Introduction to the linux command line (v0.9.4)

Joel Fenwick

Introduction

This document is a walk-through of some basic concepts behind using a command line (shell) environment in Linux. The majority of these concepts and commands are also applicable to other POSIX and unix type environments eg BSD and MacOS. Some of the minor details may vary on other systems, so we will refer to "linux" throughout.

Many of the commands covered in this document are quite sophisticated. However, the purpose of this document is to illustrate some basic tasks and demonstrate how some aspects of the system (such as background tasks) work. To get a better sense of the flexibility and capability of particular commands, please see the individual documentation / manual pages.

This guide was written with the `bash` shell but most of the content should be generally applicable.

1 Connecting

For this section we assume that you will be interacting with a remote computer running linux (eg `moss`). If your local computer (the one you are sitting in front of) provides you with a linux command line, then skip to the next section.

You will need a program which allows secure text communication with the remote machine using the `ssh` protocol[1]. On Windows, your best bet is probably a program called `putty`. On MacOS, you can just use the `ssh` command on the terminal.

1.1 PuTTY

1. Start PuTTY
2. Enter the name of the machine you wish to connect to in the "Host Name (or IP address)" box. eg: `moss.labs.eait.uq.edu.au`
3. Make sure SSH is selected as the "Connection type".
4. If this is running under your login on the local machine:
    - Enter something short like: `moss` into the "Save Sessions" box

[1] There are other mechanisms which may be available such as VNC but we won't discuss them here.

- Click "Save"
- Next time, you can just double click the name in the list rather than typing

5. Click the Open button

1.2 ssh on MacOS

Start a terminal window and type: `ssh` *username_on_remote_machine@machinename*.
For example: `ssh s1234567@moss.labs.eait.uq.edu.au`
Once you have entered *your* credentials, you should see a prompt for the remote machine.

2 The shell

The shell is the main way you will interact with the system (manipulating files, running programs, ...). When the shell is ready to receive instructions from you, it will display a prompt and a cursor. Possible prompts:

- `s1234567@moss:~/tools$`
- `moss:tools s1234567$`
- `root@moss:~/tools#` (If you see this one and you are just learning, something has gone wrong)

Which one you see, depends on the system you have connected to (prompts can be customised). All of the prompts above contain the following information:

- Your username (`s1234567` in the above example). Note that, if you are connected to a remote computer, this may be different from your login name on your local computer.
- The name of the computer the shell is running on (eg `moss`).
- the name of the directory[2] which you are currently in (called the "current" or "working" directory). This may appear as the name of just the directory itself (`tools`) or a more complete description of where the directory is (~/tools).

Note that linux uses / (whereas Windows uses \) to separate components in the description of a file's location.

To get the shell to do something, type a command at the prompt and press enter. In this document, we will represent commands to enter like this:

```
$ echo "Hello"
```

The $ represents your shell's prompt (which as we've just said, will probably have more information in it). The $ at the beginning of the line is not something you need to type. If $ appears elsewhere though, then you should type those. For example:

[2] Your system may refer to directories as "folders".

```
$ echo $USER
```

would indicate that you should type `e c h o  $ U S E R` (followed by enter — without the extra spaces). When giving examples of command use, we may show the command and its output.

```
$ hostname
moss.labs.eait.uq.edu.au
```

Note that the output of the hostname, doesn't have a prompt in front of it. Any command typeset in a box like this

```
$ #Like this
```

is one we expect you to enter. If we are talking about other commands, we will use `#Like this`.

When you press enter, as well as being executed, your text is saved in a command history. You can use the arrow keys to browse this history to avoid retyping commands.

2.1 Saying goodbye

In this section: `exit, ^D`

We will use the notation ^D to indicate the control-D character/key sequence (that is, hold control and press D).

When you have finished with a shell, it is a good idea to leave it cleanly rather than just closing the window. Different shells may have a number of ways of doing this but the following will generally work:

```
$ exit
```

or pressing ^D

Start a new shell ready for the next section.

