all **p**arent directories specified in the target. For example, typing
  `mkdir -p /home/sue/Pictures/durban/garden/monkey`
  will create the `durban`, `garden`, and `monkey` subdirectories all at the same time. Running `mkdir` in this case without the `-p` option will fail, and the command will report an error that the topmost subdirectory does not exist.

3.4 Other useful commands

The commands described in the previous sections constitute most of what you'll need to know in order to make your way around a Linux system. This section describes a few extra handy tools, and we'll leave it up to you to read about the details using `man`:

- **history**: Print out a record of all the commands you've typed. This is an extremely useful personalised "cheat sheet" as you're learning new commands.

- **top**: Show a continuously updated list of all processes running on the computer, sorted so that the processes consuming the most resources are displayed on top (exit by typing `Ctrl-c`)

- **which**: When a command is supplied as an argument, the full path to that command is returned. For example, typing `which ls` will return something like `/bin/ls`.

- **locate**: When a string is supplied as an argument, `locate` returns a list of all files/directories on the computer with a name that matches the specified string. For example, try something like `locate ipython`.

- **chmod**: Change the permissions of a file or directory. The simplest syntax is `chmod` followed by:
  1) a combination of letters specifying the user categories:
  `u` = user (i.e. you), `g` = group, `o` = other, `a` = all
  2) either `+` (plus sign) for grant permission, or `-` (minus sign) for deny permission
  3) a combination of letters specifying what permissions to grant/deny:
  `r` = read, `w` = write, `x` = execute permission.

  This is a bit dense, so I'll illustrate with a few common examples:
  `chmod a+r file.txt` — give everyone read access to `file.txt`
  `chmod og-w file.txt` — don't let anyone write to `file.txt` except yourself
  `chmod u+x file.bin` — give yourself execute permission for `file.bin`

- **echo**: This is essentially a print statement for Linux, and it's useful when writing scripts. For example, try `echo godzilla`.
- **ln**: Symbolically link a file to another location. This command is useful for creating shortcuts to files without having to make explicit copies.

3.5 Shortcuts and special characters

As you can see by now, working within Linux requires a fair amount of typing. In order to speed up the typing process, there are a few handy shortcuts. **The most important shortcut that you need to know about is TAB COMPLETION**—when you type the first few letters of a command or file, you can press `<tab>` to autocomplete the rest of the text. For example, rather than typing `chmod` in full, you can type `chm<tab>` instead. This is especially handy when dealing with e.g. long file names.

The following special characters are also extremely helpful in speeding up Linux navigation.

- *** (asterisk)**: This is the wildcard symbol that's used with file names, and it matches any set of characters. For example, let's say you have the files `cat.jpg`, `cheetah.jpg`, and `leopard.jpg`. If you typed `ls c*`, you would get a list of all files that start with "c." If you typed `ls *e*`, you would get a list of all files that contain the letter "e."
- **? (question mark)**: This is another wildcard symbol and works in the same way as the asterisk, except that only a single character is matched. For example, let's say you have the files `img3.jpg`, `img44.jpg`, and `img55.jpg`. If you typed `ls img?.jpg`, you would get `img3.jpg` in the output list because it's the only file with a single character in between "img" and ".jpg."
- **! (exclamation point)**: This symbol is used as a prefix to a partially typed command, and the last command in your history that matches the partial text will be executed. For example, let's say that the last `ls` command that you ran was
  `ls /this/is/a/really/long/file/path/that/is/annoying/to/type`.
  You could rerun that exact same command by simply typing
  `!ls`
  (Amazingly convenient, no?)
- **~ (tilde)**: This symbol represents user home directories. When used by itself, it refers to your own home directory; for example, typing `cd ~` is a shortcut for `cd /home/username`. When used as a prefix for a particular user, it refers to that user's home directory; for example, typing `cd ~sue` is a shortcut for `cd /home/sue`. Note that there is no space between the tilde and the username.
- **. (period)**: This symbol represents your present working directory. For example, if you want to copy a file to the directory that you're working in, you would type something like
  `cp ~sue/Pictures/pets/fido.jpg .`
- **.. (two periods)**: This symbol represents the *parent* of your present working directory, i.e. one level up in the tree.
- **& (ampersand)**: When this symbol is added to the end of a command and its arguments, that process runs in the "background." This is useful for commands that run for a long time and often comes in handy when you're running your own code. While the process is running in the background, you still have access to the command line and can run additional processes from there. Example usage is `emacs &`.

- **cd – (cd followed by hyphen)** : This is a special case. The command `cd -` takes you to the directory that you were previously working in. This is very useful for switching rapidly between two different working directories.

3.6 Environment variables

Within Linux, you can define "environment variables" that play special roles in your interaction with the operating system. The syntax[1] for setting and unsetting environment variables is:
`export MY_ENVIRONMENT_VARIABLE=value` (set value)
`unset MY_ENVIRONMENT_VARIABLE` (unset value)
Once a variable is set, it's accessed by prefixing the name with a dollar sign. For example, let's say we set an environment variable as a shortcut to Sue's directory of pet photos (as illustrated in the first figure):
`export SUE_PICS=/home/sue/Pictures/pets`
We can then use the variable as an argument to Linux commands:
`echo $SUE_PICS` (note the dollar sign)
`ls $SUE_PICS`
`cd $SUE_PICS`

The above example is just a simple illustration of how you can set variables to make life in Linux more convenient for yourself. However, the real power in environment variables is that there are certain, special variable names that are set aside for controlling system settings. By modifying these special environment variables, you can tweak the way the operating system works for you. Here's a list of a few common and important ones:

- **$HOME**: Your home directory.
- **$PATH**: The list of directory paths that are searched in order to find the commands that you type. BE CAREFUL WHEN MODIFYING THIS VARIABLE because if you delete the original settings, you may end up not even being able to run `ls`!
- **$PYTHONPATH**: The list of directories that are searched in order to find modules that are called by python.
- **$PYTHONSTARTUP**: The location of a user-specified python startup file.

