Getting things done with BASH and UNIX

Will Pearse

March 28, 2015

1 Preamble

I am neither a computer scientist nor a system adminstrator. I'm simply someone who finds it fun to do things with large chunks of data, and isn't afraid of error messages. None of what is in this guide is definitive, and none of it is guaranteed to be correct. However, what follows is the result of (literally) years of experience, frustration, and joy. It is my hope that by reading this, and thinking about its contents, I can pass on the first, minimise the second, and maximise the third. This guide is intended for those who are interested in computers, but would prefer to get their other tasks (in my case, biology) done over endlessly re-doing things to make them more computationally 'neat'.

The most important thing when working with a computer is to *relax*. Computers do not make mistakes, we make mistakes, and every single thing that goes wrong during your time working with a remote system will be either your, or the system administrator's, fault. Computers don't have personalities, they don't hold grudges, and, most importantly, *they don't care if you make mistakes*. If you don't care, and the computer doesn't care, then you will find yourself transported to a zen-like state of bliss where mistakes and errors simply flow over you. When you realise that you make mistakes, and that there's nothing wrong with that, you'll find yourself much happier.

This document has two very important sections: an introduction to BASH (and UNIX), and a guide on how to log into a remote system (SSH). The next two sections are useful but not as fundamental; dip into these at your leisure. I do not repeat any information in this document; the final section contains additional tips about programs you have already started using.

2 Getting things done (BASH)

A UNIX-like computer is a set of programs that all balance, at times precariously, on top of the *kernel*. The kernel is the boss, and when we want to run programs, modify its settings, insert a USB disk, etc., we need a way to communicate with it. A *shell* allows us to do that, and BASH is the most commonly-used shell. There are others, but my advice is to stick with BASH since it's available everywhere. BASH is on (almost) all Linux and

MacOS computers (open 'Terminal' and you will see it); MacOS is based on a modified form of UNIX, and Linux is essentially a *very* modified form of UNIX itself. There is a long and tangled history of UNIX-like software; you may hear terms like POSIX-compliant being used, and it is almost always acceptable to think of those things as UNIX-like and have no problems.

2.1 Moving around

You are, at all times, in a directory of some sort, in the same way as you're always in a directory in Windows Explorer or Mac's Finder. To see your *present working directory*, type `pwd` and press enter. You have just run the `pwd` command, and the directory is spat out at you as an *absolute path*: directories are separated in the path by `/`, and the first `/` represents the *root* of your file system. It is the presence of this root in the path that makes it absolute.

Unless you specify an absolute path, BASH assumes you're referring to something relative to your current position. Let's make a new directory, and then move into it, using relative paths. *Make* a *directory* (`mkdir my.new.directory`), then *change directory* into it (`cd my.new.directory`). Verify your new present working directory. You can *list* the things in your directory using `ls`; running that command now will do nothing, as we haven't done anything yet!

Once you're in a directory, it's easy to move back by remembering that, in BASH, `.` refers to your current directory, and `..` refers to one directory back from the present one. `.` will come in handy later, but for now just move back one directory (`cd ..`) and list the contents of this directory. If you ever got lost, using `cd` with no arguments (*i.e.*, don't give it a directory to move into; `cd`) will take you back to what is called your *home* directory. Move back home now. `~` is a useful shortcut for your home directory; `cd ~/my.new.directory` will probably take you to your new directory.

2.2 File manipulation

Creating files is easy. It's simple to create an empty file (`touch name.of.file`), but often you will want to do more than that. `nano` is a lightweight text ed-

itor; run `nano test.txt` to open it in a new file. Write some nonsense, then exit by pressing *control-x* (the commands are at the bottom of the file). Follow the prompts; yes you do want to save, you do want to save in the file you opened (hit enter), and you're back in BASH. You can now print the contents of the file (`cat test.txt`) by con*cat*enating files and writing their output (more in a moment). You can even quickly scroll through the contents of a file (`less test.txt`) and use `q` to *quit*, but this is only useful if we want to quickly examine a large file.