3 Looking around directories

In this section: `ls, cd, mkdir, pwd, *, ?, cat`

When you log in, your shell will usually be in your "home" directory (indicated by `~`). You can see a list of the files in the working directory with:

```
$ ls
```

It is possible that you will just get another prompt immediately. This means that the working directory is (apparently) empty. You can force the shell to show you "hidden" files, by adding an *option* to the command (specfically `-a` for "all").

```
$ ls -a
.  ..  .bashlogout .bashrc .profile
```

(You might see some different files).

Let's look somewhere more interesting.[3] The `cd` command will change the working directory.

```
$ cd /etc
```

Your prompt should change to indicate the change in working directory but you can always "print the working directory" with

```
$ pwd
/etc
```

Listing files in this directory will give a much larger list. You can get a more focused list using "wildcards" to indicate patterns of names you are interested in.

```
$ ls hos*
```

Will show all filenames in the working directory which begin with "hos". The * is not limited to the end of the filename (and a number of patterns can be combined on one line).

```
$ ls c*.conf g*.conf
```

would list all filenames in the working directory which begin with `c` or `g` and end in `.conf`. The * represents any number of missing characters (so `ls *` does the same thing as `ls`). To represent exactly one character use `?`.

```
$ ls ???.conf
```

will show all `.conf` files with 8 (3 + .conf) characters in their names.

For now, let's go back to your home directory. Typing cd on its own will take you home from wherever you are.

```
$ cd
```

You could also use:

```
$ cd ~
```

You can refer to files which are not in the working directory by adding *path* information infront of their names. A path is "how to get there from here". So we can see the files in `/etc` without `cd`-ing there. For example:

```
$ ls /etc
$ ls /etc/*.conf
```

Now let's take a look inside some files. The `cat` command (among other things) displays the contents of a file to the shell.

```
$ cat /etc/resolv.conf
```

Now try

[3] Idea of using /etc as a convenient source of files is from Michael Stonebank's University of Surrey tutorial.

```
$ cat /etc/services
```

Where there is too much information to view in one go, it is useful to display output a screen at a time.

```
$ cat /etc/services | less
```

You can then press space to view a page at a time or the arrow keys for finer control.

When you have seen as much as you need, quit from `less` by pressing 'q'. Now we will create a directory to do some work in.

```
$ mkdir TUTE
$ ls
```

You should see `TUTE` in the list of files in your home directory.
Since the directory is new, it will be empty so let's copy some files into it.

```
$ cp /etc/*.conf TUTE
```

You may see some messages about "Permission denied", don't worry about them for now. This has copied all the .conf files which you can access, from the /etc directory to the `TUTE` directory. As you can see, you can copy large numbers of files in one command, but the last name needs to be the destination for the files.

```
$ cd TUTE
$ pwd
```

If you want to put files into the working directory (instead of somewhere else), you can use `.` to refer to the working directory.

```
$ cp /etc/services .
$ ls
```

As you can see from the directory listing, when `cp` copies a file from another directory, the resulting file has the same name as the source.

Now try:

```
$ cp /etc etccopy
```

By default, `cp` won't copy directories. We could do `cp /etc/* etccopy` but that won't work because `etccopy` would need to be a directory and it doesn't exist yet. It would also be a bad idea to `cp /etc/* .` because that would put all the files in the current directory (which is not what we want).

We can copy a whole directory trees using the recursive option `-r`.

```
$ cp -r /etc etccopy
```

When -r is used, the destination directory will be created if requried.

3.1 Getting rid of things

In this section: `rmdir, rm`

You may wish to remove files or directories. Directories are removed with the `rmdir` command. For example:

```
$ mkdir dir77
$ ls
$ rmdir dir77
```

Now try:

```
$ mkdir dir78
$ cp services dir78/srv
$ rmdir dir78
rmdir: failed to remove `dir78': Directory not empty
```

The second line above, shows how to specify a different name for the copy. This will place a copy of the services file from the working directory into the dir78 subdirectory and call it `srv`.

The remove directory command fails because, as a safety measure, rmdir will not remove a directory that is not empty. We will need to remove the file first.

```
$ rm dir78/srv
$ ls dir78
$ rmdir dir78
```

It is possible to remove a subdirectory and all its subdirectories and files using the `rm -r` command but it needs to be used carefully. Note: the linux shell does not support "undelete" or anything like a recycle bin, so be careful what you delete.