4 Creating and editing text

There is a wide variety of editors that can be used to create and edit text in Linux. Some of the most common editors include `pico`, `nano`, `gedit`, `vi/vim`, and `emacs`. We'll focus on `emacs` in this section because it offers a reasonable compromise between graphical versus keyboard control. As you begin to write your own code, you'll also discover that `emacs` offers automatic syntax highlighting, which is invaluable for catching simple syntax errors as you're coding.

To start the editor, simply type one of the following:

- `emacs &` (use the ampersand to run the editor in the background)
- `emacs file.txt &` (the file you want to edit can be added as an argument)

[1]The syntax in this section is for the `bash` shell. The `tcsh` shell (and variants) are also often used in the Linux world, and the syntax there is `setenv` instead of `export`.

In both of these cases, `emacs` will open as a new window in your screen. The simplest way to proceed with file editing is to use the dropdown menus at the top of the window, which are similar to those used by e.g. Microsoft Word. However, the real power of `emacs` lies in the numerous *keyboard shortcuts* that allow you to edit text without ever having to touch the mouse. A few commonly used shortcuts include:

- `Ctrl-v`: Scroll down one screen.
- `Alt-v`: Scroll up one screen (`Alt` is actually "Meta," or `M` in `emacs`-speak, and it is occasionally mapped to the `Esc` key on some machines.
- `Ctrl-a`: Jump to the beginning of the line.
- `Ctrl-e`: Jump to the end of the line.
- `Ctrl-d`: Delete one character in front of the cursor (acts like the delete key).
- `Ctrl-k`: Delete all characters from the current cursor position to the end of the line. This shortcut, in combination with `Ctrl-a`, allows you to easily delete lots of text very quickly.
- `Ctrl-y`: Paste or "yank" text that has just been deleted by `Ctrl-k`.
- `Ctrl-n`: Move cursor to the next line (acts like up arrow).
- `Ctrl-p`: Move cursor to the previous line (acts like down arrow).
- `Ctrl-f`: Move cursor forward one character (acts like right arrow).
- `Ctrl-b`: Move cursor backward one character (acts like left arrow).
- `Ctrl-x Ctrl-s`: Save a file.
- `Ctrl-x Ctrl-f`: Open a file.
- `Ctrl-x b`: If you've opened multiple files, this command allows you to jump between them by moving to different "buffers."
- `Ctrl-x Ctrl-c`: Quit `emacs`.
- `Ctrl-s`: Search the file in the forward direction.
- `Ctrl-r`: Search the file in the backward direction.
- `Alt-q`: Justify the current block of text.
- `Ctrl-g`: If you accidentally screw up a keyboard shortcut, you can use `Ctrl-g` to escape out of the minibuffer.

There are many, many other keyboard shortcuts and commands that you can read about online. More advanced examples[2] include split screen editing, working with rectangular blocks of text (which is extremely useful for editing code), search and replace, etc. A true `emacs` ninja relies solely on keyboard shortcuts, which are much faster than pointing and clicking with the mouse. In fact, it's arguably more convenient to run the editor within a terminal, with the **graphical display turned off**:
`emacs -nw` ("nw" = no window)
Becoming comfortable with the keyboard shortcuts has a bit of a learning curve, but it's well worth investing a week or two of concerted practice in order to acquire this skill.

[2] You can even play tetris within `emacs`...

5 Connecting to other machines

You will often need to log on to other machines or copy files to/from those remote machines. The following commands will allow you to explore the broader Linux world.

- **ssh** = **s**ecure **sh**ell. If you have an account on a remote machine, use this command to log on. The syntax is `ssh username@remote.address`, and you'll be prompted for your password. There are many options that you can use with `ssh`, and one of the most common is the `-X` flag (note capitalisation), which turns on graphical forwarding.

- **scp** = **s**ecure **cp**y. This command allows you to copy files to or from a remote machine. The `scp` command takes two arguments, the **source** and the **destination**, and the syntax is as follows:
  ```
  scp file.txt username@remote.address:path/to/directory
  scp username@remote.address:path/to/file.txt /local/directory/path
  ```
  The first example copies a local file to the remote machine, and the second example copies a remote file to a local directory.

- **rsync** = enhanced remote file copying, similar to `scp` but with more options. The syntax is similar to `scp`, with source followed by destination, but here I'm adding a few handy option flags:
  ```
  rsync -auv file.txt username@remote.address:path/to/directory
  rsync -auv username@remote.address:path/to/file.txt /local/directory/path
  ```
  You can read about the flags in detail on the `man` page. The `-auv` combination tells `rsync` to preserve file details (such as modification time stamps), and if you're coping multiple files, only the ones that are new or have been modified are transferred.

6 Beyond the boot camp

Congratulations on arriving at the end of this condensed tutorial! The world of Linux is much bigger than what's described here, and there are many online resources that can help you learn more. The broader Linux community also runs many online forums where you can ask questions if you hit stumbling blocks.

As written in the intro, Linux is *free* software, which means you can download it from the web for installation on your computer! At the SKA school, you've been using `ubuntu`, which is a widespread and easy-to-use distribution. Other varieties of Linux are also available online, and it's up to you to decide which one works best for you. Good luck!