Copy your file (`cp test.txt new.copy.txt`); verify using the above information that the file is the same. We can also *move* a file to a new location (`mv new.copy.txt ~/my.new.directory/`), and even give it a new name (`mv new.copy.txt new.name.txt`); note that moving a file so it has a new name is the same as renaming it! Concatenate both of those files together now (`cat one.file another.file`). You can also *remove* (delete) a file (`new.name.txt`); there is no undo in BASH, and there are no confirmation prompts before deleting. To remove a directory, you must first make it empty and then use `rmdir`. There is much more to learn on this topic, and I have put some new information in the appendix. File manipulation commands are very powerful: do not run before you can walk, and don't start trying to move hundreds of files around at once, or delete files according to what they contain (all things you can do easily!) until you are comfortable with the above. *There is no undo!*

2.3 Running a program and permissions

One of the reasons UNIX-like computers run the Internet is they have a very strong *permissions* system. Permissions what user can read, write, and run (start a program) a file.

Create a new file called `silly.sh` (`sh` stands for *shell* script), and fill it with the following text:

```
#!/usr/bin/env bash
echo 'hello, world'
```

The first line (called the *shebang*, as in "the whole shebang") tells BASH that this is a BASH script; please just treat it as a magic incantation whose

meaning doesn't matter. The next line is BASH code; see what happens when you type the words into your console now and its meanning will become apparent.

Right now, BASH won't let us run our program, but try anyway by typing `./silly.sh` (translated as: please run `./`—this directory, `silly.sh`—this file). List the contents of your directory with more detail (`ls -l`) and you'll see a column with all the files, the dates when they were last editted, who *owns* them, and at the far left a series of confusing `r`s (read), `w`s (write), and `x`s (execute/run) permissions. They're repeated because different kinds of users have different permissions: in general, you can do anything to a file you create, but not to someone else's, and the administrator can do anything to anyone.

Let's *change* the file *mode*—give ourselves permission to *execute* the file—by typing `chmod +x silly.sh`. That's it; when you `ls` the directory, you'll probably see the colour of the file has changed (green for go!), and now you can run our silly program.

This is the most difficult thing you'll have to do, and it probably seems silly now, but it is important. Other users can't modify your files; if you were to try and modify system settings, or other users' files, you would get an error. This also helps keep the system safe; before modifying anything sensitive, adminstrators have to authenticate (type type `sudo`, for *super user do*) to do something dangerous. This way all your files and settings are safe, so long as the administrator knows what they're doing.

3 Logging in (SSH)

SSH (*secure shell*) is a safe way to log into a remote computer—it's a way of opening a shell (see above) on that computer. You can think of an SSH connection as a quantum *tunnel*; once you open that tunnel, it's like you're sat at the computer you're tunnelled into. If you want to use a file on your own computer, you must first copy it there, because you're no longer on your own computer, you're on the remote computer.

From a UNIX-like computer (MacOS, Linux, etc.), openning a connection is as simple as typing `ssh username@remote.computer`. On Windows, you will

have to download a program like *PuTTy*; the warnings about the program being illegal reflect US foreign policy, and if this concerns you I encourage you to Wikipedia the program. So, to tunnel into my computer, I would type `ssh will@Lance`, or if I only knew the IP address of the remote computer (not its name) I would use that (`ssh will@123.345.678.12`). To copy a file, I would use *secure copy* which is based around SSH; something like `scp file.to.copy user@computer:/destination/path/`. Note that I have to say where I want to send the file on the remote computer! Alternatively, I could connect to the remote computer over FTP (either through the command line or using a program like CyberDuck). When typing your password into SSH, you often won't se any asterisks or anything else appearing on screen; this is normal. To log out, simply `logout`.