3.2 Paths

Now a bit more about paths. A path will consist of a sequence of directory names separated by / characters. For example:
`./work/projects`
means — start at the current directory (**.**), then go to the `work` subdirectory, then the `projects` directory under that. If a file or subdirectory in the working directory is being referred to, the **./** is assumed and can be omitted.
eg: `work/projects`. This is an example of what is called a "relative" path because where you end up depends on the current working directory.

To refer to a directory "above" the current one (sometimes called the parent directory), use **..** For example, since we are currently in `~/TUTE`, `..` would refer to your home directory, `../../` would refer to the directory above your home directory. `../TUTE/../TUTE/../TUTE` is another way to refer to the directory we are in (although a needlessly convoluted one). There are also, "absolute" paths which always end up at the same place regardless of where they are used. In linux, every file accessible on the computer will appear in a subdirectory (or subdirectory of a subdirectory, ...) of a single "root directory". This includes

files which are actually stored on different disks or even on different servers[4]. (So linux has no concept of "drive letters".)

The root directory is labelled as `/`. So, any path starting with / (with no leading `.`), begins at the fixed root and works its way down. Eg: `/etc` is an absolute path. Paths like `~/TUTE/dir78` are also absolute, because they start at a known fixed point (ie `~`, your home directory). Also, try this to see what `~` actually expands to:

```
$ echo ~
```

3.3 Moving and renaming things

In this section: `mv`

The command to rename a file is actually the same as the one to move (as opposed to copy) a file.

```
$ ls
$ mv services newservices
$ ls
```

To put the file somewhere else, include a directory in the second name. So to move the file up to your home directory instead of renaming it, you would type `mv services ..`

4 Make it stop

In this section: `^C, ^Z, fg`

For this demonstration, we will use the `cat` command without any parameters. There is no particular significance to this, it's just a command that lets you type things without an error message.

```
$ cat
```

You won't get a prompt back while cat is running (in the foreground) so type some text to convince yourself that this is the case. Now stop the currently running command by pressing ^C.

```
$ cat
```

This time use ^Z. You will see a message something like:
`[1]+  Stopped                 cat`
and you will have a prompt. In this case, the command has only been "suspended" and is still taking up system resources. To get `cat` moving again,

```
$ fg
```

Type some text to convince yourself that you are talking to `cat` and not the shell and then stop it with ^C.

[4]MacOS users can see this at work in the /Volumes folder Any extra disks attached to the machine appear as subdirectories under /Volumes.

4.1 It's locked up (aka did you press ^S by accident?)

If your shell appears to have stopped responding, it may be worth pressing ^Q (the antidote to ^S).

5 Getting help from the man

In this section: `man, info`

Most commands will have online manual pages which describe how to use them. For example:

```
$ man cat
```

This will display the manual page using `less`[5] so you can use the same keys to navigate. Press 'q' to exit back to the shell when you are done. If you aren't sure of the command name, you can search all the manual pages for text. To look for all commands which include "directory" use:

```
$ man -k directory
```

Sometimes there can be more than one manual page with the same name. For example, `printf` is both a shell command and a function in the C programming language.

```
$ man -k printf
```

Amongst the output you will see:

printf (1)

printf (3)

The numbers in parentheses are the "section" of the manual that page belongs to. If you do not get the page you wanted by default, you can specify the section of the manual to check, like this:

```
$ man 3 printf
```

Some commands may also have information in the `info` command. For example `info ls` but info is harder to navigate.

6 Creating and editing files

There are a number of options for editing files on remote machines, including.

1. Copy the file to your local machine, edit it, then copy it back.
2. Edit the file on your local machine which shares a file system with the remote machine.
3. Run a local editor which accesses the remote files directly (eg over sshfs).
4. Run an editor on the remote computer.

[5] Some systems may use the older, `more` instead.

The first option can be slow and fiddly, especially if you need to test between each of a lot of small changes. The second option only works in certain system configurations anyway. The third option requires extra software. The first three options also *can* have another problem. When editing text files, Windows has a different opinion about what a line of text looks like compared with other systems. This can mean that a file created / saved with a windows editor can create problems on other systems[6].

The alternative is to use a textmode editor on the remote machine. In this example, we will use `nano` because it is very simple. For more "serious" work (especially if coding is involved), `vim`[7] would be better, but a tutorial on that is beyond the scope of this document.

```
$ nano dostuff
```

You will be presented with the nano editor screen. For now, enter the following:

```
echo "Hello World"
echo "Goodbye World"
```

then ^X and `Y` then enter to save and exit.

You may have recognised that the file contains shell commands, so lets try to execute the file.

```
$ ./dostuff
```

The shell will object to this and say something about permissions, so let's look at those.

7 Permissions

In this section: `chmod`

Every file in linux has some information associated with it.

```
$ ls -l
```

You will see a line which looks something like this for each file:

```
-rw-------+ 1 s1234567 uusers 739 Mar 31 09:10 filename
```

The first part describes the permissions for the file. Then comes the link count (but we won't talk about this in this document). The two things that look like names give the information about the user who owns the file and the group the file is associated with. Next is the size of the file in bytes followed by the time and date when it was last modified. After all that, is the name of the file.

So let's pull apart the permissions information. Because linux represents many things as files, there are a lot of possibilities for things which can appear in the permissions. Here we will just do the main ones[8]. The first symbol will be `-` for normal files and `d` for directories. Next will be a group of three permissions `rwx` corresponding to being allowed to:

r read the file.

[6] Of course this depends on the editor and what the files are used for. Some tools handle it just fine.

[7] Or its older, grumpier cousin `vi`.

[8] For detailed information, see `man chmod`

w write to the file — this includes both modifying the file and deleting it.

x execute the file — run it as a command.

If the position for a given permission shows as **-** instead of a letter, that means the corresponding permission is not set. So a file which can be read but not written to would have permissions like:
`-r--...`
That explains the first four positions. The next three are the same permissions but applying to the group of the file rather than the owner. So a file which the owner can read and write but the other members of the group can only read would look like.
`-rw-r--...`
The last group of three is the permissions for everyone else. A file which the owner and group members can read and write while everyone else can read would look like:
`-rw-rw-r--...`

For directories, the permissions have slightly different meanings but in summary: If you don't have **r**, then you can't see what is in the directory. If you don't have **w**, then you can't modify the directory. This includes adding new files or removing/renaming existing ones. If you don't have **x**, then you can't access anything in the directory (even if you know its name).

You may want to change permissions in order to prevent accidental modification of files, or to prevent people from reading your files. You can change permissions with the `chmod` command. Let's add the execute permission to `dostuff` we made before.

```
$ chmod +x dostuff
```

Now we can run `dostuff`.

```
$ ./dostuff
```

If we wanted to make `dostuff` read only, we could use `chmod -w dostuff` (ie use + to add a permission and - to remove it). To see other permission operations, consult `man chmod`. To change the owner or group of a file, see `man chown` and `man chgrp` (these operations may be restricted on your system).

8 (Compressed) bundles of files

In this section: `tar`

Linux can deal with with `.zip` files (see `man unzip`) but a more common method for bundling files is `tar`. A `.tar` (originally "tape archive") file is a collection of files bundled together. We will create an archive of `dir78` and its contents.

```
$ tar -cvf stuff.tar dir78
```

The arguments to `tar` we used here were:

- `-cvf` (we could have written this as `-c -v -f`).

  - c : "create" — we are making a new archive.
  - v : "verbose" — not actually required, but causes `tar` to print the name of each file as it puts it in the archive so we can see what it is up to.
  - f : "file" — the next thing will be the name of the archive we want to act on (in this case to create it).

- `stuff.tar` is the name of the file we are creating
- filenames to act on. In this case, the names of the files or directories we want to put in the archive.

```
$ ls -l stuff.tar
```

So we now have a single file, but it is not compressed. The version of `tar` on linux can also deal with tar files which are compressed using a number of different compression tools. To use *gzip* compression with tar, add a `z` to the tar options.

```
$ tar -czvf stuff.tar.gz etccopy
```

```
$ ls -l stuff*
```

(Compression may not help much for small files).