Using passwords is very risky, because they're quite easy to crack. Many administrators would prefer you use SSH keys; simply-put, you have a *private key* that can be used to generate a *public key*. There's some magic, complicated maths going on, but with a public key anyone can check that you own the private key, yet it's very hard for someone to figure out what the private key looks like using the public key. So keys are a good way to check someone is who they say they are. Don't think about how this works, just do the following:

- Get an SSH key.
- *Never share your private key with anyone ever.* If you do, tell everyone who uses that key to no longer trust it.
- Find your remote computer's administrator and ask them to use your public key for login.

Doing this means you never need to type your password when you log in through SSH, and makes everything much safer. People are always trying to hack big, fancy computers for their own nefarious purposes, and this will help keep everyone safe.

4 Running programs in parallel

You can run a program in the background by putting an ampersand after the line (*e.g.*, `echo 'hello, world' &`). This means you can carry on doing other pieces of work, or you can *run multiple programs at the same time*. Make sure you don't run more programs than you computer has processors!

You can also set a program to not stop running, even when you log out, by using `nohup` (*e.g.*, `nohup echo 'hello, world' &`). Your program's output will be put into a file called `nohup.out`. Be a *nice person* and use `top` or `htop` to check the computer's load before doing this too much, though.

If you're using *R*, make sure you use `mclapply` from *parallel* to run things. Not only does it make your code easier to read, but it also means you're using all the cores on your computer. That will make things faster; otherwise, what was the point in your learning BASH to begin with?

5 Useful BASH commands and tricks

Command	Description
`top`	Lists all the processes (programs) running on the computer, and how much of the processor is being used. `htop` is a much more friendly graphical version of `top`; if it's installed on your system use it. You can also stop programs here, a bit like control-alt-delete on a windows computer.
`make`	Used to compile a lot of programs; typically just typing `make` in the directory that contains the source code of a C/C++ or Fortan program will be sufficient. The name comes from the fact it looks for a *makefile* that describes how to compile the program you want to use. Try `make -f name.of.makefile` if that doesn't work, there's often more than one Makefile and you need to pick which version of the program you want to compile.
`wget`	*Get* something from the *Web*; use it to download programs etc.
`tar`	Used to create 'tar-balls' (zip files); `tar -xf file.tar.gz` will unzip something, `tar -cf new.tar folder another.file` will create a new .tar from the folder(s) and file(s) you specify. `unzip` does the same thing but for zip file.
`rm`	You can delete a whole folder and everything in it by running `rm` *recursively* and *forcing* it to delete things, *e.g.*, `rm -rf folder`. Many, many people have deleted more than they bargained using this tool.
`cp`	You can *recursively* copy things, meaning you'll be able to copy a directory, *e.g.*, `cp -r folder`.
`Rscript`	Runs an R script; useful in conjunction with `nohup` and friends (see above).
`apt-get`	Used on many Linux computers (notably Debain and Ubuntu) to install programs; you will almost certainly need to have `sudo` access to do this.
`sudo`	*Super user do*; allows administrators to authenticate before doing anything potentially dangerous to the system. If it's your own computer, you will be able to use this command. For the love of God be careful with it!
`tab`	Hit the tab-key when part-way through a command or file-name and BASH will try to complete it for you. If there is more than one potential match, keep hitting it and it will show all the possible completions.
`*`	You can list, copy, and delete based on wild-card matching. For example, `ls *.txt` will show you all the text files in a directory, and `rm first.try*` will remove files starting with the phrase 'first try'. Many, many people have done more than they bargained using this tool.
`|`	Pipes are one of the most powerful features of BASH. You can use them to chain commands together; for instance `ls -l | wc -l` will print out all the files in your directory (one per line; `ls -l`), and `|` takes that information and passes it to `wc -l` that counts the number of lines in its input. Voilá; you now know how many files are in the directory. Doing operations using `|` can result in orders of magnitude increase in execution speed; writing to disk is slow.