To see how much space the directory takes, we can use the `du` (disk usage) command.

```
$ du -hs etccopy
```

- h : "human readable" — we'd like sizes reported in units we understand.
- s : "summary" — just report cumulative size for directories rather than reporting on each file individually

We can use `-h` on ls as well for comparison purposes:

```
$ ls -lh stuff.*
```

To see the filenames inside a tar file, use `t` (for table of contents) instead of `c`:

```
$ tar -tzf stuff.tar.gz
```

To see the other compression methods your version of `tar` supports, look at `man tar`.

Finally, we need to be able to extract files from archives. The `tar` action for this is `x` (instead of `c` or `t`) but we want to be a little bit careful here. The tar file contains a directory called `dir78` which is also the name of a subdirectory. In such cases, tar will put the contents of the archive into that subdirectory (which is sometimes what you want but not now).

So we'll make subdirectory to put it in temporarily.

```
$ mkdir temp
$ cd temp
$ ls
$ tar -xzf ../stuff.tar.gz
$ ls
$ cd ..
```

9 What is happening on the system?

In this section: `sleep, ps, kill, top`

For some of these examples, we will need an additional command:

```
$ sleep 10
```

`sleep` just waits for the specified number of seconds and then stops.

All the commands we have run so far have been "in the foreground". That is, you can only have one command running at a time, because the shell waits for the first one to finish. However, it is possible to run multiple commands simultaneously by running them "in the background". You do this by putting a & after the command.

Try the following (remember you can use the arrow keys to review your command history):

```
$ sleep 20 &
[1] 52490
$ sleep 20 &
[2] 52491
$ sleep 20 &
[3] 52492
$ sleep 20 &
[4] 52493
$ ls
```

All of the sleep commands were running while ls was executing. To see that they weren't just paused the whole time, wait 20 seconds or so and then press enter a few times. You will see something like:

```
[1]   Done                    sleep 20
[2]   Done                    sleep 20
[3]-  Done                    sleep 20
[4]+  Done                    sleep 20
```

This is the shell telling you that background jobs have finished.

If you run a command in the background that needs input, it will automatically pause the command and let you know with a message like:
`[1]+ Stopped`

By the way, the numbers in brackets in the following can be used with the `fg` command to bring a background or "stopped" job to the foreground again. eg: `fg 3`

While this demonstration may seem a bit pointless, the point is an important one. In many cases, the remote machine will be a shared computer and what you do can negatively impact other users. In particular, running lots of programs simultaneously (or a small number with high memory or CPU time requirements). For this reason, it is important to be aware of what you are running on the computer.

For this next part, please open a second connection to the remote computer [we'll refer to this as *shell2* and the original as *shell1*].
In shell1:

```
$ cat
```

Now in shell2:

```
$ ps -fu$USER
```

The `ps` command reports on processes (running programs) currently executing on the system. By default, `ps` only shows processes running under the current shell. The options shown here will show everything you have running (regardless of which shell was involved). You will see processes called `bash` running. "Bash" is the name of the shell program.

In the output of shell2, look for the line with `cat` in the `COMMAND` column. Look up the number in the texttttPID (process ID) column.

```
$ kill put_the_number_here
```

In shell1, you should see the `cat` command terminate and the prompt return.

Another tool you can use to see what is happening is:

```
$ top
```

This displays the processes running (by default sorted by cpu usage) and updated every few seconds. Press 'q' to leave.

A note about interpreting top on heavily used systems:

- If you see something of yours running and consuming lots of resources (and you weren't expecting it to be), then do something about it (eg kill it).
- Top only samples every few seconds so just because something is at the top of the list does not necessarily mean it has consistently been using those resources.
- Be careful how you read CPU time percentages. Some systems report each CPU core as 100%, so a machine with 48 cores could have a program using 500% CPU and be perfectly fine. Other systems use 100% to represent the total CPU capacity of the computer (so using 1 CPU core could show up as 2.1%).
- If the computer appears to be running "slowly", do not immediately assume that the process leading the `top` list is responsible.
- If the computer is running consistently slowly: **contact a systems administrator**.

Exit from shell2 now, we will continue to use shell1.

9.1 Why won't it die?

Sometimes when you try to `kill` a process, it remains in the process list. There are a few possibilities here:

1. The system is a bit busy and will get to it soon. (unlikely to delay by more than a few seconds).
2. The thing you are trying to kill is stopped and will respond to the kill instruction when it starts up again.
3. The thing you are trying to kill is blocking kill instructions.
4. The thing you are trying to kill is a zombie[9]. Zombies can be identified by having a status of Z or have `<defunct>` next to their name.

Items 1 and 4 are normally resolved by a short wait. If a lot of "long lived" zombies appear with your name on them though, then you should investigate. To deal with #2, you could resume the process or you could make the `kill` command more assertive.
eg: `kill -9` *`process_id_here`*
This is also the solution for #3, since `-9` cannot be blocked. This raises the question, why not use `-9` all the time? A program may want to clean up before it exits. Using `-9` denies it that opportunity.

For the curious, even `kill -9` will not get rid of zombies. This is because they are already dead[10]. If you have processes in the process list that you really can't get rid of, contact a system administrator.

10 Redirected IO and pipes

So far everything we have done involves input from the keyboard (called "standard input" or stdin) and output to the screen ("standard out"/stdout and "standard error"/stderr). The shell allows output to be drawn from and/or sent to files. For example:

```
$ ls /etc > listoffiles
```

The > followed by a filename, means that anything the `ls` command sends to standard out (would have gone to the screen) is instead sent to the file you nominate. Note, this is a feature of the shell, not the particular command.

Now have a look in the file:

```
$ cat listoffiles
```

This can be useful if you want to record the output of a command for later processing / review. If you want to add output of a program to an existing file (eg adding to a log file) rather than replacing it, you can use `>>`.

```
$ ls >> lof2
$ ls >> lof2
$ cat lof2
```

[9] Yes, this is really what they are called.
[10] Yes, seriously.

Now try this

```
$ gcc > err
gcc: no input files
$ ls -l err
```

Notice that even though we "redirected" output, it still appeared on the screen, and ...

```
$ ls -l err
```

... the `err` file has nothing in it.

Programs in linux actually have two standard ways to output text, "standard out" and "standard error". Normally, both of these point at your screen but they can be redirected independently. In this case, `gcc` sent error messages to "standard error" rather than to "standard out" (stderr and stdout from here on).

This can be useful when you are running a program which produces a lot of output. You can ask that stdout be sent to a file and then you can see any error messages clearly.

To redirect stderr, use `2>` instead of `>`.

```
$ gcc 2> err
$ cat err
```

We can also draw input from a file instead of from the keyboard.

```
$ sort -r < listoffiles
```

(`sort -r` — sorts its input in reverse order)

Often, you will wish to combine these two steps and send the output of one program as input to another program. This is done by putting a pipe symbol `|` between the two programs. You have already seen this with `cat /etc/services | less` . But as another example: get the file names in `/etc` in reverse order, one page at a time.

```
$ ls /etc | sort -r | less
```

Or find out how many things there are in `/etc`:

```
$ ls | wc -l
```

(`wc` is a program which counts the number of lines, words and characters in its input.)

11 looking for things

In this section: `grep, find`

11.1 files

You search the file system using the `find` command:

```
$ find . -name services
```

The arguments above are:

- `.` — where to start searching. In this case the current directory.
- `-name` — we are going to specify something about the name of the files we are looking for.
- `services` — the exact name we are looking for.

We could be more vague about the names.

```
$ find . -name '*.conf'
```

You could search your home directory(and everything under it) by using `find ~ ...`. One *could* search the whole file system with `find / ...` but keep in mind that for systems that have networked file systems, this can be a very slow operation.

Find can also do much more complicated searches, including combinations or name, modification date, size, ... See the man page for more details.

11.2 content

To search for text within files, use `grep`. Grep returns all lines within its input: eg:

```
$ grep play /etc/services
```

will output all the lines of `/etc/services` which contain 'play'. If not given at least one filename, `grep` will read its input from standard in, so the following are equivalent to the above:

```
$ cat /etc/services | grep play
$ grep play < /etc/services
```

Grep can take multiple files as arguments:

```
$ grep name *.conf
```

You can also search whole directories by passing `grep` the name of the directory and using `-r`.

```
$ grep -r name /etc
```

If you are slicing and dicing text files, you may also want to look at the `cut` command.

12 The environment

Previously, we marked `dostuff` as executable but

```
$ dostuff
```

produces an error message.

In order to run programs, you must be explicit about their location (hence why `./dostuff` worked) or they must be in directories that the system expects commands to be in. These directories are listed in an *environment variable* called `PATH`.

To see its contents:

```
$ echo $PATH
```

You can add directories to the path like this:

```
$ export PATH=$PATH:~/TUTE
```

(this means PATH is whatever was in PATH before followed by ~/TUTE)—.

Now the following will work

```
$ dostuff
$ cd ..
$ dostuff
$ cd TUTE
```

Note that if a directory is "on the path", commands in it can be run from anywhere.

Two points about changing the environment. First, this change only has any affect within the running shell. When you log in again, the new shell will be using the default environment. Normally this is a good thing, but if you want a more permanent change to your environment, then you need to look at modifying your rc files (see documentation). *If you are using an HPC system, you may wish to look into modules instead of doing it manually.*

There are a number of other environment variables which control other aspects of the system. Two others you might hear about are:

- LD_LIBRARY_PATH — controls where the system looks for dynamic libraries.
- PYTHONPATH — controls places where python looks for modules.

12.1 Probably mistakes to do with PATH (and other environment variables)

The following are, in general, *not* good ideas:

- `export PATH=~/TUTE`, rather than adding something new to the path, lines like this (which don't read the previous value), replace the old value with what you write. This could mean not being able to run commands like `ls` since they won't be on the path any more.
- `export PATH=PATH:~/TUTE` — The flaw here is more subtle, but the $ is missing from the second PATH. Putting $ in front of an environment variable name means read the value out of the variable. Leaving it off means you want your PATH variable to literally contain:
  `PATH:~/TUTE`
  Which is not what you want.

- `export PATH=~/TUTE:$PATH` — This would work, but is only a good idea if you want commands in the directories which you add to be used instead of existing system commands of the same name.
- `export PATH=$PATH:.` — This will run commands from whatever directory you happen to be in. This could be a security problem if you happen to be in someone else's directory at the time or just be confusing and inconsistant. If you actually need to run something in the current (non-path) directory, then be explicit (`./dostuff`).

13 Cleaning up

Now we want to remove the `TUTE` directory and everything under it. As previously noted, linux has no undelete command so be careful when you use this:

```
$ cd ..
$ rm -rf TUTE
```

The options in use here are:

- r — "recursive" delete directories and their contents.
- f — "force" don't ask any questions

Always re-read your rm commands before executing them. An unintended space between `.` and / or between * and something else is a great way to lose lots of files.

14 Remote access to files

How you download/upload files from/to the remote computer depends on the arrangements your system administrators have made but the most common method will be via `scp` (secure copy).

On MacOS, or a local linux system the command will look like:

local → remote: `scp filenames moss.labs.eait.uq.edu.au:`

remote → local: `scp moss.labs.eait.uq.edu.au:filename .`

It looks very much like `cp` except you put the name of the remote machine followed by : in front of anything which is talking about the remote computer.

On Windows, there are graphical tools such as winscp to accomplish the same goal.

15 Tips

15.1 File names

Because shells use spaces to seprate arguments, it is better to have files without spaces in their names. You can work around this using quotes or \but it is often more trouble than it is worth. The same applies to other shell special characters like $, <, quotation marks and so on.

15.2 Querying permissions on directories

To see the permissions on a file, we use `ls -l` *`filename`*. If we try this on a directory though, it may not do what you expect:

```
$ ls -l .
```

When you use ls on a directory, it will tell you about the contents rather than the directory itself. Most of the time this is what you want, but not in this case. If you want to ask about directories rather than their contents, you need to add a `-d` to the command.

```
$ ls -ld .
```